**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC 1/SC 29/WG 11 N17139**

**Macau, CN – October 2017**

|  |  |
| --- | --- |
| **Source:** | **ISO/IEC JTC 1/SC 29/WG 11** |
| **Status:** | **Approved** |
| **Title:** | **Text of ISO/IEC 23092-1 CD, Transport and Storage of Genomic Information** |

**ISO/IEC 23092-1**

ISO TC 1/SC 29/WG 11

Secretariat: XXXX

Information Technology — ISO/IEC 23092 — Part 1: Transport and Storage of Genomic Information

CD stage

**Warning for WDs and CDs**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

© ISO 2017

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

ISO copyright office

Case postale 56 • CH-1211 Geneva 20

Tel.  + 41 22 749 01 11

Fax  + 41 22 749 09 47

E-mail  copyright@iso.org

Web  www.iso.org

Published in Switzerland.

Contents

[Foreword v](#_Toc498859154)

[Introduction vi](#_Toc498859155)

[1 Scope 1](#_Toc498859156)

[2 Normative references 1](#_Toc498859157)

[3 Terms and definitions 1](#_Toc498859158)

[4 Technical Elements 2](#_Toc498859159)

[4.1 Symbols and abbreviations 2](#_Toc498859160)

[4.1.1 Arithmetic operators 3](#_Toc498859161)

[4.1.2 Logical operators 3](#_Toc498859162)

[4.1.3 Relational operators 3](#_Toc498859163)

[4.1.4 Bitwise operators 3](#_Toc498859164)

[4.1.5 Assignment 3](#_Toc498859165)

[4.1.6 Unary operators 3](#_Toc498859166)

[5 Structure of Coded Genomic Data 3](#_Toc498859167)

[5.1 Genomic records 3](#_Toc498859168)

[5.2 Data Classes 4](#_Toc498859169)

[5.3 Access Units 5](#_Toc498859170)

[5.4 Datasets 5](#_Toc498859171)

[5.5 Selective Access 6](#_Toc498859172)

[6 Data Format 6](#_Toc498859173)

[6.1 Format structure 6](#_Toc498859174)

[6.1.1 Box order 7](#_Toc498859175)

[6.2 Syntax and semantics 8](#_Toc498859176)

[6.2.1 Method of specifying syntax in tabular form 8](#_Toc498859177)

[6.2.2 Bit ordering 9](#_Toc498859178)

[6.2.3 Specification of syntax functions and descriptors 9](#_Toc498859179)

[6.3 Syntax for representation 10](#_Toc498859180)

[6.4 Data structures common to File Format and Transport Format 10](#_Toc498859181)

[6.4.1 Dataset group 10](#_Toc498859182)

[6.4.2 Dataset 17](#_Toc498859183)

[6.4.3 Access Unit 23](#_Toc498859184)

[6.4.4 Block 27](#_Toc498859185)

[6.5 Data structures specific to File Format 28](#_Toc498859186)

[6.5.1 File Header 28](#_Toc498859187)

[6.5.2 Indexing 29](#_Toc498859188)

[6.5.3 Descriptor Stream 33](#_Toc498859189)

[6.6 Data structures specific to Transport Format 35](#_Toc498859190)

[6.6.1 Data Streams 35](#_Toc498859191)

[6.6.2 Dataset Mapping Table List 35](#_Toc498859192)

[6.6.3 Dataset Mapping Table 36](#_Toc498859193)

[6.6.4 Packet 37](#_Toc498859194)

[6.7 Reference procedure to convert Transport Format to File Format 39](#_Toc498859195)

[7 Bibliography 41](#_Toc498859196)

[Annex A Selective access strategies (informative) 42](#_Toc498859197)

[A.1 Access Unit Assembling 42](#_Toc498859198)

[A.2 Paired or multi-segment reads and Access Units 42](#_Toc498859199)

[A.3 Selective access and block based encoding 43](#_Toc498859200)

[A.4 Overlapping AUs Ranges 43](#_Toc498859201)

[A.5 Non-overlapping AUs Ranges 44](#_Toc498859202)

[Annex B Depacketization process (informative) 45](#_Toc498859203)

# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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 ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 1. [www.iso.org/directives](http://www.iso.org/directives)

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

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

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: [Foreword - Supplementary information](http://www.iso.org/iso/home/standards_development/resources-for-technical-work/foreword.htm)

The committee responsible for this document is ISO/JTC1/SC29/WG11.

This is the first edition of ISO/IEC 23092 Part 1; ISO/IEC 23092, Genomic Information Representation, is composed of the following parts:

Part 1: Transport and Storage of Genomic Information

Part 2: Coding of Genomic Information

Part 3: Genomic Information Representation APIs

Part 4: Reference Software

Part 5: Conformance Testing

# Introduction

The advent of High-Throughput Sequencing (HTS) technologies enables the adoption of genomic information in the everyday practice of several fields and the advent of genomic medicine. As a consequence, an extraordinarily growing volume of generated data is expected. The current lack of appropriate standard representations and efficient compression technologies of genomic data is widely recognized as a critical element, which seriously limits the application potential in all fields using, or planning to use, genomic data at reasonable costs.

This standard was developed in response to the worldwide demand for new effective interoperable solutions in genomic information processing at the core of all kind of applications implementing the chain, from sequencing to storage and analysis.

The technology enabled by this specification is designed to introduce a representation of the genomic information and related coding tools providing a significant advance over the state-of-the-art in:

* Coherency of the data structures for sequencing and alignment information, thanks to a non-ambiguous and compact format definition
* Size of the coded data, with novel compression algorithms selected among the best performing current technologies and further improved by ad-hoc experiments during the first part of the standardization process
* Speed and flexibility in coded data selective access, through newly designed data clustering and adapted storage methodologies
* Latency in data transmission and consequent fast availability at remote locations, re-designing a transmission protocol inspired by real-time application domains
* Privacy and protection of the information, introducing a flexible framework allowing customizable secured access at all layers of the data hierarchy
* Reliability of the technology and interoperability among tools and systems, thanks to a normative procedure to assess conformance to the standard 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 specification

The fundamental structure of the data representation is the *Genomic Record*. Without breaking the traditional approach of well-known pile-up formats, the MPEG-G Genomic Records provides a more compact, simple container that groups all in one the information related to a single DNA template, being that information sequencing as well as possibly sophisticated alignment content.

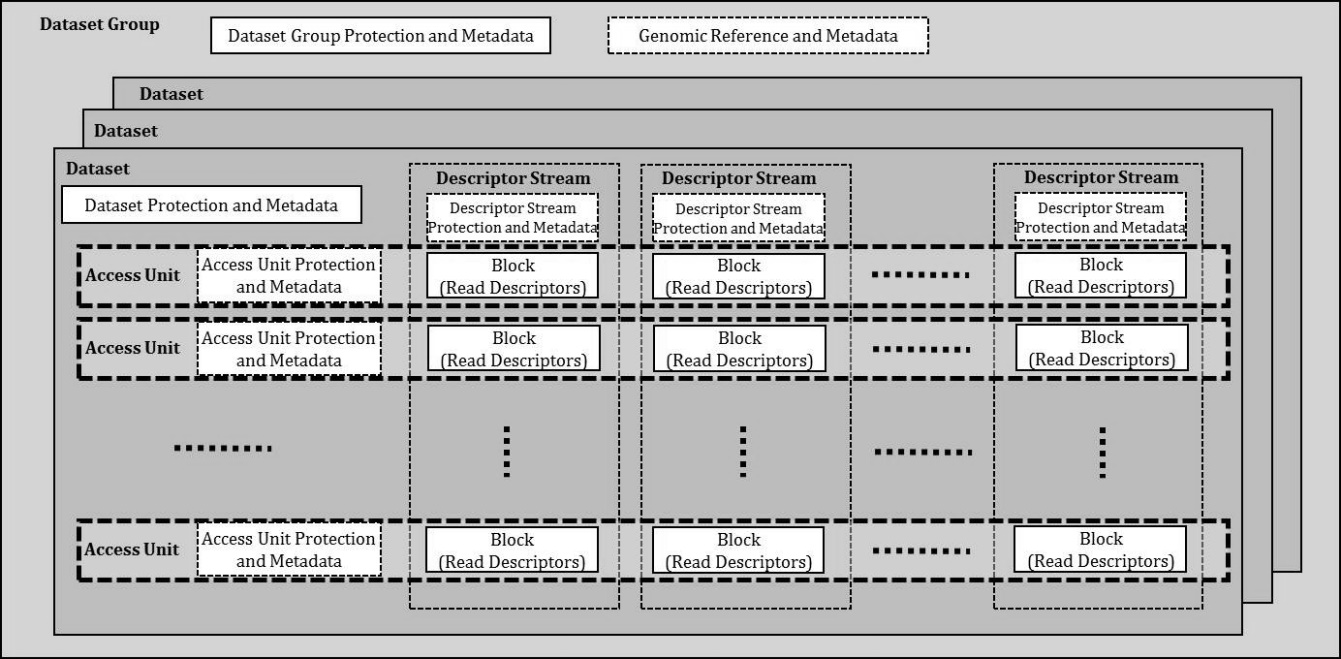
The Genomic Record is structured in mandatory and optional fields, called *Descriptors*, which cover the totality of the information provided by other state-of-the-art structures by a more compact and less ambiguous representation.

The Genomic Record is not a suitable atomic structure in terms of data compression, since all its data fields are heterogeneous: for efficient compression it is necessary to group several Records into a Cluster. Furthermore, when dealing with selective data access, the Record is quite often a too small unit to allow effective and fast information browsing.

For these reasons, this Specification introduces the concept of Access Unit, which is the fundamental structure for data coding and selective access. An Access Unit is composed by one Block for each descriptor used to represent the information of its Genomic Records, where a Block is the coded representation of all the data of the same type (descriptor) in a Cluster.

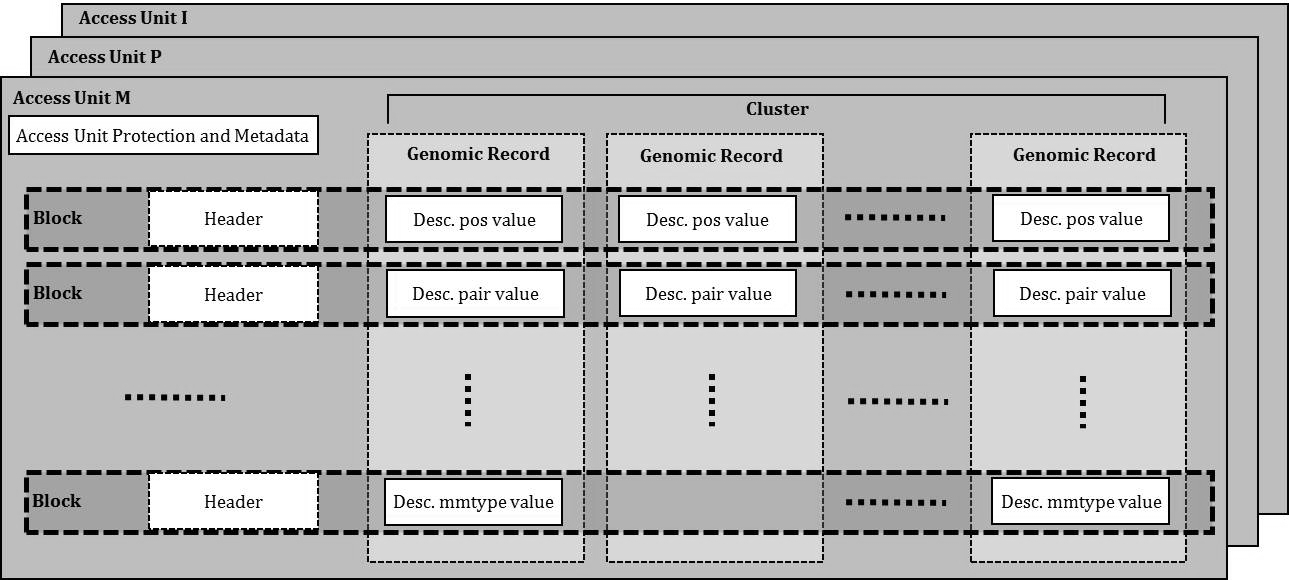
In addition to clusters of Records compressed into Access Units, genomic sequence reads are further classified in six data classes according to the result of their mapping against one or more reference sequences, when this is applicable (unmapped reads belong to one data class named “Class U”). This allows an even more powerful selective access to data, since Access Units inherit a specific data characterization (e.g. perfect matches in Class P, SNP mismatches in Class M, indels in Class I, half-mapped reads in Class HM) from the Records composing it, then providing a powerful filtering capability for very straightforward use cases.

Access Units are also the fundamental, finest grain data structure in terms of content protection and in terms of associated metadata. This means that each Access Unit can be protected individually and independently and that metadata can be associated to a single Access Unit if needed; of course this is also possible at a higher level with recursive propagation. The low level data structure is shown in **Figure 1**.



