



International
Standard

ISO/IEC 23092-2

**Information technology — Genomic
information representation —**

**Part 2:
Coding of genomic information**

*Technologies de l'information — Représentation des informations
génomiques —*

Partie 2: Codage des informations génomiques

**Third edition
2024-03**

Reference number
ISO/IEC 23092-2:2024(en)

Full document details: ISO IEC 23092-2-2024

© ISO/IEC 2024



Documents

Document Sharing Hub



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

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

© ISO/IEC 2024 – All rights reserved

Contents

	Page
Foreword	vii
Introduction	viii
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Abbreviated terms	6
5 Conventions	6
5.1 General	6
5.2 Arithmetic operators	6
5.3 Logical operators	7
5.4 Relational operators	7
5.5 Bit-wise operators	7
5.6 Assignment operators	8
5.7 Range notation	8
5.8 Mathematical functions	8
5.9 Order of operation precedence	8
5.10 Variables, syntax elements and tables	9
5.11 Text description of logical operators	10
5.12 Processes	11
6 Syntax and semantics	12
6.1 Method of specifying syntax in tabular form	12
6.2 Bit ordering	13
6.3 Specification of syntax functions and data types	13
6.4 Semantics	14
7 Data structures	14
7.1 General	14
7.2 Data unit	15
7.3 Raw reference	16
7.3.1 General	16
7.3.2 Syntax and semantics	16
7.4 Parameter set	16
7.4.1 Syntax and semantics	16
7.4.2 Encoding parameters	17
7.5 Access unit	23
7.5.1 Syntax and semantics	24
7.5.2 Access unit types	27
8 Descriptors	28
9 Sequencing reads	31
9.1 General	31
9.2 Supported symbols	31
9.3 Paired-end reads	33
9.4 Reverse-complement reads	33
9.5 Data classes	33
9.6 Aligned data	34
9.7 Unaligned data	35
10 Decoding process	35
10.1 General	35
10.2 dataset_type = 0 or 1	36
10.2.1 General	36
10.2.2 References padding	36
10.2.3 Type 1 AU (Class P)	37

10.2.4	Type 2 AU (Class N)	38
10.2.5	Type 3 AU (Class M)	38
10.2.6	Type 4 AU (Class I)	39
10.2.7	Type 5 AU (Class HM)	41
10.2.8	Type 6 AU (Class U)	41
10.3	dataset_type = 2	42
10.3.1	General	42
10.3.2	Type 1 AU	43
10.3.3	Type 2 AU	43
10.3.4	Type 3 AU	43
10.3.5	Type 4 AU	44
10.3.6	Type 6 AU	44
10.4	Genomic descriptors	44
10.4.1	General	44
10.4.2	pos	45
10.4.3	rcomp	45
10.4.4	flags	46
10.4.5	mmpo	47
10.4.6	mmtpe	49
10.4.7	clips	53
10.4.8	ureads	55
10.4.9	rln	56
10.4.10	pair	57
10.4.11	mscore	64
10.4.12	mmap	65
10.4.13	msar	68
10.4.14	rtp	69
10.4.15	rgroup	71
10.4.16	qv	71
10.4.17	rname	75
10.4.18	rft	75
10.4.19	rftt	76
10.4.20	tokentpe descriptors	77
10.5	sequence	85
10.5.1	General	85
10.5.2	Aligned reads (Classes P, N, M, I, HM)	85
10.5.3	Unmapped reads (Class HM, U)	86
10.6	e-cigar	86
10.6.1	Syntax	86
10.6.2	Decoding process for the first alignment	88
10.6.3	Decoding process for other alignments	95
10.6.4	Reference transformation	95
11	Representation of reference sequences	96
11.1	External reference	97
11.2	Embedded reference	97
11.3	Computed reference	97
11.3.1	General	97
11.3.2	Supported Algorithms	97
11.3.3	Reference transformation	98
11.3.4	PushIn	98
11.3.5	Local assembly	100
11.3.6	Global assembly	101
12	Block payload parsing process	101
12.1	General	101
12.2	Encoding Mode 0	102
12.3	Inverse binarizations	103
12.3.1	General	103
12.3.2	Binary (BI)	103

12.3.3	Truncated unary (TU)	104
12.3.4	Exponential golomb (EG)	104
12.3.5	Truncated exponential golomb (TEG)	105
12.3.6	Signed truncated exponential golomb (STEG)	105
12.3.7	Split unit-wise truncated unary (SUTU)	105
12.3.8	Signed split unit-wise truncated unary (SSUTU)	106
12.3.9	Double truncated unary (DTU)	106
12.3.10	Signed double truncated unary (SDTU)	107
12.4	Decoder configuration	107
12.4.1	Sequences and quality values	107
12.4.2	Support values	108
12.4.3	CABAC binarizations	109
12.4.4	Transformation parameters	112
12.4.5	Msar descriptor and read identifiers	113
12.4.6	State variables	114
12.5	Initialization process for context variables	117
12.6	Arithmetic decoding engine	117
12.6.1	Initialization	117
12.6.2	Arithmetic decoding process	118
12.7	Decoding process for sequence descriptors	124
12.7.1	General	125
12.7.2	Block payload decoding process	125
12.8	BSC decoding process	139
12.8.1	decoding process	139
13	Output format	141
13.1	General	141
13.2	MPEG-G record	141
13.2.1	General	141
13.2.2	number_of_template_segments	143
13.2.3	number_of_record_segments	143
13.2.4	number_of_alignments	143
13.2.5	class_ID	144
13.2.6	read_group_len	144
13.2.7	reserved	144
13.2.8	read_1_first	144
13.2.9	seq_ID	144
13.2.10	as_depth	144
13.2.11	read_len	144
13.2.12	qv_depth	144
13.2.13	read_name_len	144
13.2.14	read_name	145
13.2.15	read_group	145
13.2.16	sequence	145
13.2.17	quality_values	145
13.2.18	mapping_pos	145
13.2.19	ecigar_len	145
13.2.20	ecigar_string	145
13.2.21	reverse_comp	145
13.2.22	mapping_score	145
13.2.23	split_alignment	145
13.2.24	delta	146
13.2.25	split_pos	146
13.2.26	split_seq_ID	146
13.2.27	flags	146
13.2.28	more_alignments	146
13.2.29	next_pos	146
13.2.30	next_seq_ID	146
13.3	Initialization process	146

Annex A (informative) Tokenization of reads identifiers	150
Annex B (informative) Mapping quality	152
Annex C (informative) Inverse binarization examples	153
Annex D Block Sorting, Lossless Data Compression	157

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 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*.

This third edition cancels and replaces the second edition (ISO/IEC 23092-2:2020), which has been technically revised.

The main changes are as follows:

- inclusion of new low-complexity entropy coders in [subclause 7.4.2.2 \(Table 9\)](#): LZMA, ZSTD, BSC;
- inclusion of new indexed entropy coder in [subclause 7.4.2.2 \(Table 9\)](#): PROCRUSTES;
- inclusion of the specification of BSC decoding process in [subclause 12.8](#) and [Annex D](#);
- inclusion of a new flag (`extended_alignment_info`) in [subclause 13.2.1](#) to represent split alignment information in the compressed bitstream.

