INTERNATIONAL STANDARD

ISO/IEC 23092-6

First edition
2023-11

Information technology — Genomic information representation —

Part 6: Coding of genomic annotations

Reference number
ISO/IEC 23092-6:2023(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 document 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 document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document 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. In the IEC, see www.iec.ch/understanding-standards.

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

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

Any feedback or questions on this document 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 and www.iec.ch/national-committees.

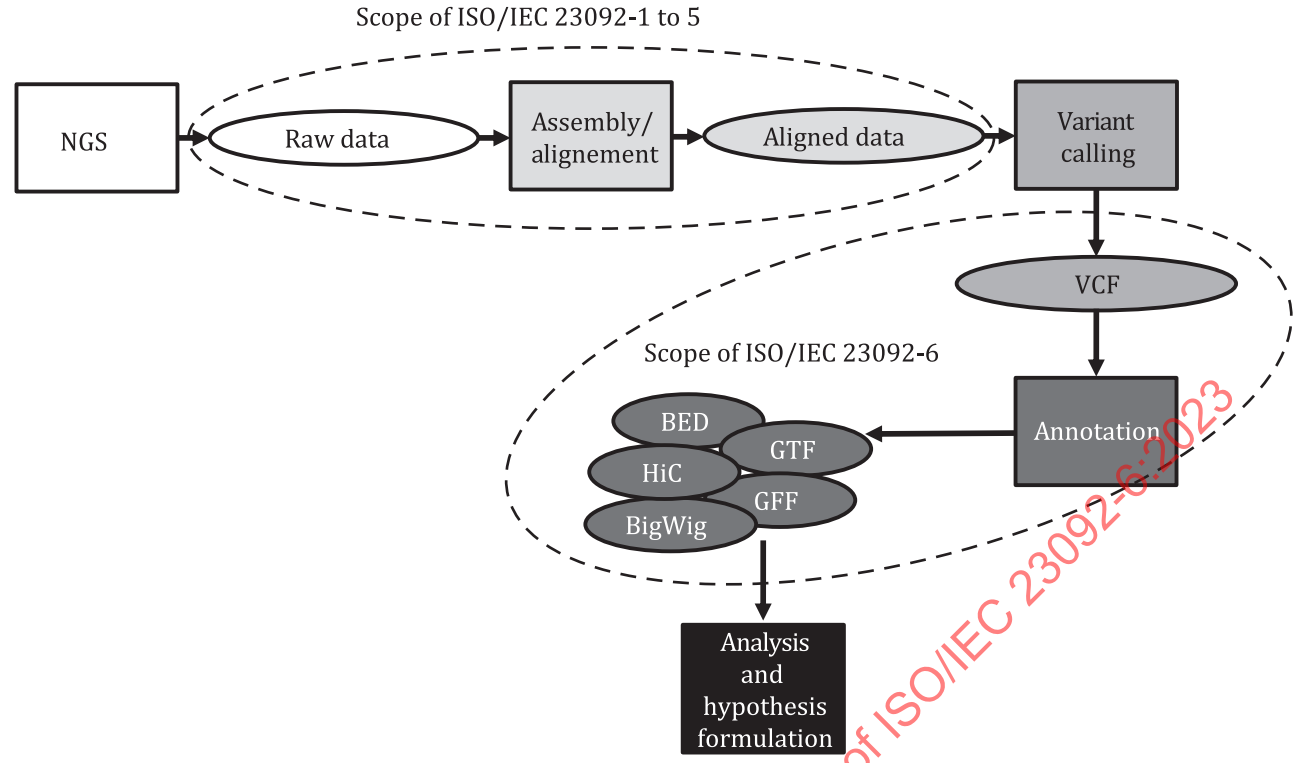
Introduction

While ISO/IEC 23092-1 to ISO/IEC 23092-5 (MPEG-G) deal with the representation of genomic information derived from the primary analysis of high-throughput sequencing (HTS) data – sequencing reads and qualities, and their alignment to a reference genome – which is only the first step in a long series. In particular, the results of primary analysis are usually processed further in order to obtain higher-level information. Such a process of aggregating information deduced from single reads and their alignments to the genome into more complex results is generally known as secondary analysis. In most HTS-based biological studies, the output of secondary analysis is usually represented as different types of annotations associated to one or more genomic intervals on the reference sequences.

Biological studies typically produce genomic annotation data such as mapping statistics, quantitative browser tracks, variants, genome functional annotations, gene expression data and Hi-C contact matrices. These diverse types of downstream genomic data are currently represented in different formats such as VCF, BED, WIG, etc., with loosely defined semantics, leading to issues with interoperability, the need for frequent conversions between formats, difficulty in the visualization of multi-modal data and complicated information exchange. Figure 1 depicts a typical pipeline for the primary and secondary analyses of HTS data, the file formats involved and the scopes of different parts of the ISO/IEC 23092 series.

Furthermore, the lack of a single format has stifled the work on compression algorithms and has led to the widespread use of general compression algorithms with suboptimum performance. These algorithms do not exploit the fact the annotation data typically comprises of multiple fields (attributes) with different statistical characteristics and instead compress them together. Therefore, while these algorithms support efficient random access with respect to genomic position, they do not allow extraction of specific fields without decompressing all the whole file.

In response to the aforementioned challenges, this document details a unified data format for the efficient representation and compression of diverse genomic annotation data for file storage or data transport. The benefits are manifold: reducing the cost of data storage, improving the speed of random data access and processing, providing support for data security and privacy in selective genomic regions, and creating linkages across different types of genomic annotation and sequencing data. The ultimate goal is to enable the secured and seamless sharing, processing and analysis of multi-modal genomic data in order to reduce the burden of data manipulation and management, so scientists can focus on biological interpretation and discovery.



Key

- sequencing generates raw reads
- read alignment
- variant calling
- variants annotations
- analysis

Figure 1 — Typical pipeline for the primary and secondary analyses of HTS data

Information technology — Genomic information representation —

Part 6: Coding of genomic annotations

1 Scope

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

— variants with genotyping information

— functional annotations

— tracks

— expression matrices

— contact matrices (from Hi-C experiments or similar).

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 specification. 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 11544, *Information technology — Coded representation of picture and audio information — Progressive bi-level image compression*

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

ISO/IEC 23092-2, *Information technology — Genomic Information Representation — Part 2: Coding of Genomic Information*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

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

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

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

3.1
access unit
logical data structure containing a coded representation of genomic information to facilitate bit stream access and manipulation

3.2
access unit start position
position of the leftmost mapped base among the first alignments of all genomic records contained in the access unit, irrespective of the strand

3.3
access unit end position
position of the rightmost mapped base among the first alignments of all genomic records contained in the access unit, irrespective of the strand

3.4
access unit range
genomic range comprised between the access unit start position and the rightmost genomic record position among all genomic records contained in the access unit

3.5
access unit covered region
genomic range comprised between the access unit start position and the access unit end position inclusive

3.6
alignment
information describing the similarity between a sequence (typically a sequencing read) 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.

3.7
allele
each of one or more alternative sequences for a genomic segment

Note 1 to entry: There can be more than one either because the genome contains more than one, almost identical, copies of the same genomic material (2 in the case of humans for all chromosomes from 1 to 22), and/or because one is considering more than one individual in the population.

3.8
annotation record
record
data structure representing a tuple of annotation information (e.g. the properties associated to a variant, a genomic feature or a generic range; it is used also to identify a ”row” when data have matrix format)

3.9
base
base pair
synonymous of nucleotide

3.10
base position
number of bases between a base and the leftmost mapped base belonging to the same genomic segment.

3.11
CIGAR string
CIGAR
textual way of representing an alignment

Note 1 to entry: Several definitions have been used by different programs, the ones 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.

3.12
cluster
aggregation of genomic records

3.13
cluster signature
signature
sequence of nucleotides that is common to most or all genomic records belonging to a cluster

3.14
contig
set of overlapping DNA segments, sequenced and assembled, that together represent a consensus region of DNA

Note 1 to entry: the term “contig” derives from “contiguous”.

3.15
dataset
compression unit containing one or more of: reference sequences; sequencing reads; and alignment information

Note 1 to entry: Datasets are specified in ISO/IEC 23092-1.

3.16
deletion
contiguous removal of one or more bases from a genomic sequence

3.17
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.

3.18
edit operation
modification of a sequence of nucleotides by means of a substitution, deletion, insertion or clip

3.19
FASTA
GIR that includes a name and a nucleotide sequence for each sequencing read

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

3.20
FASTQ
GIR that includes FASTA and quality values

3.21
first end
end 1
read 1
first segment of a paired-end template

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.

3.22
genomic descriptor
descriptor
element of the syntax used to represent a feature of a genomic sequencing read or associated information such as alignment information or quality values

3.23
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.

3.24
genomic position
position
integer number representing the zero-based position of a nucleotide within a reference sequence

3.25
genomic range
range
interval of positions on a reference sequence specified by a start position s and an end position e such that $s \leq e$

Note 1 to entry: The start and the end positions of a genomic range are always included in the range.

3.26
genomic record
record
data structure representing a tuple optionally associated with alignment information, read identifier and quality values

3.27
genomic record index
position of a genomic record in the sequence of genomic records encoded in an access unit

3.28
genomic record position
0-based position of the leftmost mapped base on the reference genome of the first alignment contained in a genomic record

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.

3.29
genomic reference
reference
collection of reference sequences

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

3.30
genomic segment
segment
contiguous sequence of nucleotides

Note 1 to entry: Typically output of the sequencing process, and sequenced from one strand of a template.

3.31
genomic variant
variant
one of the possible sequences for a genomic segment whenever more than one allele for that segment is present

Note 1 to entry: The variant can span one nucleotide (and is then usally called single nucleotide polymorphism) or more (structural variants can involve changes in thousands of contiguous bases or more). A variant can consist of an indel.

3.32
genotype
sequence of a genomic segment for a specified copy of the genome or individual whenever more than one allele for that segment is present

3.33
genotype matrix
metrix specifying which genotype is present in each copy of the genome or individual

3.34
hard clip
one or more bases originally present at either side of a read, and removed from it following alignment

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

3.35
indel
contiguous stretch of nucleotides 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”.

3.36
insertion
contiguous addition of one or more bases into a genomic sequence

3.37
leftmost read end
leftmost read
sequencing read 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

3.38
mapped base
base of the aligned read that either matches the corresponding base on the reference sequence or can be turned into the corresponding base on the reference sequence via a substitution

3.39
nucleotide
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’.

3.40
output annotation record
annotation record produced as output of the decoding process of an annotation table or a portion of it

3.41
paired-end reads
paired-end template
tuple made of two segments

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

3.42
pileup
textual representation of sequencing reads aligned to a reference sequence

3.43
ploidy
number of equivalent alleles present at each position of the genome

3.44
phased genotyping
information about consecutive genotypes along the genome which keeps information about the different copies of the genome (or different individuals) separate, whenever multiple alleles are present

3.45
quality value
quality score
number assigned to each nucleotide 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.

3.46
read group
set of reads having some property in common

3.47
read identifier
read header
read name
text string associated with each sequencing read stored in GIRs such as FASTA, FASTQ and SAM

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).

3.48
reference genome
representative example of the sequences for a species' genetic material

Note 1 to entry: Representative of the sequences of the DNA molecules present in a typical cell of that species.

3.49
reference sequence
nucleic acid sequence with biological relevance

Note 1 to entry: Each reference sequence is indexed by a one-dimensional integer coordinate system whereby each integer within range identifies a single nucleotide. Coordinate values can only be equal to or larger than zero. The coordinate system in the context of this standard is zero-based (i.e. the first nucleotide has coordinate 0 and it is said to be at position 0) and linearly increasing within the string from left to right.

3.50
rightmost read end
rightmost read
sequencing read 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

3.51
SAM
GIR that is human readable and includes FASTQ plus alignment 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.

3.52
second end
read 2
second segment of a paired-end template

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.

3.53
sequencing read
read
readout, by a specific technology more or less prone to errors, of a continuous part of a segment of nucleotides extracted from an organic sample

3.54
single-end read
tuple made of one segment

3.55
SNP
single-nucleotide polymorphism
genomic variant defined as one changed base at a specific position in the genome

3.56
soft clip
soft clipped bases
readone or more bases at either side of the read that have been ignored during the alignment process

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

3.57
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: The read comes from RNA-sequencing and contains at least one junction between two consecutive exons.

3.58
split alignment
aligned paired-end read whose ends are encoded in two different genomic records

3.59
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).

3.60
tile
non-syntactic concept describing the AU layout in terms of the original *(x,y)* coordinates when an annotation file is seen as a bi-dimensional matrix

3.61
tuple
collection of one or more segments

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

3.62
variable
parameter either inferred from syntax fields or locally defined in a process description

3.63
VCF
GIR that is human readable and stores data such as SNPs, insertions, deletions and structural variants of DNA, including annotations.

4 Abbreviated terms

AU	access unit
AAU	annotation 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.

Functions can return a tuple of n elements as output in the form of {v1, v2, ..., vn} = f(.) by including a statement “return {v1, v2, ..., vn}” at the end.

5.2 Logical operators

The following logical operators are defined:

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.

5.3 Arithmetic operators

The following arithmetic operators are defined:

+	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.
÷	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\frac{x}{y}$	Used to denote division in mathematical equations where no truncation or rounding is intended.
$\sum_{i=x}^{y} f(i)$	The 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.

5.4 Relational operators

The following relational operators are defined as follows:

>	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

The following bit-wise operators are defined as follows:

& Bit-wise "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.

| Bit-wise "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.

^ Bit-wise "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 Arithmetic 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 Arithmetic 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.

! Bit-wise not operator returning 1 if applied to 0 and 0 if applied to 1.

5.6 Assignment operators

The following arithmetic operators are defined as follows:

= 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

The following notation is used to specify a range of values:

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] a 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

The following mathematical functions are defined:

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

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

Log(x) the base-e logarithm of x

Log2(x) the base-2 logarithm of x

PopCount(x) the number of bits set to 1 in the binary representation of unsigned x

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

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

DivRem(x, y) the couple (x/y, x%y), i.e. the result and the reminder of the integer division of x by y

Binom(x, y) the binomial coefficient of x and y, i.e. x! / (y! * (x - y)!)

5.9 Array and strings operation functions

ndims(x) returns the number of dimensions of variable x, 0 if x is a scalar value. The function ignores trailing singleton dimensions.

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

array_dims[] = Size(ndimensional_array[]...[], dim) returns the dimensional size(s) of variable ndimensional_array. If dim is not specified, it returns a vector with the i^{th} element corresponding to the size of the i^{th} dimension. If dim is specified, it returns only the size of the dimension corresponding to dim, with 1 being the first dimension.

strcat(x, y) returns the concatenation of the input arrays or strings.

strlen(s) returns the number of characters in the string s.

concat(x, y, n) concatenate a 2-dimensional array. When n = 0 the concatenation is done by row (i.e., vertical-wise), for n=1 is done by columns (i.e., horizontal-wise).

horzcat(x, y) is the same as concat(x, y, n) when n = 1.

vertcat(x, y) is the same as concat(x, y, n) when n = 0.

5.10 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"

5.11 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. 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.

5.12 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.

5.13 Processes

5.13.1 General

Processes are used to describe the decoding of syntax elements (see Table 2). 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.

5.13.2 Process output operators

The dot "." operator enables identifying named members (syntax elements or sub-processes) within the output of a process. Dot operator can be applied recursively to descend into the hierarchy of sub-processes. For instance, given the sample decoding process `process_xyz` specified in the following table according to the syntax specified in subclause 6.1,

Table 2 — Sample decoding process

Syntax	Remarks
process_xyz(counter) {	A decoding process with one parameter
num	A syntax element
for(i = 0; i < num * counter; i++) {	
value[i]	A syntax element
}	
a_sub_process	A sub-decoding process within `process_xyz`
}	

another decoding process can access the members of the output of one specified run of `process_xyz` process, as specified in the following Table 3.

Table 3 — Accessing members of the output of a decoding process

Syntax	Remarks
another_process(xyz) {	*xyz is the output of one specific run of process_xyz process.*
a = xyz.counter	Access the parameter '*counter*' with which *process_xyz* was run to produce *xyz*
b = xyz.num	Access the syntax element '*num*' of the *process_xyz* run that produced *xyz*
c = xyz.value[b]	Access the syntax element '*value[b]*' of the *process_xyz* run that produced *xyz*

Table 3 *(continued)*

Syntax	Remarks
`d = xyz.a_sub_process.some_member` `}`	Access the syntax element `some_member' of sub-decoding process `a_sub_process' of the process_xyz run that produced xyz

5.14 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 4 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 4 — 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** conditioning statement	u(n)

/*A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. */
{
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. */	

Table 4 *(continued)*

Syntax	Type
for(initial statement; condition; subsequent statement) primary statement	

5.15 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.

5.16 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.

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

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.

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.

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

5.17 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.

6 Data Structures

6.1 General

Subclause 6.2 specifies the structure of a data unit, which is used as a container for one of the following data structures: raw reference (Type 0), parameter set (Type 1), access unit (Type 2), annotation table indexes (Type 3), annotation parameter set (Type 4) and annotation access unit (Type 5). Data units are produced as output of the decapsulation process specified in ISO/IEC 23092-1.

Subclause 6.3 specifies the structure of an annotation parameter set, which consists of an annotation parameter set identifier and encoding parameters for each descriptor or attribute.

Subclause 6.4 specifies the structure of an annotation access unit, which consists of an annotation access unit header followed by one or more blocks (as specified in subclause 6.4.4).

The raw reference, parameter set and access unit structures are specified in ISO/IEC 23092-2 and are out of scope of this document.

6.2 Data unit

This subclause specifies the data structure used to represent a data unit (see Table 5).

Table 5 — Data unit syntax

Syntax	Type
data_unit() {	
data_unit_type	u(8)
if (data_unit_type == 0) {	
data_unit_size	u(64)
raw_reference()	As specified in ISO/IEC 23092-2
}	
else if (data_unit_type == 1) {	
reserved	u(10)
data_unit_size	u(22)
parameter_set()	As specified in ISO/IEC 23092-2
}	
else if (data_unit_type == 2){	
reserved	u(3)
data_unit_size	u(29)
access_unit()	As specified in ISO/IEC 23092-2
}	
else if (data_unit_type == 3){	
reserved	u(10)
data_unit_size	u(22)
annotation_parameter_set()	As specified in subclause 6.3
}	
else if (data_unit_type == 4){	
reserved	u(3)
data_unit_size	u(29)
annotation_access_unit()	As specified in subclause 6.4
}	

Table 5 *(continued)*

Syntax	Type

else /*(data_unit_type > 4)*/{
/*skip data unit*/
}
}	

reserved are bits used to preserve byte alignment and shall be set to 0.

data_unit_type specifies the type of data unit. Table 6 lists the values of data_unit_type and the associated data unit types.

Table 6 — Values of data_unit_type and associated data unit types

data_unit_type	Data unit type	Reference
0	raw reference	ISO/IEC 23092-2
1	parameter set	ISO/IEC 23092-2
2	access unit	ISO/IEC 23092-2
3	annotation parameter set	subclause 6.3
4	annotation access unit	subclause 6.4

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 the raw reference data structure as specified in ISO/IEC 23092-2.

parameter_set() is the parameter_set data structure as specified in ISO/IEC 23092-2.

access_unit() is the access unit data structure as specified in ISO/IEC 23092-2.

annotation_parameter_set() is the annotation parameter set data structure as specified in subclause 6.3.

annotation_access_unit() is the annotation access unit data structure as specified in subclause 6.4.

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

6.3 Annotation parameter set

6.3.1 General

This subclause specifies the syntax and semantics of an annotation parameter set, which contains the encoding parameters used during the decoding process of an annotation access unit (see Table 7).

Table 7 — Annotation parameter set syntax

Syntax	Type	Remarks
annotation_parameter_set() {		
parameter_set_ID	u(8)	
AT_ID	u(8)	
AT_alphabet_ID	u(8)	
reserved	u(2)	
AT_coord_size	u(2)	
AT_pos_40_bits_flag	u(1)	
n_aux_attribute_groups	u(3)	

Table 7 *(continued)*

Syntax	Type	Remarks
`for (i = 0; i <= n_aux_attribute_groups; i++) {`		
`tile_config[i]`		As specified in subclause 6.3.2
`}`		
`annotation_encoding_parameters()`		As specified in subclause 6.3.3
`}`		

parameter_set_ID is the unique identifier of the parameter set containing the annotation encoding parameters.

AT_ID is the unique identifier of the annotation table to which this annotation parameter set is associated.

AT_alphabet_ID identifies the alphabet of symbols used for data encoded in access units referring to these encoding parameters. The symbols associated to each value of AT_alphabet_ID are specified in ISO/IEC 23092-2.

reserved are bits used to preserve byte alignment and shall be set to 0.

AT_coord_size is a value that indicates the number of bits required to represent a row/column coordinate of the annotation table (see **Table 8**).

tile_config is a data structure specified in subclause 6.3.2 carrying the tile configuration.

annotation_encoding_parameters is a data structure specified in subclause 6.3.3 carrying the annotation encoding parameters.

