
**Information technology — Genomic
information representation —**

**Part 6:
Coding of genomic annotations**





COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2023

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

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	vi
Introduction	vii
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Abbreviated terms	8
5 Conventions	8
5.1 General	8
5.2 Logical operators	8
5.3 Arithmetic operators	9
5.4 Relational operators	9
5.5 Bit-wise operators	9
5.6 Assignment operators	10
5.7 Range notation	10
5.8 Mathematical functions	10
5.9 Array and strings operation functions	11
5.10 Order of operation precedence	11
5.11 Variables, syntax elements and tables	12
5.12 Text description of logical operators	13
5.13 Processes	15
5.13.1 General	15
5.13.2 Process output operators	15
5.14 Method of specifying syntax in tabular form	16
5.15 Bit ordering	17
5.16 Specification of syntax functions and data types	17
5.17 Semantics	17
6 Data Structures	18
6.1 General	18
6.2 Data unit	18
6.3 Annotation parameter set	19
6.3.1 General	19
6.3.2 Tile configuration	20
6.3.3 Annotation encoding parameters	23
6.3.4 Descriptor configuration	24
6.3.5 Compressor parameter set	31
6.3.6 Attribute parameter set	32
6.4 Annotation access unit	34
6.4.1 General	34
6.4.2 Annotation access unit header	35
6.4.3 Annotation access unit types	36
6.4.4 Block	37
7 Descriptors and attributes semantics	46
7.1 General	46
7.2 Descriptors	48
7.2.1 General	48
7.2.2 Genomic intervals	48
7.2.3 Genomic variants	48
7.2.4 Functional annotations	48
7.2.5 Contact matrices	48
7.3 Attributes	48
7.4 Data types	48
7.4.1 General	48

7.4.2	Typed data.....	49
8	Decompression codecs.....	50
8.1	General.....	50
8.2	Inverse transformation algorithms.....	52
8.2.1	General.....	52
8.2.2	Lempel-Ziv-Welch transform.....	52
8.2.3	Binarization transform.....	53
8.2.4	Sparse transform.....	54
8.2.5	Delta transform.....	55
8.2.6	Run-Length Encoding transform.....	57
8.2.7	Serialization transform.....	57
8.3	Decompression algorithms.....	58
8.3.1	General.....	58
8.3.2	Context-Adaptive Binary Arithmetic Coding.....	59
8.3.3	Lempel-Ziv-Markov Chain Algorithm.....	59
8.3.4	Zstandard.....	59
8.3.5	JBIG.....	59
8.3.6	Block Sorting Coder.....	60
9	Decoding process.....	60
9.1	General.....	60
9.2	Access Units decoding process.....	60
9.2.1	General.....	60
9.2.2	Genomic variant access units.....	62
9.2.3	Functional annotation Access Units.....	64
9.2.4	Gene expression Access Units.....	65
9.2.5	Position-to-position contact intensity Access Units.....	66
9.2.6	Genome browser track Access Units.....	66
9.3	Descriptors decoding process.....	67
9.3.1	General.....	67
9.3.2	Common descriptors.....	68
9.3.3	Variant site information descriptors.....	70
9.3.4	Functional annotation descriptors.....	73
9.3.5	Genotype descriptor.....	75
9.3.6	Likelihood descriptor.....	84
9.3.7	Contact matrix descriptor.....	87
9.4	Attributes decoding process.....	102
9.5	Generic block payload decoding process.....	103
9.5.1	Descriptor payload decoding process.....	103
9.5.2	Attribute payload decoding process.....	104
10	Output format.....	107
10.1	Variant site record.....	107
10.1.1	General.....	107
10.1.2	Semantics.....	108
10.1.3	Initialization.....	110
10.2	Variant genotype record.....	111
10.2.1	General.....	111
10.2.2	Semantics.....	112
10.2.3	Initialization.....	113
10.3	Sample record.....	114
10.3.1	General.....	114
10.3.2	Semantics.....	114
10.3.3	Initialization.....	115
10.4	Functional annotation record.....	115
10.4.1	General.....	115
10.4.2	Semantics.....	116
10.4.3	Initialization.....	117
10.5	Track property record.....	118

