International Standard

ISO/IEC 23092-1

Information technology — Genomic information representation —

Part 1:
Transport and storage of genomic information

Technologie de l'information — Représentation des informations génomiques —

Partie 1: Transport et stockage des informations génomiques

Third edition
2025-01

Reference number
ISO/IEC 23092-1:2025(en)

Contents

Page

Foreword v
Introduction vii
1 Scope 1
2 Normative references 1
3 Terms and definitions 1
4 Conventions 4
4.1 Operators and functions 4
4.1.1 Arithmetic operators 4
4.1.2 Logical operators 4
4.1.3 Relational operators 4
4.1.4 Bitwise operators 4
4.1.5 Assignment operators 5
4.1.6 String/Character functions and operator 5
4.1.7 Data structure function and operator 5
4.1.8 Mathematical functions 5
4.1.9 Array operation functions 5
4.2 Syntax and semantics 6
4.2.1 Method of specifying syntax in tabular form 6
4.2.2 Bit ordering 6
4.2.3 Specification of syntax functions 6
4.2.4 Processes 7
5 Structure of coded genomic data 7
5.1 Genomic sequencing data record 7
5.2 Genomic annotation data records 8
5.3 Data classes 9
5.4 Access units 10
5.5 Datasets 10
5.6 Annotation data tile 11
5.7 Annotation tables 11
5.8 Annotation access units 11
5.9 Selective access 12
6 Data format 12
6.1 Format structure 12
6.1.1 General 12
6.1.2 Box order 17
6.2 Syntax for representation 18
6.3 Output data unit 19
6.4 Data structures common to file format and transport format 20
6.4.1 File header 20
6.4.2 Dataset group 20
6.4.3 Dataset 29
6.4.4 Access unit 40
6.4.5 Block 46
6.4.6 Annotation Table 47
6.4.7 Attribute Group 57
6.4.8 Annotation access unit 59
6.4.9 AAU block 63
6.5 Data structures specific to file format 64
6.5.1 General 64
6.5.2 Indexing 64
6.5.3 Descriptor stream 74
6.5.4 Offset 76
6.6 Data structures specific to transport format 77

6.6.1 General ... 77
6.6.2 Data streams ... 77
6.6.3 Dataset mapping table list ... 77
6.6.4 Dataset mapping table ... 78
6.6.5 Packet ... 80
6.7 Reference procedures to convert transport format to file format ... 81
6.7.1 Procedure for genomic sequencing data ... 81
6.7.2 Procedure for genomic annotation data ... 83

7 String indexing technologies ... 87
7.1 Master string index ... 87
7.1.1 General ... 87
7.1.2 Syntax ... 87
7.1.3 Master String Index Header ... 87
7.1.4 String index ... 88
7.1.5 Compressed string index ... 90
7.2 Decoding and querying processes ... 96
7.2.1 String index payload ... 96
7.2.2 Helper functions ... 97
7.2.3 Substring decoding process ... 98
7.2.4 Suffix array lookup process ... 99
7.2.5 Inverse suffix array process ... 99
7.2.6 Character decoding process ... 100
7.2.7 LF-mapping process ... 101
7.2.8 Extended LF-mapping process ... 101
7.2.9 Substring position search process ... 102
7.2.10 Searching for substring positions with the string index ... 103
7.2.11 Decoding a subset of the string index ... 104
7.2.12 Decoding all the strings of a specific annotation data tile ... 104
7.2.13 Retrieving whole strings with the string index ... 106
7.2.14 Retrieving data tile index(es) associated with a position and record indexes ... 107

8 Indexing for numeric range searches ... 110
8.1 B-Tree indexing ... 110
8.1.1 General ... 110
8.1.2 Syntax ... 110
8.1.3 Semantics ... 111

Annex A (informative) **IETF RFC 3986 specification summary ... 112**

Annex B (informative) **Selective access strategies for genomic sequencing data ... 113**

Annex C (informative) **Selective access strategies for genomic annotation data ... 116**

Annex D (informative) **Depacketization process ... 132**

Annex E (informative) **Efficient handling of symmetric annotation data ... 135**

Bibliography ... 137

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 23092-1:2020), which has been technically revised.

The main changes are as follows:

— Updates to the overall hierarchy of data structures and box order in subclause 6.1

— Extensions for the transport and storage of genomic annotation data, in addition to genomic sequencing data, in support of ISO/IEC 23092-6:2023 specifications while maintaining backward compatibility, which include:

  — An overview of genomic annotation data records in subclause 5.2, with detailed formats specified in Part 6

  — Basic annotation table information in dataset header (as specified in subclause 6.4.3.2) and annotation encoding parameters in dataset parameter set (as specified in subclause 6.4.3.7)

  — Additional data structures such as annotation table (atcn, as specified in subclause 6.4.6), attribute group (agcn, as specified in subclause 6.4.7), annotation access unit (aauc, as specified in subclause 6.4.8), AAU block (as specified in subclause 6.4.9), attribute data byte offset (adbo, as specified in subclause 6.5.2.3) and annotation table index (atix, as specified in subclause 6.5.2.4)

  — The reference procedure for conversion from transport format to file format for genomic annotation data in subclause 6.7.2

  - Data structure for B-Tree indexing (as specified in subclause 8.1) and selective access strategies for genomic annotation data (as specified in Annex C)
- Extensions in support of ISO/IEC 23092-3:2022 which include:
  - New container boxes for metrics metadata: DT_metrics (dtmt, as specified in subclause 6.4.3.4) and AU_metrics (aumt, as specified in subclause 6.4.4.5), containing statistical information (with detailed formats specified in Part 3), which allows for fast and direct extraction of statistics associated with the dataset and access unit content
  - New container boxes for clinical data linkage (CDL) metadata: DG_CDL (dgcd, as specified in subclause 6.4.2.7), DT_CDL (dtcd, as specified in subclause 6.4.3.5) and AT_CDL (atcd, as specified in subclause 6.4.6.4), for establishing linkages to external data sources, which enables access to the clinical data of individual samples
- The inclusion of FM-Index-based entropy coding algorithm (as specified in Clause 7), which provides string search capabilities in the compressed domain

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 sequence read, or a paired sequence 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 is a too small unit to allow effective and fast information retrieval.

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

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

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

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

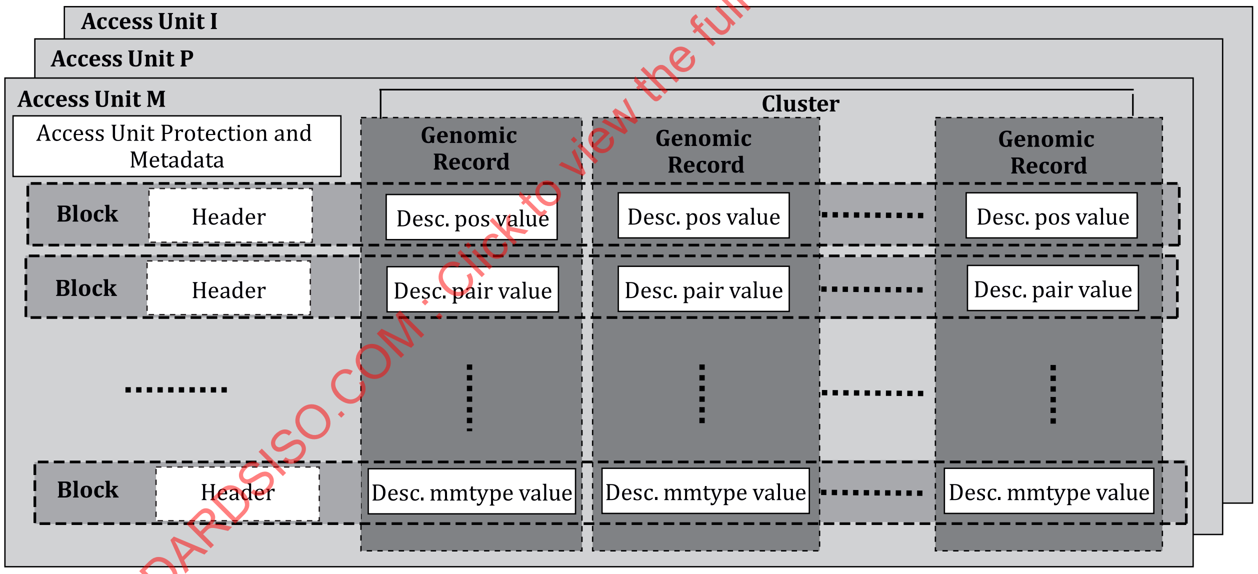


Figure 1 — Access units, blocks and genomic records

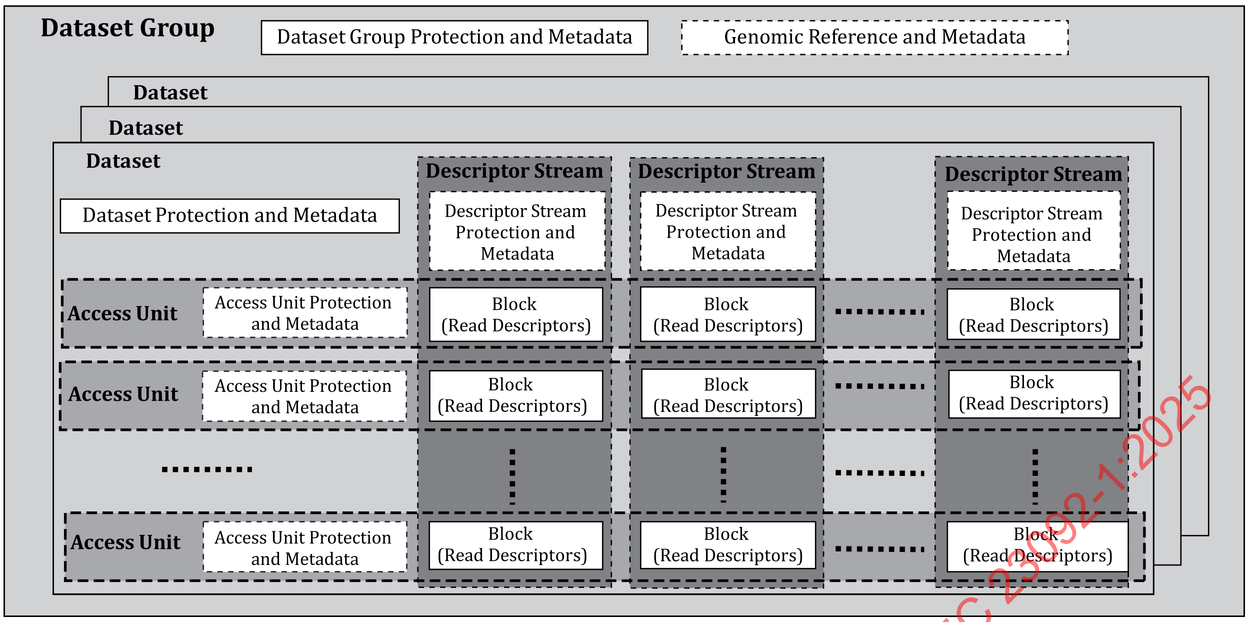

Dataset Group
Dataset Group Protection and Metadata
Genomic Reference and Metadata
Dataset
Dataset
Dataset
Dataset Protection and Metadata
Descriptor Stream
Descriptor Stream Protection and Metadata
Access Unit
Access Unit Protection and Metadata
Block (Read Descriptors)

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.

A simplified diagram of the dataset decoding process is shown in Figure 3.

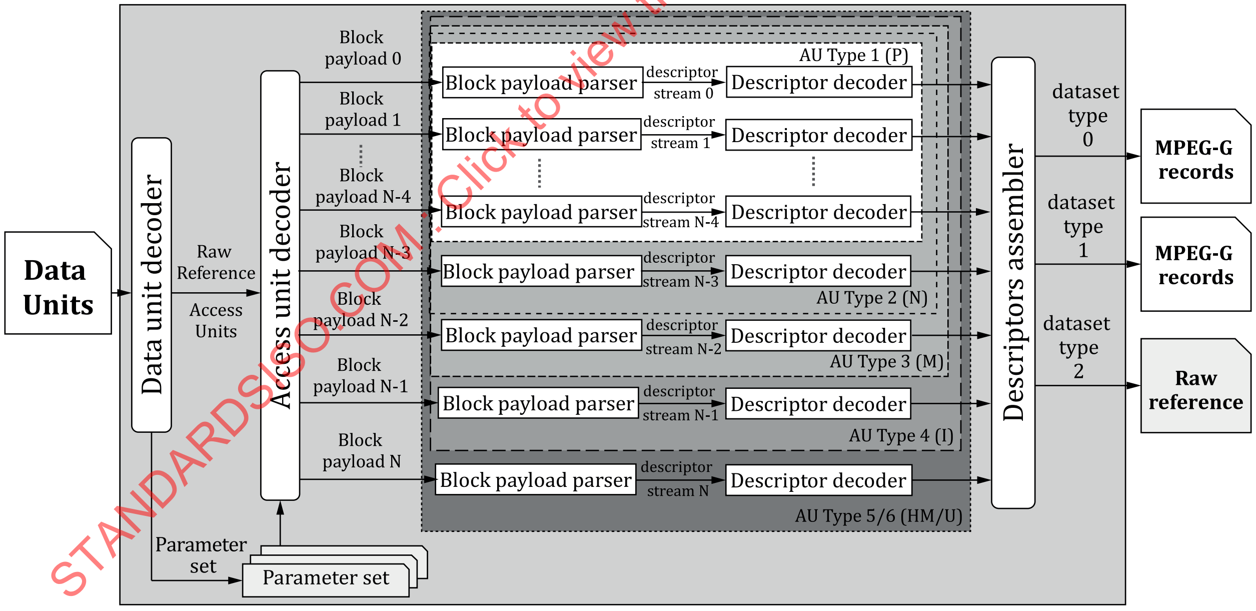


Figure 3 — Decoding process

This document defines the syntax and semantics of the data formats for both transport and storage of genomic information. According to this document, 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 units, to further optimize the selective access performance to the type of data access required by the different application scenarios. This document further provides a reference process to convert a transport stream into a file format and vice versa.

Information technology — Genomic information representation —

Part 1: Transport and storage of genomic information

1 Scope

This document specifies data formats for both transport and storage of genomic information, including the conversion process.

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 23092-2, *Information technology — Genomic information representation — Part 2: Coding of genomic information*

ISO/IEC 23092-3, *Information technology — Genomic information representation — Part 3: Metadata and application programming interfaces (APIs)*

ISO/IEC 23092-6, *Information technology — Genomic information representation — Part 6: Coding of genomic annotations*

IETF RFC 3986, *Uniform Resource Identifier (URI): Generic Syntax*

IETF RFC 7320, *URI Design and Ownership*

3 Terms and definitions

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

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

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

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

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

3.2
access unit covered region
genomic range comprised between the access unit start position and the access unit end position, inclusive

3.3
access unit start position
position of the left-most mapped base among the first alignments of all genomic records contained in the access unit, irrespective of the strand

3.4
access unit end position
position of the right-most mapped base among the first alignments of all genomic records contained in the access unit, irrespective of the strand

3.5
access unit range
genomic range comprised between the access unit start position and the right-most genomic record position among all genomic records contained in the access unit

3.6
alignment
information describing the similarity between a sequence (typically a sequencing read) and a reference sequence (for instance, a reference genome)

3.7
box
object-oriented building unit defined by a unique type identifier and length

3.8
cluster
aggregation of genomic records

3.9
cluster signature
signature
sequence of nucleotides that is common to most or all genomic records belonging to a cluster

3.10
container box
box (3.8) whose sole purpose is to contain and group a set of related boxes

3.11
data stream
set of *packets* (3.20) transporting the same data type

3.12
extended access unit start position
position of the left-most mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand

3.13
extended access unit end position
position of the right-most mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand

3.14
file format
set of data structures for the storage of coded information

3.15
genomic position
position
integer number representing the zero-based position of a nucleotide within a reference sequence

3.16
genomic region
region
genomic interval between a start nucleotide position and an end nucleotide position, inclusive

3.17
genomic range
range
interval of positions on a reference sequence defined by a start position s and an end position e such that $s \leq e$; the start and the end positions of a genomic range are always included in the range

3.18
mapped base
base of the aligned read that either matches the corresponding base on the reference sequence or can be turned into the corresponding base on the reference sequence via a substitution

3.19
packet
transmission unit transporting segments of any of the data structures defined in this document

3.20
reference genome
representative example of the sequences for a species' genetic material

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

3.21
reference sequence
nucleic acid sequence with biological relevance

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

3.22
genomic segment
segment
contiguous sequence of nucleotides, typically output of the sequencing process and sequenced from one strand of a template

3.23
sequence read
read
readout, by a specific technology more or less prone to errors, of a continuous part of a nucleic acid molecule extracted from an organic sample

3.24
syntax field
element of data represented in the data format

3.25
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.

3.26
transport format
set of data structures for the transport of coded information

3.27
variable
parameter either inferred from syntax fields or locally defined in a process description

4 Conventions

4.1 Operators and functions

NOTE The operators used in this document are similar to those used in the C programming language. However, integer division with truncation and rounding are specifically defined. The bitwise operators are defined assuming two's-complement representation of integers. Numbering and counting loops generally begin from 0.

4.1.1 Arithmetic operators

+	addition
−	subtraction (as a binary operator) or negation (as a unary operator)
*	multiplication
/	integer division with truncation of the result toward 0 (for example, 7/4 and −7/−4 are truncated to 1 and −7/4 and 7/−4 are truncated to −1)

4.1.2 Logical operators

||	logical OR
&&	logical AND
!	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.

4.1.3 Relational operators

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

4.1.4 Bitwise operators

&	AND
|	OR
>>	shift right with sign extension
<<	shift left with 0 fill

4.1.5 Assignment operators

The following arithmetic operators are defined as follows:

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

4.1.6 String/Character functions and operator

+	string concatenation, when applied on variables of string or character types

4.1.7 Data structure function and operator

sizeof(N)	size in bytes of N, where N is either a data structure or a data type
N->m	returns the data field m defined within the data structure N

4.1.8 Mathematical functions

The following mathematical functions are defined:

Ceil(x)	the smallest integer greater than or equal to x
Floor(x)	the largest integer less than or equal to x
Log(x)	the base-e logarithm of x
Log2(x)	the base-2 logarithm of x
Min(x, y)	$\begin{cases} x & ; \quad x <= y \\ y & ; \quad x > y \end{cases}$
Max(x, y)	$\begin{cases} x & ; \quad x >= y \\ y & ; \quad x < y \end{cases}$

4.1.9 Array operation functions

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

Max(x[]) returns the maximum value in a numeric array x[].

Find(x[], v) returns the index of the first element in array x[] that is equal to the value v. It returns -1 if v is not found.

4.2 Syntax and semantics

4.2.1 Method of specifying syntax in tabular form

Table 1 lists the constructs that are used to express the conditions when data elements are present.

NOTE This syntax uses the convention that a variable or expression evaluating to a non-zero value is equivalent to a condition that is true.

Table 1 — Constructs used to express the conditions when data elements are present

Construct	Description

`if (condition) {`
`    data_element`
`    . . .`
`}`
	`If the condition is true, then the first group of data elements occurs next in the bitstream.`

`else {`
`    data_element`
`    . . .`
`}`
	`If the condition is not true, then the second group of data elements occurs next in the bitstream.`

`for (i=0; i<n; i++) {`
`    data_element`
`    . . .`
`}`
	`The group of data elements occurs n times. Conditional constructs within the group of data elements may depend on the value of the loop control variable i, which is equal to zero for the first occurrence, incremented to 1 for the second occurrence, and so forth.`

As noted, the group of data elements may contain nested conditional constructs. For compactness, the {} are omitted when only one data element follows. Collections of data elements are represented as listed in Table 2.

Table 2 — Syntax used to represent collections of data elements

Syntax	Description
`data_element[]`	`data_element[] is an array of data. The number of data elements is indicated by the semantics.`
`data_element[n]`	`data_element[n] is the n`th` element of an array of data.`
`data_element[m][n]`	`data_element[m][n] is the m`th`, n`th` element of a two-dimensional array of data.`
`data_element[l][m][n]`	`data_element[l][m][n] is the l`th`, m`th`, n`th` element of a three-dimensional array of data.`

4.2.2 Bit ordering

The bit order of syntax fields in the syntax tables is specified to start with the most significant bit (MSB) and proceed to the least significant bit (LSB).

4.2.3 Specification of syntax functions

byte_aligned() is specified as follows:

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

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

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

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

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

— 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.

— 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. The maximum value of stringLength is 16384.

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

4.2.4 Processes

Processes are used to describe the sequence of operations on syntax elements (see example in Table 3). A process has a separate specification and unique invocation. All syntax elements and variables that pertain to the current syntax structure and dependent syntax structures are available in the process specification and at the time of invocation. A process specification may also have a lower-case variable explicitly specified as input. Each process specification output is explicitly defined. The output of a process can be assigned to one or many variables as specified in the return clause of the process.

Table 3 — Example decoding process

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

5 Structure of coded genomic data

5.1 Genomic sequencing data record

The genomic sequencing data record, in this document, is a data structure consisting of either a single sequence read, or paired sequence reads, and its associated sequencing and alignment information. The genomic record may contain detailed mapping and alignment data, a single or paired read identifier (read name) and quality values.

When alignment information is present, the genomic record position is defined as the position of the left-most mapped base of the genomic record on the reference genome. Genomic record positions are 0-based in the ISO/IEC 23092 series. In case of multiple alignments, the position of the first alignment in the record is considered; in such a case, the first alignment shall be the one with the leftmost position among all the alignments with the best score.

In case of unmapped reads (i.e. no alignment information present) the notion of position does not apply to the genomic record.

In case of aligned content, bases that are present in the reads of the genomic record and not present in the reference sequence (*insertions*) and bases preserved by the alignment process but not mapped on the reference sequence (*soft clips*) do not have mapping positions.

Table 4 enumerates all the types of data that a genomic record can contain. ISO/IEC 23092-2 defines technology that allows coding all and only those types of data into a set of descriptors; data, and consequently descriptors, which are mandatory or optional, are also specified in ISO/IEC 23092-2, as well as how they are used to represent multiple alignments.

Table 4 — Genomic sequencing data records

Data	Semantics
Record identifier	name of the record (e.g. read names)
Sequence reads	sequencing readout, as one or more strings of bases
Quality values	quality scores of the sequence reads
Strandedness	information about the strandedness of each read of the record
Length	length of the sequence reads
Position	position on the reference genome of the left-most mapped genomic record base
Pairing	position or distance of the mate reads (e.g. in a pair)
Flags	technical, additional alignment information (duplicates, proper pairs, failures)
Mismatches	information about position and type of each mismatch in mapped records
Clips	information about clipped bases (soft and hard clips) in mapped records
Mapping scores	mapping scores for an alignment
Multiple alignments	information about the number of alignments and the alternative alignment information about each segment of the record
Group	read group the genomic record belongs to

ISO/IEC 23092-2 defines an output record format for all types of data in Table 4. These records shall be generated by decoders compliant to ISO/IEC 23092-2 as output of the decoding process.

5.2 Genomic annotation data records

Different types of genomic annotation data records are associated with each annotation table type (AT_type, as specified in subclause 6.4.6.2). They are encoded and stored in attribute groups of different classes (AG_class, as specified in subclause 6.4.7.2) in an annotation table. Table 5 summarizes the annotation table types and their associated data records.

Table 5 — Genomic annotation data records

Annotation table type	Attribute group class	Record type	Record content
Variants	1	Variant site record	Variant site information including chromosome, position, reference allele, alternative allele, variant ID, quality value and other relevant attributes
	0	Variant genotype record	Variant genotype values for individual samples and any associated metrics
	2	Sample record	Sample information such as sample ID, demographics and clinical data
Functional annotations	0	Functional annotation record	Any number of ontological labels and attributes associated with a genomic feature specified by its genomic range, strand, name and ID
	5	Track property record	Track property containing display attributes associated with a block of functional annotation records referenced by the index of the first record in the block
Gene expression	0	Expression record	Expression values associated with a genomic feature, such as a gene or an exon, for individual samples
	1	Feature record	Feature information such as gene name and chromosome position
	2	Sample record	Sample information such as sample ID, demographics and clinical data
Contact matrices	0	Contact matrix record	The frequency of contact, measured by counts or any number of normalized values, in 3D space between any two genomic regions in a pair of the same or different chromosomes.
Tracks	0	Track data record	Any number of attributes associated with a genomic range and strand
	5	Track property record	Track property containing display attributes associated with a block of functional annotation records referenced by the index of the first record in the block

ISO/IEC 23092-6 defines the formats of the above output records. These records shall be generated by decoders compliant to ISO/IEC 23092-6 as output of the decoding process.

5.3 Data classes

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

In the case of more than one read in a template, 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 6.

Table 6 — Data classes

Class ID	Class name	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.

Genomic records of each data class are coded by means of several descriptors; conversely, a descriptor is a coding element needed to represent part of the information. Descriptors for each data class are specified in ISO/IEC 23092-2.

Descriptors are coded in blocks. Blocks are defined in subclause 6.4.5. A sequence of block payloads of a single descriptor composes a descriptor stream. All block payloads in a descriptor stream contain compressed descriptors of a single type representing reads of the same data class.

5.4 Access units

Access units (AUs) are data structures containing a coded representation of genomic sequencing data. Related metadata may be available to facilitate bitstream access and manipulation. An access unit contains either genomic sequencing records belonging to the same data class or a fragment of a reference sequence.

The access unit is the smallest data organization that can be decoded by a decoder compliant with ISO/IEC 23092-2.

Access units are orthogonal to descriptor streams: an access unit is composed of all and only those blocks of the descriptor streams that are necessary to decode the information contained in a cluster of records of a given data class.

An access unit can be of several types, as specified in Table 7, according to the class of the coded data.

Table 7 — Access unit type

Access unit type		Class of data
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

Depending on the type of coded information, an access unit can be decoded either independently of any other access unit or using information contained in other access units.

5.5 Datasets

A dataset is a data structure containing a header, optional metadata, access units for genomic sequencing data, and annotation tables for genomic annotation data. The set of access units or annotation tables composing the dataset constitutes the dataset payload.

One or more datasets are assembled into a dataset group.

5.6 Annotation data tile

An annotation data tile specifies a rectangular region of descriptor/attribute data corresponding to contiguous ranges of rows and columns (if data is two-dimensional) of an annotation table. Instructions on how to divide descriptor/attribute data into tiles are specified in tile configurations. Each data tile per descriptor/attribute is individually encoded and stored in an annotation access unit payload block. Selective access to data in specific regions of interest is achieved by identifying and decoding only the data tiles that overlap with the regions of interest.

5.7 Annotation tables

An annotation table is a container for genomic annotation data of a designated annotation table type and subtype (the associated external file format), as specified in Table 8 and Table 9. It contains a header, optional metadata, tile configurations (for specifying how data shall be divided into tiles to be individually encoded), annotation table index (for supporting random access), and attribute groups. Each attribute group has a designated class that indicates the role of the descriptor/attribute data it contains. For example, an attribute group of class 0 contains main table data (such as variant genotypes), whereas class 1 contains auxiliary data associated with the rows of the main data (such as variant site information), and class 2 contains auxiliary data associated with the columns of the main data (such as sample information).

Table 8 — Annotation table type

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

Table 9 — Annotation table sub-type

Annotation table type name	Annotation table sub-type	
	Annotation table sub-type name	Value
Any	UNDEFINED	0
VARIANTS	VCF	1
FUNCTIONAL_ANNOTATIONS	GTF	2
	GFF	3
	GENBANK	8
TRACKS	BED	4
	BEDGRAPH	5
	WIG	6
	BIGWIG	7
CONTACT_MATRICES	HIC	10
GENE_EXPRESSION	GENE_EXPRESSION	9

5.8 Annotation access units

An annotation access unit contains individually encoded payload blocks of genomic annotation data. Each payload block corresponds to a tile of data of a descriptor/attribute from specific ranges of rows and columns (if data is two-dimensional) in an annotation table. It supports two contiguity modes:

— Attribute contiguity, where each annotation access unit corresponds to a descriptor/attribute and each payload block within corresponds to a tile location in an annotation table, and

— Tile contiguity, where each annotation access unit corresponds to a tile location in an annotation table and each payload block within corresponds to a descriptor/attribute.

The annotation access unit is the smallest data organization handled by a decoder compliant with ISO/IEC 23092-6.

5.9 Selective access

For genomic sequencing data:

— In the case of selective access to a genomic region comprised between a *start* genomic position and an *end* genomic position, the decoder shall return: a) all the access units whose covered region overlaps the region defined by *start* and *end* with at least one base, and the parameter sets that are needed to decode them; b) at least the reference portion that is necessary to decode the access units identified in a).

— In the case of selective access to signed content identified by a U_cluster_signature signature the decoder shall return all the access units whose signature corresponds to U_cluster_signature, and the parameter sets that are needed to decode them.