**Figure 1 – The low level data structure: Descriptors, Records, Clusters and Blocks**

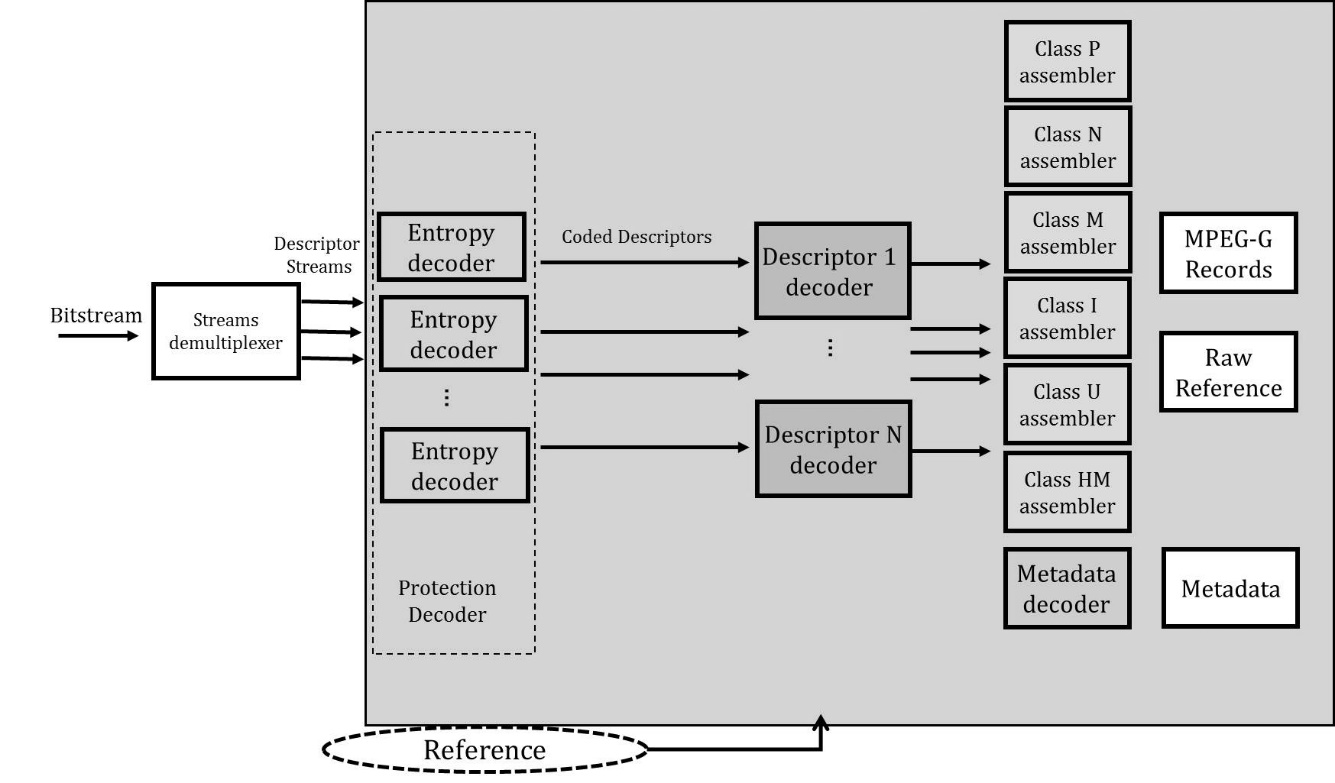
A Dataset is a coded unit containing headers and one or more Access Units. Typical Datasets may contain the complete sequencing of an individual, or a portion of it, as well as a reference genome or a subset of its chromosomes. Datasets are grouped in a Dataset Group, as shown in **Figure 2** below.



**Figure 2 – The low level data structure: Blocks, Access Units and Datasets**

In MPEG-G the coded data can be multiplexed into a normative bitstream suitable to packetization for real-time transport over typical network protocols; alternatively, coded data can be encapsulated into a standard file format for storage with the possibility to organize Blocks per descriptor stream or per Access Units, to better adapt to different application scenarios. This Specification further provides a reference process to convert a transport stream into a file and vice versa, in order to avoid differences and divergences that may arise from multiple data transfers.

This standard specifies not only the syntax and semantics of the data format, but also the decoding process to reconstruct in an unambiguous manner an encoded dataset. The decoding process is specified such that all decoders that conform to this Specification will produce numerically identical decoded output as either MPEG-G Genomic Records or Raw References. The normative decoding process applies to all layers of information from the multiplexed bitstreams in files or data streams down to descriptors and normative output. A simplified diagram of the decoding process is shown in **Figure 3**.



**Figure 3 – The decoding process**

# **Genomic Information Representation**

# Part 1:

# **Transport and Storage of Genomic Information**

# Scope

This Part of the Standard deals with data formats for both Transport and Storage of Genomic Information, with reference conversion process and informative annexes.

# Normative references

IETF – RFC 7320, *URI Design and Ownership*

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

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

# Terms and definitions

**Access Unit**

Logical data structure containing a coded representation of information to facilitate the bit stream access and manipulation.

**Alignment**

A sequence read mapped on a reference sequence

**Box**

An object-oriented building unit defined by a unique type identifier and length.

**Container box**

Box whose sole purpose is to contain and group a set of related boxes.

**Data Stream**

Set of Packets transporting the same data type.

**File Format**

Set of data structures for the storage of coded information.

**Genomic position**

Integer number representing the zero-based position of a nucleotide within a sequence.

**Genomic region**

Genomic interval between a start nucleotide position and an end nucleotide position. Both start and end position must be considered as included.

**Genomic range**

Interval of positions on a reference sequence defined by a start position *s* and an end position *e* such that *s ≤ e*; the start and the end positions are always included in the Genomic Range.

**Mapped base**

A mapped base is either:

* a base of the aligned read matching the corresponding base on the Reference Sequence

or

* a base of the aligned read that does not match the corresponding base (a.k.a. Single Nucleotide Polymorphism)

**Packet**

Transmission unit transporting segments of any of the data structures defined in this document.

**Reference genome**

Digital nucleic acid sequence database, assembled by scientists as a representative example of a species’ genetic material.

**Reference sequence**

Sequence of nucleotides associated to a one-dimensional integer coordinate system for which each integer coordinate is associated to a single nucleotide. Coordinate values can only be equal or larger than zero. This coordinate system in the context of this standard is zero-based (i.e. the first nucleotide has coordinate 0 and is said to be at position 0) and linearly increasing from left to right.

**(Genomic) Segment**

Contiguous sequence of nucleotides.

**Sequence read**

Readout, by a specific technology more or less prone to errors, of a continuous part of a segment of nucleotides extracted from an organic sample.

**Syntax field**

Element of data represented in the Data Format.

**Template**

A genomic sequence which is sequenced on a sequencing machine or assembled from sequence reads.

**Transport Format**

Set of data structures for the transport of coded information.

**Variable**

Parameter either inferred from syntax fields or locally defined in a process description.

# Technical Elements

## Symbols and abbreviations

The mathematical operators used to describe this International Standard 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.

### Arithmetic operators

+ Addition.

- Subtraction (as a binary operator) or negation (as a unary operator).

++ Increment.

-- Decrement.

\* Multiplication.

^ Power.

/ 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.

// Integer division with rounding to the nearest integer. Half-integer values are rounded away from 0 unless otherwise specified. For example 3//2 is rounded to 2, and -3//2 is rounded to -2.

DIV Integer division with truncation of the result towards -

% Modulus operator (defined only for positive numbers).

### Logical operators

|| Logical OR

&& Logical AND

! Logical NOT

### Relational operators

Greater than

Greater than or equal to

Less than

Less than or equal to

== Equal to

!= Not equal to

### Bitwise operators

& AND

| OR

Shift right with sign extension

Shift left with 0 fill

### Assignment

= Assignment operator

### Unary operators

sizeof(N) Size in bytes of N, where N is either a data structure or a data type.

# Structure of Coded Genomic Data

## Genomic records

The Genomic Record, in this Specification, is a data structure encoding either a single sequence read or a paired sequence read, optionally associated with alignment information, an identifier (read name) and quality values.

The position of a Genomic Record 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 this specification.

A base that is present in the aligned reads of the Record and not present in the Reference Sequence (an *insertion*) and bases preserved by the alignment process but not mapped on the Reference Sequence (*soft clips*) do not have mapping positions.

The following definitions further apply to the Genomic Record:

**Genomic record length**

Distance between the left-most mapped base coded in the record and the right-most mapped base coded in the record.

**Cluster**

An aggregation of Genomic Records.

## Data Classes

Genomic Records are classified in six data classes according to the result of the primary mapping(s) of their reads against one or more reference sequences.

Mapped records are classified according to the type of mismatches with respect to the reference sequences used for alignment. In case of more than one read in a template, the template belongs to the class of the read with the highest Class ID.

Records with only one read mapped (in case of paired reads) belong to one data class named “Class HM”.

Unmapped records belong to one data class named “Class U”.

The semantics of all syntax elements named class\_ID in this Specification is described in Table 1.

**Table 1 – Data Classes semantics and IDs**

|  |  |  |
| --- | --- | --- |
| **Class ID** | **Class name** | **Semantics** |
| 1 | CLASS\_P | Reads perfectly matching to the reference sequence. |
| 2 | CLASS\_N | Reads containing mismatches which are unknown bases only. |
| 3 | CLASS\_M | Reads containing mismatches which are substitutions only. |
| 4 | CLASS\_I | Reads containing mismatches which are substitutions, indels and soft clips. |
| 5 | CLASS\_HM | Half-mapped pairs where only one read is mapped. |
| 6 | CLASS\_U | Unmapped reads. |

Each Data Class is represented by means of several descriptors; conversely, a descriptor is an encoding element needed to decode part of the encoded information. Mandatory and optional descriptors for each Data Class are specified in Part 2 of this Specification.

The following definitions apply:

**Block**

Unit composed by a header and a payload, which is composed by portions of compressed descriptors of the same type.

**Descriptor Stream**

Sequence of encoded descriptor blocks used to decode a descriptor of a specific Data Class.

## Access Units

Access Units are data structure containing a coded representation of genomic information or related metadata to facilitate the bit stream access and manipulation.

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 by all and only those blocks of the descriptor streams that are necessary to decode the requested information in a cluster of a given Data Class.

Access Unit can be of several types according to the nature of the coded data.

**Table 2 – Access Unit Type**

|  |  |  |
| --- | --- | --- |
| **Access Unit Type** | | **Type 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 |

According to the type of coded information, an Access Unit can be decoded either independently of any other AU or using information contained in other Access Units.

The following definitions apply to Access Units:

**Access Unit Start Position**

Left-most Genomic Record Position among all Genomic Records contained in the Access Unit.

**Access Unit End Position**

Right-most base position among all mapped bases of all Genomic Records contained in the Access Unit.

**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.

**Access Unit Size**

Number of Genomic Records contained in an Access Unit.

**Access Unit Covered Region**

Genomic Range comprised between the AU Start Position and the AU End Position.

## Datasets

A Dataset is a compression unit containing headers and Access Units. The set of Access Units composing the Dataset constitutes the Dataset Payload.

A collection of one or more Datasets is called Dataset Group.

## Selective Access

In the case of selective access to a genomic region comprised between a *start* genomic position and an *end* genomic position the decoder will return all Genomic Records having at least one base mapped in that interval.