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

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

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

- Specifying an abstract representation of the sequencing data rather than a specific format with its direct implementation.
- Being designed at a time point when technologies and use cases are more mature. This permits addressing one limitation of the textual SAM format, for which the incremental ad-hoc addition of features followed along the years, resulting in an overall redundant and suboptimal format which was unnecessarily complicated.
- Separating free-field user-defined information with no clear semantics from the genomic data representation. This allows a fully interoperable and automatic exchange of information between different data producers.
- Allowing multiplexing of relevant metadata information with the data since data and metadata are partitioned at different conceptual levels.
- Following a strict and supervised development process which has proven successful in the last 30 years in the domain of digital media for the transport format, the file format, the compressed representation and the application program interfaces.

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

- Consistent, general and properly designed format definitions and data structures to store sequencing and alignment information. A robust framework which can be used as a foundation to implement different compression algorithms.
- Speed and flexibility in the selective access to coded data, by means of newly designed data clustering and optimized storage methodologies.
- Low latency in data transmission and consequent fast availability at remote locations, based on transmission protocols inspired by real-time application domains.
- Built-in privacy and protection of sensitive information, thanks to a flexible framework which allows customizable secured access at all layers of the data hierarchy.
- Reliability of the technology and interoperability among tools and systems, owing to the provision of a procedure to assess conformance to this document on an exhaustive dataset.
- Support to the implementation of a complete ecosystem of compliant devices and applications, through the availability of a normative reference implementation covering the totality of the ISO/IEC 23092 series.