Table 8 — AT_coord_size values that indicate the number of bits for representing the coordinate of a row/column in an annotation table

AT_coord_size	Size in bits
0	8
1	16
2	32
3	64

AT_pos_40_bits_flag is set to 1 when the variant positions are expressed as 40 bits integers. Otherwise, all variant 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.

n_aux_attribute_groups is the number of auxiliary attribute groups in the associated annotation table.

6.3.2 Tile configuration

6.3.2.1 General

The tile configuration data structure (see Table 9) contains information on how the data of individual descriptors and attributes are split into tiles. It is only needed when `attribute_contiguity == 1`.

Table 9 — Tile configuration syntax

Syntax	Type	Remarks
tile_configuration {		
AG_class	u(3)	
attribute_contiguity	u(1)	
two_dimensional	u(1)	
if (two_dimensional) {		
reserved	u(6)	
column_major_tile_order	u(1)	
symmetry_mode	u(3)	
symmetry_minor_diagonal	u(1)	
} else {		
reserved	u(3)	
}		
attribute_dependent_tiles	u(1)	
default_tile_structure()		As specified in subclause 6.3.2.2
if (attribute_dependent_tiles) {		
n_add_tile_structures	u(16)	
for (i=0; i<n_add_tile_structures; i++) {		
n_attributes[i]	u(16)	
for (j=0; j<n_attributes[i]; j++) {		
attribute_ID[i][j]	u(16)	
}		
n_descriptors[i]	u(7)	
for (j=0; j<n_descriptors[i]; j++) {		
descriptor_ID[i][j]	u(7)	
}		
additional_tile_structure[i]		As specified in subclause 6.3.2.2
}		
}		
}		

AG_class is the class of the attribute group to which this tile configuration applies.

attribute_contiguity is a flag, if set to 1, indicates that blocks are grouped into annotation access units by attribute. Otherwise, blocks are grouped into annotation access units by tile.

two_dimensional is a flag, and if set to 1, indicates that the associated descriptors/attributes are 2-dimensional. Otherwise, they are one-dimensional.

reserved are bits used to preserve byte alignment and shall be set to 0.

If two_dimensional == 1,

column_major_tile_order is a flag that is only relevant when two_dimensional == 1 and variable_size_tiles == 0 in the tile structure (subclause 6.3.2.2) associated with the descriptor/attribute. If set to 1, it indicates the tiles are in column-major order when stored as data blocks in an annotation access unit (`attribute_contiguity == 1`). Otherwise, they are in row-major order.

`symmetry_mode` specifies the symmetry mode of the attribute group data and is only effective when `two_dimensional` == 1. The possible values are specified in ISO/IEC 23092-1.

symmetry_minor_diagonal is only effective when two_dimensional == 1 and symmetry_mode == 1. It is a flag, if set to 1, indicates that the symmetry of squared matrices is along the minor diagonal of all the attributes of the main attribute group data. Otherwise, symmetry is along the principal diagonal.

attribute_dependent_tiles is a flag, and if set to 1, indicates there exist certain attributes and descriptors with tile structures different from the default. Otherwise, all attributes and descriptors in the same attribute group share the same default tile structure.

default_tile_structure is the data structure defined in subclause 6.3.2.2 carrying the default tile structure of all attributes and descriptors belonging to the same attribute group.

If attribute_dependent_tiles == 1, additional tile structures are specified using the following fields:

n_add_tile_structures is the number of additional tile structures for the attribute group

n_attributes[i] is the number of attributes sharing the i^{th} additional tile structure

attribute_ID[i][j] is the j^{th} attribute ID associated with the i^{th} additional tile structure

n_descriptors[i] is the number of descriptors sharing the i^{th} additional tile structure

descriptor_ID[i][j] is the j^{th} descriptor ID associated with the i^{th} additional tile structure

additional_tile_structure[i] is the data structure specified in subclause 6.3.2.2 carrying the i^{th} additional tile structure

6.3.2.2 Tile structure

The tile data structure (see Table 10) specifies how annotation data is divided into rectangular tiles defined by ranges of row and column indices.

Table 10 — Tile structure syntax

Syntax	Type	Remarks
tile_structure {		
reserved	u(7)	
variable_size_tiles	u(1)	
n_tiles	u(v)	v dependent on AT_coord_size as specified in subclause 6.3.1
if (variable_size_tiles) {		
for (i=0; i<n_tiles; i++) {		
for (j=0; j<(two_dimensional + 1); j++) {		two_dimensional as specified in subclause 6.3.2.1
start_index[i][j]	u(v)	v dependent on AT_coord_size as specified in subclause 6.3.1
end_index[i][j]	u(v)	v dependent on AT_coord_size as specified in subclause 6.3.1
}		
}		
} else {		
for (j=0; j<(two_dimensional + 1); j++) {		
tile_size[j]	u(v)	v dependent on AT_coord_size as specified in subclause 6.3.1
}		
}		

Table 10 *(continued)*

Syntax	Type	Remarks
}		

reserved are bits used to preserve byte alignment and shall be set to 0.

variable_size_tiles is a flag, if set to 1, indicates that the size of each tile is different, and thus the corresponding start and end indices are specified independently. Otherwise, a uniform size applies to all tiles.

n_tiles specifies the total number of tiles defined in this tile structure. The number of bits for n_tiles is the same as the number of bits for annotation table coordinates, to allow having one tile per row or column in an annotation table.

start_index[i][j] and **end_index[i][j]** is the pair of start and end indices defining the range of rows (j == 0) or columns (j == 1) for the i^{th} rectangular tile, only used when variable_size_tiles == 1.

tile_size[j] specifies the number of rows (j == 0) or columns (j == 1) per tile, only used when variable_size_tiles == 0.

6.3.3 Annotation encoding parameters

The annotation encoding parameters (see Table 11) are configuration parameters used during the decoding process of an annotation access unit.

Table 11 — Annotation encoding parameters syntax

Syntax	Type	Remarks
annotation_encoding_parameters (){		
n_filter	u(8)	
for (i = 0; i < n_filter; i++){		
filter_ID_len[i]	u(6)	
filter_ID[i]	c(filter_ID_len[i])	
desc_len[i]	u(10)	
description[i]	c(desc_len[i])	
}		
n_features_names	u(8)	
for(i = 0; i < n_features_names; i++){		
features_name_len[i]	u(6)	
features_name[i]	c(features_name_len[i])	
}		
n_ontology_terms	u(8)	
for(i = 0; i < n_ontology_terms; i++){		
ontology_term_name_len[i]	u(6)	
ontology_term_name[i]	c(ontology_term_name_len[i])	
}		
n_descriptors	u(8)	
for(i = 0; i < n_descriptors; i++) {		
descriptor_configuration[i]		See subclause 6.3.4
}		
n_compressors	u(8)	
for(i = 0; i < n_compressors; i++){		

Table 11 *(continued)*

Syntax	Type	Remarks
compressor_parameter_set[i]		See subclause 6.3.5
}		
n_attributes	u(8)	
for(i = 0; i < n_attributes; i++){		
attribute_parameter_set[i]		See subclause 6.3.6
}		
while(!byte_aligned( )){		
nesting_zero_bit	f(1)	One bit set to 0
}		
}		

n_filter specifies the number of filters.

filter_ID_len[i] specifies the i-th filter id length.

filter_ID[i] specifies the i-th filter id as an array of filter_ID_len[i] characters.

desc_len[i] specifies the i-th feature description length.

description[i] specifies the i-th description as an array of desc_len[i] characters.

n_features_names specifies the number of features.

features_name_len[i] specifies the i-th feature name length.

features_name[i] specifies the i-th feature name as an array of features_name_len[i] characters.

n_ontology_terms specifies the number of ontology terms.

ontology_term_name_len[i] specifies the i-th ontology name length.

ontology_term_name[i] specifies the i-th ontology name as an array of ontology_name_len[i] characters.

n_descriptors specifies the number of available specific decoding descriptor configurations.

descriptor_configuration[i] contains a specific descriptor decoding configuration as specified in subclause 6.3.4.

n_compressors specifies the number of available compressor parameter sets.

compressor_parameter_set[i] contains a specific compressor decoder configuration as specified in subclause 6.3.5.

n_attributes specifies the number of available attribute parameter sets.

attribute_parameter_set[i] contains a specific attribute decoder configuration as specified in subclause 6.3.6.

6.3.4 Descriptor configuration

6.3.4.1 General

This subclause specifies the data structure used to represent the descriptor configuration (see Table 12). A descriptor configuration contains information for standard descriptor decoding.

Table 12 — Descriptor configuration syntax

Syntax	Type
descriptor_configuration {	
descriptor_ID	u(8)
encoding_mode_ID	u(8)
if(descriptor_ID == GENOTYPE){	
genotype_parameters()	See subclause 6.3.4.2
} else if(descriptor_ID == LIKELIHOOD){	
likelihood_parameters()	See subclause 6.3.4.3
} else if (descriptor_ID == CONTACT){	
contact_matrix_parameters()	See subclause 6.3.4.4
}	
algorithm_parameters(encoding_mode_ID)	See subclause 8.1
}	

descriptor_ID defines the unique identifier of the descriptor (see subclause 7.1) for which the descriptor configuration is defined. Table 39 lists the values of **descriptor_ID**, ranging from 1 to 22.

encoding_mode_ID identifies the codec (referenced by algorithm_ID) that shall be used for decompressing the descriptor blocks identified by descriptor_ID. Table 42 lists the values of algorithm_ID, ranging from 16 to 31.

genotype_parameters() is the data structure defined in subclause 6.3.4.2 carrying the additional genotype parameters when descriptor_ID is equal to 15 as specified in Table 39.

likelihood_parameters() is the data structure defined in subclause 6.3.4.3 carrying the additional likelihood parameters when descriptor_ID is equal to 16 as specified in Table 39.

contact_matrix_parameters() is the data structure defined in subclause 6.3.4.4 carrying the additional contact matrix parameters when descriptor_ID is equal to 22 as specified in Table 39.

algorithm_parameters(encoding_mode_ID) is the data structure defined in subclause 8.1 carrying the decoder configuration parameters for the specific encoding mode defined by encoding_mode_ID.

6.3.4.2 Genotype parameters

This subclause specifies the syntax (see Table 13) and semantic of genotype parameters structure for descriptor GENOTYPE.

Table 13 — Genotype parameters syntax

Syntax	Type
genotype_parameters(){	
max_ploidy	u(8)
no_reference_flag	u(1)
not_available_flag	u(1)
binarization_ID	u(3)
if (binarization_ID == BIT_PLANE){	
num_bit_plane	u(8)
concat_axis	u(8)
}	
for (i = 0; i < num_variants_payloads; i++) {	
sort_variants_rows_flag[i]	u(1)
sort_variants_cols_flag[i]	u(1)

Table 13 *(continued)*

Syntax	Type
transpose_variants_mat_flag[i]	u(1)
variants_codec_ID[i]	u(8)
}	
encode_phases_data_flag	u(1)
if (encode_phases_data_flag) {	
sort_phases_rows_flag	u(1)
sort_phases_cols_flag	u(1)
transpose_phases_mat_flag	u(1)
phases_codec_ID	u(1)
} else {	
phases_value	u(1)
}	
}	

max_ploidy signals the maximum number of haplotype or ploidy for each sample.

no_reference_flag, if set to 1, signals that the genotype payload contains no reference value (i.e., denoted with "."). Otherwise, genotype payload contains reference value.

not_available_flag, if set to 1, signals that the genotype payload contains genotype with ploidy lower than max_ploidy.

binarization_ID defines the binarization method and associated decoding process (see Table 14).

Table 14 — Binarization ID values and associated decoding process

binarization_ID	Value name	Description	Decoding process
0	BIT_PLANE	Bit plane	See subclause 9.3.5.7
1	ROW_SPLIT	Split by row	See subclause 9.3.5.9
2...7	-	Reserved for future use	-

num_bit_plane specifies the number of bit planes when binarization_ID equals to BIT_PLANE. It corresponds to the number of bits required to store a genotyping matrix value.

concat_axis defines the direction along which the binary matrices are concatenated as defined in Table 15 when binarization_id equals to BIT_PLANE.

Table 15 — Values of concat_axis and associated decoding process

concat_axis	Value name	Description
0	CONCAT_ROW_DIR	Concatenate matrices in row direction
1	CONCAT_COL_DIR	Concatenate matrices in column direction
2	DO_NOT_CONCAT	No concatenate operation
3...7	-	Reserved for future use

num_variants_payloads specifies the number of variants payload contained in the genotype payload structure specified in subclause 6.4.4.3.2. Table 16 list the values of num_variants_payloads.

Table 16 — Values of num_variants_payloads and associated condition

Condition	num_variants_payload
binarization_ID == BIT_PLANE && concat_axis == CONCAT_ROW_DIR	1
binarization_ID == BIT_PLANE && concat_axis == CONCAT_COL_DIR	1
binarization_ID == BIT_PLANE && concat_axis == DO_NOT_CONCAT	num_bit_plane
binarization_ID == ROW_SPLIT	1

sort_variants_rows_flag[i], if set to 1, signals that the rows of variants_payloads[] as specified in subclause 6.4.4.3.2 with index i are sorted.

sort_variants_cols_flag[i], if set to 1, signals that the columns of variants_payloads[] as specified in subclause 6.4.4.3.2 with index i are sorted.

transpose_variants_mat_flag[i], if set to 1, signals that the binary matrix decoded from variants_payloads[] as specified in subclause 6.4.4.3.2 with index i is transposed.

variants_codec_ID[i] specifies the codec identifier, with values from 16 to 31 as specified in Table 42, required to decode variants_payload[] as specified in subclause 6.4.4.3.2 with index i.

encode_phases_data_flag, if set to 1, signals that the phasing matrix contains only one distinct value.

sort_phases_rows_flag, if set to 1, signals that the rows of phases_payload as specified in subclause 6.4.4.3.2 are sorted. Otherwise, it is set to 0 for unsorted data.

sort_phases_cols_flag, if set to 1, signals that the columns of phases_payload as specified in subclause 6.4.4.3.2 are sorted. Otherwise, it is set to 0 for unsorted data.

transpose_phases_mat_flag, if set to 1, signals that the binary matrix decoded from phases_payload as specified in subclause 6.4.4.3.2 is transposed.

phases_codec_ID specifies the codec identifier, with values from 16 to 31 as specified in Table 42, required to decode phases_payload as specified in subclause 6.4.4.3.2.

phases_value defines the values contained in the phase matrix.

6.3.4.3 Likelihood parameters

This subclause specifies the syntax (see Table 17) and semantic of likelihood parameters structure for descriptor LIKELIHOOD.

Table 17 — Likelihood parameters syntax

Syntax	Type
likelihood_parameters(){	
num_gl_per_sample	u(8)
transform_flag	u(1)
if (transform_flag){	
dtype_id	u(8)
}	
}	

num_gl_per_sample is the number of likelihood values for each sample.

transform_flag, if set to 1, signals a 2-dimensional transformation. Otherwise, it is set to 0.

dtype_id defines the identifier of data type of 2-dimensional array C[][] as specified in subclause 9.3.6.1.

6.3.4.4 Contact matrix (CM) mat parameters

This subclause specifies the syntax (see Table 18) and semantic of CM mat parameters structure for descriptor CONTACT.

Table 18 — CM mat parameters syntax

Syntax	Type
cm_mat_parameters(){	
num_samples	u(8)
for (i=0; i<num_samples; i++){	
sample_ID[i]	u(8)
sample_name[i]	st(v)
}	
num_chrs	u(8)
for (i=0 i<num_chrs; i++)	
chr_ID[i]	u(8)
chr_name[i]	st(v)
chr_length[i]	u(64)
}	
interval # *(replace resolution)*	u(32)
tile_size	u(32)
num_interval_multipliers	u(8)
for (i=0; i<num_interval_multipliers; i++);	
interval_multiplier[i]	u(32)
}	
num_norm_methods	u(8)
for (i=0; i< num_norm_methods; i++){	
norm_method_ID[i]	u(8)
norm_method_name[i]	st(v)
norm_method_mult_flag[i]	u(1)
reserved	u(7)
}	
num_norm_matrices	u(8)
for (i=0; i< num_norm_matrices; i++){	
norm_matrix_ID[i]	u(8)
norm_matrix_name[i]	st(v)
}	
}	

num_samples specifies the number of samples.

sample_ID[i] is the unique identifier of the sample at index i.

sample_name[i] is the name of the sample at index i.

num_chrs is the number of chromosomes in the dataset.

chr_ID[i] is the unique identifier of the chromosome at index i.

chr_name[i] is the name of the chromosome with index i.

chr_length[i] is the length of the chromosome with index i.

interval is the bin size of the CM matrix structure specified in 6.4.4.3.4.3.

tile_size is the maximum number of entries of the CM tile structure specified in 6.4.4.3.4.4 in both row and column direction.

num_inverval_multipliers specifies the number of entries of array inverval_multipliers. If num_interval_multipliers is greater than 1, the structure supports multiple intervals.

interval_multiplier[i] is the interval multiplier supported by this structure at index i

num_norm_methods specifies the number the normalization methods which weights are stored.

norm_method_ID[i] is the unique identifier of the normalization method at index i which weights are stored in the CM bin payload structure specified in 6.4.4.3.4.2**.**

norm_method_name[i] is the name of the normalization method with index i which weights are stored.

norm_method_mult_flag[i] if set to 1, it signals that during the on-the-fly normalization each entry of the contact matrix tile shall be multiplied by the product of two weights. Otherwise, each entry of the contact matrix tile shall be divided by the product of two weights.

reserved are bits used to preserve byte alignment and shall be set to 0.

num_norm_matrix specifies the number of normalized matrices stored.

norm_matrix_ID[i] is the unique identifier of the normalized matrix at index i.

norm_matrix_name[i] is the name of the normalized matrix at index i.

num_bin_entries specifies the number of bins of the chromosome with the identifier chr_ID and depends on interval_multiplier and interval. It is computed as follows: Ceil(chr_length[chr_ID]/(interval_multiplier*interval)).

num_tiles specifies the number of tiles of the chromosome with the identifier chr_ID given interval, interval_multiplier and tile_size. It is computed as follows: Ceil(num_bin_entries/tile_size).

target_tile_size is the decoded or target tile_size given multiplier mult. It is computed as follows: target_tile_size = Floor(tile_size/mult).

target_interval is the interval of decoded tile. It is computed as follows: target_interval = interval*mult.

target_chr_len is the chromosome length chrs_length[i] with index i given chromosome id **chr**_ID equals to chr_ID[i].

6.3.4.5 Contact matrix (CM) submat parameters

This subclause specifies the syntax (see Table 19) and semantic of CM submat parameters structure for descriptor CONTACT.

Table 19 — CM submat parameters syntax

Syntax	Type
cm_submat_parameters(){	
chr1_ID	u(8)
chr2_ID	u(8)
for (i=0; i<ntiles_in_row; i++){	
for (j=0; j<ntiles_in_col; j++){	

Table 19 *(continued)*

Syntax	Type
`if (!(is_symmetrical && i > j)){`	
`diag_transform_flag[i][j]`	u(1)
`if(diag_tranform_flag[i][j]){`	
`diag_transform_mode[i][j]`	u(2)
`}`	
`binarization_flag[i][j]`	u(1)
`}`	
`}`	
`}`	
`row_mask_exists_flag`	u(1)
`col_mask_exists_flag`	u(1)
`while(!byte_aligned()){`	
`nesting_zero_bit`	f(1)
`}`	
`}`	

chr1_ID specifies the unique identifier of the first chromosome of the chromosome pair.

chr2_ID specifies the unique identifier of the second chromosome of the chromosome pair.

ntiles_in_row specifies the number of contact matrix tiles of CM matrix payload specified in 6.4.4.3.4.3, in row direction. It is inferred from the chr_length[] and tile_size fields in the Contact matrix (CM) mat parameters defined in subclause 6.3.4.4 as follow: ntiles_in_row = Ceil(nrows/tile_size) and nrows is inferred as follows: nrows = chr_length[i] where chr_ID[i] (defined in subclause 6.3.4.4) equals to chr1_ID.

ntiles_in_col is the number of contact matrix tiles of CM matrix payload specified in 6.4.4.3.4.3, in column direction. It is inferred from the chr_length[] and tile_size fields in the Contact matrix (CM) mat parameters defined in subclause 6.3.4.4 as follow: ntiles_in_col = Ceil(ncols/tile_size) and ncols is inferred as follows: ncols = chr_length[i] where chr_ID[i] (defined in subclause 6.3.4.4) equals to chr12_ID.

is_symmetrical is set to 1 if chr1_ID equals chr2_ID.

diag_transform_flag[i][j], if set to 1, signals that diagonal transformation is applied to the CM tile structure specified in 6.4.4.3.4.4 with index i for the First dimension and index j for the second dimension.**diag_transform_mode[i][j]** specifies the diagonal transform mode for the CM tile structure specified in 6.4.4.3.4.4 with index i for the first dimension and index j for the second dimension.

binarization_flag[i][j], if set to 1, signals that the CM tile structure specified in 6.4.4.3.4.4 with index i for the first dimension and index j for the second dimension is binarized using row binarization.

row_mask_exists_flag, if set to 1, signals that row_mask_payload exists in the CM matrix payload specified in 6.4.4.3.4.3.

col_mask_exists_flag, if set to 1, signals that col_mask_payload exists in the CM matrix payload specified in 6.4.4.3.4.3.

nesting_zero_bit is one bit set to 0.