# Data Format

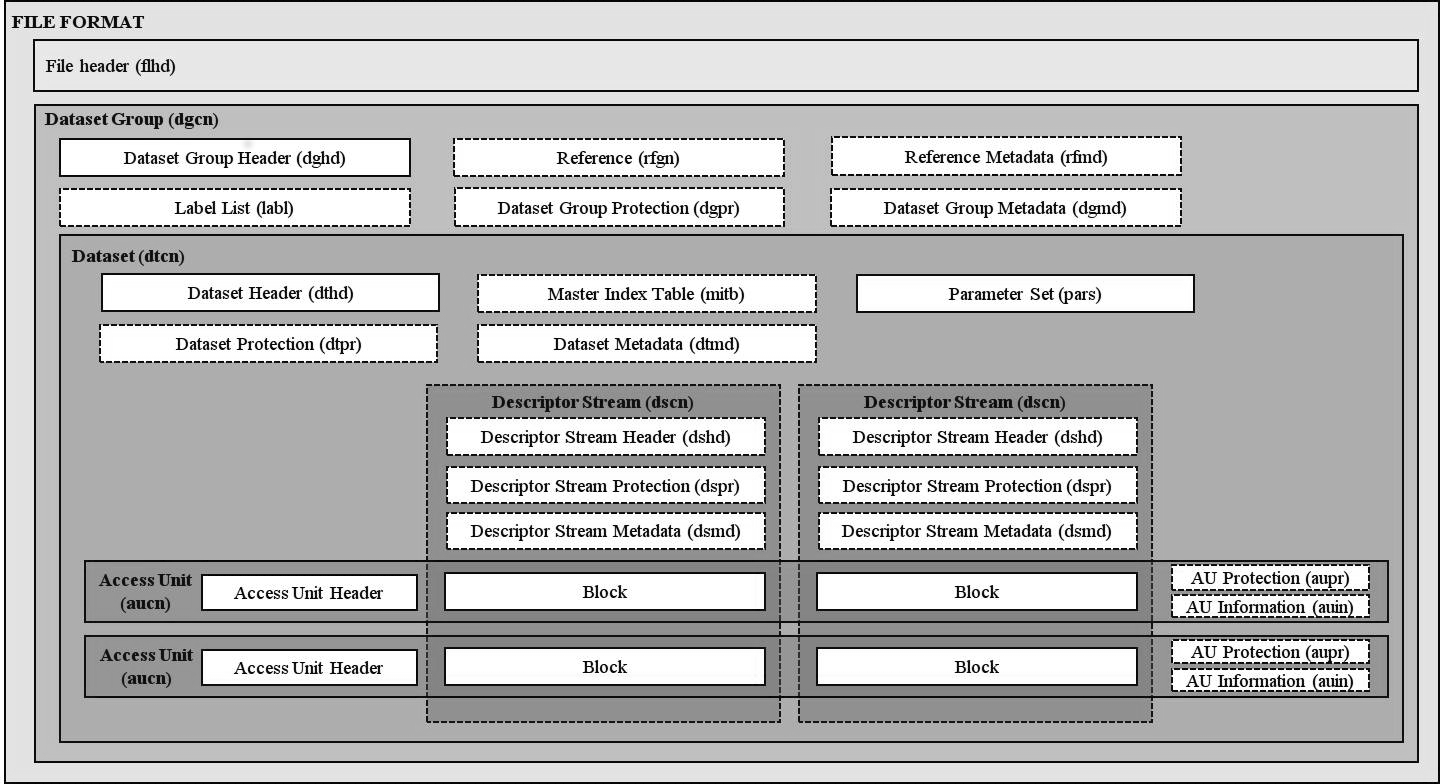
## Format structure

**Table 3** presents the overall data structures and hierarchical encapsulation levels.

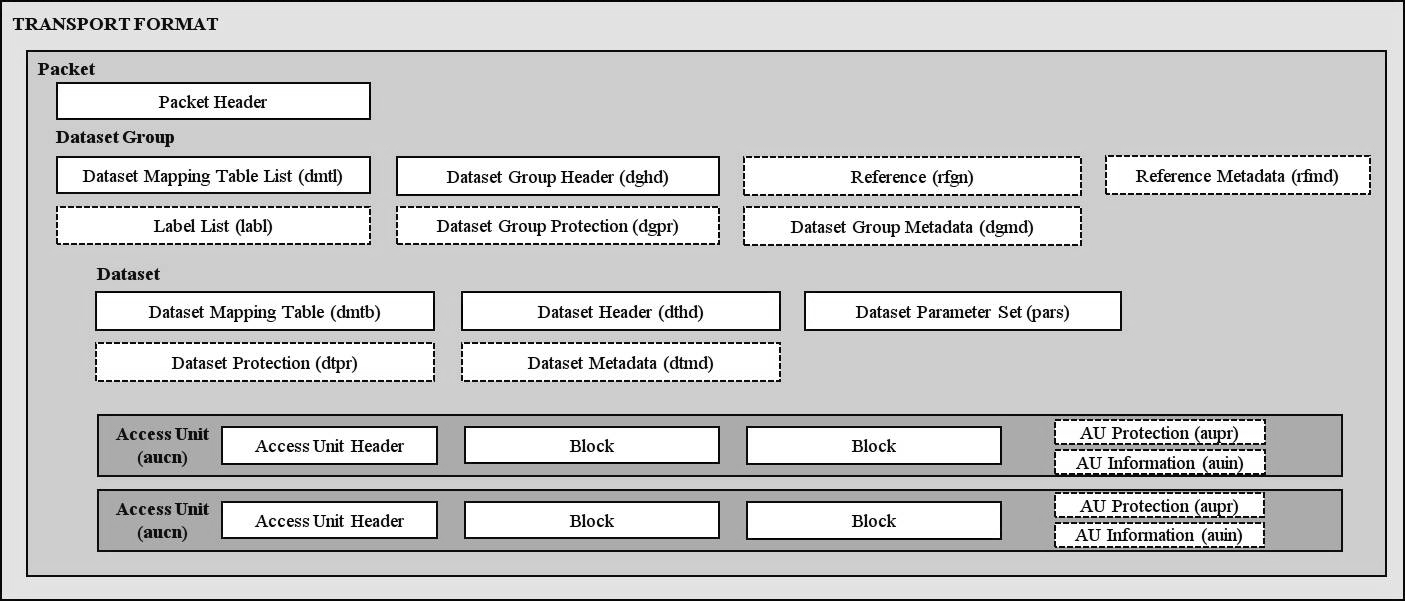
Boxes that may occur at the top-level are shown in the left-most column; indentation is used to show possible containment. Not all boxes need be used in all files; the mandatory boxes are marked with an asterisk (\*) in the “Mandatory” column: such column refers to the relevant scope (File and/or Transport). Optional boxes are represented with dashed borders in **Figure 4** and **Figure 5**. When no entry is present in the Scope column, scope is both File and Transport. See the description of the individual boxes for a discussion of what must be assumed if the optional boxes are not present.

**Table 3 – Format structure and encapsulation levels**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Structure Name (with hierarchical level)** | | | | | **Key** | **Clause** | **Scope** | **Mandatory** |
| file\_header |  |  |  |  | flhd | 6.5.1 | File | \* |
| dataset\_group |  |  |  |  | dgcn | 6.4.1 | File | \* |
|  | dataset\_group\_header |  |  |  | dghd | 6.4.1.1 |  | \* |
|  | reference |  |  |  | rfgn | 6.4.1.2 |  |  |
|  | reference\_metadata |  |  |  | rfmd | 6.4.1.3 |  |  |
|  | label\_list |  |  |  | labl | 6.4.1.4 |  |  |
|  | DG\_metadata |  |  |  | dgmd | 6.4.1.4.2 |  |  |
|  | DG\_protection |  |  |  | dgpr | 6.4.1.5.2 |  |  |
|  | dataset\_mapping\_table\_list |  |  |  | dmtl | 6.7.1 | Transport | \* |
|  | dataset |  |  |  | dtcn | 6.4.1.6.2 | File | \* |
|  |  | dataset\_header |  |  | dthd | 6.4.2.1 |  | \* |
|  |  | master\_index\_table |  |  | mitb | 6.5.2.1 | File |  |
|  |  | parameter\_set |  |  | pars | 6.4.2.2 |  | \* |
|  |  |  |  |  |  |  |  |  |
|  |  | DT\_metadata |  |  | dtmd | 6.4.2.2.2 |  |  |
|  |  | DT\_protection |  |  | dtpr | 6.4.2.3.2 |  |  |
|  |  | dataset\_mapping\_table |  |  | dmtb | 6.7.2 | Transport | \* |
|  |  | descriptor\_stream |  |  | dscn | 6.5.1.2 | File |  |
|  |  |  | descriptor\_stream\_header |  | dshd | 6.5.3.2 | File |  |
|  |  |  | DS\_metadata |  | dsmd | 6.5.3.2.2 | File |  |
|  |  |  | DS\_protection |  | dspr | 6.5.3.3.2 | File |  |
|  |  | access\_unit |  |  | aucn | 6.4.3 |  | \* |
|  |  |  | access\_unit\_header |  | auhd | 6.4.3.1 |  | \* |
|  |  |  | AU\_information |  | auin | 6.4.3.2 |  |  |
|  |  |  | AU\_protection |  | aupr | 6.4.3.3 |  |  |
|  |  |  | block |  |  | 6.4.4 |  | \* |
|  |  |  |  | block\_header |  | 6.4.4 |  |  |
| packet |  |  |  |  |  | 6.7.3 | Transport | \* |
|  | packet\_header |  |  |  |  | 6.7.3.1 | Transport | \* |



**Figure 4 – Data structures hierarchy for Storage (informative)**



**Figure 5 – Data structures hierarchy for Transport (informative)**

In Transport Format, any data structure represented in **Figure 5** shall be encapsulated in one or more Packets, as specified in clause 6.7.3.

### Box order

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

1. When present, the file\_header box ‘flhd’ shall occur before any variable-length box.
2. In File Format, it is strongly recommended that all header boxes be placed first in their container.
3. It is mandatory to access first all boxes at a higher or same hierarchy level.
4. In Transport Format, it is strongly recommended that the dataset\_mapping\_table\_list and the dataset\_mapping\_table boxes be transmitted first.
5. In File Format, it is strongly recommended that all the container boxes (dgcn, dtcn, dscn, aucn) and the block follow, when present, all other boxes of the same and higher hierarchy levels, as specified in **Table 3**.

## Syntax and semantics

### Method of specifying syntax in tabular form

The following constructs 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.

|  |  |
| --- | --- |
| **Construct** | **Description** |
| while (condition) {  data\_element  . . .  } | If the condition is true, then the group of data elements occurs next in the data stream. This repeats until the condition is not true. |
| do {  data\_element  . . . }  while (condition) | The data element always occurs at least once. The data element is repeated until the condition is not true. |
| if (condition) {  data\_element  . . .  } | If the condition is true, then the first group of data elements occurs next in the data stream. |
| else {  data\_element  . . .  } | If the condition is not true, then the second group of data elements occurs next in the data stream. |
| 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 1 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.

|  |  |
| --- | --- |
| 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+1th element of an array of data. |
| data\_element[m][n] | data\_element[m][n] is the m+1 th,n+1 th element of a two-dimensional array of data. |
| data\_element[l][m][n] | data\_element[l][m][n] is the l+1 th,m+1 th,n+1 th element of a three-dimensional array of data. |
| data\_element[i][l][m][n] | data\_element[i][l][m][n] is the i+1 th,l+1 th,m+1 th,n+1th element of a four-dimensional array of data. |
| data\_element[m..n] | data\_element[m..n] is the inclusive range of bits between bit m and bit n in the data\_element. |

### Bit ordering

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

### Specification of syntax functions and descriptors

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 descriptors specify the parsing process of each syntax element:

* b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read\_bits( 8 ).
* i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a two's complement integer representation with most significant bit written first.
* 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 descriptor 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) begins at a byte-aligned position in the bitstream and 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-aligned 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.
* c(n): sequence of n universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in **ISO/IEC 10646**.

## Syntax for representation

KLV (Key Length Value) format is used for the data structures listed in **Table 3** for which the key column is not empty.

The KLV syntax is defined as follows:

struct gen\_info

{

c(4) Key;

u(64) Length;

u(8) Value[];

}

**Note**: All data structures listed in **Table 3** and in all other syntax table contained in this Specification for which the key column is empty have no Key and no Length, but only Value[].

## Data structures common to File Format and Transport Format

### Dataset group

The Dataset Group is a collection of one or more Datasets, as specified in the Dataset Group Header syntax in **Table 5**.

The relevant container box (*dgcn* Key in **Table 4** below) is mandatory in File Format, forbidden in Transport Format.

Child boxes may be present or not, according to the column “Mandatory” in **Table 3**. Child boxes marked with suffix “[]” after their name in the Syntax column of **Table 4** may be present in multiple instances.

**Table 4 – dataset\_group syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *dataset\_group {* | *dgcn* |  |  |
| dataset\_group\_header | dghd | gen\_info | As specified in 6.4.1.1. |
| reference[] | rfgn | gen\_info | As specified in 6.4.1.2. |
| label\_list | labl | gen\_info | As specified in 6.4.1.4. |
| dataset\_mapping\_table\_list | dtml | gen\_info | As specified in 6.7.1 |
| for (i=0;i<num\_datasets;i++) { |  |  | num\_datasets: as specified in 6.4.1.1. |
| dataset[i] | dtcn | gen\_info | As specified in 6.4.1.6.2. |
| } |  |  |  |
| DG\_metadata | dgmd | gen\_info | As specified in 6.4.1.4.2. |
| DG\_protection | dgpr | gen\_info | As specified in 6.4.1.5.2. |
| *}* |  |  |  |

#### Dataset Group Header

This is a mandatory box describing the content of a Dataset Group.

##### Syntax

**Table 5 – dataset\_group\_header syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *datasets\_group­\_header {* | *dghd* |  |  |
| dataset\_group\_ID |  | u(8) |  |
| version\_number |  | u(8) |  |
| for (i=0;i<num\_datasets;i++) { |  |  |  |
| dataset\_ID[i] |  | u(16) |  |
| } |  |  |  |
| *}* |  |  |  |

##### Semantics

**dataset\_group\_ID** is used to identify this Dataset Group from any other Dataset Group. The 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.

**num\_datasets** is inferred from the Length field of datasets\_group\_header *gen\_info* header.

**dataset\_ID** is an integer number identifying the Dataset in the Dataset Group. This field shall not take the same value more than once within one version of the Dataset Group.

#### Reference

This is an optional box 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 at least one of the fields: reference\_ID, reference\_major\_version, reference\_minor\_version. In the case of two or more References sharing the same reference\_ID, priority shall be given to the one with the highest value of reference\_major\_version. In the case of two or more References sharing the same reference\_ID and the same reference\_major\_version, priority shall be given to the one with the highest value of reference\_minor\_version.

##### Syntax

**Table 6 – Reference box syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *reference {* | *rfgn* |  |  |
| 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(32) |  |
| for (seqID=0;seqID<seq\_count;seqID++) { |  |  |  |
| sequence\_name[seqID] |  | st(v) |  |
| } |  |  |  |
| external\_ref\_flag |  | u(8) |  |
| If (external\_ref\_flag) { |  |  |  |
| ref\_uri |  | st(v) | As specified in 6.4.1.2.3. |
| external\_dataset\_group\_ID |  | u(8) |  |
| external\_dataset\_ID |  | u(16) |  |
| checksum\_alg |  | u(8) |  |
| for (seqId=0;seqId<seq\_count;seqId++) { |  |  |  |
| checksum[seqId] |  | i(checksum\_size) | As specified in 6.4.1.2.4. |
| } |  |  |  |
| } |  |  |  |
| else { |  |  |  |
| dataset\_group\_ID |  | u(8) |  |
| dataset\_ID |  | u(16) |  |
| } |  |  |  |
| *}* |  |  |  |

##### Semantics

**reference\_ID** is a 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** is an unambiguous string identifier for each reference sequence contained in the Reference.

**ref\_uri** as specified in sub-clause 6.4.1.2.3.

**external\_dataset\_group\_ID** is the identifier of the Dataset Group containing the external reference, in case ref\_uri points to a reference compliant to this Specification. In this case the value shall exist in the external resource as a dataset\_group\_ID, as specified in clause 6.4.1.1. It shall be assigned to 0xFF when ref\_uri points to a reference compliant to the FASTA format, as specified in clause 6.4.1.2.4.

**external\_dataset\_ID** is the identifier of the Dataset containing the external reference, in case ref\_uri points to a reference compliant to this Specification. The value shall be equal to one of the dataset\_ID belonging to the Dataset Group identified by external\_dataset\_group\_ID. Ignore when external\_dataset\_group\_ID is equal to 0xFF.

**checksum** is the checksum computed on each reference sequence contained in the reference genome retrieved using ref\_uri, according to one of the methods specified in clause 6.4.1.2.4.

**dataset\_group\_ID** is an integer number identifying the Dataset Group containing the internal reference.

**dataset\_ID** is an integer number identifying the Dataset containing the internal reference.

##### ref\_uri semantics

It is compliant to IETF – RFC 3986 and IETF - RFC7320.

It may point to either:

* a reference compliant to this Specification, , i.e. coded as specified in clause 6.4.2, or
* a raw\_reference, as specified in clause 7.2 of ISO/IEC 23092-2, or
* a reference compliant to the FASTA specification, as described in clause 6.4.1.2.4

In the second case, the raw\_reference shall be dispatched to a decoder compliant with ISO/IEC 23092-2 as a Data Unit (specified in clause 7.1 of ISO/IEC 23092-2); data\_unit\_type shall be equal to 0 and data\_unit\_size equal to the number of bytes composing the actual raw\_reference structure.