— Examples of selective access strategies are described in Annex B.

For genomic annotation data:

— In the case of selective access to specific descriptor and attribute data in an annotation table region defined by ranges of indexes and/or genomic positions, the decoder shall identify the data tiles that overlap the region based on the relevant attribute data tile structure(s) as specified in subclause 6.4.6.7 and/or genomic range index(es) as specified in subclause 6.5.2.4.4, and return

a) if `attribute_contiguity == 0`, annotation access units corresponding to the identified data tiles and containing only the payload blocks of the selected descriptor(s) and attribute(s)

b) if `attribute_contiguity == 1`, annotation access units corresponding to the selected descriptor(s) and attribute(s) and containing only the payload blocks of the identified data tiles

c) the annotation parameter set(s) required for decoding the annotation access units identified in a) or b)

— Examples of selective access strategies are described in Annex C.

6 Data format

6.1 Format structure

6.1.1 General

Table 10 presents the overall data structures and hierarchical encapsulation levels.

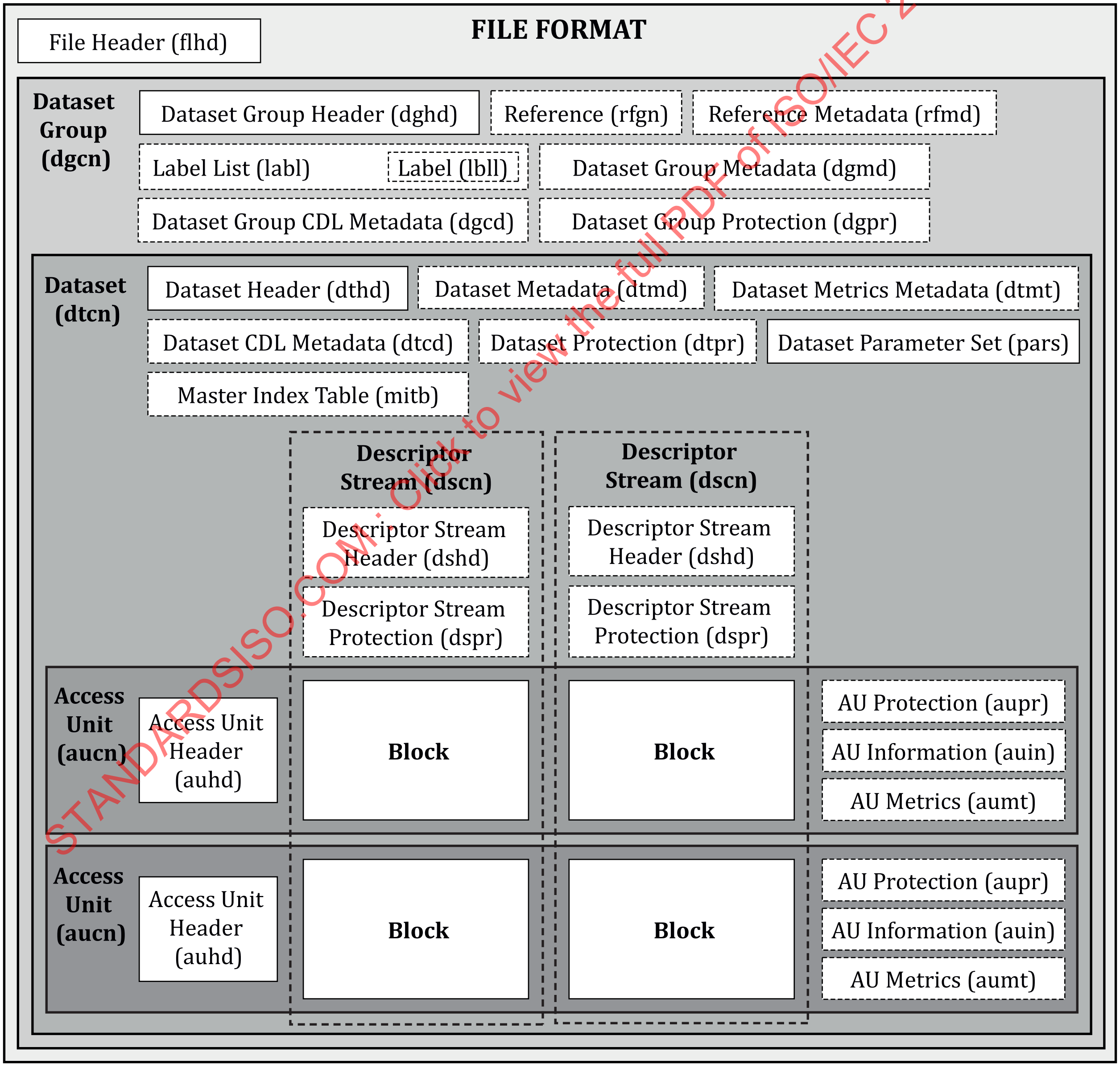
Boxes that may occur at the top-level are shown in the left-most column; indentation is used to show possible containment. Not all boxes need be used in all files; the mandatory boxes for genomic sequencing data, genomic annotation data or both are marked respectively with a plus sign (+), pound sign (#) or an asterisk (*) in the *Mandatory* column: such column refers to the relevant scope (File and/or Transport). Optional boxes are represented with dashed borders in Figure 4-7. Mandatory boxes are represented with solid borders. When no entry is present in the *Scope* column, scope is both *File* and *Transport*. See the specification of each individual box for the assumptions when the optional boxes are not present. If the box key is represented in *italic* format in Table 10, the corresponding box is represented either with no Key and no Length, but only Value in the gen_info format, as specified in subclause 6.2, for all boxes but offset, or as specified in subclause 6.5.4.1 for the offset box.

Table 10 — Format structure and encapsulation levels

Box key (with hierarchical level)							Subclause	Scope	Mandatory
flhd							6.4.1		*
dgcn							6.4.2	File	*
	dghd						6.4.2.2		*
	rfgn						6.4.2.3		
	rfmd						6.4.2.4		
	labl						6.4.2.5		
	lbll						6.4.2.5.4		
	dgmd						6.4.2.6		
	dgcd						6.4.2.7		
	dgpr						6.4.2.8		
	dmtl						6.6.3.1	Transport	*
	dtcn						6.4.3	File	*
		dthd					6.4.3.2		*
		dtmd					6.4.3.3		
		dtmt					6.4.3.4		
		dtcd					6.4.3.5		
		dtpr					6.4.3.6		
		pars					6.4.3.7		*
		mitb					6.5.2.1	File	
		msix					7.1.2		
			mshd				7.1.3		
			ssix				7.1.4		
				csix			7.1.5		
		dmtb					6.6.4	Transport	*
		dscn					6.5.3	File	
			dshd				6.5.3.2	File	
			dspr				6.5.3.3	File	
		aucn					6.4.4		+
			auhd				6.4.4.3		+
			auin				6.4.4.4		
			aumt				6.4.4.5		
			aupr				6.4.4.6		
			block				6.4.5		+
				block_header			6.4.5.2		
		atcn					6.4.6	File	#
			athd				6.4.6.2		#
			atmd				6.4.6.3		
			atcd				6.4.6.4		
			atpr				6.4.6.5		
			agtc				6.4.6.6		#
				adts			6.4.6.7		#
				adbo			6.5.2.3	File	#
			atix				6.5.2.4	File	
				grix			6.5.2.4.4	File	

Table 10 *(continued)*

Box key (with hierarchical level)							Subclause	Scope	Mandatory
				avix			6.5.2.4.5	File	
			agcn				6.4.7	File	#
				aghd			6.4.7.2		#
				aauc			6.4.8		#
					aauh		6.4.8.3		#
					aaup		6.4.8.4		
					aau_block		6.4.9		#
						aau_block_ header	6.4.9.2		#
	offset	*offset*					6.5.4	File	
packet							6.6.5	Transport	*
	packet_header						6.6.5.2	Transport	*



FILE FORMAT
File Header (flhd)
Dataset Group (dgcn)
Dataset Group Header (dghd)
Reference (rfgn)
Reference Metadata (rfmd)
Label List (labl)
Label (lbll)
Dataset Group Metadata (dgmd)
Dataset Group CDL Metadata (dgcd)
Dataset Group Protection (dgpr)
Dataset (dtcn)
Dataset Header (dthd)
Dataset Metadata (dtmd)
Dataset Metrics Metadata (dtmt)
Dataset CDL Metadata (dtcd)
Dataset Protection (dtpr)
Dataset Parameter Set (pars)
Master Index Table (mitb)
Descriptor Stream (dscn)
Descriptor Stream Header (dshd)
Descriptor Stream Protection (dspr)
Descriptor Stream (dscn)
Descriptor Stream Header (dshd)
Descriptor Stream Protection (dspr)
Access Unit (aucn)
Access Unit Header (auhd)
Block
Block
AU Protection (aupr)
AU Information (auin)
AU Metrics (aumt)
Access Unit (aucn)
Access Unit Header (auhd)
Block
Block
AU Protection (aupr)
AU Information (auin)
AU Metrics (aumt)

Figure 4 — Data structures hierarchy for storage of genomic sequencing data

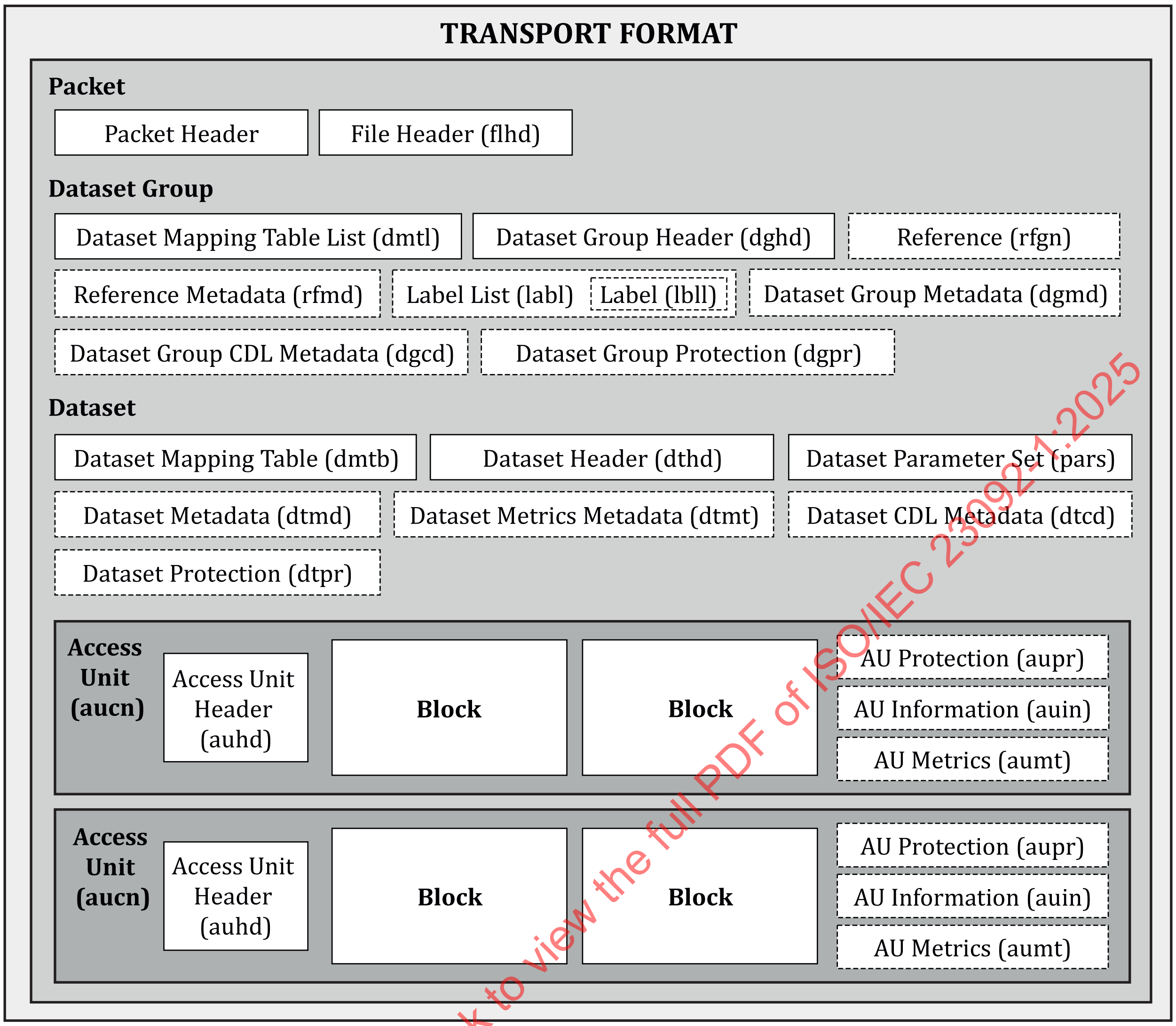

TRANSPORT FORMAT
Packet
Packet Header
File Header (flhd)
Dataset Group
Dataset Mapping Table List (dmtl)
Dataset Group Header (dghd)
Reference (rfgn)
Reference Metadata (rfmd)
Label List (labl)
Label (lbll)
Dataset Group Metadata (dgmd)
Dataset Group CDL Metadata (dgcd)
Dataset Group Protection (dgpr)
Dataset
Dataset Mapping Table (dmtb)
Dataset Header (dthd)
Dataset Parameter Set (pars)
Dataset Metadata (dtmd)
Dataset Metrics Metadata (dtmt)
Dataset CDL Metadata (dtcd)
Dataset Protection (dtpr)
Access Unit (aucn)
Access Unit Header (auhd)
Block
Block
AU Protection (aupr)
AU Information (auin)
AU Metrics (aumt)
Access Unit (aucn)
Access Unit Header (auhd)
Block
Block
AU Protection (aupr)
AU Information (auin)
AU Metrics (aumt)

Figure 5 — Data structures hierarchy for transport of genomic sequencing data

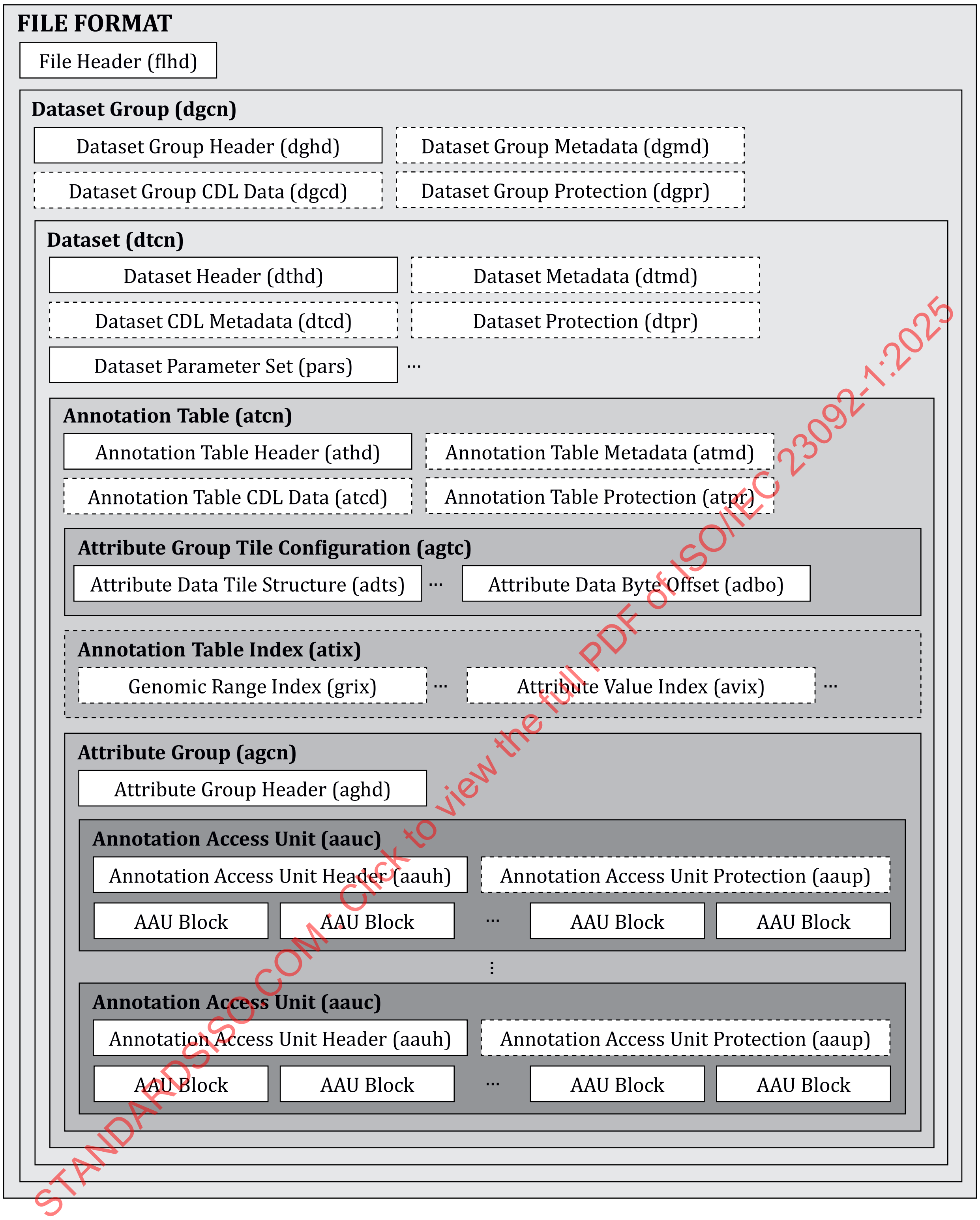

FILE FORMAT
File Header (flhd)
Dataset Group (dgcn)
Dataset Group Header (dghd)
Dataset Group Metadata (dgmd)
Dataset Group CDL Data (dgcd)
Dataset Group Protection (dgpr)
Dataset (dtcn)
Dataset Header (dthd)
Dataset Metadata (dtmd)
Dataset CDL Metadata (dtcd)
Dataset Protection (dtpr)
Dataset Parameter Set (pars)
...
Annotation Table (atcn)
Annotation Table Header (athd)
Annotation Table Metadata (atmd)
Annotation Table CDL Data (atcd)
Annotation Table Protection (atpr)
Attribute Group Tile Configuration (agtc)
Attribute Data Tile Structure (adts)
...
Attribute Data Byte Offset (adbo)
Annotation Table Index (atix)
Genomic Range Index (grix)
...
Attribute Value Index (avix)
...
Attribute Group (agcn)
Attribute Group Header (aghd)
Annotation Access Unit (aauc)
Annotation Access Unit Header (aauh)
Annotation Access Unit Protection (aaup)
AAU Block
AAU Block
...
AAU Block
AAU Block
⋮
Annotation Access Unit (aauc)
Annotation Access Unit Header (aauh)
Annotation Access Unit Protection (aaup)
AAU Block
AAU Block
...
AAU Block
AAU Block

Figure 6 — Data structures hierarchy for storage of genomic annotation data

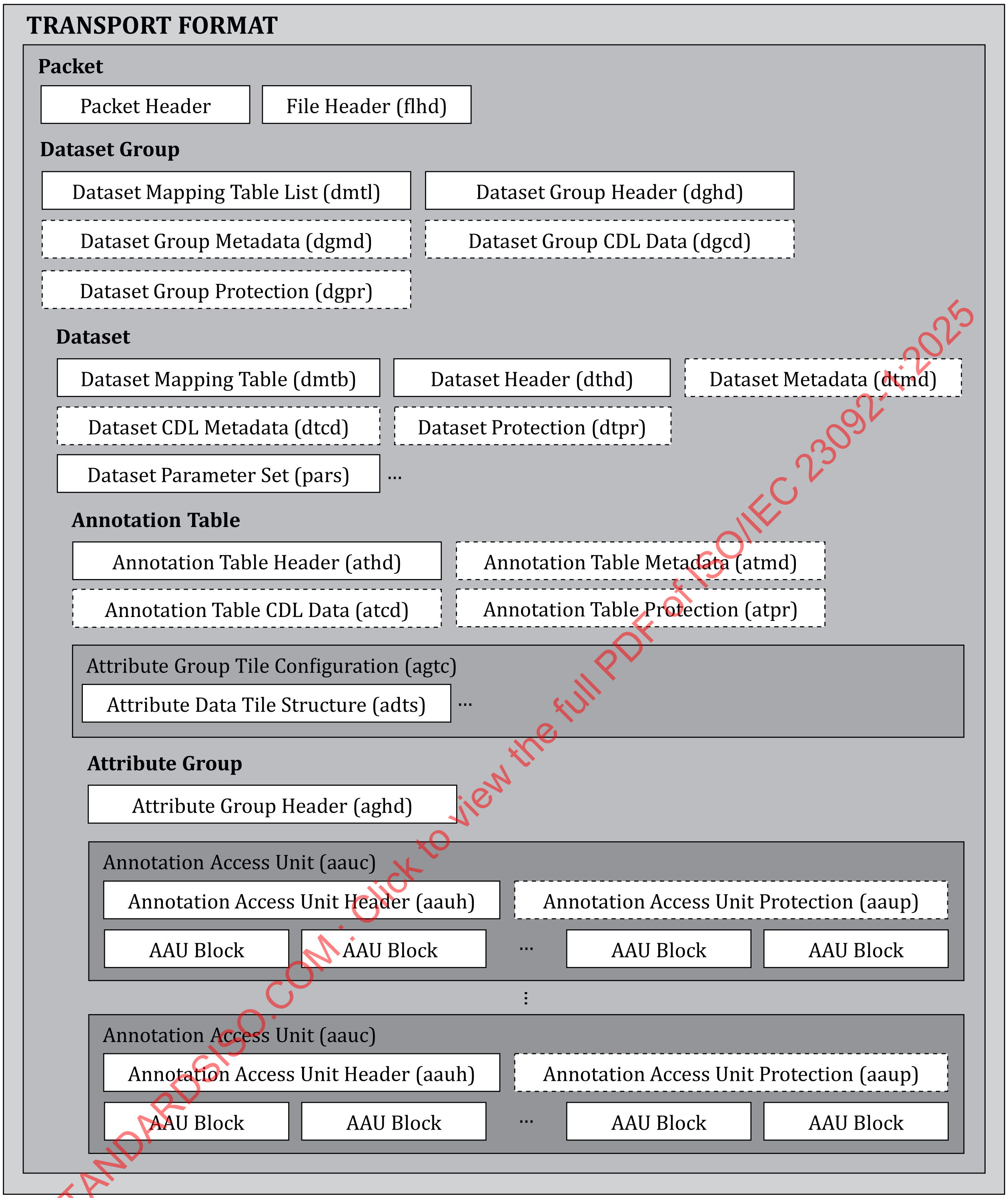

TRANSPORT FORMAT
Packet
Packet Header
File Header (flhd)
Dataset Group
Dataset Mapping Table List (dmtl)
Dataset Group Header (dghd)
Dataset Group Metadata (dgmd)
Dataset Group CDL Data (dgcd)
Dataset Group Protection (dgpr)
Dataset
Dataset Mapping Table (dmtb)
Dataset Header (dthd)
Dataset Metadata (dtmd)
Dataset CDL Metadata (dtcd)
Dataset Protection (dtpr)
Dataset Parameter Set (pars)
...
Annotation Table
Annotation Table Header (athd)
Annotation Table Metadata (atmd)
Annotation Table CDL Data (atcd)
Annotation Table Protection (atpr)
Attribute Group Tile Configuration (agtc)
Attribute Data Tile Structure (adts)
...
Attribute Group
Attribute Group Header (aghd)
Annotation Access Unit (aauc)
Annotation Access Unit Header (aauh)
Annotation Access Unit Protection (aaup)
AAU Block
AAU Block
...
AAU Block
AAU Block
Annotation Access Unit (aauc)
Annotation Access Unit Header (aauh)
Annotation Access Unit Protection (aaup)
AAU Block
AAU Block
...
AAU Block
AAU Block

Figure 7 — Data structures hierarchy for transport of genomic annotation data

In transport format, any box represented in Figure 5 and Figure 7 shall be encapsulated in one or more packets, as specified in subclause 6.6.5. The depacketization process is illustrated in Annex D. The dataset group, dataset, annotation table and attribute group are represented in Figure 5 and Figure 7 for clarity, but the corresponding container boxes (dgcn, dtcn, atcn and agcn) do not exist in transport format.

6.1.2 Box order

In order to improve interoperability, the following rules shall be followed for the order of boxes:

In file format

1) The container boxes:

   — dataset group, dataset, access unit and descriptor stream for genomic sequencing data, and

   — dataset group, dataset, annotation table, attribute group tile configuration, annotation table index, attribute group and annotation access unit for genomic annotation data shall be ordered according to the hierarchy specified in Table 10.

2) The box order inside the containers dgcn, dtcn, dscn, aucn, atcn, agtc, atix, agcn and aauc is specified in Table 13, Table 25, Table 63, Table 33, Table 40, Table 47, Table 59, Table 49 and Table 52 respectively.

3) The file header box ‘flhd’ shall occur before any variable-length box.

4) When present, the offset box ‘offs’, as specified in subclause 6.5.4, enables an indirect addressing of boxes, which, while logically respecting the ordering specified in this subclause, may be physically located in a different position in the file.

5) The contiguity of child boxes inside the containers dgcn, dtcn, dscn, and aucn shall not be broken by any box external to the container box, apart from the offset box, as specified in subclause 6.5.4

In transport format

1) The dataset_mapping_table_list, dataset_mapping_table and file header boxes shall be decoded first, and then all other boxes according to the hierarchy specified in Table 10.

2) It is recommended to transmit the boxes in hierarchical order, as specified in Table 10.

3) It is recommended, when possible, within each dataset group, to interleave boxes with hierarchical level equal to 1 (second column in Table 10), and the dthd and pars boxes belonging to different datasets, before transmitting all other boxes in the corresponding datasets.

4) For genomic annotation data, it is recommended, when possible, within each dataset, to interleave boxes with hierarchical level equal to 2 (third column in Table 10), and the athd, agtc and aghd boxes belonging to different annotation tables, before transmitting all other boxes in the corresponding annotation tables.

6.2 Syntax for representation

KLV (Key Length Value) format is used for all the data structures listed in Table 10 but the block, block_header, offset, packet and packet_header.

The KLV syntax is defined as follows:

```
struct gen_info
{
      c(4)           Key;
      u(64)          Length;
      u(8)           Value[];
}
```

The Length field specifies the number of bytes composing the entire gen_info structure, including all three fields Key, Length and Value.

The block, block_header, packet and packet_header data structures have no Key and no Length, but only Value.

The offset data structure is specified in subclause 6.5.4.

All syntax tables specified in subclauses 6.4, 6.5, and 6.6 and related subclauses, for boxes of type gen_info, represent the internal syntax of the Value[] array field only. In the scope of this document the Value[] array is referred as just Value.

6.3 Output data unit

This subclause specifies the output data unit (see Table 11) of the decapsulation processes specified in subclauses 6.4.2.3, 6.4.3.7, 6.4.4.2, and 6.4.8.2.

Table 11 — Data unit syntax

Syntax	Type	Remarks
`data_unit() {`		
`data_unit_type`	u(8)	
`if (data_unit_type == 0) {`		
`data_unit_size`	u(64)	
`raw_reference()`		As specified in ISO/IEC 23092-2
`}`		
`else if (data_unit_type == 1) {`		
`reserved`	u(10)	
`data_unit_size`	u(22)	
`parameter_set()`		As specified in ISO/IEC 23092-2
`}`		
`else if (data_unit_type == 2){`		
`reserved`	u(3)	
`data_unit_size`	u(29)	
`access_unit()`		As specified in ISO/IEC 23092-2
`}`		
`else if (data_unit_type == 3){`		
`reserved`	u(10)	
`data_unit_size`	u(22)	
`annotation_parameter_set()`		As specified in ISO/IEC 23092-6
`}`		
`else if (data_unit_type == 4){`		
`reserved`	u(3)	
`data_unit_size`	u(29)	
`annotation_access_unit()`		As specified in ISO/IEC 23092-6
`}`		
`else /*(data_unit_type > 4)*/{`		
`/*skip data unit*/`		
`}`		
`}`		

data_unit_type and data_unit_size shall be filled as specified in subclauses 6.4.2.3.6, 6.4.3.7.1, 6.4.4.1, and 6.4.8.1.

6.4 Data structures common to file format and transport format

6.4.1 File header

6.4.1.1 General

This box (*flhd* in Table 12) is mandatory and provides information about the version of this specification and of the set of other specifications the file or stream complies with.