10.5.1	General	118
10.5.2	Semantics	119
10.5.3	Initialization	119
10.6	Track data record	120
10.6.1	General	120
10.6.2	Semantics	121
10.6.3	Initialization	121
10.7	Expression record	122
10.7.1	General	122
10.7.2	Semantics	123
10.7.3	Initialization	123
10.8	Feature record	124
10.8.1	General	124
10.8.2	Semantics	125
10.8.3	Initialization	125
10.9	Contact matrix record	126
10.9.1	General	126
10.9.2	Semantics	127
10.9.3	Initialization	128
Bibliography		129

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.

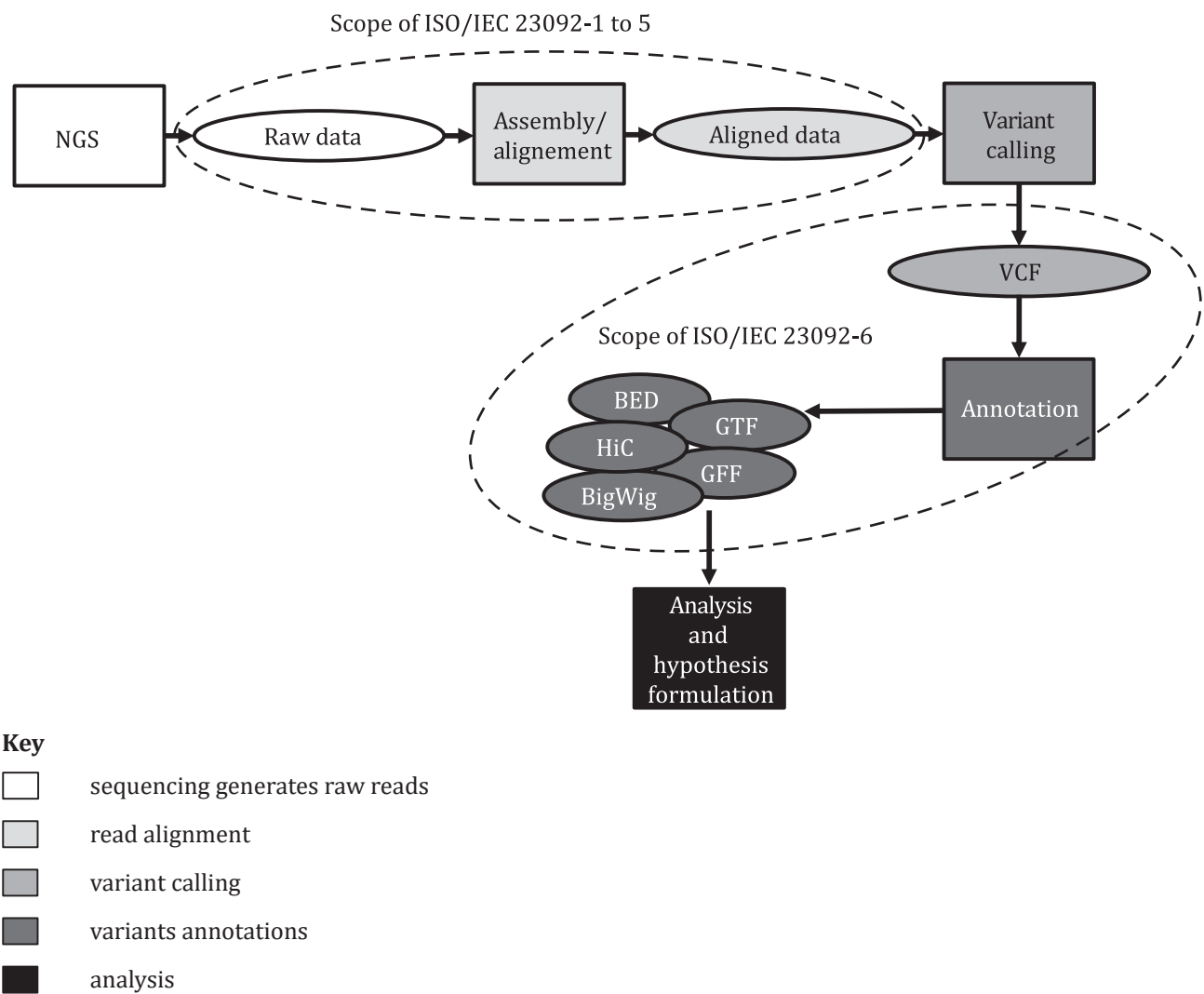


Figure 1 — Typical pipeline for the primary and secondary analyses of HTS data

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 usually 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**