In the third case, the FASTA reference shall be converted into a raw reference, according to clause 6.4.1.2.4, and the resulting raw\_reference shall be dispatched to a decoder compliant with ISO/IEC 23092-2 as a Data Unit (specified in clause 7.1 of ISO/IEC 23092-2); data\_unit\_type shall be equal to 1 and data\_unit\_size equal to the number of bytes composing the actual raw\_reference structure.

ref\_uri shall follow the IETF – RFC 3986 specification which is partially summarized here for information.

The generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment.

URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

hier-part = "//" authority path-abempty

/ path-absolute

/ path-rootless

/ path-empty

The scheme and path components are required, though the path may be empty (no characters). When authority is present, the path must either be empty or begin with a slash ("/") character. When authority is not present, the path cannot begin with two slash characters ("//"). These restrictions result in five different augmented BNF (RFC 5234) rules for a path, only one of which will match any given URI reference. See RFC 3986 for more details on the five ABNF rules. The following is an example URI and its component parts:

foo://example.com:8042/over/there?name=ferret#nose

\\_/ \\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_/ \\_\_/

| | | | |

scheme authority path query fragment

##### Supported FASTA format

The FASTA format supported by this Specification is represented as a series of lines in ASCII text format.

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

Each line starting with a ">" (greater-than) symbol shall be interpreted as the identifier (a.k.a. name) of the sequence of nucleotides represented by the following one or more lines.

Each line starting with a ">" (greater-than) symbol shall be followed by one or more lines of uppercase symbols representing nucleotides.

The following is an example of supported FASTA.

|  |  |  |
| --- | --- | --- |
| Line | Content | Description |
| 1 | >1 dna:chromosome chromosome:GRCh37:1:1:249250621:1 | 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 | Second sequence identifier |
| N+1 | TGACTATCGATCTATTAGCGGCGATGCTTCCA | Sub-sequences of nucleotides representing the entire second sequence |
| N+2 | ACGTTGACAAACCGATAAGCGGCGATGCAAAC |
| N+3 | TTGACAAACCGATAAGCGGCGATGCAAACAGT |
| … | … |
| … | … | … |

An MPEG-G compliant codec will ignore any comment line starting with a semi-colon.

##### Checksum

The identification of the hash function to be used to verify the integrity of the retrieved reference sequence is performed using checksum\_alg, an 8-bit word encoded in the Dataset Header, as specified in clause 6.4.2.1. Two values of checksum\_alg are defined below, while other values are reserved for future use.

**Table 7 – 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. |

The checksum shall be calculated on the UPPERCASE string representing the reference sequence.

#### Reference Metadata

This is an optional box containing metadata associated to a Reference.

##### Syntax

**Table 8 – reference\_metadata syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *reference\_metadata {* | *rfmd* |  |  |
| dataset\_group\_ID |  | u(8) |  |
| reference\_ID |  | u(8) |  |
| reference\_metadata\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**dataset\_group\_ID** is an integer number identifying of 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, defined in clause 6.4.1.2, present in the Dataset Group.

**reference\_metadata\_value** contains reference related metadata, specified in ISO/IEC 23092-3.

#### Label List

A label is an identifier associated to one or more Datasets, genomic regions and/or classes. Several labels can be defined for one Dataset Group in a Label List.

##### Syntax

**Table 9 – Labels 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\_ID[h] |  | st(v) |  |
| num\_datasets[h] |  | u(16) |  |
| for (i=0;i<num\_datasets[h];i++) { |  |  |  |
| dataset\_ID[h][i] |  | u(16) |  |
| reference\_ID[h][i] |  | u(8) |  |
| num\_regions[h][i] |  | u(8) |  |
| for (j=0;j<num\_regions[h][i];j++) { |  |  |  |
| seq\_ID[h][i][j] |  | u(16) |  |
| num\_classes[h][i][j] |  | u(8) |  |
| for (k=0;k<num\_classes[h][i][j];k++) { |  |  |  |
| class\_ID[h][i][j][k] |  | u(8) |  |
| } |  |  |  |
| start\_pos[h][i][j] |  | u(32) |  |
| end\_pos[h][i][j] |  | u(32) |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| *}* |  |  |  |

##### 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 clause 6.4.1.1.

**num\_labels** is the total number of labels in the Label List.

**label\_ID** is a string representing the label identifier. It must be unique for each label listed in the Label List.

**num\_datasets** is the number of Datasets containing regions labelled by label\_ID.

**dataset\_ID** 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 clause 6.4.1.1.

**reference\_ID** is the Reference identifier. It shall take the value of the reference\_ID field of one of the Reference boxes included in the same Dataset Group, as specified in clause 6.4.1.2.

**num\_regions**: is the number of regions composing label\_ID in the Dataset.

**seq\_ID** is the sequence identifier. It shall take the value of one of the seqID of at least one of the Reference boxes included in the Dataset Group, as specified in clause 6.4.1.2.

**num\_classes** is the number of classes labelled by label\_ID in the region.

**class\_ID** identifies the Data Class in the region labelled by label\_ID.

**start\_pos** is the position of the first nucleotide in the first read of the region.

**end\_pos** is the position of the first nucleotide in the last read of the region.

#### Dataset Group Metadata

This is an optional box containing metadata associated to a Dataset Group.

##### Syntax

**Table 10 – dataset\_group\_metadata syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *DG\_metadata {* | *dgmd* |  |  |
| DG\_metadata\_value[] |  |  | Specified in **ISO/IEC 23092-3**. |
| *}* |  |  |  |

##### Semantics

**DG\_metadata\_value** contains the Dataset Group metadata, specified in **ISO/IEC 23092-3**.

#### Dataset Group Protection

This is an optional box 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.

##### Syntax

**Table 11 – DG\_protection syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *DG\_protection {* | *dgpr* |  |  |
| DG\_protection\_value[] |  |  | Specified in **ISO/IEC 23092-3**. |
| *}* |  |  |  |

##### Semantics

**DG\_protection\_value** contains the Dataset Group protection information, specified in **ISO/IEC 23092-3**.

### Dataset

A Dataset is a compression unit containing sequence reads and possibly alignment information.

The relevant container box (*dtcn* in **Table 12** below) is mandatory in File Format, forbidden in Transport Format.

Child boxes may be present or not, according to the column “Mandatory” in **Table 3**. Child boxes marked with suffix “[]” after their name in the Syntax column of **Table 12** may be present in multiple instances.

**Table 12 – Dataset syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *dataset {* | *dtcn* |  |  |
| dataset\_header | dthd | gen\_info | As specified in 6.4.2.1. |
| master\_index\_table | mitb | gen\_info | As specified in 6.5.2.1. |
| dataset\_parameter\_set[] | pars | gen\_info | As specified in 6.4.2.2. |
| dataset\_mapping\_table | dmtb | gen\_info | As specified in 6.7.2. |
| access\_unit[] | aucn | gen\_info | As specified in 6.4.3. |
| if (block\_header\_flag == 0) { |  |  |  |
| descriptor\_stream[] | dscn | gen\_info | As specified in 6.5.3. |
| } |  |  |  |
| DT\_metadata | dtmd | gen\_info | As specified in 6.4.2.2.2. |
| DT\_protection | dtpr | gen\_info | As specified in 6.4.2.3.2. |
| *}* |  |  |  |

#### Dataset Header

This is a mandatory box describing the content of a Dataset.

##### Syntax

**Table 13 – dataset\_header syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *dataset\_header {* | *dthd* |  |  |
| dataset\_group\_ID |  | u(8) |  |
| dataset\_ID |  | u(16) |  |
| major\_brand |  | u(16) |  |
| minor\_version |  | u(16) |  |
| unmapped\_indexing\_flag |  | u(1) |  |
| byte\_offset\_size\_flag |  | u(1) |  |
| non\_overlapping\_AU\_range |  | u(1) |  |
| block\_header\_flag |  | u(1) | As specified in 6.4.2.1.4. |
| if (block\_header\_flag) { |  |  |  |
| mit\_flag |  | u(1) |  |
| cc\_mode\_flag |  | u(1) |  |
| block\_start\_code\_prefix\_flag |  | u(1) |  |
| } |  |  |  |
| else { |  |  |  |
| ordered\_blocks\_flag |  | u(1) | As specified in 6.4.2.1.4. |
| } |  |  |  |
| while( !byte\_aligned( ) ) |  |  | As specified in 6.2. |
| nesting\_zero\_bit |  | u(1) | Equal to 0. |
| seq\_count |  | u(16) |  |
| 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) |  |
| num\_classes |  | u(4) |  |
| for (ci=0;ci<num\_classes;ci++) { |  |  |  |
| clid[ci] |  | u(4) |  |
| num\_descriptors[ci] |  | u(5) |  |
| for (desc\_ID=0;desc\_ID<num\_descriptors[ci];desc\_ID++) { |  |  |  |
| descriptor\_ID[ci][desc\_ID] |  | u(7) |  |
| } |  |  |  |
| while( !byte\_aligned( ) ) |  |  |  |
| nesting\_zero\_bit |  | u(1) | Equal to 0. |
| alphabet\_ID |  | u(8) | As specified in 6.4.2.1.3. |
| num\_U\_clusters |  | u(32) |  |
| U\_signature\_constant\_length |  | u(1) |  |
| U\_signature\_size |  | u(8) |  |
| if (U\_signature\_constant\_length){ |  |  |  |
| U\_signature\_length |  | u(8) |  |
| } |  |  |  |
| num\_U\_access\_units |  | u(32) |  |
| multiple\_alignment\_flag |  | u(1) |  |
| multiple\_signature\_base |  | u(31) |  |
| tflag[0] |  | u(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] \*/ |  |  |  |
| } |  |  |  |
| while( !byte\_aligned( ) ) |  |  | As specified in 6.2. |
| nesting\_zero\_bit |  | u(1) | Equal to 0. |
| *}* |  |  |  |

##### Semantics

**dataset\_group\_ID** is the identifier of Dataset Group containing the Dataset including this dataset\_header.

**dataset\_ID** is the identifier of the Dataset. Its value shall be one of the IDs listed in the dataset\_group\_header.

**major\_brand** is the brand identifier, identifying the data (compression) format specification which the Value[] field of the Dataset defined in clause 6.4.1.6.2 complies with.

**minor\_version** is the minor version number of the data (compression) format specification which the Value[] field of the Dataset defined in clause 6.4.1.6.2 complies with.

**read\_length** is the length of reads in bytes. 0 in case of variable reads length.

**mit\_flag**: if set, the Master Index Table defined in clause 6.5.2.1 is present. Otherwise, the Master Index Table is not present. It is always equal to 0 in Transport Format and equal to 1 by default when not present.

**unmapped\_indexing\_flag**: if set, the indexing information concerning Access Units with AU\_type equal to U\_TYPE\_AU is present in the Master Index Table defined in clause 6.5.2.1. It is not present otherwise.

**byte\_offset\_size\_flag:** if equal to 0, the parameter byte\_offset\_size used in the Master Index Table defined in clause 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, the parameter byte\_offset\_size is equal to 64.

**non\_overlapping\_AU\_range**: if set, all Access Units in the Dataset have non-overlapping ranges.

**block\_header\_flag**: if set, all Blocks composing the Dataset are preceded by a Block Header, as specified in clause 6.4.4.1, and the Access Unit container box, defined in clause 6.4.3, is present. It is always set in Transport Format. See also 6.4.2.1.4.

**block\_start\_code\_prefix\_flag:** if set, all Block Header boxes start by the three bytes block\_start\_code\_prefix field, equal to 0x000001, as described in clause 6.4.4.1.

**cc\_mode\_flag**: if set, two Access Units of a class cannot be separated by Access Units of a different class in the storage device. If equal to 0, Access Units are ordered by Access Unit Start Position in the storage device.

**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, defined in clause 6.5.2.1.

**seq\_count** is the number of reference sequences used in this Dataset.

**reference\_ID**: is a unique identification number of the reference used in the Dataset. It shall take the value of the reference\_ID field of at least one of the Reference boxes included in the Dataset Group including this Dataset, as specified in clause 6.4.1.2.

**seq\_ID** corresponds to the seq\_ID index in the reference box identified by reference\_ID.

**seq\_blocks** is the number of Blocks per sequence. A value of 0 means “unspecified” (e.g., in Transport Format).

**dataset\_type**: 0 = non-aligned content; 1 = aligned content; 2 = reference.

**num\_classes** is the number of classes encoded in the Dataset.

**clid** identifies the class of data carried by the Block, as specified in **Table 1**. It shall take one of the values defined as Class ID in **Table 1**. clid[i+1] shall be greater than clid[i], 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** is the maximum number of Descriptors per Class encoded in the Dataset.

**descriptor\_ID** is an unambiguous Descriptor identifier as specified in Table 10 of ISO/IEC 23092-2.

**alphabet\_ID** is the identifier of the alphabet used to encode the cluster signatures. Values are described in **Table 14** in clause 6.4.2.1.3.

**num\_U\_clusters** is the number of clusters of unmapped reads.

**U\_signature\_constant\_length**: 1 = constant length; 0 = variable length.

**U\_signature\_size** is the size in bits of each integer representing an encoded signature.

**U\_signature\_length** is the length of cluster signature as number of nucleotides.

**num\_U\_access\_units** is the total number of Access Units in the Dataset containing encoded data of class U.

**multiple\_alignment\_flag**: if set it indicates the presence of multiple alignments in the Dataset.

**multiple\_signature\_base** is the default number of signatures.

**thres** is a threshold indicating the maximum difference between the Access Unit Covered Region and the Access Unit Range.

##### Alphabets

The supported alphabets for the signature are defined as:

* for DNA
  + si = {A, G, C, T, N}
  + si = {A, G, C, T, R, Y, S, W, K, M, B, D, H, V, N, ., - } (IUPAC notation)
* for RNA
  + si = {A, G, C, U, N}
  + si = {A, G, C, U, R, Y, S, W, K, M, B, D, H, V, N, ., - } (IUPAC notation)
* for amino acids
  + si = {A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}

Each Alphabet is identified by an alphabet\_ID as shown in **Table 14** below:

**Table 14 – alphabet\_ID semantics**

|  |  |  |
| --- | --- | --- |
| **alphabet\_ID** | **Alphabet** | **bits\_per\_symbol** |
| 0 | DNA non IUPAC | 3 |
| 1 | RNA non IUPAC | 3 |
| 2 | DNA IUPAC | 5 |
| 3 | RNA IUPAC | 5 |
| 4 | Amino acids | 5 |

##### Block Contiguity

The field block\_header\_flag is used to enable two possible modes of Blocks contiguity in the File:

* Descriptor Stream Contiguity (DSC) mode: Blocks belonging to the same Descriptor Stream, as specified in clause 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 belonging to the same Access Unit, as specified in clause 6.4.3, 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 field cc\_mode\_flag 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 Blocks contiguity modes are allowed by this Specification.