6.4.1.2 Syntax

Table 12 — File header syntax

Syntax	Key	Type	Remarks
file_header {	*flhd*		
major_brand		c(6)	
minor_version		c(4)	
for (i=0; i < num_compatible_brands; i++) {			
compatible_brand[i]		c(4)	
}			
}			

6.4.1.3 Semantics

major_brand is the major brand identifier. The value is equal to the 6-character code “MPEG-G”.

minor_version is an informative set of four characters for the minor version of the major brand of this document and is specified as follows:

— first two bytes: version number, as the last two digits of the year of release of the major brand

— third byte: amendment number, as integer counter from 0 to 9, 0 if no amendment yet

— fourth byte: corrigendum number, as integer counter from 0 to 9, 0 if no corrigendum yet

num_compatible_brands is inferred from the Length field in the file_header *gen_info* header as follows: num_compatible_brands = (Length − 22) / 4.

compatible_brand[i] is a 4-character code representing a compatible brand.

6.4.2 Dataset group

6.4.2.1 General

The dataset group is a collection of one or more datasets.

The relevant container box (*dgcn* in Table 13) is mandatory in file format, forbidden in transport format.

Child boxes may be present or not, according to the column *Mandatory* in Table 10. Child boxes marked with suffix “[]” after their name in the Syntax column of Table 13 may be present in multiple instances.

Table 13 — Dataset group syntax

Syntax	Key	Type	Remarks
dataset_group {	*dgcn*		
dataset_group_header	dghd	gen_info	As specified in subclause 6.4.2.2
reference[]	rfgn	gen_info	As specified in subclause 6.4.2.3
reference_metadata[]	rfmd	gen_info	As specified in subclause 6.4.2.4
label_list	labl	gen_info	As specified in subclause 6.4.2.5
DG_metadata	dgmd	gen_info	As specified in subclause 6.4.2.6
DG_CDL	dgcd	gen_info	As specified in subclause 6.4.2.7
DG_protection	dgpr	gen_info	As specified in subclause 6.4.2.8
for (i=0;i<num_datasets;i++) {			num_datasets: as specified in subclause 6.6.3.2
dataset[i]	dtcn	gen_info	As specified in subclause 6.4.3.1
}			
}			

6.4.2.2 Dataset group header

6.4.2.2.1 General

This is a mandatory box (*dghd* in Table 14) describing the content of a dataset group.

6.4.2.2.2 Syntax

Table 14 — Dataset group header syntax

Syntax	Key	Type	Remarks
dataset_group_header {	*dghd*		
dataset_group_ID		u(8)	
version_number		u(8)	
for (i=0;i<num_datasets;i++) {			
dataset_ID[i]		u(16)	
}			
}			

6.4.2.2.3 Semantics

dataset_group_ID identifies a dataset group. Each value shall be unique among all dataset_group_ID fields in the file or stream.

version_number is the version number of the dataset group. The version number shall be incremented by 1 whenever the definition of the dataset group identified by dataset_group_ID changes. Upon reaching the value 255, it wraps around to 0.

dataset_ID[i] is an integer number identifying the dataset in the dataset group. This field shall not take the same value more than once within the dataset group.

NOTE num_datasets is inferred from the Length field of datasets_group_header *gen_info* header as follows: num_datasets = (Length − 14) / 2.

6.4.2.3 Reference

6.4.2.3.1 General

This is an optional box (*rfgn* in Table 15) containing the information needed to retrieve an external or internal reference, and its description as a set of reference sequences.

It may be present in multiple instances in the same dataset group. If so, any instance shall have a different value of reference_ID.

6.4.2.3.2 Syntax

Table 15 — Reference box syntax

Syntax	Key	Type	Remarks
reference {	*rfgn*		
dataset_group_ID		u(8)	
reference_ID		u(8)	
reference_name		st(v)	
reference_major_version		u(16)	
reference_minor_version		u(16)	
reference_patch_version		u(16)	
seq_count		u(16)	
for (seqID=0;seqID<seq_count;seqID++) {			
sequence_name[seqID]		st(v)	
if (minor_version != '1900') {			minor_version as specified in subclause 6.4.1.3
sequence_length[seqID]		u(32)	
sequence_ID[seqID]		u(16)	
}			
}			
reserved		u(7)	
external_ref_flag		u(1)	
if (external_ref_flag) {			
ref_uri		st(v)	As specified in subclause 6.4.2.3.4
checksum_alg		u(8)	
reference_type		u(8)	
if (reference_type == MPEGG_REF) {			
external_dataset_group_ID		u(8)	
external_dataset_ID		u(16)	
if (minor_version == '1900')			minor_version as specified in subclause 6.4.1.3
ref_checksum		u(checksum_size)	As specified in 6.4.2.3.7
}			

Table 15 *(continued)*

Syntax	Key	Type	Remarks
`if (minor_version != '1900' || reference_type != MPEGG_REF) {`			`minor_version as specified in subclause 6.4.1.3`
`for(seqID=0;seqID<seq_count;seqID++) {`			
`ref_seq_checksum``[seqID]`		`u(checksum_size)`	`As specified in subclause 6.4.2.3.7`
`}`			
`}`			
`}`			
`else {`			
`internal_dataset_group_ID`		`u(8)`	
`internal_dataset_ID`		`u(16)`	
`}`			
`}`			

6.4.2.3.3 Semantics

dataset_group_ID is the identifier of the dataset group including this box. It shall have the same value as the dataset_group_ID field in the dataset group header of the same dataset group, as specified in subclause 6.4.2.2.

reference_ID is the identification number of the reference within the dataset group.

reference_name is a string representing a human readable name of the reference.

reference_major_version is the reference major version.

reference_minor_version is the reference minor version.

reference_patch_version is the reference patch version.

seq_count is the number of reference sequences contained in the reference genome.

sequence_name[seqID] is an unambiguous string identifier for each reference sequence contained in the reference.

sequence_length[seqID] is the number of nucleotides of each reference sequence contained in the reference.

sequence_ID[seqID] is reference sequence integer identifier.

external_ref_flag is a flag specifying whether the reference is either another dataset of the same bitstream, as specified in subclause 6.4.3, with dataset_type equal to 2 (external_ref_flag equal to 0), or a reference external to the bitstream (external_ref_flag equal to 1).

ref_uri as specified in subclause 6.4.2.3.4.

reference_type specifies the type of the external reference and can take any of the values in the first column of Table 16.

Table 16 — reference_type values

Value	Name	Semantics
0	MPEGG_REF	Reference encoded as a dataset, as specified in subclause 6.4.3, identified by fields external_dataset_group_ID and external_dataset_ID when external_ref_flag is equal to 1, or by fields internal_dataset_group_ID and internal_dataset_ID, when external_ref_flag is equal to 0, in a bitstream compliant to this document. The dataset shall have dataset_type, as specified in subclause 6.4.3.2, equal to 2.
1	RAW_REF	Raw reference, as specified in ISO/IEC 23092-2.
2	FASTA_REF	Reference of type FASTA, as specified in subclause 6.4.2.3.5
3 to 0xFF		Reserved for future use

external_dataset_group_ID is the identifier of the dataset group containing the external reference, in case ref_uri points to a reference coded in compliance to the ISO/IEC 23092 series.

external_dataset_ID is the identifier of the dataset containing the external reference, in case ref_uri points to a reference coded in compliance to the ISO/IEC 23092 series. The value shall be equal to one of the dataset_ID belonging to the dataset group identified by external_dataset_group_ID.

ref_checksum is the checksum computed, according to one of the methods specified in subclause 6.4.2.3.7, on the payload of the dataset of type 2 (Value field of dtcn box), as specified in subclause 6.4.3, retrieved using ref_uri, external_dataset_group_ID and external_dataset_ID.

ref_seq_checksum[seqID] is the checksum computed, according to one of the methods specified in subclause 6.4.2.3.7, for each reference sequence contained in the reference genome, on the content of the corresponding ref_sequence field of the raw reference, as specified in ISO/IEC 23092-2, either as is, if reference_type is equal to RAW_REF, or obtained via the conversion process specified in subclause 6.4.2.3.6, if reference_type is equal to FASTA_REF or MPEGG_REF.

internal_dataset_group_ID is an integer number identifying the dataset group containing the internal reference. An internal reference shall be of type MPEGG_REF, as specified in Table 16.

internal_dataset_ID is an integer number identifying the dataset containing the internal reference. An internal reference shall be of type MPEGG_REF, as specified in Table 16.

6.4.2.3.4 ref_uri semantics

ref_uri shall be compliant with IETF RFC 3986 and IETF RFC 7320.

The IETF RFC 3986 specification is partially summarized in Annex A.

6.4.2.3.5 Supported FASTA format

The FASTA format[2] supported by this document is represented as a series of lines in ASCII text format.

The first line in the FASTA shall start with a ">" (greater-than) symbol.

Ignore lines that only contain non printable characters and any comment line starting with a semi-colon.

Each sequence of characters following a ">" (greater-than) symbol and up to the first whitespace character shall be interpreted as the identifier (a.k.a. name) of the sequence of nucleotides represented by the following one or more lines.

Each sequence of characters following a ">" (greater-than) symbol and up to an end of line character shall be followed by one or more lines of symbols representing nucleotides.

Table 17 is an example of the supported FASTA format.

Table 17 — Example of the FASTA format

Line	Content	Description
1	>1 dna:chromosome chromosome:GRCh37:1:1:249250621:1	1 = first sequence identifier
2	ACGTTGACTATCGATCTATTAGCGGCGATGCA	Sub-sequences of nucleotides representing the entire first sequence
3	TGACTATCGATCTATTAGCGGCGATGCTTCCA	
4	ACGTTGACAAACCGATAAGCGGCGATGCAAAC	
...	...	
N	>2 dna:chromosome chromosome:GRCh37:2:1:243199373:1	2 = second sequence identifier
N+1	TGACTATCGATCTATTAGCGGCGATGCTTCCA	Sub-sequences of nucleotides representing the entire second sequence
N+2	ACGTTGACAAACCGATAAGCGGCGATGCAAAC	
N+3	TTGACAAACCGATAAGCGGCGATGCAAACAGT	
...	...	
...	...	...

A compliant codec will ignore all new line characters and any comment line starting with a semi-colon.

6.4.2.3.6 Conversion to raw reference

The reference either pointed by ref_uri (when external_ref_flag is equal to 1) and, if reference_type is equal to MPEGG_REF, identified by external_dataset_group_ID and external_dataset_ID, or (when external_ref_flag is equal to 0) identified by internal_dataset_group_ID and internal_dataset_ID fields, shall be converted into a raw reference structure, as specified in ISO/IEC 23092-2, according to the process described below.

— If either external_ref_flag is equal to 0, or external_ref_flag is equal to 1 and reference_type is equal to MPEGG_REF, as specified in Table 16, the corresponding dataset shall be decapsulated, according to subclause 6.4.3, and the output data units shall be decoded, according to the decoding process specified in ISO/IEC 23092-2.

— Else, if external_ref_flag is equal to 1 and reference_type is equal to RAW_REF, as specified in Table 16, no decapsulation is needed.

— Else, if external_ref_flag is equal to 1 and reference_type is equal to FASTA_REF, as specified in Table 16, the FASTA reference, as specified in subclause 6.4.2.3.6, shall be converted into a raw reference, by filling each ref_sequence field with the corresponding base characters in the FASTA reference, after conversion to uppercase letters and ignoring new line characters; seq_start field shall be set to 0, seq_end field shall be set to the number of characters composing each sequence minus 1, and the sequence_ID field structure shall be set either incrementally, starting from 0, for any reference sequence present in the FASTA reference, if minor_version is equal to '1900' or to the value of sequence_ID field in the reference box, as specified in subclause 6.4.2.3.2.

In all of the above three cases the output raw reference shall be encapsulated as payload of a data unit, as specified in subclause 6.3, with:

— data_unit_type equal to 0,

— data_unit_size equal to the sum of 9 (the number of bytes used for data_unit_type and data_unit_size) and the number of bytes composing the raw_reference structure.

6.4.2.3.7 Checksum

The identification of the hash function to be used to verify the identity of the related reference (ref_checksum field) or reference sequences (ref_seq_checksum[i] fields) is performed using checksum_alg, as specified in subclause 6.4.2.3.2. Two values of checksum_alg are defined in Table 18, while other values are reserved for future use.

Table 18 — Checksum values

checksum_alg value	Checksum algorithm	checksum_size	Rationale
0x00	MD5	128	Supported as checksum algorithm only for backward compatibility, but it is not recommended for the creation of new content due to the extensive collision vulnerabilities it suffers.
0x01	SHA-256	256	Currently recommended for all hash function-based applications and it shall be used for the integrity check of all new content.
0x02 to 0xFF			Reserved for future use.

6.4.2.4 Reference metadata

6.4.2.4.1 General

This is an optional box (*rfmd* in Table 19) containing metadata associated to a reference.

6.4.2.4.2 Syntax

Table 19 — Reference metadata syntax

Syntax	Key	Type	Remarks
reference_metadata {	*rfmd*		
dataset_group_ID		u(8)	
reference_ID		u(8)	
reference_metadata_value()			As specified in ISO/IEC 23092-3
}			

6.4.2.4.3 Semantics

dataset_group_ID is an integer number identifying the dataset group including this reference_metadata.

reference_ID is a unique identification number of the reference to which this reference_metadata refers to. It shall be equal to the reference_id value of one of the reference boxes, as specified in subclause 6.4.2.3, present in the dataset group.

reference_metadata_value() contains reference related metadata, as specified in ISO/IEC 23092-3.

6.4.2.5 Label List

6.4.2.5.1 General

This box (*labl* in Table 20) lists the labels, as specified in subclause 6.4.2.5.2, associated to a dataset group.

6.4.2.5.2 Syntax

Table 20 — Label list syntax

Syntax	Key	Type	Remarks
label_list {	*labl*		
dataset_group_ID		u(8)	
num_labels		u(16)	
for (h=0; h < num_labels; h++) {			
label[h]	*lbll*	gen_info	As specified in subclause 6.4.2.5.4
}			
}			

6.4.2.5.3 Semantics

dataset_group_ID is the identifier of the dataset group including this label list. It shall have the same value as the dataset_group_ID field in the dataset group header of the same dataset group, as specified in subclause 6.4.2.2.

num_labels is the total number of labels in the label list.

6.4.2.5.4 Label

6.4.2.5.4.1 General

A label (*lbll* in Table 21) is an identifier associated to one or more datasets, genomic regions and/or classes.

6.4.2.5.4.2 Syntax

Table 21 — Label syntax

Syntax	Key	Type	Remarks
label {	*lbll*		
label_ID		st(v)	
num_datasets		u(16)	
for (i=0;i < num_datasets; i++) {			
dataset_ID[i]		u(16)	
num_regions[i]		u(8)	
for (j=0; j < num_regions[i]; j++) {			
seq_ID[i][j]		u(16)	
num_classes[i][j]		u(4)	
for (k=0; k < num_classes[i][j]; k++) {			
class_ID[i][j][k]		u(4)	
}			
start_pos[i][j]		u(40)	
end_pos[i][j]		u(40)	
}			
}			

Table 21 *(continued)*

Syntax	Key	Type	Remarks
`while(!byte_aligned())`			As specified in subclause 4.2.3
`nesting_zero_bit`		f(1)	Equal to 0
}			

6.4.2.5.4.3 Semantics

label_ID is a string representing the label identifier in the label list specified in subclause 6.4.2.5. The variable stringLength for this field, as specified in subclause 4.2.3, concerning the st(v) data type, shall be higher than 0.

num_datasets is the number of datasets containing regions labelled by label_ID.

dataset_ID[i] is the identifier of a dataset labelled by label_ID. It shall take one of the values of dataset_ID listed in the dataset group header of the same dataset group, as specified in subclause 6.4.2.2.

num_regions[i] is the number of regions labelled by label_ID in the dataset.

seq_ID[i][j] is the sequence identifier. It shall take the value of one of the sequence_ID fields of at least one of the reference boxes included in the dataset group, as specified in subclause 6.4.2.3.

num_classes[i][j] is the number of classes labelled by label_ID in the region.

class_ID[i][j][k] identifies the data class in the region labelled by label_ID, as specified in Table 6.

start_pos[i][j] is the position of the left-most mapped base among the first alignments of all genomic records encoded in the access units covering the region, irrespective of the strand.

end_pos[i][j] is the position of the right-most mapped base among the first alignments of all genomic records encoded in the access units covering the region, irrespective of the strand.

6.4.2.6 Dataset group metadata

6.4.2.6.1 General

This is an optional box (*dgmd* in Table 22) containing metadata associated to a dataset group.

6.4.2.6.2 Syntax

Table 22 — Dataset group metadata syntax

Syntax	Key	Type	Remarks
`DG_metadata {`	*dgmd*		
`if (minor_version != '1900') {`			minor_version as specified in subclause 6.4.1.3
`dataset_group_ID`		u(8)	
`}`			
`DG_metadata_value()`			As specified in ISO/IEC 23092-3
}			

6.4.2.6.3 Semantics

dataset_group_ID is the identifier of the dataset group containing this metadata.

DG_metadata_value() contains the dataset group metadata, as specified in ISO/IEC 23092-3.

6.4.2.7 Dataset group clinical data linkage metadata

6.4.2.7.1 General

This is an optional box (*dgcd* in Table 23) containing clinical data linkage metadata associated to a dataset group.

6.4.2.7.2 Syntax

Table 23 — Dataset group clinical data linkage metadata syntax

Syntax	Key	Type	Remarks
DG_CDL {	*dgcd*		
dataset_group_ID		u(8)	
DG_CDL_value()			As specified in ISO/IEC 23092-3
}			

6.4.2.7.3 Semantics

dataset_group_ID is the identifier of the dataset group containing this clinical data linkage metadata.

DG_CDL_value() contains the dataset group clinical data linkage metadata, as specified in ISO/IEC 23092-3.

6.4.2.8 Dataset group protection

6.4.2.8.1 General

This is an optional box (*dgpr* in Table 24) containing protection information associated to a dataset group.

When present this box contains information that a decoder needs to properly handle a protected dataset group.

6.4.2.8.2 Syntax

Table 24 — Dataset group protection syntax

Syntax	Key	Type	Remarks
DG_protection {	*dgpr*		
if (minor_version != '1900') {			minor_version as specified in subclause 6.4.1.3
dataset_group_ID		u(8)	
}			
DG_protection_value()			As specified in ISO/IEC 23092-3
}			

6.4.2.8.3 Semantics

dataset_group_ID is the identifier of the dataset group containing this protection metadata.

DG_protection_value() contains the dataset group protection information, as specified in ISO/IEC 23092-3.

6.4.3 Dataset

6.4.3.1 General

A dataset is a collection of access units encoding either records or a reference.

The corresponding container box (*dtcn* in Table 25) is mandatory in file format, forbidden in transport format.

Child boxes may be present or not, according to the column "Mandatory" in Table 10. Child boxes marked with suffix "[]" after their name in the Syntax column of Table 25 may be present in multiple instances.

Table 25 — Dataset syntax

Syntax	Key	Type	Remarks
dataset {	*dtcn*		
dataset_header	dthd	gen_info	As specified in subclause 6.4.3.2
DT_metadata	dtmd	gen_info	As specified in subclause 6.4.3.3
DT_metrics	dtmt	gen_info	As specified in subclause 6.4.3.4
DT_CDL	dtcd	gen_info	As specified in subclause 6.4.3.5
DT_protection	dtpr	gen_info	As specified in subclause 6.4.3.6
dataset_parameter_set[]	pars	gen_info	As specified in subclause 6.4.3.7
if(dataset_type < 3){			As specified in subclause 6.4.3.2
if (MIT_flag == 1) {			As specified in subclause 6.4.3.2
master_index_table	mitb	gen_info	As specified in subclause 6.5.2.1
}			
access_unit[]	aucn	gen_info	As specified in subclause 6.4.4.1
if (block_header_flag == 0) {			
descriptor_stream[]	dscn	gen_info	As specified in subclause 6.5.3.1
}			
}			
else if(dataset_type == 3) {			
annotation_table[]	atcn	gen_info	As specified in subclause 6.4.6.1
}			
}			

6.4.3.2 Dataset header

6.4.3.2.1 General

This is a mandatory box (*dthd* in Table 26) describing the content of a dataset.

6.4.3.2.2 Syntax

Table 26 — Dataset header syntax

Syntax	Key	Type	Remarks
dataset_header {	*dthd*		
dataset_group_ID		u(8)	
dataset_ID		u(16)	
version		c(4)	
multiple_alignment_flag		u(1)	
byte_offset_size_flag		u(1)	
non_overlapping_AU_range_flag		u(1)	
pos_40_bits_flag		u(1)	
block_header_flag		u(1)	
if (block_header_flag) {			
MIT_flag		u(1)	
CC_mode_flag		u(1)	
}			
else {			
ordered_blocks_flag		u(1)	
}			
seq_count		u(16)	
if (seq_count > 0) {			
reference_ID		u(8)	
for (seq = 0; seq < seq_count; seq++) {			
seq_ID[seq]		u(16)	
}			
for (seq = 0; seq < seq_count; seq++) {			
seq_blocks[seq]		u(32)	
}			
}			
dataset_type		u(4)	
if (MIT_flag == 1) {			
num_classes		u(4)	
for (ci = 0; ci < num_classes; ci++) {			
clid[ci]		u(4)	
if (!block_header_flag) {			
num_descriptors[ci]		u(5)	
for(di = 0; di < num_descriptors[ci]; di++) {			
descriptor_ID[ci][di]		u(7)	
}			
}			
}			
}			
parameters_update_flag		u(1)	
alphabet_ID		u(7)	
num_U_access_units		u(32)	
if (num_U_access_units > 0) {			

Table 26 *(continued)*

Syntax	Key	Type	Remarks
reserved		u(62)	For backward compatibility
U_signature_flag		u(1)	
if(U_signature_flag) {			
U_signature_constant_length		u(1)	
if (U_signature_constant_length){			
U_signature_length		u(8)	
}			
}			
reserved_flag		u(1)	
if (reserved_flag){			
reserved		u(8)	For backward compatibility
}			
reserved_flag		u(1)	
}			
if (seq_count > 0) {			
tflag[0]		f(1)	Equal to 1
thres[0]		u(31)	
for (i = 1; i < seq_count; i++) {			
tflag[i]		u(1)	
if(tflag[i] == 1)			
thres[i]		u(31)	
else /* tflag[i] == 0 */			
/* thres[i] = thres[i-1] */			
}			
}			
if(dataset_type == 3){			
n_ann_tables		u(7)	
for (i = 0; i < n_ann_tables; i++) {			
AT_ID[i]		u(8)	
AT_index_exists[i]		u(1)	
}			
}			
else if (dataset_type > 3){			
/* reserved for future use */			
}			
while(!byte_aligned())			As specified in subclause 4.2.3
nesting_zero_bit		f(1)	Equal to 0
}			

6.4.3.2.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this dataset_header.

dataset_ID is the identifier of the dataset. Its value shall be one of the dataset_IDs listed in the dataset_group_header.

version is the combination of version number, amendment number and corrigendum number of ISO/IEC 23092-2 to which the Value field of the dataset complies, and is specified as follows:

— first two bytes: version number, as the last two digits of the year of release of the major brand

— third byte: amendment number, as integer counter from 0 to 9, 0 if no amendment yet

— fourth byte: corrigendum number, as integer counter from 0 to 9, 0 if no corrigendum yet

multiple_alignment_flag: if set to 1 it indicates the presence of multiple alignments in the dataset. It shall never be set to 1 if dataset_type is equal to 2.

byte_offset_size_flag: if equal to 0, the variable byteOffsetSize used in the master index table, as specified in subclause 6.5.2.1, and representing the number of bits used to encode the master index table fields named AU_byte_offset and block_byte_offset, is equal to 32; if set to 1, the variable byteOffsetSize is equal to 64.

non_overlapping_AU_range: if set to 1, all access units with the same AU_type in the dataset have non-overlapping ranges.

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 subclauses 6.4.4.3.2, 6.5.2.1.2 and 6.5.2.4.4.2 the value of the variable posSize is set to 32 when pos_40_bits is equal to 0 and set to 40 otherwise.

block_header_flag: if set, all blocks composing the dataset are preceded by a block header, as specified in subclause 6.4.5.2. It is always set to 1 in transport format. See also subclause 6.4.3.2.5.

MIT_flag: if set, the master index table, as specified in subclause 6.5.2.1 is present in the dataset. Otherwise, the master index table is not present in the dataset. It is always equal to 0 in transport format and set to 1 by default when block_header_flag is equal to 0.

CC_mode_flag: if set, two access units of the same type, as specified in Table 7, cannot be separated by access units of a different type in the storage device. If equal to 0, access units are ordered by access unit start position in the storage device. See also subclause 6.4.3.2.5.

ordered_blocks_flag: if set, blocks are ordered in the descriptor stream by increasing value of the entry AU_start_position of the master index table, as specified in subclause 6.5.2.1. See also subclause 6.4.3.2.5.

seq_count is the number of reference sequences used in this dataset, in case of file format; in case of transport format and if dataset type is different than 0, it shall be equal to the seq_count field in the reference box, as specified in subclause 6.4.2.3.1, referenced by this dataset by the reference_id field. It shall always be equal to 0 when dataset_type is equal to 0.

reference_ID is a unique identification number of the reference used by the dataset for alignment. It shall take the value of the reference_ID field of any of the reference boxes included in the dataset group including this dataset, as specified in subclause 6.4.2.3. If dataset_type is equal to 2, and if the external_ref_flag field of the reference box pointed by reference_id, as specified in subclause 6.4.2.3, is equal to 0, then the reference box pointed by reference_id, as specified in subclause 6.4.2.3, shall have either a value of the internal_dataset_ID field different than the value of the dataset_ID field in this dataset header, or a value of the internal_dataset_group_ID field different than the value of the dataset_group_ID field in this dataset header.

seq_ID[seq] its value shall correspond to any of the values of the sequence_ID variable in the reference box identified by reference_ID, as specified in subclause 6.4.2.3.

seq_blocks[seq] is the number of access units per reference sequence with a different value of access_unit_ID, as specified in subclause 6.4.4.3. A value of 0 means “unspecified” (e.g., in transport format).

dataset_type specifies the type of data encoded in the dataset. The possible values are: 0 = non-aligned content; 1 = aligned content; 2 = reference; 3 = annotation.

num_classes is the number of classes encoded in the dataset.

clid identifies the class of data carried by the access unit, as specified in Table 6. It shall take any of the values defined as Class ID in Table 6. clid[ci+1] shall be greater than clid[ci], for ci in the range between 0 and (num_classes – 2), inclusive. Variable ci is used as a local identifier for the class in the other syntax tables included in the same dataset.

num_descriptors[ci] is the maximum number of descriptors per class encoded in the dataset.

descriptor_ID[ci][di] is a descriptor identifier as specified in ISO/IEC 23092-2.

parameters_update_flag: if equal to 1, the fields multiple_alignment_flag, pos_40_bits_flag, U_signature_flag, U_signature_constant_length and U_signature_length in this dataset header are also present in all the dataset parameter sets, as specified in subclause 6.4.3.7.2, contained in the same dataset. In such a case (parameters_update_flag equal to 1), such fields shall have the following values in the dataset header:

— multiple_alignment_flag shall be equal to 1 if the multiple_alignment_flag field in at least one of the dataset parameter sets, as specified in subclause 6.4.3.7.3, is equal to 1; otherwise it shall be equal to 0.

— pos_40_bits shall be equal to 1 if the pos_40_bits field in at least one of the dataset parameter sets, as specified in subclause 6.4.3.7.3, is equal to 1; otherwise it shall be equal to 0.

— alphabet_id shall be equal to 1 if the alphabet_id field in at least one of the dataset parameter sets, as specified in subclause 6.4.3.7.3, is equal to 1; otherwise it shall be equal to 0.

— U_signature_flag shall be equal to 1 if the U_signature_flag field in at least one of the dataset parameter sets, as specified in subclause 6.4.3.7.3, is equal to 1; otherwise it shall be equal to 0.

— U_signature_constant_length shall be equal to 0.

When parameters_update_flag is equal to 1:

— when used in the access unit header, as specified in subclause 6.4.4.3, such fields will take the value they have in the dataset parameter set, as specified in subclause 6.4.3.7, referenced by the access unit via the parameter_set_ID field, as specified in subclause 6.4.4.3

— when used in the master index table, as specified in subclause 6.5.2.1, such fields will take the values present in the dataset header, as specified in this subclause.

alphabet_ID is the identifier of the alphabet used to encode the cluster signatures. Values are described in Table 27 in subclause 6.4.3.2.4.

num_U_access_units is the total number of access units in the dataset containing encoded data of class U, in case of file format; in case of transport format, it shall always be set to 1 and does not have any specific semantics apart from the use as a flag in Table 26.

U_signature_flag: if set to 1 it indicates the presence of cluster signatures in the access units contained in the dataset.

U_signature_constant_length: if set to 1 all cluster signatures in the dataset have the same length.

U_signature_length is the length of all cluster signatures in the dataset, as number of nucleotides.

tflag[i]: if set to 1 it indicates that it is followed by a threshold **thres**[i]**.**

thres[i] is a threshold indicating the maximum difference between the access unit covered region and the access unit range.

n_ann_tables is the number of annotation tables available in the dataset when dataset type is set to 3.