matrix 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 $\{v1, v2, \dots, vn\} = f(.)$ by including a statement "return $\{v1, v2, \dots, vn\}$ " 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.
$\div$	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 $x \geq 0$ and $y > 0$.

5.4 Relational operators

The following relational operators are defined as follows:

$>$	Greater than
$\geq$	Greater than or equal to
$<$	Less than
$\leq$	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.
$x \gg y$	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.
$x \ll y$	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
$Min(x, y)$	$\begin{cases} x & ; \ x \leq y \\ y & ; \ x > y \end{cases}$
$Max(x, y)$	$\begin{cases} x & ; \ x \geq y \\ y & ; \ x < y \end{cases}$
$DivRem(x, y)$	the couple $(x/y, x\%y)$, i.e. the result and the remainder 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
<code>process_xyz(counter) {</code>	A decoding process with one parameter
num	A syntax element
for(<code>i = 0; i < num * counter; i++</code>) {	
value[i]	A syntax element
}	
a_sub_process	A sub-decoding process within <code>process_xyz</code>
}	

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
<code>another_process(xyz) {</code>	<i>xyz is the output of one specific run of <code>process_xyz</code> process.</i>
a = xyz.counter	Access the parameter ‘counter’ with which <code>process_xyz</code> was run to produce <code>xyz</code>
b = xyz.num	Access the syntax element ‘num’ of the <code>process_xyz</code> run that produced <code>xyz</code>
c = xyz.value[b]	Access the syntax element ‘value[b]’ of the <code>process_xyz</code> run that produced <code>xyz</code>

Table 3 (continued)

Syntax	Remarks
<pre> d = xyz.a_sub_process.some_member } </pre>	Access the syntax element ' <i>some_member</i> ' of sub-decoding process ' <i>a_sub_process</i> ' of the <i>process_xyz</i> run that produced <i>xyz</i>

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
<pre> /* 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 </pre>	u (n)
<pre> /*A group of statements enclosed in curly brackets is a compound statement and is treated func- tionally as a single statement. */ { statement statement ... } </pre>	
<pre> /* 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 </pre>	
<pre> /* 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) </pre>	
<pre> /* 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 </pre>	
<pre> /* 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 subse- quent statement until the condition is no longer true. */ </pre>	

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

Table 5 (continued)

Syntax	Type
<pre> else /*(data_unit_type > 4)*/{ /*skip data unit*/ } </pre>	

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
<pre> annotation_parameter_set() { parameter_set_ID AT_ID AT_alphabet_ID reserved AT_coord_size AT_pos_40_bits_flag n_aux_attribute_groups </pre>	<pre> u(8) u(8) u(8) u(2) u(2) u(1) u(3) </pre>	

Table 7 (continued)

