**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC1/SC29/WG11**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC 1/SC 29/WG 11 N17321**

**Gwangju, KR – January 2018**

|  |  |
| --- | --- |
| **Source** | **ISO/IEC JTC 1/SC 29/WG 11** |
| **Status** | **Approved** |
| **Title** | **Study of CD ISO/IEC 23092-2 Coding of Genomic Information** |

**ISO 23092-2:####(X)**

ISO TC ###/SC 29/WG 11

Secretariat: XXXX

Information Technology — ISO/IEC 23092 — Part 2: Coding 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 2018, Published in Switzerland

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

Ch. de Blandonnet 8 • CP 401

CH-1214 Vernier, Geneva, Switzerland

Tel.  + 41 22 749 01 11

Fax  + 41 22 749 09 47

copyright@iso.org

www.iso.org

**Contents**

[Foreword vii](#_Toc506144007)

[Introduction viii](#_Toc506144008)

[1 Scope 1](#_Toc506144009)

[2 Normative references 1](#_Toc506144010)

[3 Terms and definitions 1](#_Toc506144011)

[4 Abbreviations 6](#_Toc506144012)

[5 Conventions 6](#_Toc506144013)

[5.1 General 6](#_Toc506144014)

[5.2 Arithmetic Operators 6](#_Toc506144015)

[5.3 Logical Operators 6](#_Toc506144016)

[5.4 Relational Operators 7](#_Toc506144017)

[5.5 Bit-wise Operators 7](#_Toc506144018)

[5.6 Assignment Operators 7](#_Toc506144019)

[5.7 Strings Operators 8](#_Toc506144020)

[5.8 Range notation 8](#_Toc506144021)

[5.9 Mathematical functions 8](#_Toc506144022)

[5.10 Order of operation precedence 9](#_Toc506144023)

[5.11 Variables, syntax elements and tables 10](#_Toc506144024)

[5.12 Text description of logical operators 11](#_Toc506144025)

[5.13 Processes 12](#_Toc506144026)

[6 Syntax and Semantics 12](#_Toc506144027)

[6.1 Method of specifying syntax in tabular form 12](#_Toc506144028)

[6.2 Bit ordering 14](#_Toc506144029)

[6.3 Specification of syntax functions and data types 14](#_Toc506144030)

[6.4 Semantics 15](#_Toc506144031)

[7 MPEG-G coded data structures 16](#_Toc506144032)

[7.1 Data Unit syntax and semantics 16](#_Toc506144033)

[7.2 Raw Reference 17](#_Toc506144034)

[7.2.1 Decoder input 17](#_Toc506144035)

[7.2.2 Decoder output 17](#_Toc506144036)

[7.2.3 Syntax and semantics 17](#_Toc506144037)

[7.3 Parameter Set 18](#_Toc506144038)

[7.4 Access Unit 19](#_Toc506144039)

[7.5 Access Unit Header 19](#_Toc506144040)

[7.6 Block 21](#_Toc506144041)

[7.7 Block Header 21](#_Toc506144042)

[7.8 Block Payload syntax and semantics 22](#_Toc506144043)

[8 Encoding Parameters 23](#_Toc506144044)

[8.1 Quality Values Parameter Set syntax and semantics 26](#_Toc506144045)

[8.2 Computed Reference Parameter Set 26](#_Toc506144046)

[9 Access Units 28](#_Toc506144047)

[9.1 Types of Access Units 28](#_Toc506144048)

[9.2 Descriptors in Access Units 29](#_Toc506144049)

[10 Sequence Reads 31](#_Toc506144050)

[10.1 Supported symbols 31](#_Toc506144051)

[10.2 Paired reads 31](#_Toc506144052)

[10.3 Data Classes 32](#_Toc506144053)

[10.4 Aligned Data 32](#_Toc506144054)

[10.5 Raw Data 33](#_Toc506144055)

[11 Decoding process 35](#_Toc506144056)

[11.1 General 35](#_Toc506144057)

[11.2 Data Units 35](#_Toc506144058)

[11.3 Access Units 35](#_Toc506144059)

[11.3.1 References Padding 35](#_Toc506144060)

[11.3.2 Type 1 (Class P) 36](#_Toc506144061)

[11.3.3 Type 2 (Class N) 38](#_Toc506144062)

[11.3.4 Type 3 (Class M) 38](#_Toc506144063)

[11.3.5 Type 4 (Class I) 38](#_Toc506144064)

[11.3.6 Type 5 (Class HM) 39](#_Toc506144065)

[11.3.7 Type 6 (Class U) 39](#_Toc506144066)

[11.3.8 Reference sequences 40](#_Toc506144067)

[11.3.9 Quality Values 40](#_Toc506144068)

[11.3.10 Read identifiers decoding process 41](#_Toc506144069)

[11.4 Genomic descriptors 49](#_Toc506144070)

[11.4.1 pos 50](#_Toc506144071)

[11.4.2 rcomp 51](#_Toc506144072)

[11.4.3 flags 51](#_Toc506144073)

[11.4.4 mmpos 52](#_Toc506144074)

[11.4.5 mmtype 53](#_Toc506144075)

[11.4.6 clips 56](#_Toc506144076)

[11.4.7 ureads 57](#_Toc506144077)

[11.4.8 rlen 57](#_Toc506144078)

[11.4.9 pair 58](#_Toc506144079)

[11.4.10 mscore 61](#_Toc506144080)

[11.4.11 Descriptors for multiple alignments 61](#_Toc506144081)

[11.4.12 rtype 68](#_Toc506144082)

[11.4.13 rgroup 69](#_Toc506144083)

[11.4.14 QVIndex 70](#_Toc506144084)

[11.4.15 QvCodebookIdentifier 70](#_Toc506144085)

[11.4.16 token\_type 70](#_Toc506144086)

[11.4.17 rftp 71](#_Toc506144087)

[11.4.18 rftt 72](#_Toc506144088)

[11.5 sequence 72](#_Toc506144089)

[11.5.1 Aligned reads (Classes P, N, M, I, HM) 72](#_Toc506144090)

[11.5.2 Unmapped reads (Class HM, U) 73](#_Toc506144091)

[11.6 e-cigar 73](#_Toc506144092)

[12 Coded representation of reference sequences 76](#_Toc506144093)

[12.1 External Reference 76](#_Toc506144094)

[12.2 Embedded Reference 76](#_Toc506144095)

[12.3 Computed Reference 76](#_Toc506144096)

[12.3.1 Reference computation algorithms 76](#_Toc506144097)

[13 Parsing process 80](#_Toc506144098)

[13.1 General 80](#_Toc506144099)

[13.2 CABAC parsing process 80](#_Toc506144100)

[13.2.1 Binarizations 80](#_Toc506144101)

[13.2.2 Decoder Configuration 88](#_Toc506144102)

[13.2.3 Decoder configuration for read identifiers 95](#_Toc506144103)

[13.2.4 Initialization process for context variables 96](#_Toc506144104)

[13.2.5 Arithmetic decoding engine 97](#_Toc506144105)

[13.2.6 Decoding process for sequence descriptors 105](#_Toc506144106)

[14 MPEG-G output format 114](#_Toc506144107)

[14.1 global\_ID 115](#_Toc506144108)

[14.2 read\_name 116](#_Toc506144109)

[14.3 class\_type 116](#_Toc506144110)

[14.4 seq\_ID 116](#_Toc506144111)

[14.5 read\_group 116](#_Toc506144112)

[14.6 read 1\_first 116](#_Toc506144113)

[14.7 flags 116](#_Toc506144114)

[14.8 number\_of\_template\_segments 116](#_Toc506144115)

[14.9 number\_of\_record\_segments 116](#_Toc506144116)

[14.10 read\_len 117](#_Toc506144117)

[14.11 quality\_values 117](#_Toc506144118)

[14.12 sequence 117](#_Toc506144119)

[14.13 number\_of\_alignments 117](#_Toc506144120)

[14.14 mapping\_pos 117](#_Toc506144121)

[14.15 split\_alignment 117](#_Toc506144122)

[14.16 split\_pos 117](#_Toc506144123)

[14.17 split\_seq\_ID 118](#_Toc506144124)

[14.18 delta 118](#_Toc506144125)

[14.19 ecigar\_len 118](#_Toc506144126)

[14.20 ecigar\_string 118](#_Toc506144127)

[14.21 reverse\_comp 119](#_Toc506144128)

[14.22 more\_alignments 119](#_Toc506144129)

[14.23 next\_pos 119](#_Toc506144130)

[14.24 next\_seq\_ID 120](#_Toc506144131)

[15 Annex A. Tokenization of Reads Identifiers (informative) 121](#_Toc506144132)

[16 Annex B. Examples of decoding processes (informative) 122](#_Toc506144133)

[16.1 clips descriptor 122](#_Toc506144134)

[16.2 pair descriptor 123](#_Toc506144135)

[17 Annex C. Mapping quality (informative) 124](#_Toc506144136)

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 2 (see [www.iso.org/directives](https://www.iso.org/directives-and-policies.html)).

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 (see [www.iso.org/patents](https://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 voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: [www.iso.org/iso/foreword.html](https://www.iso.org/iso/foreword.html).

This document was prepared by ISO/JTC1, Subcommittee SC29, Working Group 11.

This is the first edition of ISO/IEC 23092 Part 2. 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 has been 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 employed by this specification is designed to provide a representation of the genomic information and the related entropy coding that together enable to achieve significant advances over the state-of-the-art in terms of:

* format definition and coherent data structures for sequencing and alignment information, removing ambiguities present in legacy formats
* 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 standardization process
* speed and flexibility in the selective access to coded data, by means of newly designed data clustering and optimized 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, by providing 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 MPEG-G data representation is the *Genomic Record*. The Genomic Record is a data structure consisting of either a single sequence read, or a paired sequence read, and its associated sequencing and alignment information; it may contain detailed mapping and alignment data, a single or paired read identifier (read name) and quality values.

Without breaking traditional approaches, the Genomic Record introduced in the MPEG-G standard provides a more compact, simpler and manageable data structure grouping all the information related to a single DNA template from simple sequencing data to sophisticated alignment information.

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

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

The Access Unit is the smallest data structure that can be decoded by a decoder compliant with Part 2 of this Specification. An Access Unit is composed of one block for each descriptor used to represent the information of its Genomic Records; therefore, a Block payload is the coded representation of all the data of the same type (i.e. a descriptor) in a Cluster.

In addition to clusters of Records compressed into Access Units, genomic sequence reads are further classified in data classes: five classes are defined according to the result of their alignment against one or more reference sequences; the last one contains either reads that could not be mapped or raw sequencing data.

according to the result of their mapping against one or more reference sequences, with the exception of unmapped reads which are assigned to a specific data class called Class U. The classification of sequence reads into classes enables to develop powerful selective data access. In fact Access Units inherit a specific data characterization (e.g. perfect matches in Class P, substitutions in Class M, indels in Class I, half-mapped reads in Class HM) from the Genomic Records composing them, and thus constitute a data structure capable of providing powerful filtering capability for the efficient support of of many different use cases.

Access Units are also the fundamental, finest grain data structure in terms of content protection and in terms of metadata association. In other words each Access Unit can be protected individually and independently and specific metadata can be associated to single Access Units. **Figure 1** shows how Access Units, Blocks and Genomic Records relate to each other in the MPEG-G data structure.

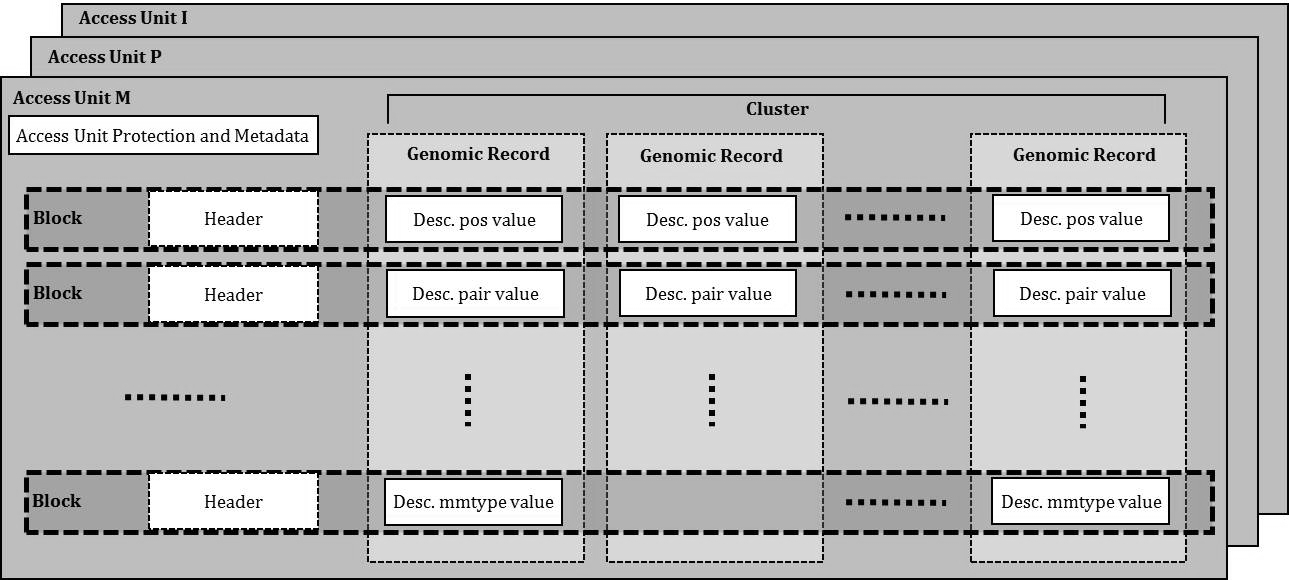


Figure 1 –Access Units, Blocks and Genomic Records

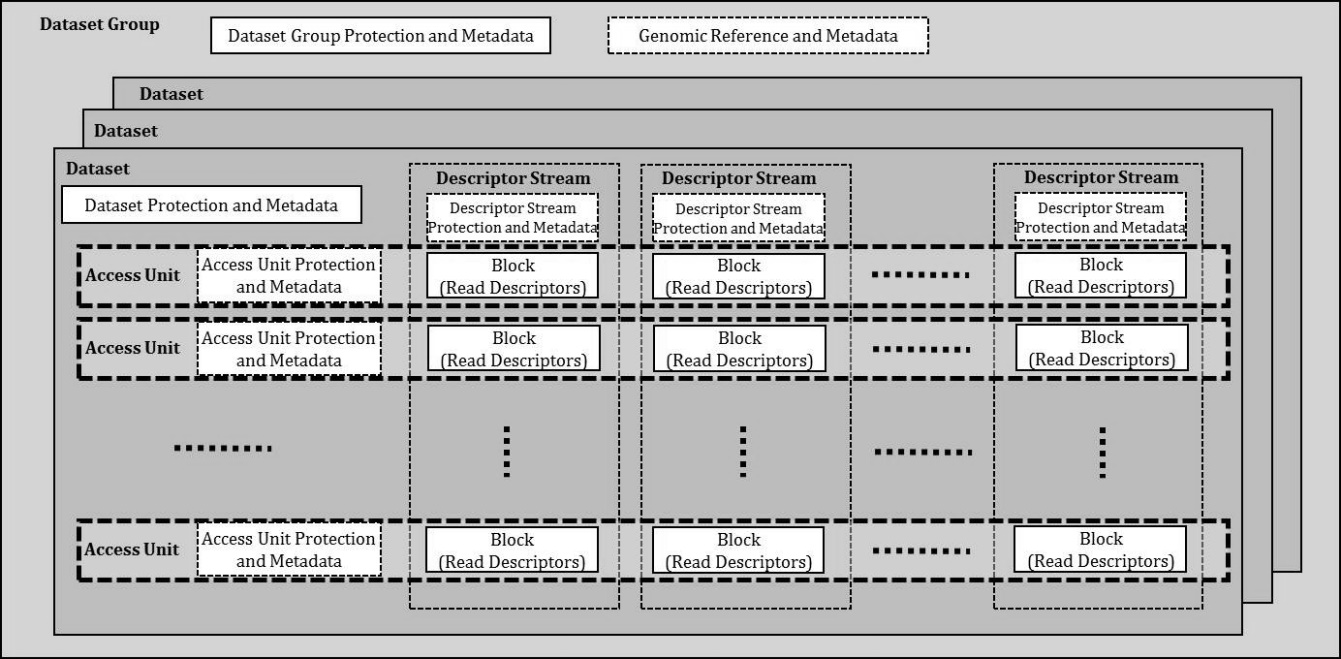


Figure 2 – The high level data structure: Datasets and Dataset Group

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

According to the MPEG-G standard, the compressed sequencing data can be multiplexed into a normative bitstream suitable to packetization for real-time transport over typical network protocols. In storage use cases coded data can be encapsulated into a standard MPEG-G file format with the possibility to organize Blocks per Descriptor Stream or per Access Units, to further optimize the selective access performance to the type of data access required by the different application scenarios. This Specification further provides a reference process to convert a normative MPEG-G transport stream into a normative MPEG-G file format and vice versa.

This document defines not only the syntax and semantics of the compressed genome sequencing data representation, but also the deterministic decoding process that reconstructs any MPEG-G Dataset. The decoding process is fully specified such that all decoders that conform to this document will produce identical decoded output. The normative decoding process includes all hyerarchies of data structures from the multiplexed bitstreams included in MPEG-G files or the data streams in streaming scenarios, to the descriptors blocks and to the normative output. A simplified diagram of the decoding process is shown in **Figure 3**.

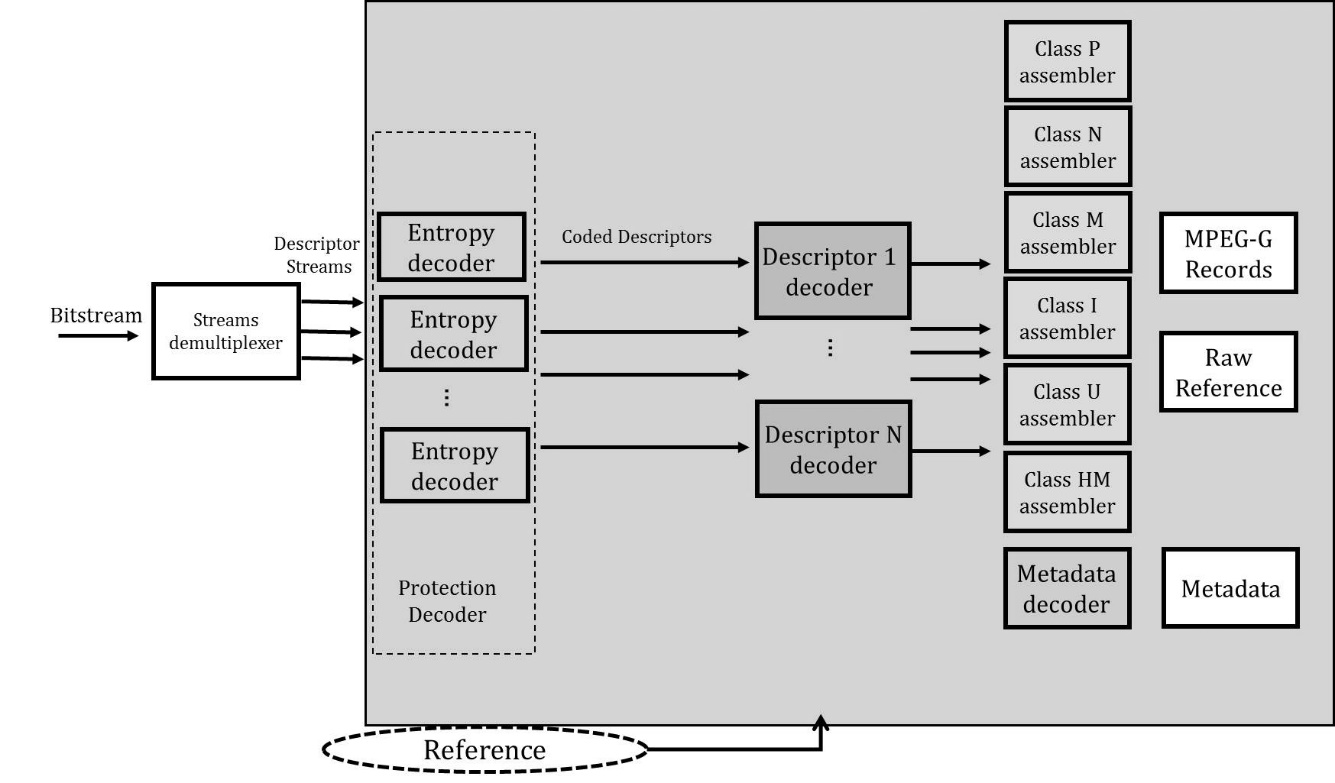


Figure 3 – The decoding process

Information Technology — ISO/IEC 23092 — Part 2: Coding of Genomic Information

# Scope

This document provides specifications for the normative representation of the following type of genomic data:

* Genomic sequence reads identifiers which are treated in Clause 11.3.10.
* Genomic sequence reads (both raw reads and aligned reads) which are treated in Clause 10.
* Reference sequences which are treated in Clause 12.
* Quality Values which are treated in Clause 11.3.9.