AT_ID[] shall correspond to the value of the AT_ID variable in one of the annotation tables contained in the dataset, as specified in subclause 6.4.6.2.

AT_index_exists[i] is a flag, if set to 1, indicates the presence of index data for the annotation table identified by AT_ID[i].

6.4.3.2.4 Alphabets

Each Alphabet is identified by an alphabet_ID as shown in Table 27:

Table 27 — alphabet_ID semantics

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

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

S_0[0]='A', S_0[1]='C', S_0[2]='G', S_0[3]='T' , S_0[4]='N'

S_1[0]='A', S_1[1]='C', S_1[2]='G', S_1[3]='T', S_1[4]='R', S_1[5]='Y', S_1[6]='S', S_1[7]='W', S_1[8]='K', S_1[9]='M', S_1[10]='B' , S_1[11]='D', S_1[12]='H', S_1[13]='V', S_1[14]='N', S_1[15]='-'

6.4.3.2.5 Block contiguity

The field block_header_flag is used to enable two possible modes of block contiguity in the file:

— descriptor stream contiguity (DSC) mode: blocks, as specified in subclause 6.4.3.2.5, belonging to the same descriptor stream, as specified in subclause 6.5.3, are stored in contiguous areas of the storage device. This mode is enabled by the condition block_header_flag equal to 0.

— access unit contiguity (AUC) mode: blocks, as specified in subclause 6.4.5, belonging to the same access unit, as specified in subclause 6.4.4, are stored in contiguous areas of the storage device. This mode is enabled by the condition block_header_flag equal to 1.

When block_header_flag is equal to 1, the CC_mode_flag field is used to enable two possible modes of access units contiguity in the file, named:

— genomic region contiguity (AUC-GRC) mode: access units are ordered by access unit start position in the storage device. This mode is enabled by the condition CC_mode_flag equal to 0.

— class contiguity (AUC-CC) mode: two access units of one class cannot be separated by access units of a different class in the storage device. This mode is enabled by the condition CC_mode_flag equal to 1.

No other block contiguity modes are allowed by this document.

When block_header_flag is equal to 0 (DSC mode), the field ordered_blocks_flag is used to indicate whether the blocks are ordered in the storage device according to the left-most aligned position of the left-most read in the access unit (field AU_ref_start_position in access unit header, as specified in subclause 6.4.4.3, or master index table, as specified in subclause 6.5.2.1). If ordered_blocks_flag is equal to 1, the file offsets for a given descriptor stream and for each block are sorted in ascending order (disregarding blocks for which the block_byte_offset in the master index table is equal to ((1 << byteOffsetSize) -1). In this mode the first byte not belonging to the block is the first byte of the next available block if any (otherwise the descriptor_stream_size, which can be inferred from the Length field of the gen_info header of descriptor stream container box with Key *dscn*, should be used).

If ordered_blocks_flag is equal to 0, the blocks may be stored in any order in the descriptor stream. In order to infer the offset of the first byte not belonging to the block, the decoder has to search, among all offsets provided for the descriptor stream which are not equal to ((1 << byteOffsetSize) -1), the smallest value greater than the offset of the block, if any (otherwise the descriptor_stream_size should be used as above).

6.4.3.3 Dataset metadata

6.4.3.3.1 General

This is an optional box (*dtmd* in Table 28) containing metadata associated to the dataset.

6.4.3.3.2 Syntax

Table 28 — Dataset metadata syntax

Syntax	Key	Type	Remarks
DT_metadata {	*dtmd*		
if (minor_version != '1900') {			minor_version as specified in subclause 6.4.1.3
dataset_group_ID		u(8)	
dataset_ID		u(16)	
}			
DT_metadata_value()			As specified in ISO/IEC 23092-3
}			

6.4.3.3.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this metadata.

dataset_ID is the identifier of the dataset containing this metadata.

DT_metadata_value() contains dataset metadata, as specified in ISO/IEC 23092-3.

6.4.3.4 Dataset metrics metadata

6.4.3.4.1 General

This is an optional box (*dtmt* in Table 29) containing metrics metadata associated to the dataset, used only for genomic sequencing data.

6.4.3.4.2 Syntax

Table 29 — Dataset metrics metadata syntax

Syntax	Key	Type	Remarks
DT_metrics {	*dtmt*		
dataset_group_ID		u(8)	
dataset_ID		u(16)	
DT_metrics_value()			As specified in ISO/IEC 23092-3
}			

6.4.3.4.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this metrics metadata.

dataset_ID is the identifier of the dataset containing this metrics metadata.

DT_metrics_value() contains dataset metrics metadata, as specified in ISO/IEC 23092-3.

6.4.3.5 Dataset clinical data linkage metadata

6.4.3.5.1 General

This is an optional box (*dtcd* in Table 30) containing clinical data linkage metadata associated to the dataset.

6.4.3.5.2 Syntax

Table 30 — Dataset clinical data linkage metadata syntax

Syntax	Key	Type	Remarks
DT_CDL {	*dtcd*		
dataset_group_ID		u(8)	
dataset_ID		u(16)	
DT_CDL_value()			As specified in ISO/IEC 23092-3
}			

6.4.3.5.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this clinical data linkage metadata.

dataset_ID is the identifier of the dataset containing this clinical data linkage metadata.

DT_CDL_value() contains dataset clinical data linkage metadata, as specified in ISO/IEC 23092-3.

6.4.3.6 Dataset protection

6.4.3.6.1 General

This is an optional box (*dtpr* in Table 31) containing protection information associated to the dataset.

When present this box contains information that a decoder needs to properly handle a protected dataset.

6.4.3.6.2 Syntax

Table 31 — Dataset protection syntax

Syntax	Key	Type	Remarks
DT_protection {	*dtpr*		
if (minor_version != '1900') {			minor_version as specified in subclause 6.4.1.3
dataset_group_ID		u(8)	
dataset_ID		u(16)	
}			
DT_protection_value()			As specified in ISO/IEC 23092-3
}			

6.4.3.6.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this protection metadata.

dataset_ID is the identifier of the dataset containing this protection metadata.

DT_protection_value() contains dataset protection information, as specified in ISO/IEC 23092-3.

6.4.3.7 Dataset parameter set

6.4.3.7.1 General

This is a mandatory box (*pars* in Table 32) describing any of the parameter sets associated to the dataset identified by dataset_ID in the dataset group identified by dataset_group_ID.

It may be present in multiple instances in the same dataset.

When the dataset_type is smaller than 3, the decapsulation of this box shall result in a data unit, as specified in subclause 6.4, with:

— data_unit_type equal to 1,

— data_unit_size equal to the sum of 5 (the number of bytes used for data_unit_type and data_unit_size), 2 (the number of bytes for parent_parameter_set_ID and parameter_set_ID, as specified in subclause 6.4.3.7.2) and the number of bytes composing the encoding_parameters() structure, as specified in subclause 6.4.3.7.2 and

— as payload a parameter_set() structure composed of the parent_parameter_set_ID, parameter_set_ID and encoding_parameters() fields, as specified in subclause 6.4.3.7.2.

Such data unit can be dispatched to a decoder compliant with ISO/IEC 23092-2.

When the dataset_type is 3, the decapsulation of this box shall result in a data unit, as specified in subclause 6.3, with:

— data_unit_type equal to 3,

— data_unit_size equal to the sum of 5 (the number of bytes used for data_unit_type and data_unit_size), 1 (the number of bytes for parameter_set_ID, as specified in subclause 6.4.6.2) and the number of bytes composing the annotation_encoding_parameters() structure specified in ISO/IEC 23092-6, and,

— as payload an annotation_parameter_set() structure composed of the parameter_set_ID and annotation_encoding_parameters() fields.

Such data unit can be dispatched to a decoder compliant with ISO/IEC 23092-6.

6.4.3.7.2 Syntax

Table 32 — Dataset parameter set syntax

Syntax	Key	Type	Remarks
dataset_parameter_set {	*pars*		
dataset_group_ID		u(8)	
dataset_ID		u(16)	
parameter_set_ID		u(8)	
parent_parameter_set_ID		u(8)	
if (minor_version != '1900' && parameters_update_flag && dataset_type < 3) {			minor_version as specified in subclause 6.4.1.3 The other fields as specified in subclause 6.4.3.2
multiple_alignment_flag		u(1)	
pos_40_bits_flag		u(1)	
alphabet_ID		u(8)	

Table 32 *(continued)*

Syntax	Key	Type	Remarks
`    if (num_U_access_units > 0) {`			As specified in subclause 6.4.3.2
`      **U_signature_flag**`		u(1)	
`      if(U_signature_flag) {`			
`        **U_signature_constant_length**`		u(1)	
`        if (U_signature_constant_length) {`			
`          **U_signature_length**`		u(8)	
`        }`			
`      }`			
`    }`			
`    while(!byte_aligned())`			As specified in subclause 4.2.3
`      **nesting_zero_bit**`		f(1)	
`  }`			
`  if (dataset_type < 3) {`			
`    encoding_parameters()`			As specified in ISO/IEC 23092-2
`  }`			
`  else if (dataset_type == 3){`			
`    annotation_encoding_parameters()`			As specified in ISO/IEC 23092-6
`  }`			
`  else {`			
`    /* reserved for future use*/`			
`  }`			
`}`			

6.4.3.7.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this dataset parameter set. It shall be equal to the dataset_group_ID of the containing dataset group.

dataset_ID is the identifier of the dataset including this dataset parameter set. It shall be equal to the dataset_id of the containing dataset.

parameter_set_ID is the identifier of this dataset parameter set within the dataset.

parent_parameter_set_ID is the identifier of any of the dataset parameter sets within the dataset. Referencing an existing dataset parameter set from another parameter set enables the generation of a hierarchy of dataset parameter sets to be associated to an access unit, as specified in subclause 6.5.4. If the value of parent_parameter_set_ID is equal to the value of parameter_set_ID, then the dataset parameter set is at the top level in the hierarchy. Set to 0 in case of dataset type 3.

multiple_alignment_flag: if set to 1 it indicates the presence of multiple alignments in the access units associated to this dataset parameter set. It shall never be set to 1 if dataset_type is equal to 2.

pos_40_bits_flag is set to 1 when the mapping positions in the access units associated to this dataset parameter set are expressed as 40 bits integers. Otherwise, all mapping positions are expressed as 32 bits integers. In the scope of subclauses 6.4.4.3.2, 6.5.2.1.2 and 6.5.2.4.4.2, the value of the variable posSize is set to 32 when pos_40_bits is equal to 0 and set to 40 otherwise.

alphabet_ID is the identifier of the alphabet used to encode the cluster signatures in the access units associated to this dataset parameter set. Values are described in Table 27 in subclause 6.4.3.2.4.

U_signature_flag: if set to 1 it indicates the presence of cluster signatures in the access units associated to this dataset parameter set.

U_signature_constant_length: if set to 1 all cluster signatures in the access units associated to this dataset parameter set have the same length.

U_signature_length is the length of all cluster signatures in the access units associated to this dataset parameter set, as number of nucleotides.

encoding_parameters() is an encoding_parameters() structure as specified in ISO/IEC 23092-2.

annotation_encoding_parameters() is an encoding_parameters() structure as specified in ISO/IEC 23092-6.

6.4.4 Access unit

6.4.4.1 General

The access unit (*aucn* in Table 33) is a collection of one or more blocks representing genomic information.

The decapsulation of this mandatory box shall result in a data unit, as specified in subclause 6.3, with:

— data_unit_type equal to 2,

— data_unit_size equal to the sum of 5 (the number of bytes used for data_unit_type and data_unit_size) and the number of bytes composing the output access unit structure, as specified in subclause 6.4.4.2,

— as payload the output access unit structure, as specified in subclause 6.4.4.2.

Such data unit can be dispatched to a decoder compliant with ISO/IEC 23092-2, along with all the parameter sets that are needed to decode it.

Table 33 — Access unit syntax

Syntax	Key	Type	Remarks
access_unit {	*aucn*		
access_unit_header	auhd	gen_info	As specified in subclause 6.4.4.3
AU_information	auin	gen_info	As specified in subclause 6.4.4.4
AU_metrics	aumt	gen_info	As specified in subclause 6.4.4.5
AU_protection	aupr	gen_info	As specified in subclause 6.4.4.6
if (block_header_flag) {			As specified in subclause 6.4.3.2
for (i=0; i < num_blocks; i++) {			As specified in subclause 6.4.4.3
block[i]			As specified in subclause 6.4.5
}			
}			
}			

6.4.4.2 Access unit decapsulation

Output to this process is an output access unit structure composed of:

— an access_unit_header structure, as specified in subclause 6.4.4.3, where MIT_flag shall be set to 0, and which is composed of:

  — the entire Value field of the access_unit_header child box present in the access unit box, as specified in Table 33, possibly followed, if MIT_flag in the dataset header, as specified in subclause 6.4.3.2, is equal to 1, by

  — the set of fields in the access unit header, as specified in subclause 6.4.4.3, which are enclosed within the if (MIT_flag==0) condition branch (such as sequence_ID, AU_start_position, AU_end_position, etc.) and which shall be derived from the corresponding parameters in the master index table, as specified in subclause 6.5.2.1:

    — The seq index shall be used to infer the sequence_ID field value from the seq_ID array in the dataset header, as specified in subclause 6.4.3.2

    — AU_start_position shall be set to the corresponding AU_start_position field in the master index table, as specified in subclause 6.5.2.1

    — AU_end_position , shall be set to the corresponding AU_end_position field in the master index table, as specified in subclause 6.5.2.1;

— the set of blocks either contained in the access unit box, as specified in Table 33, if block_header_flag is equal to 1, or to be retrieved from the descriptor streams, as specified in subclauses 6.5.2 and 6.5.3.1, if block_header_flag is equal to 0; in the second case, a block header, as specified in subclause 6.4.5.2, shall be prepended to all the blocks, where the descriptor_ID field shall be equal to the descriptor_ID field of the corresponding descriptor stream header, as specified in subclause 6.5.3.2, and the block_payload_size field shall be equal to the number of bytes composing the block payload as retrieved from the descriptor stream. For self indexed descriptors the block payload is retrieved from the corresponding decompressed string index defined in subclause 7.1.1.

6.4.4.3 Access unit header

6.4.4.3.1 General

This mandatory box (*auhd* in Table 34) contains information associated to the access unit.

6.4.4.3.2 Syntax

Table 34 — Access unit header syntax

Syntax	Key	Type	Remarks
access_unit_header {	*auhd*		
access_unit_ID		u(32)	
num_blocks		u(8)	
parameter_set_ID		u(8)	
AU_type		u(4)	
reads_count		u(32)	
if (AU_type == N_TYPE_AU || AU_type == M_TYPE_AU) {			
mm_threshold		u(16)	
mm_count		u(32)	
}			
if (dataset_type == 2) {			

Table 34 *(continued)*

Syntax	Key	Type	Remarks
ref_sequence_id		u(16)	
ref_start_position		u(posSize)	posSize as specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
ref_end_position		u(posSize)	posSize as specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
}			
if (MIT_flag == 0) {			As specified in subclause 6.4.3.2
if (AU_type != U_TYPE_AU) {			
sequence_ID		u(16)	
AU_start_position		u(posSize)	posSize as specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
AU_end_position		u(posSize)	posSize as specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
if (multiple_alignment_flag) {			As specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
extended_AU_start_position		u(posSize)	posSize as specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
extended_AU_end_position		u(posSize)	posSize as specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
}			
}			
else {			
if (U_signature_flag) {			As specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
num_signatures		u(16)	
for (i = 0; i < num_signatures; i++) {			

Table 34 *(continued)*

Syntax	Key	Type	Remarks
`if (!U_signature_constant_length) {`			As specified in subclause 6.4.3.2.3 or 6.4.3.7, depending on parameters_update_flag field
`U_signature_length``[i]`		`u(8)`	
`}`			
`U_cluster_signature``[i]`		`u(U_signature_size)`	U_signature_size inferred as specified in the U_cluster_signature semantics in subclause 6.4.4.3.3
`}`			
`}`			
`}`			
`}`			
`while(!byte_aligned())`			As specified in subclause 4.2.3
`nesting_zero_bit`		`f(1)`	
`}`			

6.4.4.3.3 Semantics

access_unit_ID is the identifier of the access unit among all access units with same AU_type, and, if AU_type is not equal to U_TYPE_AU, with the same sequence_ID.

num_blocks is the number of blocks, as specified in subclause 6.4.5, in the access unit.

parameter_set_ID is a unique identifier, in the dataset containing this access unit, of the dataset parameter set at the lowest level in the hierarchy of dataset parameter sets, which shall be returned by a decapsulator compliant to this document along with the decapsulated access unit, as specified in subclause 6.4.4. Such hierarchy of dataset parameter sets is enabled by the parent_parameter_set_ID and parameter_set_ID fields of the dataset parameter set, as specified in subclause 6.4.3.7.

AU_type identifies the type of access unit and the type of data (class) carried therein as specified in Table 7 in subclause 5.4.

reads_count is a counter of the genomic sequence 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 as no reads would be below threshold.

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. It shall always be set to 0 if the threshold is set to 0.

ref_sequence_id in case of access unit carrying (part of) a reference sequence (dataset_type, as specified in subclause 6.4.3.2), is the identifier of such reference sequence.

ref_start_position: in case of an access unit carrying (part of) a reference sequence (dataset_type, as specified in subclause 6.4.3.2), it specifies the position on the reference sequence of the left-most nucleotide encoded in this access unit.

ref_end_position: in case of an access unit carrying (part of) a reference sequence (dataset_type, as specified in subclause 6.4.3.2), it specifies the position on the reference sequence of the right-most nucleotide encoded in this access unit.

sequence_ID is the identifier of the reference sequence this access unit refers to. It shall be equal to one of the values of the seq_ID field listed in the dataset header, as described in subclause 6.4.3.2.

AU_start_position is the position of the left-most 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 right-most 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 left-most 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 right-most mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand.

num_signatures is the number of signatures in the access unit.

U_signature_length[i] is the length of one cluster signature as number of nucleotides.

U_cluster_signature[i] is the signature of the cluster this access unit belongs to. The length in bits of this field, named U_signature_size in Table 34, shall be calculated as follows:

$$\text{U\_signature\_size} = \text{signature\_length} * \text{bits\_per\_symbol}$$

with bits_per_symbol as specified in Table 27, and signature_length corresponding either to U_signature_length as specified in subclause 6.4.3.2 when U_signature_constant_length (as specified in subclause 6.4.3.2.2) is equal to 1 or to the signature-specific U_signature_length[i] as specified inTable 34, when U_signature_constant_length (specified in subclause 6.4.3.2.2) is equal to 0.

The j^{th} base in a signature shall be inferred as follows:

$S_{\text{alphabet\_ID}}$[(U_cluster_signature[i] >> ((signature_length − j − 1) * bits_per_symbol)) & ((1 << bits_per_symbol) − 1)]

with alphabet_id as specified in subclause 6.4.3.2.2 and $S_{\text{alphabet\_ID}}$ as specified in subclause 6.4.3.2.4.

6.4.4.4 Access unit information

6.4.4.4.1 General

This is an optional box (*auin* in Table 35) containing accessory information associated to the access unit.

6.4.4.4.2 Syntax

Table 35 — Access unit information syntax

Syntax	Key	Type	Remarks
AU_information {	*auin*		
if (minor_version != '1900') {			minor_version as specified in subclause 6.4.1.3
dataset_group_ID		u(8)	
dataset_ID		u(16)	
}			
AU_information_value()			As specified in ISO/IEC 23092-3.
}			

6.4.4.4.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this access unit information.

dataset_ID is the identifier of the dataset containing this access unit information.

AU_information_value() contains accessory information related to the access unit, as specified in ISO/IEC 23092-3.

6.4.4.5 Access unit metrics metadata

6.4.4.5.1 General

This is an optional box (*aumt* in Table 36) containing metrics metadata associated to the access unit.

6.4.4.5.2 Syntax

Table 36 — Access unit metrics metadata syntax

Syntax	Key	Type	Remarks
AU_metrics {	*aumt*		
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AU_metrics_value()			As specified in ISO/IEC 23092-3.
}			

6.4.4.5.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this access unit metrics metadata.

dataset_ID is the identifier of the dataset containing this access unit metrics metadata.

AU_metrics_value() contains access unit metrics metadata, as specified in ISO/IEC 23092-3.

6.4.4.6 Access unit protection

6.4.4.6.1 General

This is an optional box (*aupr* in Table 37) containing protection information associated to the access unit.

When present this box contains information that a decoder needs to properly handle a protected access unit.

6.4.4.6.2 Syntax

Table 37 — Access unit protection syntax

Syntax	Key	Type	Remarks
AU_protection {	*aupr*		
if (minor_version != '1900') {			minor_version as specified in subclause 6.4.1.3
dataset_group_ID		u(8)	
dataset_ID		u(16)	
}			
AU_protection_value()			As specified in ISO/IEC 23092-3
}			

6.4.4.6.3 Semantics

dataset_group_ID is the identifier of the dataset group containing the dataset including this access unit protection metadata.

dataset_ID is the identifier of the dataset containing this access unit protection metadata.

AU_protection_value() contains access unit protection information, as specified in ISO/IEC 23092-3.

6.4.5 Block

6.4.5.1 General

A block (see Table 38) is composed of a block header (as specified in subclause 6.4.5.2) and a block payload, containing compressed descriptors of the same type (descriptor_ID) and class (class_ID). In DSC mode, as specified in subclause 6.4.3.2.5, only the block payload is present in the descriptor stream, as specified in subclause 6.5.3.

Table 38 — Block syntax

Syntax	Key	Type	Remarks
block {	block		
block_header			As specified in subclause 6.4.5.2
for (i = 0; i < block_payload_size; i++) {			block_payload_size as specified in subclause 6.4.5.2
block_payload[i]		u(8)	
}			
}			

block_payload[i] is the i^th byte of block payload, which contains compressed descriptors of the same type (descriptor_ID) and class (class_ID).

6.4.5.2 Block header

6.4.5.2.1 General

This box (see Table 39) contains information associated to the block.

This box shall replace the block header provided by the underlying codec and specified in ISO/IEC 23092-2.

6.4.5.2.2 Syntax

Table 39 — Block header syntax

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

6.4.5.2.3 Semantics

descriptor_ID is the descriptor identifier, as specified in ISO/IEC 23092-2.

block_payload_size is the number of bytes composing the block payload.

6.4.6 Annotation Table

6.4.6.1 General

Annotation Table is the main container of annotation data. The corresponding container box (*atcn* in Table 40) is mandatory in datasets of type 3 in data format, forbidden in transport format. It includes the main attribute group, and optionally one or multiple auxiliary attribute groups, that contain the main and auxiliary attribute data.

Table 40 — Annotation table syntax

Syntax	Key	Type	Remarks
annotation_table {	atcn		
AT_header	*athd*	gen_info	As specified in subclause 6.4.6.2
AT_metadata	atmd	gen_info	As specified in subclause 6.4.6.3
AT_CDL	atcd	gen_info	As specified in subclause 6.4.6.4
AT_protection	atpr	gen_info	As specified in subclause 6.4.6.5
AG_tile_configuration[]	agtc	gen_info	As specified in subclause 6.4.6.6
if (AT_index_exists) {			As specified in subclause 6.4.3.2
AT_index	atix	gen_info	As specified in subclause 6.5.2.4
}			
main_attribute_group	agcn	gen_info	As specified in subclause 6.4.7
aux_attribute_group[]	agcn	gen_info	As specified in subclause 6.4.7
}			

6.4.6.2 Annotation table header

6.4.6.2.1 General

This is a mandatory box (*athd* in Table 41) containing the annotation table header.

6.4.6.2.2 Syntax

Table 41 — Annotation table header syntax

Syntax	Key	Type	Remarks
annotation_table_header {	*athd*		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in subclause 6.4.6.2.3
if (is_transport_mode) {			
dataset_group_ID		u(8)	
dataset_ID		u(16)	
}			
AT_ID		u(8)	
AT_type		u(4)	
AT_subtype		u(4)	
AT_subtype_major_version		u(4)	
AT_subtype_minor_version		u(4)	
AT_name		st(v)	Limited to 260 characters, excluding the NULL terminator
AT_version		u(8)	
AT_coord_size		u(2)	
AT_genomic_pos_sort_mode		u(2)	
AT_pos_40_bits_flag		u(1)	
parameter_set_ID		u(8)	
n_aux_attribute_groups		u(3)	
AG_info_exists		u(1)	
if (AG_info_exists) {			
for (i=0; i<=n_aux_attribute_groups; i++) {			
AG_class[i]		u(3)	
AG_group_name[i]		st(v)	Limited to 260 characters, excluding the NULL terminator
n_fields_AG[i]		u(16)	
for (j = 0; j < n_fields_AG[i]; j++) {			
is_attribute[i][j]		u(1)	
if (is_attribute[i][j]) {			
field_ID[i][j]		u(16)	

Table 41 *(continued)*

Syntax	Key	Type	Remarks
`            } else {`			
`              field_ID[i][j]`		u(7)	
`            }`			
`          }`			
`        }`			
`      }`			
`  while(!byte_aligned())`			As specified in 4.2.3
`    nesting_zero_bit`		f(1)	
`}`			

6.4.6.2.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each annotation table header to be properly assembled after transport:

dataset_group_ID is the identifier of the dataset group containing the dataset to which this annotation table header belongs.

dataset_ID is the identifier of the dataset containing this annotation table header.

AT_ID is the identifier of the annotation table to which this header belongs. Its value shall be one of the values in AT_ID[] of the dataset header.

AT_type is the type of the genomic data encoded in the annotation table. Possible values are specified in Table 8.

AT_subtype is the subtype of the genomic data encoded in the annotation table. Possible values are specified in Table 9.

AT_subtype_major_version is the major version of the annotation table subtype. If no subtype is provided, the default value 0 should be used.

AT_subtype_minor_version is the major version of the annotation table subtype. If no subtype is provided, the default value 0 should be used.

AT_name is the name of the annotation table. The maximum allowed size is 64 characters.

AT_version is the version of the annotation table for keeping track of data updates.

AT_coord_size is a value that indicates the number of bits required to represent the row/column coordinates of the annotation table (see Table 42).

Table 42 — Annotation table coordinate size syntax

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

AT_genomic_pos_sort_mode is a value that indicates whether the annotation data in the main attribute group (AG_class == 0) and, if two-dimensional, also the data associated with the rows (AG_class == 1) and columns (AG_class == 2) of the main attribute group are sorted by genomic position. If that is the case, a genomic range index shall be available for specifying the genomic range(s) of each tile in the dimension(s) that are sorted by genomic position (see Table 43).

Table 43 — Annotation table position sort mode syntax

AT_genomic_pos_sort_mode	Dimensions of main attribute group data sorted by genomic position
0	None
1	Only rows
2	Only columns
3	Both rows and columns

AT_pos_40_bits_flag is set to 1 when the variant positions are expressed as 40 bits integers. Otherwise, all variant positions are expressed as 32 bits integers. In the scope of subclause 6.5.2.4, the value of the variable posSize is set to 32 when pos_40_bits_flag is equal to 0 and set to 40 otherwise.

parameter_set_ID is the identifier of the dataset parameter set, specified in the dataset containing this annotation table, required for the decoding of the annotation access units

n_aux_attribute_groups specifies the number of auxiliary attribute groups in the annotation table.

AG_info_exists is a flag, if set to one, indicates that additional information about the attribute group exists.

For each attribute group i, if AG_info_exists == 1, the associated attributes and descriptors, and their order of presentation are specified with the following fields:

> **AG_class**[i] is the class of the attribute group as specified in subclause 6.4.7.2.
>
> **AG_name**[i] is the name of the attribute group.
>
> **n_fields_AG**[i] is the number of attributes and descriptors in the attribute group.
>
> For each descriptor/attribute j of attribute group i,
>
>> **is_attribute**[i][j] is a flag, and if set to 1, indicates that field_ID[i][j] is the ID of an attribute. Otherwise, it is the ID of a descriptor.
>>
>> **field_ID**[i][j] is the ID of the attribute or descriptor, coded respectively as 16-bit or 7-bit unsigned integers.

6.4.6.3 Annotation table metadata

6.4.6.3.1 General

This is an optional box (*atmd* in Table 44) containing metadata associated to the annotation table.

6.4.6.3.2 Syntax

Table 44 — Annotation table metadata syntax

Syntax	Key	Type	Remarks
AT_metadata {	*atmd*		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in subclause 6.4.6.2.3
if (is_transport_mode) {			
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AT_ID		u(8)	
}			
ATMD_general_exists		u(1)	
if (ATMD_general_exists) {			
ATMD_general_size		u(16)	
ATMD_general()			As specified in ISO/IEC 23092-3
}			
ATMD_analytics_exists		u(1)	
if (ATMD_analytics_exists) {			
ATMD_analytics_size		u(16)	
ATMD_analytics()			As specified in ISO/IEC 23092-3
}			
ATMD_linkages_exists		u(1)	
if (ATMD_linkages_exists) {			
ATMD_linkages_size		u(16)	
ATMD_linkages()			As specified in ISO/IEC 23092-3
}			
ATMD_history_exists		u(1)	
if (ATMD_history_exists) {			
ATMD_history_size		u(16)	
ATMD_history()			As specified in ISO/IEC 23092-3
}			
reserved		u(4)	Trailing zeros for byte alignment
}			

6.4.6.3.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each annotation table metadata to be properly assembled after transport:

dataset_group_ID is the identifier of the dataset group containing the dataset and its child annotation table to which this metadata belongs.

dataset_ID is the identifier of the dataset containing the annotation table to which this metadata belongs.