6.3.5 Compressor parameter set

Compressor parameter set is a box that contains the definitions of a compressor, including the sequence of transform and compression algorithms with their parameter settings to be applied on an attribute (seen in Table 20**).**

Table 20 — Compressor parameter set syntax

Syntax	Type	Remarks
compressor_parameter_set {		
compressor_ID	u(8)	
n_compressor_steps	u(4)	
for (i=0; i<n_compressor_steps; i++) {		
compressor_step_ID[i]	u(4)	
algorithm_ID[i]	u(5)	
use_default_pars[i]	u(1)	
if(!use_default_pars[i]){		
algorithm_parameters(algorithm_ID)		As specified in subclause 8.1
}		
n_in_vars[i]	u(4)	
for (j=0; j<n_in_vars[i]; j++){		
in_var_ID[i][j]	u(4)	
prev_step_ID[i][j]	u(4)	
prev_out_var_ID[i][j]	u(4)	
}		
n_completed_out_vars[i]	u(4)	
for (j=0; j<n_completed_out_vars[i]; j++) {		
completed_out_var_ID[i][j]	u(4)	
}		
}		
while(!byte_aligned( )){		
nesting_zero_bit	f(1)	One bit set to 0
}		
}		

compressor_ID is the unique identifier of the compressor within the annotation table. Note that the value 0 is reserved for no compression.

N_compressor_steps is the number of processing steps in this compressor.

For each compressor step i, the following fields are specified:

compressor_step_ID[i] is the identifier of the compressor step.

algorithm_ID[i] is the ID of the algorithm to be applied in this step. The list of IDs and information on the corresponding algorithms are specified in subclause 8.1, Table 42.

use_default_pars[i] is a flag if set to 1 indicates the default parameters of the algorithm are used.

n_in_vars[i] is the number of input variables for the step.

For each input variable *j* of step *i*, the following fields are specified:

in_var_ID[i][j] is the ID of an input variable predefined for the algorithm referenced by algorithm_ID[i].

prev_step_ID[i][j] and prev_out_var_ID[i][j] are respectively the ID of the previous step and the ID of its output variable with data that is passed to the current step through the input variable referenced by in_var_ID[i][j].

n_completed_out_vars[i] is the number of output variables of step *i* that require no further processing in the subsequent steps of the compressor.

Completed_out_var_ID[i][j] is the ID of the *j* th completed output variable of step *i* that requires no further processing by the compressor.

6.3.6 Attribute parameter set

Attribute parameter set is a box that contains the definitions of an attribute, including some basic information and configuration of its associated compressor (see Table 21).

Table 21 — Attribute parameter set syntax

Syntax	Type
attribute_parameter_set {	
attribute_ID	u(16)
attribute_name_len	u(8)
attribute_name	c(attribute_name_len)
attribute_type	u(8)
attribute_num_array_dims	u(2)
for (i = 0; i < attribute_num_array_dims; i++) {	
attribute_array_dims[i]	u(8)
}	
attribute_default_val	x(v), where the type x (u or st) and v are dependent on attribute_type as specified in subclause 7.4
attribute_miss_val_flag	u(1)
if (attribute_miss_val_flag) {	
attribute_miss_default_flag	u(1)
if (!attribute_miss_default_flag) {	
attribute_miss_val	x(v), where the type x (u or st) and v are dependent on attribute_type as specified in subclause 7.4
}	
attribute_miss_str	st(v)
}	
compressor_ID	u(8)
n_steps_with_dependencies	u(4)
for (i = 0; i < n_steps_with_dependencies; i++) {	
dependency_step_ID[i]	u(4)
n_dependencies[i]	u(4)
for (j = 0; j < n_dependencies[i]; j++) {	
dependency_var_ID[i][j]	u(4)
dependency_is_attribute[i][j]	u(1)
if (dependency_is_attribute[i][j]) {	
dependency_ID	u(16)

Table 21 *(continued)*

Syntax	Type

} else {
dependency_ID
}
}
}
while(!byte_aligned( )){
nesting_zero_bit
}
}
	u(7)
	
	f(1)

attribute_ID is the identifier of the attribute, unique within an annotation parameter set.

attribute_name_len is the length of the attribute name.

attribute_name is the name of the attribute with length attribute_name_len.

attribute_type specifies the data type of the attribute as specified in Table 40.

attribute_num_array_dims specifies the number of array dimensions of an attribute entry, 0 if it is a scalar value.

attribute_array_dims[] contains elements each specifying the size of an array dimension. It is omitted if attribute_num_array_dims = 0.

attribute_default_val is the default value of the attribute (i.e., it can be used for sparse encoding when most values equal to the default are excluded).

attribute_miss_val_flag is a flag, if set to 1, indicates the attribute contains missing values. Otherwise, it has no missing value.

attribute_miss_default_flag is a flag, if set to 1, indicates that the bit pattern consisting of all ones is used to represent empty or NULL values. It is only specified if attribute_miss_val_flag == 1 and only applicable if the attribute type is numeric (signed/unsigned, integer/float/double),

attribute_miss_val is the value for representing empty or NULL values. It is only specified if attribute_miss_flag == 1 and attribute_miss_default_flag == 0.

attribute_miss_str is the string to be presented in place of a missing value of the attribute after the decoding process. It is only specified if attribute_miss_flag == 1.

Compressor_ID is the ID of one of the compressor parameter sets, as specified in subclause 6.3.5, defined in the dataset that is used for the coding of this attribute.

n_steps_with_dependencies is the number of steps in the compressor referenced by compressor_ID that involve dependency variables for its process.

For each of the steps in the compressor referenced by compressor_ID that involves dependency variables, the following fields are specified:

dependency_step_ID[i] is the ID of the compressor step, as defined in subclause 6.3.5, with dependency variables that need to be specified.

n_dependencies[i] is the number of dependency variables for the compressor step referenced by compressor_step_ID[i]

For each of the dependency variables of the compressor step referenced by compressor_step_ID[i], the following fields are specified:

dependency_var_ID[i][j] is the ID of the *j* th dependency variable that is predefined for the algorithm associated with the compressor step.

dependency_is_attribute[i][j] is a flag if set to 1 indicates that the variable containing the data to be passed to the algorithm of the compressor step is an attribute. Otherwise, it is a descriptor.

dependency_ID is the ID of the descriptor/attribute containing data to be passed to the algorithm of the compressor step through the dependency variable referenced by dependency_var_ID[i][j].

nesting_zero_bit is one bit set to 0.

6.4 Annotation access unit

6.4.1 General

The Annotation Access Unit (AAU) (see Table 22) contains the compressed representation of genomic annotations.

Table 22 — Annotation access unit syntax

Syntax	Type
annotation_access_unit() {	
AT_ID	u(8)
AT_type	u(4)
AT_subtype	u(4)
AG_class	u(3)
reserved	u(5)
annotation_access_unit_header	See subclause 6.4.2
for (i=0; i < n_blocks; i++) {	n_blocks as specified in subclause 6.4.2
block[i]	See subclause 6.4.4
}	
}	

AT_ID is the annotation table ID used for identifying the dataset parameter set that contains the parameters required for the decoding of this annotation access unit.

AT_type is the type of the genomic data encoded in the associated annotation table. Possible values are specified in Table 24 of subclause 6.4.3.

AT_subtype is the subtype of the genomic data encoded in the associated annotation table. Possible values are specified in Table 25 of subclause 6.4.3.

AG_class is the attribute group class to which this annotation access unit belongs, used for identifying the attribute group-specific tile configuration required for the decoding of this annotation access unit.

reserved are bits used to preserve byte alignment and shall be set to 0.

annotation_access_unit_header is the annotation access unit header as specified in subclause 6.4.2

block[i] is the data structure carrying the block data specified in subclause 6.4.4.

6.4.2 Annotation access unit header

This subclause specifies the annotation access unit header syntax (see Table 23) and its semantics. This structure contains only selected fields of the annotation_access_unit_header defined in ISO/IEC 23092-1 required for the decoding process.

Table 23 — Annotation access unit header syntax

Syntax	Type	Remarks
annotation_access_unit_header {		
if (attribute_contiguity) {		As specified in subclause 6.3.1
is_attribute	u(1)	
if (is_attribute) {		
attribute_ID	u(16)	
} else {		
descriptor_ID	u(7)	
}		
if (two_dimensional && !variable_size_tiles) {		two_dimensional as specified in subclause 6.3.1 variable_size_tiles as specified in subclause 6.3.2.2 in the tile structure associated with the descriptor/attribute
if (column_major_tile_order) {		As specified in subclause 6.3.1
n_tiles_per_col	u(v)	v depends on AT_coord_size as specified in subclause 6.3.1
} else {		
n_tiles_per_row	u(v)	v depends on AT_coord_size as specified in subclause 6.3.1
}		
n_blocks	u(v)	v depends on AT_coord_size as specified in subclause 6.3.1
}		
} else {		
tile_index_1	u(v)	v depends on AT_coord_size as specified in subclause 6.3.1
tile_index_2_exists	u(1)	
if (tile_index_2_exists) {		
tile_index_2	u(v)	v depends on AT_coord_size as specified in subclause 6.3.1
}		
n_blocks	u(16)	
}		
while(!byte_aligned( ))		
nesting_zero_bit	f(1)	One bit set to 0
}		

If attribute_contiguity == 1,

is_attribute is a flag, if set to 1, indicates that this annotation access unit corresponds to an attribute. Otherwise, it corresponds to a descriptor.

If **is_attribute** == 1,

attribute_ID is the ID of the attribute whose data is contained in this annotation access unit.

If **is_attribute** == 0,

descriptor_ID is the ID of the descriptor whose data is contained in this access unit.

If the attribute is 2-dimensional and variable_size_tiles == 0,

n_tiles_per_col is the number of tiles per column, only specified if column_major_tile_order == 1.

n_tiles_per_row is the number of tiles per row, only specified if column_major_tile_order == 0.

If attribute_contiguity == 0,

tile_index_1 is the index of the tile whose data is contained in this annotation access unit. If two_dimensional == 1 and variable_size_tiles == 0, it corresponds to the row index of the tile.

tile_index_2_exists is a flag, if set to 1, indicates that a second tile index exists.

If (**tile_index_2_exists** == 1),

tile_index_2 is the column index of the tile whose data is contained in this annotation access unit, only specified if two_dimensional == 1 and with variable_size_tiles == 0.

n_blocks is the number of blocks contained in the annotation access unit.

6.4.3 Annotation access unit types

AAUs can be of different types according to the nature of the coded annotation table data. An annotation access unit contains encoded annotation table records belonging to a single annotation table type as shown in Table 24.

Table 24 — Class of encoded data per annotation table type

Annotation table type		Data type
Ann table type name	**Value**	
VARIANTS	1	Genomic variants
FUNCTIONAL_ANNOTATIONS	2	Functional annotations
GENE_EXPRESSION	3	Gene expression values
CONTACT_MATRICES	4	Position-to-position contact intensity values
TRACKS	5	Genome browser tracks

For each annotation table type, the sub-type specifies the external data formats used as template as specified in Table 25.

Table 25 — Annotation table sub-type encoding

Data Type	Annotation table sub-type	
	Ann table sub-type name	**Value**
Any	UNDEFINED	0
VARIANTS	VCF	1
FUNCTIONAL_ANNOTATIONS	GTF	2
	GFF	3
	GENBANK	8

Table 25 *(continued)*

Data Type	Annotation table sub-type	
	Ann table sub-type name	Value
TRACKS	BED	4
	BEDGRAPH	5
	WIG	6
	BIGWIG	7
CONTACT_MATRICES	HIC	10
GENE_EXPRESSION	GENE_EXPRESSION	9

6.4.4 Block

6.4.4.1 General

This subclause describes the syntax (see Table 26) and semantics of block structure.

Table 26 — Annotation access unit block syntax

Syntax	Type
`block() {`	
`  block_header()`	See subclause 6.4.4.2
`  block_payload()`	See subclause 6.4.4.3
`}`	

block_header is the data structure carrying the block header specified in subclause 6.4.4.2.

block_payload is the data structure carrying the block payload specified in subclause 6.4.4.3.

6.4.4.2 Block header

This subclause describes the block header syntax (see Table 27) and semantics.

Table 27 — Block header syntax

Syntax	Type	Remarks
`block_header() {`		
`  if (attribute_contiguity == 0) {`		As specified in subclause 6.3.2.1
`    descriptor_ID`	`u(8)`	
`    if (descriptor_ID == ATTRIBUTE) {`		
`      attribute_ID`	`u(16)`	
`    }`		
`  }`		
`  reserved`	`u(2)`	
`  indexed`	`u(1)`	
`  block_payload_size`	`u(29)`	
`}`		

descriptor_ID signals the descriptor identifier as specified in subclause 7.1.

attribute_ID if descriptor_ID is set to ATTRIBUTE (see Table 39), it signals the attribute identifier.

reserved are bits used to preserve byte alignment and shall be set to 0.

indexed, is set to 1, if the block has been compressed with PROCRUSTES as specified in Table 42, and 0 otherwise.

block_payload_size specifies the size of the block payload in bytes.

6.4.4.3 Block payload

6.4.4.3.1 General

This subclause specifies the syntax (see Table 28) and semantics of the block payload structure containing entropy-coded descriptors or attributes.

Table 28 — Block payload syntax

Syntax	Type
block_payload(descriptor_ID) {	
if (**descriptor_ID** == GENOTYPE) {	
genotype_payload()	See subclause 6.4.4.3.2
} else if(**descriptor_ID** == LIKELIHOOD){	
likelihood_payload()	See subclause 6.4.4.3.3
} else if (**descriptor_ID** == CONTACT){	
for (i=0; i<num_chrs;i++)	num_chrs is defined in 6.3.4.4
cm_bin_payload[i]	See subclause 6.4.4.3.4.2
}	
cm_mat_payload()	See subclause 6.4.4.3.4.3
} else {	
generic_payload()	c(block_payload_size)
}	
while(!byte_aligned()){	
nesting_zero_bit	f(1)
}	
}	

genotype_payload is the data structure specified in 6.4.4.3.2 carrying encoded genotype payload.

likelihood_payload is the data structure specified in 6.4.4.3.3 carrying encoded likelihood payload.

cm_bin_payload[i] is the data structure specified in 6.4.4.3.4.2 carrying CM bin payload of the i^{th} chromosome.

num_chrs specifies the number of chromosomes in the payload as specified in subclause 6.3.4.4.

cm_mat_payload is the data structure specified in 6.4.4.3.4.3 carrying encoded CM matrix payload.

generic_payload is the encoded data buffer carrying descriptor or attribute encoded data.

nesting_zero_bit is one bit set to 0.

6.4.4.3.2 Genotype payload

6.4.4.3.2.1 General

This subclause specifies the syntax (see Table 29) and semantics of the genotype payload structure, containing entropy-coded genotype information.

Table 29 — Genotype payload syntax

Syntax	Type
genotype_payload() {	
for (i = 0; i < num_variants_payloads; i++) {	See subclause 6.3.4.2.
variants_payload_size[i]	u(32)
variants_payload[i]	See subclause 6.4.4.3.2.2
if (sort_variants_row_flag[i]) {	See subclause 6.3.4.2.
sort_variants_row_ids_payload_size[i]	u(32)
sort_variants_row_ids_payload[i]	See subclause 6.4.4.3.2.3
}	
if (sort_variants_col_flags[i]) {	See subclause 6.3.4.2.
sort_variants_col_ids_payload_size[i]	u(32)
sort_variants_col_ids_payload[i]	See subclause 6.4.4.3.2.3
}	
}	
if (binarization_id == ROW_SPLIT) {	binarization_ID is specified in subclause 6.3.4.2
variants_amax_payload_size	u(32)
variants_amax_payload	See subclause 6.4.4.3.2.4
}	
if (encode_phase_data) {	See subclause 6.3.4.2.
phases_payload_size	u(32)
phases_payload	See subclause 6.4.4.3.2.2
}	
}	

num_variants_payloads is specified in subclause 6.3.4.2.

variants_payload_size[i] is the size in bytes of the binary matrix payload structure named variants_payloads[i] with index i.

variants_payloads[i] is the binary matrix payload structure as specified in subclause 6.4.4.3.2.2.

sort_variants_rows_flag[i] signals whether the rows of variants_payloads structure specified with index i are sorted as specified in subclause 6.3.4.2.

sort_variants_row_ids_payload_size[i] is the size in bytes of the row_col_ids_payload structure sort_variants_row_ids_payloads with index i.

sort_variants_row_ids_payload[i] is the row_col_ids_payload structure with index i as specified in subclause 6.4.4.3.2.3.

sort_variants_cols_flag[i] signals whether the columns of variants_payloads with index i are sorted.

Sort_variants_col_IDs_payload_size[i] is the size in bytes of the row_col_ids_payload structure sort_variants_col_ids_payloads with index i.

sort_variants_col_IDs_payload[i] is the row_col_ids_payload structure with index i as specified in subclause 6.4.4.3.2.3.

binarization_ID is specified in subclause 6.3.4.2 and signals the binarization method used. Table 30 lists the possible values of binarization_ID and associated decoding process.

variants_amax_payload_size is the size in bytes of the binary matrix payload structure named variants_amax_payload.

variants_amax_payload is the amax_payload structure specified in subclause 6.4.4.3.2.4.

encode_phases_data_flag is specified in subclause 6.3.4.2.

phases_payload_size is the size in bytes of the binary matrix payload structure named phases_payload**.**

phases_payload is the binary matrix payload structure specified in subclause 6.4.4.3.2.2.

6.4.4.3.2.2 Binary matrix payload

This subclause specifies the syntax (see Table 30) and semantics of the binary matrix payload structure, containing entropy-coded binary matrix.

Table 30 — Bin mat payload syntax

Syntax	Type
bin_mat_payload() {	
if (codec_ID == JBIG) {	
payload	See ISO/IEC 11544
} else if (codec_ID == CABAC) {	
nrows	u(32)
ncols	u(32)
payload()	See ISO/IEC 23092-2
}	
}	

codec_ID specifies the compression algorithm identifier compression_algorithm_id required to decode either jbig_payload() or cabac_payload(). The value of codec_ID shall be CABAC or JBIG as specified in Table 42. The codec_ID is specified as variant_codec_ID for variants_payload[i] or phases_codec_ID for phases_payload, respectively, in subclause 6.3.4.2.

nrows is the number of rows of the decoded binary matrix.

ncols is the number of columns of the decoded binary matrix.

payload is a JBIG BIE (Bi-level Image Entity) structure compliant to ISO/IEC 11544. Otherwise, payload is a bitstream compliant to ISO/IEC 23092-2. To decode the bitstream, if codec_ID is CABAC, the following ISO/IEC 23092-2 syntax element values are assumed: transform_ID_subseq is 0, transform_ID_subsym is 0, binarization_ID is 0, coding_sybsym_size is 1, mpegg_symbol_size is 1, coding_order is 2, bypass_flag is 0, adaptive_mode_flag is 1 and num_contexts is 2.

6.4.4.3.2.3 Row column ids payload

This subclause specifies the syntax (see Table 31) and semantics of the row col IDs payload structure.

Table 31 — Row col IDs payload syntax

Syntax	Type
`row_col_ids_payload() {`	
`  for (i=0; i < nelements; i++){`	nelements is specified in subclause 9.3.5.3
`    row_col_ids_elements[i]`	u(nbits_per_elem)
`  }`	
`}`	

nelements is the number of elements of array row_col_ids[]. The value is either the number of rows nrows or the number of columns ncols in decoding process described in subclause 9.3.5.3.

nbits_per_elem is the bit length of each element and it is inferred with the formula Ceil(Log2(num_entries)), where num_entries is either nrows or ncols of bin_mat[][] as specified in subclause 9.3.5.3.

row_col_ids_elements[i] is the value of element of array row_col_ids[] with index i.

6.4.4.3.2.4 Amax payload

This subclause specifies the syntax (see Table 32) and semantics of the amax payload structure.

Table 32 — Amax payload syntax

Syntax	Type
`amax_payload() {`	
`  nelems`	u(32)
`  nbits_per_elem`	u(8)
`  for (i=0; i<nelements; i++){`	nelements is specified in subclause 9.3.5.3
`    is_one_flag[i]`	b(1)
`    if (is_one_flag[i])`	
`      amax_elements[i]`	u(nbits_per_entry)
`    }`	
`  }`	
`}`	

nelems specifies the number of elements of 1-dimensional array amax[] in the process described in subclause 9.3.5.2.

nbits_per_elem is the bit length of each element of the array amax[] in the process described in subclause 9.3.5.2.

is_one_flag[i], if set to 1, signals the value of entry of array amax[] at index i is 1 in the process described in subclause 9.3.5.2.

amax_elements[i] is the value with offset of amax[] with index i. nbits_per_entry in the process described in subclause 9.3.5.2. Thevalue of amax[i] is 1 if is_one_flag[i] is one, otherwise: amax[i] = amax_elements[i] + 1.