# Normative references

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

ISO/IEC 23092-1 Information technology -- Genomic Information Representation -- Part 1: Transport and Storage of Genomic Information

ISO/IEC 23092-3 Information technology -- Genomic Information Representation -- Part 3: APIs

ISO/IEC 23092-4 Information technology -- Genomic Information Representation -- Part 4: Reference SW

ISO/IEC 23092-5 Information technology -- Genomic Information Representation -- Part 5: Conformance

# Terms and definitions

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

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

— IEC Electropedia: available at <http://www.electropedia.org/>

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

Access Unit

Logical data structure containing a coded representation of genomic information to facilitate the bit stream access and manipulation. An Access Unit contains Genomic Records belonging to the same Data Class.

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 Access Unit Start Position and the Access Unit End Position inclusive.

alignment

A sequence read mapped on a reference sequence.

BAM

Compressed binary version of SAM.

base

In the context of this document “base” is used as synonymous of “nucleotide”.

base position

The number of bases between a base and the left-most mapped base belonging to the same Genomic Segment.

CIGAR string

A CIGAR string is a sequence of base lengths and the associated operations used to indicate bases that align (either a match/mismatch) with the reference, bases that are deleted from the reference, insertions that are not in the reference, clipped bases and splicing information.

**cluster**An aggregation of Genomic Records.

cluster signature

Sequence of nucleotides that is common to most or all Genomic Records belonging to a cluster.

contig

A contig (from contiguous) is a set of overlapping DNA segments that together represent a consensus region of DNA.

Dataset

A Dataset is a compression unit specified in ISO/IEC 23092-1 containing sequence reads and possibly alignment information.

deletion

Deletion of one or more nucleotide base pairs from a genomic sequence

E-CIGAR

Extended CIGAR syntax specified in this document as a superset of the CIGAR syntax. E-CIGAR enables the unambiguous representation of substitutions, spliced reads and splices strandedness.

edit operation

Modification of a sequence of nucleotides by means of a substitution, deletion or insertion.

FASTA

GIR that includes read headers and sequence reads (nucleotides sequences)

FASTQ

GIR that includes FASTA and Quality Values

first read

read 1

Sequence read generated by a paired-end sequencing run and stored in the first FASTQ file.

genomic descriptor

descriptor  
Element of the syntax used in this document to represent a feature of a genomic sequence read or associated information such as alignment information or Quality Values.

genomic range

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

genomic record

Data structure encoding either a single sequence read or a paired sequence read optionally associated with alignment information, read identifier and quality values.

genomic record length

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

genomic record position

Position of the left-most mapped base of the Genomic Record on the Reference Genome. A base present in the aligned read and not present in the Reference Sequence (a.k.a. insertion) and bases preserved by the alignment process but not mapped on the Reference Sequence (a.k.a. soft clips) do not have mapping positions. Positions are 0-based.

hard clips

Bases removed at either side of a read. Hard clips, or hard-clipped bases, are portions of an aligned sequence read that were ignored during the alignment process.

indel

An additional or missing nucleotide in a genomic sequence with respect to a reference genomic sequence (from insertion and deletion).

insertion

Addition of one or more nucleotide base pairs into a genomic sequence

MAF

Mutation Annotation Format. File format used to mark genes and other biological features in a DNA sequence.

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)

paired-ends

A couple of reads produced from the same (short) DNA fragment by sequencing both ends.

pileup

Textual representation of sequence reads aligned to a reference sequence.

quality value

quality score

A quality value is assigned to each nucleotide base call in automated sequencing processes. It expresses the base-call accuracy.

read identifier

read header  
read name

Each sequence read stored in the FASTA and FASTQ formats starts with a textual field called “read identifier”, “read header” or “read name” containing a sequence identifier and an optional description.

reference genome

A reference genome is a digital nucleic acid sequence database, assembled by scientists as a representative example of a species’ genetic material.

reference sequence

A reference sequence is a 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 to 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 it is said to be at position 0) and linearly increasing from left to right.

SAM

GIR that is human readable and includes FASTQ + alignment and analysis information.

second read

read 2

Sequence read generated by a paired-end sequencing run and stored in the second FASTQ file.

segment

genomic segment

A contiguous sequence of nucleotides

sequence read

The 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.

soft clips

Soft clipped bases are portions of an aligned sequence read which do not match well to the reference genome on either side of the read and are therefore ignored for the alignment, but still kept in the aligned read.

spliced read

Aligned read which cover non-continuous portions of the reference genome.

template

A genomic sequence that is sequenced on a sequencing machine.

# Abbreviations

AU Access Unit

GIR Genomic Information Representation

RBSP Raw Byte Sequence Payload

# Conventions

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

## General

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

## Arithmetic Operators

The following arithmetic operators are defined as follows:

+ Addition

− Subtraction (as a two-argument operator) or negation (as a unary prefix operator)

\* Multiplication, including matrix multiplication

xy Exponentiation. Specifies x to the power of y. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation.

/ Integer division with truncation of the result toward zero. For example, 7 / 4 and −7 / −4 are truncated to 1 and −7 / 4 and 7 / −4 are truncated to −1.

÷ Used to denote division in mathematical equations where no truncation or rounding is intended.

 Used to denote division in mathematical equations where no truncation or rounding is intended.

 The summation of f( i ) with i taking all integer values from x up to and including y.

x % y Modulus. Remainder of x divided by y, defined only for integers x and y with x >= 0 and y > 0.

## Logical Operators

The following logical operators are defined as follows:

x && y Boolean logical "and" of x and y

x || y Boolean logical "or" of x and y

! Boolean logical "not"

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

## Relational Operators

The following relational operators are defined as follows:

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

== Equal to

!= Not equal to

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

## Bit-wise Operators

The following bit-wise operators are defined as follows:

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

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

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

x >> y Arithmetic right shift of a two's complement integer representation of x by y binary digits. This function is defined only for non-negative integer values of y. Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation.

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

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

## Assignment Operators

The following arithmetic operators are defined as follows:

= Assignment operator

++ Increment, i.e., *x++* is equivalent to *x =* *x* + 1; when used in an array index, evaluates to the value of the variable prior to the increment operation.

− − Decrement, i.e., x− − is equivalent to x = x − 1; when used in an array index, evaluates to the value of the variable prior to the decrement operation.

+= Increment by amount specified, i.e., x += 3 is equivalent to x = x + 3, and x += (−3) is equivalent to x = x + (−3).

−= Decrement by amount specified, i.e., x −= 3 is equivalent to x = x − 3, and x −= (−3) is equivalent to x = x − (−3).

## Strings Operators

The following strings operators are defined as follows:

tostr(i) returns the string representation of the integer i

strcat(s1, s2) returns the concatenatenation of strings s1 and s2

## Range notation

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

x = y..z x takes on integer values starting from y to z, inclusive, with x, y, and z being integer numbers and z being greater than y.

## Mathematical functions

The following mathematical functions are defined:

Abs( x ) = (5‑1)

Ceil( x ) the smallest integer greater than or equal to x (5‑2)

Clip1Y( x ) = Clip3( 0, ( 1 << BitDepthY ) − 1, x ) (5‑3)

Clip1C( x ) = Clip3( 0, ( 1 << BitDepthC ) − 1, x ) (5‑4)

Clip3( x, y, z ) =  (5‑5)

Cos( x ) the trigonometric cosine function operating on an argument x in units of radians (5‑6)

Floor( x ) the largest integer less than or equal to x (5‑7)

GetCurrMsb( cl, pl, pm, ml ) =  (5‑8)

Ln( x ) the natural logarithm of x (the base-e logarithm, where e is natural logarithm base constant 2.718 281 828...) (5‑9)

Log2( x ) the base-2 logarithm of x (5‑10)

Log10( x ) the base-10 logarithm of x (5‑11)

Min( x, y ) =  (5‑12)

Max( x, y ) =  (5‑13)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (5‑14)

Sign( x ) =  (5‑15)

Sqrt( x ) =  (5‑16)

Swap( x, y ) = ( y, x ) (5‑17)

## Order of operation precedence

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

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

Table 1 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

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

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

|  |
| --- |
| operations (with operands x, y, and z) |
| "x++", "x− −" |
| "!x", "−x" (as a unary prefix operator) |
| xy |
| "x \* y", "x / y", "x  y""", "x % y" |
| "x + y", "x − y" (as a two-argument operator), "" |
| "x << y", "x >> y" |
| "x < y", "x <= y", "x > y", "x >= y" |
| "x = = y", "x != y" |
| "x & y" |
| "x | y" |
| "x && y" |
| "x | | y" |
| "x ? y : z" |
| "x..y" |
| "x = y", "x += y", "x −= y" |

## Variables, syntax elements and tables

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

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

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

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in clauses 7 and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in clause 5.2) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

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

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

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

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

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

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

## Text description of logical operators

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

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

may be described in the following manner:

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

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

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

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

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

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

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

– condition 1b

– ...

– Otherwise, statement n

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

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

may be described in the following manner:

When condition 0, statement 0

When condition 1, statement 1

## Processes

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

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

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

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

# Syntax and Semantics

## Method of specifying syntax in tabular form

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

The following table lists examples of the syntax specification format. When **syntax\_element** appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| /\* A statement can be a syntax element with an associated data type or can be an expression used to specify conditions for the existence, type and quantity of syntax elements, as in the following two examples \*/ |  |
| **syntax\_element** | ue(v) |
| conditioning statement |  |
|  |  |
| /\*A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. \*/ |  |
| { |  |
| statement |  |
| statement |  |
| ... |  |
| } |  |
| /\* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true \*/ |  |
| while( condition ) |  |
| statement |  |
|  |  |
| /\* A "do ... while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true \*/ |  |
| do |  |
| statement |  |
| while( condition ) |  |
|  |  |
| /\* An "if ... else" structure specifies a test of whether a condition is true and, if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed \*/ |  |
| if( condition ) |  |
| primary statement |  |
| else |  |
| alternative statement |  |
|  |  |
| /\* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. \*/ |  |
| for( initial statement; condition; subsequent statement ) |  |
| primary statement |  |

## 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 data types

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

byte\_aligned( ) is specified as follows:

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

more\_data\_in\_block\_payload( ) is specified as follows:

* If byte\_aligned( ) is equal to TRUE and the current position in the block\_payload syntax structure is block\_size \* 8 bits minus the padding bits from the beginning of the block\_payload syntax structure, the return value of more\_data\_in\_block\_payload( ) is equal to FALSE.
* Otherwise, the return value of more\_data\_in\_block\_payload( ) is equal to TRUE.

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

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

* ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this data type is specified in clause13.
* b(8): byte having any pattern of bit string (8 bits). The parsing process for this data type is specified by the return value of the function read\_bits( 8 ).
* f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this data type is specified by the return value of the function read\_bits( n ).
* i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function read\_bits( n ) interpreted as a two's complement integer representation with most significant bit written first.
* se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this data type is specified in clause 13.2.1.3.
* 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.

NOTE – The st(v) syntax data type is only used in this Specification when the current position in the bitstream is a byte-aligned position.

* u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function read\_bits( n ) interpreted as a binary representation of an unsigned integer with most significant bit written first.
* ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this data type is specified in clause 13.2.1.3.
* u7(v): variable sized unsigned integer computed by iteratively reading 8 bits, where the least significant 7 bits are interpreted as a binary representation of an unsigned integer *v*, with the most significant bit written first, and the 8th bit signaling if the iteration should stop. The parsing process for this data type is specified below:

s =0

v = 0

do {

c = read\_bits( 8 );

v |= (c & 0x7f) << s;

s += 7;

} while (c & 0x80)

* c(n): sequence of n universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in ISO/IEC 10646.

## Semantics

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

# MPEG-G coded data structures

Table 2 in clause 7.1 specifies the syntax for a Data Unit: a Data Unit can contain a Raw Reference structure, a Parameters Set structure or an Access Unit structure.

Table 4 in clause 7.2.3 specifies the syntax for a Raw Reference structure: a Raw Reference contains reference data as strings of characters representing the reference bases.

Table 5 in clause 7.3 specifies the syntax for a Parameters Set structure containing a Parent Parameters Set identifier, a Parameter Set identifier and Encoding Parameters as specified in Clause 8 of this document.

Table 6 in clause 7.4 specifies the syntax for an Access Unit. An Access Unit structure contains an Access Unit Header prefix, followed by one or more Block structures.

Table 7 in clause 7.5 specifies the syntax for an Access Unit Header.

Table 8 in clause 7.6 specifies the syntax for a Block structure.

Each Block structure which follows the Access Unit Header contains a Block Header prefix, specified in Table 9 in clause 7.7, followed by a Block Payload specified in clause 7.8.

A Block Payload contains encoded genomic descriptors of a single type which is identified by a descriptor\_ID.

## Data Unit syntax and semantics

Table 2. Data Unit syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| data\_unit() { |  |
| **data\_unit\_type** | u(8) |
| if (data\_unit\_type == 0) { |  |
| **data\_unit\_size** | u(64) |
| raw\_reference() | Raw Reference |
| } |  |
| else if (data\_unit\_type == 1) { |  |
| **data\_unit\_size** | u(32) |
| parameter\_set() | Parameter Set |
| } |  |
| else if(data\_unit\_type == 2){ |  |
| **data\_unit\_size** | u(29) |
| **reserved** | u(3) |
| access\_unit() | Access Unit |
| } |  |
| else /\*(data\_unit\_type > 2)\*/{ |  |
| /\*skip data unit\*/ |  |
| } |  |
| } |  |

**data\_unit\_type** specifies the type of Data Unit.

Table 3: Values of data\_unit\_type and associated data unit types.

|  |  |  |
| --- | --- | --- |
| **Data\_unit\_type** | **Data unit type** | **Clause** |
| 0 | Raw Reference | 7.2 |
| 1 | Parameter Set | 7.3 |
| 2 | Access Unit | 7.4 |

**data\_unit\_size** is the total size in bytes of the Data Unit including the 5 bytes used for **data\_unit\_type** and **data\_unit\_size**.

**raw\_reference** is a raw­\_reference structure as specified in clause 7.2.

**parameter\_set** is a parameter\_set structure as specified in clause 7.3.

**access\_unit** is an access\_unit structure as specified in clause 7.4.

A conformant bitstream containing at least one Data Unit of type Access Unit shall contain at least one Data Unit of type Parameter Set.

A conformant bitstream may contain only Data Units of type Raw Reference.

## Raw Reference

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

* deliver any reference sequence to the decoder,
* return decoded reference sequences or part thereof from the decoder.

### Decoder input

If a raw reference is needed to decode content, such raw reference shall be made available to the decoder prior to any other Data Unit. If the corresponding Data Unit of type 0 (**data\_unit\_type** equal to 0, raw\_reference) is not present in the byte stream with the content to decode, the application invoking the decoder (e.g. test application or tool compliant to ISO/IEC 23092-1) shall take care to identify the requested byte stream containing the reference and feed it to the decoder first. Decoding of content that needs a raw reference is unspecified when the raw reference is not available.

### Decoder output

The raw\_reference structure shall be used by the decoder to make available for further processing - as described in clause 7.2.1 - a Genomic Reference Sequence or part thereof encoded in an Access Unit. The decoder shall select the appropriate output to be generated (i.e. MPEG-G Record or Raw Reference) according to the value of the **dataset\_type** encoding parameter as specified in clause 8.

### Syntax and semantics

Table 4. Raw Reference syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| raw\_reference() { |  |
| **reference\_ID** | u(8) |
| **seq\_count** | u(16) |
| for (seqId=0; seqId<seq\_count; seqId++) { |  |
| **sequence\_ID** | u(16) |
| **seq\_start** | u(2^max\_bits\_pos) |
| **seq\_end** | u(2^max\_bits\_pos ) |
| for (i=0; i<=seq\_end – seq\_start; i++) { |  |
| **sequence**[seqId][i] | u(8) |
| } |  |
| } |  |
| } |  |

**reference\_ID** is a unique identifier of the Raw Reference.

**seq\_count** is the number of sequences in the Raw Reference.

**sequence\_ID** is a unique identifier of the Reference Sequence.

**seq\_start** is the coordinate of the first base present in the Sequence array

**seq\_end** is the coordinate of the last base present in the Sequence array

**sequence[seqId][i]** is the ASCII character representing the ith base in the Reference Sequence identified by **sequence\_ID**.

## Parameter Set

This clause specifies the Parameter Set syntax and semantics. A conformant bitstream containing at least one Data Unit of type Access Unit shall contain at least one Parameter Set.

Table 5. Parameters Set syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| parameter\_set() { |  |
| **parent\_parameter\_set\_ID** | u(12) |
| **parameter\_set\_ID** | u(12) |
| encoding\_parameters() |  |
| } |  |

**parent\_parameter\_set\_ID** is the unique identifier of an existing Parameter Set. Referencing an existing Parameter Set from another Parameter Set enables the generation of a hierarchy of Parameter Sets where the values of the encoding parameters of each node override the corresponding values of the parent node. If equal to parameter\_set\_ID, the Parameter Set is at the top level in the hierarchy.

**parameters\_set\_ID** is the unique identifier of the Parameter Set. All encoding parameters present in this Parameter Set override those present in the Parameter Set identified by parent\_parameter\_set\_ID.

**encoding\_parameters** are the encoding parameters as specified in clause 8 of this document.

## Access Unit

This clause specifies the Access Unit syntax and semantics. A conformant bitstream containing at least one Access Unit shall contain at least one Parameter Set.

Table 6. Access Unit syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| access\_unit() { |  |
| access\_unit\_header() | Access Unit Header |
| for (blockId=0; blockId<num\_blocks; blockId++) { |  |
| block[blockId]() | Block |
| } |  |
| } |  |

**access\_unit\_header** is specified in clause 7.5.

**num\_blocks** specifies the number of blocks encoded in the Access Unit and it is encoded in the access\_unit\_header as specified in clause 7.5.

**block[blockId]** is a Block structure as specified in clause 7.6.

## Access Unit Header

This clause specifies the Access Unit Header syntax and semantics.

Table 7. Access Unit Header syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| access\_unit\_header() { |  |
| **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 OR AU\_type == M\_TYPE\_AU ){ |  |
| **mm\_threshold** | u(16) |
| **mm\_count** | u(32) |
| } |  |
| **sequence\_ID** | u(16) |
| if (AU\_Type!= U\_TYPE\_AU ) |  |
| { |  |
| **AU\_start\_position** | u(2^max\_bits\_pos) |
| **AU\_end\_position** | u(2^max\_bits\_pos) |
| if (multiple\_alignments\_flag) { |  |
| **extended\_AU\_start\_position** | u(2^max\_bits\_pos) |
| **extended\_AU\_end\_position** | u(2^max\_bits\_pos) |
| } |  |
| } |  |
| else { |  |
| if (multiple\_signature\_base != 0) { |  |
| **U\_cluster\_signature**[0] | u(U\_signature\_size) |
| if (U\_cluster\_signature[0] != 0xFFFFFFFF) { |  |
| for (i=1; i<multiple\_signature\_base; i++) { |  |
| **U\_cluster\_signature**[i] | u(U\_signature\_size) |
| } |  |
| } |  |
| else { |  |
| **num\_signatures** | u(16) |
| for (i=0; i<num\_signatures; i++) { |  |
| **U\_cluster\_signature**[i] | u(U\_signature\_size) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

**access\_unit\_ID** is an unambiguous identifier, zero-based, 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.

**num\_blocks** specifies the number of Blocks in the Access Unit. It is equal to the number of different types of descriptors used 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 Class of data carried therein as specified in clause 9.1.

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

**mm\_threshold** specifies 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** specifies 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 a unique identifier of the Raw Reference to be used to decode the Access Unit as specified in clause 13.2.6.2.10. A value is reserved when no reference is used.

**sequence\_ID** is a unique identifier of the Reference Sequence to be used to decode the Access Unit as specified in clause 13.2.6.2.10. It corresponds to the **sequence\_ID** element in Table 4.

**AU\_start\_position** is the indexing “lower” information. For mapped AUs this is the position of the left-most mapped base encoded in the Access Unit.

**AU\_end\_position** is the indexing “higher” information. For mapped AUs this is the position of the right-most mapped base encoded in the Access Unit.

**extended\_ AU\_start\_position** specifies 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 10.3.

**extended\_AU\_end\_position** specifies 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 10.3.

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

**num\_signatures** specifies the number of signatures used to index unmapped reads.

**multiple\_alignments\_flag** and **multiple\_signature\_base** are specified in clause 7.3.

## Block

This clause specifies the Block syntax and semantics. A Block Payload shall be byte-aligned. Byte alignment is obtained by the use of 0-bits padding in the least significant positions of last byte of a Block Payload. The number of padding bits is encoded in the Block Header as specified in clause 7.7.

Table 8. Block syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| block() { |  |
| block\_header() | Block Header |
| block\_payload() | Block Payload |
| } |  |

**block\_header** is a Block Header structure as specified in clause 7.7.

**block\_payload** is a Block Payload structure as specified in clause 7.8.

## Block Header

This clause describes the Block Header syntax and semantics.

Table 9. Block Header syntax.

|  |  |
| --- | --- |
| Syntax | Type |
| block\_header() { |  |
| **descriptor\_ID** | u(7) |
| **reserved** | u(6) |
| **padding** | u(3) |
| **block\_size** | u(32) |
| } |  |

**descriptor\_ID** is the identifier of the type of encoded genomic descriptor carried by this Block as specified in Table 17.