AT_ID is the identifier of the annotation table containing this metadata.

ATMD_general_exists is a flag, if set to 1, indicates the inclusion of the following fields:

ATMD_general_size specifies the size of ATMD_general() in number of bytes.

ATMD_general() contains annotation table general metadata as specified in ISO/IEC 23092-3.

ATMD_analytics_exists is a flag, if set to 1, indicates the inclusion of the following fields:

ATMD_analytics_size specifies the size of ATMD_analytics() in number of bytes.

ATMD_analytics() contains annotation table analytics metadata as specified in ISO/IEC 23092-3.

ATMD_linkages_exists is a flag, if set to 1, indicates the inclusion of the following fields:

ATMD_linkages_size specifies the size of ATMD_linkages() in number of bytes.

ATMD_linkages() contains annotation table linkage metadata as specified in ISO/IEC 23092-3.

ATMD_history_exists is a flag, if set to 1, indicates the inclusion of the following fields:

ATMD_history_size specifies the size of ATMD_history() in number of bytes.

ATMD_history() contains annotation table history metadata as specified in ISO/IEC 23092-3.

6.4.6.4 Annotation table clinical data linkage metadata

6.4.6.4.1 General

This is an optional box (*atcd* in Table 45) containing clinical data linkage metadata associated to the annotation table.

6.4.6.4.2 Syntax

Table 45 — Annotation table clinical data linkage metadata syntax

Syntax	Key	Type	Remarks
AT_CDL {	*atcd*		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in subclause 6.4.6.2.3
if (is_transport_mode) {			
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AT_ID		u(8)	
}			
AT_CDL_value()			As specified in ISO/IEC 23092-3
}			

6.4.6.4.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each annotation table CDL metadata to be properly assembled after transport:

dataset_group_ID is the identifier of the dataset group containing the dataset and its child annotation table to which this CDL metadata belongs.

dataset_ID is the identifier of the dataset containing the annotation table to which this CDL metadata belongs.

AT_ID is the identifier of the annotation table containing this CDL metadata.

AT_CDL_value() contains annotation table clinical data linkage metadata, as specified in ISO/IEC 23092-3.

6.4.6.5 Annotation table protection

6.4.6.5.1 General

This is an optional box (*atpr* in Table 46) containing protection information associated to the annotation table.

When present this box contains information that a decoder needs to properly handle a protected annotation table.

6.4.6.5.2 Syntax

Table 46 — Annotation table protection syntax (correct condition)

Syntax	Key	Type	Remarks
AT_protection {	*atpr*		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in subclause 6.4.6.2.3
if (is_transport_mode) {			
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AT_ID		u(8)	
}			
AT_protection_value()			As specified in ISO/IEC 23092-3
}			

6.4.6.5.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each annotation table protection metadata to be properly assembled after transport:

dataset_group_ID is the identifier of the dataset group containing the dataset and its child annotation table to which this protection metadata belongs.

dataset_ID is the identifier of the dataset containing the annotation table to which this protection metadata belongs.

AT_ID is the identifier of the annotation table containing this protection metadata.

AT_protection_value() contains annotation table protection information, as specified in ISO/IEC 23092-3.

6.4.6.6 Attribute group tile configuration

6.4.6.6.1 General

This is a box (*agtc* in Table 47) that contains the tile configuration associated with an attribute group referenced by AG_class. It includes the definition of one or multiple tile structures based on their ranges of row and column (if two-dimensional) indexes in the annotation table, and the byte-offset pointers to individual annotation access units and their subsidiary payload blocks.

6.4.6.6.2 Syntax

Table 47 — Attribute group tile configuration syntax

Syntax	Key	Type	Remarks
attribute_group_tile_configuration {	agtc		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in subclause 6.4.6.2.3
if (is_transport_mode) {			
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AT_ID		u(8)	
}			
AG_class		u(3)	
attribute_dependent_tiles		u(1)	
default_tile_structure	adts	gen_info	As specified in subclause 6.4.6.7
if (attribute_dependent_tiles) {			
n_add_tile_structures		u(16)	
for (i=0; i<n_add_tile_structures; i++) {			
n_attributes[i]		u(16)	
for (j=0; j<n_attributes[i]; j++){			
attribute_ID[i][j]		u(16)	
}			
n_descriptors[i]		u(7)	
for (j=0; j<n_descriptors[i]; j++){			
descriptor_ID[i][j]		u(7)	
}			

Table 47 *(continued)*

Syntax	Key	Type	Remarks
`        additional_tile_structure[i]`	adts	gen_info	As specified in subclause 6.4.6.7
`      }`			
`    }`			
`    attribute_data_byte_offset`	adbo	gen_info	As specified in subclause 6.5.2.3
`    while (!byte_aligned())`			As specified in subclause 4.2.3
`      **nesting_zero_bit**`		f(1)	
`}`			

6.4.6.6.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each attribute group tile configuration to be properly assembled after transport:

> **dataset_group_ID** is the identifier of the dataset group containing the dataset and its child annotation table to which this tile configuration belongs.
>
> **dataset_ID** is the identifier of the dataset containing the annotation table to which this tile configuration belongs.
>
> **AT_ID** is the identifier of the annotation table containing this tile configuration.

AG_class is the class of the attribute group to which this tile configuration applies as specified in subclause 6.4.7.2.

attribute_dependent_tiles is a flag, and if set to 1, indicates there exist certain attributes and descriptors with tile structures different from the default. Otherwise, all attributes and descriptors in the same attribute group share the same default tile structure. Note that if attribute_contiguity (as specified in subclause 6.4.7.2) of the attribute group identified by AG_class is 0, attribute_dependent_tiles shall always be set to 0.

default_tile_structure is the default tile structure of all attributes and descriptors belonging to the same attribute group.

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

> **n_add_tile_structures** is the number of additional tile structures for the attribute group
>
> **n_attributes**[i] is the number of attributes sharing the ith additional tile structure
>
> **attribute_ID**[i][j] is the jth attribute ID associated with the ith additional tile structure
>
> **n_descriptors**[i] is the number of descriptors sharing the ith additional tile structure
>
> **descriptor_ID**[i][j] is the jth descriptor ID associated with the ith additional tile structure
>
> additional_tile_structure[i] is the ith additional tile structure

attribute_data_byte_offset is a data structure that contains the byte-offset pointers to the annotation access units and their subsidiary payload blocks.

6.4.6.7 Attribute data tile structure

6.4.6.7.1 General

This is a box (*adts* in Table 48) specifying how descriptor/attribute data should be divided into rectangular tiles defined by ranges of row and column indexes.

6.4.6.7.2 Syntax

Table 48 — Attribute data tile structure syntax

Syntax	Key	Type	Remarks
attribute_data_tile_structure {	*adts*		
reserved		u(7)	
variable_size_tiles		u(1)	
n_tiles		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
if (variable_size_tiles) {			
for (i = 0; i < n_tiles; i++) {			
for (j = 0; j <= two_dimensional; j++) {			two_dimensional is defined in the header of the associated attribute group as specified in subclause 6.4.7.2
start_index[i][j]		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
end_index[i][j]		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
}			
}			
} else {			
for (j = 0; j <= two_dimensional; j++)			
tile_size[j]		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
}			
}			
}			

6.4.6.7.3 Semantics

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

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

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

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

6.4.7 Attribute Group

6.4.7.1 General

Attribute group is a container box that allows attributes and descriptors to be grouped by their roles, e.g. main data, row information, column information, etc., in the annotation table. The container box (*agcn* in Table 49) is mandatory in data format, and forbidden in transport format.

There are two ways to organize data payloads in an attribute group:

If `attribute_contiguity` equals 1, payload blocks are grouped into annotation access units by descriptor/attribute, and ordered by their tile positions within an annotation access unit.

If `attribute_contiguity` equals 0, payload blocks are grouped into annotation access units by tile, which are then ordered by tile positions within an attribute group.

It should be noted that for two-dimensional tiles, either row-major or column-major ordering can be applied depending on the value of column_major_tile_order as specified in subclause 6.4.7.2.

Table 49 — Attribute group syntax

Syntax	Key	Type	Remarks
attribute_group {	*agcn*		
attribute_group_header	aghd	gen_info	As specified in subclause 6.4.7.2
annotation_access_units[]	aauc	gen_info	As specified in subclause 6.4.8
}			

6.4.7.2 Attribute Group Header

6.4.7.2.1 General

This is a mandatory box (*aghd* in Table 50) describing the content of an attribute group.

6.4.7.2.2 Syntax

Table 50 — Attribute group header syntax

Syntax	Key	Type	Remarks
attribute_group_header {	*aghd*		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in subclause 6.4.7.2.3
if (is_transport_mode) {			
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AT_ID		u(8)	
}			
AG_class		u(3)	
attribute_contiguity		u(1)	
two_dimensional		u(1)	
if (two_dimensional) {			
reserved		u(6)	

Table 50 *(continued)*

Syntax	Key	Type	Remarks
column_major_tile_order		u(1)	
symmetry_mode		u(3)	
symmetry_minor_diagonal		u(1)	
} else {			
reserved		u(3)	
}			
for (i = 0; i <= two_dimensional; i++)			
dimension_size[i]		u(v)	v dependends on AT_coord_size as specified in subclause 6.4.6.2
}			
}			

6.4.7.2.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each attribute group header to be properly assembled after transport:

dataset_group_ID is the identifier of the dataset group containing the dataset and its child annotation table to which this attribute group header belongs.

dataset_ID is the identifier of the dataset containing the annotation table to which this attribute group header belongs.

AT_ID is the identifier of the annotation table containing this attribute group header.

AG_class specifies the functional class of this attribute group. The possible values are:

0 – main attribute data

1 – auxiliary attribute data mapped to the rows of the main data

2 – auxiliary attribute data mapped to the columns of the main data

3 – auxiliary linkage attributes mapped to the rows of the main data

4 – auxiliary linkage attributes mapped to the columns of the main data

5-7 – any auxiliary attributes defined by the user

Within an annotation table, there shall be a main attribute group, and no more than one instance of each class of auxiliary attribute group. For auxiliary attribute group classes 1-4, if the data is two-dimensional, the mapping is always between the rows of the auxiliary data and the rows (for classes 1 and 3) or columns (for classes 2 and 4) of the main data. For auxiliary attribute group classes 3 and 4, it can consist of multiple linkage attributes that specify respectively the URIs to other files, and the dataset group IDs and dataset IDs that point to specific MPEG-G data objects to which the rows/columns of the main data are linked.

attribute_contiguity is a flag, if set to 1, indicates that the payload blocks are grouped into annotation access units by descriptor/attribute. Otherwise, payload blocks are grouped into annotation access units by tile. In both cases, the same syntax of the annotation access unit as defined in subclause 6.4.8 applies.

two_dimensional is a flag, and if set to 1, indicates that all descriptors and attributes in the group are 2-dimensional. Otherwise, all descriptors and attributes in the group are 1-dimensional.

column_major_tile_order is a flag that is only relevant for a descriptor/attribute when two_dimensional == 1 and variable_size_tiles == 0 in the corresponding attribute data tile structure as specified in subclause 6.4.6.7. If set to 1, it indicates the tiles are in column-major order. Otherwise, they are in row-major order.

symmetry_mode specifies the symmetry mode of the attribute group data and is only effective when two_dimensional == 1. The possible values are specified in Table 51.

Table 51 — Symmetry mode

symmetry_mode	type	remarks
0	non symmetric	any rectangular matrix
1	symmetric	only for squared matrices
2	skew-symmetric	symmetric with only zero’s along the diagonal
3-7	reserved for future use.	

For symmetry modes 1-3, descriptor/attribute values in the reflected half to the right of the principal/minor diagonal (inclusive of the diagonal if skew-symmetric) shall be processed as missing values. A suggested usage of the symmetry mode and related settings for the efficient handling of symmetric annotation data is provided in Annex E.

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

dimension_size[i] specifies the total number of rows (i == 0) or columns (i == 1) in the data of each descriptor/attribute within the group when two_dimensional == 1, or simply the number of elements when two_dimensional == 0. For example, the genotypes of variant calls and their likelihoods are 2-dimensional attributes, with variants in rows and samples in columns. Therefore, the first and second elements of dimension_size should be respectively the total number of variants and the total number of samples. In the case of transport when data generation is ongoing, a value of 0 can be applied. At the completion of data generation and transport, the value(s) of dimension_size[] should be computed and reassigned using the process as specified in subclause 6.7.2.

6.4.8 Annotation access unit

6.4.8.1 General

The annotation access unit is a collection of one or more blocks representing genomic annotation data. The decapsulation of this box (*aauc* in Table 52) shall result in a data unit, as specified in subclause 6.3, with:

— data_unit_type equal to 4,

— data_unit_size is computed according to the following expression: length in annotation_access_unit gen_info – length in AAU_protection gen_info,

— as payload the annotation access unit specified in ISO/IEC 23092-6 containing the annotation access unit header and the blocks.

Such data unit can be dispatched to a decoder compliant with ISO/IEC 23092-6, along with the annotation parameter set and annotation indexes that are needed to decode it.

Table 52 — Annotation access unit syntax

Syntax	Key	Type	Remarks
annotation_access_unit {	*aauc*		
AAU_header	aauh	gen_info	As specified in subclause 6.4.8.3
AAU_protection	aaup	gen_info	As specified in subclause 6.4.8.4
for (i=0;i<num_blocks;i++) {			As specified in subclause 6.4.8.3
AAU_block[i]			As specified in subclause 6.4.9
}			
}			

6.4.8.2 Annotation access unit decapsulation

Output of this process is an annotation access unit structure composed of:

— An annotation access unit header structure as specified in subclause 6.4.8.3.

— The set of annotation blocks belonging to the annotation access unit, as specified in subclause 6.4.9. The content of each block should be considered its original, compressed content if the *indexed* flag defined in subclauses 6.4.9.2 and 6.4.8.3 is set to 0, and the uncompressed content extracted from the string index if the *indexed* flag it set to to 1. The block shall be decompressed as specified in subclause 1.

6.4.8.3 Annotation access unit header

6.4.8.3.1 General

This is a mandatory box (*aauh* in Table 53) describing the content of an annotation access unit.

6.4.8.3.2 Syntax

Table 53 — Annotation access unit header syntax

Syntax	Key	Type	Remarks
annotation_access_unit_header {	*aauh*		
is_transport_mode			A computed boolean value, if equals 1, indicates transport mode is enabled, as specified in 6.4.8.3.3
if (is_transport_mode) {			
reserved		u(5)	
dataset_group_ID		u(8)	
dataset_ID		u(16)	
AT_ID		u(8)	
AG_class		u(3)	
}			
if (attribute_contiguity) {			As specified in subclause 6.4.7.2
is_attribute		u(1)	
if (is_attribute) {			
reserved		u(7)	
attribute_ID		u(16)	

Table 53 *(continued)*

Syntax	Key	Type	Remarks
`    } else {`			
`      `**descriptor_ID**		u(7)	
`    }`			
`    if (two_dimensional && !variable_size_tiles) {`			two_dimensional as specified in subclause 6.4.7.2 variable_size_tiles as specified in subclause 6.4.6.7 of the tile structure associated with the descriptor/attribute
`      if (column_major_tile_order) {`			
`        `**n_tiles_per_col**		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
`      } else {`			
`        `**n_tiles_per_row**		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
`      }`			
`    }`			
`    `**n_AAU_blocks**		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
`  } else {`			
`    `**tile_index_1**		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
`    `**reserved**		u(7)	
`    `**tile_index_2_exists**		u(1)	
`    if (tile_index_2_exists) {`			
`      `**tile_index_2**		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2
`    }`			
`    `**n_AAU_blocks**		u(16)	
`  }`			
`  while(!byte_aligned())`			As specified in subclause 4.2.3
`    `**nesting_zero_bit**		f(1)	
`}`			

6.4.8.3.3 Semantics

is_transport_mode is a computed boolean value, if equals 1, indicates the transport format is applied. Otherwise, the file format is used. It can be evaluated by checking if the dataset_mapping_table_list, as specified in subclause 6.6.3, is present (is_transport_mode == 1) or not (is_transport_mode == 0).

The following trail of upper-level container IDs are only included in the transport format to allow each annotation access unit to be properly assembled after transport:

dataset_group_ID is the identifier of the dataset group containing the dataset, its child annotation table and corresponding attribute group to which this annotation access unit belongs.

dataset_ID is the identifier of the dataset containing the annotation table and corresponding attribute group to which this annotation access unit belongs.

AT_ID is the identifier of the annotation table containing the attribute group to which this annotation access unit belongs.

AG_class is the class of the attribute group containing this annotation access unit.

If attribute_contiguity == 1,

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

If is_attribute == 1,

attribute_ID is the ID of one of the attributes, defined in the annotation_encoding_parameters() (ISO/IEC 23092-6) of dataset parameter set (subclase 6.4.3.7).

If is_attribute == 0,

descriptor_ID is the ID of one of the descriptors, defined in the annotation_encoding_parameters() (ISO/IEC 23092-6) of dataset parameter set (subclase 6.4.3.7).

If the attribute is two dimensional and variable_size_tiles == 0,

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

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

If attribute_contiguity == 0,

tile_index_1 is the index of the tile whose data is contained in this annotation access unit. If the attribute group is two dimensional and with variable_size_tiles == 0, it corresponds to the row index of the tile.

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

If (tile_index_2_exists == 1),

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

n_AAU_blocks is the number of AAU blocks, as specified in subclause 6.4.9. It can be set to 0 if all blocks in the annotation access unit consist of empty/missing values.

6.4.8.4 Annotation access unit protection

6.4.8.4.1 General

This is an optional box (*aaup* in Table 54) containing protection information associated to the annotation access unit.

When present this box contains information that a decoder needs to properly handle a protected annotation access unit.

6.4.8.4.2 Syntax

Table 54 — Annotation access unit protection syntax

Syntax	Key	Type	Remarks
AAU_protection {	*aaup*		
AAU_protection_value()			As specified in ISO/IEC 23092-3
}			

6.4.8.4.3 Semantics

AAU_protection_value() contains annotation access unit protection information, as specified in ISO/IEC 23092-3.

6.4.9 AAU block

6.4.9.1 General

An AAU block (see Table 55) is composed of an AAU block header (as specified in subclause 6.4.9.2) and a block payload, containing compressed data of an attribute or descriptor in a specific tile.

Table 55 — AAU block syntax

Syntax	Key	Type	Remarks
aau_block {	*aau_block*		
aau_block_header		*aau_block_header*	As specified in subclause 6.4.9.2
for (i = 0; i < block_payload_size; i++) {			block_payload_size as specified in subclause 6.4.9.2
block_payload[i]		u(8)	
}			
}			

block_payload[i] is the ith byte of block payload, which contains a compressed tile of data in an attribute or descriptor.

6.4.9.2 AAU Block header

6.4.9.2.1 General

This box (see Table 56) contains information associated to the AAU block.

This box shall replace the block header provided by the underlying codec and specified in ISO/IEC 23092-6.

6.4.9.2.2 Syntax

Table 56 — AAU block header syntax

Syntax	Key	Type	Remarks
AAU_block_header {	*aau_block_header*		
if (attribute_contiguity == 0) {			As specified in subclause 6.4.7.2
descriptor_ID		u(8)	
if (descriptor_ID == 31) {			
attribute_ID		u(16)	
}			
}			
reserved		u(1)	
indexed		u(1)	
encrypted		u(1)	
block_payload_size		u(29)	
}			

6.4.9.2.3 Semantics

descriptor_ID is the descriptor identifier, as specified in ISO/IEC 23092-6.

attribute_ID specifies the attribute identifier if descriptor_ID is set to 31 , as specified in ISO/IEC 23092-6.

indexed is set to 1 if the block has been compressed with a string index, as specified in subclause 7.1.4, and 0 otherwise

encrypted is a flag, if set to 1, indicates the block payload is encrypted. Otherwise, it is not encrypted. The decryption mechanism is described in ISO/IEC 23092-3:2023.

block_payload_size is the number of bytes composing the block payload. It can be set to 0 if all values in the block payload are empty/missing.

6.5 Data structures specific to file format

6.5.1 General

This subclause specifies additional data structures specific to the indexed storage of genomic information. They complement the data structures specified in subclause 6.4.

6.5.2 Indexing

6.5.2.1 Master index table

6.5.2.1.1 General

The master index table (*mitb* in Table 57) provides the indexing information needed to perform selective access on specific parts of the genomic sequencing dataset.

It is present in the dataset when MIT_flag, as specified in subclause 6.5.3.2, is equal to 1. It is not present otherwise.

The first part of the master index table shall be ordered by increasing AU_start_position[seq_ID][ci][au_id] values; the second part of the master index table shall be ordered by increasing U_cluster_signature[uau_id][0] values.

The special value ((1<<byteOffsetSize)-1) assigned to AU_byte_offset[seq_ID][ci][au_id] represents an empty access unit. It is used to maintain synchronization among access units having different AU_type but covering the same genomic range.

The special value ((1<<byteOffsetSize)-1) assigned to block_byte_offset[seq_ID][ci][au_id][di] represents an empty block. It is used to maintain synchronization among blocks belonging to the same access unit.

6.5.2.1.2 Syntax

Table 57 — Master index table syntax

Syntax	Key	Type	Remarks
master_index_table {	*mitb*		
for (seq = 0; seq < seq_count; seq++) {			seq_count as specified in subclause 6.4.3.2
for (ci = 0; ci < num_classes; ci++) {			num_classes as specified in subclause 6.4.3.2
if (clid[ci] != CLASS_U) {			CLASS_U value as specified in subclause 5.3. clid as specified in subclause 6.4.3.2
for (au_id = 0; au_id < seq_blocks[seq]; au_id++) {			seq_blocks as specified in subclause 6.4.3.2
AU_byte_offset[seq][ci][au_id]		u(byteOffsetSize)	byteOffsetSize as specified in subclause 6.4.3.2
AU_start_position[seq][ci][au_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
AU_end_position[seq][ci][au_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
if (dataset_type == 2) {			As specified in subclause 6.4.3.2
ref_sequence_id[seq][ci][au_id]		u(16)	
ref_start_position[seq][ci][au_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
ref_end_position[seq][ci][au_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
}			
if (multiple_alignment_flag) {			As specified in subclause 6.4.3.2
extended_AU_start_position[seq][ci][au_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
extended_AU_end_position[seq][ci][au_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
}			
if (!block_header_flag) {			As specified in subclause 6.4.3.2
for (di = 0; di < num_descriptors[ci]; di++) {			num_descriptors as specified in subclause 6.4.3.2
block_byte_offset[seq][ci][au_id][di]		u(byteOffsetSize)	byteOffsetSize as specified in subclause 6.4.3.2

Table 57 *(continued)*

Syntax	Key	Type	Remarks
}			
}			
}			
}			
}			
}			
for (uau_id = 0; uau_id < num_U_access_units; uau_id++) {			num_U_access_units as specified in subclause 6.4.3.2
AU_byte_offset[uau_id]		u(byteOffsetSize)	byteOffsetSize as specified in subclause 6.4.3.2
if (dataset_type == 2) {			As specified in subclause 6.4.3.2
U_ref_sequence_id[uau_id]		u(16)	
U_ref_start_position[uau_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
U_ref_end_position[uau_id]		u(posSize)	posSize as specified in subclause 6.4.3.2.3
} else {			
if (U_signature_flag != 0) {			As specified in subclause 6.4.3.2
num_signatures		u(16)	
for (i = 0; i < num_signatures; i++) {			
if (!U_signature_constant_length) {			As specified in subclause 6.4.3.2
U_signature_length[uau_id][i]		u(8)	
}			
U_cluster_signature[uau_id][i]		u(U_signature_size)	U_signature_size inferred as specified in the U_cluster_signature semantics in subclause 6.4.4.3.3
}			
}			
}			
}			
while(!byte_aligned())			As specified in subclause 4.2.3
nesting_zero_bit		f(1)	
if(!block_header_flag) {			As specified in subclause 6.4.3.2.2
for (di = 0; di < num_descriptors[num_classes - 1]; di++) {			num_descriptors as specified in subclause 6.4.3.2.2
block_byte_offset[uau_id][di]		u(byteOffsetSize)	byteOffsetSize as specified in subclause 6.4.3.2
}			
}			
}			
}			

6.5.2.1.3 Semantics

AU_byte_offset[seq][ci][au_id] is the byte offset of the first byte in the access unit, with respect to the first byte of the Value field of the dataset (dtcn) gen_info structure (0-based). It is equal to ((1<<byteOffsetSize)-1)

if the access unit is empty: in such a case the fields AU_start_position[seq][ci][au_id], AU_end_position[seq][ci][au_id], extended_AU_start_position[seq][ci][au_id], extended_AU_end_position[seq][ci][au_id], ref_sequence_id[seq][ci][au_id], ref_start_position[seq][ci][au_id] and ref_end_position[seq][ci][au_id] shall be ignored.

AU_start_position[seq][ci][au_id] is the position of the left-most mapped base among the first alignments of all genomic records encoded in the access unit irrespective of the strand. AU_start_position[seq][ci][i+1] shall always be greater than or equal to AU_start_position[seq][ci][i].

AU_end_position[seq][ci][au_id] is the position of the right-most mapped base among the first alignments of all genomic records encoded in the access unit irrespective of the strand.

ref_sequence_id[seq][ci][au_id]: in case of an access unit carrying (part of) a reference sequence, it specifies the ID of such reference sequence.

ref_start_position[seq][ci][au_id]: in case of an access unit carrying (part of) a reference sequence, it specifies the position on the reference sequence of the first nucleotide encoded in this access unit.

ref_end_position[seq][ci][au_id]: in case of an access unit carrying (part of) a reference sequence, it specifies the position on the reference sequence of the last nucleotide encoded in this access unit.

extended_AU_start_position[seq][ci][au_id] specifies the position of the left-most mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand.

extended_AU_end_position[seq][ci][au_id] specifies the position of the right-most mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand.

block_byte_offset[seq][ci][au_id][di] is the byte offset of the first byte in the block, with respect to the first byte of the Value field of the dataset (dtcn) gen_info structure (0-based). If the block is empty and block_header_flag is equal to 1, it shall be equal to ((1 << byteOffsetSize) − 1). If the block is empty and block_header_flag is equal to 0, it shall be equal either to the block_byte_offset value of the next block in the descriptor stream or, for the last block in the descriptor stream, to the sum of the block_byte_offset value of the first block in the descriptor stream and the descriptor stream payload size.

NOTE The descriptor stream payload size can be inferred as the L field of the dscn gen_info header, minus the L field of the dshd gen_info header, minus the L field of the dspr gen_info header.

AU_byte_offset[uau_id] is the byte offset of the first byte in the unmapped access unit, with respect to the first byte of the Value field of the dataset (dtcn) gen_info structure (0-based). It is equal to ((1<<byteOffsetSize)-1) if the access unit is empty: in such a case the fields U_ref_sequence_id[uau_id], U_ref_start_position[uau_id] and U_ref_end_position[uau_id] shall be ignored and num_signatures shall be set to 0.

num_U_access_units is the total number of access units in the dataset containing encoded data of class U. It is encoded in the dataset header, as specified in subclause 6.4.3.2.

U_ref_sequence_id[uau_id], in case of an access unit carrying (part of) a reference sequence, is the identifier of such reference sequence.

U_ref_start_position[uau_id], in case of an access unit carrying (part of) a reference sequence, specifies the position on the reference sequence of the left-most nucleotide encoded in this access unit.

U_ref_end_position[uau_id], in case of an access unit carrying (part of) a reference sequence, specifies the position on the reference sequence of the right-most nucleotide encoded in this access unit.

num_signatures is the number of signatures.

U_signature_length[uau_id][i] is the length of cluster signatures as number of nucleotides.

U_cluster_signature[uau_id][i] is the i^{th} signature of the cluster the access unit belongs to. U_cluster_signature[uau_id][i+1] shall always be greater than or equal to U_cluster_signature[uau_id][i]. The length in bits of this field, named U_signature_size in Table 34, shall be calculated as follows:

U_signature_size = signature_length * bits_per_symbol with bits_per_symbol as specified in Table 27, and with signature_length corresponding either to U_signature_length as specified in subclause 6.4.4.3.3 when U_signature_constant_length (as specified in subclause 6.4.4.3.3) is equal to 1 or to the signature-specific U_signature_length[uau_id][i] as specified in Table 57 when U_signature_constant_length (as specified in subclause 6.4.4.3.3) is equal to 0.