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

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

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

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

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

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

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

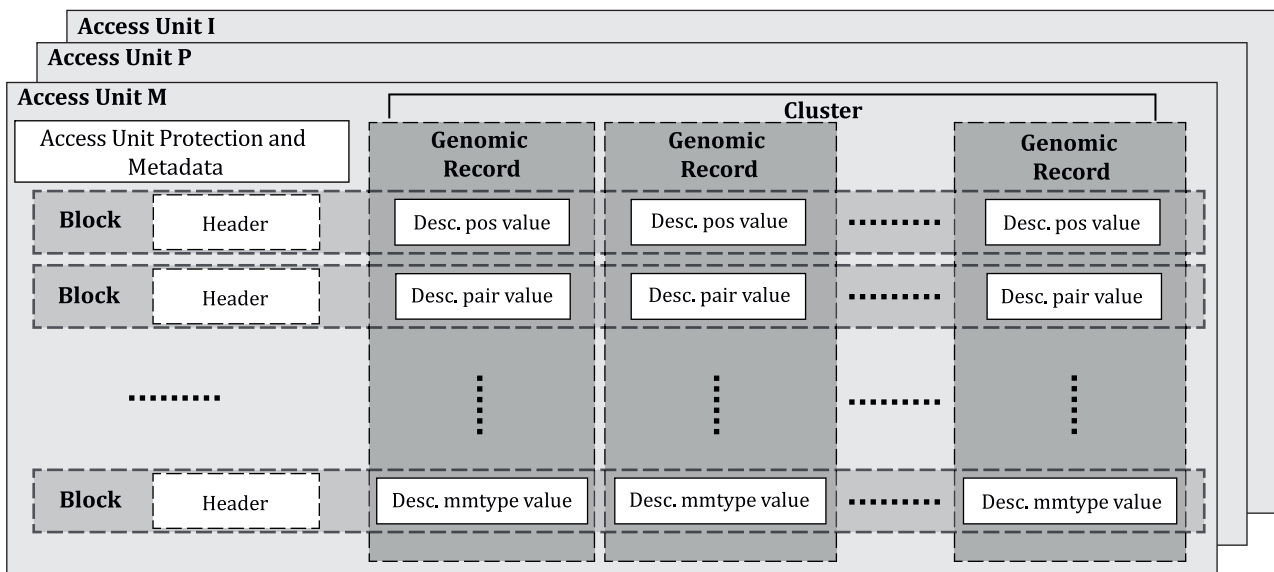


Figure 1 — Access units, blocks and genomic records

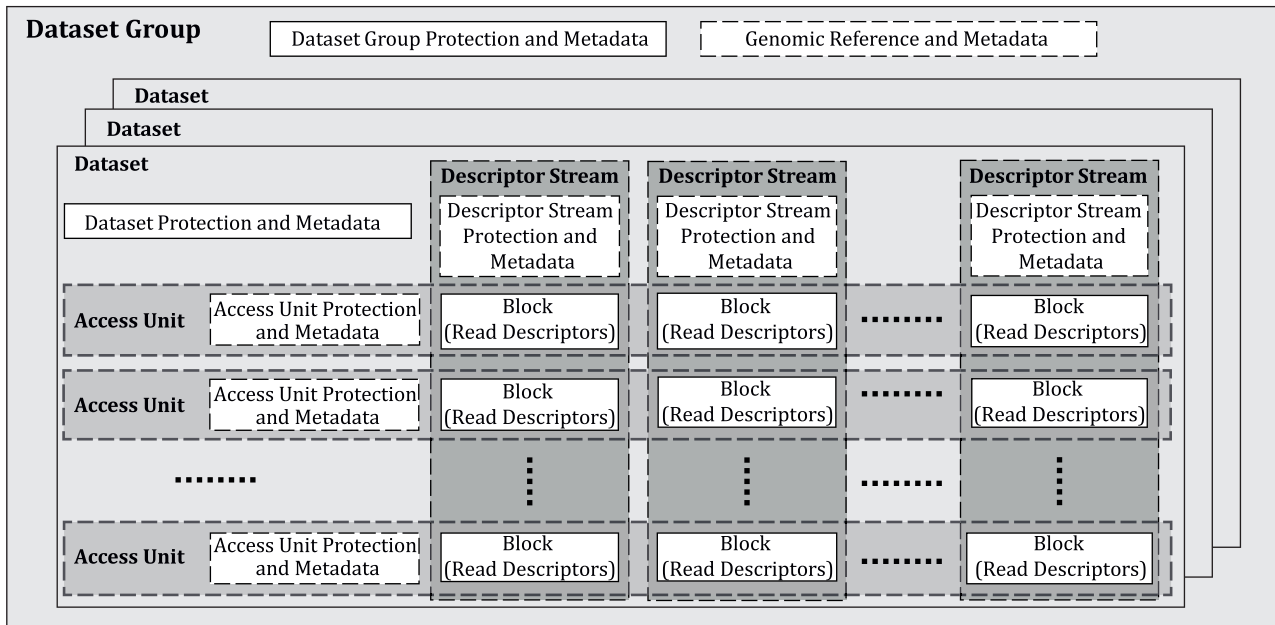


Figure 2 — High-level data structure: datasets and dataset group

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

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

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

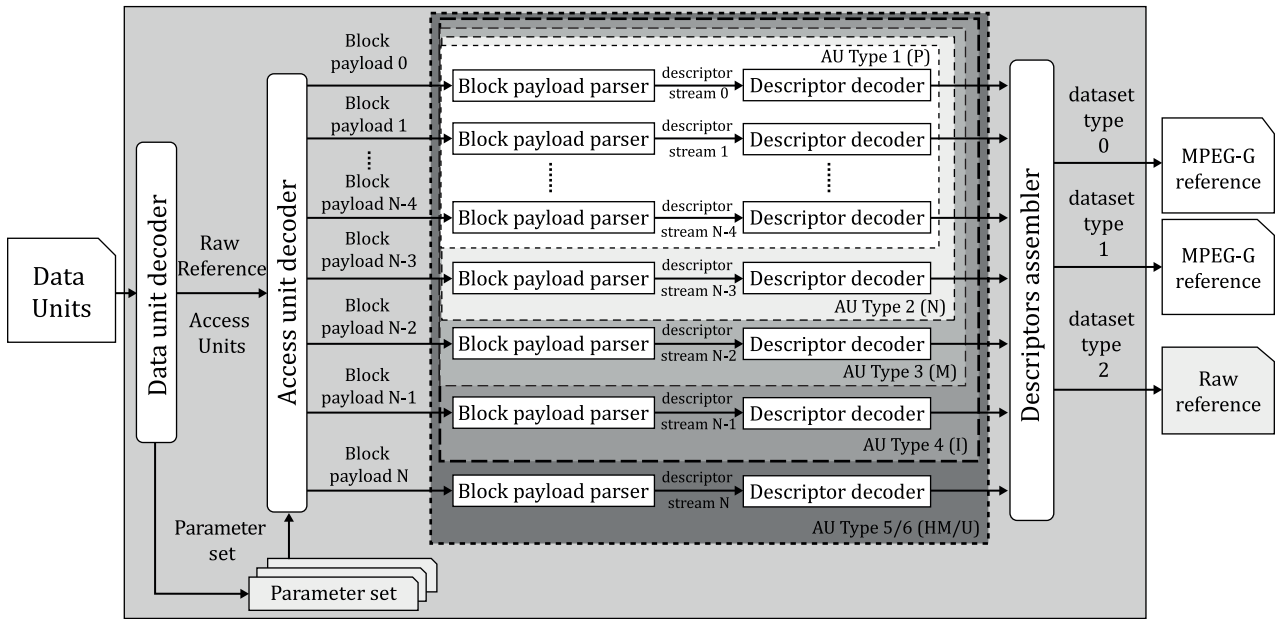


Figure 3 — The decoding process

Information technology — Genomic information representation —

Part 2: Coding of genomic information

1 Scope

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

- unaligned sequencing reads including read identifiers and quality values;
- aligned sequencing reads including read identifiers and quality values;
- reference sequences.

2 Normative references

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

ISO/IEC 10646, *Information technology — Universal coded character set (UCS)*

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

3 Terms and definitions

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

ISO and IEC maintain 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 alignment

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

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

3.2**CIGAR string**

CIGAR

textual way of representing an *alignment* ([3.1](#))

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

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

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

3.4**deletion**

contiguous removal of one or more bases from a genomic sequence

3.5**E-CIGAR**

extended CIGAR syntax specified as a superset of the CIGAR syntax

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

3.6**edit operation**modification of a sequence of *nucleotides* ([3.20](#)) by means of a substitution, *deletion* ([3.4](#)), *insertion* ([3.18](#)) or clip**3.7****FASTA**GIR that includes a name and a *nucleotide* ([3.20](#)) sequence for each *sequencing read* ([3.28](#))

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

3.8**FASTQ**GIR that includes *FASTA* ([3.7](#)) and *quality values* ([3.22](#))**3.9****first end**

end 1

read 1

first segment of a paired-end *template* ([3.33](#))

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

3.10**genomic descriptor**

descriptor

element of the syntax used to represent a feature of a genomic *sequencing read* ([3.28](#)) or associated information such as *alignment* ([3.1](#)) information or *quality values* ([3.22](#))**3.11****genomic information representation**

way to describe a sequence and some information associated with it

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

3.12**genomic record**

record

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

3.13**genomic record index**

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

3.15**genomic reference**

reference

collection of reference sequences

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

3.16**hard clip**

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

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

3.17**indel**

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

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

3.18**insertion**

contiguous addition of one or more bases into a genomic sequence

3.19**leftmost read end**

leftmost read

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

3.20**nucleotide**

base

base pair

monomer of a nucleic acid polymer such as DNA or RNA

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

3.21**paired-end read**

paired-end template

tuple (3.34) made of two segments

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

3.22**quality value**

quality score

number assigned to each *nucleotide* (3.20) base call in automated sequencing processes

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

3.23**read group**

set of reads having some property in common

3.24**read identifier**

read header

read name

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

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

3.25**rightmost read***sequencing read* (3.28) generated by a paired-end sequencing run and mapped at a position on the reference sequence which is greater than the mapping position of the other read in the pair**3.26****SAM**GIR that is human readable and includes FASTQ plus *alignment* (3.1) and analysis information

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

3.27**second end**

read 2

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

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

3.28**sequencing read**

read

readout, by a specific technology more or less prone to errors, of a continuous part of a segment of *nucleotides* (3.20) extracted from an organic sample**3.29****single-end read***tuple* (3.34) made of one segment**3.30****soft clip**

soft clipped bases

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

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

3.31**spliced read**

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

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

3.32**split alignment**

aligned *paired-end read* ([3.21](#)) whose ends are encoded in two different *genomic records* ([3.12](#))

3.33**template**

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

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

3.34**tuple**

collection of one or more segments

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

3.35**decoded genomic descriptor**

result of multiplexing the *decoded symbols* ([3.37](#)) of one or more *descriptor subsequences* ([3.36](#))

3.36**descriptor subsequence**

ordered collection of *decoded symbols* ([3.37](#))

3.37**decoded symbol**

value needed to reconstruct a *descriptor subsequence* ([3.36](#))

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

3.38**transformed subsequence**

ordered collection of *transformed symbols* ([3.39](#))

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

3.39**transformed symbol**

concatenation of one or more *decoded subsymbols* ([3.40](#))

3.40**decoded subsymbol**

output of an inverse subsymbol transformation applied on a *transformed subsymbol* ([3.41](#))

Note 1 to entry: See subclause [12.7.2.7](#). If no inverse subsymbol transformation is applied, the decoded subsymbol shall be equal to the transformed subsymbol.

3.41**transformed subsymbol**

decoded cabac subsymbol

atomic value yielded by the cabac decoding process

4 Abbreviated terms

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

5 Conventions

5.1 General

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

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

5.2 Arithmetic operators

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

5.3 Logical operators

$x \&\& y$ Boolean logical AND of x and y

$x \parallel y$ Boolean logical OR of x and y

$!$ Boolean logical NOT

$x ? y : z$ if x is TRUE or not equal to 0, evaluates to the value of y ; otherwise, evaluates to the value of z

5.4 Relational operators

$>$ greater than

$\geq$ 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

$\&$ AND

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

$|$ OR

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

$\wedge$ 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.

$x \gg y$ 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.

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

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

5.6 Assignment operators

=	assignment operator
++	increment i.e., $x++$ is equivalent to $x = x + 1$; when used in an array index, evaluates to the value of the variable prior to the increment operation.
--	decrement i.e., $x--$ is equivalent to $x = x - 1$; when used in an array index, evaluates to the value of the variable prior to the decrement operation.
+=	increment by amount specified i.e., $x += 3$ is equivalent to $x = x + 3$, and $x += (-3)$ is equivalent to $x = x + (-3)$.
-=	decrement by amount specified i.e., $x -= 3$ is equivalent to $x = x - 3$, and $x -= (-3)$ is equivalent to $x = x - (-3)$.
=	compound bitwise OR

5.7 Range notation

$x = y..z$	x takes on integer values starting from y to z , inclusive, with x , y , and z being integer numbers and z being greater than y
$\text{array}[x, y]$	sub-array containing the elements of array comprised between position x and y included If x is greater than y , the resulting sub-array is empty.