**reserved** bits used to preserve byte alignment.

**padding** specifies the number of padding bits added at the end of the encoded payload in order to make it byte aligned.

**block\_size** specifies the size in bytes of the Block Payload.

## Block Payload syntax and semantics

This clause specifies the syntax and semantics of the Block Payload structure containing entropy-coded descriptors.

Table 10. Block Payload syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| block\_payload() { |  |
| if(descriptor\_ID < 16){ |  |
| encoded\_descriptor\_sequences(descriptor\_ID) | As specified in 13.2.6.2.1 |
| } |  |
| else if(descriptor\_ID == 16){ |  |
| encoded\_read\_identifiers() | As specified in 11.3.10.1 |
| } |  |
| while( !byte\_aligned( ) ) |  |
| **nesting\_zero\_bit** | f(1) |
| } |  |

encoded\_descriptor\_sequences(descriptor\_ID) is a data structure specified in clause 13.2.6.2.1 carrying the encoded genomic descriptors for sequences and Quality Values specified in clause 9.2.

encoded\_read\_identifiers() is a data structure specified in clause 11.3.10.1 carrying encoded read identifiers.

**nesting\_zero\_bit** is one bit set to 0.

# Encoding Parameters

The Encoding Parameters are configuration parameters used by the decoder during the decoding process. They are carried by the **encoding\_parameters** data structure as specifiedin this clause.

Table 11. Encoding Parameters syntax.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| encoding\_parameters() { |  |
| **dataset\_type** | u(4) |
| **alphabet\_ID** | u(8) |
| **reads\_length** | u(29) |
| **max\_au\_data\_unit\_size** | u(29) |
| **max\_bits\_pos** | u(8) |
| **qv\_depth** | u(3) |
| **as\_depth** | u(3) |
| **terminator\_size\_minus1** | u(2) |
| **num\_classes** | u(4) |
| for(j=0; j < num\_classes; j++) |  |
| **class\_ID[j]** | u(4) |
| for(desc\_ID=0; desc\_ID < **NUM\_DESCRIPTORS**; desc\_ID++){ |  |
| **encoding\_mode\_ID [**desc\_ID**]** | u(8) |
| **enc\_cfg\_flag[**desc\_ID**]** | u(1) |
| if(enc\_cfg\_flag[desc\_ID] == 1){ |  |
| if(desc\_ID < 16) |  |
| decoder\_configuration(encoding\_mode\_ID [desc\_ID]) | As specified in 13.2.2 |
| else if(desc\_ID == 16) |  |
| decoder\_configuration\_read\_IDs (encoding\_mode\_ID [desc\_ID]) | As specified in 13.2.3 |
| } |  |
| } |  |
| **num\_groups** | u(8) |
| for(j=0; j < num\_groups; j++) |  |
| **rgroup\_ID[j]** | st(v) |
| **multiple\_alignments\_flag** | u(1) |
| **spliced\_reads\_flag** | u(1) |
| **multiple\_signature\_flag** | u(1) |
| **multiple\_signature\_base** | u(31) |
| **U\_signature\_size** | u(8) |
| if(multiple\_alignments\_flag == 1){ |  |
| **mscore\_exponent** | u(4) |
| **mscore\_fractional** | u(6) |
| } |  |
| **qvps\_flag** | u(1) |
| if(qvps\_flag == 1) |  |
| parameter\_set\_qvps(num\_classes) |  |
| **crps\_flag** | u(1) |
| if(crps\_flag == 1) |  |
| parameter\_set\_crps(num\_classes) |  |
| while( !byte\_aligned( ) ) |  |
| **nesting\_zero\_bit** | f(1) |
| } |  |

**dataset\_type** specifies the type of data encoded in Access Units referring to this encoding parameters as specified in ISO/IEC 23092-1. The possible values are: 0 = non-aligned content; 1 = aligned content; 2 = reference.

**alphabet\_ID** identifies the alphabet of symbols used for data encoded in Access Units referring to this encoding parameters.

**reads\_length** specifies the length in nucleotides of sequence reads in case of constant reads length. The value 0 indicates the presence of variable reads length (conveyed per each Genomic Record by the **rlen** syntax element as specified in clause 11.4.8).

**max\_au\_data\_unit\_size** is the maximum value permitted to the field data\_unit\_size in the data unit header, when data\_unit\_type is equal to 2, as specified in clause 7.1. A value of 0 indicates an unspecified maximum data unit size.

**qv\_depth** specifies the number of Quality Values associated to each coded nucleotide. A value of 0 means that no Quality Values are encoded.

**as\_depth** specifies the number of alignment scores associated to each coded alignment. A value of 0 means that no alignment scores are encoded.

**terminator\_size\_minus1** specifies the size in bytes minus 1 of the terminator symbol used for the **mmpos** descriptor specified in clause 11.4.4. The terminator value shall be the largest unsigned integer representable with the number of bytes specified by **terminator\_size\_minus1** as shown in

|  |  |  |
| --- | --- | --- |
| **terminator\_size\_minus1 value** | **size in bytes** | **terminator value** |
| 0 | 1 | 0xff |
| 1 | 2 | 0xffff |
| 2 | 3 | 0xffffff |
| 3 | 4 | 0xffffffff |

Table 12. Terminator sizes and values.

**num\_classes** specifies the number of Data Classes encoded in all Access Units referring to the current Parameters Set.

**class\_ID** is one of the data class identifiers specified in clause 10.3. For any value of ci greater than 0 it shall always be class\_ID[ci] > class\_ID[ci - 1].

**NUM\_DESCRIPTORS** is a constant counting the number of genomic descriptors specified in this document and it is set to 17.

**encoding\_mode\_ID** identifies one of the coding modes specified in clause 8.

**enc\_cfg\_flag** is a flag signaling the presence of a decoder configuration.

**num\_groups** specifies the number of read groups present in all Access Units referring to the current Parameters Set. The **rgroup** descriptor specified in clause 11.4.13 shall have values between 0 and **num\_groups**. If **num\_groups** is set to 0, the **rgroup** descriptor shall not be present in the AUs referring to this Parameter Set.

**rgroup\_ID** is the null-terminated string identifier of a read group.

**multiple\_alignments\_flag** is a flag signaling the presence of multiple alignments in the Access Unit. When set to 0 no multiple alignments are present.

**spliced\_reads\_flag** is a flag signaling the presence of spliced reads in the Access Unit. When set to 0 no spliced reads are present.

**multiple\_signature\_flag** is a flag signaling the use of multiple signatures in an Access Unit of type 6 (Class U).

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

**U\_signature\_size** is the size in bits of each integer representing an encoded signature. Each signature can be represented by one or more integers in the Master Index Table.

**decoder\_configuration** is a data structure containing the decoder configuration parameters as specified in clause 13.2.2.

decoder\_configuration\_read\_IDs(encoding\_mode\_ID) data structure containing the decoder configuration parameters as specified in clause 13.2.3.

**mscore\_exponent** specifies the number of bits used to encode the exponent part of the multiple alignments score encoded in the **mscore** descriptor. As specified in IEEE RFC 754 [1] the value can go from 0 to 11. The mscore descriptor is specified in clause 11.4.10.

**mscore\_fractional** specifies the number of bits used to encode the fractional part of the multiple alignments score encoded in the **mscore** descriptor. As specified in IEEE RFC 754 [1] the value can go from 0 to 52. The mscore descriptor is specified in clause 11.4.10.

See clause 11.4.13 for more details on how multiple alignments scores are encoded.

**qvps\_flag** signals the presence of a **parameter\_set\_qvps** element

**parameter\_set\_qvps** is the Quality Values Parameter Set as specified in clause 8.1. If not present, the parent Quality Values Parameter Set identified by parent\_parameter\_set\_ID shall be used.

**crps\_flag** signals the presence of a **parameter\_set\_crps** element.

**parameter\_set\_crps** is the Computed Reference Parameter Set as specified in clause 8.2. If not present, the **parameter\_set\_crps** of the parent Parameter Set identified by parent\_parameter\_set\_ID shall be used.

**nesting\_zero\_bit** is one bit set to 0.

**desc\_ID** specifies the descriptor identifiers descriptor\_ID specified in Table 17.

## Quality Values Parameter Set syntax and semantics

Table 13. Syntax of the configuration parameters for the quantization of Quality Values.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| parameter\_set\_qvps(num\_classes) { |  |
| for (c = 0; c < num\_classes; c++) { |  |
| **class\_ID**[c] | u(4) |
| **qv\_num\_codebooks**[c] | u(16) |
| for (b = 0; b < qv\_num\_codebooks[c]; b++) { |  |
| **qv\_num\_codebook\_entries**[c][b] | u(8) |
| for (e = 0; e < qv\_num\_codebook\_entries[c][b]; e++) { |  |
| **qv\_recon**[c][b][e] | u(8) |
| } |  |
| } |  |
| } |  |
| } |  |

**class\_ID**[c] specifies one of the Data Class identifiers specified in clause 10.3. For any value of c greater than 0 it shall always be class\_ID[c] > class\_ID[c - 1].

**qv\_num\_codebooks**[c] is the number of Quality Value Codebooks used by an AU of the Data Class with the Data Class identifier class\_ID[c] referring to this Quality Values Parameter Set.

**qv\_num\_codebook\_entries**[c][b] is the number of qv\_recon elements in the Quality Value Codebook QVCodebook = b corresponding to the Data Class with the Data Class identifier class\_ID[c].

**qv\_recon**[c][b][e] is the Quality Value reconstructed from Quality Value Index QVIndex = e with Quality Value Codebook QVCodebook = b, corresponding to the Data Class with the Data Class identifier class\_ID[c].

## Computed Reference Parameter Set

This clause specifies the data structure used to carry parameters related to the reference computation algorithms specified in clause 12.3.

Table 14. Syntax of the configuration parameters for the computation of reference sequences.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| parameter\_set\_crps(num\_classes) { |  |
| for (c = 0; c <num\_classes; c++) { |  |
| **class\_ID** | u(8) |
| **cr\_alg\_ID** | u(8) |
| if(cr\_alg\_ID > 0x01){ |  |
| **cr\_buf\_max\_reads** | u(64) |
| **cr\_buf\_max\_size** | u(64) |
| } |  |
| } |  |
| } |  |

**class\_ID** specifies one of the Data Class identifiers specified in clause 10.3. For any value of c greater than 0 it shall always be class\_ID[c] > class\_ID[c - 1].

**cr\_alg\_ID** identifies the reference computation algorithm for the current Access Unit as specified in clause 12.3.1. The possible values for cr\_alg\_ID are listed in Table 15.

Table 15. Values of cr\_alg\_ID and corresponding reference computation algorithms.

|  |  |
| --- | --- |
| **cr\_alg\_ID** | **algorithm** |
| 0 | NoComp |
| 1 | RefTransform |
| 2 | PushIn |
| 3 | Local Assembly |

**cr\_buf\_max\_reads** specifies the maximum number of decoded reads to be stored during the decoding process.

**cr\_buf\_max\_size** is the maximum size in bytes of decoded reads to be stored during the decoding process.

# Access Units

An Access Unit (AU) is specified as a logical data structure containing a coded representation of genomic information or related metadata structured enable the data access and manipulation of compressed data. It is the smallest data structure that can be decoded by a decoder implementing the normative behavior described in this document.

According to the type of coded information carried by the AU, it can be decoded either independently of any other AU or using information contained in other AUs.

## Types of Access Units

AUs can be of different types according to the nature of the coded data. An Access Unit contains either a reference sequence, or a portion thereof, or encoded reads or read pairs belonging to a single class of data among those listed in section 10.3.

Table 16. Type of encoded data per 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 |

The blocks of descriptors encoded in one Access Unit as specified in clause 7.6 are those corresponding to sequence reads belonging to one class of data as specified in section 10.3. Mandatory and optional descriptors carried by each Access Unit type are listed in Table 17.

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

Figure 1 shows how a read is encoded in an Access Unit covering the position of the left-most mapped base in the read.

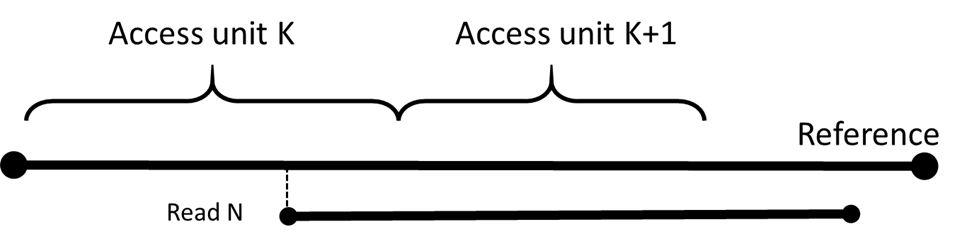


Figure 1 – A read mapped on a Reference Sequence is encoded in an Access Unit covering the position of the left-most mapped base in the read. In this figure the mapping position of Read N falls in AU K and therefore Read N is encoded in AU K, but some bases fall in the region covered by AU K+1 and AU K+2.

## Descriptors in Access Units

Table 17 specifies that the only mandatory descriptors are those required to represent the sequences of nucleotides, whereas read names and quality values are optional.

Table 17 – Descriptors required for each type of Access Unit.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **descriptor\_ID** | **Genomic descriptor** | **Access Units types** | | | | | |
|  |  | **1**  **(class P)** | **2**  **(class N)** | **3**  **(class M)** | **4**  **(class I)** | **5**  **(class HM)** | **6**  **(class U)** |
| **sequence reads** | | | | | | | |
| 0 | **pos** | mandatory | mandatory | mandatory | mandatory | mandatory | optional |
| 1 | **rcomp** | mandatory | mandatory | mandatory | mandatory | mandatory | optional |
| 2 | **flags** | mandatory | mandatory | mandatory | mandatory | mandatory | optional |
| 3 | **mmpos** | forbidden | mandatory | mandatory | mandatory | optional | forbidden |
| 4 | **mmtype** | forbidden | forbidden | mandatory | mandatory | optional | forbidden |
| 5 | **clips** | forbidden | forbidden | forbidden | mandatory | optional | forbidden |
| 6 | **ureads** | forbidden | forbidden | forbidden | forbidden | mandatory | mandatory |
| 7 | **rlen** | mandatory\* | mandatory\* | mandatory\* | mandatory\* | mandatory\* | mandatory\* |
| 8 | **pair** | mandatory\*\* | mandatory\*\* | mandatory\*\* | mandatory\*\* | mandatory\*\* | forbidden |
| 9 | **mscore** | optional | optional | optional | optional | optional | forbidden |
| 10 | **mmap** | optional | optional | optional | optional | optional | forbidden |
| 11 | **msar** | forbidden | forbidden | forbidden | optional | optional | forbidden |
| 12 | **rtype** | forbidden | forbidden | forbidden | forbidden | forbidden | optional |
| 13 | **rgroup** | optional | optional | optional | optional | optional | forbidden |
| **quality values** | | | | | | | |
| 14 | **QVIndex** | optional | optional | optional | optional | optional | optional |
| 15 | **QVCodebookIdentifier** | optional | optional | optional | optional | optional | optional |
| **read names** | | | | | | | |
| 16 | **token\_type** | optional | optional | optional | optional | optional | optional |
| **reference sequences** | | | | | | | |
| 17 | **rftp** | optional | optional | optional | optional | optional | optional |
| 18 | **rftt** | optional | optional | optional | optional | optional | optional |

\* the rlen descriptors is mandatory only in case of variable reads length

\*\* mandatory only in case of paired reads

Descriptors **pos***,* **rcomp** and **flags** are always mandatory for all classes of data, whereas other descriptors are only mandatory for some classes of data.

Table 18 shows the dependencies among descriptors for a conformant decoding process.

Table 18. Dependencies among descriptors.

|  |  |  |
| --- | --- | --- |
| **ID** | **Descriptor** | **Required descriptors** |
| **sequence reads** | | | |
| 0 | pos | flags, rcomp | |
| 1 | rcomp | pos, flags | |
| 2 | flags | pos, rcomp | |
| 3 | mmpos | pos, mmtype | |
| 4 | mmtype | pos, mmpos | |
| 5 | clips | pos, mmtype, mmpos | |
| 6 | ureads |  | |
| 7 | rlen | pos | |
| 8 | pair | pos | |
| 9 | mscore | pos | |
| 10 | mmap | pos | |
| 11 | msar | mmap | |
| 12 | rtype | pos | |
| 13 | rgroup | pos | |
| **Quality Values** | | | |
| 14 | QVIndex | QVCodebookIdentifier | |
| 15 | QVCodebookIdentifier | QVIndex | |
| **read names** | | | |
| 16 | token\_type |  | |
| **reference sequences** | | | |
| 17 | rftp | rftt | |
| 18 | rftt | rftp | |

# Sequence Reads

This clause specifies the semantics of genomic descriptors used to represent nucleotides segments and associated alignment information. Each template produced by a sequencing machine or alignment generated by an aligner is encoded in a Genomic Record by means of a subset of the genomic descriptors described in this clause. The genomic descriptors are extracted from a compliant bitstream according to the processes described in clause 13.2.6 and the genomic templates with the associated alignment information can be reconstructed from the decoded genomic descriptors according to the decoding processes described in clause 11.4.

## Supported symbols

The following alphabets are supported by this specification:

* 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)

A decoder compliant with this specification shall always use the symbol “-“ to represent a gap when the IUPAC notation is used.

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

Table 19. Identifiers of alphabets supported for sequence reads representation.

|  |  |  |
| --- | --- | --- |
| alphabet\_ID | used alphabet |  |
| 0 | DNA non IUPAC | S0 = {A, G, C, T, N} |
| 1 | DNA IUPAC | S1 = { A, G, C, T, R, Y, S, W, K, M, B, D, H, V, N, ., -} |
| 2 | RNA non IUPAC | S2 = {A, G, C, U, N} |
| 3 | RNA IUPAC | S3 = { A, G, C, U, R, Y, S, W, K, M, B, D, H, V, N, ., -} |
| 4 .. 255 | reserved |  |

## Paired reads

In case reads are generated in pairs by sequencing devices (e.g. Illumina and BGI), each pair can be encoded as a single logical data structure named Genomic Record where the mapping position of one of the reads is represented using the **pair** descriptor as specified in clause 11.4.9. The information linking one read to its mate is referred to as “pairing information” in this document.

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

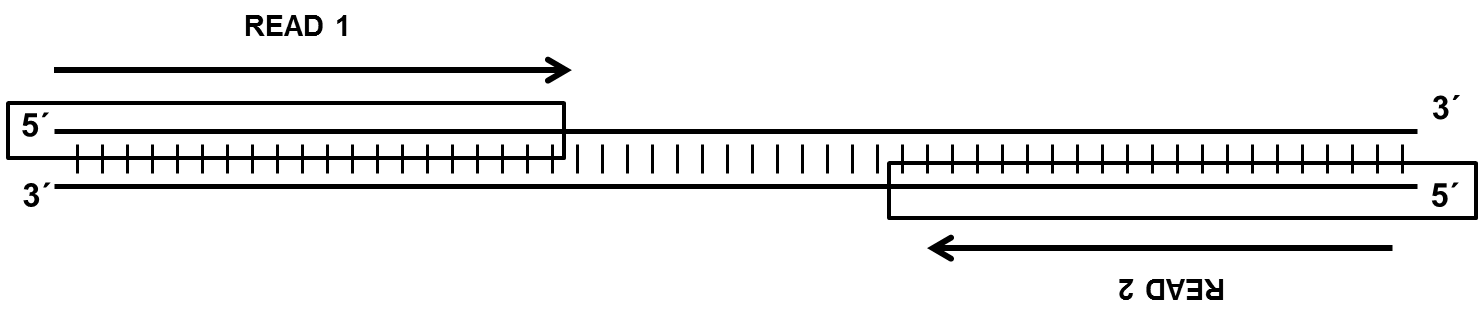


Figure 2 – In this example read 1 is sequenced from the forward strand while read 2 from the reverse strand.

Positions of mismatches with respect to the used reference sequence shall be encoded as offset from the left-most mapped base of the first read. The second read is considered to be contiguous to the first and the calculation of the actual position of mismatches on the second read requires adding the pairing distance. This is described in clause 11.4.9.

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

## Data Classes

Genomic sequence reads are classified in six data classes according to the result of their mapping against one or more reference sequences.

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

Mapped reads or templates 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.

The data classes and their descriptions are provided in Table 20.

Table 20. Sequence data classes.

|  |  |  |
| --- | --- | --- |
| **class\_ID** | **Class Name** | **Description** |
| 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 at least one substitution, and optionally unknown bases, but no insertions, no deletions and no clipped bases. |
| 4 | Class\_I | Reads containing at least one insertion, deletion or clipped base, and optionally unknown bases or substitutions. |
| 5 | Class\_HM | Half-mapped pairs where only one read is mapped. |
| 6 | Class\_U | Unmapped reads. |

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

## Aligned Data

In the context of this document, aligned genomic data are genomic segments which require the use of an external or internal Reference Genome (as specified in Clause 12) to be decoded.

This clause specifies the types of descriptors contained in the Blocks Payload specified in clause 7.8. Each Block contains binary coded descriptors of a single type identified by the descriptor\_ID present in the Block Header as specified in clause 7.7.