The j^{th} nucleotide in a signature shall be inferred as follows:

$S_{alphabet\_ID}$[(U_cluster_signature[i] >> ((signature_length – j – 1) * bits_per_symbol)) & ((1 << bits_per_symbol) – 1)]

with alphabet_id as specified in subclause 6.4.3.2 and $S_{alphabet\_ID}$ as specified in subclause 6.4.3.2.4.

block_byte_offset[uau_id][di]: same semantics as block_byte_offset[seq][ci][au_id][di] above

6.5.2.2 String index

Whenever string information coming from annotation data has been indexed in the file, a string index will be present. The index at the same time stores the text and makes it searchable. Thus, one will need to access the index both in case of a string search and to decode the original string information.

How to query and decode the string index is specified in Clause 0.

6.5.2.3 Attribute data byte offset

6.5.2.3.1 General

This is a box (*adbo* in Table 58) that contains the byte-offset pointers to the annotation access units within the associated attribute group and their subsidiary payload blocks.

6.5.2.3.2 Syntax

Table 58 — Attribute data byte offset syntax

Syntax	Key	Type	Remarks
attribute_data_byte_offset {	*adbo*		
for (i=0; i<n_fields_AG; i++) {			n_fields_AG As specified in subclause 6.4.6.2
is_attribute[i]		u(1)	
if (is_attribute[i])			
attr_desc_ID[i]		u(16)	
else			
attr_desc_ID[i]		u(7)	
}			
for (i=0; i<n_AAUs; i++) {			n_AAUs is the number of annotation access units in the associated attribute group.
AAU_offset[i]		u(byteOffsetSize)	byteOffsetSize as specified in subclause 6.4.3.2
for (j=0; j<n_AAU_blocks[i]; j++) {			n_AAU_blocks, as specified in subclause 6.4.8.3, is the number of blocks in the AAU pointed by AAU_offset[i]

Table 58 *(continued)*

Syntax	Key	Type	Remarks
AAU_block_offset[i][j]		u(byteOffsetSize)	byteOffsetSize as specified in subclause 6.4.3.2
}			
}			
while (!byte_aligned())			As specified in subclause 4.2.3
nesting_zero_bit		f(1)	
}			

6.5.2.3.3 Semantics

The following fields are for specifying the ordering of the attributes or descrioptors for the storage of annotation access units (attribute_contiguity == 1) or payload blocks (attribute_contiguity == 0) within the attribute group:

> **is_attribute**[i] is a flag, and if set to 1, indicates that **attr_desc_ID[i]** is the ID of an attribute. Otherwise, it is the ID of a descriptor.
>
> **attr_desc_ID**[i] is the ID of the ith attribute or descriptor, coded respectively as 16-bit or 7-bit unsigned integers

AAU_offset[i] is the byte offset, counting from the beginning of the associated attribute group container to the ith annotation accesss unit. Its value should be 0 if the annotation access unit does not exist when the payloads are all empty.

AAU_block_offset[i][j] is the byte offset, counting from the beginning of the ith annotation access unit container, to the jth payload block within that container. Note that even for empty payloads, payload_offset shall be included and set to 0.

6.5.2.4 Annotation table index

6.5.2.4.1 General

Annotation Table Index (*atix* in Table 59) consists of one or multiple indexing data structures that support query on the annotation data by row and column indexes, genomic ranges and string values of specific attributes.

6.5.2.4.2 Syntax

Table 59 — Annotation table index syntax

Syntax	Key	Type	Remarks
annotation_table_index {	*atix*		
for (i = 0; i <= n_aux_attribute_groups; i++) {			As specified in subclause 6.4.6.2
AG_class[i]		u(3)	As specified in subclause 6.4.7.2
if (AT_genomic_pos_sort_mode > 0) {			As specified in subclause 6.4.6.2
default_genomic_range_index[i]	grix	gen_info	As specified in subclause 6.5.2.4.4

Table 59 *(continued)*

Syntax	Key	Type	Remarks
`if (attribute_dependent_tiles[i]) {`			As specified in subclause 6.4.6.6
`for (j = 0; j < n_add_tile_structures[i]; j++) {`			As specified in subclause 6.4.6.6
`additional_genomic_range_index[i][j]`	grix	gen_info	As specified in subclause 6.5.2.4.4
`}`			
`}`			
`}`			
`n_attr_value_indexes``[i]`		u(8)	
`for (j = 0; j < n_attr_value_indexes[i]; j++) {`			
`attribute_value_index[i][j]`	avix	gen_info	As specified in subclause 6.5.2.4.5
`}`			
`}`			
`while (!byte_aligned())`			As specified in subclause 4.2.3
`nesting_zero_bit`		f(1)	
`}`			

6.5.2.4.3 Semantics

The following indexing data is specified for each attribute group i (as defined in subclause 6.4.7) in the annotation table (i = 0 for the main attribute group and i > 0 for the auxiliary attribute groups):

AG_class[i] is the functional class of attribute group i as defined in subclause 6.4.7.2.

default_genomic_range_index[i] is the genomic range index corresponding to default_tile_structure (defined in subclause 6.4.6.6) of attribute group i.

additional_genomic_range_index[i][j] is the genomic range index corresponding to additional_tile_structure[j] (defined in subclause 6.4.6.6) of attribute group i.

n_attr_value_indexes[i] is the number of attribute value indexes for attribute group i.

attribute_value_index[i][j] is the jth attribute value index for attribute group i.

6.5.2.4.4 Genomic range index

6.5.2.4.4.1 General

This is a box (*grix* in Table 60) that contains the genomic range index associated with an attribute data tile structure (subclause 6.4.6.7). It specifies the genomic range(s) of each tile in the dimension(s) that are sorted by genomic coordinate (chromosome and position).

6.5.2.4.4.2 Syntax

Table 60 — Genomic range_index syntax

Syntax	Key	Type	Remarks
genomic_range_index {	*grix*		
index_2d		u(1)	
for (i = 0; i <= index_2d; i++) {			
n_chr[i]		u(16)	
for (j = 0; j < n_chr[i]; j++) {			
chr_ID[i][j]		st(v)	
}			
}			
if (!variable_size_tiles) {			variable_size_tiles as specified in subclause 6.4.6.7
for (i = 0; i <= index_2d; i++) {			
tile_count[i]		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2
for (j = 0; j < tile_count[i]; j++) {			
start_chr_idx[j][i]		u(16)	
start_pos[j][i]		u(posSize)	posSize as specified in subclause 6.4.6.2
end_chr_idx[j][i]		u(16)	
end_pos[j][i]		u(posSize)	posSize as specified in subclause 6.4.6.2
}			
}			
} else {			
tile_count		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2
for (i = 0; i < tile_count; i++) {			
for (j = 0; j <= index_2d; j++) {			
start_chr_idx[i][j]		u(16)	
start_pos[i][j]		u(posSize)	posSize as specified in subclause 6.4.6.2
}			
for (j = 0; j <= index_2d; j++) {			
end_chr_idx[i][j]		u(16)	
end_pos[i][j]		u(posSize)	posSize as specified in subclause 6.4.6.2
}			
}			
}			
}			

6.5.2.4.4.3 Semantics

index_2d is a flag, if set to 1, indicates this genomic range index applies to both the rows and columns of the associated annotation data. Otherwise, it only applies to one dimension. It shall be set to 1 if the following conditions hold:

two_dimensional == 1,

AT_genomic_pos_sort_mode == 3, and

!(symmetry_mode > 0 && !variable_size_tiles && tile_size[0] == tile_size[1]).

Note that `two_dimensional` and `symmetry_mode` are specified in subclause 6.4.7.2, `AT_genomic_pos_sort_mode` in subclause 6.4.6.2, and `variable_size_tiles` and `tile_size` in subclause 6.4.6.7. The third condition is to exclude the case where the data is symmetric and split into square tiles of a uniform size, thus resulting in identical genomic ranges in both dimensions.

n_chr`[i]` is the number of chromosomes defined in the annotation data in the row/column dimension for which the genomic ranges of the tiles are being specified. If `index_2d == 1`, `i == 0` shall correspond to the row dimension, and `i == 1` shall correspond to the column dimension.

chr_ID`[i][j]` is the jth chromosome IDs in the row/column dimension, in the order as they appear in the annotation data.

tile_count`[i]` is, in the case of `variable_size_tiles == 0`, the number of tiles in the row/column dimension for which the genomic ranges are being specified; whereas in the case of `variable_size_tiles == 1`, it is the total number of tiles, `n_tiles`, as specified in subclause 6.4.6.7. The process for the assignment of value(s) to `tile_count[]` is specified in Table 61.

Table 61 — Process for assigning value(s) to tile_count

Syntax	Remarks

`if (variable_size_tiles == 1 ||`
`    two_dimensional == 0) {`
	variable_size_tiles as specified in subclause 6.4.6.7; two_dimensional as specified in subclause 6.4.7.2

`  tile_count = n_tiles`	n_tiles as specified in subclause 6.4.6.7
`} else {`	
`  if (attribute_contiguity) {`	attribute_contiguity as specified in subclause 6.4.7.2
`    if (column_major_tile_order) {`	column_major_tile_order as specified in subclause 6.4.7.2
`      n_tiles_1 = n_tiles_per_col`	n_tiles_per_col as specified in subclause 6.4.8.3
`      n_tiles_2 = Ceil(n_AAU_blocks / n_tiles_1)`	n_AAU_blocks as specified in subclause 6.4.8.3 *Ceil(x)* is a function that returns the smallest integer ≥ x
`    } else {`	
`      n_tiles_2 = n_tiles_per row`	n_tiles_per_row as specified in subclause 6.4.8.3
`      n_tiles_1 = Ceil(n_AAU_blocks / n_tiles_2)`	
`    }`	
`  else {`	
`    n_tiles_1 = max_tile_idx_1()`	max_tile_idx_1() returns the maximum value of tile_index_1, as specified in subclause 6.4.8.3, among all annotation access units.

Table 61 *(continued)*

Syntax	Remarks
`    n_tiles_2 = max_tile_idx 2()`	`max_tile_idx_2() returns the maximum value of tile_index_2, as specified in subclause 6.4.8.3, among all annotation access units.`
`  }`	
`  if (AT_genomic_pos_sort_mode == 1) {`	`AT_genomic_pos_sort_mode as specified in subclause 6.4.6.2`
`    tile_count = n_tiles_1`	
`  } else if (AT_genomic_pos_sort_mode == 2) {`	
`    tile_count = n_tiles_2`	
`  } else if (AT_genomic_pos_sort_mode == 3) {`	
`    if (index_2d == 0) {`	
`      tile_count = n_tiles_1`	
`    } else {`	
`      tile_count[0] = n_tiles_1`	
`      tile_count[1] = n_tiles_2`	
`    }`	
`  }`	
`}`	

(**start_chr_idx**[i][j], **start_pos**[i][j]) specifies the start (top/leftmost) genomic coordinate, i.e. chromosome and position, of a tile. For variable_size_tiles == 0, i corresponds to the row/column tile index; whereas for variable_size_tiles == 1, i corresponds to the overall tile index. For index_2d == 1, j == 0 refers to the row dimension and j == 1 refers to the column dimension.

(**end_chr_idx**[i][j], **end_pos**[i][j]) specifies the end (bottom/rightmost) genomic coordinate, i.e. chromosome and position, of a tile. For variable_size_tiles == 0, i corresponds to the row/column tile index; whereas for variable_size_tiles == 1, i corresponds to the overall tile index. For index_2d == 1, j == 0 refers to the row dimension and j == 1 refers to the column dimension.

6.5.2.4.5 Attribute value index

6.5.2.4.5.1 General

This is an optional box (*avix* in Table 62) containing information and data of a supplementary index that supports query by descriptor/attribute values.

6.5.2.4.5.2 Syntax

Table 62 — Attribute value index syntax

Syntax	Key	Type	Remarks
`attribute_value_index {`	*`avix`*		
`  `**`n_index_descriptors`**		`u(8)`	
`  for (i = 0; i < n_index_descriptors; i++)`			
`    `**`index_descriptor_ID`**`[i]`		`u(8)`	
`  `**`n_index_attributes`**		`u(8)`	
`  for (i = 0; i < n_index_attributes; i++)`			

Table 62 *(continued)*

Syntax	Key	Type	Remarks
`    index_attribute_ID[i]`		u(16)	
`  compress`		u(1)	
`  if (compress)`			
`    compress_algorithm_ID`		u(8)	
`  index_type`		u(7)	
`  if (index_type == 0) {`			Set to n_index_descriptors + n_index_attributes
`    num_strings = n_index_descriptors + n_index_attributes`			
`    master_string_index(num_strings, 0)`			As specified in subclause 7
`  } else if (index_type == 1) {`			
`    b_tree_index()`			As specified in subclause 8.1
`  } else if (index_type == 2) {`			
`    master_string_index(num_strings, 1)`			
`  }`			
`}`			

6.5.2.4.5.3 Semantics

n_index_descriptors is the number of descriptors associated with this attribute value index.

index_descriptor_ID[i] is the ID of a descriptor whose data values can be queried using this attribute value index. It shall be one of the descriptor IDs in annotation_encoding_parameters() of the dataset parameter set (subclause 6.4.3.7) referenced by parameter_set_ID in the annotation table header (subclause 6.4.6.2).

n_index_attributes is the number of attributes associated with this attribute value index.

index_attribute_ID[i] is the ID of an attribute whose data values can be queried using this attribute value index. It shall be one of the attribute IDs in annotation_encoding_parameters() of the dataset parameter set (subclause 6.4.3.7) referenced by parameter_set_ID in the annotation table header (subclause 6.4.6.2).

compress is a flag, if set to 1, indicates the index data following index_type is compressed by the algorithm referenced by compress_algorithm_ID. Otherwise, the index data is uncompressed.

compress_algorithm_ID specifies the ID of the compression algorithm being applied on the index data if compress == 1. Its value shall be one of the values of encoding_mode_ID defined in ISO/IEC 23092-2, except for the value 0 which corresponds to the CABAC algorithm.

index_type specifies the type of this attribute value index. Possible values include 0 for string index, 1 for B-Tree index, 2 for string index with tiles. Other values are reserved for future use.

6.5.3 Descriptor stream

6.5.3.1 General

A descriptor stream is a stream of data of a certain class and descriptor, encoded as described in ISO/IEC 23092-2.

This is a mandatory box (*dscn* in Table 63) when the syntax element block_header_flag in the dataset header, as specified in subclause 6.5.3.2, is equal to 0; it is forbidden otherwise.

Child boxes may be present or not, according to the column “Mandatory” in Table 10.

Table 63 — Descriptor stream syntax

Syntax	Key	Type	Remarks
descriptor_stream {	*dscn*		
descriptor_stream_header	dshd	gen_info	As specified in subclause 6.5.3.2
DS_protection	dspr	gen_info	As specified in subclause 6.5.3.3
for (i = 0; i < num_blocks; i++) {			num_blocks as specified in subclause 6.5.3.2
for (j = 0; j < block_payload_size[i]; j++) {			
block_payload[i][j]		u(8)	As specified in ISO/IEC 23092-2
}			
}			
}			

block_payload_size[i] is inferred from the master index table field block_byte_offset, as specified in subclause 6.5.2.1, as difference between either block_byte_offset[i+1] or the variable descriptor_stream_size, as described in subclause 6.4.3.2.5, and block_byte_offset[i].

block_payload[i][j] is the jth byte of the block payload.

6.5.3.2 Descriptor stream header

6.5.3.2.1 General

This is a box (*dshd* in Table 64) describing a descriptor stream. It is mandatory whenever the descriptor stream, as specified in subclause 6.5.3, is present, forbidden otherwise.

6.5.3.2.2 Syntax

Table 64 — Descriptor stream header syntax

Syntax	Key	Type	Remarks
descriptor_stream_header {	*dshd*		
reserved		u(1)	
descriptor_ID		u(7)	
class_ID		u(4)	
num_blocks		u(32)	
while(!byte_aligned())			As specified in subclause 4.2.3
nesting_zero_bit		f(1)	Equal to 0
}			

6.5.3.2.3 Semantics

descriptor_ID identifies the type of compressed descriptors carried by the descriptor stream. It shall have one of the values specified as descriptor_ID[ci][di] in subclause 6.5.3.2.

class_ID identifies the class of data carried by the block, as specified in Table 6.

num_blocks is the number of blocks composing the descriptor stream.

6.5.3.3 Descriptor stream protection

6.5.3.3.1 General

This is an optional box (dspr in Table 65) containing protection information associated to a descriptor stream.

When present this box contains information that a decoder needs to properly handle a protected descriptor stream.

6.5.3.3.2 Syntax

Table 65 — Descriptor stream protection syntax

Syntax	Key	Type	Remarks
DS_protection {	*dspr*		
DS_protection_value()			As specified in ISO/IEC 23092-3
}			

6.5.3.3.3 Semantics

DS_protection_value(): descriptor stream protection information, as specified in ISO/IEC 23092-3.

6.5.4 Offset

This box allows an indirect addressing of boxes in a different physical position in the file, while preserving their logical position as described in this document. It shall be placed in the mandatory position of the addressed box, as specified in subclause 6.1.2, so that the logical position of the addressed box would still be respecting such mandatory ordering.

In case of boxes not marked with suffix "[]" after their name in the Syntax column of any of the tables in subclauses 6.4 and 6.5, and which can be present in only one instance, if an associated offset box is present then multiple instances of the same original box may be physically present in the File, but only the box addressed by the offset box shall be considered as valid, while the other instances of the same box shall be ignored.

In case of boxes marked with suffix "[]" after their name in the Syntax column of any of the tables in subclauses 6.4 and 6.5, and which may be present in multiple instances, if the associated offset box is present it shall be present in as many instances as the addressed boxes.

In case one instance of the offset box is not referring to any box yet but just present as placeholder for a new box which may potentially be added, then the Offset field, as specified in subclause 6.5.4.1, shall take the (1<<64)–1 value, which shall be ignored.

6.5.4.1 Syntax

```
struct offset
{
      c(4)          Key;
      c(4)          SubKey;
      u(64)         Offset;
}
```

6.5.4.1.1 Semantics

Key is the key of the offset box, being equal to *offs*.

SubKey is the key of the box being addressed by the offset box. Its usage is restricted to the following boxes, as specified in Table 10: dghd, rfgn, rfmd, labl, lbll, dgmd, dgcd, dgpr, dtcn, dtmd, dtmt, dtcd, dtpr, atcn, atmd, atcd, atpr, atix, avix.

Offset is the byte offset of the first byte in the referenced box, with respect to the first byte of the file (0-based). If equal to (1<<64)–1 then the offset box is not addressing any box and shall be ignored. The value of Offset shall be larger than the offset of the last byte of any dgcn box in the file.

6.6 Data structures specific to transport format

6.6.1 General

This subclause specifies the data structures specific to the transport of genomic information, in addition to the data structures specified in subclause 6.4.

6.6.2 Data streams

A data stream is identified by a unique Stream_ID, equal to the SID field of packet header as specified in subclause 6.6.5.2, and it can transport any of the following data structures:

— file_header, as specified in subclause 6.4.5.1: this data stream shall be unique and composed by one or more packets with Stream ID (SID in packet header, as specified in subclause 6.6.5.2) equal to 1,

— data structures containing transport information (dataset mapping table list as specified in subclause 6.6.3, dataset mapping table as specified in subclause 6.6.4),

— dataset group header, as specified in subclause 6.4.2.2,

— reference, as specified in subclause 6.4.2.3,

— label list, as specified in subclause 6.4.2.5,

— dataset header, as specified in subclause 6.4.3.2,

— dataset parameter set, as specified in subclause 6.4.3.7,

— access unit, as specified in subclause 6.4.4,

— annotation table header, as specified in subclause 6.4.6.2,

— attribute group tile configuration, as specified in subclause 6.4.6.6,

— attribute group header, as specified in subclause 6.4.7.2,

— annotation access unit, as specified in subclause 6.4.8, and

— metadata and protection information, as specified in subclauses 6.4.2.4, 6.4.2.6, 6.4.2.7, 6.4.2.8, 6.4.3.3, 6.4.3.4, 6.4.3.5, 6.4.3.6, 6.4.6.3, 6.4.6.4 and 6.4.6.5.

6.6.3 Dataset mapping table list

6.6.3.1 General

This is a mandatory box (*dmtl* in Table 66) containing a list of all Stream_IDs of data streams transporting the dataset mapping tables, as specified in subclause 6.6.4, available in the datasets of a dataset group. Each of the listed data streams is identified by a unique dataset_mapping_table_SID.

The dataset mapping table list contains, along with the dataset mapping table described in subclause 6.6.4, the necessary and sufficient information to de-packetize and de-capsulate the transport format for a specific dataset in a dataset group.

Each dataset mapping table list is transported within one or more packets with Stream ID (SID in packet header, as specified in subclause 6.6.5.2) equal to 0.

6.6.3.2 Syntax

Table 66 — Dataset mapping table list syntax

Syntax	Key	Type	Remarks
dataset_mapping_table_list {	*dmtl*		
dataset_group_ID		u(8)	
for (i=0;i<num_datasets;i++) {			
dataset_mapping_table_SID[i]		u(16)	
}			
}			

6.6.3.3 Semantics

dataset_group_ID is the dataset group ID, as in the dataset group header, as specified in subclause 6.4.2.2.

num_datasets is inferred from the Length field in dataset_mapping_table_list *gen_info* header,as follows: (Length – 13) / 2.

dataset_mapping_table_SID[i] is the stream ID associated to the data stream containing the dataset_mapping_table. Values 0 and 1 cannot be used as reserved for the dataset mapping table list, as specified in subclause 6.6.3, and the file header, as specified in subclause 6.4.1. The same value of dataset_mapping_table_SID cannot be contained in dataset_mapping_tables with different dataset_group_ID.

6.6.4 Dataset mapping table

6.6.4.1 General

This is a mandatory box (*dmtb* in Table 67) listing all data streams transporting data related to the dataset identified by dataset_ID.

The dataset mapping table associates data types (access units, metadata boxes, protection boxes, etc.) and Stream IDs (SID) found in packet header, as specified in subclause 6.6.5.2.

Table 68 provides the association between data type values and data structures. The dataset mapping table contains, along with the dataset mapping table list specified in Table 66, the necessary and sufficient information to de-packetize and de-capsulate the transport format for a dataset in a dataset group.

The datasets mapping table can be periodically re-transmitted, either updated or identical.

6.6.4.2 Syntax

Table 67 — Dataset mapping table syntax

Syntax	Key	Type	Remarks
dataset_mapping_table {	*dmtb*		
dataset_ID		u(16)	
for (i=0;i<num_data_streams;i++) {			
data_type[i]		u(8)	
reserved		u(3)	

Table 67 *(continued)*

Syntax	Key	Type	Remarks
data_SID[i]		u(13)	
}			
}			

6.6.4.3 Semantics

dataset_ID is the dataset_ID as specified in subclause 6.4.3.2.

data_type[i] identifies the type of data carried by packets identified by subsequent data_SID[i] field, according to Table 68.

Note num_data_streams is inferred using the Length field in dataset_mapping_table gen_info, as follows: num_data_streams = (Length − 14) / 3.

Table 68 — data_type field semantics

data_type	Data structure	Subclause
0	Dataset group header (dghd)	6.4.2.2
1	Reference (rfgn)	6.4.2.3
2	Label list (labl)	6.4.2.5
3	Dataset header (dthd)	6.4.3.2
4	Dataset parameter set (pars)	6.4.3.7
5	Dataset group metadata (dgmd)	6.4.2.6
6	Reference metadata (rfmd)	6.4.2.4
7	Dataset metadata (dtmd)	6.4.3.3
8	Dataset group protection (dgpr)	6.4.2.8
9	Dataset protection (dtpr)	6.4.3.6
10 to 15	Access unit (aacn) with AU_type equal to data_type – 9	6.4.4
16	Dataset metrics metadata (dtmt)	6.4.3.4
17	Dataset_group clinical data linkage metadata (dgcd)	6.4.2.7
18	Dataset clinical data linkage metadata (dtcd)	6.4.3.5
19	Annotation table clinical data linkage metadata (atcd)	6.4.6.4
20	Annotation table header (athd)	6.4.6.2
21	Annotation table metadata (atmd)	6.4.6.3
22	Annotation table protection (atpr)	6.4.6.5
23	Attribute group tile configuration (agtc)	6.4.6.6
24	Attribute group header (aghd)	6.4.7.2
25	Annotation access unit (aauc)	6.4.8
26 to 255	Reserved for future use	

data_SID[i] is the i^th stream ID (SID) in the packet header, as specified in subclause 6.6.5.2, of packets transporting the corresponding data_type, as specified in Table 68. Its value shall be different than 0, as 0 is reserved for the dataset mapping table list as specified in subclause 6.6.3, and 1, as 1 is reserved for the file header, as specified in subclause 6.4.1. The same value of data_SID for data_type equal to either 3, 4, 7, 9, 10 – 15, 16, 18, or 19 - 25 cannot be contained in different dataset_mapping_tables with different dataset_ID or pointed, via the dataset_mapping_table_SID, by dataset_mapping_table_lists with different dataset_group_ID. The same value of data_SID for data_type equal to either 0, 1, 2, 5, 6, 8 or 17 cannot be contained in different dataset_mapping_tables pointed, via the dataset_mapping_table_SID, by dataset_mapping_table_lists with different dataset_group_ID.

6.6.5 Packet

6.6.5.1 General

A packet (see Table 69) is a transmission unit transporting in its payload segments of any of the data structures listed in Table 68.

The packet payload contains bytes from any of the data structures specified in Table 10, with the corresponding entry in the Scope column either being equal to Transport or empty.

Table 69 — Packet syntax

Syntax	Key	Type	Remarks
packet {			
packet_header		packet_header	As specified in subclause 6.6.5.2
for(i = 0; i < packet_size – sizeof(packet_header); i++) {			packet_size as specified in subclause 6.6.5.2
packet_payload[i]		u(8)	
}			
}			

Where:

packet_payload[i] is the i-th byte composing the packet payload.

6.6.5.2 Packet header

6.6.5.2.1 General

This is a mandatory box (see Table 70) describing the packet.

6.6.5.2.2 Syntax

Table 70 — Packet header syntax

Syntax	Key	Type	Remarks
packet_header {			
SID		u(13)	
reserved		u(3)	
marker_bit		u(1)	
sequence_number		u(8)	
packet_size		u(15)	
}			

6.6.5.2.3 Semantics

SID is the Stream ID of the data stream containing this packet. It unambiguously identifies data carried by this packet, according to the dataset_mapping_table, as specified in subclause 6.6.4. It shall be unique for each data stream.

marker_bit is equal to 1 in the last packet containing data of a certain box, it is equal to 0 in all other packets. It allows identifying the end of a box when split in multiple packets. In case of box carried in a single packet it shall always be set to 1. A packet shall never contain data of two different boxes.

sequence_number is a packet counter linearly increasing by 1. It is needed to identify packet losses as gaps in sequence_numbers for each individual data stream. It wraps around at 255.

packet_size is the number of bytes composing the packet, including header and payload. If packet_size is equal to 5, then this is the last packet containing data of the dataset associated to SID; it shall be different than 5 in all other cases.

6.7 Reference procedures to convert transport format to file format

The conversion procedures in this subclause shall be considered as reference procedures in order to guarantee that:

— the resulting file is compliant to the file format specification, and

— the resulting file retains exactly the same information as the original stream compliant with the transport format.

Any other process producing the same output from the same conformant bitstream can be equally considered compliant to the ISO/IEC 23092 series. At the same time, complying implementations are not expected to follow the exact algorithm used by these reference procedures.

NOTE These conversion procedures are needed to save in a storage device the data received via a transport session, so that the resulting file is compliant to the file format, as specified in subclause 6.3.

6.7.1 Procedure for genomic sequencing data

This subclause describes a reference procedure to update the parameters seq_count and seq_blocks[seq] in the dataset header, as specified in subclause 6.4.3.2, and to compile the master index table, when present, as specified in subclause 6.5.2.1, from the syntax elements present in the access unit header, as specified in subclause 6.4.4.3. The procedure is the following:

1. The dataset header field block_header_flag is determined by the contiguity mode of the output file, as specified in subclause 6.4.3.2.5.

2. seq_count is initialized to 0 and is incremented by 1 every time sequence_ID in the access_unit_header, as specified in subclause 6.4.4.3, is different than any previously received sequence_ID.

3. seq_blocks[seq_count] is initialized to 0 and incremented by 1 every time the (sequence_ID, access_unit_ID) vector in the access_unit_header, as specified in subclause 6.4.4.3, is different than any previous (sequence_ID, access_unit_ID) vector.