5.8 Mathematical functions

$\text{Ceil}(x)$	smallest integer greater than or equal to x	(1)
$\text{Floor}(x)$	largest integer less than or equal to x	(2)
$\text{Log2}(x)$	base-2 logarithm of x	(3)
$\text{Min}(x, y) = \begin{cases} x & ; x \leq y \\ y & ; x > y \end{cases}$		(4)
$\text{Max}(x, y) = \begin{cases} x & ; x \geq y \\ y & ; x < y \end{cases}$		(5)

5.9 Order of operation precedence

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

- Operations of a higher precedence are evaluated before any operation of a lower precedence.
- Operations of the same precedence are evaluated sequentially from left to right.

[Table 1](#) specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

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

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

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

5.10 Variables, syntax elements and tables

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

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

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

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

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

Functions that are not syntax functions (including mathematical functions specified in [subclause 5.2](#)) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case

letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

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

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

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

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

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

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

5.11 Text description of logical operators

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

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

may be described in the following manner:

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

- If condition 0, statement 0
- Otherwise, if condition 1, statement 1
- ...
- Otherwise (informative remark on remaining condition), statement n

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

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

```

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

```

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

- If all of the following conditions are true, statement 0:
 - condition 0a
 - condition 0b
- Otherwise, if one or more of the following conditions are true, statement 1:
 - condition 1a
 - condition 1b
- ...
- Otherwise, statement n

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

```

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

```

may be described in the following manner:

- When condition 0, statement 0
- When condition 1, statement 1

5.12 Processes

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

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

- If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower-case input or output variables of the process specification.
- Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

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

6 Syntax and semantics

6.1 Method of specifying syntax in tabular form

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

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

Table 2 — Examples of the syntax specification format

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

Table 2 (continued)

Syntax	Type
/* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. */	
for(initial statement; condition; subsequent statement)	
primary statement	

6.2 Bit ordering

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

6.3 Specification of syntax functions and data types

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

byte_aligned() is specified as follows:

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

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

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

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

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

- ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this data type is specified in [subclause 12.6.2.2](#).
- ae(t): context-adaptive arithmetic entropy-coded termination syntax. The parsing process for this data type is specified in [subclause 12.6.2.5](#).
- f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this data type is specified by the return value of the function read_bits(n).
- i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function read_bits(n) interpreted as a two's complement integer representation with most significant bit written first.
- se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this data type is specified in [subclause 12.3.4.2](#).
- st(v): null-terminated string encoded as universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in ISO/IEC 10646. The parsing process is specified as follows: st(v) reads and returns a series of bytes from the bitstream, beginning at the current position and continuing up to but not including the next byte that is equal to 0x00, and advances the bitstream pointer by (stringLength + 1) * 8 bit positions, where stringLength is equal to the number of bytes returned.