Once decoded, each descriptor shall be used to initialize one or more MPEG-G record fields as specified in clause 10. Table 21 lists the descriptors used for aligned reads with a brief description and reference to the corresponding clause.

Table 21. Descriptors used to represent aligned sequence reads.

|  |  |  |  |
| --- | --- | --- | --- |
| **descriptor\_ID** | **descriptor** | **semantics** | **clause** |
| 0 | **pos** | read mapping position | 11.4.1 |
| 1 | **rcomp** | strand information for reads in a template | 11.4.2 |
| 2 | **flags** | additional alignment information usually produced by aligners | 11.4.3 |
| 3 | **mmpos** | position of mismatches in reads | 11.4.4 |
| 4 | **mmtype** | type of mismatches | 11.4.5 |
| 5 | **clips** | information on clipped bases (i.e. soft clips or hard clips) | 11.4.6 |
| 6 | **ureads** | unmapped reads encoded verbatim | 11.4.7 |
| 7 | **rlen** | read lengths | 11.4.8 |
| 8 | **pair** | It can represent:   * The absolute signed distance from one segment to the next   OR   * The absolute position on a reference sequence of a read in a template | 11.4.9 |
| 9 | **mscore** | provides a score per alignment | 11.4.13 |
| 10 | **mmap** | used to represent multiple alignments | 11.4.11.1 |
| 11 | **msar** | supports spliced alignments and alternative secondary alignments which do not preserve the same contiguity of mapping of the primary alignment | 11.4.11 |
| 13 | **rgroup** | identifier of the read group each Genomic Record belongs to | 11.4.13 |

## Raw Data

Raw reads belong to class U only. They are encoded as unmapped reads in aligned datasets. Some of the descriptors specified for reads aligned to an external or internal reference as specified in clause 10.4 are used to encode raw reads. This is motivated by the fact that raw reads are encoded using reference sequences built from the data to be encoded. The reference used for mapping is computed according to the procedures described in Clause 12.3.

Table 22 - Descriptors used to represent raw sequence reads.

|  |  |  |  |
| --- | --- | --- | --- |
| **descriptor\_ID** | **Descriptor** | **Semantics** | **Clause** |
| 0 | **pos** | read mapping position | 11.4.1 |
| 1 | **rcomp** | strand information for reads in a template | 11.4.2 |
| 2 | **flags** | used to cover a part of SAM flags. | 11.4.3 |
| 4 | **mmpos** | mismatch position | 11.4.4 |
| 5 | **mmtype** | type of edit operations:  substitutions  deletions  insertions | 11.4.5 |
| 6 | **clips** | string of nucleotides with variable length (e.g. soft clips) | 11.4.6 |
| 7 | **ureads** | unmapped reads encoded verbatim | 11.4.7 |
| 8 | **rlen** | unsigned integer representing the number of bases in the read minus one. | 11.4.8 |
| 13 | **rtype** | This identifies the subset of descriptors needed to decode the read. | 11.4.10 |

# Decoding process

This clause describes the decoding process an MPEG-G compliant decoder has to conform to in order to reconstruct the genomic information encoded in a MPEG-G bitstream.

Input to this process is a bitstream. Output of this process can be:

1. a Raw Reference as specified in clause 7.2.
2. a list of decoded MPEG-G records as specified in clause 0.

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

## General

The decoding process operates as follows:

1. The decoding of Data Units is specified in clause 11.2.
2. The decoding of Access Units is specified in clause 11.3.

## Data Units

Inputs to this process are Data Units as specified in clause 7.1.

Outputs of this process are the MPEG-G coded data structures encapsulated within the Data Units.

The decoding process for each Data Unit extracts the MPEG-G coded data structure from the Data Unit and then parses the MPEG-G coded data structure.

## Access Units

Input to the processes described in the following clauses are decoded genomic descriptors generated as output of the parsing process specified in clause 13. The genomic descriptors are contained in the decoded\_symbols data structure specified in clause 13.2.2.

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

decoded\_symbol[descriptor\_ID][descriptor\_subsequence\_ID][ jdescriptor\_ID, descriptor\_subsequence\_ID]

where jdescriptor\_ID, descriptor\_subsequence\_ID is used to read the arrays of decoded genomic descriptors as specified in clause 13.2.2. For descriptors where subsequences are not present descriptor\_subsequence\_ID is always set to 0.

At the beginning of the decoding process of one AU all indexes jdescriptor\_ID, descriptor\_subsequence\_IDare initialized to 0.

Output of this process is:

1. if dataset\_type, as specified in clause 8, is equal to 2 a Raw Reference structure as specified in clause 7.2.
2. if dataset\_type, as specified in clause 8, is less than 2 a sequence of MPEG-G records as specified in clause 0.

### References Padding

In case of AUs of type P, N, M, I and HM, if the Raw Reference structure containing the Reference Sequence to be used during the decoding process specifies a seq\_start that is greater than AU\_start\_pos or an seq\_end that is less than AU\_end\_pos, the decoder shall pad with “N” the missing portions of Reference Sequence. This is shown in Figure 3.

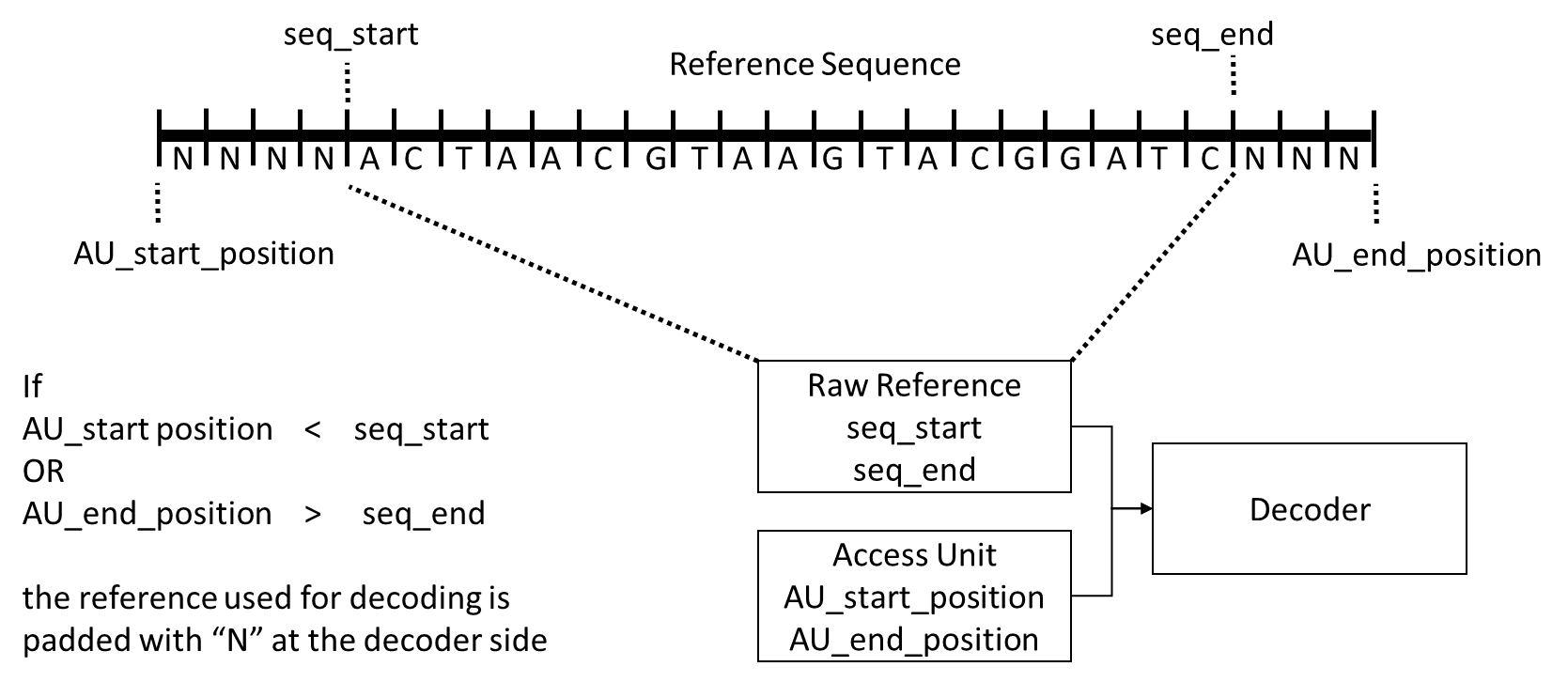


Figure 3. The decoder shall pad the Reference Sequence with “N” in case the AU covered region is wider than the available Reference Sequence segment.

### Type 1 (Class P)

Type 1 Access Units encode aligned sequence reads which perfectly match to the used Reference Sequence.

The decoding process of a binary decoded Access Unit of type 1 is as follows:

1. Decode the **pos** descriptor as specified in clause 11.4.1. The number of **pos** descriptors relevant for each Genomic Record corresponds to the number of alignments of the read or read pair contained in the Genomic Record. This is signaled by the **mmap** descriptor as specified in clause 11.4.11.1;
2. Decode the **rcomp** descriptor as specified in clause 11.4.2. The number of **rcomp** descriptors relevant for each Genomic Record corresponds to the number of segments in all alignments of the read or read pair contained in the Genomic Record. This is signaled by the **mmap** descriptor as specified in clause 11.4.11.1;
3. Decode the **flags** descriptor as specified in clause 11.4.3.
4. If present, decode the following optional descriptors:
   1. decode the **rlen** descriptor as specified in clause 11.4.8;
   2. (paired-end reads), decode the **pair** descriptor as specified in clause 11.4.3. The number of **pair** descriptors relevant to each Genomic Record is equal to number\_of\_alignments \* (number\_of\_record\_segments - 1). These two variables are calculated as specified in points 5.h and 5.i below;
   3. decode the **mscore** descriptor as specified in clause 11.4.10;
   4. decode the **mmap** descriptor as specified in clause 11.4.11.1;
   5. decode the **msar** descriptor as specified in clause 11.4.11.6;
   6. decode the **token\_type** descriptor as specified in clause 11.4.16;
   7. decode the **QVIndex** descriptor as specified in clause 11.4.14;
   8. decode the **QVCodebookIdentifier** descriptor as specified in clause 0;
5. The relevant MPEG-G record fields specified in clause 0 are initialized as follows:
   1. global\_ID is set as specified in clause 14.1;
   2. read\_name is set to the decoded read identifier as specified in clause 11.4.16;
   3. class\_type is set to 1;
   4. seq\_ID is set to the value of **sequence\_ID** as specified in clause 7.5;
   5. read\_1\_first is set as specified in clause 11.4.9;
   6. flags is set to the value of the decoded **flags** descriptor;
   7. number\_of\_template\_segments is set:
      1. to 1 if neither **pair** nor **mmap** descriptors are present in this AU;
      2. to 2 if the **pair** descriptor is present but no **mmap** descriptors are present in this AU;
      3. to 1 or 2 if both the **pair** and **mmap** descriptors are present in this AU. Clause 11.4.11 specifies how to calculate number\_of\_template\_segments in this case;
   8. number\_of\_record\_segments is set:
      1. to 1 if neither **pair** nor **mmap** descriptors are present in the current AU;
      2. as specified in clause 11.4.9 if the **pair** descriptor is present but no **mmap** descriptors is present in the current AU;
      3. as specified in clause 11.4.11.1 if both **pair** and **mmap** descriptors are present in the current AU;
   9. number\_of\_alignments is set:
      1. to 1 if the **mmap** descriptor is not present in the current AU;
      2. as specified in clause 11.4.11.1 if the **mmap** descriptor is present in the current AU;
   10. read\_len is an array of number\_of\_record\_segments values representing the length in nucleotides of each segment of the Genomic Record;
       1. if neither the **pair** nor the **mmap** descriptors are present for the current AU, the array contains a single value equal to the **rlen** descriptor value decoded for the current Genomic Record;
       2. if the **pair** descriptor is present for the current AU, but not the **mmap** descriptor, the values in the array are all set to **read\_len** as specified in clause 8;
       3. if **mmap** is present for the current AU, the values of the array are calculated as specified in clause 11.4.11.1;
   11. sequence is an array of number\_of\_record\_segments strings of symbols from the alphabet identified by alphabet\_ID as specified in clause 8; each string has a length equal to the corresponding element in the read\_len array. The sequence of nucleotides is computed as specified in clause 11.5;
   12. quality\_values is an array of number\_of\_record\_segments strings of symbols computed as specified in clauses 11.4.14 and 0;
   13. the mapping\_pos array contains number\_of\_alignments **pos** values decoded as specified in clause 11.4.1. In case of multiple alignments mapping\_pos[0] shall contain the mapping position of the primary alignment;
   14. the ecigar\_len array contains number\_of\_alignments \* number\_of\_record\_segments values representing the length in characters of the corresponding ecigar\_string string as specified in clause 14.19;
   15. the ecigar\_string array contains number\_of\_alignments \* number\_of\_record\_segments strings. Each string ecigar\_string[i][j] is computed as specified in clause 11.6 to represent the mismatches, indels, clipped bases and information on multiple alignments and spliced reads of the jth segment of the ith alignment;
   16. The field reverse\_comp for each segment in each alignment as specified in clause 14.21 shall be initialized according to the **rcomp** value as follows:
       1. For each alignment, one value of **rcomp** is decoded;
       2. The value of reverse\_comp for each segment of the alignment is set to the value of the corresponding bit of the **rcomp** value starting from the left-most segment of read 1 and continuing till the right-most segment of read 2; bits are considered from the MSB to the LSB.
   17. the split\_alignment array contains number\_of\_alignments \* (number\_of\_record\_segments - 1) values. Each value split\_alignment [i][j] is set to 1 when the (j+1)th segment of the ith segment is coded in a different Genomic Record as specified in clause 14.15;
   18. delta is an array of number\_of\_alignments \* (number\_of\_record\_segments - 1) values computed as specified in clause 11.4.9; in case of multiple alignments the value delta[0][0] contains the pairing distance of the primary alignment;
   19. split\_pos is an array of number\_of\_alignments \* (number\_of\_record\_segments - 1) values wherein each value split\_pos[i][j] contains a valid value only if the corresponding split\_alignment[i][j] value is equal to 1. The value of split\_pos[i][j] is computed as specified in clause 11.4.9;
   20. split\_seq\_ID is an array of number\_of\_alignments \* (number\_of\_record\_segments - 1) values wherein each value split\_seq\_ID [i][j] contains a valid value only if the corresponding split\_alignment[i][j] value is equal to 1. The value of split\_seq\_ID[i][j] is computed as specified in clause 11.4.9.
   21. more\_alignment is set to 1 only if the subsequence with descriptor\_subsequence\_ID equal to 0 contains the value 6 for the current Genomic Record as specified in Table 53;
   22. if more\_alignments is set to 1, next\_pos and next\_seq\_ID are computed as specified in clause 11.4.9.

### Type 2 (Class N)

Access Units of type 2 (Class N) are decoded by following the process described for AUs of type 1 (Class P) in clause 11.4.1 and then applying the information on unknown bases (symbol N) carried by the **mmpos** descriptor as specified in clause 11.4.4.

Additional inputs to this process is the offset[]vector specified in clause 11.4.4.

The decoded sequence shall be computed by replacing with the symbol ‘N’ each base at a position represented by a decoded **mmpos** value in the sequence obtained as specified in clause 11.5.1.

The substitutions are applied as follows.

For each value of offset[i] belonging to read 1, the base at the corresponding position in the sequence computed as specified in clause 11.5.1 is replaced by a symbol ‘N’.

sequence[offset[i]] = ‘N’

For paired end reads, the same applies to the sequence representing read 2 for values of offset[i] belonging to read 2 as specified in clause 11.4.4.

### Type 3 (Class M)

Access Units of type 3 (Class M) are decoded by following the process described for AUs of type 1 (Class P) in clause 11.4.1 and then applying the information on substitutions obtained by following the decoding process of **mmpos** and **mmtype** descriptors as specified in clauses 11.4.4 and 11.4.5.

Additional inputs to this process are the offset[] and substitutions[] vectors specified in clauses 11.4.4 and 11.4.5.

The substitutions are applied as follows.

For each value of offset[i] belonging to read 1, the base at the corresponding position in the sequence computed as specified in clause 11.5.1 is replaced by the corresponding symbol mismatch[i] obtained as specified in clause 11.4.5.

sequence[offset[i]] = symbol corresponding to substitutions [i] in Table 30.

For paired end reads, the same applies to the sequence representing read 2 for values of offset[i] belonging to read 2 as specified in clause 11.4.4.

### Type 4 (Class I)

Access Units of type 4 (Class I) are decoded by following the process described for AUs of type 1 (Class P) in clause 11.4.1 and then applying the edit operations represented by the decoded **mmpos, mmtype** and **clips** descriptors as specified in clauses 11.4.4, 11.4.5 and 11.4.6.

Additional inputs to this process are:

* the offset[] vector specified in clause 11.4.4.
* the substitutions[] and indels vectors specified in clause 11.4.5.
* the soft\_clips[][] matrix and the hard\_clips[] vector specified in clause 11.4.6.

The substitutions are applied as follows.

For each value of offset[i] belonging to read 1, the base at the corresponding position in the sequence computed as specified in clause 11.5.1 is replaced by the corresponding symbol mismatch[i] obtained as specified in clause 11.4.5.

sequence[offset[i]] = symbol corresponding to substitutions[i] in Table 30.

Insertions and deletions are applied as specified in Table 23.

Table 23. How to apply mismatches to sequence reads according to the decoded values of mmtype.

|  |  |
| --- | --- |
| **Semantics of indels[i] as specified in Table 30.** | **Operation to be performed on sequence[]** |
| insertion | sequence[offset[i]] is shifted right by one position and the symbol corresponding to indels[i] in Table 30 is inserted. |
| deletion | sequence[offset[i]] is deleted. |

Information on clipped bases is applied as follows:

**Soft clips**

The following applies to each non-empty row of the soft\_clips bi-dimensional vector computed as per clause 11.4.6:

* soft\_clips[0] shall be prepended to the sequence representing read 1;
* soft\_clips[1] shall be concatenated to the sequence representing read 1;
* soft\_clips[2] shall be prepended to the sequence representing read 2;
* soft\_clips[3] shall be concatenated to the sequence representing read 2.

**Hard clips**

The hard\_clips[] vector is used to compute the ecigar\_string element of the MPEG-G record as specified in clause 11.6.

### Type 5 (Class HM)

Class HM applies only to paired end reads. Access Units of type 5 are decoded as follows:

1. The mapped read is decoded by following the process described for Class I in clause 11.3.5.
2. The unmapped read is obtained by decoding the **uread** syntax element as described in clause 11.3.7.

### Type 6 (Class U)

Class U data can be decoded in two modes:

1. as entropy coded **ureads** descriptors to be decoded as specified in clause 13.2.6.2.6. The decoding process of unmapped sequence reads from **ureads** descriptors is specified in clause 11.4.7.
2. using the “PushIn” computed reference algorithm specified in clause 12.3.1.2. In this case the genomic sequence reads are decoded as for other class of data by using the **rtype** descriptor as specified in clause 11.4.11.6.

### Reference sequences

Input to this process is one AU of type 1, 2, 3, 4 or 6.

Output of this process is a raw\_reference structure as specified in clause 7.2.3.

Clause 8 specifies that all AUs referring to a Parameter Set having **dataset\_type** set to 2 contain an encoded Reference Genome or portions thereof. According to the value of **AU\_type** specified in clause 7.5 the decoding process is as specified in clauses 11.3.2, 11.3.3, 11.3.4, 11.3.5 and 11.3.7 for Classes P, N, M, I and U.

The elements of the raw\_reference syntax specified in clause 7.2.3 shall be set as follows:

**reference\_ID** is set to the value of reference\_ID found in the Dataset Header defined in ISO/IEC 23092-1.

**seq\_count** is set to the number of different values of **sequence\_ID,** specified in clause 7.5, found in the headers of the AUs with **dataset\_type** equal to 2 referring to the same Parameter Set.

For each value of **sequence\_ID,** specified in clause 7.5, found in the headers of the AUs with **dataset\_type** equal to 2 referring to the same Parameter Set the following applies:

**sequence\_ID** in the raw\_reference syntaxis set to **sequence\_ID**

**seq\_start** shall be set to the smallest value of AU\_start\_position among all AUs with **dataset\_type** equal to 2 referring to the same Parameter Set.

**seq\_end** shall be set to the largest value of AU\_end\_position among all AUs with **dataset\_type** equal to 2 referring to the same Parameter Set.

**seq\_start** and **seq\_end** elements specified in clause 7.2.3 shall be set to the values of AU\_start\_position and AU\_end\_position respectively.

### Quality Values

#### Decoding of Unquantized Quality Values

The syntax of encoded quality value contains compressed representation of unquantized quality value descriptor sequences as defined below.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| encoded\_quality\_value\_identifiers() { |  |
| **region\_len** | u(4) |
| **max\_r** | u(4) |
| **min\_q** | u(8) |
| **max\_q** | u(8) |
| do { |  |
| encoded\_quality\_value\_stream() |  |
| i++ |  |
| } while(i < num\_quality value\_streams && more\_data\_in\_block\_payload( )) |  |
| } |  |

**region\_len** signals the length of the subregion for calculating region context for decoding of quality value. The actual region length is region\_len\*10.