4. The coordinates of the master index table, as specified in subclause 6.5.2.1, are calculated as follows during the process:

    a. seq = seq_count

    b. ci is incremented by 1 every time a new value of AU_type in the access unit header, as specified in subclause 6.4.4.3, is received; the array clid, as specified in subclause 6.4.3.2, is filled with the AU_type value accordingly;

    c. au_id = seq_blocks[seq];

    d. If block_header_flag is equal to 0, di is incremented by 1 every time a new value of the descriptor_ID field in the block header, as specified in subclause 6.4.5.2, is received; the array descriptor_id, as specified in subclause 6.4.3.2, is filled with the descriptor_id value accordingly.

5. If class_ID is different than CLASS_U:

    a. The master index table entry named AU_start_position[seq][ci][au_id] is calculated as follows:

       AU_start_position[seq][ci][au_id] = AU_start_position in the access unit header, as specified in subclause 6.4.4.3.

b. The master index table entry named AU_end_position[seq][ci][au_id] is calculated as follows:

   AU_end_position[seq][ci][au_id] = AU_end_position in the access unit header, as specified in subclause 6.4.4.3.

c. If multiple_alignment_flag is equal to 1:

   i. The master index table entry named extended_AU_start_position[seq][ci][au_id] is calculated as follows:

      extended_AU_start_position[seq][ci][au_id] = extended_AU_start_position in the access unit header, as specified in subclause 6.4.4.3.

   ii. The master index table entry named extended_AU_end_position[seq][ci][au_id] is calculated as follows:

      extended_AU_end_position[seq][ci][au_id] = extended_AU_end_position in the access unit header, as specified in subclause 6.4.4.3.

Else if class_ID == CLASS_U and the dataset header field U_signature_flag is equal to 1

a. If the dataset header field U_signature_constant_length is equal to 0, the master index table entries named U_cluster_signature_length[uau_id][i] are calculated as follows:

   U_cluster_signature_length[uau_id][i] = U_cluster_signature[i] in the access unit header, as specified in subclause 6.4.4.3.

   where uau_id = au_id.

b. The master index table entries named U_cluster_signature[uau_id][i] are calculated as follows:

   U_cluster_signature [uau_id][i] = U_cluster_signature[i] in the access unit header, as specified in subclause 6.4.4.3.

   where uau_id = au_id.

6. If block_header_flag is equal to 0:

   a. Every time a block with a certain descriptor_ID contained in the access unit carrying data of class class_ID is received, where the (class_ID, descriptor_ID) vector has not been received yet, a new descriptor stream container box *dscn*, named DSCN[ci][di], is created and a variable named blockPtr[ci][di] is defined and initialized to 0.

   b. A variable named headerSize is created and set as the number of bytes following the first byte of the Value field of the dataset (dtcn) gen_info structure (0-based) and preceding the first byte of DSCN[ci][di].

   c. blockPtr[ci][di] is incremented by header_size + Length of any child gen_info box of DSCN[ci][di].

   d. The payload of all blocks identified by class_ID and descriptor_ID in the block header is copied into the Value field of DSCN[ci][di].

   e. For each of the above blocks, a new variable named blockPtr[ci][di][au_id] is defined and assigned the current value of blockPtr[ci][di].

   f. For each block, blockPtr[ci][di] is incremented by block_payload_size.

   g. Once the session is terminated and before writing the resulting data into the output file, a variable named dsOffset is defined, initialized to 0.

   h. After writing each DSCN[ci][di] in the output file, dsOffset is incremented by sizeof(gen_info header) + Length(DSCN[ci][di]).

i. For each DSCN[ci][di] written in the output file, the master index table entry named block_byte_offset[seq][ci][au_id][di] is updated as follows:

block_byte_offset[seq][ci][au_id][di] = dsOffset + blockPtr[ci][di][au_id]

Else if block_header_flag is equal to 1:

a. A buffer named datasetPayload is created.

b. For each access unit:

i. The access unit header is written into datasetPayload.

ii. Each block composing the access unit is written into the datasetPayload.

c. For each access unit, AU_byte_offset[seq][ci][au_id] is equal to the offset of the first byte of the access unit with regards to the first byte of the datasetPayload.

d. At the end of the session, datasetPayload is written into the output file as dataset payload, as specified in subclause 5.4.

7. At the end of the process, typically when the end user stops the execution of the transport process, the master index table shall be re-ordered, per each combination of either seq and class_ID indexes or uau_id index, by increasing value of, respectively, AU_start_position[seq][ci][au_id] or U_cluster_signature[uau_id][0].

6.7.2 Procedure for genomic annotation data

This subclause describes a reference procedure to update selected fields in dataset headers (as specified in subclause 6.4.3.2), attribute group headers (as specified in subclause 6.4.7.2), annotation access unit headers (as specified in 6.4.8.3) and attribute data tile structures (as specified in 6.4.6.7), and generate data structures of annotation table index (as specified in 6.5.2.4) and attribute data byte offset (as specified in 6.5.2.3), as needed, using information in the transported containers individually identified by a trail of container IDs that also indicate their organization in the file. The procedure is the following:

1. Build a tree structure of container IDs as shown in Figure 8 by adding a node to the tree whenever a new container box is received based on the list of IDs that uniquely identify the box in transport format. In Figure 8, child nodes belonging to the same parent are indexed by integers 1 to n, and the number of child nodes n can be different in each container.

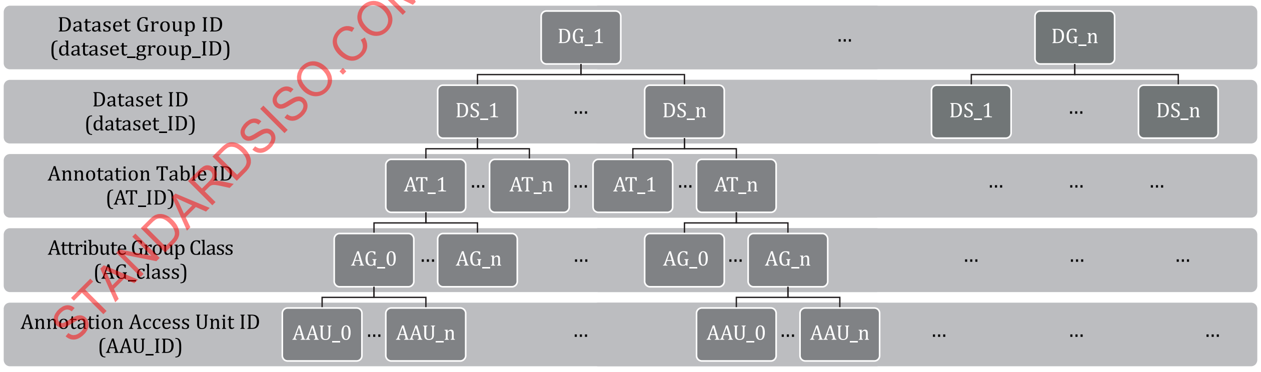

Dataset Group ID (dataset_group_ID)
Dataset ID (dataset_ID)
Annotation Table ID (AT_ID)
Attribute Group Class (AG_class)
Annotation Access Unit ID (AAU_ID)
DG_1
DG_n
DS_1
DS_n
AT_1
AT_n
AG_0
AG_n
AAU_0
AAU_n
...

Figure 8 — Tree structure of container IDs

2. For each dataset header (as specified in subclause 6.4.3.2) identified by dataset_group_ID and dataset_ID, if n_ann_tables == 0, update the fields values as follows:

a. n_ann_tables is set to the number of annotation table IDs associated with the dataset node.

b. AT_ID[i] is set to the ID of the ith annotation table associated with the dataset node.

c. AT_index_exists[i] is set to 1 if the ith annotation table associated with the dataset node is accompanied by an annotation table index (atix). Otherwise, it is set to 0.

3. For each annotation table identified by dataset_group_ID, dataset_ID and AT_ID, generate an annotation table index (atix, as specified in subclause 6.5.2.4) if one of the following is true:

   a. AT_genomic_pos_sort_mode > 0, in which case one or multiple genomic range indexes (grix, as specified in subclause 6.5.2.4.4) shall be provided, or

   b. One or multiple attribute value indexes (avix, as specified in 6.5.2.4.5) are available for the annotation table.

4. For each annotation access unit header (as specified in 6.4.8.3) identified by dataset_group_ID, dataset_ID, AT_ID, AG_class and an annotation access ID, which can be a descriptor/attribute ID (when attribute_contiguity == 1) or tile index(es) (when attribute_contiguity == 0), if n_AAU_blocks equals 0, update its value to the number of blocks contained in the annotation access unit.

5. For each attribute group header (as specified in subclause 6.4.7.2) identified by dataset_group_ID, dataset_ID, AT_ID and AG_class, if dimension_size[0] equals 0, update the value(s) in dimension_size[] with the process specified in Table 71.

Table 71 — Process to compute dimension_size

Syntax	Remarks
`if (variable_size_tiles == 0) {`	variable_size_tiles (in subclause 6.4.6.7) of default_tile_structure (in subclause 6.4.6.6) associated with the attribute group identified by AG_class
`    if (attribute_contiguity == 0) {`	attribute_contiguity as specified in subclause 6.4.7.2 associated with the attribute group identified by AG_class
`        dimension_size[0] = max_tile_idx_1() * tile_size[0] + nrows_btm_tile()`	max_tile_idx_1() returns the maximum value of tile_index_1 (0-based), as specified in subclause 6.4.8.3, among all annotation access units associated with the attribute group identified by AG_class tile_size[] (in subclause 6.4.6.7) of default_tile_structure (in subclause 6.4.6.6) associated with the attribute group identified by AG_class nrows_btm_tile() returns the number of rows in a decoded bottom tile with tile_index_1 == max_tile_idx_1()
`        if (two_dimensional) {`	two_dimensional as specified in subclause 6.4.7.2 associated with the attribute group identified by AG_class
`            if (tile_index_2_exists) {`	tile_index_2_exists as specified in subclause 6.4.8.3 of any of the annotation access units associated with the attribute group identified by AG_class

Table 71 *(continued)*

Syntax	Remarks
`dimension_size[1] = max_tile_idx 2() * tile_size[1] + ncols_right_tile()`	max_tile_idx_2() returns the maximum value of tile_index_2 (0-based), as specified in subclause 6.4.8.3, among all annotation access units associated with the attribute group identified by AG_class ncols_right_tile() returns the number of columns in a decoded rightmost tile with tile_index_2 == max_tile_idx_2()
`} else {`	
`dimension_size[1] = ncols_right_tile()`	
`}`	
`}`	
`} else {`	
`if (two_dimensional == 0) {`	
`dimension_size[0] = (n_AAU_blocks − 1) * tile_size[0] + len_last_block()`	n_AAU_blocks as specified in subclause 6.4.8.3 tile_size (in subclause 6.4.6.7) of default_tile_structure (in subclause 6.4.6.6) associated with the attribute group identified by AG_class len_last_block() returns the number of elements in the decoded vector of the last block of a descriptor/attribute associated with default_tile_structure
`} else {`	
`if (column_major_tile_order) {`	column_major_tile_order as specified in subclause 6.4.7.2
`n_tiles_1 = n_tiles_per_col`	n_tiles_per_col as specified in subclause 6.4.8.3
`n_tiles_2 = Ceil(n_AAU_blocks / n_tiles_1)`	n_AAU_blocks as specified in subclause 6.4.8.3
`} else {`	
`n_tiles_2 = n_tiles_per row`	n_tiles_per_row as specified in subclause 6.4.8.3
`n_tiles_1 = Ceil(n_AAU_blocks / n_tiles_2)`	
`}`	
`dimension_size[0] = (n_tiles_1 − 1) * tile_size[0] + nrows_last_block()`	nrows_last_block() returns the number of rows in the decoded tile of the last block of a descriptor/attribute associated with default_tile_structure
`dimension_size[1] = (n_tiles_2 − 1) * tile_size[1] + ncols_last_block()`	ncols_last_block() returns the number of columns in the decoded tile of the last block of a descriptor/attribute associated with default_tile_structure
`}`	
`}`	
`} else {`	

Table 71 *(continued)*

Syntax	Remarks
`    dimension_size[0] = Max(end_index[][0]) + 1`	end_index[][] (0-based, in subclause 6.4.6.7) of default_tile_structure (in subclause 6.4.6.6) associated with the attribute group identified by AG_class
`    if (two_dimensional) {`	
`        dimension_size[1] = Max(end_index[][1]) + 1`	
`    }`	
`}`	

6. For each attribute group tile configuration (as specified in subclause 6.4.6.6) identified by dataset_group_ID, dataset_ID, AT_ID and AG_class, if n_tiles == 0 in any of the associated attribute data tile structures (as specified in subclause 6.4.6.7), update its value with the process specified in Table 72. If variable_size_tiles == 1, n_tiles shall always be set when an attribute data tile structure is generated.

Table 72 — Process to compute n_tiles

Syntax	Remarks
`if (variable_size_tiles == 0) {`	variable_size_tiles as specified in subclause 6.4.6.7
`    if (attribute_contiguity == 0) {`	attribute_contiguity as specified in subclause 6.4.7.2 associated with the attribute group identified by AG_class
`        n_tiles = max_tile_idx_1() + 1`	max_tile_idx_1() returns the maximum value of tile_index_1 (0-based), as specified in subclause 6.4.8.3, among all annotation access units associated with the attribute group identified by AG_class
`        if (tile_index_2_exists) {`	tile_index_2_exists as specified in subclause 6.4.8.3 of any of the associated with the attribute group identified by AG_class
`            n_tiles = n_tiles * (max_tile_idx_2() + 1)`	max_tile_idx_2() returns the maximum value of tile_index_2 (0-based), as specified in subclause 6.4.8.3, among all annotation access units associated with the attribute group identified by AG_class
`        }`	
`    } else {`	
`        n_tiles = n_AAU_blocks`	n_AAU_blocks as specified in subclause 6.4.8.3
`    }`	
`}`	

7. Generate an attribute data byte offset (adbo, as specified in subclause 6.5.2.3) data structure for each attribute group tile configuration (as specified in subclause 6.4.6.6).

8. Remove from the beginning of each received container any trail of IDs of the upper-level containers required only for the transport format, before ouputting all data structures into the hierarchy of containers required for the file format.

7 String indexing technologies

7.1 Master string index

7.1.1 General

When used for the string compression of genomic annotation data with dataset_type == 3 as specified in subclause 6.4.3.2, the master string index (MSI, see Table 73) self-indexes a subset of the string fields within records, for all the records within annotation access units and for all the annotation access units within a dataset. The list of string fields encoded in a MSI is specified by fields n_index_descriptors, index_descriptor_ID[], n_index_attributes and index_attribute_ID[] specified in subclause 6.5.2.4.5. All the associated descriptors and attributes shall share the same tile structure in the attribute group tile configuration as specified in subclause 6.4.6.6.

When the MSI is used, the corresponding subset of string fields shall be encoded only in the MSI, i.e. the said string fields shall not be encoded also in the annotation access units. The index at the same time stores the text and makes it searchable. Therefore, when one or more MSIs are present in a dataset, the annotation access unit decoding processes shall decode the corresponding string fields from each MSI. The query and decoding processes for the string index are specified in subclause 7.2.

When used for the string compression of genomic sequencing data with dataset_type < 3 as specified in subclause 6.4.3.2, a compressed string index shall be generated for the data of an indexed descriptor in each access unit and stored in the corresponding block payload.

7.1.2 Syntax

Table 73 — Syntax of master string index

Syntax	Key	Type	Remarks
master_string_index(numStrings, useTiles) {	*msix*		
master_string_index_header()	mshd		As specified in subclause 7.1.3
for(j = 0; j < num_indices; j++) {			num_indices as specified in subclause 7.1.3.1
string_index[j](numStrings, useTiles)	ssix		string_indexencodes a list of strings as specified in subclause 7.1.4
}			
}			

7.1.3 Master String Index Header

7.1.3.1 Syntax and semantics

Table 74 — Syntax of master_string_index_header

Syntax	Key	Type	Remarks
master_string_index_header {	*mshd*		
string_index_id		u(16)	
num_indices		u(8)	
for(i=0; i < num_indices; i++){			
is_attribute[i]		u(1)	
if(is_attribute[i]){			
attribute_ID[i]		u(16)	

Table 74 *(continued)*

Syntax	Key	Type	Remarks
`      } else {`			
`         descriptor_ID[i]`		u(7)	
`      }`			
`   }`			
`   while(!byte_aligned())`			As specified in subclause 4.2.3
`      nesting_zero_bit`		f(1)	One bit set to 0
}			

string_index_id is a unique identifier of the master string index

num_indices is the number of sub-indexes in which this MSI is split. It shall be greater than 0.

is_attribute[i] is a flag set to 1 if the corresponding descriptor or attribute is an attribute, 0 otherwise.

attribute_ID[i] is the identifier of the string attribute payload which is stored in the master string index as a self index.

descriptor_ID[i] is the identifier of the string descriptor payload which is stored in the master string index as a self index.

7.1.4 String index

7.1.4.1 General

A string index block (see Table 75) is a portion of a master string index that encodes one or more strings for each record, for a variable number of annotation data tiles each containing a variable number of records.

The list of strings encoded within a string index is referred to in the following as "compressed string index".

The list of strings obtained by decoding a compressed string index from a String Index is referred to in the following as "uncompressed string index" or "string index payload".

7.1.4.2 Syntax

Table 75 — Syntax of string_index

Syntax	Key	Type	Remarks
string_index(numStrings, useTiles) {	*ssix*		
`  if (useTiles == 0) {`			
`    num_AUs`		u(24)	
`    for (i = 0; i < num_AAUs; i++) {`			
`      au_id[i]`		u(32)	
`      if(i > 0) {`			
`        au_offset[i]`		u(32)	
`      }`			
`    }`			
`  } else if (useTiles == 1) {`			
`     n_tiles`		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
`    for (i = 0; i < n_tiles; i++) {`			

Table 75 *(continued)*

Syntax	Key	Type	Remarks
tile_index_1[i]		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
if (two_dimensional && !variable_size_tiles) {			two_dimensional as specified in subclause 6.4.7.2 variable_size_tiles as specified in subclause 6.4.6.6.4 of the tile structure associated with the indexed descriptors and attributes
tile_index_2[i]		u(v)	v dependent on AT_coord_size as specified in subclause 6.4.6.2.3
}			
if(i > 0) {			
tile_offset[i]		u(32)	
}			
}			
}			
coding_mode		u(8)	String Index coding mode (i.e. compression profiles, to enable controlling the trade-off between compression and querying speed)
compressed_string_index	csix	gen_info	A compressed binary list of strings, as specified in subclause 7.1.5 and in Figure 4
}			

7.1.4.3 Semantics

num_AUs is the number of (annotation) access units encoded in this string index.

au_id[i] is the (annotation) access unit ID of the ith annotation access unit encoded in this string index.

au_offset[i] is the byte position in the uncompressed string index (i.e. uncompressed payload) encoded in compressed_string_index of the first string of the first record of the ith annotation access unit encoded in this string index.

n_tiles is the number of data tiles encoded in this string index.

tile_index_1[i] is the index of the ith tile encoded in this string index. If the associated attribute group is two dimensional and with variable_size_tiles == 0, it corresponds to the row index of the tile.

tile_index_2[i] is the column index of the ith tile encoded in this string index, only specified if the associated attribute group is two dimensional and with variable_size_tiles == 0.

tile_offset[i] is a non-negative integer number representing the byte position, within the uncompressed string index (i.e. string index payload, as specified in subclause 7.1.4) encoded in compressed_string_index, of the first string of the first record of the ith tile encoded in this string index. It is specified for $0 < i < \text{n\_tiles}$. For i == 0, the offset is always 0. tile_offset[i + 1] is never smaller than tile_offset[i].

coding_mode is the string index coding mode, which identifies the compression profile required to decode this string index, with specific settings for controlling the trade-off between compression and querying speed. At the moment only value 0 is allowed – it represents the current implementation of the compressed FM-index. Other values are reserved for future extensions.

compressed_string_index is a compressed list of strings, as specified in subclause 7.1.5 and in Figure 9.

7.1.5 Compressed string index

The compressed string index is the binary representation of an implementation of a compressed Ferragina-Manzini full-text index. It supports a number of operations (such as text decoding and backward searches) in the compressed space. It is used as the underlying engine for all the data structures requiring string indexing.

The index is implemented on the top of a number of auxiliary data structures (mainly a wavelet tree, a compressed bitmap and compressed counters) that are detailed in the following subclauses.

7.1.5.1 Helper functions

This subclause defines the CompactCountersSize() helper function (see Table 76) used throughout subclause 7.1.5. This function computes the size in bytes of the 64-bit aligned compressed string compact counters data structure specified in subclause 7.1.5.5. The inputs are:

— n, of type u(32), the number of counter values in the data structure

— bits_num, of type u(32), the size in number of bits of each counter value

Table 76 — Function CompactCountersSize

Decoding Process	Remarks
`CompactCountersSize(n, bits_num) {`	
`  return 2 * sizeof(sa_t) + 8 * Ceil((bits_num*n)/64)`	
`}`	

7.1.5.2 Compressed string index

In this subclause we describe the main data structure for the compressed string index.

Table 77 — Syntax of compressed string index

Syntax	Key	Type	Remarks
compressed_string_index {	*csix*		
n		u(32)	Length of the uncompressed payload encoded in this string index
alpha		u(32)	Alphabet size. It must be greater than 0
sampling_rate_sa		u(32)	Interval between two explicitly stored SA values
sampling_rate_lf		u(32)	Interval between two explicitly stored LF values
for(i = 0; i < 256; i++) {			
encode[i]		u(16)	Encoded symbol corresponding to numerical ASCII value i
}			
for(i = 0; i < 256; i++) {			

Table 77 *(continued)*

Syntax	Key	Type	Remarks
decode[i]		c(1)	Alphabet symbol corresponding to encoded symbol i, i.e. encode[i] returns the numerical ASCII value corresponding to i
}			
sa_bytes = CompactCountersSize(Ceil(n / sampling_rate_sa),Ceil(Log2(n)))			Size in bytes of the data structure sa
lf_bytes = CompactCountersSize(Ceil(n / sampling_rate_lf),Ceil(Log2(n)))			Size in bytes of the data structure lf
sa			compressed_string_compact_counters of size sa_bytes, as specified in subclause 7.1.5.5
lf			compressed_string_compact_counters of size sa_bytes, as specified in subclause 7.1.5.5
wt()			compressed wavelet tree, as specified in subclause 7.1.5.3
}			

n is the length in bytes of the uncompressed payload encoded in this string index.

alpha is the number of symbols (i.e. byte values) used by this string index. It must be greater than 0 because a string index encodes at the very least the uncompressed payload terminator '\0'.

sampling_rate_sa is the interval between two suffix array (SA) values explicitly stored in the string index.

sampling_rate_lf is the interval between two last-to-first (LF) values explicitly stored in the string index.

sa is the compressed string compact counters of SA values, sampled at intervals of sampling_rate_sa starting from 0, explicitly stored in this string index.

lf is the compressed string compact counters of LF values, sampled at intervals of sampling_rate_lf starting from 0, explicitly stored in this string index.

wt is the compressed wavelet tree storing the Burrows-Wheeler Transform (BWT) of the original text encoded in this string index.

encode[i] contains the symbol by which byte value i is encoded in this string index.

decode[i] contains the byte value of the i^{th} symbol encoded in this string index. Symbol '\0' (byte value 0) is reserved internally to mark the end of the whole uncompressed payload. Symbol '\1' (byte value 1) is reserved as string terminator, as specified in subclause 7.2.1 All other symbols are valid symbols for the encoded payload.

7.1.5.3 Compressed string wavelet tree

This subclause described the data structure (see Table 77) specifying the compressed wavelet tree storing the Burrows-Wheeler Transform (BWT) of the original text encoded in the string index.

Table 78 — Syntax of compressed string wavelet tree

Syntax	Type	Remarks
compressed_string_wavelet_tree {		
n	u(32)	Length of the uncompressed payload encoded in the string index
alpha	u(32)	Alphabet size. It must be greater than 0
for(i = 0; i < 257; i++) {		
c[i]	u(32)	Cumulative number of the occurrences of the encoded symbol corresponding to numerical ASCII value i
}		
red_isa_0	u(32)	ISA[0] i.e. the value in 0 of the Inverse Suffix Array for the original string, before the lexicographic terminator is removed
char_red_isa_0	u(32)	The last character in the original string, which replaces the lexicographic terminator in the reduced BWT
num_nodes	u(32)	Number of wavelet tree nodes
for(i = 0; i < num_nodes; i++){		
nodes[i]()	compressed_string_wavelet_node	As specified in subclause 7.1.5.4
bitmap_bytes[i]()	u(32)	Lengths in bytes of the compressed bitmaps associated with each wavelet tree node
}		
i = 0		
while (i < num_nodes) {		
nodes[i] = compressed_string_wavelet_node (i, bitmap_bytes)	compressed_string_wavelet_node	As specified in sub-clause 7.1.5.4
i++		
}		
root = nodes[0]		The root node of the wavelet tree
for (i = 0; i < num_nodes; i++) {		
bitmap[i] = compressed_string_bitmap(bitmap_bytes[i])	compressed_string_bitmap	As specified in subclause 7.1.5.6
}		

7.1.5.4 Compressed string wavelet node

In this subclause we describe the data structure specifying one node of the compressed wavelet tree storing the Burrows-Wheeler Transform (BWT) of the original text encoded in the string index.

Table 79 — Syntax of compressed string wavelet node

Syntax	Type	Remarks
compressed_string_wavelet_node(curr_node, bitmap_bytes) {		
min_char	u(16)	Minimum encoded character for this node
max_char	u(16)	Maximum encoded character for this node
off_bitmap_bytes[]		bitmap_bytes array offset by curr_node positions
curr_node++		
is_terminal = (max_char == min_char) ? 1 : 0		Flag to indicate if the node has any children
if (!is_terminal) {		
left = compressed_string_wavelet_node (curr_node, off_bitmap_bytes)	compressed_string_wavelet_node	Left child node
right = compressed_string_wavelet_node (curr_node, off_bitmap_bytes)	compressed_string_wavelet_node	Right child node
}		
}		

min_char is the minimum encoded character in the interval defining the node.

max_char is the maximum encoded character in the interval defining the node.

bitmap_bytes is the length in bytes of the payload encoded in this node.

left is the leaf node at the left side of this node.

right is the leaf node at the right side of this node.

7.1.5.5 Compressed string compact counters

This data structure (see Table 80) specifies compact vectors of integer counters whereby each integer is represented as a bitfield of fixed width.

Table 80 — Syntax of compressed_string_compact_counters

Syntax	Type	Remarks
compressed_string_compact_counters {		
n	u(32)	
bits_number	u(32)	
for (i = 0; i < n; i++) {		
bitmap[i]	u(bits_number)	The counters
}		

Table 80 *(continued)*

Syntax	Type	Remarks
padding_bits = 64 * Ceil((bits_number * n) / 64) − bits_number * n	u(32)	
padding	u(padding_bits)	Padding to the next 64-bit boundary with bits of 0
}		

n is the number of counters stored in the vector.

bits_number is the dimension in bits of each counter stored in the vector.

bitmap[i] is a bitfield containing the i^th^ counter value. It is padded to the next 64-bit boundary.

7.1.5.6 Compressed string bitmap

This data structure (see Table 81) implements compressed bitmaps. The input variable size_in_bytes, of type u(32), is the size in bytes of the data structure.

Table 81 — Syntax of compressed string bitmap

Syntax	Type	Remarks
compressed_string_bitmap(size_in_bytes) {		
n	u(32)	
superblock_size	u(24)	
block_size	u(8)	
superblock_bytes	u(32)	
classes_bits	u(32)	
superblock_ranks	compressed_string_sampled_bounded_counters	As specified in subclause 7.1.5.7
superblock_offsets	compressed_string_sampled_bounded_counters	As specified in subclause 7.1.5.7
block_bits = Ceil(Log2((block_size))		
num_blocks = classes_bits / block_bits		
for (i = 0; i < num_blocks; i++) {		
classes[i]	u(block_bits)	
}		
bitmap_size = 8* size_in_bytes − 128 − 8 * superblock_bytes - classes_bits		Size in bits of the bitmap
bitmap	u(bitmap_size)	Compressed bitmap. The sequence of classes and bitmap is padded to the next 64-bit boundary
padding_bits = 64 * Ceil((classes_bits + bitmap_bits) / 64) − (classes_bits + bitmap_bits)		
padding	u(padding_bits)	Padding to the next 64-bit boundary with bits of 0
}		

n is the length of the uncompressed bitmap.

superblock_size is the number of bitmap blocks that form a superblock.

block_size is the number of bits in the uncompressed bitmap that form a bitmap block.

superblock_bytes is the combined size in bytes of superblock_ranks and superblock_offsets.

classes_bits is the combined size in bits of all elements in classes[].

superblock_ranks is the ranks of individual superblocks. The rank of a superblock is the number of ones in the uncompressed bitmap up to and including the superblock.

superblock_offsets is the byte offsets, each counting from the beginning of the compressed bitmap to individual superblocks in the compressed bitmap.

classes[i] is the number of ones in the ith bitmap block.

bitmap is the compressed bitmap consisting of a sequence of index values, one for each bitmap block and in the same order, that together with block_size and the corresponding block classes, can point to the pre-defined bit patterns, which give the uncompressed bitmap blocks.