- $u(n)$: unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function `read_bits( n )` interpreted as a binary representation of an unsigned integer with most significant bit written first.
- $ue(v)$: unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this data type is specified in [subclause 12.3.4](#).
- $u7(v)$: variable sized unsigned integer computed by iteratively reading 8 bits, where the least significant 7 bits are interpreted as a binary representation of an unsigned integer v , with the most significant bit written first, and the 8th bit signaling if the iteration should stop. The parsing process for this data type is specified below:


```

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

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

6.4 Semantics

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

7 Data structures

7.1 General

[Subclause 7.2](#) specifies the structure of a data unit. A data unit is a data structure used as container for a raw reference structure, a parameter set structure or an access unit structure. [Table 3](#) and [Table 4](#) specify the data unit syntax and the values of associated data unit types.

[Subclause 7.3](#) specifies the structure of a raw reference.

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

[Subclause 7.5](#) specifies the structure of an access unit. An access unit consists of an access unit header, followed by one or more blocks. [Table 20](#) in [subclause 7.5.1.2](#) specifies the syntax for an access unit header.

Each block consists of a block header, as specified in [subclause 7.5.1.3.2](#), followed by a block payload as specified in [subclause 7.5.1.3.3](#).

7.2 Data unit

Table 3 — Data unit syntax

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

data_unit_type specifies the type of data unit. [Table 4](#) lists the values of data_unit_type and the associated data unit types.

Table 4 — Values of data_unit_type and associated data unit types

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

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

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

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

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

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

7.3 Raw reference

7.3.1 General

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

- deliver reference sequences to the decoder,
- return decoded reference sequences or part thereof from the decoder.

If a raw reference is required to decode access units, this raw reference shall be made available to the decoder prior to any other data unit. [Table 5](#) specifies the syntax and data type of row references.

7.3.2 Syntax and semantics

Table 5 — Raw reference syntax

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

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

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

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

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

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

7.4 Parameter set

7.4.1 Syntax and semantics

This subclause specifies the parameter set syntax and semantics. [Table 6](#) specifies syntax and data type of parameter sets.

Table 6 — Parameter set syntax

Syntax	Type
<code>parameter_set() {</code>	
parameter_set_ID	<code>u(8)</code>
parent_parameter_set_ID	<code>u(8)</code>
<code>encoding_parameters()</code>	
}	

parameter_set_ID is the unique identifier of the parameter set.

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

encoding_parameters() are the encoding parameters as specified in [subclause 7.4.2](#).

7.4.2 Encoding parameters

7.4.2.1 General

The encoding parameters are configuration parameters used during the decoding process are specified in [Table 7](#).

Table 7 — Encoding parameters syntax

Syntax	Type
<code>encoding_parameters() {</code>	
dataset_type	u(4)
alphabet_ID	u(8)
read_length	u(24)
number_of_template_segments_minus1	u(2)
reserved	u(6)
max_au_data_unit_size	u(29)
pos_40_bits_flag	u(1)
qv_depth	u(3)
as_depth	u(3)
num_classes	u(4)
for(j=0; j < num_classes; j++)	This for loop specifies the order of data classes for the entire syntax structure.
class_ID[j]	u(4)
for(i=0; i < NUM_DESCRIPTOR; i++){	
class_specific_dec_cfg_flag[i]	u(1)
if(class_specific_dec_cfg_flag[i] == 0) {	
descriptor_configuration(i)	Descriptor configuration, as specified in subclause 7.4.2.2 , applied to all classes.
} else {	
for(j=0; j< num_classes ; j++) {	
descriptor_configuration(i)	Descriptor configuration, as specified in 7.4.2.2 , applied to the class identified by class_ID[j].
}	
}	
num_groups	u(16)
for(j=0; j < num_groups; j++)	
rgroup_ID[j]	st(v)
multiple_alignments_flag	u(1)

Table 7 (continued)

Syntax	Type
spliced_reads_flag	u(1)
extended_alignment_info_flag	u(1)
reserved	u(29)
signature_flag	u(1)
if(signature_flag != 0){	
signature_constant_length_flag	u(1)
if(signature_constant_length_flag != 0){	
signature_length	u(8)
}	
}	
for (c = 0; c < num_classes; c++) {	
qv_coding_mode	u(4)
if(qv_coding_mode == 1){	
qvps_flag	u(1)
if(qvps_flag)	
parameter_set_qvps(class_ID[c])	See subclause 7.4.2.3 .
else	
qvps_preset_ID	u(4)
}	
qv_reverse_flag	u(1)
}	
crps_flag	u(1)
if(crps_flag)	
parameter_set_crps()	See subclause 7.4.2.4 .
while(!byte_aligned())	
nesting_zero_bit	f(1)
}	

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

alphabet_ID identifies the alphabet of symbols used for data encoded in access units referring to these encoding parameters. [Table 35](#) shows the alphabets associated to each value of **alphabet_ID**.

read_length specifies the length in bases of sequencing reads. The value 0 indicates the presence of variable read lengths or when there are multiple alignments with splices. Variable read lengths are signalled genomic record as specified in [subclause 10.4.9](#).

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

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

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

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

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

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

class_ID is one of the data class identifiers specified in [subclause 9.5](#). For any value of *ci* greater than 0 it shall always be $\text{class_ID}[ci] > \text{class_ID}[ci - 1]$.

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

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

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

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

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

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

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

reserved is set to 0 and reserved for future use.

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

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

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

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

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

qvps_preset_ID signals the ID of the quality values parameter set preset as specified in [subclause 10.4.16](#).

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

qv_reverse_flag signals if the decoded qv string shall be reversed in the decoding process specified in [subclause 10.4.16.2](#).

crps_flag signals the presence of a parameter_set_crps() element.

parameter_set_crps() is the computed reference parameter set as specified in [subclause 7.4.2.4](#). If not present, the computed reference parameters set of the parent parameter set identified by parent_parameter_set_ID shall be used.

nesting_zero_bit is one bit set to 0.

7.4.2.2 Descriptor configuration syntax and semantics

Table 8 — Descriptor configuration syntax

Syntax	Type
<code>descriptor_configuration(desc_ID) {</code>	
dec_cfg_preset	u(8)
<code>if(dec_cfg_preset == 0){</code>	
encoding_mode_ID	u(8)
<code>if((desc_ID != 11 && desc_ID != 15) (encoding_mode_ID != 0))</code>	
<code>decoder_configuration(encoding_mode_ID)</code>	As specified in 12.4.1
<code>else if(desc_ID == 11 desc_ID == 15){</code>	
<code>decoder_configuration_tokentype(encoding_mode_ID)</code>	As specified in 12.4.5 .
<code>}</code>	
<code>}</code>	
<code>else{</code>	
<code>/* reserved for future use */</code>	
<code>}</code>	