Syntax	Type	Remarks
<pre> for (i = 0; i <= n_aux_attribute_groups; i++) { tile_config[i] } annotation_encoding_parameters() </pre>		As specified in subclause 6.3.2 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
<pre> tile_configuration { AG_class attribute_contiguity two_dimensional if (two_dimensional) { reserved column_major_tile_order symmetry_mode symmetry_minor_diagonal } else { reserved } attribute_dependent_tiles default_tile_structure() if (attribute_dependent_tiles) { n_add_tile_structures for (i=0; i<n_add_tile_structures; i++) { n_attributes[i] for (j=0; j<n_attributes[i]; j++) { attribute_ID[i][j] } n_descriptors[i] for (j=0; j<n_descriptors[i]; j++) { descriptor_ID[i][j] } additional_tile_structure[i] } } } </pre>	u(3) u(1) u(1) u(6) u(1) u(3) u(1) u(3) u(1) u(16) u(16) u(16) u(7) u(7)	As specified in subclause 6.3.2.2 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
<pre> tile_structure { reserved variable_size_tiles n_tiles if (variable_size_tiles) { for (i=0; i<n_tiles; i++) { for (j=0; j<(two_dimensional + 1); j++) { start_index[i][j] end_index[i][j] } } } else { for (j=0; j<(two_dimensional + 1); j++) { tile_size[j] } } } </pre>	u(7) u(1) u(v) u(v) u(v) u(v) u(v)	v dependent on AT_coord_size as specified in subclause 6.3.1 two_dimensional as specified in subclause 6.3.2.1 v dependent on AT_coord_size as specified in subclause 6.3.1 v dependent on AT_coord_size as specified in subclause 6.3.1 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
<pre> annotation_encoding_parameters () { n_filter for (i = 0; i < n_filter; i++) { filter_ID_len[i] filter_ID[i] desc_len[i] description[i] } n_features_names for(i = 0; i < n_features_names; i++){ features_name_len[i] features_name[i] } n_ontology_terms for(i = 0; i < n_ontology_terms; i++){ ontology_term_name_len[i] ontology_term_name[i] } n_descriptors for(i = 0; i < n_descriptors; i++) { descriptor_configuration[i] } n_compressors for(i = 0; i < n_compressors; i++){ </pre>	<pre> u(8) u(6) c(filter_ID_len[i]) u(10) c(desc_len[i]) u(8) u(6) c(features_name_len[i]) u(8) u(6) c(ontology_term_name_len[i]) u(8) u(8) </pre>	See subclause 6.3.4

Table 11 (continued)

Syntax	Type	Remarks
<pre> compressor_parameter_set[i] } n_attributes for(i = 0; i < n_attributes; i++){ attribute_parameter_set[i] } while(!byte_aligned()){ nesting_zero_bit } } </pre>	u(8) f(1)	See subclause 6.3.5 See subclause 6.3.6 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
<pre> descriptor_configuration { descriptor_ID encoding_mode_ID if(descriptor_ID == GENOTYPE){ genotype_parameters() } else if(descriptor_ID == LIKELIHOOD){ likelihood_parameters() } else if (descriptor_ID == CONTACT){ contact_matrix_parameters() } algorithm_parameters(encoding_mode_ID) } </pre>	u(8) u(8) See subclause 6.3.4.2 See subclause 6.3.4.3 See subclause 6.3.4.4 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
<pre> genotype_parameters() { max_ploidy no_reference_flag not_available_flag binarization_ID if (binarization_ID == BIT_PLANE) { num_bit_plane concat_axis } for (i = 0; i < num_variants_payloads; i++) { sort_variants_rows_flag[i] sort_variants_cols_flag[i] } } </pre>	u(8) u(1) u(1) u(3) u(8) u(8) u(1) u(1)

Table 13 (continued)

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

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
<pre>likelihood_parameters() { num_gl_per_sample transform_flag if (transform_flag) { dtype_id } }</pre>	<pre>u(8) u(1) u(8)</pre>

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
<pre> cm_mat_parameters() { num_samples for (i=0; i<num_samples; i++){ sample_ID[i] sample_name[i] } num_chrs for (i=0 i<num_chrs; i++) chr_ID[i] chr_name[i] chr_length[i] } interval # (replace resolution) tile_size num_interval_multipliers for (i=0; i<num_interval_multipliers; i++){ interval_multiplier[i] } num_norm_methods for (i=0; i<num_norm_methods; i++){ norm_method_ID[i] norm_method_name[i] norm_method_mult_flag[i] reserved } num_norm_matrices for (i=0; i<num_norm_matrices; i++){ norm_matrix_ID[i] norm_matrix_name[i] } } </pre>	u(8) u(8) st(v) u(8) u(8) st(v) u(64) u(32) u(32) u(8) u(32) u(8) u(8) st(v) u(1) u(7) u(8) u(8) 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_interval_multipliers specifies the number of entries of array `interval_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: $\text{Ceil}(\text{chr_length}[\text{chr_ID}] / (\text{interval_multiplier} * \text{interval}))$.