6.4.4.3.3 Likelihood payload

This subclause specifies the syntax (see Table 33) and semantics of the likelihood payload structure.

Table 33 — Likelihood payload syntax

Syntax	Type
likelihood_payload() {	
nrows	u(32)
ncols	u(32)
payload_size	u(32)
payload	u(payload_size)
if (transform_flag){	
additional_payload_size	u(32)
additional_payload	u(additional_payload_size)
}	
}	

transform_flag, if set to 1, signals a 2-dimensional transformation as specified in subclause 6.3.4.3.

nrows describes the number of rows of 2-dimensinal code array L[][].

ncols describes the number of rows of 2-dimensinal code array L[][].

payload_size is the size in bytes of the payload field.

payload is the compressed array L[][]. The size of this payload is payload_size bytes.

additional_payload_size is the size in Bytes of the payload field.

additional_payload is the compressed array values V[][]. The size of this field is additional_payload_size bytes.

6.4.4.3.4 Contact matrix (CM) payload

6.4.4.3.4.1 General

This subclause specifies the syntax and the semantic related to the contact matrix (CM) payload.

6.4.4.3.4.2 CM bin payload

This subclause specifies the syntax (see Table 34) and semantics of the CM bin payload structure.

Table 34 — Contact matrix bin payload syntax

Syntax	Type
cm_bin_payload(){	
chr_ID	u(8)
sample_ID	u(16)
interval_multiplier	u(32)
for (i =0; i<num_norm_methods; i++){	num_norm_methods is specified in 6.3.4.4
for (j =0; j<num_bin_entries; j++){	num_bin_entries is specified in 6.3.4.4
weight_values[i][j]	f(64)
}	
}	
}	

chr_ID is the identifier of the chromosome.

sample_ID is the identifier of the sample.

interval_multiplier specifiy the multiplier of the interval to compute the num_bin_entries. The valid values for interval_multiplier is one of the entries of the array interval_multipliers specified in 6.3.4.4.

num_norm_methods is the number the normalization methods which weights are stored, specified in 6.3.4.4.

num_bin_entries is the number of bins of the chromosome with the identifier chr_id and depends on interval_multiplier and interval, specified in 6.3.4.4.

weight_values[i][j] is the j^{th} weight value of the i^{th} normalization method.

6.4.4.3.4.3 CM matrix payload

This subclause specifies the syntax (see Table 35) and semantics of the CM mat payload structure.

Table 35 — CM matrix payload syntax

Syntax	Type
cm_mat_payload(){	
sample_ID	u(8)
for(i=0; i<ntiles_in_row; i++){	
for(j=0; j<ntiles_in_col; j++){	
if (!(is_symmetrical && i>j)) {	
tile_codec_ID[i][j]	u(8)
tile_payload_size[i][j]	u(32)
tile_payload[i][j]	codec payload
}	
}	
}	
for (k=0; k< num_norm_matrices; k++){	
for(i=0; i<ntiles_in_row; i++){	
for(j=0; j<ntiles_in_col; j++){	
if (!(is_symmetrical && i>j)) {	
norm_matrix_payload_size[k][i][j]	u(32)
norm_matrix_payload[k][i][j]	codec payload
}	
}	
}	
}	
if (row_mask_exists_flag){	
row_mask_payload_size	u(32)
row_mask_payload()	CM mask payload
}	
if (!mirror_flag && col_mask_exists_flag){	
col_mask_payload_size	u(32)
col_mask_payload()	CM mask payload
}	
}	

sample_ID is a unique identifier of the sample represented by the CM matrix payload. Valid values for sample_ID is specified in 6.3.4.4.

ntiles_in_row equals num_tiles for chromosome with identifier chr1_ID. num_tiles is specified in 6.3.4.4 and chr1_ID is specified in 6.3.4.5.

ntiles_in_col equals num_tiles for chromosome with identifier chr2_ID. num_tiles is specified in 6.3.4.4 and chr2_ID is specified in 6.3.4.5.

is_symmetrical, is set to 1, if chr1_ID equals to chr2_ID, otherwise 0.

tile_codec_ID[i][j] is the unique codec identifier of tile with index i and j that specifies the decoding method of tile_tile_payloads with index i and j. The values of tile_codec_ID is listed in Table 42. It can range between 0 to 5 inclusive.

tile_payload_size[i][j] is the size in bytes of tile_payloads at index i and j.

tile_payload[i][j]() is the CM tile payload structure as specified in 6.4.4.3.4.4 with index i and j.

num_norm_matrices is the number of normalized matrices as specified in 6.3.4.4.

norm_matrix_payload_size[k][i][j] is the size in bytes of the compressed k^{th} normalized CM matrix with index i and j.

norm_matrix_payload[k][i][j] is the payload of the compressed k^{th} normalized contact matrix with index i and j.

row_mask_payload_size is the size in bytes of the CM mask payload structure as specified in 6.4.4.3.4.5.

row_mask_payload() is the CM mask payload structure as specified in 6.4.4.3.4.5.

col_mask_payload_size is the size in bytes of the CM mask payload structure as specified in 6.4.4.3.4.5.

col_mask_payload() is the CM mask payload structure as specified in 6.4.4.3.4.5. If the is_symmetrical of the corresponding parameter set is set to 1, the content of col_mask_payload is identical to the row_mask_payload and therefore is not stored.

6.4.4.3.4.4 CM tile payload

This subclause specifies the syntax (see Table 36) and semantics of the CM tile payload structure.

Table 36 — CM tile payload syntax

Syntax	Type
cm_tile_payload(){	
codec_ID	u(8)
if (codec_ID != JBIG) {	
tile_nrows	u(32)
tile_ncols	u(32)
payload	codec payload
}	
else {	
payload	JBIG BIE
}	
}	

codec_ID specifies the unique codec identifier that specifies the decoding method of payload. The values of codec_ID is listed in Table 42. It can range between 0 to 5 inclusive.

tile_nrows is the number of rows of the CM tile structure.

tile_ncols is the number of columns of the CM tile structure.

payload is a bitstream correspond to a decompression method according to codec_ID. If codec_ID is JBIG, payload is a JBIG BIE structure compliant to ISO/IEC 11544. The payload_size is inferred from tile_payload_size[i][j] in the CM matrix payload structure as follows: payload_size = tile_payload_size[i][j] - 1 for coded_ID equals to JBIG and payload_size = tile_payload_size[i][j] -9 for codec_ID != JBIG with i and j are the indices of the corresponding CM tile payload.

6.4.4.3.4.5 CM mask payload

This subclause specifies the syntax (see Table 37) and semantics of the CM mask payload structure.

Table 37 — CM mask payload syntax

Syntax	Type
cm_mask_payload(){	
transform_ID	u(2)
if (transform_id == 0){	
first_val	u(1)
for (i=0; i<num_bin_entries; i++){	
mask_array[i]	u(1)
}	
}	
else {	
for (i=0; i<num_rl_entries; i++){	
rl_content[i]	u(nbits_per_val)
}	
}	
while(!byte_aligned()){	
nesting_zero_bit	f(1)
}	
}	

transform_ID, if set to value other than zero, signals that CM mask payload structure is transformed using run-length encoding. The number of bits required to store each value of CM mask payload structure and whether the payload is run-length encoding transformed is specified in Table 38.

Table 38 — Values of transform_id and associated transformation flags and parameters

transform_ID	Enable run-length transformation	nbits_per_val
0	false	1
1	true	8
2	true	16
3	true	32

num_bin_entries is the number of bins of the chromosome with the identifier chr_ID and depends on interval_multiplier and interval, specified in 6.3.4.4.

mask_array[i] is the mask value at index i.

first_val is the first value of the CM mask payload structure if transform_ID != 0. The first value is used to inverse transform the run-length encoded mask array.

rl_content[i] is the value of run-length encoded mask at index i.

nesting_zero_bit is one bit set to 0.

7 Descriptors and attributes semantics

7.1 General

This clause specifies the semantics of descriptors and the syntax of attributes as define in Table 39. Descriptors are the output of the decoding process specified in Clause 9. Attributes are data structures representing genomic annotations which cannot be represented by a single numeric or textual descriptor.

Table 39 — Descriptors used to represent genomic annotations

descriptor_ID	Name	Type	Number of descriptor sub-sequences	Semantics
1	SEQUENCEID	uint16	1	Identifier of a reference sequence (e.g. chromosome)
2	STARTPOS	uint64	1	Position, on the reference sequence identified by sequenceID, of the first base of the annotation.
3	ENDPOS	uint64	1	Position, on the reference sequence identified by sequenceID, of the last base of the annotation.
4	STRAND	b(2)	1	0 = forward strand, 1 = reverse strand, 2 = undefined strand
5	NAME	string	1	Short text associated with the annotation
6	DESCRIPTION	string	1	Long text associated with the annotation
7	LINKNAME	string	1	Linked reference name (e.g. annotation name or sequence read name)
8	LINKID	uint8	1	Identifier of the reference box pointing to an internal or external dataset or annotation table
9	DEPTH	float32	1	Read depth for the variant
10	SEQQUALITY	float32	1	Root Mean Square (RMS) sequencing quality of the bases supporting the variant
11	MAPQUALITY	float32	1	RMS mapping quality
12	MAPNUMQUALITY0	uint32	1	Number of reads with MAPQ == 0 covering this record

Table 39 *(continued)*

descriptor_ID	Name	Type	Number of descriptor sub-sequences	Semantics
13	REFERENCE	string	1	Reference base(s): Each base shall be one of the symbols in the alphabet identified by alphabetID (case insensitive). The value in the startPos descriptor refers to the position of the first base in the String
14	ALTERN	string	1	Alternate base(s): list of alternate non-reference alleles.
15	GENOTYPE	string	1	Genotype information.
16	LIKELIHOOD	float32[]	1	Genotype likelihood information.
17	FILTER	uint8	1	Position plus 2 of the filter description in the list carried by the descriptor configuration specified in subclause 9.3.3.5 and Table 81; 0 represents "Pass" 1 represents "Missing"
18	FEATURENAME	uint32	1	Index of the textual identifier of the feature, defined in subclause 6.3.3.
19	FEATUREID	uint32	1	ID from an Ontology, defined in subclause 6.3.1.
20	ONTOLOGYNAME	uint8	1	Textual identifier from a given Ontology (such as Pseudogene, mRNA, scRNA), defined in subclause 6.3.1.
21	ONTOLOGYID	uint	1	Numerical Identifier from a given Ontology
22	CONTACT	—	Variable, as specified in subclause 9.3.7	Sparse representation of contact matrix that consists of multiple fields. The exact data type depends on the decoded field as defined in subclause 9.3.7.
23..30	Reserved for future use			
31	ATTRIBUTE		1	Identifier used to signal a generic configurable attribute

7.2 Descriptors

7.2.1 General

This subclause specifies descriptors which are common to all annotations referring to genomic intervals on a reference sequence.

7.2.2 Genomic intervals

Values from 0 to 8 inclusive specified in Table 39 provide the syntax and the semantics of descriptors which are common to genomic variants, functional annotations and tracks to represent genomic regions and associated textual descriptions.

7.2.3 Genomic variants

Values from 9 to 17 inclusive specified in Table 39 provide the syntax and the semantics of descriptors representing genomic variant features.

7.2.4 Functional annotations

Values from 18 to 21 inclusive specified in Table 39 provide the syntax the and semantics of descriptors representing functional annotations.

7.2.5 Contact matrices

Value 22 to 23 specified in Table 39 provides the syntax and the semantics of descriptors representing contact matrices.

7.3 Attributes

Value 31 specified in Table 39 signal a configurable attribute.

7.4 Data types

7.4.1 General

This clause specifies the syntax of data types as specified in Table 40.

Table 40 — Supported data types for the variables of an algorithm

Data Type ID	Name	Description	Number of Bits	Remarks
0	`string`	null terminated string	Variable	
1	`char`	character	8	
2	`bool`	boolean flag	1	
3	`int8`	integer	8	Value 0x80 represents missing value
4	`uint8`	unsigned integer	8	Value 0xFF represents missing value
5	`int16`	integer	16	Value 0x8000 represents missing value
6	`uint16`	unsigned integer	16	Value 0xFFFF represents missing value
7	`int32`	integer	32	Value 0x80000000 represents missing value

Table 40 *(continued)*

Data Type ID	Name	Description	Number of Bits	Remarks
8	uint32	unsigned integer	32	Value 0xFFFFFFFF represents missing value
9	int64	integer	64	Value 0x800000000000000000 represents missing value
10	uint64	unsigned integer	64	Value 0xFFFFFFFFFFFFFFFF represents missing value
11	float	IEEE-754 32 bits	32	Value 0x7FC00000 represents not a number and value 0x7F800001 represents missing value
12	double	IEEE-754 64 bits	64	Value 0x7FFF000000000000 represents not a number and value 0x7FFF000000000001 represents missing value
13..255	Reserved	-	-	

7.4.2 Typed data

This clause specifies the syntax (see Table 41) of typed data structure that includes metadata for preserving the data type and array form of a data block. All input/output/dependency variables defined in the decoding processes and algorithms implicitly adopt this typed data structure, as specified in subclause 7.4.2, to ensure them being correctly processed when passed between operations.

Table 41 — Typed data syntax

Syntax	Type	Remarks
typed_data {		
data_type_ID	u(8)	As specified in Table 40
num_array_dims	u(2)	Number of array dimensions, 0 for scalar value
n_elements = 1		
for (i = 0; i < num_array_dims; i++) {		
array_dims[i]	u(32)	Size of the i^{th} array dimension
n_elements = n_elements * array_dims[i]		Number of elements in the array
}		
for (i = 0; i < n_elements; i++) {		
data_block[i]	u(v)	v depends on the number of bits of the data type ID as specified in Table 40
}		
}		

A function td_to_matrix(t_dat), where the input variable t_dat is of the typed_data structure, is defined to convert the data_block array within t_dat to a matrix with dimensions as specified in array_dims and with the data type associated with data_type_ID as specified in Table 40. Note that the values in t_dat is in row-major order. For example, if num_array_dims == 3 (a three-dimensional array), the assignment of values in the output matrix is given by out_matrix[i][j][k] = t_dat[m*n*i + n*j + k], where m = array_dims[1] and n = array_dims[2] are the total numbers of elements in respectively the second and third dimensions.

8 Decompression codecs

8.1 General

This clause specifies the interface and processing steps of the list of decompression and inverse-transformation algorithms summarized in Table 42.

Table 42 — Transformation and decompressor identifiers

algorithm_ID	Name	Description	Reference to Inverse Transformation or Decoding Algorithm
0	CABAC	Context-Adaptive Binary Arithmetic Coding	ISO/IEC 23092-2
1	LZMA	Lempel-Ziv-Markov Chain Algorithm	ISO/IEC 23092-3
2	ZSTD	Zstandard	IETF RFC 8478
3	BSC	Block Sorting Coder	ISO/IEC 23092-2
4	PROCRUSTES	FM-index based compression	ISO/IEC 23092-1
5	JBIG	Joint Bi-level Image Experts Group	ISO/IEC 11544
6...15	Reserved	-	-
16	LZW	Lempel-Ziv-Welch	As specified in subclause 8.2.2
17	BIN	Binarization	As specified in subclause 8.2.3
18	Sparse	Sparse Transform	As specified in subclause 8.2.4
19	DEL	Delta Transform	As specified in subclause 8.2.5
20	RLE	Run-length Encoding	As specified in subclause 8.2.6
21	SER	Serialize Transform	As specified in subclause 8.2.7
20...31	Reserved	-	-

Four categories of variables define the interface of each algorithm:

— Input parameters – static value settings that determine the operation condition of the algorithm

— Input data variables – variables that contain the input data on which the algorithm is applied and any associated dynamic data required by the algorithm

— Dependency data variables – variables that contain the dependency data required by the algorithm to operate on the input data

— Output data variables – variables that contain the output data computed by the algorithm and any associated dynamic data required by future inverse operations to reconstruct the input data

All input/output/dependency variables adopt the typed_data structure to ensure them being correctly processed when passed between operations. The roles of input and output data variables defined in the decompression and inverse-transformation algorithms will be swapped when used in the compression and transformation steps of an encoding process specified in Compressor Parameter Set.

For each input parameter, the following information should be provided within an algorithm:

— Parameter ID – identifier of the parameter

— Name – name of the parameter

— Data type – data type ID of the parameter (see Table 40)

— Number of dimensions – number of array dimensions of the parameter, 0 if it is a scalar value

— Dimensions – a vector of elements each specifying the size of an array, not required if the parameter is scalar

— Default value(s)– the values(s) assumed by the parameter by default

Algorithm parameters is a data structure for specifying the parameter settings of the algorithm referenced by algorithm_ID. Its syntax is specified in Table 43.

Table 43 — Algorithm parameters syntax

Syntax	Type	Remarks
algorithm_parameters(algorithm_ID) {		
n_pars	u(4)	
for (i = 0; i < n_pars; i++) {		
par_ID[i]	u(4)	
par_type[i]	u(8)	
par_num_array_dims[i]	u(2)	
for (j = 0; j < par_num_array_dims[i]; j++) {		
par_array_dims[i][j]	u(8)	
}		
if (**par_num_array_dims**[i] == 0) {		
par_val[i]	u(v)	v dependent on the parameter data type as specified in subclause 7.4
} else if (par_num_array_dims[i] == 1) {		
for (j = 0; j < par_array_dims[i][0]; j++) {		
par_val[i][j]	u(v)	v dependent on the parameter data type as specified in subclause 7.4
}		
} else if (par_num_array_dims[i] == 2) {		
for (j = 0; j < par_array_dims[i][0]; j++) {		
for (k = 0; k < par_array_dims[i][1]; k++) {		
par_val[i][j][k]	u(v)	v dependent on the parameter data type as specified in subclause 7.4
}		
}		
} else if (par_num_array_dims[i] == 3) {		
for (j = 0; j < par_array_dims[i][0]; j++) {		
for (k = 0; k < par_array_dims[i][1]; k++) {		
for (l = 0; l < par_array_dims[i][2]; l++) {		
par_val[i][j][k][l]	u(v)	v dependent on the parameter data type as specified in subclause 7.4
}		
}		
}		
}		
}		
}		

n_pars is the number of parameters that need to be modified for executing the algorithm referenced by algorithm_ID. Its value shall not be greater than the number of parameters defined in the algorithm.

For each parameter i being modified, the following fields are required:

par_ID[i] is the ID of one of the parameters defined in the algorithm.

par_type[i] is the data type ID of the parameter.

par_num_array_dims[i] is the number of dimensions of the parameter, 0 if it is a scalar value.

par_array_dims[i][] contains elements each specifying the size of an array dimension. It is omitted if attribute_num_array_dims = 0.

par_val[i][][][][] contains the parameter value(s). Its number of dimensions is equal to (par_array_dims + 1) and any extra dimensions can be omitted.

8.2 Inverse transformation algorithms

8.2.1 General

In this subclause the inverse transformations are specified. In the case that the input of the algorithm shall be 1 dimensional array and the input has more than one dimension, a reshaping operation is done implicitly.

8.2.2 Lempel-Ziv-Welch transform

inverse_lzw_transform is a process to inverse transform Lempel-Ziv-Welch transformed payload. Its interface is defined by 1 input parameter (Table 44), 1 input data variable (Table 45) and 1 output data variable (Table 46).

Table 44 — Input parameter(s)

ID	Name	Data Type	Num. of Dimensions	Dimensions	Default Value(s)	Remarks
0	"codebook"	(Any type, default: 1 – char)	1	(Variable, default: [128])	(A vector of ASCII characters with code from 0 to 127)	An array containing all possible original symbols

Table 45 — Input data variable(s)

ID	Name	Permissible Data Type(s)	Num. of Dimensions	Remarks
0	"in_symbols"	Any type	1	Shall be the same data type as the data type of the codebook.

Table 46 — Output data variable(s)

ID	Name	Permissible Data Type(s)	Num. of Dimensions	Remarks
0	"out_symbols"	Any type	1	Shall be the same data type as the data type of the codebook.

The process can be described as follows:

1. initialize context w with first transformed symbol from in_symbols[]

2. for each symbol in in_symbols[]:
    a. if symbol exists in codebook
        i. get entry from codebook
    b. entry is equal to w
        i. entry is equal to w concatenated with the the first element of w
    c. store entry into out_symbols[]
    d. add entry to codebook
    e. replace w with entry

The process of inverse LZW transformation is specified in Table 47.

Table 47 — Process of inverse LZW transformation

Syntax	Remarks

`inverse_lzw_transform(in_symbols, codebook) {`
`  i = 0`
`  context[] = codebook[in_symbols[0]]`
`  out_symbols = {}`
`  for (j=1; j< size(in_symbols); j++) {`
`  if (in_symbols [j] < size(codebook)) {`
`  entry = codebook[in_symbols[j]]`
`  else if (in_symbols[j] == size(codebook)) {`
`  entry = vertcat(context,context[0])`
`  }`
`  out_symbols = vertcat(out_symbols, entry)`
`  codebook[size(codebook)] = vertcat(context,entry[0])`
`  context = entry`
`  }`
`}`
	Initialize context
	Initialize output symbols
	In this step vertcat and horzcat should not make any difference.