<code>}</code>	

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

encoding_mode_ID compression algorithm value as specified in [Table 9](#).

`decoder_configuration(encoding_mode_ID)` signals the decoder configuration parameters as specified in [subclause 12.4.1](#).

`decoder_configuration_tokentype(encoding_mode_ID)` signals the decoder configuration parameters as specified in [subclause 12.4.5](#).

Table 9 — Encoding mode values

encoding_mode_ID	Name	Description	Algorithm reference
0	CABAC	Context-Adaptive Binary Arithmetic Coding	See subclause 12.6
1	LZMA	Lempel-Ziv-Markov Chain Algorithm	ISO/IEC 23092-3:2022
2	ZSTD	Zstandard	https://tools.ietf.org/html/rfc8478
3	BSC	Block Sorting Coder	See subclause 12.8
4	PROCRUSTES	FMinindex-based compressor	ISO/IEC 23092-1:2020

7.4.2.3 Quality values parameter set syntax and semantics

7.4.2.3.1 General

Table 10 — Syntax of the quality values parameter set

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

[Table 10](#) specifies the syntax of the quality values parameter set.

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

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

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

`qvNumCodebooksAligned` is the state variable indicating the number of quality value codebooks used for aligned reads computed as specified in [Table 11](#).

Table 11 — Computation of `qvNumCodebooksAligned`

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

7.4.2.3.2 Quality values parameter set presets

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

7.4.2.3.2.1 Support of all printable ASCII characters

This set of parameters (refer to [Table 12](#)) supports the representation of all printable ASCII characters. It is identified by `qvps_preset_ID` equal to 0.

Table 12 — Parameters for the support of all printable ASCII characters

Parameter name	Value
qv_num_codebooks_total	1
qv_num_codebook_entries	94

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

$$\mathbf{qv_recon}[0][i] = i + 33$$

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

This set of parameters ([Table 13](#)) supports the representation of quantized quality values in the range 0..41 with an offset equal to 33. It is identified by qvps_preset_ID equal to 1.

Table 13 — Parameters for quantized quality values, offset 33, range 0-41

Parameter name	Value
qv_num_codebooks_total	1
qv_num_codebook_entries	8

[Table 14](#) shows how the reconstructed quality values **qv_recon**[0][i] are derived from the quality value indexes.

Table 14 — Values of qv_recon for each value of entry when qvps_ID is equal to 1

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

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

This set of parameters supports the representation of quantized quality values in the range 0..40 with an offset equal to 64 as specified in [Table 15](#). It is identified by qvps_preset_ID equal to 2.

Table 15 — Parameters for quantized quality values, offset 64, range 0-40

Parameter name	Value
qv_num_codebooks_total	1
qv_num_codebook_entries	8

[Table 16](#) specify how the reconstructed quality values **qv_recon**[0][i] are derived from the quality value indexes.

Table 16 — Values of `qv_recon` for each value of `i` when `qvps_preset_ID` is equal to 2

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

7.4.2.4 Computed Reference parameter set

This subclause specifies the data structure used to carry parameters related to the reference computation algorithms specified in [subclause 11.3](#). [Table 17](#) specifies the syntax of the computed reference parameter set.

Table 17 — Syntax of the computed reference parameter set

Syntax	Type
<code>parameter_set_crps() {</code>	
<code>cr_alg_ID</code>	<code>u(8)</code>
<code>if(cr_alg_ID == 2 cr_alg_ID == 3){</code>	
<code>cr_pad_size</code>	<code>u(8)</code>
<code>cr_buf_max_size</code>	<code>u(24)</code>
<code>}</code>	
<code>}</code>	

`cr_alg_ID` signals the reference computation algorithm as specified in [subclause 11.3.4](#). The possible values for `cr_alg_ID` are listed in [Table 18](#). The values 0 and 5..255 are reserved.

Table 18 — Values of `cr_alg_ID` and corresponding reference computation algorithms

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

`cr_pad_size` is the number of bases used for padding in the process specified in [subclause 11.3.4](#).

`cr_buf_max_size` is the maximum size in bytes of the buffer used in the decoding process as specified in [subclause 11.3.4](#) and [subclause 11.3.5](#).

7.5 Access unit

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

7.5.1 Syntax and semantics

7.5.1.1 General

This subclause specifies the access unit syntax ([Table 19](#)) and semantics.

Table 19 — Access unit syntax

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

`access_unit_header()` is specified in [subclause 7.5.1.2](#).

num_blocks specifies the number of blocks encoded in the access unit and it is encoded in the `access_unit_header` as specified in [subclause 7.5.1.2](#).

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

7.5.1.2 Access unit header

This subclause specifies the access unit header syntax and semantics. [Table 20](#) specify the syntax of the access unit header.

Table 20 — Access unit header syntax

Syntax	Type
<code>access_unit_header() {</code>	
<code>access_unit_ID</code>	u(32)
<code>num_blocks</code>	u(8)
<code>parameter_set_ID</code>	u(8)
<code>AU_type</code>	u(4)
<code>reads_count</code>	u(32)
<code>if (AU_type == N_TYPE_AU AU_type == M_TYPE_AU) {</code>	
<code>mm_threshold</code>	u(16)
<code>mm_count</code>	u(32)
<code>}</code>	
<code>if (dataset_type == 2) {</code>	
<code>ref_sequence_ID</code>	u(16)
<code>ref_start_position</code>	u(posSize)
<code>ref_end_position</code>	u(posSize)
<code>}</code>	
<code>if (AU_Type != U_TYPE_AU)</code>	
<code>{</code>	
<code>sequence_ID</code>	u(16)
<code>AU_start_position</code>	u(posSize)
<code>AU_end_position</code>	u(posSize)
<code>if (multiple_alignments_flag) {</code>	Specified in subclause 7.4.2 .

Table 20 (continued)

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

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

num_blocks specifies the number of Blocks in the access unit.

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

AU_type identifies the type of access unit and the class of data carried therein as specified in [subclause 7.5.2](#).

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

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

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

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

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

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

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

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

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

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

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

num_signatures specifies the number of signatures used to index unmapped reads as specified in ISO/IEC 23092-1:2020.

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

signature is the unsigned integer representing the signature of the cluster this access unit belongs to, as specified in ISO/IEC 23092-1:2020. The length in bits of this field, named `signatureSize` shall be calculated using the **signature_length** specified in [Table 20](#) as follow:

$$\text{signatureSize} = \text{signature_length} * \text{bits_per_symbol}$$

with `bits_per_symbol` corresponding to `BitsPerSymbol(Salphabet_ID)` as specified in [Table 35](#) with `alphabet_ID` as specified in [subclause 7.4.2](#), and with `signature_length` corresponding either to `signature_length` as specified in [subclause 7.4.2](#) when `signature_constant_length_flag` (as specified in [subclause 7.4.2](#)) is equal to 1 or to the signature-specific `signature_length[i]` specified in [Table 20](#) when `signature_constant_length_flag` (specified in [subclause 7.4.2](#)) is equal to 0. The *j*-th base in a signature is represented by the `u(bits_per_symbol)` value computed as follows:

$$\text{signature_base}[i][j] = S_{\text{alphabet_ID}}[(\text{signature}[i] \gg ((\text{signature_length} - j - 1) * \text{bits_per_symbol})) \& ((1 \ll \text{bits_per_symbol}) - 1)]$$