**max\_r** is the maximum number of repetitions of quality value encoded. Note that if the actual number of repetitions are larger than max\_r, it will be decomposed and encoded as a series of max\_r following remaining value.

**min\_q** is the minimum quality value encoded.

**max\_q** is the maximum quality value encoded.

The output of this process is quality value descriptor stream qv[i][j]. The decoding process is illustrated below:

|  |  |
| --- | --- |
| **Decoding process** | **Type** |
| DecodeQV(i) { |  |
| ctxTable[ ] = contexts\_creation\_initialization() |  |
| j = 0 |  |
| q=r=0; |  |
| max\_ctx\_q = round(max\_rlen\*10/region\_len)\*max\_q+max\_q+1; |  |
| do { |  |
| ctxIdx = context\_selection((j==0) ? 0:round(j\*10/region\_len)\*max\_q + q + 1) |  |
| q = decodedCabacSymbol() |  |
| ctxIdx = context\_selection( round(j\*10/region\_len)\*max\_q + q+max\_ctx\_q) |  |
| r = decodedCabacSymbol() |  |
| while ( decodedCabacSymbol =max\_r){ |  |
| r += decodedCabacSymbol() |  |
| } |  |
| do { |  |
| qv[i][j++]=q+min\_q |  |
| } while (r--) |  |
| } while (j < num\_quality value\_streams < && more\_data\_in\_block\_payload( )) |  |
| } |  |

#### Decoding of Quantized Quality Values

Inputs to this process are the Block Payloads (as specified in clause 7.8) for descriptor\_ID = 14, which corresponds to the Quality Value Indexes, and for descriptor\_ID = 15, which corresponds to the Quality Value Codebook Identifiers.

The Quality Values are decoded by transforming the Quality Value Indexes into reconstructed Quality Values. The transformation is done by a lookup in a Quality Value Codebook. Quality Value Codebooks are identified by Quality Value Codebook Identifiers. The encoder can select different codebooks (i.e. different quantization characteristics) per genomic position.

The jth QVCodebookIdentifier value corresponds to the jth genomic position after AU\_start\_position. Each QVIndex value corresponds to a base in a read. The absolute position of the base, and thus of the QVIndex value, can be computed according to clause 10.4.

Output of this process is a reconstructed Quality Value represented by the variable qvRecon.

This process consists of the following steps:

For each QVIndex value:

1. Compute the absolute position p of the QVIndex value.
2. Select the Quality Value Codebook, identified by the QVCodebookIdentifier value for position p, as b..
3. Initialize the variable qvRecon with the Quality Value Codebook entry qv\_recon[c][b][QVIndex] , where c is the Data Class identifier of the current AU, as specified in clause 8.1.

In the case that the QVIndex descriptor is multiplexed, the QVIndex descriptor is decomposed into qv\_num\_codebooks descriptor subsequences. Each descriptor substream corresponds to one Quality Value Codebook as specified in clause 11.4.15.

Multiplexing of the QVIndex descriptor can prove to be beneficial in terms of compression performance, because the encoder can model for each QVIndex symbol the conditional probability of the symbol given the corresponding codebook.

The process to multiplex these subsequences to output this descriptor stream is specified as follows.

1. For each read in an Access Unit:
   1. For each base in a read:
      1. Calculate the absolute position of the base, and thus the corresponding Quality Value, according to clause INSERT LINK TO 11.41, as j.
      2. Get the codebook QVCodebookIdentifier[j] at position j as b.
      3. Insert 1 byte from substream b.

### Read identifiers decoding process

Inputs to this process is the Block Payload (as specified in clause 7.8) for descriptor\_ID = 16, which corresponds to the read identifiers. The **encoded\_read\_identifiers**()of this Block Payload internally contains a list of compressed representation of **token\_type** descriptor sequences (as specified in clause 0).

Output of this process is the list of decompressed representation of these **token\_type** descriptor sequences, which serve as input to the assembly process (specified in clause 11.3.10.3) to reconstruct the read identifiers.

#### Syntax and semantics

The syntax of **encoded\_read\_identifiers**() containing a list of compressed representation of **token\_type** descriptor sequences is specified below.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| encoded\_read\_identifiers() { |  |
| **num\_read\_identifiers** | u(32) |
| **num\_token\_type\_sequences** | u(16) |
| i = 0 |  |
| do { |  |
| encoded\_token\_type\_sequence(i) |  |
| i++ |  |
| } while(i < num\_token\_type\_sequences && more\_data\_in\_block\_payload( )) |  |
| } |  |

**num\_read\_identifiers** specifies the number of read identifiers encoded in the current Block Payload.

**num\_token\_type\_sequences** specifies the number of **token\_type** descriptor sequences in the current Block Payload.

**encoded\_token\_type\_sequence**(i) specifies the data structure containing the byte-aligned compressed representation of the ith token\_type descriptor sequence. Its syntax is specified below.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| encoded\_token\_type\_sequence(i) { |  |
| **type\_ID** | u(4) |
| **method\_ID** | u(4) |
| if(method\_ID == 0x0) |  |
| **ref\_type\_ID** | u(16) |
| decode\_token\_type\_sequence(i, method\_ID) |  |
| } |  |

**type\_ID** specifies the type of the ith **token\_type** descriptor sequence. This process internally maintains a state variable typeNum, which is initialized with -1 for every Block Payload of this descriptor (descriptor\_ID = 16) and is incremented for every **type\_ID = 0**. The current values of state variable typeNum and **type\_ID** are then used to generate a “mapped” value of **type\_ID** as

|  |
| --- |
| if(type\_ID == 0)  typeNum++ mappedTypeId = (typeNum<<4) | (type\_ID & 15) |

This mappedTypeId may be signaled as **ref\_type\_ID** in a future record if its payload is an exact duplicate of a previously decoded record.

**method\_ID** specifies the compression method (among those listed in Table 24) used for the ith **token\_type** descriptor sequence.

Table 24: Description of compression methods for token\_type descriptor sequence.

|  |  |  |
| --- | --- | --- |
| **method\_ID** | **Description** | |
| 0 | DUP | The compressed payload is an exact duplicate of the previously decoded token\_type\_sequence with mappedTypeId = ref\_type\_ID as specified in clause 11.3.10.2.1. |
| 1 | CAT | The null coding, ideal for small data. Its syntax is specified in clause 11.3.10.2.2. |
| 2 | RLE | Run length coding, ideal for long list of repeated symbols. Its syntax is specified in clause 11.3.10.2.3 |
| 3 | CABAC\_ORDER\_0 | Zero order coding with CABAC. Its syntax is specified in clause 11.3.10.2.4. |
| 4 | CABAC\_ORDER\_1 | First order coding with CABAC binarization. Its syntax is specified in clause 11.3.10.2.5. |
| 5 | X4 | A recursive decorrelation method to split a token\_type\_sequence into four equisized interleaved subsequences (whenever size is divisible by 4 and greater than 32), each of them being coded with one of the above methods except method\_ID 0x0. Its syntax is specified in clause 11.3.10.2.6. |
| 0x6 .. 0xf | reserved |  |

**ref\_type\_ID** is the mappedTypeId of a previously decoded token\_type sequence of which payload of current token\_type sequence is an exact duplicate.

**decode\_token\_type\_sequence(i, method\_ID)** specifies the syntax for decoding the ith **token\_type** descriptor sequence using the decoding method indicated by method\_ID. Its syntax is specified below.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| decode\_token\_type\_sequence(i, method\_ID) { |  |
| if(method\_ID == 0x0) |  |
| DUP(i) |  |
| else { |  |
| **uncompressed\_size** | u7(v) |
| if(method\_ID == 0x1) |  |
| CAT(i) |  |
| else if(method\_ID == 0x2) |  |
| RLE(i) |  |
| else if(method\_ID == 0x3) |  |
| CABAC\_ORDER\_0(i) |  |
| else if(method\_ID == 0x4) |  |
| CABAC\_ORDER\_1(i) |  |
| else if(method\_ID == 0x5) |  |
| X4(i) |  |
| } |  |
| } |  |

**uncompressed\_size** represents the size (in number of tokens) of decompressed payload of the ith **token\_type** descriptor sequence.

#### Decoding process for compression methods

The input to this process is the data structure encoded\_token\_type\_sequence() specifying the byte-aligned compressed representation of the ith token\_type descriptor sequence, which is decoded with one of the compression methods listed in Table 24 and specified in this clause.

The output of this process is the decompressed representation of the ith **token\_type** descriptor sequence.

##### DUP

Input to this process is ref\_type\_ID, which is a mappedTypeId of a previously decoded token\_type sequence.

Output of this process is the decompressed payload of current token\_type sequence, which is obtained by copying the decompressed payload of the reference token\_type sequence uniquely identified by mappedTypeId.

##### CAT

This clause specifies the decoding process for the method CAT, which may be used to decompress a token\_type descriptor sequence. The output of this process is a reconstructed token\_type descriptor sequence.

|  |  |
| --- | --- |
| **Decoding process** | **Type** |
| CAT(i) { |  |
| j = 0 |  |
| do { |  |
| **decoded\_tokens[i][j++]** | ue(v) |
| } while(j <uncompressed\_size && more\_data\_in\_block\_payload( )) |  |
| } |  |

**decoded\_tokens[i][j]** specifies the jth token in the ith decompressed **token\_type** descriptor sequence.

**more\_data\_in\_block\_payload**( ) is specified in clause 6.3.

##### RLE

This clause specifies the decoding process for the method RLE, which isused to decompress a token\_type descriptor sequence. The output of this process is a reconstructed token\_type descriptor sequence.

|  |  |
| --- | --- |
| **Decoding process** | **Type** |
| RLE(i) { |  |
| j = 0 |  |
| do { |  |
| **tmp\_value** | u(8) |
| if(tmp\_value == rle\_guard) { |  |
| **tmp\_value** | u(8) |
| if(tmp\_value == 0) |  |
| decoded\_tokens[i][j++] = rle\_guard |  |
| else { |  |
| rle\_len = 0 |  |
| s = 0 |  |
| do { |  |
| if(s != 0) |  |
| **tmp\_value** | u(8) |
| run\_len |= (tmp\_value & 0x7f) << s; |  |
| s += 7 |  |
| } while(tmp\_value & 0x80) |  |
| **tmp\_value** | ue(v) |
| for(r=0; r < ren\_len; r++) { |  |
| decoded\_token[i][j++] = tmp\_value |  |
| } |  |
| } |  |
| } else |  |
| decoded\_token[i][j++] = tmp\_value |  |
| } while(j <uncompressed\_size && more\_data\_in\_block\_payload( )) |  |
| } |  |

**rle\_guard** specifies the guard value signaled in decoder configuration for read identifiers (see 13.2.3).

**decoded\_token[i][j]** specifies the jth token in the ith decompressed **token\_type** descriptor sequence.

**more\_data\_in\_block\_payload**( ) is specified in clause 6.3.

##### CABAC\_ORDER\_0

This clause specifies the decoding process for the method CABAC\_ORDER\_0, which may be used to decompress a token\_type descriptor sequence. The output of this process is a reconstructed token\_type descriptor sequence.

|  |  |
| --- | --- |
| **Decoding process** | **Type** |
| CABAC\_ORDER\_0(i) { |  |
| ctxTable[ ] = contexts\_creation\_initialization(0) |  |
| j = 0 |  |
| do { |  |
| k=encoded\_tokens[i][j] = 0 |  |
| for (s=0; s<output\_symbol\_size; s+= coding\_symbol\_size[k++]) { |  |
| ctxIdx = context\_selection(0, NULL) |  |
| decodedCabacSymbol | ae(v) |
| encoded\_tokens[i][j] |= decodedSymbol <<s |  |
| } |  |
| j++; |  |
| } while (j <uncompressed\_size && more\_data\_in\_block\_payload( )) |  |
| } |  |

**decoded\_tokens[i][j]** specifies the jth token in the ith decompressed **token\_type** descriptor sequence.

**contexts\_creation\_initialization()** and **context\_selection()** represent the initialization of ctxTable (specified in clause 14.2.6.2.3) and selection of context index ctxIdx inside ctxTable (specified in clause 14.2.6.2.5), respectively, using the parameters signaled in the decoder configuration for read identifiers (specified in clause 13.2.3).

**decodedCabacSymbol** specifies the decoded symbol as specified in clause 13.2.6.2.6..

**more\_data\_in\_block\_payload**( ) is specified in clause 6.3.

##### CABAC\_ORDER\_1

This clause specifies the decoding process for the method CABAC\_ORDER\_1, which isbe used to decompress a token\_type descriptor sequence. The output of this process is a reconstructed token\_type descriptor sequence.

|  |  |
| --- | --- |
| **Decoding process** | **Type** |
| CABAC\_ORDER\_1(i) { |  |
| ctxTable[ ] = contexts\_creation\_initialization(0) |  |
| lut\_values[ ][ ] = decode\_LUTs\_Order\_1(0, ctxTable, 0) |  |
| j = 0 |  |
| prev\_value[1] = {0} |  |
| do { |  |
| decoded\_tokens[i][j] = 0 |  |
| for (s=0; s<output\_symbol\_size; s+= coding\_symbol\_size) { |  |
| ctxIdx = context\_selection(0, prev\_value) |  |
| decodedCabacSymbol | ae(v) |
| decodedSymbol = LUT\_values[prev\_value[0]][ decodedCabacSymbol ] |  |
| prev\_value[0] = decodedSymbol |  |
| decoded\_tokens[i][j] |= decodedSymbol <<s |  |
| } |  |
| j++ |  |
| } while(j <uncompressed\_size && more\_data\_in\_block\_payload( )) |  |
| } |  |

**LUT\_values[ ][ ]** specifies the 2-dimensional look-up-table signaled in the payload of the ith **token\_type** descriptor sequence. This look-up-table is retrieved using the process decode\_LUTs\_Order\_1() specified in 13.2.6.2.4.

**decoded\_tokens[i][j]** specifies the jth token in the ith decompressed **token\_type** descriptor sequence.

**contexts\_creation\_initialization()** and **context\_selection()** specifies the initialization of ctxTable (specified in clause 14.2.6.2.3) and selection of context index ctxIdx inside ctxTable (specified in clause 14.2.6.2.5), respectively, using the parameters signaled in the decoder configuration for read identifiers (specified in clause 13.2.3).

**decodedCabacSymbol** specifies the decoded symbol as specified in clause 13.2.6.2.6.

**decodedSymbol** is the value read from the LUT using decodedCabacSymbol as index.

**more\_data\_in\_block\_payload**( ) is specified in clause 6.3.

##### X4

This clause specifies the decoding process for the method X4, which is be used to decompress a token\_type descriptor sequence. The output of this process is a reconstructed token\_type descriptor sequence.

|  |  |
| --- | --- |
| **Decoding process** | **Type** |
| X4(i) { |  |
| x4\_method\_IDs | u(16) |
| for (s=0; s<4; s++) { |  |
| method\_ID = (x4\_method\_IDs >>(s\*4)) & 0xf | u(4) |
| decoded\_tokens\_x4[s][] = decode\_token\_type\_sequence(s, method\_ID) |  |
| } |  |
| /\* Multiplexing of interleaved subsequences \*/ |  |
| for(j=0, j<uncompressed\_size; j += 4) { |  |
| for(s=0, s<4; s++) { |  |
| decoded\_tokens[i][j+s] = decoded\_tokens\_x4[s][j>>2] |  |
| } |  |
| } |  |
| } |  |

**x4\_method\_IDs** specifies the four compression methods (among those listed in Table 24 except method\_ID = 0x0) used to decompress the four interleaved subsequences, where the method\_ID for the sth subsequence can be derived as method\_ID = (x4\_method\_IDs >>(s\*4)) & 0xf.

**decoded\_tokens\_x4[s][j]** specifies the jth byte token in the sth decompressed interleaved subsequence.

**decoded\_tokens[i][j]** specifies the jth byte token in the ith decompressed **token\_type** descriptor sequence.

#### Assembly process for token\_type descriptor sequences

Input to this process is decoded\_tokens[ ][ ], which is the decompressed representation of encoded\_read\_identifiers() specified in clause 11.3.10.1, containing a list of num\_token\_type\_streams decompressed **token\_type** descriptor sequences.

Output of this process is the data structure read\_ids[ ] containing a list of read identifiers as strings.

|  |
| --- |
| **Syntax** |
| Assemble\_ Read\_Identifiers(decoded\_tokens[ ][ ]) { |
| cIdx = 0 |
| refIdx = 0 |
| t = 0 |
| read\_ids[ ] = {“”} |
| do { |
| tokType = get\_tok\_type(decoded\_tokens[t<<4]) |
| distance = get\_tok\_int(decoded\_tokens[t<<4 | tokType]) |
| refIdx = cIdx – distance |
| if(tokType == 0x00) /\* Token: DUP \*/ |
| strcpy(read\_ids[cIdx], read\_ids[refIdx]) |
| else { /\* Token: DIFF \*/ |
| for (t=1; t< num\_token\_type\_sequence; t++) { |
| tokType = get\_tok\_type(decoded\_tokens[t<<4]) |
| if(tokType == 0x09) /\* Token: END \*/ |
| break |
| tokStr = extract\_tok\_value (decoded\_tokens, tokType, t, refIdx) |
| strcat(read\_ids[cIdx], tokStr) |
| } |
| } |
| } while ( cIdx < num\_read\_identifiers && strlen(read\_ids[cIdx++]) > 0) |
| } |

**num\_read\_identifiers** specifies the number of read identifiers encoded in the current Block Payload.

**get\_tok\_type(decoded\_tokens[ ])** pops and returns one byte from data structure decoded\_tokens[ ].

**get\_tok\_int(decoded\_tokens[ ])** pops four bytes from data structure decoded\_tokens[ ] and returns little-endian representation of their corresponding 32-bit integer.

**strcpy(dst, src)** specifies the string copying operation from the source string to the destination string.

**strcat(dst, src)** specifies the string concatenation operation of source string to the destination string.

**strlen(str)** returns the length of the input string.

**extract\_tok\_value()** pops and returns token value based on its type (as listed in Table 39) and the co-located tokens in the reference read identifier. The syntax of extract\_tok\_value () is described in below.

|  |
| --- |
| **Syntax** |
| extract\_tok\_value(decoded\_tokens[ ][ ], tokType, t, refIdx) { |
| tokIdx = (t << 4) | tokType |
| if(tokType == 0x02) /\* Token: CHAR \*/ |
| tmp\_str = get\_tok\_string(decoded\_tokens[tokIdx]) |
| else if(tokType == 0x03) /\* Token: CHAR \*/ |
| tmp\_str = parseToStr(get\_tok\_char(decoded\_tokens[tokIdx])) |
| else if(tokType == 0x04) /\* Token: DIGITS \*/ |
| tmp\_str = parseToStr(get\_tok\_digits(decoded\_tokens[tokIdx])) |
| else if(tokType == 0x05) /\* Token: DELTA \*/ |
| tmp\_str = parseToStr(get\_tok\_delta(decoded\_tokens[tokIdx], refIdx)) |
| else if(tokType == 0x06) /\* Token: DIGITS0 \*/ |
| tmp\_str = parseToStr(get\_tok\_digits0(decoded\_tokens[tokIdx])) |
| else if(tokType == 0x07) /\* Token: DELTA0 \*/ |
| tmp\_str = parseToStr(get\_tok\_delta0(decoded\_tokens[tokIdx], refIdx)) |
| else if(tokType == 0x08) /\* Token: MATCH \*/ |
| tmp\_str = parseToStr(get\_tok\_match(refIdx)) |
| return tmp\_str |
| } |

**get\_tok\_string(decoded\_tokens[ ])** pops and returns a null terminated string from data structure decoded\_tokens[ ] as described for token STRING in Table 39.

**get\_tok\_char(decoded\_tokens[ ])** pops and returns a one byte UTF-8 character from data structure decoded\_tokens[ ] as described for token CHAR in Table 39.

**get\_tok\_digits(decoded\_tokens[ ])** pops and returns a four bytes digit value from data structure decoded\_tokens[ ] as described for token DIGITS in Table 39.

**get\_tok\_delta(decoded\_tokens[ ], refIdx)** pops a one byte delta value from data structure encoded\_tokens[ ] as described for token DELTA in Table 39. It returns the sum of this delta value and the digit value of the co-located DIGITS token in the reference read identifier identified by refIdx.

**get\_tok\_digits0(decoded\_tokens[ ])** pops two values, one byte fixed-width length value and four bytes digit value from data structure decoded\_tokens[ ] as described for token DIGITS0 in Table 39. It returns 0-padded fixed-width representation of the digit value.

**get\_tok\_delta0(decoded\_tokens[ ], refIdx)** pops a one byte delta value from data structure decoded\_tokens[ ] as described for token DELTA in Table 39. It returns the 0-padded fixed-width representation of the sum of this delta value and the digit value of the co-located DIGITS0 token in the reference read identifier identified by refIdx.

**get\_tok\_match(refIdx)** returns the token value of the co-located token in the reference read identifier identified by refIdx as described for token MATCH in Table 39.

**parseToStr(value)** returns the string representation of the input value.

## Genomic descriptors

Input to the decoding processes described in this clause is the decoded\_symbols data structure specified in clause 13.2.6.2.1 indexed by the descriptor\_ID identifying each genomic descriptor.