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 the Master Index Table, defined in clause 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 ordered in increasing order (disregarding non-available Blocks for which the offset is equal to 1 as 0xFFFFFFFF). 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 0xFFFFFFFF, the smallest value greater than the offset of the block, if any (otherwise the descriptor\_stream\_size should be used as above).

#### Dataset Parameter Set

This is a mandatory box describing the Parameter Set 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.

##### Syntax

**Table 15 – parameter\_set syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *dataset\_parameter\_set {* | *pars* |  |  |
| dataset\_group\_ID |  | u(8) |  |
| dataset\_ID |  | u(16) |  |
| parameter\_set() |  |  | Specified in ISO/IEC 23092-2. |
| *}* |  |  |  |

##### Semantics

**dataset\_group\_ID** is the identifier of the Dataset Group containing the Dataset including this 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 Parameter Set. It shall be equal to the dataset\_id of the containing Dataset.

**parameter\_set()** is a parameter\_set structure as specified in clause 10 of Part 2. It includes the parameter\_set\_ID field used in subclause 6.4.3.1. This field shall be dispatched to a decoder compliant with ISO/IEC 23092-2, as the parameter\_set() field of a Data Unit (specified in clause 7.1 of ISO/IEC 23092-2); data\_unit\_type shall be equal to 1 and data\_unit\_size equal to the number of bytes composing this field.

#### Dataset Metadata

This is an optional box containing metadata associated to the Dataset.

##### Syntax

**Table 16 – dataset\_metadata syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *DT\_metadata {* | *dtmd* |  |  |
| DT\_metadata\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**DT\_metadata\_value** contains Dataset metadata, specified in ISO/IEC 23092-3.

#### Dataset Protection

This is an optional box containing protection information associated to the Dataset.

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

##### Syntax

**Table 17 – dataset\_protection syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *DT\_protection {* | *dtpr* |  |  |
| DT\_protection\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**DT\_protection\_value** contains Dataset protection information. Specified in ISO/IEC 23092-3.

### Access Unit

This mandatory box describes the content of an Access Unit.

This box shall be dispatched to a decoder compliant with ISO/IEC 23092-2, as the access\_unit() field of a Data Unit (specified in clause 7.1 of ISO/IEC 23092-2); data\_unit\_type shall be equal to 2 and data\_unit\_size equal to the number of bytes composing this box.

**Note**: in DSC mode (block\_header\_flag equal to 0) the Blocks shall be extracted from the Descriptor Streams and included in the resulting Access Unit as if block\_header\_flag were equal to 1.

**Table 18 – Access Unit syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *access\_unit {* | *aucn* |  |  |
| access\_unit\_header | auhd |  | As specified in 6.4.3.1. |
| if (block\_header\_flag) { |  |  |  |
| for (i=0;i<num\_blocks;i++) { |  |  | As specified in 6.4.3.1 |
| Block[i] |  |  | As specified in 6.4.4. |
| } |  |  |  |
| } |  |  |  |
| AU\_information | auin |  | As specified in 6.4.3.2. |
| AU\_protection | aupr |  | As specified in 6.4.3.3. |
| *}* |  |  |  |

#### Access Unit Header

This mandatory box describes the Access Unit.

##### Syntax

**Table 19 – Access Unit Header syntax**

**1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *access\_unit\_header* *{* | *auhd* |  |  |
| access\_unit\_ID |  | u(32) |  |
| num\_blocks |  | u(8) |  |
| parameter\_set\_ID |  | u(12) |  |
| 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 (mit\_flag == 0) { |  |  | As specified in 6.4.2.1. |
| reference\_ID |  | u(8) |  |
| sequence\_ID |  | u(16) |  |
| if (AU\_type!= U\_TYPE\_AU || dataset\_type == 2) |  |  |  |
| { |  |  |  |
| AU\_start\_position |  | u(32) |  |
| AU\_end\_position |  | u(32) |  |
| if (multiple\_alignment\_flag) { |  |  | As specified in 6.4.2.1. |
| extended\_AU\_start\_position |  | u(32) |  |
| extended\_AU\_end\_position |  | u(32) |  |
| } |  |  |  |
| } |  |  |  |
| else { |  |  |  |
| if (multiple\_signature\_base != 0) { |  |  | As specified in 6.4.2.1. |
| U\_cluster\_signature[0] |  | u(U\_signature\_size) | U\_signature\_size: as specified in 6.4.2.1. |
| if (U\_cluster\_signature[0] != 0xFFFFFFFF) { |  |  |  |
| for (i=1;i<multiple\_signature\_base;i++) { |  |  | As specified in 6.4.2.1. |
| U\_cluster\_signature[i] |  | u(U\_signature\_size) | U\_signature\_size: as specified in 6.4.2.1. |
| } |  |  |  |
| } |  |  |  |
| else { |  |  |  |
| num\_signatures |  | u(16) |  |
| for (i=0;i<num\_signatures;i++) { |  |  | As specified in 6.4.2.1. |
| U\_cluster\_signature[i] |  | u(U\_signature\_size) | U\_signature\_size: as specified in 6.4.2.1. |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| *}* |  |  |  |

##### Semantics

**access\_unit\_ID** (see 6.4.2.2.2) is an unambiguous identifier, linearly increasing by 1. It is encoded with respect to each reference sequence (identified by a combination of reference\_ID and sequence\_ID), i.e., it is reset when a new reference sequence starts. It may take the same value for different Access Units having different values of AU\_type.

**num\_blocks** is the number of Blocks in the Access Unit.

**parameter\_set\_ID** is a unique identifier of the Parameters Set in the Dataset to be used to decode the Access Unit this Access Unit Header refers to.

**AU\_type** identifies the type of Access Unit and the type of data (class) carried therein as specified in **Table 2** in subclause 5.3.

**reads\_count** is a counter of the genomic sequence reads encoded in the Access Unit.

**mm\_threshold** represents the number of mismatches (N type or M type) in a read. If set to 0 the feature of counting mismatches in encoded reads is disabled as no reads would be below threshold.

**mm\_count** represents the number of reads encoded in the Access Unit which are below the specified threshold. It shall always be set to 0 if the threshold is set to 0.

**reference\_ID** is an unambiguous ID, identifying the reference this Access Unit refers to. It shall take the value of the reference\_ID field of one of the Reference boxes included in the same Dataset Group, as specified in clause 6.4.1.2.

**sequence\_ID** is an unambiguous ID, identifying 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 clause 6.4.2.1.

**AU\_start\_position** is the position of the left-most base in the Access Unit.

**AU\_end\_position** is the position of the right-most base in the Access Unit.

**extended\_AU\_start\_position** is the absolute position on the reference sequence of the left-most mapped base among all alignments of all Genomic Records contained in the Access Unit and belonging to a Data Class as specified in clause 5.1.

**extended\_AU\_end\_position**is the absolute position on the reference sequence of the right-most mapped base among all alignments of all Genomic Records contained in the Access Unit and belonging to a Data Class as specified in clause 5.1.

**U\_cluster\_signature** is the signature of the cluster the Access Unit containing this Access Unit belongs to.

**num\_signatures** is the number of signatures.

#### Access Unit Information

This is an optional box containing information associated to the Access Unit.

##### Syntax

**Table 20 – Access Unit Information syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *AU\_information {* | *auin* |  |  |
| AU\_information\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**AU\_information\_value** contains information related to the Access Unit, as specified in ISO/IEC 23092-3.

#### Access Unit Protection

This is an optional box 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.

##### Syntax

**Table 21 – Access Unit Protection syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *AU\_protection {* | *dtpr* |  |  |
| AU\_protection\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**AU\_protection\_value** contains Access Unit protection information, as specified in ISO/IEC 23092-3.

### Block

A Block is composed by a Block Header, which is present only when the Dataset Header field block\_header\_flag, specified in clause 6.4.2.1, is equal to 1, and a Block Payload,containing compressed descriptors of the same type (descriptor\_ID) and class (class\_ID); it shall be byte-aligned, as specified in ISO/IEC 23092-2. In DSC mode, as specified in clause 6.4.2.1.4, it is a segment of a Descriptor Stream and the Block Header is not present.

**Table 22 – Block syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *block {* |  |  |  |
| if (block\_header\_flag) { |  |  |  |
| block\_header |  |  | As specified in 6.4.4.1. |
| } |  |  |  |
| for (i=0;i<block\_payload\_size;i++) |  |  | block\_payload\_size: as specified in 6.4.4.1. |
| block\_payload[i] |  |  |  |
| } |  |  |  |
| if (padding\_flag) { |  |  | As specified in 6.4.4.1. |
| padding\_size |  | u(32) |  |
| for (i=0;i<padding\_size;i++) { |  |  |  |
| padding\_byte[i] |  | u(8) |  |
| } |  |  |  |
| } |  |  |  |
| *}* |  |  |  |

Where:

**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).

**padding\_size** is the number of padding bytes.

**padding\_byte[i]** is the i-th padding byte.

#### Block Header

This box describes the Block.

It is mandatory when the block\_header\_flag field, encoded in the Dataset Header, as specified in clause 6.4.2.1, is equal to 1; optional otherwise.

**Note**: this box shall replace the Block Header provided by the underlying codec and specified in Part 2 of this Specification.

##### Syntax

**Table 23 – block\_header syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *block\_header {* |  |  |  |
| if (block\_start\_code\_prefix\_flag) { |  |  |  |
| block\_start\_code\_prefix |  | u(24) |  |
| } |  |  |  |
| descriptor\_ID |  | u(7) |  |
| padding\_flag |  | u(1) |  |
| reserved |  | u(8) |  |
| block\_size |  | u(32) |  |
| *}* |  |  |  |

##### Semantics

**block\_start\_code\_prefix** identifies the beginning of a Block. It is equal to 0x000001.

**descriptor\_ID** unambiguously identifies the Descriptor, as specified in ISO/IEC 23092-2. It shall have one of the values specified as descriptor\_ID[class\_ID][desc\_ID] in clause 6.4.2.1.

**padding\_flag**: if set, the Block contains additional padding bytes (after the payload) which are not part of the payload, as described in **Table 22**.

**block\_payload\_size** is the number of bytes composing the Block payload.

## Data structures specific to File Format

This clause specifies the data structures specific to the storage of genomic information, in addition to the data structures defined in clause 6.4 and relevant clauses of this Specification.

### File Header

This box is mandatory and provides information about the major and minor version of the File Format specification and about the set of other specifications the File complies with.

#### Syntax

**Table 24 – 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) |  |
| } |  |  |  |
| *}* |  |  |  |

#### Semantics

**major\_brand** is the major brand identifier. The value is equal to the 6 characters code “MPEG-G”.

**minor\_version** is an informative integer for the minor version of the major brand.

**num\_compatible\_brands** is inferred from the Length field in the file\_header *gen\_info* header.

**compatible\_brand[i]** is a 4 characters code representing a compatible brand.

### Indexing

#### Master Index Table

The Master Index Table provides the indexing information needed to perform selective access on specific parts of the Dataset.  
It is present in the Dataset when mit\_flag, defined in clause 6.4.2.1, 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 0xFFFFFFFF assigned to block\_byte\_offset[seq\_ID][ci][AU\_ID][desc\_ID] represents an empty Block. It is used to maintain synchronization among Blocks belonging to the same Access Unit.

##### Syntax

**Table 25 – master\_index\_table syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *master\_index\_table {* | *mitb* |  |  |
| for (seq=0;seq<seq\_count;seq++) { |  |  | seq is equal to the *seq* loop variable in clause 6.4.2.1. |
| for (ci=0; ci<num\_classes;ci++) { |  |  | ci as specified in 5.1. |
| if (clid[ci] != CLASS\_U || dataset\_type == 2) { |  |  | CLASS\_U value as specified in 5.1. clid as specified in clause 5.1. |
| for (AU\_ID=0;AU\_ID<seq\_blocks[seq];AU\_ID++) { |  |  |  |
| AU\_start\_position[seq][ci][AU\_ID] |  | u(32) |  |
| AU\_end\_position[seq][ci][AU\_ID] |  | u(32) |  |
| if (multiple\_alignment\_flag) { |  |  | As specified in 6.4.2.1. |
| extended\_AU\_start\_position[seq][ci][AU\_ID] |  | u(32) |  |
| extended\_AU\_end\_position[seq][ci][AU\_ID] |  | u(32) |  |
| } |  |  |  |
| AU\_byte\_offset[seq][ci][AU\_ID] |  | u(byte\_offset\_size) | byte\_offset\_size: as specified in 6.4.2.1. |
| if (block\_header\_flag) { |  |  |  |
| AU\_size[seq][ci][AU\_ID] |  | u(32) |  |
| } |  |  |  |
| else { |  |  |  |
| for (desc\_ID=0;desc\_ID < num\_descriptors[seq][ci];desc\_ID++) { |  |  | desc\_ID and num\_descriptors: as specified in 6.4.2.1. |
| block\_byte\_offset[seq][ci][AU\_ID][desc\_ID] |  | u(byte\_offset\_size) | desc\_ID, byte\_offset\_size): as specified in 6.4.2.1. |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| if (unmapped\_indexing\_flag && dataset\_type != 2) { |  |  | As specified in 6.4.2.1. |
| for (UAU\_ID=0;UAU\_ID<num\_U\_access\_units;UAU\_ID++) { |  |  |  |
| if (multiple\_signature\_base != 0) { |  |  | As specified in 6.4.2.1. |
| U\_cluster\_signature[UAU\_ID][0] |  | u(U\_signature\_size) | U\_signature\_size: as specified in 6.4.2.1. |
| if (U\_cluster\_signature[UAU\_ID][0] != 0xFFFFFFFF) { |  |  |  |
| for (i=1;i<multiple\_signature\_base;i++) { |  |  |  |
| U\_cluster\_signature[UAU\_ID][i] |  | u(U\_signature\_size) | U\_signature\_size: as specified in 6.4.2.1. |
| } |  |  |  |
| } |  |  |  |
| else { |  |  |  |
| num\_signatures |  | u(16) |  |
| for (i=0;i<num\_signatures;i++) { |  |  | As specified in 6.4.2.1. |
| U\_cluster\_signature[UAU\_ID][i] |  | u(U\_signature\_size) | U\_signature\_size: as specified in 6.4.2.1. |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| while( !byte\_aligned( ) ) |  |  | As specified in 6.2. |
| nesting\_zero\_bit |  | u(1) | Equal to 0. |
| if (block\_header\_flag) { |  |  |  |
| AU\_byte\_offset[UAU\_ID] |  | u(32) |  |
| AU\_size[UAU\_ID] |  | u(32) |  |
| } |  |  |  |
| else { |  |  |  |
| for (desc\_ID=0;desc\_ID < num\_descriptors[num\_classes-1];desc\_ID++) { |  |  | desc\_ID and num\_descriptors: as specified in 6.4.2.1. |
| block\_byte\_offset[UAU\_ID][desc\_ID] |  | u(32) | desc\_ID: as specified in 6.4.2.1. |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| } |  |  |  |
| *}* |  |  |  |

##### Semantics

**seq\_count** is the total number of reference sequences. It is encoded in the Dataset Header, as specified in clause 6.4.2.1. Reference sequences shall have the same order as in the Dataset Header, as specified in clause 6.4.2.1.