7.1.5.7 Compressed string sampled bounded counters

This data structure (see Table 82) implements bounded counters (i.e., such that counter(i)≤i) sampled at regular intervals.

Table 82 — Syntax of compressed string sampled bounded counters

Syntax	Type	Remarks
compressed_string_sampled_bounded_counters {		
n	u(32)	
sampling_rate	u(32)	
major_rate = sampling_rate * Ceil(16. * Log(n)/ Log(sampling_rate))		Superblock sampling rate
size_minor = CompactCountersSize(Ceil(n / sampling_rate),Ceil(Log2(major_rate)))		Size of minor counters
size_major = CompactCountersSize(Ceil(n / major_rate),Ceil(Log2(n)))		Size of major counters
major	compressed_string_compact_counters	
minor	compressed_string_compact_counters	
}		

n is the length of the unsampled counters.

sampling_rate is the interval at which the counters are sampled.

major consists of counter values that correspond to major intervals, each containing multiple superblocks. The ith major counter value major_count[i] gives the cumulative number of 1 bits up to but excluding the ith major interval, counting from the beginning of the uncompressed bitmap, with major_count[0] = 0.

minor consists of counter values that correspond to the superblocks specified in subclause 7.1.5.6 7.1.5.6. The ith minor counter value minor_count[i] gives the cumulative number of 1 bits up to and including the ith superblock, counting from 0 at the beginning of each major interval.

7.2 Decoding and querying processes

The following subclauses specify how uncompressed string index payloads shall be structured, queried, and decoded. In turn, such processes are defined in terms of a number of additional processes, each one operating on one or more of the data structures that make up the string index.

7.2.1 String index payload

7.2.1.1 General

The string index payload is the uncompressed string index (see Table 83) encoded within compressed_string_index. If useTiles is 0, it contains a list of strings and the associated optional record indexes, ordered per annotation access unit (following the same order of the annotation access units in Table 75) and, for each annotation access unit, per record (following the same order of the records within the annotation access unit). The total number of strings in the uncompressed index is totNumRecords*numStrings, where totNumRecords is the total number of records of all annotation access units identified by aau_id[] . If useTiles is 1, it contains a list of strings and the associated optional record indexes, ordered per annotation data tile (following the same order of data tile index(es) in Table 74) and, for each annotation data tile, per record. The total number of strings in the uncompressed index is totNumRecords * numStrings, where totNumRecords is the total number of records of all annotation data tiles identified by tile_index_1[] and, if applicable, also tile_index_2[]. The variables numStrings, tile_index_1[] and tile_index_2[] are specified in subclause 7.1.4. The uncompressed payload terminator '\0' shall be appended to the string index payload before generating the compressed string index.

7.2.1.2 Syntax and semantics

Table 83 — Syntax of uncompressed string index encoded in the compressed string index element of a string index element.

Syntax	Type	Remarks
uncompressed_string_index(si) {		si is a string index as specified in subclause 7.1.4. uncompressed_index(si) is the result of decoding si.compressed_index
for(i = 0; i < totNumRecords; i++) {		totNumRecords is the total number of records of all annotation data tiles identified by tile_index_1[] and, if applicable, also tile_index_2[].
record_index[i][]	u(8)[]	
for(j = 0; j < si.numStrings; j++) {		
string[i][j][]	u(8)[]	
string_terminator	u(8)	
}		
}		
}		

Figure 4 summarizes an example of the uncompressed index specified in this subclause with numStrings equal to 3.

record_index[i] is an optional element (not present when its length is zero), whose presence is signaled by setting the most significant bit on all the bytes of record_index[i]. Setting the most significant bit also prevents from obtaining false-positive results when searching for substrings, since all bytes in the string[i][j] field have the most significant bit unset as specified in this subclause for string[i][j] element. The number of bytes is comprised between 0 (i.e. record_index[i] is not present) and 6. All bytes in record_index[i][] shall have the two most significant bits set to 0b10. If n > 1, record_index[i][0] shall not be equal to 0x80.

When record_index[i] is present and it is *N* bytes long, it represents a non-negative integer value as specified in the following expression:

$$recordIndexValue[i] = \sum_{n=0}^{N-1} (record\_index[i][n] \,\&\, 0x7F) \ll ((N-1-n)*7)$$

where recordIndexValue[i] corresponds to the 0-based index, within the corresponding Annotation Access Unit, of the Record corresponding to string[i][] strings.

string[i][j] is the jth encoded descriptor/attribute of the ith record. All bytes in string[i][j][] shall encode a valid UTF-8 string, as specified in ISO/IEC 10646, that does not contain any occurrence of bytes 0x00 or 0x01. If useTiles is 0, the strings shall be ordered per annotation access Unit (following the same order of the annotation access units in Table 75) and, for each annotation access unit, per record (following the same order of the Records within the annotation access unit). If useTiles is 1, the strings shall be ordered per annotation data tile, following the same order of the data tile index(es) in Table 75 and, for each data tile, per record). If the data tile is two dimensional, a record shall correspond to a cell in the tile arranged in row-major order. For each record, the strings of the indexed descriptors and attributes shall be concatenated in the same order as their IDs listed in the attribute value index as specified in subclause 6.5.2.4.5.

string_terminator is a single byte equal to 0x01 (i.e. '\1').

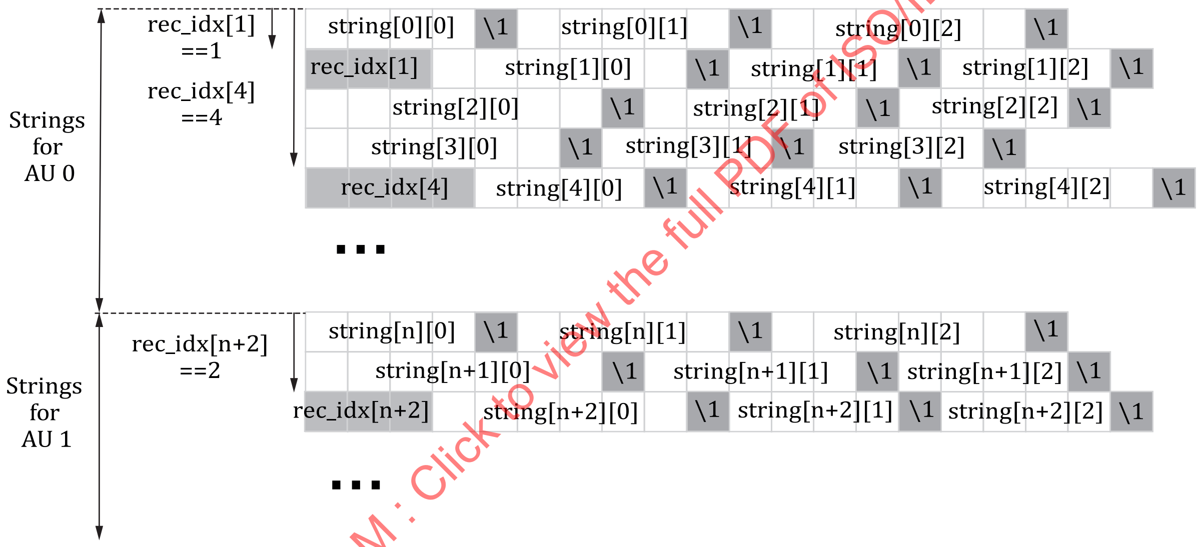

Strings for AU 0
rec_idx[1] ==1
rec_idx[4] ==4
string[0][0] \1 string[0][1] \1 string[0][2] \1
rec_idx[1] string[1][0] \1 string[1][1] \1 string[1][2] \1
string[2][0] \1 string[2][1] \1 string[2][2] \1
string[3][0] \1 string[3][1] \1 string[3][2] \1
rec_idx[4] string[4][0] \1 string[4][1] \1 string[4][2] \1
...
Strings for AU 1
rec_idx[n+2] ==2
string[n][0] \1 string[n][1] \1 string[n][2] \1
string[n+1][0] \1 string[n+1][1] \1 string[n+1][2] \1
rec_idx[n+2] string[n+2][0] \1 string[n+2][1] \1 string[n+2][2] \1
...

Figure 9 — Example of uncompressed index of string index, in the case of numStrings equal to 3

7.2.2 Helper functions

This subclause defines helper functions which are used throughout subclause 7.2 to describe the rest of the decoding processes.

7.2.2.1 CompactCountersGet

The helper function CompactCountersGet shall return the counter at a specified index from the data structure of type compressed_string_compact_counters.

The inputs of this helper function are:

— cc is a data structure of type compressed_string_compact_counters as specified subclause 7.1.5.5.

— i, of type u(32), the index to the counter to be returned .

The output of this function is the counter at position i of the compact counters cc of type compressed_string_compact_counters.

7.2.2.2 SampledBoundedCountersGet

The helper function SampledBoundedCountersGet shall return the counter at a specified index from the data structure of type compressed_string_sampled_bounded_counters.

The inputs of this helper function are:

— sbc is a data structure of type compressed_string_bounded_counters as specified in subclause 0

— i is an integer denoting the index into the counters stored by the data structure sbc of type compressed_string_bounded_counters.

The output of this helper function is the counter at index i*sbc->sampling_rate into the data structure sbc of type compressed_string_bounded_counters, and has type u(32).

7.2.2.3 BitmapDecodeBlock

The helper function BitmapDecodeBlock shall return the decoded bitmap block of a particular block stored under the bitmap field of the data structure compressed_string_bitmap defined in subclause 7.1.5.6.

The inputs of this helper function are:

— block_size, of type u(8), denoting the length of the bitmap block contained in the input variable encoded.

— ones, of type u(8) is the number of non-zero bits in the input variable encoded.

— encoded, of type u(64), contains a bitmap block.

The output of this function is the variable res of type u(64), which is the decoded contents of the block contained within the variable encoded.

7.2.2.4 BitmapGetRankAndBit

The helper function BitmapGetRankAndBit shall return the rank and the bit at a particular position into the uncompressed bitmap field of the data structure compressed_string_bitmap defined in subclause 7.1.5.6.

The inputs of this helper function are:

— bitmap, stored under compressed_string_bitmap->bitmap as defined in subclause 7.1.5.6.

— pos, of type u(32), the position for which the rank and the bit shall be computed in the uncompressed bitmap

The outputs of this helper function are:

— rank, of type u(32), the number of non-zero bits up until to the position specified by pos, excluding pos itself

— bit, of type u(1), the bit at position pos

7.2.3 Substring decoding process

This process specifies in Table 84 how to decode a range of characters (i.e., a substring of the original uncompressed string) from a compressed string index.

Inputs are:

— csi, a compressed string index, as specified in subclause 7.1.5, from which to extract a substring.

— start, the u(32) position in the uncompressed string index of the first byte to decode (inclusive)

— end, the u(32) position in the uncompressed string index of the last byte to decode (inclusive)

start shall not be greater than end.

end shall be smaller than csi->len.

The output is the variable res, containing the substring specified by the inputs start and end, which has type st(v).

Table 84 — Decoding process to extract a substring from a compressed string index

Decoding process	Description
`compressed_string_index_decode(csi, start, end) {`	
`  pos = compressed_string_index_inverse( csi, end + 1 >= csi.n ? 0 : end + 1)`	As specified in sub-clause 7.2.4
`  res = ""`	It has type st(v). Empty string
`  len = end - start + 1`	
`  for(i = 0; i < len; i++) {`	
`    ch = compressed_string_index_decode_char( csi, pos)`	Type c(1). As specified in sub-clause 7.2.5
`    res = ch + res`	
`    pos = compressed_string_index_LF(csi, pos)`	As specified in sub-clause 7.2.6
`  }`	
`}`	

7.2.4 Suffix array lookup process

This process specifies in Table 85 how to compute the suffix array of a given position in a compressed string index.

The inputs are:

— csi is the compressed string index, as specified in subclause 7.1.5.2

— pos, the u(32) position in the uncompressed string index for which the suffix array of the position shall be computed.

pos shall be smaller than csi.n.

The output is the variable res, which contains the suffix array element at position pos and has type u(32).

Table 85 — Decoding process to compute the suffix array of a position of the compressed string index

Decoding Process	Remarks
`compressed_string_index_lookup(csi, pos) {`	
`  dist = 0`	
`  while (pos % csi.sampling_rate_sa > 0) {`	
`    pos = compressed_string_index_LF(csi, pos)`	As specified in sub-clause 7.2.6
`    dist++`	
`  }`	
`  res = ( CompactCountersGet( csi.sa, pos / csi.sampling_rate_sa) + dist ) % csi.n`	As specified in sub-clause 7.2.1.1
`}`	

7.2.5 Inverse suffix array process

This process specifies in Table 86 how to compute the inverse suffix array (ISA) element i for a given position pos in a compressed string index such that SA[i]=pos.

The inputs are:

— csi is the Compressed String Index, as specified in subclause 7.1.5.2

— pos is the u(32) position for which the ISA shall be computed.

pos shall always be smaller than csi->len.

The output is the variable i, which contains the inverse suffix array element at position pos and has type u(32).

Table 86 — Decoding process to compute the inverse suffix array of a position of the Compressed String Index

Decoding Process	Remarks
compressed_string_index_inverse(csi, pos) {	
`pos = csi.n - pos`	
`dist = pos % csi.sampling_rate_lf`	
`i = CompactCountersGet( csi.lf, pos / csi.sampling_rate_lf)`	As specified in sub-clause 7.2.1
`while (dist > 0) {`	
`i = compressed_string_index_LF(csi, i)`	As specified in sub-clause 7.2.6
`dist--`	
`}`	
}	

7.2.6 Character decoding process

This process specifies in Table 87 reconstructing the character which used to be at a specified position in the original, uncompressed string given a compressed string index and a position within it.

The inputs are:

— csi is the compressed string index, as specified in subclause 7.1.5.2

— pos is the u(32) position of the character to decode

The output is res, which contains the character decoded, and has type u(8).

Table 87 — Decoding process to extract one character from a compressed string index

Decoding Process	Remarks
compressed_string_index_decode_char(csi, pos) {	
`where = pos`	
`node = csi.wt.root`	
`while (!node->is_terminal) {`	
`{ rank, bit } = BitmapGetRankAndBit(node.bitmap, where)`	As specified in sub-clause 7.2.1
`if (bit) {`	
`where = rank`	
`node = node.right`	
`} else {`	
`where -= rank`	
`node = node.left`	
`}`	

Table 87 *(continued)*

Decoding Process	Remarks
`  }`	
`  res = node.min_char`	
}	

7.2.7 LF-mapping process

This process specifies in Table 88 how to compute the last-to-first mapping (LF-mapping) for a position given a compressed string index and a position within it,

— csi is the Compressed String Index, as specified in subclause 7.1.5.2

— pos is the u(32) position of the LF-mapping to compute

The output is res, which contains the LF-mapping for the position pos, and has type u(32).

Table 88 — Process to compute the LF-mapping for a position of the compressed string index

Decoding Process	Remarks
compressed_string_index_LF(csi, pos) {	
`  where = pos`	
`  node = csi.wt.root`	
`  while (!node.is_terminal) {`	
`    { rank, bit } = BitmapGetRankAndBit(node.bitmap, where)`	As specified in subclause 7.2.1.4
`    if (bit) {`	
`      where = rank`	
`      node = node.right`	
`    } else {`	
`      where -= rank`	
`      node = node.left`	
`    }`	
`  }`	
`  if ( pos == csi.wt.red_isa_0 ) {`	
`    return csi.wt.c[node.min_char]`	
`  }`	
`  if ( node.min_char == csi.wt.char_red_isa_0 && pos < csi.wt.red_isa_0) {`	
`    return csi.wt.c[node.min_char] + where + 1`	
`  }`	
`  res = csi.wt.c[node.min_char] + where`	
}	

7.2.8 Extended LF-mapping process

This process specifies in Table 89 how to compute the LF-mapping of the last occurrence of encoded character ch in the compressed string index with position ≤ pos. If ch does not exist in position ≤ pos or pos == 0, it returns the total number of occurrences of all characters in the alphabet with lexicographic order smaller than that of ch. If pos == n, it returns the total number of occurrences of all characters in the alphabet with lexicographic order ≤ that of ch. The inputs are:

— csi, a compressed string index, as specified in subclause 7.1.5.2

— ch, a character

— pos, a u(32) position of the extended LF-mapping to compute

The output is res, which contains the extended LF-mapping for the position pos, and has type u(32).

Table 89 — Process to compute the extended LF-mapping for a position of the compressed string index

Decoding Process	Remarks
compressed_string_index_ELF(csi, ch, pos) {	
if (pos == 0) {	
res = csi.wt.c[ch]	
} else if (pos==csi.wt.n) {	
res = csi.wt.c[ch + 1]	
} else {	
res = pos	
node = csi.wt.root	
while (!node.is_terminal) {	
{ rank, bit } = BitmapGetRankAndBit(node.bitmap, res)	As specified in subclause 7.2.1.4
if (node.min_char == ch) {	
res -= rank	
node = node.left	
} else {	
res = rank	
node = node.right	
}	
}	
res += wt.c[ch]	
}	
}	

7.2.9 Substring position search process

This process specifies in Table 90 how to find all the occurrences of a pattern in the original, uncompressed string given a compressed string index and a substring. The inputs are:

— csi, a compressed string index, as specified in subclause 7.1.5.2, in which the positions of the selected substring are searched.

— text, the searched substring. It shall be a valid UTF-8 string, as specified in ISO/IEC 10646

The output is the variable res, which contains the starting positions of the text in the uncompressed string index, which has type array of u(32), and can be empty.

Table 90 — Decoding process to search for substring positions within a compressed string index

Decoding Process	Remarks
compressed_string_index_search(csi, text) {	
lo = 0	
hi = csi.n	
len = Size(text)	Number of bytes in text
while(len > 0) {	

Table 90 *(continued)*

Decoding Process	Remarks
`    len = len - 1`	
`    ch = text[len]`	`len-th byte in text, with type u(8)`
`    code = csi.encode[ch]`	
`    if ((0x80 & code) > 0) {`	`Interpret the first bit as the sign bit`
`       code = -(0x7F & code)`	`Set code to negative if the sign bit is set`
`    }`	
`    if (code < 0 || lo == hi) {`	`No result found`
`      len = 0`	
`      lo = hi`	
`    } else {`	
`      lo = compressed_string_index_ELF( csi, ch, lo)`	`As specified in subclause 7.2.7`
`      hi = compressed_string_index_ELF( csi, ch, hi)`	`As specified in subclause 7.2.7`
`    }`	
`  }`	
`  res = []`	`Empty array of u(32) with type u(32)[hi - lo]`
`  for(i = lo; i < hi; i++) {`	
`    res[i - lo] = compressed_string_index_lookup(csi, i)`	`As specified in 7.2.3`
`  }`	
`}`	

7.2.10 Searching for substring positions with the string index

This process specifies in Table 91 how all positions within the uncompressed index of a given substring are searched with the string index.

— si, a string index as specified in subclause 7.1.4

— text, the searched substring, which shall be a valid UTF-8 string, as specified in ISO/IEC 10646

The output is the variable res, which contains the starting locations of the text, which has type array of u(32), and can be empty.

Table 91 — Searching substring positions with the string index

Decoding Process	Remarks
SI_search_substrings(si, text) {	
positions[] = compressed_string_index_search(si.compressed_string_index, text)	Compressed String Index search operation, as specified in subclause 7.2.8 This operation returns an array of positions. The returned array may be empty. The positions in the returned array are byte positions within the uncompressed string index. The positions have type u(32)[].
res = positions[]	
}	

7.2.11 Decoding a subset of the string index

This procedure specifies in Table 92 how to decode the string index between a given start and end position, inclusive.

The inputs are:

— si, a string index as specified in subclause 7.1.4

— start, the u(32) starting position of the substring in the uncompressed string index

— end, the u(32) ending position of the substring in the uncompressed string index

start shall not be greater than end.

The output is the variable decodedPayload, which contains the decoded payload, and has type st(v).

Table 92 — Decoding a substring at a given position with the string index

Decoding Process	Remarks
SI_decode(si, start, end) {	
decodedPayload = compressed_string_index_decode(si.compressed_string_index, start, end)	Compressed String Index decode operation as specified in sub-clause 7.2.2.
}	

7.2.12 Decoding all the strings of a specific annotation data tile

This process specifies in Table 93 how to decode all the strings for one specific annotation data tile encoded in a string index.

The inputs are:

— si, a string index as specified in subclause 7.1.4.

— auId, corresponds to one of the ID of one of the annotation access units encoded in si.

— tileIndex1 and, optionally, tileIndex2, both of type u(64), corresponding to respectively the first and second indexes of the annotation data tile encoded in string index si.

The output is the variable res, which contains, the bi-dimensional array of strings, with the index for first dimension identifying a record within the data tile, and the index for the second dimension identifying a string within that record.

Table 93 — Decoding all the strings of one specific annotation data tile from the string index

Decoding Process	Remarks
SI_decode_tile(si, auId, tileIndex1, tileIndex2) {	
end = si.compressed_string_index->len	u(32)
searching = 1	
if (useTiles == 0) {	
i = si.num_AAUs – 1	
} else if (useTiles == 1) {	
i = si.n_tiles - 1	
tile_index_2_exists = (two_dimensional && !variable_size_tiles)	Check if there exists a second tile index. two_dimensional as specified in subclause 6.4.7.2 variable_size_tiles as specified in subclause 6.4.6.6.4 of the tile structure associated with the indexed descriptors and attributes
}	
while(searching) {	Look for the Annotation Access Unit with ID equal to auId
if (useTiles == 0) {	
search_condition = aau_id[i] != auId	
} else if (useTiles == 1) (	
search_condition = si.tile_index_1[i] != tileIndex1 || (tile_index_2_exists && Si.tile_index_2[i] != tileIndex2)	
}	
if (search_condition) {	
end = si.tile_offset[i]	u(32)
i--	
} else {	
searching = 0	
if(i > 0) {	
start = si.tile_offset[i]	u(32)
} else {	
start = 0	u(32)
}	
}	
}	
decodedPayload = SI_decode(si, start, end – 1)	Concatenation of all the strings, record indexes and string separators for the matching data tile, according to the string index payload specified in sub-clause 1.1.1. SI_decode is specified in 7.2.10. Has type st(v)
recordCount = 0	
stringCount = 0	
currString = ""	Empty string, has type st(v)

Table 93 *(continued)*

Decoding Process	Remarks
`res = {}`	Empty bi-dimensional array of strings, with the index for first dimension identifying a record within the data tile, and the index for the second dimension identifying a string within that record. Has type st(v)[][]
`len = Size(decodedPayload)`	Number of bytes in decodedPayload has type u(32)
`for(i = 0; i < len; i++) {`	
`  ch = decodedPayload[i]`	Has type c(1)
`  chVal = Ord(ch)`	Where Ord() returns the numerical ASCII value of c has type u(8)
`  if(chVal == 1) {`	Check for string terminator
`    res[recordCount][stringCount] = currString`	
`    currString = ""`	Empty string
`    if(stringCount < si.numStrings - 1) {`	
`      stringCount++`	
`    } else {`	
`      stringCount = 0`	
`      recordCount++`	
`    }`	
`  } else if (currString != "" || (chVal & 0xC0 != 0x80)) {`	At the beginning of a string, skip the bytes containing the optional record index, otherwise concatenate the current byte to the current string.
`    currString = currString + ch`	
`  }`	
`}`	
`}`	

7.2.13 Retrieving whole strings with the string index

This process specifies in Table 94 how to decode a whole string and its start position, corresponding to a position within the uncompressed index, e.g. one position from the list of positions returned by SI_search_substrings() as specified in subclause 7.2.10, with the String Index.

The inputs are:

— si, is a string index as specified in subclause 7.1.4

— pos, the position in the uncompressed index of type u(64)

The outputs are the variables string and start; string has type st(v) and contains the uncompressed string, and start has type u(32) and contains the position of the uncompressed string within the uncompressed index.

Table 94 — Retrieving whole strings with the String Index

Decoding Process	**Remarks**
SI_decode_string(si, pos) {	
string = SI_decode(si, pos, pos)	SI_decode() as specified in subclause 7.2.10
searching = 1	Has type u(1)
recIndexBytes = 0	Has type u(32)

for(i = pos − 1;
      i >= 0 && searching; i--) {	

ch = SI_decode(si, i, i)	SI_decode() as specified in subclause 7.2.10. Has type c(1).
chVal = Ord(ch)	Where Ord() returns the numerical ASCII value of ch, has type u(8)
if(chVal == 1) {	
searching = 0	
} else {	
string = ch + string	
if(chVal & 0xC0 == 0x80) {	A record index may be encoded here
recIndexBytes = recIndexBytes + 1	
} else {	
recIndexBytes = 0	
}	
}	
}	
string = strslice(string, recIndexBytes)	
start = i + 1 + recIndexBytes	Has type u(32)
searching = 1	Has type u(1)
for(i = pos + 1; searching; i++) {	
ch = SI_decode(si, i, i)	SI_decode() as specified in subclause 7.2.10. Has type c(1).
chVal = Ord(ch)	Where Ord() returns the numerical ASCII value of ch,has type u(8).
if(chVal != 1) {	Check for string terminator
string = string + ch	
} else {	
searching = 0	
}	
}	
}	

strslice(string, recIndexBytes) is a function that removes the first recIndexBytes characters at the beginning of the string.

7.2.14 Retrieving data tile index(es) associated with a position and record indexes

This process is specified in Table 95. Given a position within the uncompressed index of a byte that belongs to a string encoded in the compressed index, e.g. one position from the list of positions returned by SI_search_substrings() as specified in subclause 7.2.10, the annotation access unit ID of the annotation access unit that contains the said string, the index of the record that contains the said string if useTiles is 0, or the data tile index(es) of the annotation data tile containing the said string if useTiles is 1, the index of the record containing the said string, and the index of the said string within the said record are decoded with the string index as described in the following points:

1. The input byte position pos identifies the string str that contains the byte at position pos within the uncompressed index.

2. If useTiles is 0, the ID of the annotation access unit that contains str is determined by comparing pos against the values of aau_offset[] as specified in Table 75, and retrieving the corresponding value au aau_id[] as specified in Table 75:

   If pos < aau_offset[1], then the resulting Annotation Access Unit ID is aau_id[0]

   If pos >= aau_offset[num_AAUs-1], then the resulting Annotation Access Unit ID is aau_id[num_AAUs-1], with num_AAUs as specified in Table 75

   Otherwise, for all the values of i such that aau_offset[i] <= pos < aau_offset[i+i], the resulting Annotation Access Unit ID is aau_id[i].

   If useTiles is 1, the tile index(es) of the annotation data tile containing str is determined by comparing pos against the values of tile_offset[] as specified in Table 75, and retrieving the corresponding values in tile_index_1[] and, if available, also in tile_index_2[] au aau_id[] as specified in Table 75:

      If pos < tile_offset[1], then the resulting tile indexes are the tile_index_1[0] and, if available, also tile_index_2[0].

      If pos >= tile_offset[n_tiles-1], then the resulting tile indexes are tile_index_1 and, if available, also tile_index_2[n_tiles-1], with n_tiles specified in Table 75.

      Otherwise, for all the values of i such that tile_offset[i] <= pos < tile_offset[i+i], the resulting tile indexes are tile_index_1[i] and, if available, also tile_index_2[i].

3. Decode the compressed string index backward from position pos - 1 either until decoding a whole record index recordIndex (with record index as specified in subclause 7.1.4) or until reaching the beginning of the compressed index. If the beginning of the compressed index is reached, then the recordIndex is set to 0. While decoding backward, count the number of string terminators recordIndex (with string terminators as specified in subclause 7.1.4)

   If a data tile is two dimensional, recordIndex corresponds to a cell at (i, j), which are 0-based row and column indexes counting from the top-left corner of the tile such that

      recordIndex = i * n_cols_in_tile + j

   where n_cols_in_tile is the number of columns in the data tile, since cells are concatenated in row- major order in the uncompressed string index.

4. Given the number of indexed strings per record numStrings as specified in subclause 7.1.4 and the Annotation Access Unit determined at point 2, the index within the said Annotation Access Unit of the Record that contains str is equal to recordIndex + stringIndex / numStrings if useTiles is 0.

   Given the number of indexed strings per record numStrings as specified in subclause 7.1.4 and the annotation data tile determined at point 2, the index within the said data tile of the record that contains str is equal to recordIndex + stringIndex / numStrings if useTiles is 1.

5. Given the number of indexed strings per record numStrings as specified in subclause 7.1.4 and the record determined at point 3, the index within the said Record of the string str is equal to stringIndex % numStrings.

The inputs are:

— si, a string index as specified in subclause 7.1.4

— pos, a position within the uncompressed index of type u(64)