Each descriptor\_ID is associated to K arrays decoded\_symbols[descriptor\_ID][N] with 0 ≤ N < K.

If K is greater than 1 the input to the decoding process of the genomic descriptor are K subsequences of values decoded\_symbols[descriptor\_ID][descriptor\_subsequence\_ID] as specified in clause 13.2.2.

The subsequences are identified as subsequenceN and values of the subsequences are read by means of indexes jM,N where M = descriptor\_ID and N = descriptor\_subsequence\_ID.

### pos

Input to this process is the array decoded\_symbols[descriptor\_ID][0] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 0 and the current value of j0,0.

Output of this process are number\_of\_alignments values used to set the mapping\_pos field in the MPEG-G record as specified in clause 11.3.2.

At the end of the decoding process j0,0 is incremented by number\_of\_alignments.

In the context of this decoding process the vector pos[] is defined as:

pos = decoded\_symbols[0][0]

The first decoded **pos** descriptor (pos[0]) in each AU is always equal to 0 as it specifiesthe offset of the left-most mapped Genomic Record carried by the AU from the AU\_start\_position field of the Access Unit Header, as specified in clause 7.5, which is the position of the left-most mapped Genomic Record itself.

#### Single alignment

If number\_of\_alignments is equal to 1, the values of **posi** (with i > 0) specifiesthe signed offset between the ith Genomic Record mapping position and the (i-1)th Genomic Record mapping position. Therefore the nth Genomic Record mapping position pn on the Reference Sequence (identified by **sequence\_ID** in the AU Header) shall be calculated as:

where *p0* is equal to the value of the AU\_start\_position field of the Access Unit Header, as specified in clause 7.5.

#### Multiple alignments

If number\_of\_alignments is greater than 1, the values of each **posk** (with 0 < k < number\_of\_alignments - 1) specifies the signed offset between the kth alignment mapping position and the (k-1)th alignment mapping position in the current Genomic Record. For k equal to 0, **pos0** specifies the signed offset between the primary alignment in the current Genomic Record and the primary alignment of the previous Genomic Record. Therefore the mapping position p0,n on the Reference Sequence (identified by **sequence\_ID** in the AU Header) of the primary alignment of the nth Genomic Record shall be calculated as:

where pos0,nis equal to the value of the first decoded **pos** descriptor for the nth Genomic Record.

The mapping positions pk,n on the Reference Sequence (identified by **sequence\_ID** in the AU Header) of the secondary alignments of the nth Genomic Record shall be calculated as:

where:

K = number\_of\_alignments – 1

posi,nis the ith decoded **pos** descriptor for the nth Genomic Record.

### rcomp

Input to this process is the array decoded\_symbols[descriptor\_ID][0] specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 1 and the current value of j1,0.

Output of this process are number\_of\_alignments values of decoded\_symbols[descriptor\_ID][0][j1,0] used to set the reverse\_comp field in the MPEG-G record as specified in clause 11.3.2.

At the end of the decoding process j1,0 is incremented by number\_of\_alignments.

Each decoded **rcomp** descriptor conveys information about the *strandedness* of a read. Each bit of a decoded **rcomp** descriptor is a flag indicating if the read is on the forward (bit set to 0) or reverse (bit set to 1) strand. Table 25 shows values and semantics of the **rcomp** descriptor for single and paired end reads.

Table 25. rcomp values and semantics.

|  |  |  |
| --- | --- | --- |
| **Value** | **Semantics** | |
| **Single read** | **Paired reads** |
| 0 | read on forward strand | both reads on forward strand |
| 1 | read on reverse strand | first read on forward strand, second on reverse strand |
| 2 | reserved | first read on reverse strand, second on forward strand |
| 3 | reserved | both reads on reverse strand |
| 4 .. 255 | reserved | reserved |

### flags

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 2 and the current value of j2,0.

Output of this process is the value decoded\_symbols[descriptor\_ID][0][j2,0] used to set the flags field in the MPEG-G record as specified in clause 11.3.2.

At the end of the decoding process j2,0 is incremented by 1.

The **flag** syntax element carries additional alignment information usually produced by aligners as specified in Table 26.

Table 26. Semantics for each bit of the flag descriptor.

|  |  |
| --- | --- |
| **bit position from LSB** | **Semantics** |
| 0 | read is PCR or optical duplicate |
| 1 | read fails platform/vendor quality checks |
| 2 | read mapped in proper pair |
| 3 .. 7 | reserved |

### mmpos

Input to this process are:

* two subsequences decoded\_symbols[descriptor\_ID][descriptor\_subsequence\_ID] as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 3 and descriptor\_subsequence\_ID are equal to 0 and 1 as specified in Table 55
* the current values of j3,0. and j3,1

Output of this process are

* the arrays read1\_offsets[] and read2\_offsets[] containing offsets of the mismatches in the sequence read or read pair computed as specified in clause 11.5 when class\_type is equal to 2, 3, 4, 5 or 6
* the number of offsets num\_offsets

At the end of the decoding process j3,0 and j3,1 are incremented by the number of decoded offsets.

In this description, subsequenceN is the subsequence identified by descriptor\_subsequence\_ID = N (i.e. subsequenceN = decoded\_symbols[3][N]).

Each value of subsequence0 signals if the corresponding value of subsequence1 is the last offset associated to the current Genomic Record. For each Genomic Record the process described in Table 27 shall stop when a bit set to 1 is found in subsequence0.

The offsets in case of single reads shall be calculated following the process described in Table 27.

Table 27. Determination of the offset of mismatches from the first mapped base of a Genomic Record in case of single reads.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| previous\_offset = 0  num\_offsets = 0  j = 0 |  |
| if(subsequence0[j3,0] == 0){ | if there is at least one mismatch in the current Genomic Record |
| do{ |  |
| read1\_offsets[j] = subsequence1[j3,1] + previous\_offset |  |
| previous\_offset = read1\_offsets[j] |  |
| j3,0++, j3,1++, j++ |  |
| }while(subsequence0[j3,0] == 0) | loop on subsequence0 until a 1 is found |
| num\_offsets = j |  |
| } |  |

Table 28. Determination of the offset of mismatches from the first mapped base of a Genomic Record in case of paired-end reads.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| previous\_offset = 0  j = 0 |  |
| if(subsequence0[j3,0] == 0){ |  |
| do{ |  |
| offset[j] = subsequence1[j3,1] + previous\_offset |  |
| if(previous\_offset < len1 && offset[j] ≥ len1) { | len1 is the number of mapped bases in read 1 in a read pair including insertions. |
| offset[j] = subsequence1 [j3,1] + previous\_offset - len1 | first mismatch of read 2 |
| first\_mm\_read2 = j |  |
| } |  |
| else{ |  |
| offset[j] = previous\_offset + subsequence1 [j3,1] |  |
| } |  |
| previous\_offset = offset[j] |  |
| j3,0++, j3,1++, j++ | increment read and write pointers |
| }while(subsequence0[j3,0] == 0) | loop on subsequence0 until a 1 is found |
| read1\_offsets = offsets[0, first\_mm\_read2) |  |
| read2\_offsets = offsets[first\_mm\_read2, j) |  |
| num\_offsets = j |  |
| } |  |

### mmtype

Input to this process are:

* three subsequences decoded\_symbols[descriptor\_ID][descriptor\_subsequence\_ID] as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 4 and descriptor\_subsequence\_ID are equal to 0, 1 and 2 as specified in Table 56
* the number of offset num\_offsets calculated for the current Genomic Record as specified in clause 11.4.4
* the current values of j4,0, j4,1 and j4,2

Output of this process are two vectors containing values identifying the type of edit operations to be performed on the sequence read or read pair computed as specified in clause 11.5 when class\_type is equal to 2, 3, 4, 5 or 6.

* the vector mismatch\_type[] contains values for the type of mismatch. 0 signals substitutions, 1 signals insertions and 2 signals deletions.
* the vector mismatches[] contains the symbols to be used for substitutions and insertions.

The two vectors shall be calculated following the process described in Table 29.

Table 29. Determination of the mismatch\_type[] and mismatches[] vectors from decoded\_symbols[4].

|  |  |
| --- | --- |
| **Decoding step** |  |
| previous\_offset = 0  j = i = k = 0 |  |
| while(j < num\_offsets){ |  |
| mismatch\_type[i] = subsequence0[j4,0] |  |
| if( subsequence0[j4,0] == 0 ) | It’s a substitution |
| mismatches[i] = Salphabet\_ID[subsequence1[j4,1]] | Salphabet\_ID as specified in clause 10.1 |
| } else { | It’s an indel |
| if (subsequence2[j4,2] == 5){ | deletion |
| mismatch\_type[i] = 2 |  |
| } |  |
| else if (alphabet\_ID == 0 || alphabet\_ID == 2){ | insert non-IUPAC |
| mismatches[i] = Salphabet\_ID [subsequence2[j4,2] – num\_symbols - 1] | num\_symbols is equal to the number of symbols of the alphabet identified by alphabet\_ID |
| } |  |
| else { | insert IUPAC |
| mismatches[i] = Salphabet\_ID [subsequence2[j4,2] – num\_symbols] | num\_symbols is equal to the number of symbols of the alphabet identified by alphabet\_ID |
| } |  |
| } |  |
| j4,0++, j4,1++, j4,2++, j++, i++ |  |
| } |  |

The types of edit operations associated to each value of mismatches\_type[] and mismatches[] are listed in Table 30.

Table 30. Determination of the mmtype descriptor values and semantics according to the used alphabet.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Value** | **Semantics** | | | |
|  | alphabet\_ID = 0 | alphabet\_ID = 1 | alphabet\_ID = 2 | alphabet\_ID = 3 |
| 0 | Substitute with ‘A’ | Substitute with ‘A’ | Substitute with ‘A’ | Substitute with ‘A’ |
| 1 | Substitute with ‘C’ | Substitute with ‘C’ | Substitute with ‘C’ | Substitute with ‘C’ |
| 2 | Substitute with ‘G’ | Substitute with ‘G’ | Substitute with ‘G’ | Substitute with ‘G’ |
| 3 | Substitute with ‘T’ | Substitute with ‘T’ | Substitute with ‘U’ | Substitute with ‘U’ |
| 4 | Substitute with ‘N’ | Substitute with ‘N’ | Substitute with ‘N’ | Substitute with ‘N’ |
| 5 | Deletion | Deletion | Deletion | Deletion |
| 6 | Insert ‘A’ | Substitute with ‘R’ | Insert ‘A’ | Substitute with ‘R’ |
| 7 | Insert ‘C’ | Substitute with ‘Y’ | Insert ‘C’ | Substitute with ‘Y’ |
| 8 | Insert ‘G’ | Substitute with ‘S’ | Insert ‘G’ | Substitute with ‘S’ |
| 9 | Insert ‘T’ | Substitute with ‘W’ | Insert ‘U’ | Substitute with ‘W’ |
| 10 | Insert ‘N’ | Substitute with ‘K’ | Insert ‘N’ | Substitute with ‘K’ |
| 11 | N/A | Substitute with ‘M’ | N/A | Substitute with ‘M’ |
| 12 | N/A | Substitute with ‘B’ | N/A | Substitute with ‘B’ |
| 13 | N/A | Substitute with ‘D’ | N/A | Substitute with ‘D’ |
| 14 | N/A | Substitute with ‘H’ | N/A | Substitute with ‘H’ |
| 15 | N/A | Substitute with ‘V’ | N/A | Substitute with ‘V’ |
| 16 | N/A | Substitute with ‘-’ | N/A | Substitute with ‘-’ |
| 17 | N/A | Insert ‘A’ | N/A | Insert ‘A’ |
| 18 | N/A | Insert ‘C’ | N/A | Insert ‘C’ |
| 19 | N/A | Insert ‘G’ | N/A | Insert ‘G’ |
| 20 | N/A | Insert ‘T’ | N/A | Insert ‘T’ |
| 21 | N/A | Insert ‘N’ | N/A | Insert ‘N’ |
| 22 | N/A | Insert ‘R’ | N/A | Insert ‘R’ |
| 23 | N/A | Insert ‘Y’ | N/A | Insert ‘Y’ |
| 24 | N/A | Insert ‘S’ | N/A | Insert ‘S’ |
| 25 | N/A | Insert ‘W’ | N/A | Insert ‘W’ |
| 26 | N/A | Insert ‘K’ | N/A | Insert ‘K’ |
| 27 | N/A | Insert ‘M’ | N/A | Insert ‘M’ |
| 28 | N/A | Insert ‘B’ | N/A | Insert ‘B’ |
| 29 | N/A | Insert ‘D’ | N/A | Insert ‘D’ |
| 30 | N/A | Insert ‘H’ | N/A | Insert ‘H’ |
| 31 | N/A | Insert ‘V’ | N/A | Insert ‘V’ |
| 32 | N/A | Insert ‘-’ | N/A | Insert ‘-’ |

### clips

Input to this process are:

* four subsequences decoded\_symbols[descriptor\_ID][descriptor\_subsequence\_ID] as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 5.
* number\_of\_template\_segments computed as specified in clause 11.3.2.
* the variable current\_record\_count is the number of processed Genomic Records.
* the current values of j5,0, j5,1, j5,2 and j5,3.

The four subsequences are identified by subsequences\_ID from 0 to 3 as specified in Table 54.

Output of this process is one bi-dimensional vector soft\_clips and a vector hard\_clips as specified in Table 31.

The decoding process of the clips descriptor is provided in Table 31 where:

* subsequenceN is the subsequence identified by descriptor\_subsequence\_ID = N
* subsequence0[j5,0] represents the next Genomic Record containing clipped bases
* subsequence1[j5,1] represent the type and position of clipped bases
* soft\_clips and hard\_clips are the output of this decoding process
  + soft\_clips[0] and soft\_clips[2] contain strings of charachters representing soft clips preceding the first mapped base of read1 and read2 respectively
  + soft\_clips[1] and soft\_clips[3] contain strings of charachters representing soft clips following the last mapped base of read1 and read2 respectively
  + hard\_clips[0] and hard\_clips[2] contain the number of hard clips preceding the first mapped base of read1 and read2 respectively
  + hard \_clips[1] and hard \_clips[3] contain the number of hard clips following the last mapped base of read1 and read2 respectively
* the semantics of subsequence1 is as follows:

|  |  |
| --- | --- |
| **subsequence1 values** | **semantics** |
| 0 | soft clips before start of read 1 |
| 1 | soft clips after  end of read 1 |
| 2 | soft clips before start of read 2 |
| 3 | soft clips after end of read 2 |
| 4 | hard clips before start of read 1 |
| 5 | hard clips after  end of read 1 |
| 6 | hard clips before start of read 2 |
| 7 | hard clips after end of read 2 |

Table 31. Decoding process of the clips descriptor.

|  |  |
| --- | --- |
| **Decoding process** | **Description** |
| j5,0, j5,1, j5,2, j5,3 | read pointers for the subsequences |
| if(current\_record\_count == subsequence0[j5,0]){ |  |
| do{ |  |
| if(subsequence1[j5,1]<= 3){ | soft clips |
| do{ |  |
| soft\_clips[subsequence1[j5,1]] [j5,2] = subsequence2[j5,2] | soft clipped base symbol |
| j5,2++ |  |
| } while(subsequence2[j5,2]!= 0xff OR subsequence2[j5,2]!= 0xfe) | continue reading symbols of clipped bases until a terminator is reached |
| } |  |
| else{ | hard clips |
| hard\_clips[subsequence1[j5,1]-4] = subsequence3[j5,3] | store the number of hard clips |
| j5,3++ | increment pointer for subsequence3 |
| } |  |
| j5,1++ | increment pointer for subsequence1 |
| } while(subsequence2[j5,2]!= 0xff OR subsequence3[j5,3] == 0xfe) |  |
| } |  |
| j5,0++ | increment pointer for subsequence0 |
| } |  |
| current\_record\_count++ |  |

more\_elements\_in\_subsequence0() returns TRUE until the last value of subsequence0 has been processed.

### ureads

Input to this process is the array decoded\_symbols[descriptor\_ID][0] structure as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 6 and the current value of j6,0.

Each **ureads** descriptor is a string of ASCII characters representing the sequence of nucleotides symbols belonging to the current alphabet as specified in clause 10.1.

NOTE: The length of each string shall be inferred either from **reads\_length** in the encoding parameters as specified in clause 7.3 in case of constant reads length (e.g. paired reads), or from the corresponding **rlen** syntax element (specified in clause 11.4.8).

### rlen

The **rlen** descriptor is used:

* in case of variable length reads when **reads\_length** = 0 in the Parameter Set (see clause 8)
* in presence of multiple alignments.

In case of variable reads length, the number of rlen descriptors associated to the current Genomic Record is equal to number\_of\_record\_segments.

Input to this process are:

* the array decoded\_symbols[descriptor\_ID][0] as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 7
* the value number\_of\_record\_segments computed as specified in clause 11.3.2
* the current value of j7,0

Output of this process are number\_of\_record\_segments values decoded\_symbols[descriptor\_ID][0][j] (with j7,0 ≤ j < j7,0 + number\_of\_record\_segments) used to set the number\_of\_record\_segments elements of the read\_len field in the MPEG-G record as specified in clause 11.3.2 step 5.j.

At the end of the decoding process j7,0 is incremented by number\_of\_record\_segments.

A decoded **rlen** descriptor represents the length of the current sequence read as number of bases minus one including soft clips.

### pair

Table 32 lists the possible decoding cases for the pair descriptor with the associated description.

Table 32. Specification of the decoding case for the pair descriptor.

|  |  |
| --- | --- |
| **Decoding case** | **Description** |
| same\_rec | read 1 and read 2 are encoded in the same genomic record |
| R1\_ split | read 1 in pair is on the same reference sequence but coded separately |
| R2\_split | read 2 in pair is on the same reference sequence but coded separately |
| R1\_diff\_ref\_seq | read 1 is on a different reference sequence |
| R2\_diff\_ref\_seq | read 2 is on a different reference sequence |
| R1\_unpaired | read 1 is unpaired |
| R2\_unpaired | read 2 is unpaired |
| more\_align | additional alignment present on another reference sequence |

Input to this process are:

* ten subsequences decoded\_symbols[descriptor\_ID][descriptor\_subsequence\_ID] as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 8. The description of each subsequence is provided in Table 53.
* the current values of j8,0, j8,1, j8,2, j8,3, j8,4, j8,5, j8,6, j8,7, j8,8 and j8,9

Output of this process is a variable segment\_flag set to

* 0 if both segments are coded in the Genomic Record
* 1 if only read 1 is coded in the Genomic Record
* 2 if only read 2 is coded in the Genomic Record
* 3 if, in case of multiple alignments, an additional alignment of read 1 is encoded in another AU

If subsequence0[j8,0] is less than 7 additional out shall include:

* a signed integer delta

OR

* an unsigned integers abs\_pos

OR

* two unsigned integers abs\_pos and seqID

In the following descriptions of the decoding process subsequenceN indicates the subsequence identified by descriptor\_subsequence\_ID equal to N.

The decoding process of one **pair** descriptor is shown in Table 33.

Table 33. Decoding process of the pair descriptor subsequences.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| if(subsequence0[j8,0] == 1){ | proper pair |
| isSignNegative = subsequence1[j8,1] & 0x0001; |  |
| delta = subsequence1[j8,1] >> 1; | -32767 ≤ delta ≤ 32767 |
| if (isSignNegative) delta = -delta; |  |
| segment\_flag = 0 |  |
| j8,1++ |  |
| } |  |
| else if (subsequence0[j8,0] == 2){ |  |
| abs\_pos = subsequence2[j8,2] | Absolute mapping position of read 1 on the same reference sequence. The maximum value is 2^ **max\_bits\_pos** -1 |
| segment\_flag = 2 |  |
| j8,2++ |  |
| } |  |
| else if (subsequence0[j8,0] == 3){ |  |
| abs\_pos = subsequence3[j8,3] | Absolute mapping position of read 2 on the same reference sequence. The maximum value is 2^ **max\_bits\_pos** -1 |
| segment\_flag = 1 |  |
| j8,3++ |  |
| } |  |
| else if (subsequence0[j8,0] == 4){ |  |
| seqID = subsequence4[j8,4] | Identifier of the reference sequence read 1 is mapped to. |
| abs\_pos = subsequence7[j8,7] | Absolute mapping position of read 1 on the reference sequence identified by subsequence4[j8,4]. The maximum value is 2^ **max\_bits\_pos** -1. |
| segment\_flag = 2 |  |
| j8,4++, j8,7++, |  |
| } |  |
| else if (subsequence0[j8,0] == 5){ |  |
| seqID = subsequence5[j8,5] | Identifier of the reference sequence read 2 is mapped to. |
| abs\_pos = subsequence8[j8,8] | Absolute mapping position of read 2 on the reference sequence identified by subsequence5[j8,5]. The maximum value is 2^ **max\_bits\_pos** -1. |
| segment\_flag = 1 |  |
| j8,5++, j8,8++, |  |
| } |  |
| else if (subsequence0[j8,0] == 6){ |  |
| seqID = subsequence6[j8,6] | Identifier of the reference sequence an additional alignment of read 1 is mapped to in case of multiple alignments. |
| abs\_pos = subsequence9[j8,9] | Absolute mapping position of an additional alignment of read 1 on the reference sequence identified by subsequence6[j8,6]. The maximum value is 2^ **max\_bits\_pos** -1. |
| j8,6++, j8,9++ |  |
| segment\_flag = 4 |  |
| } |  |
| else if (subsequence0[j8,0] == 7){ |  |
| segment\_flag = 1 | read 1 is unpaired |
| } |  |
| else if (subsequence0[j8,0] == 8){ |  |
| segment\_flag = 2 | read 2 is unpaired |
| } |  |
| j8,0++ |  |

### mscore

The **mscore** descriptor provides a score per alignment. It shall be used to represent mapping/alignment score per read generated by genomic sequence reads aligners.