**num\_classes** is equal to the field num\_classes of Dataset Header, as specified in clause 6.4.2.1.

**seq\_blocks** is the number of Blocks per reference sequence; it is encoded in the Dataset Header, as specified in clause 6.4.2.1.

**AU\_ID** is the Access Unit identifier. AU\_IDs shall be assigned to Access Units by increasing value of AU\_start\_position.

**AU\_start\_position** is the absolute position on the reference sequence of the left-most mapped base among all Genomic Records contained in the AU and belonging to a Data Class as specified in clause 5.1. It is equal to 0xFFFFFFFF if the Access Unit is empty.

**AU\_end\_position** is the absolute position on the reference sequence of the right-most mapped base among all Genomic Records contained in the AU and belonging to a Data Class as specified in clause 5.1. It is equal to 0xFFFFFFFF if the Access Unit is empty.

**extended\_AU\_start\_position** is the absolute position on the reference sequence of the left-most mapped base among all alignments of all Genomic Records contained in the AU and belonging to a Data Class as specified in clause 5.1. It is equal to 0xFFFFFFFF if the Access Unit is empty.

**extended\_AU\_end\_position** is the absolute position on the reference sequence of the right-most mapped base among all alignments of all Genomic Records contained in the AU and belonging to a Data Class as specified in clause 5.1. It is equal to 0xFFFFFFFF if the Access Unit is empty.

**AU\_byte\_offset** 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).

**AU\_size** is the Access Unit size in bytes.

**num\_descriptors** is encoded in the num\_descriptors field of the Dataset Header, as specified in clause 6.4.2.1.

**block\_byte\_offset** 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 is equal to 0xFFFFFFFF. If the Block is empty and block\_header\_flag is equal to 0, it is 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 Descriptor Stream payload size.

Note: the Descriptor Stream payload size can be inferred as the difference between the L field of the dscn gen\_info header and the L field of the dshd gen\_info header.

**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 clause 6.4.2.1.

**UAU\_ID** is the identifier of Access Units with AU\_type equal to U\_TYPE\_AU. UAU\_IDs are assigned to Access Units by increasing value of U\_cluster\_signature[0].

**U\_cluster\_signature[UAU\_ID][i]** is the i-th signature of the cluster the Access Unit belongs to.

**num\_signatures** is the number of signatures.

#### U\_cluster\_signature coding

A cluster signature is represented as one or more N bits integers according to the following steps:

* According to the specific clustering algorithm adopted, the clusters signatures will be of variable or constant length. If the length is constant, a global parameter U\_signature\_length is specified in the Dataset Header (clause 6.4.2.1) to define the signature length as number of nucleotides. Otherwise the global parameter U\_signature\_constant\_length in the Dataset Header is equal to 1 accordingly and the length is not present.
* Each symbol of the supported alphabet (see **Table 14**) is uniquely associated to a binary representation of length equal to:
  + **M** =ceil(log2(cardinality of the supported alphabet)) in case of constant signature length
  + **M** = ceil(log2(cardinality of the supported alphabet) + 1) in case of variable signature length
* In case of variable signatures length, the sequence of all 0 bits is reserved to represent a special symbol called **terminator** used to signal the end of a coded signature.
* In case of constant signatures length, which is referred to as **SL**:
  + if **M** x **SL** <= N the binary representations of contiguous symbols in the signature are concatenated in a single bitstring possibly padding with 0 the Most Significant Bits if **M** \* **SL** < N.
  + if **M** \* **SL** > N the binary representations of contiguous symbols in the signature are concatenated in two or more bitstrings possibly padding with 0 the Most Significant Bits of each bitstring if **bs** is not an exact divisor of N.
  + the number of integers to be read is known as

where the *ceil* function returns the least integer that is greater than or equal to its argument.

* In case of variable signatures length, which in this Specification is referred to as **SLi** for the ith signature:
  + if **M** \* **SLi** <= N the binary representations of contiguous symbols in the signature are concatenated in a single bitstring. After all coded symbols have been concatenated the bitstring is terminated with a terminator symbol added at the Most Significant Bits positions and if necessary the remaining Most Significant Bits are padded with 0.
  + If **M** \* **SLi** > N the binary representations of contiguous symbols in the signature are concatenated in two or more bitstrings possibly padding with 0 the Most Significant Bits of each bitstring if **bs** is not an exact divisor (also known as aliquot part) of N. The last bitstring is terminated by the terminator symbol and further padded with 0 at the Most Significant Bits positions.
  + The decoder shall detect the first integer containing the terminator symbol in its binary representation in order to stop reading integers for a given signature.

### Descriptor Stream

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 when the syntax element block\_header\_flag in the Dataset Header, as specified in clause 6.4.2.1, is equal to 0; it is forbidden otherwise.

Child boxes may be present or not, according to the column “Mandatory” in **Table 3**. Child boxes marked with suffix “[]” after their name in the Syntax column of **Table 26** may be present in multiple instances.

**Table 26 – descriptor\_stream syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *descriptor\_stream {* | *dscn* |  |  |
| descriptor\_stream\_header | *dshd* | gen\_info | As specified in 0. |
| for (i=0;i<num\_blocks;i++) { |  |  | num\_blocks: as specified in 6.5.3.2.2. |
| for (j=0;j<block\_payload\_size[i];j++) { |  |  |  |
| block\_payload\_byte[i][j] |  | u(8) | As specified in 6.4.4. |
| } |  |  |  |
| } |  |  |  |
| DS\_metadata | *dsmd* | gen\_info | As specified in 6.5.3.2.2. |
| if (protection\_flag) { |  |  |  |
| DS\_protection | *dspr* | gen\_info | As specified in 6.5.3.3.2. |
| } |  |  |  |
| *}* |  |  |  |

Where:

**block\_payload\_size[i]** is inferred from the Master Index Table field block\_byte\_offset, as specified in 6.5.2.1, as difference between block\_byte\_offset[i+1] and block\_byte\_offset[i].

**block\_payload\_byte[i][j]** is the j-th byte of the Block payload. The Block payload is specified in ISO/IEC 23092-2.

#### Descriptor Stream Header

This is a box describing a Descriptor Stream. It is mandatory whenever the Descriptor Stream, described in clause 6.5.1.2, is present, forbidden otherwise.

##### Syntax

**Table 27 – descriptor\_stream\_header syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *descriptor\_stream\_header {* | *dshd* |  |  |
| descriptor\_ID |  | u(7) |  |
| class\_ID |  | u(4) |  |
| num\_blocks |  | u(32) |  |
| protection\_flag |  | u(1) |  |
| while( !byte\_aligned( ) ) |  |  | As specified in 6.2. |
| nesting\_zero\_bit |  | u(1) | Equal to 0. |
| *}* |  |  |  |

##### 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][desc\_ID] in clause 6.4.2.1.

**class\_ID** identifies the class of data carried by the Block, as specified in **Table 1**.

**num\_blocks** is the number of Blocks composing the Descriptor Stream.

**protection\_flag**: if set, the protection information of the Descriptor Stream shall be found in *DS\_protection,* else the protection information of the Descriptor Stream shall be found in *DT\_protection.*

#### Descriptor Stream Metadata

This is an optional box containing metadata associated to a Descriptor Stream.

##### Syntax

**Table 28 – DS\_metadata syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *DS\_metadata {* | *dsmd* |  |  |
| DS\_metadata\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**DS\_metadata\_value**: Descriptor Stream metadata, specified in ISO/IEC 23092-3.

#### Descriptor Stream Protection

This is an optional box 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. It is not present when protection\_flag is equal to 0, it is optional otherwise.

##### Syntax

**Table 29 – DS\_protection syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *DS\_protection {* | *dspr* |  |  |
| DS\_protection\_value[] |  |  | Specified in ISO/IEC 23092-3. |
| *}* |  |  |  |

##### Semantics

**DS\_protection\_value**: Descriptor Stream protection information, specified in ISO/IEC 23092-3.

## Data structures specific to Transport Format

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

### Data Streams

A Data Stream in MPEG-G is identified by a unique Stream ID and it can transport any of the following data structures:

* data structures containing transport information (Dataset Mapping Table List as specified in clause 6.7.1, Dataset Mapping Table as specified in clause 6.7.2),
* Dataset Group Header, as specified in clause 6.4.1.1
* Reference, as specified in clause 6.4.1.2,
* Label List, as specified in clause 6.4.1.4,
* Dataset Header, as specified in clause 6.4.2.1,
* Dataset Parameter Set, as specified in clause 6.4.2.2,
* Access Unit Header, as specified in clause 6.4.3.1,
* Blocks, as specified in clause 6.4.4,
* Metadata, and protection information, as specified in clauses 6.4.1.4.2, 6.4.1.5.2, 6.4.2.2.2, 6.4.2.3.2, 6.4.3.2, 6.4.3.3, 6.5.3.2.2 and 6.5.3.3.2.

### Dataset Mapping Table List

This is a mandatory box containing a list of all Stream\_IDs of Data Streams transporting the Dataset Mapping Tables available in 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 clause 6.7.2, the necessary and sufficient information to de-packetize and de-capsulate the Transport Format.

Each Dataset Mapping Table List is transported within a single Packet with Stream ID (SID in Packet Header, as specified in clause 6.7.3.1) equal to 0.

#### Syntax

**Table 30 – 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 |  | u(16) |  |
| } |  |  |  |
| *}* |  |  |  |

#### Semantics

**dataset\_group\_ID** is the Dataset Group ID, as in the Dataset Group Header, as specified in clause 6.4.1.1.

**num\_datasets** is inferred from the Length field in dataset\_mapping\_table\_list *gen\_info* header.

**dataset\_mapping\_table\_SID** is the Stream ID associated to the Data Stream containing the dataset\_mapping\_table . Value equal to 0 cannot be used as reserved for the Dataset Mapping Table List defined in **Table 30**.

### Dataset Mapping Table

This is a mandatory box listing all Data Streams transporting data related to the Dataset identified by dataset\_ID.

The Dataset Mapping Table associates data types (Blocks, metadata boxes, protection boxes, etc.) and Stream IDs (SID) found in Packet Header defined in clause 6.7.3.1.

**Table 32** provides the correspondence between Data Type values and data structures.  
The Dataset Mapping Table contains, along with the Dataset Mapping Table List specified in **Table 30**, the necessary and sufficient information to de-packetize and de-capsulate the Transport Format.

The Datasets Mapping Table is sent as payload of a single Packet.

The Datasets Mapping Table can be periodically re-transmitted, either updated or identical, in order to enable synchronization points in the streamed data.

#### Syntax

**Table 31 – dataset\_mapping\_table syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *dataset\_mapping\_table* { | *dmtb* |  |  |
| dataset\_ID |  | u(16) |  |
| dataset\_ef\_length |  | u(16) |  |
| if (dataset\_ef\_length>0) |  |  |  |
| extension\_field() |  | u(dataset\_ef\_length\*8) |  |
| for (i=0;i<num\_data\_streams;i++) { |  |  |  |
| data\_type |  | u(8) |  |
| reserved |  | u(3) |  |
| data\_SID |  | u(13) |  |
| data\_component\_ef\_length |  | u(16) |  |
| if (data\_component\_ef\_length>0) |  |  |  |
| data\_component\_extension\_field() |  | u(data\_component\_ef\_length\*8) |  |
| } |  |  |  |
| *}* |  |  |  |

#### Semantics

**dataset\_ID** is the dataset\_ID as in Dataset Header defined in clause 6.4.2.1.

**dataset\_ef\_length** is the Extension Field length in bytes. Equal to 0 if extension\_field is not present.

**extension\_field** is an optional extension field. Present if dataset\_ef\_length is higher than 0.

**num\_data\_streams** is inferred using the Length field in dataset\_mapping\_table gen\_info and dataset\_ef\_length.

**data\_type** identifies the type of data carried by Packets identified by subsequent data\_SID field, according to **Table 32**.