8.2.3 Binarization transform

inverse_binarization_transform is a process to reverse the binary representation of a transformed value back to its original value. Its interface is defined by 1 input parameter (Table 48), 1 input data variable (Table 49) and 1 output data variable (Table 50).

Table 48 — Input parameter(s)

ID	Name	Data Type	Num. of Dimensions	Dimensions	Default Value(s)	Remarks
0	nbits_per_symbol	4 – uint8	0	-	-	Required and no default value.

Table 49 — Input data variable(s)

ID	Name	Permissible Data Type(s)	Num. of Dimensions	Remarks
0	“in_symbols”	bool	1	

Table 50 — Output data variable(s)

ID	Name	Permissible Data Type(s)	Num. of Dimensions	Remarks
0	“out_symbols”	Any type	1	The number of bits of the data type shall be greater of equals to nbits_per_symbol.

The process of inverse binarization transformation is specified in Table 51.

Table 51 — Process of inverse binarization transformation

Syntax	Remarks

`inverse_binarization_transform(in_symbols, nbits_per_symbol) {`
`  i = 0`
`  j = 0`
`  while(i < size(in_symbols) {`
`  for (k=0; k<nbits_per_symbol; k++) {`
`  out_symbols[j] += in_symbols[i+k] << k`
`  }`
`  j++`
`  i += k`
`  }`
`}`	

8.2.4 Sparse transform

inverse_sparse_transform is a process to change sparse matrix representation back to dense matrix representation. Its interface is defined by 1 input parameter (Table 52), 5 input data variable (Table 53) and 1 output data variable (Table 54).

Table 52 — Input parameter(s)

ID	Name	Data Type	Num. of Dimensions	Dimensions	Default Value(s)	Remarks
0	“fill_value”	Any type	0	-	0	The predefined values of the 2d array by default. Shall be the same data type as the data type of “values”.

Table 53 — Input data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	“row_indi-ces”_	All unsigned integer types - 4,6,8,10	Shall be zero or positive integers
1	“col_indices”	All unsigned integer types - 4,6,8,10	Shall be zero or positive integers
2	“values”	Any type	
3	“n_rows”	All unsigned integer types - 4,6,8,10	Shall be zero or positive integers
4	“n_cols”	All unsigned integer types - 4,6,8,10	Shall be zero or positive integers

Table 54 — Output data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	“out_sym-bols”	Any type	The same as the data type of “values”

The process of inverse sparse transformation is specified in Table 55.

Table 55 — Process of inverse sparse transformation

Syntax	Remarks

`inverse_sparse_transform(row_indices, col_indices, values`
`  n_rows, n_cols, fill_value) {`
`  for (i =0; i < n_rows; i++){`
`  for (j=0; j < n_cols; j++){`
`  out_symbols[i][j] = fill_value`
`  }`
`  }`
`  num_entries = size(values)`
`  for (i_entry=0; i_entry < num_entries; i_entry++) {`
`  i = row_indices[i_entry]`
`  j = col_indices[i_entry]`
`  k = values[i_entry]`
`  out_symbols[i][j] = k`
`  }`
`}`
	Initialize 2-dimen-sional array

8.2.5 Delta transform

inverse_delta_transform is a process to reconstruct data transformed by delta transform coding. Its interface is defined by 1 input parameter (Table 56), 1 input data variable (Table 57) and 1 output data variable (Table 58). If the input variable in_symbols is an array with number of dimensions ≥ 2, then the parameter dim specifies the dimension over which the inverse delta transform should be applied. Although the algorithm in Table 59 only supports an input array of up to two dimensions, it can be extended to support any number of dimensions.

Table 56 — Input parameter(s)

ID	Name	Data Type	Num. of Dimensions	Dimensions	Default Value(s)	Remarks
0	dim	4 – uint8	0	-	1	The value shall not be greater than the number of dimensions of the input variable in_symbols.

Table 57 — Input data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	“in_symbols”	All numeric data types – 3-12	

Table 58 — Output data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	“out_symbols”	All numeric data types – 3-12	Shall be the same type as the datatype of “in_symbols”

The process of inverse delta transformation is specified in Table 59.

Table 59 — Process of inverse delta transformation

Syntax	Remarks

inverse_delta_transform(in_symbols, dim) {
  n_dims = ndims(in_symbols)
  if (n_dims == 1) {
  out_symbols[0] = in_symbols[0]
  for (i=1; i<Size(in_symbols); i++) {
  out_symbols[i] = out_symbols[i-1] + in_symbols[i]
  }
  } else if (n_dims == 2) {
  if (dim == 1) {
  for (j=0; j<Size(in_symbols, 2); j++) {
  out_symbols[0][j] = in_symbols[0][j]
  for (i=1; i<Size(in_symbols, 1); i++) {
  out_symbols[i][j] = out_symbols[i-1][j] + in_symbols[i][j]
  }
  }
  } else if (dim == 2) {
  for (i=0; i<Size(in_symbols, 1); i++) {
  out_symbols[i][0] = in_symbols[i][0]
  for (j=1; j<Size(in_symbols, 2); j++) {
  out_symbols[i][j] = out_symbols[i][j-1] + in_symbols[i][j]
  }
  }
  }
  }
}	

8.2.6 Run-Length Encoding transform

inverse_rle_transform is a process to reconstruct data transformed by run-length encoding transform. Its interface is defined by 0 input parameter, 2 input data variables (Table 60) and 1 output data variable (Table 61).

Table 60 — Input data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	"in_symbols"	Any type	
1	"run_lengths"	All unsigned integer types - 4,6,8,10	Shall be positive integers

Table 61 — Output data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	"out_symbols"	Any type	Shall be the same type as the datatype of "in_symbols"

The process of inverse RLE transformation is specified in Table 62.

Table 62 — Process of inverse RLE transformation

Syntax	Remarks

inverse_rle_transform() {
  i = 0
  num_entries = Size(in_symbols)
  for (i_entry=0; i<size(entries); i++) {
  current_symbol = in_symbols[i_entry]
  current_run_length = run_lengths[i_entry]
  for (k=0; k<current_run_length; k++){
  out_symbols[i] = current_symbol
  i = i + 1
  }
  }
}	

8.2.7 Serialization transform

Inverse_serialization_transform is a process to reconstruct an array. Its interface is defined by 1 input parameter (Table 63), 2 input data variables (Table 64) and 1 output data variable (Table 65). Although the algorithm in Table 66 only supports conversion to a 2-dimensional array, it can be extended to support conversion to an array of any number of dimensions.

Table 63 — Input parameter(s)

ID	Name	Data Type	Num. of Dimensions	Dimensions	Default Value(s)	Remarks
1	order	2 – bool	0	-	1	In general, an order of 0 indicates a value assignment priority from high to low dimensions, whereas an order of 1 indicates a priority from low to high dimensions. For a 2-d output array, if order equals 0, row-wise assignment is used, otherwise column-wise assignment.

Table 64 — Input data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	“in_symbols”	All numeric data types – 3-12	
1	“array_dims”	All unsigned integer types - 4,6,8,10	Shall be a vector of at least two elements, with the i^{th} element specifying the size of the i^{th} dimension of the output array.

Table 65 — Output data variable(s)

ID	Name	Permissible Data Type(s)	Remarks
0	“out_symbols”	All numeric data types – 3-12	Shall be the same type as the datatype of “in_symbols”

The process of inverse serialization transformation is specified in Table 66.

Table 66 — Process of inverse serialization transformation

Syntax	Remarks

`inverse_serialization_transform(in_symbols, array_dims, order) {`
`  if (ndims(in_symbols) == 1) {`
`  if (size(array_dims) == 2) {`
`  k = 0`
`  if (order == 0) {`
`  for (i = 0; i < out_dims[0]; i++) {`
`  for (j = 0; j < out_dims[1]; j++) {`
`  out_symbols[i][j] = in_symbols[k]`
`  k++`
`  }`
`  }`
`  } else {`
`  for (j = 0; j < out_dims[1]; j++) {`
`  for (i = 0; i < out_dims[0]; i++) {`
`  out_symbols[i][j] = in_symbols[k]`
`  k++`
`  }`
`  }`
`  }`
`  }`
`  }`
`}`	

8.3 Decompression algorithms

8.3.1 General

General input parameters for all the decompression algorithms, unless otherwise specified, are listed in Table 67.

Table 67 — General input variables

ID	Name	Type	Remarks
0	input_size	u(32)	
1	input_buffer	c(input_size)	

General output parameters for all the decompression algorithms, unless otherwise specified, are listed in Table 68.

Table 68 — General output variables

ID	Name	Type	Remarks
0	output_size	u(32)	
1	output_buffer	c(output_size)	

8.3.2 Context-Adaptive Binary Arithmetic Coding

The decoding algorithm and the corresponding decoding parameters of CABAC are specified in ISO/IEC 23092-2.

8.3.3 Lempel-Ziv-Markov Chain Algorithm

The decoding algorithm and the corresponding decoding parameters of LZMA are specified in ISO/IEC 23092-3.

8.3.4 Zstandard

The configuration parameters required by the ZSTD compression algorithm are specified in Table 69 and the decompression algorithm is specified in IETF RFC 8478.

Table 69 — Configuration parameters of ZSTD

Syntax	Type
zstd_parameters() {	
use_dictionary_flag	u(1)
if(use_dictionary_flag){	
dictionary_size	u(16)
dictionary	c(dictionary_size)
}	
}	

use_dictionary_flag is set to 1 when ZSTD codec requires the use of a predefined binary dictionary. Otherwise set to 0.

dictionary_size specifies the size in Bytes of the ZSTD buffer when use_dictionary_flag is set to 1.

dictionary specifies the ZSTD buffer when use_dictionary_flag is set to 1.

8.3.5 JBIG

The decoding algorithm and the corresponding decoding parameters of JBIG are specified in ISO/IEC 11544.

8.3.6 Block Sorting Coder

The configuration parameters required by the BSC compression algorithm are specified in Table 70 and the decompression algorithm is specified in ISO/IEC 23092-2.

Table 70 — Configuration parameters of BSC

Syntax	Type
`bsc_parameters() {`	
`lzpHashSize`	u(8)
`lzpMinLen`	u(8)
`blockSorter`	u(8)
`coder`	u(8)
`features`	u(16)
`}`	

lzpHashSize specifies the hash table size if LZP enabled, 0 otherwise. If not 0, values can range from 10 to 28 inclusive.

lzpMinLenspecifies the minimum match length if LZP enabled, 0 otherwise. If not 0, values can range from 4 to 255 inclusive.

blockSorter specifies the block sorting algorithm among 0 for ST3, 1 for ST4, 2 for ST5, 3 for ST6, 4 for ST7, 5 for ST8, and 6 for BWT.

coder specifies the entropy coding algorithm, 0 for MFT and 1 for QLFC.

features specifies the set of additional features.

9 Decoding process

9.1 General

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

The input to this process is a list of data units, with one being the type of annotation access unit (see subclause 6.2). The output of this process is a list of ISO/IEC 23092-6 series output MPEG-G annotation records as specified in Clause 10.

The decoding process is specified such that all decoders that conform to this document will produce numerically identical decoded output as MPEG-G annotation records. Any decoding process that produces identical decoded output MPEG-G annotation records to those produced by the process described herein conforms to the decoding process requirements of this document.

9.2 Access Units decoding process

9.2.1 General

This subclause specifies the decoding process of a data unit of type Annotation Access Unit, as specified in subclause 6.2. The decoding process of each annotation access unit refers to the encoding parameters carried by the annotation parameter set identified by the `AT_ID` specified in subclause 6.4.1. There are two categories of decoding processes depending on the payload type:

— Descriptors as specified in subclause 9.3

— Attributes as specified in subclause 9.4

In the context of the generic descriptor payload decoding process, each decoded symbol is identified by

`decoded_symbols[descriptor_ID][descriptor_subsequence_ID][j`$_{descriptor\_ID}$`]`

where $j_{descriptor\_ID}$ is the index to read the decoded symbols. The valid values of `descriptor_ID` are specified in Clause 7. The values of descriptor_subsequence_ID are between 0 and the number of descriptor subsequences minus 1 as specified in Clause 7. When the number of descriptor subsequences is 1, the compact notation `decoded_symbols[descriptor_ID][j`$_{descriptor\_ID}$`]` is used. At the beginning of the decoding process of one AU all indexes $j_{descriptor\_ID}$ are initialized to 0.

In the context of the generic attribute payload decoding process, each decoded value (1-dimensional) or array of values (2-dimensional) is identified by

`decoded_data[attribute_ID][k`$_{attribute\_ID}$`]`

where $k_{attribute\_ID}$ is the index to read the decoded value(s). At the beginning of the decoding process of one AU all indexes $k_{attribute\_ID}$ are initialized to 0.

When all descriptors and attributes within an access unit have been decoded (assuming `attribute_contiguity == 0`), each output record is reconstructed as per Clause 10 according to the annotation table type (`AT_type`) and the attribute group class (`AG_class`) contained in `annotation_access_unit` (subclause 6.4.1).

To indicate the location of the decoded data in the overall annotation table, global row and column indexes (0-based) can be computed using the tile index(es) (`attribute_contiguity == 0`) or block index (`attribute_contiguity == 1`) in the annotation access unit header (subclause 6.4.2) together with the tile configuration referenced by `AG_class` in the annotation parameter set (subclause 6.3) referenced by `AT_ID`. The process for computing the global annotation table indexes is specified in Table 71. Note that the `count` variable is equal to the current value of $j_{descrioptor\_ID}$ for descriptors or $k_{attribute\_ID}$ for attributes (`descriptor_ID == ATTRIBUTE`) in the decoding process; `block_index` is the 0-based index of the payload block in the annotation access unit; and `decoded_vals` is the vector of decoded descriptor/attribute values in a record. The outputs of this process are `AT_row_index` and, if `two_dimensional == 1`, also `AT_col_index_from` and `AT_col_index_to`.

Table 71 — Process for computing the global annotation table indexes of an annotation Access Unit

Process	Remarks

`AT_col_index_from = 0`
`AT_col_index_to = 0`
`if (variable_size_tiles) {`
	As specified in the associated tile structure (subclause 6.3.2.2)

`  if (attribute_contiguity) {`	As specified in the associated tile configuration (subclause 6.3.2)
`    t_index = block_index` `  } else {`	
`    t_index = tile_index_1`	`tile_index_1` as specified in subclause 6.4.2
`  }`	
`  AT_row_index = start_index[t_index][0] + count`	`start_index` as specified in the associated tile structure (subclause 6.3.2.2)
`  if (two_dimensional) {`	As specified in the associated tile configuration (subclause 6.3.2)
`    AT_col_index_from = start_index[t_index][1]`	`start_index` as specified in the associated tile structure (subclause 6.3.2.2)
`    AT_col_index_to = end_index[t_index][1]`	`end_index` as specified in the associated tile structure (subclause 6.3.2.2)

Table 71 *(continued)*

Process	Remarks

`  }`
`} else {`
`  t_index_2 = 0`	

`  if (attribute_contiguity) {`	As specified in the associated tile configuration (subclause 6.3.2)
`    if (two_dimensional) {`	As specified in the associated tile configuration (subclause 6.3.2)
`      if (column_major_tile_order) {`	As specified in the associated tile configuration (subclause 6.3.2)
`        t_index_1 = block_index % n_tiles_per_col`	`n_tiles_per_col` as specified in subclause 6.4.2
`        t_index_2 = block_index / n_tiles_per_col` `      } else {`	
`        t_index_1 = block_index / n_tiles_per_row`	`n_tiles_per_row` as specified in subclause 6.4.2

`        t_index_2 = block_index % n_tiles_per_row`
`      }`
`    } else {`
`      t_index_1 = block_index`
`    }`	

`  } else {`	
`    t_index_1 = tile_index_1`	`tile_index_1` as specified in subclause 6.4.2
`    if (tile_index_2_exists) {`	
`      t_index_2 = tile_index_2`	`tile_index_2` as specified in subclause 6.4.2
`    }` `  }`	
`  AT_row_index = t_index_1 * tile_size[0] + count`	`tile_size` as specified in the associated tile structure (subclause 6.3.2.2)
`  if (two_dimensional) {`	As specified in the associated tile configuration (subclause 6.3.2)
`    AT_col_index_from = t_index_2 * tile_size[1]`	`tile_size` as specified in the associated tile structure (subclause 6.3.2.2)

`    AT_col_index_to = AT_col_index_from +`
`        Size(decoded_vals, 1) - 1`
`  }`
`}`	

9.2.2 Genomic variant access units

9.2.2.1 Variant site information

Type 1 annotation access units associated with AG_class == 1 encode variant site information.

The decoding process of one variant site record (subclause 10.1) 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:

— Compute the variantIndex variable as AT_row_index specified in subclause 9.2.1;

— Compute the sequenceID variable as specified in subclause 9.3.2.1;

— Compute the startPos and endPos variables as specified in subclause 9.3.2.2;

— Compute the strand variable as specified in subclause 9.3.2.3;

— Compute the nameLen, name, descriptionLen and description variables as specified in subclause 9.3.2.4;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the depth, seqQual and mapQual variables as specified in subclause 9.3.3.1;

— Compute the mapNumQuality0 variable as specified in subclause 9.3.3.2;

— Compute the referenceLen and reference variables as specified in subclause 9.3.3.3;

— Compute the alternLen and altern variables as specified in subclause 9.3.3.4;

— Compute the filtersLen and filter variables as specified in subclause 9.3.3.5;

— Compute the info_count variable as n_attributes specified in subclause 6.3.1;

— Compute the info_tag_len[i], info_tag[i] and info_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the info_array_len[i] variable as equal to 1 if attribute_num_array_dims == 0 or equal to attribute_array_dims[0] if attribute_num_array_dims == 1, with all variables contained in the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the info_value[i][] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6;

9.2.2.2 Variant genotype information

Type 1 annotation access units associated with AG_class == 0 encode variant genotype information.

The decoding process of one variant genotype record (subclause 10.2) 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:

— Compute the variantIndex variable as AT_row_index specified in subclause 9.2.1;

— Compute the sampleIndexFrom and sampleIndexTo variables as AT_col_index_from and AT_col_to specified in subclause 9.2.1;

— Compute genotypePresent as equal to 1 if there exists a descriptor configuration (subclause 6.3.4) with descriptor_ID == GENOTYPE or 15 in the associated annotation encoding parameters. Otherwise, it is equal to 0;

— Compute likelihoodPresent as equal to 1 if there exists a descriptor configuration (subclause 6.3.4) with descriptor_ID == LIKELIHOOD or 16 in the associated annotation encoding parameters. Otherwise, it is equal to 0;

— Compute the num_alleles_per_sample , alleles and phasing variable as specified in subclause 9.3.5;

— Compute the n_likelihoods and likelihoods variables as specified in subclause 9.3.6;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the format_count variable as n_attributes specified in subclause 6.3.1;

— Compute the format_len[i], format[i] and format_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the format_array_len[i] variable as equal to 1 if attribute_num_array_dims == 0 or equal to attribute_array_dims[0] if attribute_num_array_dims == 1, with all variables contained in the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the format_value[i][j][k] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6; j is the sample index; and k < format_array_len[i].

9.2.2.3 Sample information

Type 1 annotation access units associated with AG_class == 2 encode sample information.

The decoding process of one sample record (subclause 10.3) 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:

— Compute the sampleIndex variable as AT_row_index_from specified in subclause 9.2.1;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the sample_attr_count variable as n_attributes specified in subclause 6.3.1;

— Compute the sample_attr_len[i], sample_attr[i] and sample_attr_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the sample_attr_array_len[i] variable as equal to 1 if attribute_num_array_dims == 0 or equal to attribute_array_dims[0] if attribute_num_array_dims == 1, with all variables associated with the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the sample_attr_value[i][] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6.

9.2.3 Functional annotation Access Units

9.2.3.1 Functional annotation

Type 2 annotation access units associated with AG_class == 0 encode functional annotation information.

The decoding process of one functional annotation record (subclause 10.4) within a binary decoded access unit of type 2, which shall be repeated for all the records within the same access unit, is as follows:

— Compute the annotationIndex variable as AT_row_index_from specified in subclause 9.2.1;

— Compute the sequenceID variable as specified in subclause 9.3.2.1;

— Compute the startPos and endPos variables as specified in subclause 9.3.2.2;

— Compute the featureNameLen and featureName variables as specified in 9.3.4.1;

— Compute the featureID variable as specified in 9.3.4.2;

— Compute the ontologyNameCount, ontologyNameLen and ontologyName variables as specified in 9.3.4.3;

— Compute the ontologyID variable as specified in 9.3.4.4;

— Compute the strand variable as specified in subclause 9.3.2.3;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the attr_count variable as n_attributes specified in subclause 6.3.1;

— Compute the attr_tag_len[i], attr_tag[i] and attr_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the attr_value[i] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6.

9.2.3.2 Track property information

Type 2 annotation access units associated with AG_class == 5 encode track property information.