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: $\text{Ceil}(\text{num_bin_entries} / \text{tile_size})$.

target_tile_size is the decoded or target `tile_size` given multiplier `mult`. It is computed as follows: $\text{target_tile_size} = \text{Floor}(\text{tile_size} / \text{mult})$.

target_interval is the interval of decoded tile. It is computed as follows: $\text{target_interval} = \text{interval} * \text{mult}$.

target_chr_len is the chromosome length `chr_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
<pre>cm_submat_parameters() { chr1_ID chr2_ID for (i=0; i<ntiles_in_row; i++){ for (j=0; j<ntiles_in_col; j++){</pre>	<pre>u(8) u(8)</pre>

Table 19 (continued)

Syntax	Type
<pre> if (!(is_symmetrical && i > j)){ diag_transform_flag[i][j] if(diag_tranform_flag[i][j]){ diag_transform_mode[i][j] } binarization_flag[i][j] } } } row_mask_exists_flag col_mask_exists_flag while(!byte_aligned()){ nesting_zero_bit } </pre>	u(1) u(2) u(1) u(1) u(1) 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: $\text{ntiles_in_row} = \text{Ceil}(\text{nrows}/\text{tile_size})$ and **nrows** is inferred as follows: $\text{nrows} = \text{chr_length}[i]$ 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: $\text{ntiles_in_col} = \text{Ceil}(\text{ncols}/\text{tile_size})$ and **ncols** is inferred as follows: $\text{ncols} = \text{chr_length}[i]$ 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
<pre> compressor_parameter_set { compressor_ID n_compressor_steps for (i=0; i<n_compressor_steps; i++) { compressor_step_ID[i] algorithm_ID[i] use_default_pars[i] if(!use_default_pars[i]){ algorithm_parameters(algorithm_ID) } n_in_vars[i] for (j=0; j<n_in_vars[i]; j++){ in_var_ID[i][j] prev_step_ID[i][j] prev_out_var_ID[i][j] } n_completed_out_vars[i] for (j=0; j<n_completed_out_vars[i]; j++) { completed_out_var_ID[i][j] } } while(!byte_aligned()){ nesting_zero_bit } } </pre>	<pre> u(8) u(4) u(4) u(5) u(1) u(4) u(4) u(4) u(4) u(4) u(4) f(1) </pre>	As specified in subclause 8.1 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
<pre> attribute_parameter_set { attribute_ID attribute_name_len attribute_name attribute_type attribute_num_array_dims for (i = 0; i < attribute_num_array_dims; i++) { attribute_array_dims[i] } attribute_default_val attribute_miss_val_flag if (attribute_miss_val_flag) { attribute_miss_default_flag if (!attribute_miss_default_flag) { attribute_miss_val } attribute_miss_str } compressor_ID n_steps_with_dependencies for (i = 0; i < n_steps_with_dependencies; i++) { dependency_step_ID[i] n_dependencies[i] for (j = 0; j < n_dependencies[i]; j++) { dependency_var_ID[i][j] dependency_is_attribute[i][j] if (dependency_is_attribute[i][j]) { dependency_ID } } } } </pre>	<pre> u(16) u(8) c(attribute_name_len) u(8) u(2) u(8) x(v), where the type x (u or st) and v are dependent on attribute_type as specified in subclause 7.4 u(1) u(1) x(v), where the type x (u or st) and v are dependent on attribute_type as specified in subclause 7.4 st(v) u(8) u(4) u(4) u(4) u(4) u(1) u(16) </pre>

Table 21 (continued)

Syntax	Type
<pre> } else { dependency_ID } } while(!byte_aligned()){ nesting_zero_bit } } </pre>	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
<pre>annotation_access_unit() { AT_ID AT_type AT_subtype AG_class reserved annotation_access_unit_header for (i=0; i < n_blocks; i++) { block[i] } }</pre>	u(8) u(4) u(4) u(3) u(5) See subclause 6.4.2 n_blocks as specified in subclause 6.4.2 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](#).