**Table 32 – data\_type field Semantics**

|  |  |  |
| --- | --- | --- |
| **data\_type** | **Data structure** | **Clause** |
| 0 | dataset\_group\_header | 6.4.1.1 |
| 1 | reference | 6.4.1.2 |
| 2 | label\_list | 6.4.1.4 |
| 3 | dataset\_header | 6.4.2.1 |
| 4 | dataset\_parameter\_set | 6.4.2.2 |
| 5 | dataset\_group\_metadata | 6.4.1.5 |
| 6 | dataset\_metadata | 6.4.2.3 |
| 7 | access\_unit\_information | 6.4.3.2 |
| 8 | descriptor\_stream\_metadata | 6.5.3.3 |
| 9 | dataset\_group\_protection | 6.4.1.6 |
| 10 | dataset\_protection | 6.4.2.4 |
| 11 | access\_unit\_protection | 6.4.3.3 |
| 12 | descriptor\_stream\_protection | 6.5.3.4 |
| 13 to 18 | access\_unit\_header with AU\_type equal to data\_type - 12. | 6.4.3.1 |
| 19 to 27 | Reserved for future use. |  |
| 28 to 44 | block with descriptor\_ID equal to data\_type – 28. | 6.4.4 |
| 45 to 255 | Reserved for future use. |  |

**data\_SID** is the Stream ID (SID) in Packet Header, as specified in clause 6.7.3.1, of Packets transporting the corresponding data\_type in the Dataset. Value equal to 0 cannot be used in the Dataset Mapping Table as reserved for the Dataset Mapping Table List defined in **Table 30**.

**data\_component\_ef\_length**: Length in bytes of data\_component\_extension\_field. Equal to 0 if data\_component\_extension\_field is not present.

**data\_component\_extension\_field**: Optional extension field containing information about the data component identified by data\_type.

### Packet

Packets are transmission units transporting in their payload segments of any of the data structures listed in **Table 32**.

The Packet payload contains bytes from one of the data structures listed in clause 6.6.1.

**Table 33 – Packet syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *packet {* |  |  |  |
| packet\_header |  | packet\_header | As specified in 6.7.3.1. |
| for (i=0;i<packet\_size–sizeof(packet\_header);i++) { |  |  | packet\_size: as specified in 6.7.3.1. |
| packet\_payload[i] |  | u(8) |  |
| } |  |  |  |
| *}* |  |  |  |

Where:

**packet\_payload[i]** is the i-th byte composing the Packet payload.

#### Packet Header

This is a mandatory box describing the Packet.

##### Syntax

**Table 34 – packet\_header syntax**

|  |  |  |  |
| --- | --- | --- | --- |
| **Syntax** | **Key** | **Type** | **Remarks** |
| *packet\_header* { |  |  |  |
| SID |  | u(13) |  |
| reserved |  | u(2) | To maintain byte-alignment. |
| access\_unit\_marker\_bit |  | u(1) |  |
| sequence\_number |  | u(8) |  |
| packet\_size |  | u(15) |  |
| extension\_flag |  | u(1) |  |
| if (extension\_flag) { |  |  |  |
| extension\_field |  |  | Reserved for future use. |
| } |  |  |  |
| *}* |  |  |  |

##### Semantics

**SID** is the Stream ID of the Data Stream containing this Packet. Unambiguously identifies data carried by this Packet, according to the dataset\_mapping\_table. It shall be unique for each Data Stream.

**access\_unit\_marker\_bit** is equal to 1 in the last Packet of the Access Unit, it is equal to 0 in all other packets. It allows identifying the end of an Access Unit.

**sequence\_number** is a packet counter linearly increasing by 1. It is needed to identify packet losses as gaps in SNs for each individual data stream. It wraps around at 255.

**packet\_size** is the number of bytes composing the Packet, including header, extension fields and payload.

**extension\_flag** is equal to 1 if extension field is present, it is equal to 0 otherwise.

**extension\_field** is an optional field, present if extension\_flag is equal to 1, not present otherwise.

## Reference procedure to convert Transport Format to File Format

This section describes the procedure to update the parameters seq\_count and seq\_blocks[seq] in the Dataset Header, defined in clause 6.4.2.1, and to compile the Master Index Table, defined in clause 6.5.2.1, from the syntax elements present in the Access Unit Header, defined in clause 6.4.3.1.

Such a procedure shall be considered as a reference procedure in order to guarantee that:

1. the resulting file is compliant to the File Format specification, and
2. the resulting file retains exactly the same information as the original Transport stream.

Any other process producing the same output from the same conformant bitstream can be equally considered as compliant to this Specification. At the same time, complying ISO/IEC 23092-1 implementations are not expected to follow the exact algorithm used by this reference procedure.

**Note**: this conversion procedure is used when the user needs to save in a storage device the data received via a Transport session, so that the resulting file is compliant to the File Format defined in clause 6.4 of this Specification.

The procedure is the following:

1. The Dataset Header field block\_header\_flag is determined by the contiguity mode of the output file.
2. seq\_count is initialized to 0 and is incremented by 1 every time sequence\_ID is different than any previously received sequence\_ID.
3. seq\_blocks[sequence\_ID] is incremented by 1 every time the (sequence\_ID, access\_unit\_ID) vector is different than any previous (sequence\_ID, access\_unit\_ID) vector.
4. The coordinates of the Master Index Table, as specified in clause 6.4.3.1, are calculated as follows:
   1. seq = seq\_count
   2. ci is inferred as follows:
      1. clid is inferred from **Table 2** using the value of AU\_type in the access\_unit\_header, defined in clause 6.4.3.1;
      2. ci is inferred from clid using the mapping loop in **Table 13**.
   3. AU\_ID = seq\_blocks[sequence\_ID]
   4. If block\_header\_flag is equal to 0:
      1. descriptor\_ID is equal to the descriptor\_ID field in Block Header, defined in clause 6.3.8.1, else (block\_header\_flag is equal to 1) descriptor\_ID = 0.
      2. desc\_ID is equal to the desc\_ID index of the entry of vector descriptor\_ID[ci][desc\_ID] in the Dataset Header, defined in clause 6.4.2.1, for which descriptor\_ID[clid][desc\_ID] is equal to descriptor\_ID.
5. If class\_ID is different than CLASS\_U:
   1. 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

* 1. 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

* 1. If multiple\_alignment\_flag is equal to 1:
     1. 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

* + 1. 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

Else if class\_ID == CLASS\_U

* 1. The Master Index Table entries named U\_cluster\_signature [UAU\_ID][0] is calculated as follows:

U\_cluster\_signature [UAU\_ID][0] = U\_cluster\_signature[0]

where UAU\_ID = AU\_ID.

* 1. If U\_cluster\_signature [UAU\_ID][0] != 0xFFFFFFFF), the Master Index Table entries named U\_cluster\_signature [UAU\_ID][i] are calculated as follows:

for (i=1;i<multiple\_signature\_base;i++) {

U\_cluster\_signature[UAU\_ID][i] = U\_cluster\_signature[i]

}

where UAU\_ID = AU\_ID.

* 1. Else (U\_cluster\_signature [UAU\_ID][0] == 0xFFFFFFFF), the Master Index Table entries named U\_cluster\_signature [AU\_ID][i] are calculated as follows:

for (i=0;i<multiple\_signature\_base;i++) {

U\_cluster\_signature[UAU\_ID][i] = U\_cluster\_signature[i]

}

where UAU\_ID = AU\_ID.

1. If block\_header\_flag is equal to 0:
   1. 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][descriptor\_ID], is created and a variable named block\_ptr[ci][descriptor\_ID] is defined and initialized to 0.
   2. block\_ptr[ci][descriptor\_ID] is incremented by 68 + Length of any child gen\_info of DSCN[ci][descriptor\_ID].
   3. The payload of all Blocks identified by class\_ID and descriptor\_ID in the Block Header is copied in the Value[] field of DSCN[ci][descriptor\_ID].
   4. For each of the above Blocks, a new variable named block\_ptr[ci][descriptor\_ID][AU\_ID] is defined and assigned the current value of block\_ptr[ci][descriptor\_ID].
   5. For each Block, block\_ptr[ci][descriptor\_ID] is incremented by block\_payload\_size.
   6. Once the session is terminated and before writing the resulting data into the output file, a variable named DS\_offset is defined, initialized to 0.
   7. After writing each DSCN[ci][descriptor\_ID] in the output file, DS\_offset is incremented by sizeof(gen\_info header) + Length(DSCN[ci][descriptor\_ID]).
   8. For each DSCN[ci][descriptor\_ID] written in the output file, the Master Index Table entry named block\_byte\_offset[seq][ci][AU\_ID][desc\_ID] is updated as follows:

block\_byte\_offset[seq][ci][AU\_ID][desc\_ID] = DS\_offset + block\_ptr[ci][descriptor\_ID][AU\_ID]

Else if block\_header\_flag is equal to 1:

* 1. A buffer named DatasetPayload is created.
  2. For each Access Unit:
     1. The Access Unit Header is written into DatasetPayload.
     2. Each Block composing the Access Unit is written into the DatasetPayload.
  3. 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.
  4. The Master Index Table entry named AU\_size[seq][ci][AU\_ID] is calculated as the sum of sizeof(access\_unit\_header) and the sum of sizeof(Block) for all received Blocks belonging to the same Access Unit.
  5. At the end of the session, DatasetPayload is dumped into the output file as Dataset Payload, as specified in clause 5.4.

1. At the end of the process, typically when the end user stops the execution of the transport process, the Master Index Table must 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].

# Bibliography

1. ISO/IEC JTC 1/SC 29/WG 11 N17139 - Text of ISO/IEC 23092-2 CD, Coding of Genomic Information, October 2017.
2. ISO/IEC JTC 1/SC 29/WG 11 N17141 – Text of ISO/IEC 23092-3 WD, Genomic Information Metadata and APIs, October 2017.

# Selective access strategies (informative)

## Access Unit Assembling

AUs can be assembled and indexed according to either one of the following two modes:

1. Overlapping Access Units: there are no constraints on the Access Unit Ranges.
2. Non-overlapping Access Units: Access Units Ranges cannot overlap.

Once Genomic Records are aggregated in AUs according to one of the two modes described above, selective access to the coded genomic information can be supported according to the different mechanisms described in the following clauses.

## Paired or multi-segment reads and Access Units

Each AU contains Genomic Records with mapping position comprised in the Access Unit Range. Therefore it is clear that Genomic Records with mapping position near the end position of the AU Range will likely be partially mapped on the genomic region covered by the following AU.

The presence of paired reads poses an even greater challenge in implementing a consistent random access mechanism, especially when:

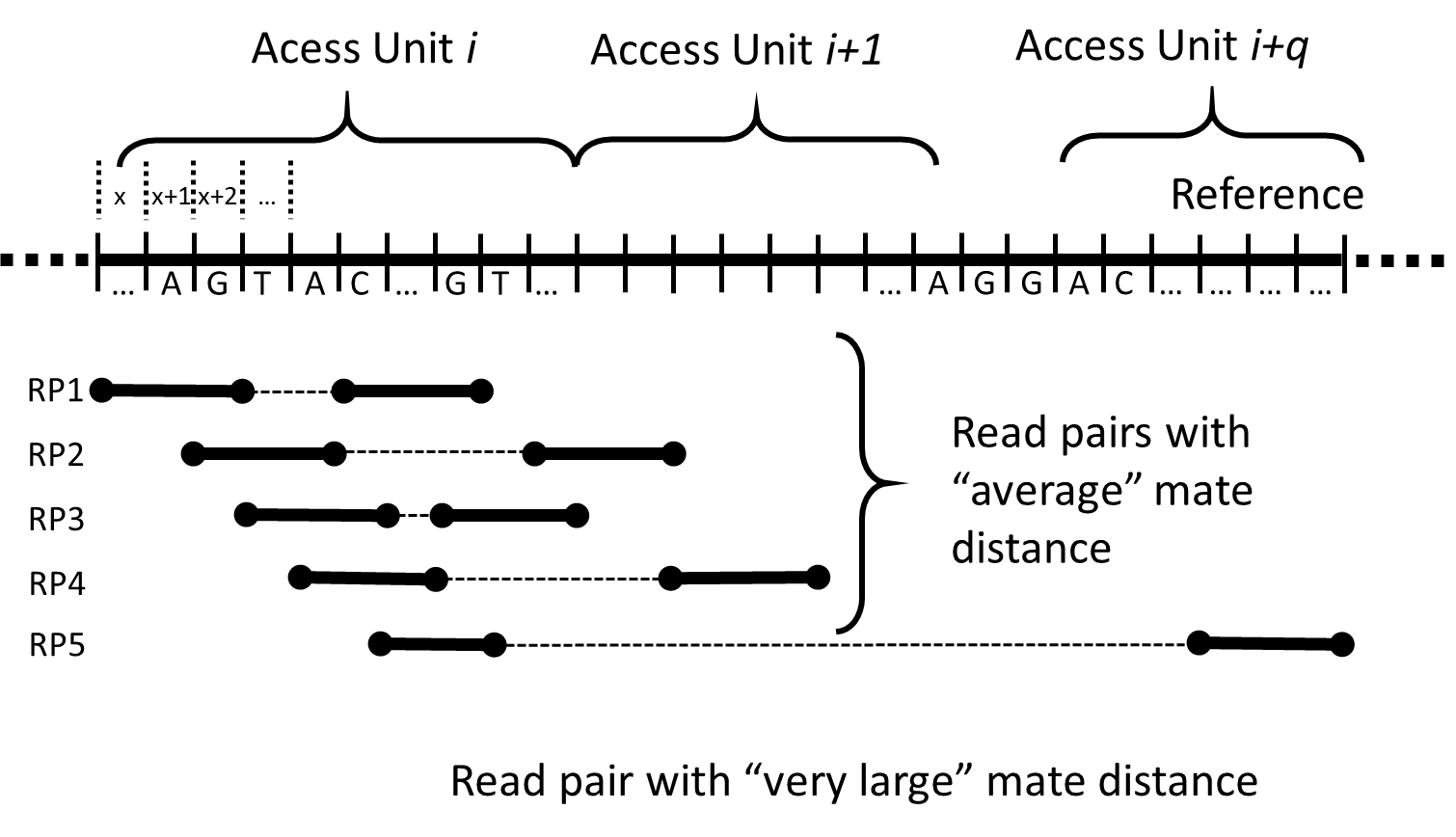
1. Read pairs are mapped near the edges of the Access Unit Covered Region.
2. The gap between a read and its mate is extremely large on the same reference.

An example is shown in **Figure 6**.