with `Salphabet_ID` as specified in [Table 35](#) with `alphabet_ID` as specified in [subclause 7.4.2](#)

`posSize` is specified in [subclause 7.4.2](#).

7.5.1.3 Block

7.5.1.3.1 General

This subclause specifies the block syntax ([Table 21](#)) and semantics.

Table 21 — Block syntax

Syntax	Type
<code>block() {</code>	
<code>block_header()</code>	block header
<code>block_payload()</code>	block payload
<code>}</code>	

block_header is a block header structure as specified in [subclause 7.5.1.3.2](#).

block_payload is a block payload structure as specified in [subclause 7.5.1.3.3](#).

7.5.1.3.2 Block header

This subclause describes the block header syntax ([Table 22](#)) and semantics.

Table 22 — Block header syntax

Syntax	Type
<code>block_header() {</code>	
reserved	u(1)
descriptor_ID	u(7)
reserved	u(3)
block_payload_size	u(29)
<code>}</code>	

reserved is set to 0 and used to preserve byte alignment.

descriptor_ID signals the descriptor type as specified in [Table 25](#). Its value shall be unique among all blocks in the access unit.

block_payload_size specifies the size in bytes of the block payload.

7.5.1.3.3 Block payload

This subclause specifies the syntax ([Table 23](#)) and semantics of the block payload structure containing entropy-coded descriptors.

Table 23 — Block payload syntax

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

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

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

nesting_zero_bit is one bit set to 0.

7.5.2 Access unit types

AUs can be of different types according to the nature of the coded data. An access unit contains encoded genomic records belonging to a single data class as shown in [Table 24](#).

Table 24 — Class of encoded data per access unit type

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

The blocks of descriptors encoded in one access unit as specified in [subclause 7.5.1.3](#) are those corresponding to sequencing reads belonging to one class of data as specified in [subclause 9.5](#). Descriptors carried by each access unit type are listed in [Table 25](#).

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

8 Descriptors

When dataset_type specified in [subclause 7.4.2](#) is equal to 0 or 1, the only mandatory descriptors are those required to represent the sequences of nucleotides, whereas read names and quality values are optional.

Descriptors are the output of the decoding process specified in [clause 12](#).

Descriptors required for the representation of sequencing reads, quality values, read names and transformed reference sequences are reported in [Table 25](#). Descriptors are specified in [subclause 10.4](#) and its subclauses.

Subsequence semantics and types for each descriptor_ID of [Table 25](#) are reported in [Table 26](#), [Table 27](#), [Table 28](#), [Table 29](#), [Table 30](#), [Table 31](#), [Table 32](#), [Table 33](#), [Table 34](#).

Table 25 — Genomic descriptors

descriptor_ID	Genomic descriptor name	Number of descriptor subsequences	Decoding process
sequencing reads			
0	pos	2	10.4.2
1	rcomp	1	10.4.3
2	flags	Variable, as specified in subclause 10.4.4 .	10.4.4
3	mmpo	2	10.4.5
4	mmtpe	3	10.4.6
5	clips	4	10.4.7
6	ureads	1	10.4.8
7	rln	1	10.4.9
8	pair	8	10.4.10
9	mscore	1	10.4.11
10	mmap	5	10.4.12
11	msar	Variable, as specified in subclause 10.4.13 .	10.4.13
12	rtp	1	10.4.14
13	rgroup	1	10.4.15
quality values			

Table 25 (continued)

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

Table 26 — Subsequences for descriptor_ID = 0 (pos descriptor)

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

Table 27 — Subsequences for descriptor_ID = 2 (flags descriptor)

subsequence_ID	Semantics	Type
0	Read is PCR or optical duplicate.	Unsigned integer with value either 0 or 1.
1	Read fails platform/vendor quality checks.	Unsigned integer with value either 0 or 1.
2	Read mapped in proper pair	Unsigned integer with value either 0 or 1.
3	Not primary alignment	Unsigned integer with value either 0 or 1.
4	Supplementary alignment	Unsigned integer with value either 0 or 1.

Table 28 — Subsequences for descriptor_ID = 3 (mmpos descriptor)

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

Table 29 — Subsequences for descriptor_ID = 4 (mmtype descriptor)

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

Table 30 — Subsequences for descriptor_ID = 5 (clips descriptor)

subsequence_ID	Semantics	Type
0	Record identifier	Unsigned integer.
1	Type/Position flag	Unsigned integer.
2	Nucleotides indexes with terminators	Unsigned integer.
3	Hard clips length	Unsigned integer.

Table 31 — Subsequences for descriptor_ID = 8 (pair descriptor)

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

Table 32 — Subsequences for descriptor_ID = 10 (mmap descriptor)

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

Table 33 — Subsequences for descriptor_ID = 11 and 15 (msar and rname descriptors)

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

Table 34 — Subsequences for descriptor_ID = 14 (qv descriptor)

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

9 Sequencing reads

9.1 General

This clause specifies the semantics of genomic descriptors used to represent nucleotides segments and associated alignment information. Each template produced by a sequencing machine or alignment generated by an aligner is encoded in a genomic record by means of a subset of the genomic descriptors described in this clause. The genomic descriptors are extracted from a compliant bitstream according to the processes described in [subclause 12.7](#) and the genomic templates with the associated alignment information can be reconstructed from the decoded genomic descriptors according to the decoding processes described in [subclause 10.4](#).

9.2 Supported symbols

The supported alphabets are specified in [Table 35](#).

Table 35 — Identifiers of alphabets supported for sequencing reads representation

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

Each alphabet is identified by an **alphabet_ID** as shown [Table 35](#).

The notation $S_{\text{alphabet_ID}}[\text{index}]$ specifies a conversion from a numerical index to an ASCII character corresponding to a symbol of the alphabet identified by **alphabet_ID**, as specified in [Table 36](#):

Table 36 — Conversions from numerical indexes to ASCII characters corresponding to alphabet symbols

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

The notation $\text{Code}_{\text{alphabet_ID}}[\text{symbol}]$ specifies the inversion conversion of $S_{\text{alphabet_ID}}[\text{index}]$, such that $\text{Code}_{\text{alphabet_ID}}[S_{\text{alphabet_ID}}[\text{index}]]$ is always equal to index for any valid value of index as specified in [Table 36](#).

Each alphabet symbol **Sym** is associated with a complementary symbol **Complement(Sym)** as specified in [Table 37](#).

Table 37 — Complementary alphabet symbols

$S_0[\text{index}]$	$S_0[\text{Complement}(\text{index})]$	$S_1[\text{index}]$	$S_1[\text{Complement}(\text{index})]$
$S_0[0] = \text{"A"}$	$S_0[3] = \text{"T"}$	$S_1[0] = \text{"A"}$	$S_1[3] = \text{"T"}$
$S_0[1] = \text{"C"}$	$S_0[2] = \text{"G"}$	$S_1[1] = \text{"C"}$	$S_1[2] = \text{"G"}$
$S_0[2] = \text{"G"}$	$S_0[1] = \text{"C"}$	$S_1[2] = \text{"G"}$	$S_1[1] = \text{"C"}$
$S_0[3] = \text{"T"}$	$S_0[0] = \text{"A"}$	$S_1[3] = \text{"T"}$	$S_1[0] = \text{"A"}$
$S_0[4] = \text{"N"}$	$S_0[4] = \text{"N"}$	$S_1[4] = \text{"R"}$	$S_1[5] = \text{"Y"}$
N/A		$S_1[5] = \text{"Y"}$	$S_1[4] = \text{"R"}$
N/A		$S_1[6] = \text{"S"}$	$S_1[6] = \text{"S"}$
N/A		$S_1[7] = \text{"W"}$	$S_1[7] = \text{"W"}$
N/A		$S_1[8] = \text{"K"}$	$S_1[9] = \text{"M"}$
N/A		$S_1[9] = \text{"M"}$	$S_1[8] = \text{"K"}$
N/A		$S_1[10] = \text{"B"}$	$S_1[13] = \text{"V"}$
N/A		$S_1[11] = \text{"D"}$	$S_1[12] = \text{"H"}$
N/A		$S_1[12] = \text{"H"}$	$S_1[11] = \text{"D"}$
N/A		$S_1[13] = \text{"V"}$	$S_1[10] = \text{"B"}$
N/A		$S_1[14] = \text{"N"}$	$S_1[14] = \text{"N"}$
N/A		$S_1[15] = \text{"."}$	$S_1[15] = \text{"."}$

9.3 Paired-end reads

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

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

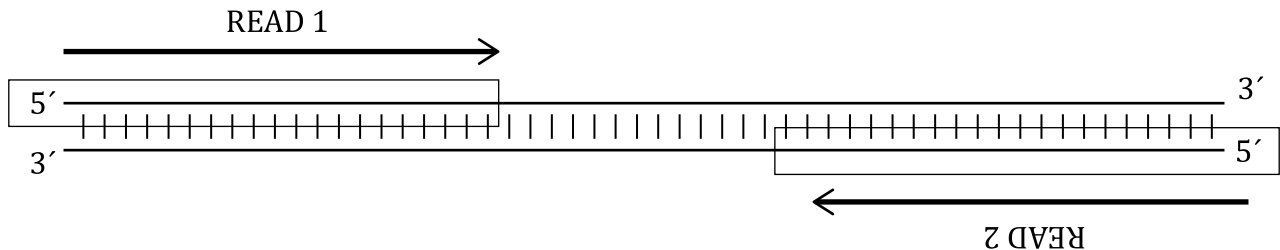


Figure 4 — Read 1 sequenced from the forward strand and read 2 from the reverse strand

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

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

9.4 Reverse-complement reads