The decoding process of one track property record (subclause 10.5) within a binary decoded access unit of type 5, which shall be repeated for all the records within the same access unit, is as follows:

— Compute the trackIndex variable as AT_row_index_from specified in subclause 9.2.1;

— Compute the trackType variable as AT_subtype in `annotation_access_unit` specified in subclause 6.4.1

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the track_property_count variable as n_attributes specified in subclause 6.3.1;

— Compute the track_property_len[i], track_property[i] and track_property_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the track_property_array_len[i] variable as equal to 1 if attribute_num_array_dims == 0 or equal to attribute_array_dims[0] if attribute_num_array_dims == 1, with all variables associated with the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the track_property_value[i][] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6.

9.2.4 Gene expression Access Units

9.2.4.1 Expression information

Type 3 annotation access units associated with AG_class == 0 encode expression information.

The decoding process of one expression record (subclause 10.7) within a binary decoded access unit of type 3, which shall be repeated for all the records within the same access unit, is as follows:

— Compute the featureIndex variable as AT_row_index specified in subclause 9.2.1;

— Compute the sampleIndexFrom and sampleIndexTo variables as AT_col_index_from and AT_col_to specified in subclause 9.2.1;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the expr_attr_count variable as n_attributes specified in subclause 6.3.1;

— Compute the expr_attr_len[i], expr_attr[i] and expr_attr_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the expr_attr_array_len[i] variable as equal to 1 if attribute_num_array_dims == 0 or equal to attribute_array_dims[0] if attribute_num_array_dims == 1, with all variables contained in the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the expr_attr_value[i][j][k] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6; j is the sample index; and k < expr_attr_array_len[i].

9.2.4.2 Feature information

Type 3 annotation access units associated with AG_class == 1 encode feature information.

The decoding process of one feature record (subclause 10.8) within a binary decoded access unit of type 3, which shall be repeated for all the records within the same access unit, is as follows:

— Compute the featureIndex variable as AT_row_index specified in subclause 9.2.1;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the feature_attr_count variable as n_attributes specified in subclause 6.3.1;

— Compute the feature_attr_len[i], feature_attr[i] and feature_attr_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the feature_attr_array_len[i] variable as equal to 1 if attribute_num_array_dims == 0 or equal to attribute_array_dims[0] if attribute_num_array_dims == 1, with all variables contained in the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the feature_attr_value[i][] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6.

9.2.4.3 Sample information

Type 3 annotation access units associated with AG_class == 2 encode sample information.

The decoding process of one sample record (subclause 10.3) within a binary decoded access unit of type 3, which shall be repeated for all the records within the same access unit, is specified in subclause 9.2.2.3.

9.2.5 Position-to-position contact intensity Access Units

Type 4 annotation access units encode contact matrix information.

The decoding process of one contact record (subclause 10.9) within a binary decoded access unit of type 4, which shall be repeated for all the records within the same access unit, is as follows:

— Compute the seqID, n_values, value_names_len and value_names variables as specified in subclause 6.3.4.4;

— Compute the startPos, endPos, counts and values variables as specified in subclause 9.3.7.1;

— Compute the linkedRecord, linkName and linkID variables as specified in subclause 9.3.2.5

9.2.6 Genome browser track Access Units

9.2.6.1 Track data

Type 5 annotation access units associated with AG_class == 0 encode genome browser track information.

The decoding process of one track data record (subclause 10.6) within a binary decoded access unit of type 5, which shall be repeated for all the records within the same access unit, is as follows:

— Compute the trackDataIndex variable as AT_row_index_from specified in subclause 9.2.1;

— Compute the sequenceID variables as specified in subclause 9.3.2.1;

— Compute the startPos and endPos variables as specified in subclause 9.3.2.2;

— Compute the name and description variables as specified in subclause 9.3.2.4;

— Compute the strand variable as specified in subclause 9.3.2.3;

— Compute the linkedRecord, linkNameLen, linkName and linkID variables as specified in subclause 9.3.2.5;

— Compute the attr_count variable as n_attributes specified in subclause 6.3.1;

— Compute the attr_tag_len[i], attr_tag[i] and attr_type[i] variables as attribute_name_len, attribute_name and attribute_type of the i^{th} attribute parameter set specified in subclause 6.3.6;

— Compute the attr_value[i] variable as attr_vals[attribute_ID] specified in subclause 9.4, where attribute_ID is in the i^{th} attribute parameter set specified in subclause 6.3.6.

9.2.6.2 Track property information

Type 5 annotation access units associated with AG_class == 5 encode track property information.

The decoding process of one track property record (subclause 10.5) within a binary decoded access unit of type 5, which shall be repeated for all the records within the same access unit, is specified in subclause 9.2.3.2.

9.3 Descriptors decoding process

9.3.1 General

This subclause specifies the decoding process of the descriptors listed in subclause 7.2.

The input to the processes described in the following subclauses are decoded genomic descriptors contained in the `decoded_symbols` data structure produced as result of the payload decoding process specified in subclause 9.5.1.

In the context of the decoding process of genomic descriptors described in subclause 9.3. each decoded symbol is identified by `decoded_symbols[descriptor_ID][`$j_{\texttt{descriptor\_ID}}$`]` where the index $j_{\texttt{descriptor\_ID}}$ is used to read each decoded symbol used in the decoding process described for each descriptor. The valid values of `descriptor_ID` are specified in Clause 7.

The output of this process is a sequence of output records as specified in Clause 10.

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 Annotation 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.

1. Decode the sequenceID variable as specified in subclause 9.3.2.1
2. Decode the startPos and endPos variables as specified in subclause 9.3.2.2
3. Decode the strand variable as specified in subclause 9.3.2.3
4. Decode the name and description variables as specified in subclause 9.3.2.4

5. Decode the linkName and linkID as specified in subclause 9.3.2.5.

9.3.2 Common descriptors

9.3.2.1 sequenceID

The input to this process (see Table 72) are:

— the array `decoded_symbols[descriptor_ID]` specified in subclause 9.5.1 when `descriptor_ID` is equal to 1 for `sequenceID` (sequence identifier).

— the current value j_1

The output of this process is the variable sequenceID.

Table 72 — Decoding process of sequenceID

Process	Remarks
`sequenceID = decoded_symbols[1][`j_1`]` j_1`++`	

9.3.2.2 startPos, endPos and previousStartPos

The input to this process (see Table 73) is the array `decoded_symbols[descriptor_ID]` array specified in subclause 9.5.1 when `descriptor_ID` is equal to 2 (for `startPos`) and 3 (for `endPos`) and the current values of j_2 and j_3 respectively; the variable `previousStartPos` produced by the previous iteration of this same process.

The outputs are the variables `startPos`, `endPos` and `previousStartPos`.

Table 73 — Decoding process of startPos, endPos and previousStartPos

Process	Remarks

`if(`j_2 `> 0) {`
`  startPos =`
`  previousstartPos + decoded_symbols[2][`j_2`]`
`}`
`else{`
`  startPos =`
`  AAU_start_position + decoded_symbols[2][`j_2`]`
`}`
`previousStartPos = startPos`
j_2`++`
`endPos = startPos + decoded_symbols[3][`j_3`]`
j_3`++`
	AAU_start_position is specified in subclause 6.4.2

9.3.2.3 strand

The input to this process (see Table 74) is the `decoded_symbols[descriptor_ID]` array specified in subclause 9.5.1 when `descriptor_ID` is equal to 4 and the current values of j_4.

The output of this process is the variable `strand`.

The strand syntax element carries information related to the strand of the genomic annotation.

Table 74 — Decoding process of strand

Process	Remarks
`strand = decoded_symbols[4][j4] && 0x03` `j4++`	Only 2 bits are relevant

9.3.2.4 name and description

The inputs to this process (see Table 75) are:

— the array `decoded_symbols[descriptor_ID]` as specified in subclause 9.5.1 when `descriptor_ID` is equal to 5 for `name` and 6 for `description`.

— the current value of j_5 and j_6

The outputs of this process are the variables `name`, `nameLen`, `description` and `descriptionLen`. Both `name` and `description` are arrays of ASCII characters.

Table 75 — Decoding process of name and description

Process	Remarks

`name = {}`
`nameLen = 0`
`while(decoded_symbols[5][j5] != '\0') {`
`  name = strcat(name, decoded_symbols[5][j5])`
`  j5++`
`  nameLen++`
`}`
`j5++`
	Empty string.

`description = {}`
`descriptionLen = 0`
`while(decoded_symbols[6][j6] != '\0') {`
`  description = strcat(description,`
`  decoded_symbols[6][j6])`
`  j6++`
`  descriptionLen ++`
`}`
`j6++`
	Empty string.

9.3.2.5 linkName and linkID

The inputs to this process (see Table 76) are:

— the array `decoded_symbols[descriptor_ID]` as specified in subclause 9.5.1 when `descriptor_ID` is equal to 7 for `linkName` and 8 for `linkID`; the current value of j_7 and j_8.

The outputs of this process are the variables linkName, linkNameLen and linkID.

Table 76 — Decoding process of linkName and linkID

Process	Remarks

decodeLinkInformation() {
  linkName = {}
  linkNameLen = 0
  linkID = decoded_symbols[8][j_8]
  if (linkID == 255){
  linkedRecord = 0
  } else {
  linkedRecord = 1
  }
  if (linkedRecord == 1){
  while(decoded_symbols[7][j_7] != '\0') {
  linkName = strcat(decodedName,
  decoded_symbols[7][j_7])
  j_7++
  linkNameLen++
  }
  }
  j_7++
  j_8++
}
	Empty string.

9.3.3 Variant site information descriptors

9.3.3.1 depth, qual, seqQuality and mapQuality

The inputs to this process (see Table 77) are:

— the array decoded_symbols[descriptor_ID] specified in subclause 9.5.1 when descriptor_ID is equal to 9 for the descriptor depth (read depth for a variant), 10 for the descriptor seqQuality (Root Mean Square (RMS) sequencing quality of the bases supporting the variant) and 11 for the descriptor mapQuality (RMS mapping quality).

— the current values j_9, j_0, j_{10} and j_{11}

The outputs of this process are the variables depth, seqQual and mapQual.

Table 77 — Decoding process of depth, seqQuality and mapQuality

Process	Remarks
depth = decoded_symbols[9][j_9]	IEEE 754 32-bit floating point number
seqQual = decoded_symbols[10][j_{10}]	IEEE 754 32-bit floating point number
mapQual = decoded_symbols[11][j_{11}]	IEEE 754 32-bit floating point number
j_9++	
j_0++	
j_{10}++	
j_{11}++	

9.3.3.2 mapNumQuality0

The input to this process (see Table 78) are:

— the array `decoded_symbols[descriptor_ID]` specified in subclause 9.5.1 when `descriptor_ID` is equal to 12.

— the current value j_{12}

The output of this process is the variable `mapNumQuality0` containing the number of reads with MAPQ == 0.

Table 78 — Decoding process of mapNumQuality0

Process	Remarks
`mapNumQuality0 = decoded_symbols[12][`j_{12}`]` j_{12}++	

9.3.3.3 reference

The inputs to this process (see Table 79) are:

— the array `decoded_symbols[descriptor_ID]` as specified in subclause 9.5.1 when `descriptor_ID` is equal to 13.

— the current value of j_{13}.

— the array $S_{alphabet\_ID}[]$ as specified in ISO/IEC 23092-2, for the value of `alphabet_ID` specified in subclause 6.3.1.

The output of this process is the string `reference` and the value `referenceLen`.

Table 79 — Decoding process of reference

Process	Remarks

`reference = {}`
`referenceLen = 0`
`while(decoded_symbols[13][`j_{13}`] != 7) {`
 	
`  reference = strcat(reference,`
$S_{alphabet\_ID}$`[decoded_symbols[13][`j_{13}`]])`
j_{13}++
`  referenceLen++`
`}`
j_{13}++
	Empty string
	
	Use the value 7 to signal the end of the current reference string

9.3.3.4 altern

The inputs to this process (see Table 80) are:

— the array `decoded_symbols[descriptor_ID]` as specified in subclause 9.5.1 when `descriptor_ID` is equal to 14.

— the current value of j_{14}

— the array $S_{alphabet\_ID}[]$ as specified in ISO/IEC 23092-2:2021, for the value of `alphabet_ID` specified in subclause 6.3.1.

The output of this process is an array of strings `altern`, an array of integers `alternLen` and the variable `numAltern`, which represents the number of decoded altern strings.

Table 80 — Decoding process of Altern

Process	Remarks
i = 0	
while(decoded_symbols[14][j_{14}] != 7) {	Use the value 7 to signal the end of the current altern array
altern[i] = {}	Empty string
alternLen[i] = 0	
while(decoded_symbols[14][j_{14}] < 6) {	Use the value 6 to signal the next element in the altern array
altern[i] = strcat(altern[i], $S_{alphabet\_ID}$[decoded_symbols[14][j_{14}]])	
j_{14}++	
alternLen[i]++	
}	
j_{14}++	
i++	
}	
numAltern = i	

9.3.3.5 filters

The input to this process (see Table 81) is:

— the array `decoded_symbols[descriptor_ID]` array specified in subclause 9.5.1 when `descriptor_ID` is equal to 17

— the current values of j_{17}

— the lists of `filter_ID` and `filter_ID_len`, and the variable `n_filter`

The output of this process is the string `filters` and its length in characters `filtersLen`.

Table 81 — Decoding process of filter

Process	Remarks
while (decoded_symbols[17][j_{17}] != 0xFF){	
if(decoded_symbols[17][j_{17}] == 0) {	
filters = "PASS"	
} else if(decoded_symbols[17][j_{17}] == 1) {	
filters = "MISSING"	
} else {	
filters = strcat(filters, filter_ID[decoded_symbols[17][j_{17}]] - 2)	
filters = strcat(filters, ";")	
}	
filtersLen = strlen(filters)	the function strlen returns the number of characters in the string decodedFilters
j_{17}++	

Table 81 *(continued)*

Process	Remarks
}	

9.3.4 Functional annotation descriptors

9.3.4.1 featureName

The input to this process (see Table 82) is:

— the array `decoded_symbols[descriptor_ID]` array specified in subclause 9.5.1 when `descriptor_ID` is equal to 18

— the current values of j_{18}

— the list of `features_name` and `features_name_len`

The output of this process is the string `featureName` and its length in characters `featureNameLen`.

Table 82 — Decoding process of featureName

Process	Remarks

`if(n_features_names == 0){`
 	
`  featureName = "MISSING"`
`  featureNameLen = 7`
`} else if(decoded_symbols[18][`j_{18}`] == 0xFFFFFF)`
`  featureName = "MISSING"`
`  featureNameLen = 7`
`  `j_{18}`++`
`} else {`
`  i = decoded_symbols[18][`j_{18}`]`
`  featureName = features_name[i]`
`  featureNameLen = features_name_len[i]`
`  `j_{18}`++`
`}`
	`n_features_names` as specified in subclause 6.3.1.

9.3.4.2 featureID

The inputs to this process (see Table 83) are:

— the array `decoded_symbols[descriptor_ID]` specified in subclause 9.5.1 when `descriptor_ID` is equal to 19

— the current value j_{19}

The output of the process is the variable `featureID` containing the feature identifier where 0xFFFFFF is used as undefined value.

Table 83 — Decoding process of featureID

Process	Remarks

if(Size(decoded_symbols[19]) == 0){
  featureID = 0xFFFFFF
} else {
  featureID = decoded_symbols[19][j_{19}]
  j_{19}++
}
	0xFFFFFF used for undefined value
	
	0xFFFFFF used for undefined value

9.3.4.3 ontologyName

The input to this process (see Table 84) is:

— the array `decoded_symbols[descriptor_ID]` array specified in subclause 9.5.1 when `descriptor_ID` is equal to 20

— the current values of j_{20}

— the list of `ontology_term_name` and `ontology_term_name_len` as specified in subclause 6.3.3.

The output of this process is the array of strings `ontologyName`, the array `ontologyNameLen` and the variable `ontologyNameCount`. The `ontologyNameCount` variable shall be equal to `ontologyIDCount` specified in subclause 9.3.4.4.

Table 84 — Decoding process of ontologyNames

Process	Remarks

i = 0
while (decoded_symbols[20][j_{20}] != 0xFFFFFF){
  if(decoded_symbols[20][j_{20}] == 0xFFFFFF){
  ontologyName[i] = “”
 	
  ontologyNameLen[i] = 0
  } else {
  j = decoded_symbols[20][j_{20}]
  ontologyName[i] = ontology_term_name[j]
  ontologyNameLen[i] = ontology_term_name_len[j]
  }
  i++
  j_{20}++
}
j_{20}++
ontologyNameCount = i
	Initialize empty string

9.3.4.4 ontologyID

The inputs to this process (see Table 85) are:

— the array `decoded_symbols[descriptor_ID]` as specified in subclause 9.5.1 when `descriptor_ID` is equal to 21.

— the current value of j_{21}

— the variable `numOntologyID` which represents the number of values to be decoded

The output of this process is an array of strings `ontologyID` and the variable `ontologyIDCount`. The `ontologyIDCount` variable shall be equal to `ontologyNameCount` specified in subclause 9.3.4.3.

Table 85 — Decoding process of ontologyID

Process	Remarks

i = 0
while (decoded_symbols[21][j_{21}] != 0xFFFFFE){
  ontologyID[i] = decoded_symbols[21][j_{21}]
  j_{21}++
  i++
}
j_{21}++
ontologyIDCount = i
	0xFFFFFF used for undefined value

9.3.5 Genotype descriptor

9.3.5.1 General

This subclause describes the decoding process of the descriptor payloads when the `descriptor_ID` is equal to `GENOTYPE` or `15`.

The input to this process (see Table 86) is `gt_matrix` from process specified in subclause 9.3.5.2 when `descriptor_ID` is equal to 15 and the current value of j_{15}.

The output of this process are the variables `alleles` and `phasing`.

Table 86 — Decoding process of genotype

Process	Remarks

n_samples = Size(allele_3d_array[][][], 2)
 	
num_alleles_per_sample = Size(allele_3d_array[][][], 3)
for(i=0; i<n_samples;i++){
  for (j=0; j<num_alleles_per_sample; j++){
  alleles[i][j] = allele_3d_array[j_{15}][i][j]
 	
  }
  for (j=0; j<num_alleles_per_sample-1; j++){
  phasing[i][j] = phases_3d_arr [j_{15}][i][j]
 	
  }
}
j_{15}++
	From process specified in subclause 9.3.5.2
	
	From process specified in subclause 9.3.5.2
	
	From process specified in subclause 9.3.5.2

9.3.5.2 Decode genotype payload

The inputs of this process are (see Table 87):

— the genotype payload structure as specified in subclause 6.4.4.3.2.

— the genotype parameters structure as specified in subclause 6.3.4.2.

The output of this process is the 3-dimensional array `allele_3d_array[][][]` and `phases_3d_arr[][][]`.

Table 87 — Decoding process of genotype payload

Process	Remarks
`decode_genotype_payload(genotype_payload, genotype_parameters) {`	
`  for (i=0; i<num_variants_payloads; i++) {`	
`    bin_mats[i][][] = decode_bin_mat(variants_codec_ID[i],`	As specified in subclause 9.3.5.3
`                                     sort_variants_rows_flag[i],`	
`                                     sort_variants_cols_flag[i],`	
`                                     transpose_variants_mat_flag[i],`	
`                                     variants_payload_size[i],`	
`                                     variants_payload[i],`	
`                                     variants_row_IDs_payload_size[i],`	
`                                     variants_row_IDs_payload[i],`	
`                                     variants_col_IDs_payload_size[i],`	
`                                     variants_col_IDs_payload[i])`	
`  }`	
`  if (binarization_ID == BIT_PLANE) {`	
`    allele_mat[][][] = debinarize_bit_plane(bin_mats[][][],`	As specified in subclause 9.3.5.7.
`                                     num_bit_plane,`	
`                                     concat_axis)`	
`  } else if (binarization_ID == ROW_SPLIT) {`	
`    amax[] = variant_amax_payload()`	As specified in 6.4.4.3.2.4
`    allele_mat[][][] = debinarize_row_split(bin_mats[0][][],`	As specified in subclause 9.3.5.9.
`                                     amax[])`	
`  }`	
`  allele_mat[][][] = revert_adaptive_max_val(allele_mat[][][],`	As specified in subclause 9.3.5.10.
`                                     no_reference_flag,`	
`                                     not_available_flag)`	
`  if (encode_phase_data) {`	
`    phases_mat[][][] = decode_bin_mat(phases_payload,`	As specified in subclause 9.3.5.3
`                                     phases_row_ids_payload,`	
`                                     phases_col_ids_payload,`	
`                                     phases_codec_ID)`	
`  } else {`	
`    phases_mat[0][0][0] = phases_value`	
`  }`	
`  {allele_3d_array[][][],phases_3d_arr[][][]} = reconstruct_3d_arrays(`	As specified in subclause 9.3.5.11.
`                                     allele_matrix[][][],`	
`                                     phasing_mat[][][],`	
`                                     max_ploidy)`	
`}`	

variants_payload[i] is a binary matrix payload for variants as specified in subclause 6.4.4.3.2.

variants_row_IDs_payload[i] is a row column ids payload for variants as specified in subclause 6.4.4.3.2.3 if sort_variants_rows_flag[i] is True, otherwise it is empty array.

variants_col_IDs_payload[i] is a row column ids payload for variants as specified in subclause 6.4.4.3.2.3 if sort_variants_cols_flag[i] is True, otherwise it is empty array.