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 9 and the current value of j9,0.

Output of this process is the value decoded\_symbols[descriptor\_ID][j9,0] to be interpreted as IEEE RFC 754 floating point value as specified below.

At the end of the decoding process j9,0 is incremented by 1.

In the scope of this document, aligment scores shall be expressed using an exponent and fractional part. The number of bits used to represent the exponent and the fractional part are specified in the encoding parameters (see clause 8). Figure 4 shows how this is specified in IEEE RFC 754 for an 11-bits exponent and a 52-bits fractional part.

The score of each alignment shall be represented by:

* One sign bit (S)
* 11 bits for the exponent (E)
* 53 bit for the mantissa (M)

|  |
| --- |
| 1 11 52  +-+-----------+----------------------------------------------------+  |S| Exp | Mantissa |  +-+-----------+----------------------------------------------------+  63 62 51 0 |

Figure 4. Alignment scores are expressed as 64-bit double precision floating point values.

The base (radix) to be used for the calculation of scores is 10, therefore:

score = -1s x 10E x M

### Descriptors for multiple alignments

The following descriptors are specified for the support of multiple alignments.

#### mmap

The **mmap** descriptor is used to signal on how many positions the read or the left-most read of a pair has been aligned and if spliced alignments are present. A Genomic Record containing multiple alignments is associated with one **mmap** descriptor.

Input to this process are:

* two subsequences decoded\_symbols[descriptor\_ID][descriptor\_subsequence\_ID] as specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 11 and descriptor\_subsequence\_ID are equal to 0 and 1 as specified in Table 57
* the current values of j11,0. and j11,1

Output of this process is the variable number\_of\_alignments, the variable number\_of\_record\_segments, a vector segments[] containing the number of segments per each alignment of the first read.

In the following clauses, subsequenceN is the subsequence identified by descriptor\_subsequence\_ID = N. iN is an array pointer used to read the elements of subsequenceN.

#### Multiple alignments without splices

If no splices are present in the Access Unit, **spliced\_reads\_flag** is unset.

##### Single reads

In case of single reads without splices

number\_of\_aligments = number\_of\_record\_segments = subsequence0[j11,0]

##### Paired-end reads

In paired-end sequencing without splices:

* subsequence0 contains one unsigned integer N per Genomic Record representing the number of alignments of the first read in the pair.
* subsequence1 contains N unsigned integers Mi, (0 ≤ i < N) representing the number of alignments of the second read associated to the ith alignment of the first read. This value is equal to the number of **pair** descriptors to be decoded for the ith alignment of the first read.
* The decoding process shown in Table 34 applies.

Table 34. Decoding process of mmap in case of paired-end reads without splices.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| number\_of\_aligments = subsequence0[j11,0] | total number of alignments of the first read |
| while(i < number\_of\_alignments) { |  |
| number\_of\_record\_segments++; |  |
| segments[i] = subsequence1[j11,1] | segments[i] is the number of alignments of the second read associated to the ith alignments of the first read. |
| } |  |

If N = number\_of\_aligments, and Mi = segments[i]then P = is the total number of alignments of the second read.

A special value of Mi ( = 0) indicates that the ith alignment of the first read is paired with an alignment of the second read which is already paired with a kth alignment of the first read with k < i (then there is no *new* alignment detected, which is consistent with the equation above).

When Mi is 0, the associated value of **pair** shall link to an existing second read alignment; a syntax error will be raised otherwise and the alignment considered broken. Clause 11.4.9 specifies that one decoding case the **pair** descriptor is specified for alignments of the first read belonging to other AUs ranges.

#### Multiple alignments with splices

If the dataset is encoded with spliced reads, the **msar** descriptor enables representation of splices length and strandedness as specified in clause 11.4.11.6.

Together with subsequence0 and subsequence1, input to this decoding process is a vector of decoded **rlen** descriptors associated to the current Genomic Record as specified in clause 11.4.8.

After having decoded the **mmap** and the **msar** descriptors, the decoder knows how many reads or read pairs have been encoded to represent the multiple mappings and how many segments are composing each read or read pair mapping. This is shown in Figure 5.

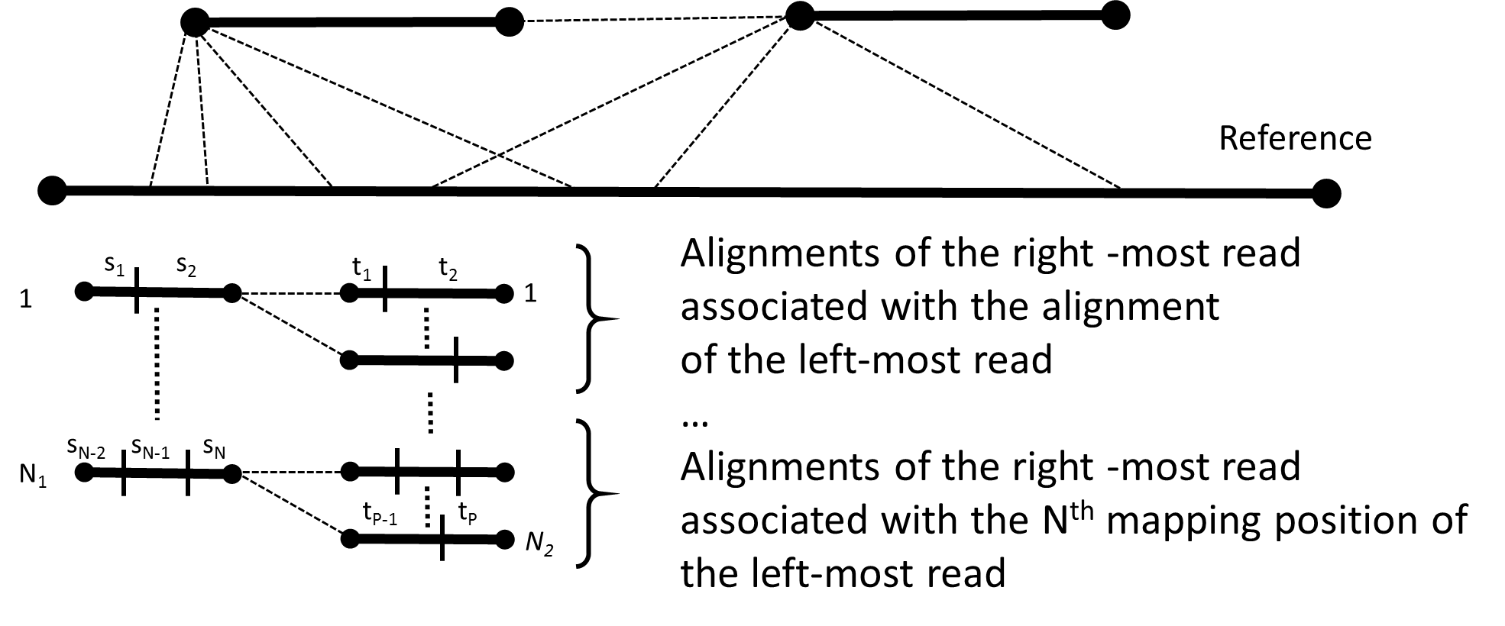


Figure 5 – Multiple alignments with splices.

##### Single reads

In case of single read sequencing with splices input to the process is the array decoded\_symbols[descriptor\_ID] where descriptor\_ID is equal to 7 (**rlen** descriptor) and the j7,0 index associated.

Output of this process are:

* the variables number\_of\_alignments, number\_of\_record\_segments and num\_pair
* the array segments[] containing the number of segments per each alignment

The decoding process shown in Table 35 applies

Table 35. Decoding process of mmap in case of single reads with splices.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| number\_of\_alignments = subsequence0[j11,0] | total number of alignments |
| number\_of\_record\_segments = 0 |  |
| i = 0 |  |
| while(i < number\_of\_alignments) { |  |
| number\_of\_record\_segments += subsequence1[j11,0] |  |
| segments[i] = subsequence1[j11,0] | segments[i] is the number of segments of the ith alignments. |
| i++ |  |
| j11,0 |  |
| } |  |

The number of **pair** descriptors num\_pair associated to the current Genomic Record can be calculated as

num\_pair = number\_of\_record\_segments – number\_of\_alignments + M0

where:

* M0 is the number of Mi with value 0
* num\_pair has to be incremented by 1 in case the special value for the **pair** descriptor indicating the more\_align coding case specified in clause 11.4.9 is present as first **pair** descriptors associated with the current Genomic Record (it indicates the presence of alignments in other AUs).

##### Paired-end reads

In case of paired-end sequencing with splices additional inputs to the process are:

* the array decoded\_symbols[descriptor\_ID] where descriptor\_ID is equal to 7 (**rlen** descriptor) and the j7,0 index associated.
* the array decoded\_symbols[descriptor\_ID] where descriptor\_ID is equal to 12 (**msar** descriptor) and the j12,0 index associated.
* four values for the number of soft clips in the array num\_sclips[] containing the lengths of the four elements of the bi-dimensiona array soft\_clips[] computed as specified in clause 11.4.6
* four values for the number of hard clips in the arrady hard\_clips[] computed as specified in clause 11.4.6

Output of this process are:

* the variables number\_of\_alignments, number\_of\_record\_segments and num\_pair
* the array segments[] containing the number of segments per each alignment

The decoding process is specified in Table 35.

Table 36. Decoding process of mmap in case of single reads with splices.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| number\_of\_read1\_segments = subsequence0[j11,0] | total number of alignments of the first read |
| number\_of\_record\_segments = number\_of\_read1\_segments |  |
| number\_of\_alignments = 1 |  |
| rlen = decoded\_symbols[7] |  |
| msar = decoded\_symbols[12] |  |
| segments[0] = 1 |  |
| i = 0 |  |
| len1 = num\_sclips[0] + num\_sclips[1] + hard\_clips[0] + hard\_clips[1] + rlen[j7,0] |  |
| j7,0++ |  |
| number\_of\_read1\_segments-- |  |
| while(len1 < read\_lengths) { | Check if the primary alignment of read 1 has splices |
| len1 += rlen[j7,0] |  |
| j7,0 ++ |  |
| segments[0]++ |  |
| number\_of\_read1\_segments-- |  |
| } |  |
| segments[0] += subsequence1[j11,1] | Splices in the primary alignment |
| number\_of\_record\_segments += subsequence1[j11,1] |  |
| number\_of\_read2\_segments += subsequence1[j11,1] |  |
| j11,1++ |  |
| i = 1 |  |
| while(number\_of\_read1\_segments > 0){ |  |
| number\_of\_alignments++ |  |
| segments[i] = count\_splices(msar[j12,0]) | count\_splices counts the number of splices in a **msar** descriptor according to the syntax specified in clause 14.20. |
| number\_of\_read1\_segments -= segments[i] |  |
| j12,0++ |  |
| i++ |  |
| } |  |
| i = 1 |  |
| while(i < number\_of\_alignments){ |  |
| number\_of\_read2\_segments += subsequence1[j11,1] |  |
| number\_of\_record\_segments += subsequence1[j11,1] |  |
| segments[i] = subsequence1[j11,1] | segments[i] is the number of segments of the ith alignments. |
| j11,1++ |  |
| i++ |  |
| } |  |
| i = 0 |  |
| while(i < number\_of\_read2\_segments){ |  |
| i += count\_splices(msar[j12,0]) |  |
| number\_of\_read2\_alignments++; |  |
| } |  |

The number of **pair** descriptors num\_pair shall be calculated as

num\_pair = number\_of\_record\_segments - number\_of\_read1\_alignments + M0

where M0 is the number of subsequence1[j11,1] values equal to 0

#### Multiple alignments on different sequences

It may happen that the alignment process finds alternative mappings to another reference sequence than the one where the primary mapping is positioned.

For read pairs that are uniquely aligned, a **pair** descriptor shall be used to represent the absolute read positions when there is for example a chimeric alignment with the mate on another chromosome. The **pair** descriptor shall be used to signal the reference and the position of the next record containing further alignments for the same template. The last record (e.g. the third if alternative mappings are coded in 3 different AUs) shall contain the reference and position of the first record.

#### Multiple alignments with insertions, deletions, unmapped portions

When an alternative secondary mapping does not preserve the contiguity of the reference region where the sequence is aligned, it may be impossible to reconstruct the exact mapping generated by the aligner because the actual sequence (and then the descriptors related to mismatches such as substitutions or indels) is only coded for the primary alignment. The **msar** descriptor shall be used to represent how secondary alignments map on the reference sequence in case they contain indels and/or soft clips. If **msar** is represented by the special symbol “\*” as specified in clause 11.4.11.6 for a secondary alignment, the decoder shall reconstruct the secondary alignment sequence from the primary alignment sequence and the secondary alignment mapping positions.

#### msar

The **msar** (Multiple Segments Alignment Record) descriptor supports spliced reads and alternative secondary alignments that contain indels or soft clips in case of class I data.

**msar** is intended to convey information related to secondary aligments on:

* a mapped segment length
* a different mapping contiguity (i.e. CIGAR string) for a secondary alignment and/or spliced read

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 11 and the current value of j11,0.

Each **msar** descriptor is a string of ASCII characters following the syntax of the extended CIGAR specified in clause 14.20 plus the additional symbol specified in Table 37.

Table 37. Special symbol used for the msar descriptor in addition to the syntax specified in clause 14.20.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Semantics** | **Description** |
| \* | The secondary alignment does not contain indels or soft clips | This is present when the reconstruction of a secondary alignment does not require any additional information than the alignment position and the primary alignment |

#### Multiple alignments strandedness

The **rcomp** descriptor specified in clause 11.4.2 shall be used to specify the strandedness of each segment of a primary alignment in a Genomc Record. One **rcomp** descriptor is present per segment; it is set to 0 when the segment is on the forward strand and it is set to 1 when the segment is mapped on the reverse strand.

The strandedness information for secondary alignments is conveyed by the **msar** descriptor. One **msar** descriptor is used per each read in a secondary alignment.

#### Scores of multiple alignments

In case of multiple alignments at least one **mscore** as specified in clause 11.4.10 shall be present per each alignment. The primary alignment is the one associated to the highest score.

The **mscore** descriptor allows signaling the mapping score of an alignment. In single reads sequencing it will have number\_of\_alignmens values per template; in paired-end sequencing it will have a value for each alignment of the entire template (number of different alignments of the first read possibly + the number of further second read alignments, i.e. when Mi - 1 > 0).

Number of scores = number\_of\_alignmens + M0

where M0 represent the total number of Mi = 0.

The number of scores associated to each alignment is signaled by the encoding parameter **as\_depth** as specified in clause 8.

### rtype

The **rtype** descriptor is used to signal the subset of descriptors used to encode one unmapped read or read pair (Class U) in a Genomic Record as shown in Table 38.

The **rtype** descriptor also enables mixing reference-based and reference-less compression in the same Dataset. In this scenario **rtype** = 0 signals reference based encoded records, while **rtype** > 0 signals the set of descriptors to be used for reference less compression(in this case descriptors refer to the computed reference, when needed).

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 12 and the current value of j12,0.

Output of this process is the invTransfSym value itself used by the decoder to select the appropriate descriptors for further decoding the Genomic Record.

Table 38. Semantics of the rtype descriptor.

|  |  |  |
| --- | --- | --- |
| **rtype** | **type of encoded reads** | **description** |
| not used | aligned reference based | the entire Dataset is encoded using reference based compression for mapped reads |
| 0 | reference based | the Dataset contains both read (pairs) encoded using reference based compression and reference less compression. Descriptors for this record use the external or embedded reference according to the Class of the AU |
| > 0 | raw and aligned reference less | 1 = class P descriptors used  2 = class N descriptors used  3 = class M descriptors used  4 = class I descriptors used  5 = class U descriptors used |

#### PushIn

When class U data are compressed using the “PushIn” computed reference algorithm specified in clause 12.3.1.2, the decoding process shall follow the one described for classes P, N, M, I in clauses 11.3.2 to 11.3.5 (for rtype values 1 to 4 respectively), or by ureads as described in clause 11.3.7 (rtype equal to 5). The process to be followed is indicated by the descriptor rtype as specified in clause 11.4.10.

The following table provides a description on the use of the **pos** and **pair** descriptors in this decoding process.

|  |  |
| --- | --- |
| **descriptor** | **semantics** |
| pos | Matching position of the read on the pushIn computed reference, with coordinate as described in clause 12.3.1.2 |
| pair | Reserved for future use. |

### rgroup

The **rgroup** descriptor identifies the read group the Genomic Record belongs to. It is an unsigned integer with values going from 0 to **num\_groups** - 1. The presence of read groups in an Access Unit is signaled by **num\_groups** > 0 in the Parameter Set as specified in clause 8.

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 13 and the current value of j13,0.

Output of this process is the invTransfSym value itself interpreted as the unsigned integer identifier of the read group the Genomic Record belongs to.

### QVIndex

The value of QVIndex specifies how to retrieve a reconstructed Quality Value in the Quality Value Codebook identified by the QVCodebookIdentifier descriptor. For each encoded Quality Value, one QVIndex value shall be present.

**QVIndex**[j] specifies the Quality Value at the jth genomic position after AU\_start\_position. Genomic positions not covered by the Access Unit do not increment j when decoding Quality Values.

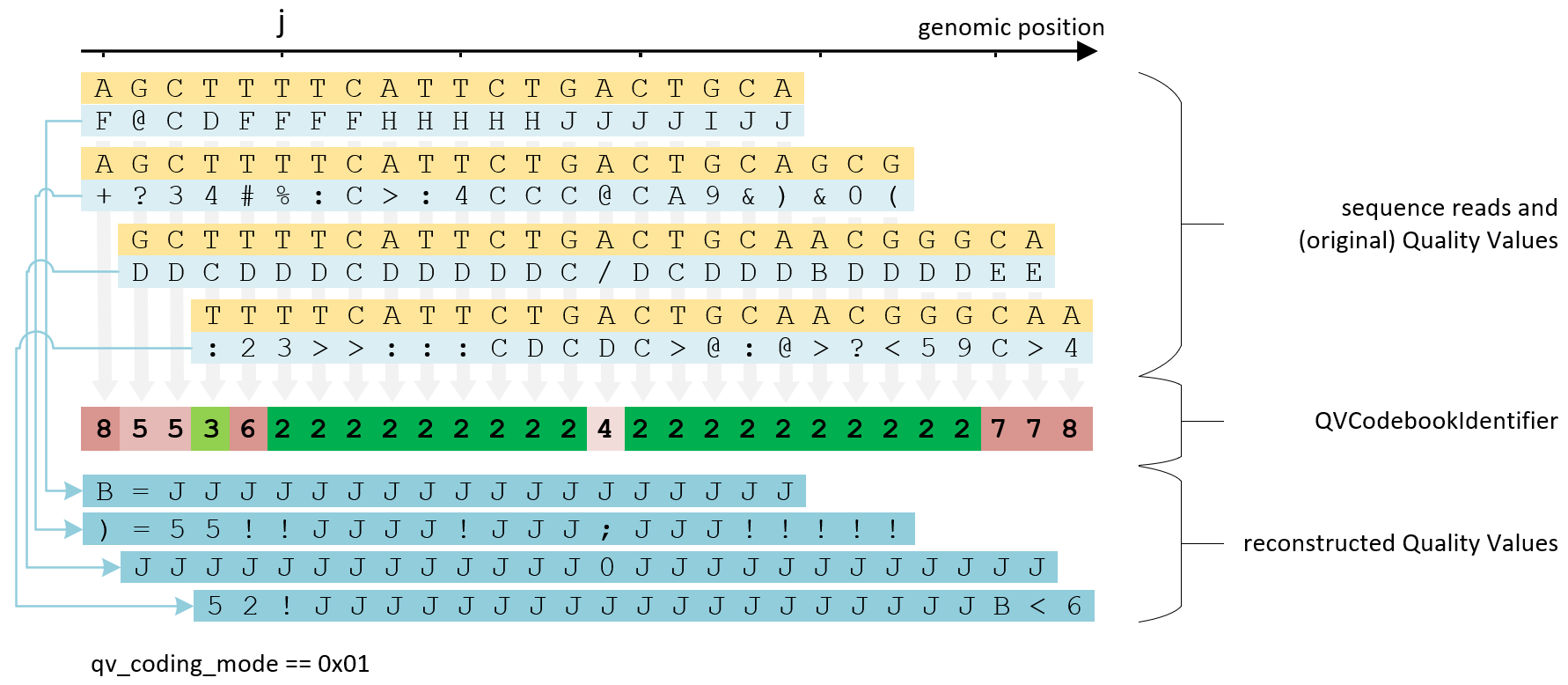
Clause 11.4.1 specifies how to calculate the absolute position of left-most mapped base in each read, and thus every Quality Value, in a read.