A proper handling of cases 1 and 2 requires the definition of a simple decoding behaviour when a genomic region (to be accessed) is specified, so that it covers only a portion of one or more encoded read pairs. This is due to the fact that, when encoding read pairs in a single data structure (the Genomic Record) the possibility exists of encoding either:

* part of a read outside of the Access Unit Range (this is also true in case of single reads, but the problem is amplified when dealing with paired-end reads), or
* a complete read in an AU with Range not covering the read left-most mapping position, or
* the combination of the two above.

This is shown in **Figure 6** for different values of the mate distance and read mapping positions.



**Figure 6 – Reads pairs encoded in Access Units**

Supposing, for sake of clarity, that the region to be accessed is coincident with AUi, the figure shows that read pairs RP1 and RP3 are mapped to a region entirely covered by AUi, while part of the nucleotides in pairs RP2 and RP4 (a complete read in the case of RP4) map to positions covered by AUi+1, and then outside the specified region; RP5 has a mate that is far away from AUi.

## Selective access and block based encoding

Given the definition of Genomic Record Length (GRL) provided in Clause 3, a parameter *T* is defined as the GRL threshold above which a read pair must be encoded in two separate Genomic Records. This will prevent that reads that are mapped far away from their mate are encoded in an AU that is unrelated[[1]](#footnote-1) to their mapping position (and therefore they would not be retrieved in case of access to their mapping region).

The parameter T is encoded, per reference sequence, in the Dataset Header, as specified in clause 6.4.2.1, as Thres[i], where i is the identifier of the reference sequence.

In the context of random access this threshold indicates the maximum difference between the Access Unit Covered Region and the Access Unit Range.

## Overlapping AUs Ranges

This clause describes how to perform selective access to coded genomic data mapped on a genomic region defined by a start position ***S*** and an end position ***E***, when Access Units have overlapping Ranges.

The Master Index Table, defined in clause 6.5.2.1, shall be parsed first from the beginning, in order to test the value of the AU start positions *si* and end positions *ei* and decode all AUs having

* *si* ≤***S***AND *ei* ≥***E***

OR

* ***S*** ≤*si* ≤***E***

OR

* ***S*** ≤*ei* ≤***E***

In practice, being the Master Index Table ordered by increasing AU Start Position, the decoder can stop parsing when *si* > **E**.

## Non-overlapping AUs Ranges

When performing selective access to a genomic region defined by a start position ***S*** and an end position ***E***, the decoder, in order to produce the requested output, shall decode all Access Unit AUi for which:

1. ***S***≤ *si* ≤***E***

OR

1. ***S***≤ *ei* ≤***E***

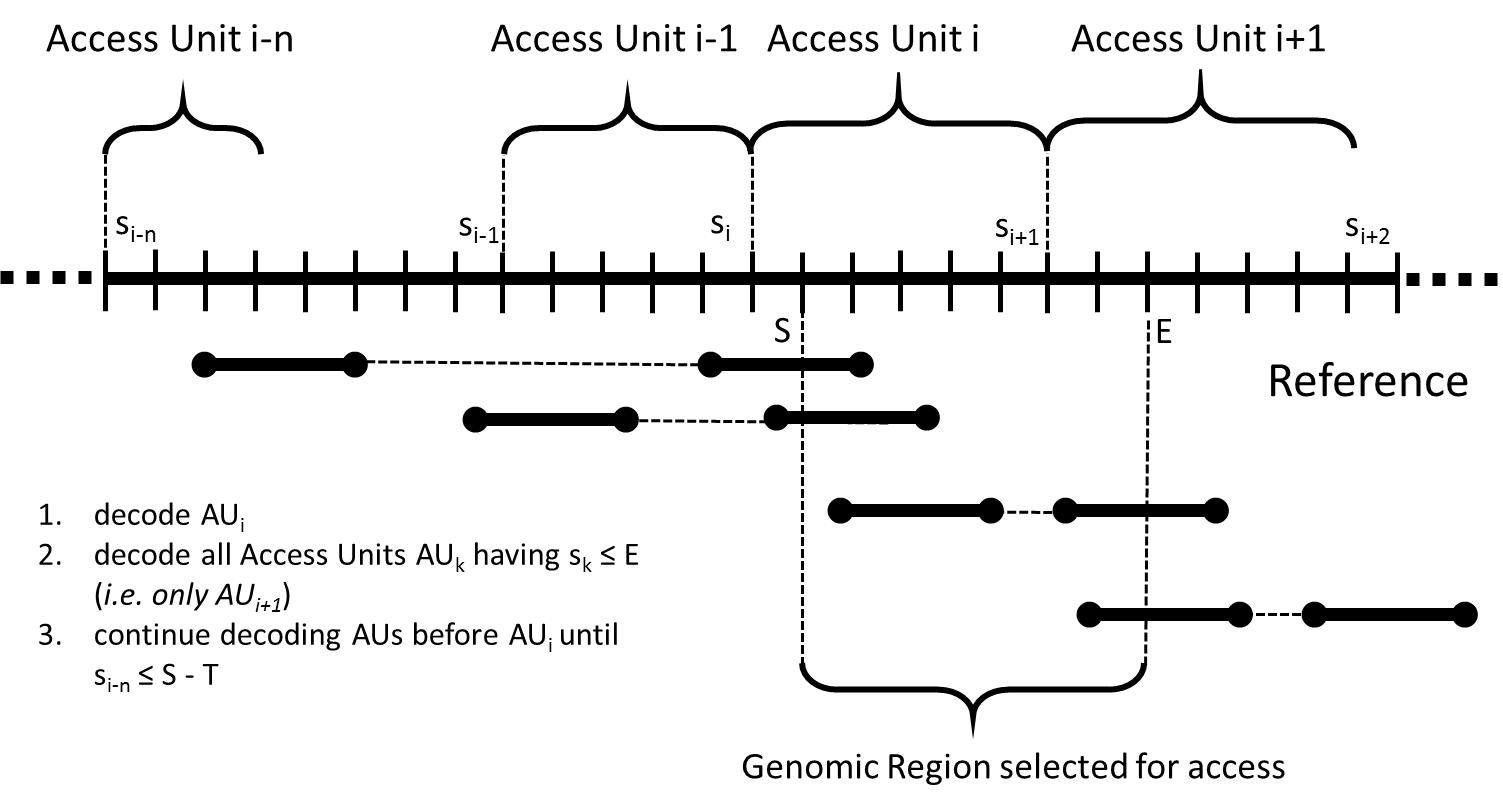
OR

1. *si*≤ ***S*** AND *ei* ≥***E***

where *si* denotes the ith AU\_start\_position (indexed in the Master Index Table, as specified in clause 6.5.2.1) and *ei* denotes the AU\_end\_position of the ith AU (indexed in the Master Index Table, as specified in clause 6.5.2.1).

Given the Master Index Table ordering, in this scenario the decoder shall:

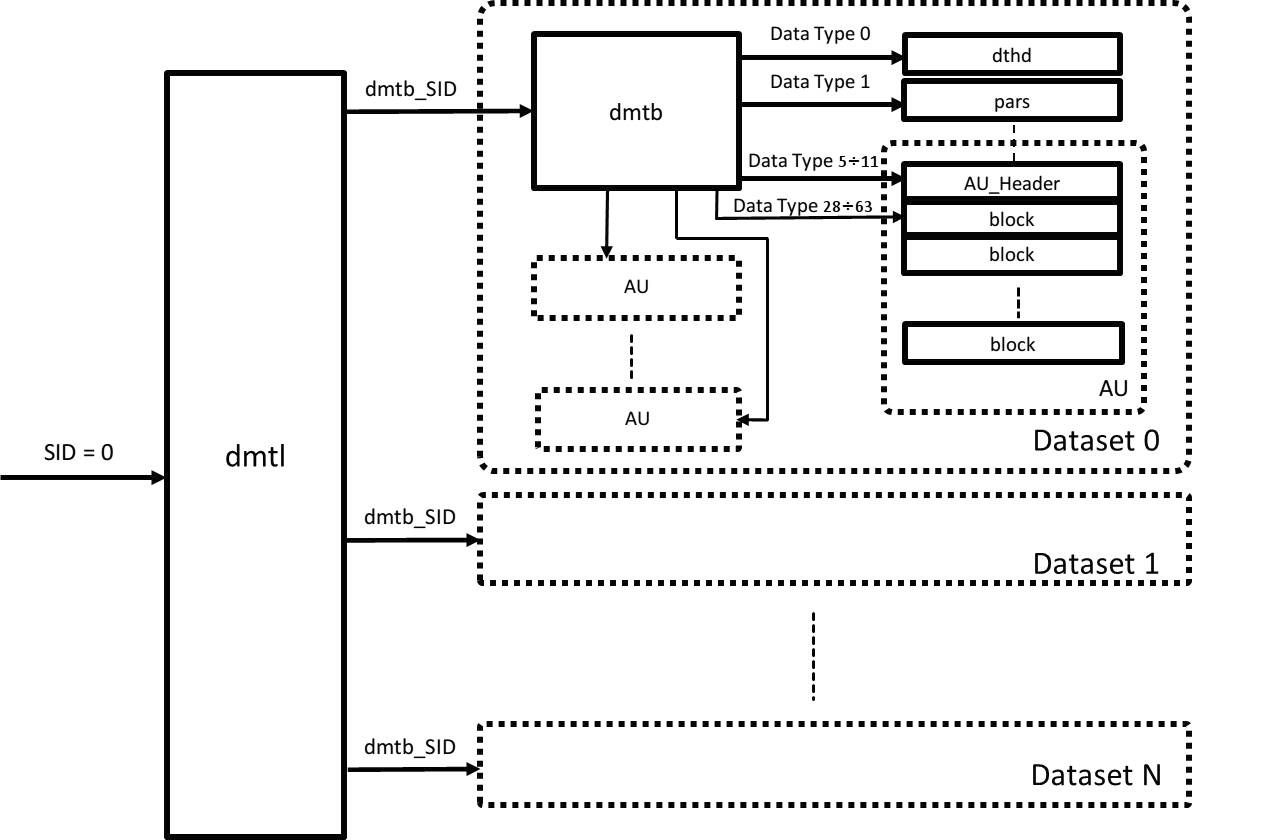
1. search for the AU having the largest *si* ≤ ***S***
2. if *ei* ≥***E*** jump to d else
3. search for all AUs having *si* ≤ ***E***
4. search for additional AUs having start position *si* < ***S*** satisfying 2.b or 2.c. The defined threshold ***T*** shall be used to stop the search for more AUs to be decoded when *si*< ***S*** – ***T***



**Figure 7 – Example of decoding process for selective access to a region comprised between positions S and E**

# Depacketization process (informative)

**Figure 8** describes as an example the typical flow of the depacketization process of a Stream composed by N Datasets and formatted according to the Transport Format defined in clause 6.5.



**Figure 8 – Depacketization process flow diagram**

A typical depacketization process, starting either from the first packet or from any other synchronization point, is as follows:

* The assumption in this clause is that the field block\_start\_code\_prefix\_flag, defined in clause 6.4.2.1, is equal to 1, as the field block\_start\_code\_prefix, defined in clause 6.4.4.1, is useful in order to unambiguously detect a packet containing a Block Header.
* The process starts when the first packet with SID equal to 0 is received: it contains a Dataset Mapping Table List, where the SIDs of all Dataset Mapping Tables are listed.
* The second step is performed when a packet with SID equal to one of the SIDs listed in the Dataset Mapping Table List is received: it contains a Dataset Mapping Table, where the mapping between different Data Types and SIDs is provided.
* The next valid packet should typically have SID equal to the entry in the Dataset Mapping Table, associated to the Data Type of dataset\_header, according to **Table 32**, hence it contains a Dataset Header.
* The next valid packet should typically have SID equal to the entry in the Dataset Mapping Table, associated to the Data Type of Dataset Parameter Set, according to **Table 32**, hence it contains a Dataset Parameter Set.
* The next valid packet should typically have SID equal to the entry in the Dataset Mapping Table, associated to the Data Type of access\_unit\_header of an Access Unit of a certain type, according to **Table 32**, hence it contains an Access Unit Header.
* The next valid packet should typically have SID equal to the entry in the Dataset Mapping Table, associated to the Data Type of a Block with a certain descriptor\_ID, according to **Table 32**. Being the first packet of this Block (this can be double-checked by checking the block\_start\_code\_prefix field is present, as the first packet might have been either received out-of-order or lost), it contains the Block Header and in particular the block\_payload\_size field.
* A set of packets shall typically follow, all having SID equal to the entry in the dmtb, associated to the Data Type of a Block with the same descriptor\_ID as above, according to **Table 32**, where the first three bytes are different than the block\_start\_code\_prefix field, hence these packets contain data from the same Block as above. The Block is reconstructed by appending the payloads of each of these packets, and a variable named Block\_Payload\_Size is cumulated until it matches the block\_payload\_size field in the Block Header: once this condition is met, the Block is closed.
* As soon as a packet is received having SID equal to the entry in the Dataset Mapping Table, associated to the Data Type of a Block, and Access Unit Marker Bit set to 1, the Access Unit is closed.
* In the same way, packets carrying data composing other Access Units and Blocks are received and depacketized.
* Order of packets in the previous steps is not guaranteed; hence the decoder shall bufferize when needed as packet order is not the expected one.
* Every time the sequence\_number for a Data Stream shows a gap, an error condition is detected as either a packet loss, if the sequence\_number gap is not filled by the next packets within a certain range decided by the user application, or an out-of-order packet condition, if the sequence\_number gap is filled by the next packets within a certain range decided by the user application.
* Incoming packets carrying data of Access Unit Headers of a certain type and Blocks of a certain Class/Descriptor can be filtered based on selective access rules, using the SID and the Dataset Mapping Table.
* At any time the user application can decide to stop the reception and decapsulation of the incoming Stream and possibly dump the decapsulated data in a local storage device, with a format compliant to either the Transport Format defined in clause 6.6 or the File Format defined in clause 6.4.4.1.2; in the latter case a conversion based on the process described in clause 6.7.3.1.2 or similar is needed.

1. By unrelated it is intended that it is neither the block itself nor an adjacent one. [↑](#footnote-ref-1)