9.3.5.3 Decode binary matrix payload

The inputs to this process are (see Table 88):

— the codec identifier codec_ID

— the flag sort_variants_row_flag that signals whether the rows of binary matrix is sorted

— the flag sort_variants_col_flag that signals whether the columns of binary matrix is sorted

— the flag transpose_flag that signals whether the processed matrix is transposed.

— the binary matrix payload bin_mat_payload as specified in subclause 6.4.4.3.2.2.

— the row column ids payload row_ids_payload as specified in subclause 6.4.4.3.2.3.

— the column ids payload col_ids_payload as specified in subclause 6.4.4.3.2.3.

— the amax payload amax_payload as specified in subclause 6.4.4.3.2.4.

The output of this process is an array bin_mat[][].

Table 88 — Decoding process of bin mat

Process	Remarks
`decode_bin_mat(codec_ID, sort_variants_row_flag,`	
`  sort_variants_col_flag, transpose_flag,`	
`  bin_mat_payload_size, bin_mat_payload,`	
`  row_IDs_payload_size, row_IDs_payload,`	
`  col_IDs_payload_size, col_IDs_payload,`	
`  amax_payload) {`	
`  if (codec_ID == JBIG) {`	
`  bin_mat[][] = decode_jbig(bin_mat_payload_size,` `  bin_mat_payload)`	2-dimensional Boolean array
`  } else if(codec_ID == CABAC) {`	
`  bin_mat[][] = decode_cabac_matrix(bin_mat_payload)`	2-dimensional Boolean array
`  }`	
`  {nrows, ncols} = Size(bin_mat[][])`	
`  if (sort_variants_row_flag) {`	
`  row_IDs[] = decode_row_col_IDs(nrows, row_IDs_payload)`	As specified in 6.4.4.3.2.3
`  bin_mat[][] = sort_bin_mat_rows(bin_mat[][], row_IDs[])`	As specified in 9.3.5.5
`  }`	
`  if (sort_variants_col_flag) {`	
`  col_IDs[] = decode_row_col_IDs(ncols, col_IDs_payload)`	As specified in 6.4.4.3.2.3
`  bin_mat[][] = sort_bin_mat_cols(bin_mat[][], col_IDs[])`	As specified in 9.3.5.6
`  }`	
`  if (transpose_flag) {`	

Table 88 *(continued)*

Process	Remarks
bin_mat[][] = transpose_mat(bin_mat[][]) } }	

decode_jbig is a decoding process compliant with ISO/IEC 11544.

decode_cabac_matrix is a process described in subclause 9.3.5.4.

9.3.5.4 Decode cabac matrix

The inputs to this process are (see Table 89):

— the size in bytes of `bin_mat_payload` as specified in subclause 6.4.4.3.2.2

— the `cabac_payload` as specified in subclause 6.4.4.3.2.2.

The output of this process is the 2-dimensional array `bin_mat[][]`.

Table 89 — Decoding process of cabac matrix

Process	Remarks

decode_cabac_matrix (bin_mat_payload_size, bin_mat_payload) {
decoded_symbols = decode_descriptor_subsequence(
descriptor_ID,
0,
num_encoded_symbols,
block_payload_size)
for (i=0; i<nrows; i++) {
for (j=0; j<ncols; j++) {
bin_mat[i][j] = decoded_symbols[j+i*ncols]
}
}
}
	nrows as specified in 6.4.4.3.2.2
	ncols as specified in 6.4.4.3.2.2

decode_descriptor_subsequence is a process to decode current block, as specified in ISO/IEC 23092-2. To decode the bitstream the following ISO/IEC 23092-2 syntax element values are assumed: transform_ID_subseq is 0, transform_ID_subsym is 0, binarization_ID is 0, coding_sybsym_size is 1, mpegg_symbol_size is 1, coding_order is 2, bypass_flag is 0, adaptive_mode_flag is 1 and num_contexts is 2.

9.3.5.5 Sort rows of binary matrix

The inputs to this process (see Table 90) are:

— the array bin_mat[][]

— the array row_IDs[]

The output of this process is an array sorted_bin_mat[][].

This process sorts the rows or 1st dimension of bin_mat[][].

Table 90 — Sorting process of bin mat rows

Process	Remarks

sort_bin_mat_rows(bin_mat[][], row_IDs[]) {
{nrows, ncols} = Size(bin_mat[][])
for (i=0; i< nrows; i++) {
for (j=0; j<ncols; j++){
sorted_bin_mat[i][j] = bin_mat[row_IDs[i]][j]
}
}
}	

9.3.5.6 Sort columns of binary matrix

The inputs to this process (see Table 91) are:

— the array bin_mat[][]

— the array col_IDs[]

The output of this process is an array sorted_bin_mat[][].

This process sorts the columns or 2nd dimension of bin_mat[][]

Table 91 — Sorting process of bin mat cols

Process	Type

sort_bin_mat_cols(bin_mat[][], col_IDs[]){
{nrows,ncols} = Size(bin_mat[][])
for (j=0; j< ncols; j++) {
for (i=0; i<nrows; i++) {
sorted_bin_mat[i][j] = bin_mat[i][col_IDs[j]]
}
}
}	

9.3.5.7 Debinarize binary matrices using bit plane

The input to this process (see Table 92) are:

— the array bin_mats[][][]

— the value nmats describes the original number of bit planes before concatenation.

— the value axis describes the direction bit planes were concatenated during encoding.

The output of this process is an array allele_mat[][].

This process can be described as follows:

1. Split binary matrix into binary matrices if axis is less than 2
2. Concatenate binary matrices bin_mats[][][] into an array named allele_mat[][]. The index of the 1st dimension of bin_mats[][][] describes the original bit position in the allele matrix.

Table 92 — Binarization process of bit plane

Process	Remarks

debinarize_bit_plane(bin_mats[][][],nmats, axis) {
  if (axis < 2){
  bin_mats[][][] = split_matrix(
  bin_mats[0][][], nmats, axis)
  }
  {bitlen, nrows, ncols} = Size(bin_mats[][][])
  for(i=0; i<nrows; i++) {
  for(j=0; j<ncols; j++) {
  allele_mat[i][j] = bin_mats[k][i][j]
  for (k=1; k<bitlen; k++){
  allele_mat[i][j] += bin_mats[k][i][j] << k
  }
  }
  }
}
	As described in subclause 9.3.5.8

9.3.5.8 Split matrix

The inputs to this process (see Table 93) are:

— the array mat[][]

— the number of matrices nmats

— the direction which the matrix is split axis

The output of this process is an array mats[][][].

This process can be described as follows:

1. Initialize an array of 3-dimensional array mats[][][]
2. Compute the number of rows (2nd dimension) of mats[][][] if axis equals 0 or columns (3rd dimension) of mats[][][] if axis equals 1 .
3. Copy value from mat[][] to mats[][][].

Table 93 — Process of splitting a matrix

Process	Remarks

split_matrix(mat[][], nmats, axis) {
  {nrows, ncols} = Size(mat[][])
  if (axis == 0){
  nelems = nrows / nmats
  for (i_mat=0; i_mat<nmats; i_mat++) {
  for (i=0; i<nelems;i++) {
  for (j=0; j<ncols; j++){
  mats[i_mat][i][j] = mat[i+i_mat*nelems][j]
  }
	Split in row direction
	
	Copy value

Table 93 *(continued)*

Process	Remarks

}
}
} else if (axis == 1) {
nelems = ncols / nmats
for (i_mat=0; i_mat<nmats; i_mat++) {
for (i=0; i<nrows;i++) {
for (j=0; j<nelems; j++){
mats[i_mat][i][j] = mat[i][j+i_mat*nelems]
}
}
}
}
}
	Split in column direction
	
	Copy value

9.3.5.9 Debinarize binary matrices using row split

The inputs to this process (see Table 94) are:

— the array bin_mat[][]

— the array amax[]

The output of this process is an array allele_mat[][].

Table 94 — Debinarization process of the row split method

Process	Remarks

debinarize_row_split(bin_mat[][], amax[]) {
nrows = Size(amax[])
ncols = Size(bin_mat[][], 2)
for (i=0; i<nrows; i++){
for (j=0; j<ncols; j++){
allele_mat[i][k] = 0
}
}
l = 0
for (i=0; i<nrows; i++) {
for (j=0; j<amax[i]; j++) {
for (k=0; k< ncols; k++){
allele_mat[i][k] += (j+1) * bin_mat[l][k]
}
l++
}
}
}	

9.3.5.10 Revert adaptive max value

The inputs to this process (see Table 95) are:

— the array allele_mat[][]

— the flags no_reference_flag and not_available_flag

The output of this process is an array allele_mat[][].

The process can be described as follows:

1. Find the maximum value of the array allele_mat[][] and store the value as max_val
2. If not_available_flag is 1, replace all the values in allele_mat[][] that equal max_val with -2, then max_val is decreased by one
3. If no_reference_flag is 1, replace with -1 all the values in allele_mat[][] that equal max_val.

Table 95 — Reversion process of of maximum value

Process	Remarks

`revert_adaptive_max_val(allele_mat[][],`
`  no_reference_flag,`
`  not_available_flag) {`
`  max_val = 0`
`  {nrows, ncols} = Size(allele_mat[][])`
`  for (i=0; i<nrows; i++) {`
`  for (j=0; j<ncols; j++) {`
`  if (max_val < allele_mat[i][j]) {`
`  max_val = allele_mat[i][j]`
`  }`
`  }`
`  }`
	Find maximum value of allele matrix

`  if (not_available_flag) {`
`  for (i=0; i<nrows; i++) {`
`  for (j=0; j<ncols; j++) {`
`  if (allele_mat[i][j] == max_val) {`
`  allele_mat[i][j] = -2`
`  }`
`  }`
`  }`
`  max_val -= 1`
`  }`
	Replace all maximum value in allele matrix with -2

`  if (no_reference_flag) {`
`  for (i=0; i<nrows; i++) {`
`  for (j=0; j<ncols; j++) {`
`  if (allele_mat[i][j] == max_val) {`
`  allele_mat[i][j] = -1`
`  }`
`  }`
`  }`
	Replace all maximum value in allele matrix with -1

Table 95 *(continued)*

Process	Remarks
} }	

9.3.5.11 Reconstruct 3d arrays

The inputs to this process (see Table 96) are:

— the array allele_mat[][].

— the array phasing_mat[][].

— the value max_ploidy as described in subclause 6.3.4.2

The outputs of this process are arrays allele_3d_array[][][] and phases_3d_array[][][].

Table 96 — Reconstruction process of 3-dimensional array

Process	Remarks

reconstruct_3d_arrays(allele_mat[][],
phasing_mat[][],
max_ploidy) {
allele_3d_array[][][] = split_matrix(allele_mat[][][],
max_ploidy,
2)
{ndepths, nrows, ncols} = Size(allele_3d_array[][][])
[p_nrows, p_ncols] = Size(phasing_mat[][])
p_ndepths = ndepths-1
if (p_ndepths > 0){
if (p_nrows == 1 && p_ncols == 1) {
for (k=0; k<p_ndepths; k++) {
for (i=0; i<nrows; i++) {
for (j=0; j<ncols; j++) {
phasing_3d_array[k][i][j] = phasing_mat[0][0]
}
}
}
} else {
phasing_3d_array[][][] = split_matrix(phasing_mat[][],
p_ndepths,
2)
}
} else {
phasing_3d_array[0][0][0] = 0
}
}
	As described in subclause 9.3.5.8.
	ndepths equals to max_ploidy
	Initialize phases_3d_array with value phases_mat[0][0]

9.3.5.12 Reconstruct genotype matrix

The input to this process (see Table 97) are:

— The array allele_3d_array[][][].

— The array phases_3d_array[][][].

The output of this process is the 2-dimensional array gt_matrix[i][j].

Table 97 — Reconstruction process of genotype matrix

Process	Remarks

`reconstruct_genotype_matrix(allele_3d_array[][][],`
`  phases_3d_array[][][]) {`
`  {nrows, ncols, p} = Size(allele_3d_array)`
`  for (i=0; i<nrows; i++) {`
`  for (j=0; j<ncols; j++) {`
`  gt_matrix[i][j] == ""`
`  gt_matrix[i][j] = strcat(gt_matrix[i][j],`
`  allele_3d_array[i][j][0])`
`  for (k=1; k<p; k++) {`
`  if (phases_3d_array[i][j][k-1] == 1) {`
`  gt_matrix[i][j] = strcat(gt_matrix[i][j], "/")`
`  } else if (phases_3d_array[i][j][k-1] == 0) {`
`  gt_matrix[i][j] = strcat(gt_matrix[i][j], "|")`
`  }`
`  gt_matrix[i][j] = strcat(gt_matrix[i][j],`
`  allele_3d_array[i][j][k-1])`
`  }`
`  }`
`  }`
`}`
	Initialize empty string
	Store value of `allele_3d_array[][]` as char `c(1)`.
	Write phasing information
	Write phasing information
	Store value as char

9.3.6 Likelihood descriptor

This subclause describes the decoding process of the descriptor payloads when the `descriptor_ID` is equal to `LIKELIHOOD`.

The input to this process (see Table 98) are:

— the 3-dimensional array GL[][][] from process 9.3.6.2

— the current values of j_{16}.

The output of this process are the variables `likelihood` and `n_likelihoods`.

Table 98 — Decoding process of likelihood

Process	Remarks

n_samples = Size(GL[][][], 2)
n_likelihoods = Size(GL[][][], 3)
for (i=0; i< n_samples; i++){
  for (j=0; j< n_likelihoods; j++){
  likelihood[i][j] = GL[j_{16}++][i][j]
  }
}
j_{16}++
	From process specified in subclause 9.3.6.1.

9.3.6.1 Decode likelihood payload

The inputs of this process (see Table 99) are:

— the likelihood payload likelihood_payload as specified in subclause 6.4.4.3.3.

— the likelihood parameter likelihood_parameters as specified in subclause 6.3.4.3

The output of this process is a 3-dimensional array GL[][][]

Table 99 — Decoding process of likelihood payload

Process	Remarks

decode_likelihood_payload(likelihood_payload, likelihood_parameters) {
  if (transform_flag){
  C[][] = decode_code_matrix(nrows, ncols, payload)
  V[] = decode_values_vector(additional_payload)
  L[][] = inverse_trans_mip_rep(C[][], V[])
  } else {
  L[][] = decode_likelihood_matrix(payload)
  }
  GL = reconstruct_gl_matrix(L[][], num_gl_per_sample)
}
	u(dtype_id)
	floating point with 32bits precision.

num_gl_per_sample is part of likelihood_parameters specified in 6.3.4.3.

additional_payload is part of likelihood_payload specified in 6.4.4.3.3.

dtype_id is the identifier of data type of 2-dimensional array C[][]. Table 40 lists all of the possible values of dtype_id.

ncols is a part of likelihood payload specified in 6.4.4.3.3.

nrows is a part of likelihood payload specified in 6.4.4.3.3.

payload is part of likelihood payload specified in 6.4.4.3.3.

additional_payload is a part of likelihood payload specified in 6.4.4.3.3.

9.3.6.2 Decode code matrix

The inputs of this process (see Table 100) are:

— Number of rows `nrows`.

— the number of columns `ncols` as specified in subclause 6.4.4.3.3.

— the lzma payload `payload` as specified in subclause 6.4.4.3.3.

The output of this process is 2-dimensional array matrix `C[][]` of type u(dtype_id) as specified in Table 40.

Table 100 — Reshaping code matrix from an array to a 2-dimensional array

Process	Remarks

`decode_code_matrix(nrows, ncols) {`
`  C_vector[] = decode_lzma(payload)`
`  for (i = 0; i< nrows; i++){`
`  for (j=0; j < ncols; j++){`
`  C[i][j] = C_vector[i*ncols+j]`
`  }`
`  }`
`}`
	`u(dtype_id)`
	`u(dtype_id)`

`dtype_id` is the identifier of data type of 2-dimensional array `C[][]`. Table 40 lists the possible values of `dtype_id`.

9.3.6.3 Inverse transform minimal representation

The inputs of this process (see Table 101) are:

— 2-dimensional array C[][] called code matrix

— 1-dimensional array V[] called value vector. It can be array of type string or single precision floating-point

The output of this process is 2-dimensional array likelihood matrix `L[][]` of type float

Table 101 — Inverse transformation for minimal representation

Process	Remarks

`inverse_trans_min_rep() {`
`  for (i=0; i<m; i++) {`
`  for (j=0; j<n; j++) {`
`  L[i][j] = V[C[i][i]]`
`  }`
`  }`
`}`
	`u(nbits_per_elem)`

m is the number of rows of 2-dimensional array C.

n is the number of columns of code matrix C.

9.3.6.4 Decode likelihood matrix

The inputs of this process (see Table 102) are:

— the number of rows nrows as specified in subclause 6.4.4.3.3.

— the number of columns ncols as specified in subclause 6.4.4.3.3.

— the lzma payload payload as specified in subclause 6.4.4.3.3.

The output of this process is 2-dimensional array code matrix `L[][]` of type float.

Table 102 — Decoding process of likelihood matrix

Process	Remarks

`decode_likelihood_matrix() {`
`  L_vector[] = decode_payload(payload)`
`  for (i = 0; i< nrows; i++){`
`  for (j=0; j < ncols; j++){`
`  L[i][j] = L_vector[i*ncols+j]`
`  }`
`  }`
`}`
	floating point with 32 bits precision.

9.3.6.5 Reconstruct gl matrix

The inputs of this process (see Table 103) are:

— 2-dimensional array likelihood matrix L[][]

— Number of values per sample p with type of integer

The output of this process is the 3-dimensional array likelihood `GL[][][]`.

Table 103 — Reconstruction process of gl matrix

Process	Remarks

`reconstruct_gl_matrix() {`
`  [nrows, ncols] = Size(L[][])`
`  N = ncols / p`
`  for (i=0; i< nrows; i++) {`
`  for (j=0; j<N; j++) {`
`  for (k=0; k<p; k++) {`
`  GL[i][j][k] = L[i][j*p+k]`
`  }`
`  }`
`  }`
`}`
	Get number of samples

9.3.7 Contact matrix descriptor

The inputs to this process (see Table 104) are:

— the array start1[] from process specified in subclause 9.3.7.1.

— the array start2[] from process specified in subclause 9.3.7.1.

— the array end1[] from process specified in subclause 9.3.7.1.

— the array end2[] from process specified in subclause 9.3.7.1.

— the array count[] from process specified in subclause 9.3.7.1.

— the 2-dimensional array norm_mats_otf[][] from process specified in subclause 9.3.7.1.

— the 2-dimensional array norm_mats[][] from process specified in subclause 9.3.7.1.

— the value chr1_id specified in subclause 6.3.4.4.

— the value chr2_id specified in subclause 6.3.4.4.

— the value num_norm_methods from parameter set specified in subclause 6.3.4.4.

— the array norm_methods_name[] from parameter set specified in subclause 6.3.4.4.

— the value num_norm_matrices from parameter set specified in subclause 6.3.4.4.

— the array norm_matrices_name[] from parameter set specified in subclause 6.3.4.4.

Table 104 — Decoding process of contact matrix descriptor

Process	Remarks
`sampleID = sample_ID`	From process specified in subclause 9.3.7.1
`sampleNameLen = strlen(sample_name)`	From process specified in subclause 9.3.7.1
`sampleName = sample_name`	From process specified in subclause 9.3.7.1
`chr1ID = chr1_ID`	From process specified in subclause 9.3.7.1
`chr1NameLen = strlen(chr1_name)`	From process specified in subclause 9.3.7.1
`chr1Name = chr1_name`	From process specified in subclause 9.3.7.1
`chr2ID = chr2_ID`	From process specified in subclause 9.3.7.1
`chr2NameLen = strlen(chr2_name)`	From process specified in subclause 9.3.7.1
`chr2Name = chr2_name`	From process specified in subclause 9.3.7.1
`numCounts = size(counts[])`	From process specified in subclause 9.3.7.1
`chr1StartPos = start1[]`	From process specified in subclause 9.3.7.1
`chr1EndPos = end1[]`	From process specified in subclause 9.3.7.1
`chr2StartPos = start2[]`	From process specified in subclause 9.3.7.1
`chr2EndPos = end2[]`	From process specified in subclause 9.3.7.1
`count = counts[]`	From process specified in subclause 9.3.7.1
`numNormCounts = 0`	
`for (c=0; c<num_norm_methods ; c++){`	From parameter set specified in subclause 6.3.4.4
`  normCountNameLen[c] = strlen(norm_methods_name[c])`	From parameter set specified in subclause 6.3.4.4
`  normCountName[c] = norm_methods_name[c]`	From parameter set specified in subclause 6.3.4.4
`  normCount[c][] = norm_mats_otf[c][]`	From process specified in subclause 9.3.7.1
`  numNormCounts++`	
`}`	
`for (c=0; c<num_norm_matrices ; c++){`	From parameter set specified in subclause 6.3.4.4

Table 104 *(continued)*

Process	Remarks
normCountNameLen[num_norm_methods+c] = strlen(norm_matrices_name[c])	From parameter set specified in subclause 6.3.4.4
normCountName[num_norm_methods+c] = norm_matrices_name[c]	From parameter set specified in subclause 6.3.4.4
normCount[num_norm_methods+c][] = norm_mats[c][]	From process specified in subclause 9.3.7.1
numNormCounts++	
}	

chr1_name is the chromosome name of the first chromosome in chromosome pair. It corresponds to the chr_name[i] specified in 6.3.4.4 where chr_ID[i] is equals to chr1_ID.

chr2_name is the chromosome name of the second chromosome in chromosome pair. It corresponds to the chr_name[i] specified in 6.3.4.4 where chr_ID[i] is equals to chr2_ID.