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

$$\text{ReverseComplementRead}[n] = \text{Complement}(\text{Read}[\text{Size}(\text{Read}[]) - n - 1]), \text{ for } n \text{ in } 0 \dots \text{Size}(\text{Read}[]) - 1.$$

9.5 Data classes

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

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

The data classes and their descriptions are specified in [Table 38](#).

Table 38 — Sequence data classes

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

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

9.6 Aligned data

In the context of this document, aligned genomic data are genomic segments which require the use of an external or embedded reference genome (as specified in [subclause 10.6.2.2](#)) to be decoded.

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

Once decoded, each descriptor shall be used to initialize one or more output record fields as specified in [Clause 13.3](#). [Table 39](#) lists the descriptors used for aligned reads with a brief description and reference to the corresponding clause.

Table 39 — Descriptors used to represent aligned sequencing reads

descriptor_ID	descriptor	Semantics	subclause
0	pos	Read mapping position.	10.4.2
1	rcomp	Strand information for reads in a template.	10.4.3
2	flags	Additional alignment information usually produced by aligners.	10.4.4
3	mmpos	Position of mismatches in reads.	10.4.5
4	mmtyp	Type of mismatches.	10.4.6
5	clips	Information on clipped bases (i.e. Soft clips or hard clips).	10.4.7
6	ureads	Unmapped reads encoded verbatim.	10.4.8
7	rln	Read lengths.	10.4.9
8	pair	Represents: 1.a The unsigned distance from one segment to the next. OR 1.b The absolute position on a reference sequence of a segment in a template. AND 2 Information signaling if the leftmost mapped read in the genomic record is read 1.	10.4.10
9	mscore	Provides a score per alignment .	10.4.11

Table 39 (continued)

descriptor_ID	descriptor	Semantics	subclause
10	mmap	Used to represent multiple alignments.	10.4.12
11	msar	Supports spliced alignments and alternative secondary alignments which do not preserve the same contiguity of mapping of the primary alignment.	10.4.13
13	rgroup	Identifier of the read group each genomic record belongs to.	10.4.15

9.7 Unaligned data

Unaligned reads belong to class U only. They are encoded as unmapped reads in aligned datasets. Some of the descriptors specified for reads aligned to an external or internal reference as specified in [subclause 9.6](#) are used to encode unaligned reads (see [Table 40](#)). This is motivated by the fact that unaligned reads are encoded using reference sequences built from the data to be encoded. The reference used for mapping is computed according to the procedures described in [subclause 11.3](#).

Table 40 — Descriptors used to represent raw sequencing reads

descriptor_ID	Descriptor	Semantics	Subclause
0	pos	Read mapping position.	10.4.2
1	rcomp	Strand information for reads in a template.	10.4.3
2	flags	Additional alignment information usually produced by aligners.	10.4.4
3	mmpos	Mismatch position.	10.4.5
4	mmtype	Type of edit operations: — substitutions; — deletions; — insertions.	10.4.6
5	clips	String of nucleotides with variable length (e.g. soft clips).	10.4.7
6	ureads	Unmapped reads encoded verbatim.	10.4.8
7	rlen	Unsigned integer representing the number of bases in the read minus one.	10.4.9
8	pair	Represents: 1.a The unsigned distance from one segment to the next. OR 1.b The absolute position on a computed reference sequence of a segment in a template. AND 2 Information signaling if the first read in the genomic record is read 1.	10.4.10
12	rtype	This identifies the subset of descriptors needed to decode the read.	10.4.11
13	rgroup	Identifier of the read group each genomic record belongs to.	10.4.15

10 Decoding process

10.1 General

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

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

- a) a raw reference as specified in [subclause 7.3](#).
- b) a list of ISO/IEC 23092 series records as specified in [Clause 13](#).

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

10.2 dataset_type = 0 or 1

10.2.1 General

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

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

decoded_symbols[descriptor_ID][descriptor_subsequence_ID][j_{descriptor_ID, descriptor_subsequence_ID}]

where j_{descriptor_ID, descriptor_subsequence_ID} is the index to read the decoded symbols as specified in [subclause 12.1](#). The valid values of descriptor_ID are specified in [Table 25](#). The values of descriptor_subsequence_ID are between 0 and the number of descriptor subsequences minus 1 as specified in [Table 25](#).

At the beginning of the decoding process of each AU all indexes j_{descriptor_ID, descriptor_subsequence_ID} are initialized to 0.

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

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

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

10.2.2 References padding

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

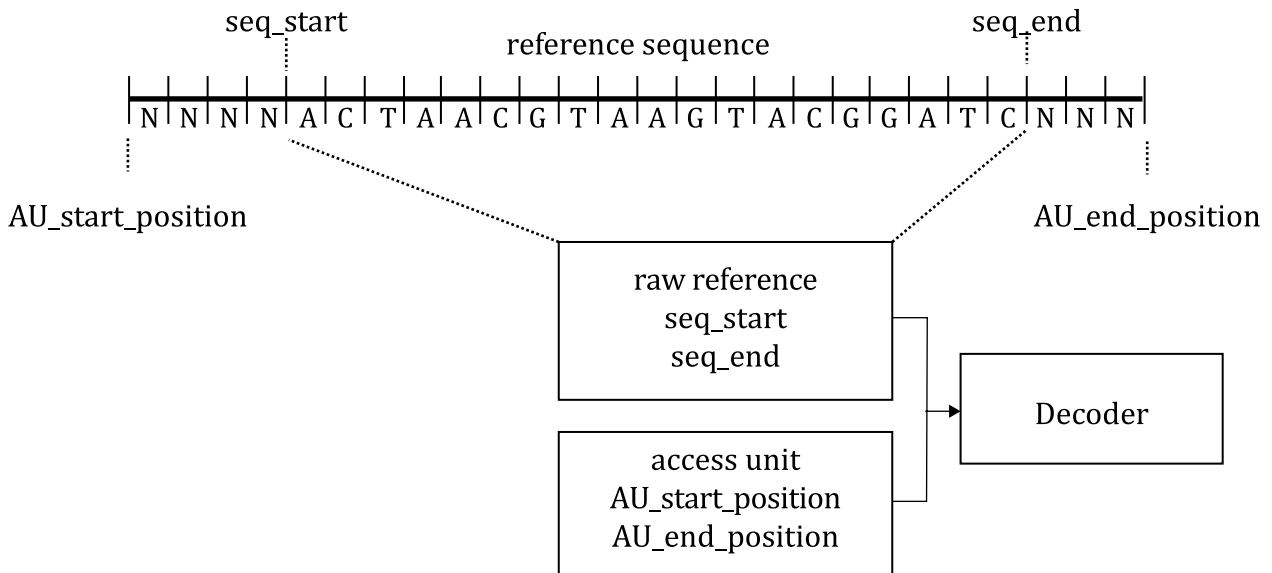


Figure 5 — Padding process for a reference sequence

10.2.3 Type 1 AU (Class P)

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

At the beginning of the decoding process of each AU the variable `msar_k` is initialized to 0.

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

- Set a `classId` variable equal to the value of **AU_type** as specified in [subclause 7.5.1.2](#).
- Decode the variables `numberOfRecordSegments`, `numberOfAlignedRecordSegments`, `numberOfMappedRecordSegments` and `unpairedRead` as specified in [subclause 10.4.10](#).
- Compute the arrays `softClips[][]`, `softClipSizes[][]` and `hardClips[][]` as specified in [subclause 10.4.7](#).
- Compute the arrays `readLength[]`, `numberOfSplicedSeg[]`, `splicedSegLength[][]` and `splicedSegMappedLength[][]` as specified in [subclause 10.4.9](#).
- Decode the output variables specified in [subclause 10.4.12](#) containing the alignment and mapping information.
- Decode the **pos** descriptor as specified in [subclause 10.4.2](#).
- Decode the output variables specified in [subclause 10.4.10](#) containing pairing and/or splicing information.
- Decode the **rcomp** descriptor as specified in [subclause 10.4.3](#).
- If **num_groups** specified in [subclause 7.4.2](#) is greater than 0 decode the **rgroup** descriptor as specified in [subclause 10.4.15](#).
- Decode the `readName` variable as specified in [subclause 10.4.17](#).
- If **as_depth** specified in [subclause 7.4.2](#) is greater than 0 decode the **mscore** descriptor as specified in [subclause 10.4.11](#).
- If **multiple_alignments_flag** specified in [subclause 7.4.2](#) is 1 decode the **msar** descriptor as specified in [subclause 10.4.13](#).

m) If present, decode the following optional descriptors:

- i) decode the **flags** descriptor as specified in [subclause 10.4.4](#).
- ii) decode the **qv** descriptor as specified in [subclause 10.4.16](#).

n) If this process is being applied to access units of type 1 (Class P) (i.e., if this process is not being applied to access units of other types as specified in [subclauses 10.2.4](#), [10.2.5](#) and [10.2.6](#)), or if **crps_flag** specified in [Table 7](#) is equal to 1 and **cr_alg_ID** specified in [Table 17](#) is equal to 2, 3, or 4 and the value of **rtype** descriptor specified in [Table 67](#) is equal to 1, decode the read sequences as specified in [subclause 10.5.2](#).

10.2.4 Type 2 AU (Class N)

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

Additional inputs to this process are

- the array `splicedSequence[][]` specified in [subclause 10.5.1](#)
- the `mismatchOffsets[][]` and `numMismatches[]` arrays specified in [subclause 10.4.5](#)

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

The substitutions are applied as specified in [Table 41](#).

Table 41 — Sequence decoding process for class N

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

10.2.5 Type 3 AU (Class M)

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

Additional inputs to this process are

- the `mismatchOffsets[][]`, `numMismatches[]` arrays specified in [subclause 10.4.5](#);
- the `mismatches[][]` arrays specified in [subclauses 10.4.6](#).

The substitutions are applied as specified in [Table 42](#).