9.3.7.1 Decode contact matrix (CM)

The inputs of this process (see Table 105) are:

— the CM mat parameters structure main_params specified in 6.3.4.4,

— the CM submat parameters structure params specified in 6.3.4.5,

— the CM mat payload structure mat_payload specified in 6.4.4.3.4.3,

— the interval multiplier mult,

— the CM bin payload structure bin_payload1 specified in 6.4.4.3.4.3 with chr_ID equals to params.chr1_ID, sample_ID equals to mat_payload.sample_ID and interval_multiplier equals to mult,

— the CM bin payload structure bin_payload2 specified in 6.4.4.3.4.3 with chr_ID equals to params.chr2_ID, sample_ID equals to mat_payload.sample_id and interval_multiplier equals to mult.

The outputs of this process are:

— the identifier of sample sample_ID equals to mat_payload.sample_ID

— the name of sample sample_name equals to main_params.sample_name[i] where main_params.sample_ID[i] equals to mat_payload.sample_ID,

— the identifier of sequence 1 chr1_ID equals to params.chr1_ID,

— the name of sequence 1 chr1_name equals to main_params.chr_name[i] where main_params.chr_ID[i] equals to params.chr1_ID.

— the array of start position values of sequence 1 start1[],

— the array of end position values of sequence 1 end1[],

— the identifier of sequence 2 chr2_ID equals to params.chr2_ID,

— the name of sequence 2 chr2_name equals to main_params.chr_name[i] where main_params.chr_ID[i] equals to params.chr2_ID.

— the array of start position values of sequence 2 start2[],

— the array of end position values of sequence 2 end2[],

— the array of count values count[],

— the array norm_mats_otf[][] if main_params.num_norm_methods is greater than 1,

— the array norm_mats[][] if main_params.num_norm_matrices is greater than 1,

Table 105 — Decoding process of contact matrix

Process	Remarks
decode_contact_matrix(main_params, params,mat_payload, mult bin_payload1, bin_payload2){	
[row_mask, col_mask] = decode_cm_mask(main_params, params, mat_payload)	Specified in 9.3.7.2
start1[] = {}	Initialize array
end1[] = {}	Initialize array
start2[] = {}	Initialize array
end2[] = {}	Initialize array
count[] = {}	Initialize array
for (k=0; k< main_params.num_norm_methods; k++)	
norm_mats_otf [k][] = {}	Initialize array
}	
for (k=0; k< main_params.num_norm_matrices; k++)	
norm_mats[k][] = {}	
}	
for (i=0; i<params.ntiles_in_row; i++){	
for (j=0; j<params.ntiles_in_col; j++){	
if (params.is_symmetrical && i>j){	
continue	
}	
if (mat_payload.tile_payload_sizes[i][j]==0){	
continue	
}	
tile[][] = decode_cmat_tile(mat_payload.tile_payload[i][j], mat_payload.tile_codec_ID[i][j], params.binarizaton_flags[i][j])	Specified in 9.3.7.4
if (params.binarizaton_flags[i][j] == 1){	
tile[][] = debinarize_mat(tile[][])	Specified in 9.3.7.5
}	
if (params.diagonal_transform_flags[i][j]==1){	
tile[][] = inverse_diag_transform(tile, params.diagonal_tranform_modes[i][j])	Specified in 9.3.7.6
}	
{start1_idx, end1_idx} = comp_start_end_idx(main_params,params.chr1_id, 1,i)	Specified in 9.3.7.7

Table 105 *(continued)*

Process	Remarks
{start2_idx, end2_idx} = comp_start_end_idx(main_params,params.chr2_id, 1,j)	Specified in 9.3.7.7
tile_row_mask = slice(row_mask, start1_idx, end1_idx)	Specified in 9.3.7.8
tile_col_mask = slice(col_mask, start2_idx, end2_idx)	Specified in 9.3.7.8
if (mult != 1){	
tile = conv_noop(tile, mult, tile_row_mask, tile_col_mask)	Specified in 9.3.7.11
{tmp_nrows, tmp_ncols} = Shape(tile)	
tile_row_mask = create_ones_mask(tmp_nrows)	Specified in 9.3.7.12
tile_col_mask = create_ones_mask(tmp_ncols)	Specified in 9.3.7.12
{start1_idx, end1_idx} = comp_start_end_idx(main_params, params.chr1_id, mult, i)	Specified in 9.3.7.7
{start2_idx, end2_idx} = comp_start_end_idx(main_params, params.chr2_id, mult, j)	Specified in 9.3.7.7
}	
start1_vect = comp_start(main_params, mult, start1_idx, end1_idx, tile_row_mask)	Specified in 9.3.7.9
start2_vect = comp_start(main_params, mult, start2_idx, end2_idx, tile_col_mask)	Specified in 9.3.7.9
end1_vect = comp_end(start1_arr, main_params, params.chr1_id)	Specified in 9.3.7.10
end2_vect = comp_end(start2_arr, main_params, params.chr2_id)	Specified in 9.3.7.10
{start1_desc, end1_desc,start2_desc, end2_desc} = tile_to_desc(tile, start1_arr, end1_arr,start2_arr, end2_arr)	Specified in
start1 = cat(start1, start1_desc)	Concatenate
end1 = cat(end1, end1_desc)	Concatenate
start2 = cat(start2, start2_desc)	Concatenate
end2 = cat(end2, end2_desc)	Concatenate
count = cat(count, count_desc)	concatenate
for (k=0; k< main_params.num_norm_methods; k++){	
mult_flag = main_params norm_methods_mult_flag[k]	
weight_values1 = bin_payload1.weight_values[k]	
tile_weight_vals1 = slice(weight_values1, start1_idx, end1_idx)	Specified in 9.3.7.8

Table 105 *(continued)*

Process	Remarks

weight_values2 = bin_payload2.weight_values[k]
tile_weight_vals2 = slice(weight_values2,
start2_idx, end2_idx)
kth_norm_tile = comp_otf_norm_mat(
tile,tile_row_mask, tile_col_mask,
tile_weight_vals1, tile_weight_vals2,
mult_flag)
norm_mats_otf[k] = Cat(norm_mats_otf[k],
kth_norm_tile)
}
for (k=0; k< main_params.num_norm_matrices; k++){
kth_data = decode(
mat_payload.norm_matrix_payloads[k][i][j])
norm_mats [k] = cat(norm_mats[k], tile_add_info_k)
}
}
}
}
	Specified in 9.3.7.8
	Specified in 9.3.7.14
	Use codec specific decode function

9.3.7.2 Decode contact matrix (CM) mask

The inputs of this process (see Table 106) are:

— the CM main params structure main_params specified in 6.3.4.4,

— the CM params structure params specified in 6.3.4.5,

— the CM matrix payload mat_payload specified in 6.4.4.3.4.3.

The output of this process are arrays row_mask[] and col_mask[].

Table 106 — Decoding process of mask payload

Process	Remarks

decode_cm_masks(main_params, params, mask_payload) {
row_nentries = main_params.num_bin_entries
if (params.row_mask_exists_flag){
row_mask[] = decode_cm_mask_payload(
mat_payload.row_mask_payload,
row_nentries)
}
else {
row_mask[] = create_ones_mask(row_nentries)
}
col_nentries = main_params.num_bin_entries
if (params.is_symmetrical){
	Given params.chr1_id, multiplier equals 1
	Specified in 9.3.7.3
	Specified in 9.3.7.12
	Given params.chr2_id, multiplier equals to 1

Table 106 *(continued)*

Process	Remarks

col_mask[] = row_mask[]
 	
} else if (params.col_mask_exists_flag){
col_mask = decode_cm_mask_payload (
mat_payload.col_mask_payload,
col_nentries)
} else {
col_mask[] = create_ones_mask(col_nentries)
}
}
	Because the masks are identical, using either row_nentries or col_nentries is allowed
	
	Specified in 9.3.7.12

9.3.7.3 Decode contact matrix (CM) mask payload

The inputs of this process (see Table 107) are:

— the CM mask payload mask_payload specified in 6.4.4.3.4.5,

— the number of entries num_bin_entries.

The output of this process is an array mask[].

Table 107 — Decoding process of CM mask payload

Process	Remarks

decode_cm_masks(mask_payload, num_bin_entries) {
if (mask_payload.transform_id == 0){
mask_array[] = mask_payload.mask_array[]
} else {
j = 0;
val = mask_payload.first_val
for (k=0; k<mask_payload.num_rl_entries; k++){
rl = mask_payload.rl_content[k]
for (i=0; i<rl; i++){
mask_array[j+i] = val
}
val = !val
 	
j += rl
for (; j<num_bin_entries; j++){
mask_array[j] = val
}
}
}
}
	Inverse value from 1 to 0 or vice versa.

9.3.7.4 Decode contact matrix (CM) tile

The input of this process (Table 108) are:

— the CM tile payload structure tile_payload specified in 6.4.4.3.4.4,

— unique codec identifier codec_ID The values of codec_ID is listed in Table 42. It can range between 0 to 5 inclusive.

— the binarization flag bin_flag.

The output of this process is a an array tile[][].

Table 108 — Decoding process of contact matrix tile

Process	Remarks

`decode_cmat_tile(tile_payload, codec_ID, bin_flag){`
`  if (codec_ID != JBIG) {`
`  decoded_symbols[] = decode(payload)`
	Use codec specific decode function

`  k = 0`
`  for (i=0; i< tile_payload.tile_nrows; i++) {`
`  for (j=0; j< tile_payload.tile_ncols; j++) {`
`  tile[i][j] = decoded_symbols[k]`
`  k = k + 1`
`  }`
`  }`
`  }`	

`  else {`
`  tile[][] = decode_jbig(tile_payload)`
`  }`
`}`
	Input is JBIG BIE structure

9.3.7.5 Debinarize tile

The input of this process (see Table 109) is:

— the array tile[][].

The output of this process is an array `trans_tile[][]`

Table 109 — Inverse transformation of row binarization

Process	Remarks

`debinarize_tile(tile[][]){`
`  {tile_nrows, tile_ncols} = Size(tile)`
`  trans_-ile_ncols = tile_ncols - 1`
`  trans_tile_nrows = 0`	

`  for (i=0; i<tile_nrows; i++){`	Compute the number of rows

`  trans_tile_nrows += tile[i][0]`
`  }`
`  irow = 0`	

Table 109 *(continued)*

Process	Remarks

bit_pos = 0
for(i=0; i< tile_nrows; i++){
for (j=1; j< tile_ncols; j++) {
trans_tile[i_row][j-1] += tile[i][j] << bit_pos
}
if (tile[i][0] == 1){
irow++
bit_pos = 0
}
else {
bit_pos++
}
}
}
	Marker found

9.3.7.6 Inverse diagonal transform

The input of this process (see Table 110) is:

— the array `tile[][]`.

— the diagonal transform mode `mode`.

The output of this process is an array `trans_tile[][]`.

Table 110 — Inverse transformation process of diagonal transformation

Process	Remarks
inverse_diag_transform(tile[][], mode){	
{nrows, ncols} = Size(tile[][])	Get the dimensions of tile
k = 0	
l = 0	
if (mode == 0) {	
num_diags = ncols	
for (i=0; i<ncols; i++){	trans_tile has the size of ncols by ncols
for (j=0; j<ncols; j++){	
trans_tile[i][j] = 0	Initialize with 0
}	
}	
for (k_diag=0; k_diag<num_diags; k_diag++){	
if (k == nrows){	
break	
}	
if (k_diag > 0) {	
j_offset = k_diag	
}	

Table 110 *(continued)*

Process	Remarks

```
    else {
      j_offset = 0
    }
    end_diag = ncols - j_offset
    for (i=0; i<end_diag; i++){
      j = i + j_offset
      trans_tile[i][j] = tile[k][l++]
      if (l == ncols){
        l = 0
        k++
        if (k == nrows){
          break
        }
      }
    }
  else {
    if (mode==1) {
      diag_idx = {0}
      i = 1
      for (diag_id=1; diag_id < Max(nrows, ncols); diag_id++){
        if (diag_id < ncols){
          diag_idx[i++] = diag_id
        }
        if (diag_id < nrows){
          diag_idx[i++] = -diag_id
        }
      }
    }
    else if (mode == 2) {
      i = 0
      for (diag_id = -nrows+1; diag_id < ncols; diag_id++) {
        diag_idx[i++] = diag_id
      }
    }
    else if (mode == 3) {
      i = 0
      for (diag_id = ncols-1; diag_id > -nrows; diag_id--) {
        diag_idx[i++] = diag_id
      }
    }
    for (o = 0; o < Size(diag_idx); o++) {
      diag_id = diag_idx[o]
      if (diag_id > 0 && diag_id < ncols) {
        i_offset = 0
```

Table 110 *(continued)*

Process	Remarks

j_offset = diag_id
}
else if (diag_id < 0 && -diag_id < nrows) {
i_offset = -diag_id
j_offset = 0
}
else if (diag_id == 0) {
i_offset = 0
j_offset = 0
} else {
continue
}
end_diag = Min(nrows-i_offset, ncols-j_offset)
for (k_diag=0; k_diag<end_diag; k_diag++) {
i = k_diag + i_offset
j = k_diag + j_offset
trans_tile[i][j] = tile[k][l++]
if (l == ncols){
l = 0
k++
}
}
}
}
}	

9.3.7.7 Compute start end indices

The inputs of this process (see Table 111) are:

— the number of entries nentries,

— the tile size tile_size,

— the tile index tile_idx.

The output of this process are integers start_idx and end_idx.

Table 111 — Computation process of start and end indices

Process	Remarks

comp_start_end_idx (nentries, tile_size, tile_idx){
start_idx = tile_idx * tile_size
end_idx = Min(nentries, start_idx + tile_size)
}	

9.3.7.8 Slice array

The inputs of this process (see Table 112) are:

— the array mask[],

— the start index start_idx,

— the end index end_idx.

The output of this process is an array sliced_mask[].

Table 112 — Process of slicing an array given start and end indices

Process	Remarks

slice(mask[], start_idx, end_idx){
  start_idx = Max(start_idx, 0)
  end_idx = Min(end_idx, Size(mask[])
  j = 0
  for (i = start_idx; i<end_idx; i++) {
  sliced_mask[j++] = mask[i]
  }
}	

9.3.7.9 Compute start array

The inputs of this process (see Table 113) are:

— the interval interval,

— the start index start_idx,

— the end index end_idx,

— the array tile_mask[].

The output of this process is an array start_arr[].

Table 113 — Initialization process of a start array

Process	Remarks

comp_start(interval, start_idx, end_idx, tile_mask[]){
  j = 0
  for (i=start_idx; i<end_idx; i++){
  if (tile_mask[i]){
  start_arr[j++] = i*interval
  }
  }
}	

9.3.7.10 Compute end array

The inputs of this process (see Table 114) are:

— the chromosome length chr_len.

— the interval interval.

— the start array start_arr[].

The output of this process is an array end_arr[].

Table 114 — Initialization process of an end array

Process	Remarks

comp_end_arr(chr_len, interval, start_arr[]){
  nentries = Size(start_arr[])
  for (i = 0; i<nentries; i++){
  end_arr[i] = start_arr[i] + interval
  }
  end_arr[nentries-1] = Min(end_arr[nentries-1],target_chr_len)
}	

9.3.7.11 Convolution no-op

The input of this process (see Table 115) are:

— the array tile[][].

— the window size ws.

— the Boolean array tile_row_mask[].

— the Boolean array tile_col_mask[].

The output of this process is a 2-dimensional array out_tile[][].

Table 115 — Computation of tile with higher interval using convolutional no-operation method

Process	Remarks

conv_noop (tile[][], ws, tile_row_mask[], tile_col_mask[]){
  {nrows, ncols} = Size(tile)
  out_nrows = Ceil(Size(tile_row_mask[])/ws)
  out_ncols = Ceil(Size(tile_col_mask[])/ws)
  for (i = 0; i < out_nrows; i++){
  for (j = 0; j < out_ncols; j++){
  out_tile[i][j] = 0
  }
  }
  i_tile = 0
  for (i = 0; i < Ceil(Size(tile_row_mask[])); i++){
  if (tile_row_mask[i]){
  i_otile = Floor(i/ws)
  j_tile = 0
  for (j = 0;j < Size(tile_col_mask[]); j++){
  if (tile_col_mask[i]) {
  j_otile = Floor(j/ws)
  out_tile[i_otile][j_otile] += tile[i_tile][j_tile]
  }
  }
	Initialize out_tile with zeros

Table 115 *(continued)*

Process	Remarks
i_tile++ } } }	

9.3.7.12 Create ones mask

The input of this process (see Table 116) is:

— the number of entries nentries.

The output of this process is a array mask[].

Table 116 — Initialization of an ones array

Process	Remarks

create_ones_mask (nentries){
if (i = 0; i < nentries; i++){
mask[i] = 1
}
}	

9.3.7.13 Tile to descriptor

The input of this process (see Table 117) are:

— the array tile[][].

— the array start1_arr[].

— the array end1_arr[].

— the array start2_arr[].

— the array end2_arr[].

The output of this process are arrays start1_desc[], end1_desc[], start2_desc[], end2_desc[] and count_desc[].

Table 117 — Computation from tile to descriptors

Process	Remarks

recon_tile_desc(tile[][], start1_arr[], end1_arr[],
start2_arr[], end2_arr[]) {
{tile_nrows, tile_ncols} = Size(tile[][])
k = 0
for (j=0; j<tile_ncols; j++){
 	
for (i=0; i<tile_nrows; i++){
if (tile[i][j] > 0){
row_idx[k] = i
	Find all entries of the tile[][] which values > 0 and store the position in the tile / matrix

Table 117 *(continued)*

Process	Remarks

col_idx[k++] = j
}
}
}
for (k=0; k<Size(row_idx); k++){
 	
i = row_idx[k]
j = col_idx[k]
tile_start1[k] = start1[i]
tile_end1[k] = end1[i]
tile_start2[k] = start2[j]
tile_end2[k] = end2[j]
tile_count[k] = tile[i][j]
}
}
	Create sparse representation of tile[][]

9.3.7.14 Compute on-the-fly normalized tile

The input of this process (see Table 118) are:

— the array tile[][].

— the array row_mask[].

— the array col_mask[].

— the array weight_values1[].

— the array weight_values2[].

— the flag mult_flag.

The output of this process is an array norm_counts[].

Table 118 — Computation process of on the fly normalized tile

Process	Remarks

comp_otf_norm_mat (tile[][], row_mask[], col_mask[],
weight_values1[], weight_values2[],
mult_flag)
nrows = Shape(row_mask)
ncols = Shape(col_mask)
nentries = 0
t_i = 0
for (i=0; i<nrows; i++){
if (row_mask[i] != 1){
continue
}
t_j = 0
for (j=0; j<ncols; j++){	

Table 118 *(continued)*

Process	Remarks

`if (col_mask[j] != 1){`
`  continue`
`}`
`if (tile[t_i][t_j] != 0){`
`  weight = weight_values1[i] * weight_values2[j]`
`  if (mult_flag == 1){`
`  norm_counts[nentries] = tile[t_i][t_j] *`
`  weight`
`  } else {`
`  norm_counts[nentries] = tile[t_i][t_j] /`
`  weight`
`  }`
`  nentries++`
`}`
`t_j=++`
`t_i++`
`}`
`}`
`}`	

9.4 Attributes decoding process

The input to this process (see Table 119) are:

— `attribute_ID` as specified in subclause 6.4.2 (`attribute_contiguity == 1`) or subclause 6.4.4.2 (`attribute_contiguity == 0`)

— the array `decoded_data[attribute_ID]` as specified in subclause 9.5.2

— the current value of $k_{attribute\_ID}$

The output of this process is the variable `attr_vals[attribute_ID]`.

Table 119 — Decoding process of attribute value

Process	Remarks

`n_cols = Size(decoded_data[attribute_ID], 2)`
`for (i = 0; i < n_cols; i++) {`
`  attr_vals[attribute_ID][i] =`
`  decoded_data[attribute_ID][`$k_{attribute\_ID}$`][i]`
`}`
$k_{attribute\_ID}$`++`	