### QvCodebookIdentifier

The QVCodebookIdentifier descriptor shall be used to identify the Quality Value Codebook to reconstruct a Quality Value. Multiple Quality Value Codebooks can be used in one Access Unit.

Clause 11.4.1 specifies how to calculate the absolute mapping position of the left-most mapped base in each read, and thus every Quality Value, in a read.

**Figure 6** shows how codebook identifiers relate to sequence reads, Quality Values, reconstructed Quality Values, and genomic positions.



**Figure 6. Each genomic position is associated to a QVCodebookIdentifier descriptor. According to the corresponding QVIndex, the reconstructed Quality Value is read from the codebook identified by QVCodebookIdentifier.**

### token\_type

Sequence Read Identifiers are encoded as a sequence of **token\_type** descriptors which can be of three types:

* strings
* digits
* single characters

A read identifier is represented as set of differences and matches with respect to one of the previously decoded reads identifiers. The first identifier coded in an Access Unit always starts with a DIFF token followed by the value 0.

|  |  |  |  |
| --- | --- | --- | --- |
| **Descriptor\_ID** | **Descriptor** | **Semantics** | **Comments** |
| 16 | token\_type | The token type followed by parameters when necessary | See Table 39 for possible values and the related semantics. |

The **token\_type** descriptor can take the values listed in the table below. **token\_type** descriptors can possibly be followed by one or more parameters.

Table 39. Reads identifiers token\_type values and relates semantics.

|  |  |  |  |
| --- | --- | --- | --- |
| **Token Value** | **Token Name** | **Parameters** | **Semantics** |
| 0 | DUP | unsigned integer DISTANCE ranging from 0 to 2^32-1 | Indicates that this identifier is an exact duplicate of the identifier DISTANCE records ago, with “1” being the previously decoded identifier and counting backwards in the list of previously decoded identifiers. The value of DISTANCE shall always refer to a read identifier coded in the current Access Unit. If a DUP token is found no further tokens are required to decode the read identifier. DUP can only occur at the first token position. |
| 1 | DIFF | unsigned integer  DISTANCE ranging from 0 to 2^32-1 | Indicates which identifier this token is being compared against, usually “1” to indicate the previous identifier. DIFF can only occur at the first token position.  The first identifier of a coded Access Units always starts with “DIFF 0”. |
| 2 | STRING | st(v) | This is an arbitrary run of characters and need not be purely alphabetical. STRING is always null-terminated. |
| 3 | CHAR | c(1) | UTF-8 character as specified in ISO/IEC 10646. |
| 4 | DIGITS | unsigned integer ranging from 0 to 2^32-1 | a numerical value no more than 9 digits long and not starting with a leading zero |
| 5 | DDELTA | unsigned integer ranging from 0 to 2^8-1 | a numerical delta to a previous DIGITS value, between 0 and 255 |
| 6 | DIGITS0 | an 8-bit length and a 32-bit unsigned integer | a fixed-width numerical value no more than 8 digits long, possibly starting with a leading zero |
| 7 | DDELTA0 | 8-bit unsigned integer | a numerical delta to a previous DIGITS0 value. The same fixed length is assumed. |
| 8 | MATCH | none | The next token value is identical to the token at the same position in the read identifier the currently decoded read identifier is compared against (regardless of token type) |
| 9 | END | none | a marker to terminate the decoding of this identifier |

### rftp

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 17 and the current value of j17,0.

Output of this process is an array offset[] containing the offsets of the transformations to be applied to a decoded raw reference as specified in clause 12.3.1.1.

The decoding process for **rftp** is exactly the same as the one specified for **mmpos** in clause 11.4.4.

### rftt

Input to this process is the decoded\_symbols[descriptor\_ID] array specified in clause 13.2.6.2.1 when **descriptor\_ID** is equal to 18 and the current value of j18,0.

Output of this process is an array transf[] containing the type of transformations to be applied to a decoded raw reference as specified in clause 12.3.1.1.

The decoding process for **rftt** is exactly the same as the one specified for **mmtype** in clause 11.4.5.

## sequence

This section specifies how sequences of nucleotides are computed by a conformant decoder and stored in the sequence field of the MPEG-G record as specified in clause 14.12. For class HM, the mapped read is computed as specified in clause 11.5.1 while the unmapped read as specified in clause 11.5.2.

### Aligned reads (Classes P, N, M, I, HM)

Each sequence field sequence[i] (with 0≤ i < number\_of\_record\_segments) of the MPEG-G record specified in clause 14.12. is computed as follows:

1. The value mapping\_pos[0] is computed as specified in clause11.3.2 step 5.m;
2. The position pRef in the reference sequence identified by seq\_ID as specified in clause 14.4 is computed as follows:

pRef0 = mapping\_pos[0] - **seq\_start**

where **seq\_start** is specified in clause 7.2 for **sequence\_ID** equal to seq\_ID;

1. sequence[0] = sequence[seqId][ pRef0, pRef0+ read\_len[0]), where:
   1. sequence[seqId][i] is specified as in clause 7.2
   2. read\_len[0] is computed as specified in clause 11.3.2 step 5.j
2. if number\_of\_record\_segments > 1
   1. pRef1= mapping\_pos[0] + read\_len[0] + delta[0][0]

where delta[0][0] is as specified in clause11.3.2 step 5.r

* 1. sequence[1] = sequence[seqId][ pRef1, pRef1+ read\_len[1]), where:
     1. sequence[seqId] is specified as in clause 7.2
     2. read\_len[1] is computed as specified in clause 11.3.2 step 5.j

1. according to the value of the class\_type field calculated as specified in clause 11.3.2 the following applies:
   1. class\_type equal to 1: sequence[i] is completely decoded;
   2. class\_type equal to 2: sequence[i] is computed according to the process described in clause 11.3.3;
   3. class\_type equal to 3: sequence[i] is computed according to the process described in clause 11.3.4;
   4. class\_type equal to 4 sequence[i] is computed according to the process described in clause 11.3.5;
   5. class\_type equal to 5: sequence[i] is computed according to the process described in clause 11.3.6;

### Unmapped reads (Class HM, U)

In case of Genomic Records of class HM the mapped sequence is computed as specified in clause 11.5.1 and the unmappe sequence as specified in clause 11.4.7

In case of Genomic Records of class U the unmapped sequences of nucleotides are computed as specified in clause 11.4.7.

## e-cigar

Input to this process are:

* read\_len computed as specified in clause 11.3.2

For class\_type equal to 2, 3, and 4

* the read1\_offsets[] (in case of e-cigar for read 1) or read2\_offsets[] (in case of e-cigar for read 1) array computed as per clause 11.4.4 and the number of elements num\_offsets contained therein. In the scope of this decoding process such array will be referred to as offsets[].

For class\_type equal to 3, and 4

* the mismatches\_type[] and mismatches[]arrays computed as per clause 11.4.5

For class\_type equal to 4

* the soft\_clips[][] and the hard\_clips[]arrays computed as per clause 11.4.6

The output is an ASCII string ecigar\_string compliat to the syntax specified in clause 14.20.

The decoding process is shown in Table 40.

Table 40. Decoding process for the ecigar\_string field in the MPEG-G record.

|  |  |
| --- | --- |
| **Decoding step** | **Description** |
| ecigar\_string = ‘ ‘ | initialize to empty string |
| if(class\_type == 1){ | Class P |
| ecigar\_string = strcat(tostr(read\_len), ‘=’) |  |
| } |  |
| else if(class\_type == 2 ){ | Class N |
| ecigar\_string = strcat(tostr(offsets[0]), ‘=’) |  |
| ecigar\_string = strcat(ecigar\_string, ‘N’) |  |
| i = 1 |  |
| while(i < num\_offsets){ |  |
| delta = offsest[i] – offsets[i-1] |  |
| if(delta == 1){ |  |
| ecigar\_string = strcat(ecigar\_string, ‘N’) |  |
| } else { |  |
| ecigar\_string = strcat(ecigar\_string, strcat(tostr(delta), ‘=’)) |  |
| ecigar\_string = strcat(ecigar\_string, ‘N’) |  |
| } |  |
| i++ |  |
| } |  |
| delta = read\_len – offsets[i -1 ] |  |
| ecigar\_string = strcat(ecigar\_string, strcat(tostr(delta), ‘=’)) |  |
| } |  |
| else if(class\_type == 3){ | Class M |
| ecigar\_string = strcat(tostr(offset[0]), ‘=’) |  |
| ecigar\_string = strcat(ecigar\_string, mismatches [0]) |  |
| i = 1 |  |
| while(i < num\_offset){ |  |
| delta = offsets[i] – offsets[i-1] |  |
| if(delta == 1){ |  |
| ecigar\_string = strcat(ecigar\_string, mismatches[i])) |  |
| } else { |  |
| ecigar\_string = strcat(ecigar\_string, strcat(tostr(delta), ‘=’)) |  |
| ecigar\_string = strcat(ecigar\_string, mismatches [i]) |  |
| } |  |
| i++ |  |
| } |  |
| delta = read\_len – offsets[i -1 ] |  |
| ecigar\_string = strcat(ecigar\_string, strcat(tostr(delta), ‘=’)) |  |
| } |  |
| else if(class\_type == 4){ | Class I |
| if(soft\_clips[0] != 0) { |  |
| ecigar\_string = strcat(strcat(‘(‘, tostr(soft\_clips[0])),’)’) | soft clips are present before the left-most mapped base |
| } |  |
| else if(hard\_clips[0] != 0) |  |
| ecigar\_string = strcat(strcat(‘[‘, tostr(hard\_clips[0])),’]’) | hard clips are present before the left-most mapped base |
| previous\_offset = 0 |  |
| i = 0, j = 0 |  |
| while(i < num\_offset){ |  |
| count = 0 |  |
| delta = offset[i] – previous\_offset |  |
| if(delta == 1){ |  |
| if(mismatch\_type[i] == 0){ | substitution |
| ecigar\_string = strcat(ecigar\_string, mismatch[j])) |  |
| j++ |  |
| } |  |
| else if(mismatch\_type[i] == 1){ | insertion |
| while(mismatch\_type[i] == 1) |  |
| count++, i++ |  |
| ecigar\_string = strcat(ecigar\_string, tostr(count)) |  |
| ecigar\_string = strcat(ecigar\_string, ‘+’)) |  |
| j++ |  |
| } |  |
| else if(mismatch\_type[i] == 2){ | deletion |
| while(mismatch\_type[i] == 2) |  |
| count++, i++ |  |
| ecigar\_string = strcat(ecigar\_string, tostr(count)) |  |
| ecigar\_string = strcat(ecigar\_string, ‘-’)) |  |
| } |  |
| } else { |  |
| ecigar\_string = strcat(ecigar\_string, strcat(delta, ‘=’)) |  |
| ecigar\_string = strcat(ecigar\_string, substitutions[i]) |  |
| } |  |
| i++ |  |
| } |  |
| delta = read\_len – offset[i -1 ] – soft\_clips[1] – soft\_clips[0] |  |
| if(delta > 0) |  |
| ecigar\_string = strcat(ecigar\_string, strcat(delta, ‘=’)) |  |
| if(soft\_clips[1] != 0) { |  |
| ecigar\_string = strcat(strcat(‘(‘, tostr(soft\_clips[1])),’)’) | soft clips are present after the right-most mapped base |
| } |  |
| else if(hard\_clips[1] != 0) |  |
| ecigar\_string = strcat(strcat(‘[‘, tostr(hard\_clips[1])),’]’) | hard clips are present after the right-most mapped base |
| } |  |

# Coded representation of reference sequences

The reference sequence is usually part of an available reference genome (split into chromosomes and other sequences), but can in principle have any origin. With respect to a MPEG-G bitstream compliant with ISO/IEC 23092-1, the following types of reference sequences are supported:

* **External Reference**: the reference sequence is coded as an independent resource either locally or remotely and must be retrieved to enable the decoding of the bitstream.
* **Embedded Reference**: the reference sequence is coded within the bitstream as MPEG-G Dataset.
* **Computed Reference**: the reference sequence can be computed using the information contained in an MPEG-G Dataset encoding sequence reads.

In the scope of this document embedded and computed references are referred to as internal references.

## External Reference

The reference used for compression is not included in the bitstream. A normative mechanism for unique identification is specified in ISO/IEC 23092-1.

## Embedded Reference

The reference is stored in the bitstream as Dataset as specified in ISO/IEC 23092-1.

## Computed Reference

A Computed Reference is used:

* to encode aligned sequence reads without using the reference sequences used for alignment,
* to encode raw (unmapped) reads.

In case of aligned reads it can be beneficial to support encoding and decoding without requiring access to the reference sequences used for alignment.

This approach uses the sequence reads to be encoded to build a local consensus assembly to perform reference-based encoding. In this case all reads shall be encoded using class U descriptors, but the classification in P, N, M, I and HM classes shall be preserved.

When sequence reads are encoded using a Computed Reference, the **rtype** descriptor currently specified in clause 11.4.10 shall be used to:

1. signal the set of descriptors needed to decode the current record,
2. signal the type of reference (Embedded Reference or Computed Reference) needed to decode the current record.

### Reference computation algorithms

The following reference creation algorithms are supported. **cr\_alg\_ID** is specified in clause 8.2.

Table 40. Supported reference computation algorithms.

|  |  |  |
| --- | --- | --- |
| **cr\_alg\_ID** | Name | Description |
| 0 | NoComp | No reference is computed |
| 1 | RefTransform | To improve compression efficiency, an available reference can be modified before decoding sequence data. This is described in clause 12.3.1.1. |
| 2 | PushIn | The reference is created by simple concatenation of already decoded reads, with padding. This is described in clause 12.3.1.2. |
| 3 | Local Assembly | The reference is created by performing a local assembly. This algorithm applies only to aligned data as described in section 10.4. |

#### Reference transformation

To improve compression efficiency, an available reference sequence can be modified before decoding sequence data. The transformation consists of one or more substitutions at given positions to match the corresponding bases in reads covering those positions. For example, if several reads in a pileup present the same substitution (A instead of C present in the reference sequence) at position Pi, transforming the reference sequence by replacing the existing C at position Pi, with an A would reduce the number of mismatches in the reads to be encoded. An example is provided in Figure 7.

The descriptors listed below shall be used for reference transformation.

Table 41. Descriptors used to implement reference transformation.

|  |  |  |
| --- | --- | --- |
| **Descriptor** | **Semantics** | **Comments** |
| **rftp** | reference transformation position | position of difference between reference and contig used for prediction |
| **rftt** | reference transformation type | type of difference between reference and contig used for prediction. Same syntax as described for **mmtype** in section 11.4.5. |

##### rftp

Each decoded **rftp** descriptor is a 32-bit unsigned integer representing the absolute position of a reference transformation on the current Reference Sequence.

##### rftt

Each decoded **rftt** is an ASCII character representing the symbol to be replaced in the Reference Sequence at the position represented by the corresponding **rftp** descriptor.

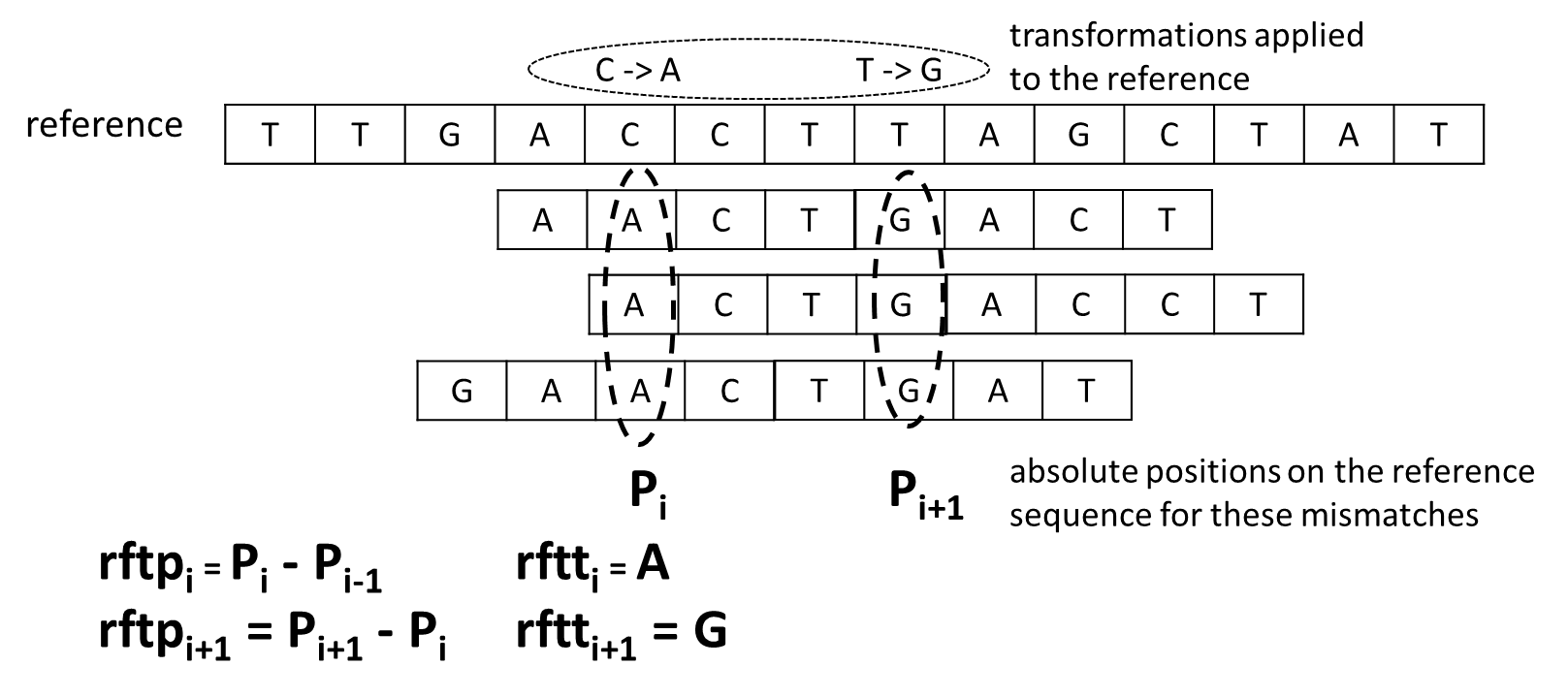


Figure 7. Use of rftp and rftt descriptors to implement reference transformation.

##### Decoding process

When **cr\_alg\_ID ==** 1 the decoder shall first apply the reference transformation to the Raw Reference structure received as input and then use it for reference-based decoding as specified in Clause 0.

#### PushIn

##### General

The reference is created by pushing into a buffer, i.e. concatenating, already decoded reads. The reference is built from crBufNumReads decoded reads, each composed by a sequence of symbols from one of the alphabets as specified in section 10.1.

A decoded read is pushed in front of the computed reference buffer only if it is different from the previous one. The computed reference obtained in this way is padded at its beginning and its end.

##### Process for the construction of the reference

Inputs to this process are:

* a buffer crBuf which contains crBufNumReads and which size in bytes is the variable crBufSize,
* an already decoded non-aligned read.

Output of this process is the updated buffer crBuf and the updated variable and crBufSize.

This process is executed only when the last decoded read is different from the previous last decoded read; it consists of the following steps:

1. If the variable crBufNumReads is greater than cr\_buf\_max\_reads (clause 8.2), oldest reads are removed from the buffer crBuf until crBufNumReads is smaller than cr\_buf\_max\_reads.
2. If (crBufSize + 2\*crPadSize + the size of the last decoded read) is greater than cr\_buf\_max\_size, oldest reads are pushed out of the buffer crBuf until (crBufSize + 2\*crPadSize + the size of the last decoded read) is smaller than or equal to cr\_buf\_max\_size.
3. The last read, decoded as described in clause 11.3.7, is pushed to buffer crBuf in front of the previous last decoded read (after the right-most crPadSize positions).
4. crPadSize right-most positions of crBuf are padded with the right-most base of the newly inserted read.
5. The whole buffer, except the left-most crPadSize positions, is pushed back until the left-most base of the oldest read is at crPadSize position.
6. crPadSize left-most positions of crBuf are padded with the left-most base of the oldest read remaining in crBuf.

The left-most position in the buffer shall have position 0; by consequence the left-most base of the oldest read shall have position crPadSize.

#### Local assembly

The reference is created by computing a local sliding consensus reference sequence (i.e. by performing a local assembly). This algorithm applies only to aligned data as described in section 12.3.1.3.1.

The list crBuf is built during the decoding process. A number of already decoded sequence reads may be needed and are stored in the list crBuf. The number of decoded sequence reads stored in the list crBuf is stored in the variable crBufNumReads. The current size in bytes of the list crBuf is stored in the variable crBufSize.

##### Process for adding a decoded aligned read to the list crBuf

Inputs to this process are:

* a list crBuf which contains crBufNumReads and which size in bytes is the variable crBufSize,
* an already decoded aligned read.

Output of this process is the updated list crBuf and the updated variables crBufNumReads and crBufSize.

This process consists of the following steps:

1. If the variable crBufNumReads is greater than cr\_buf\_max\_reads, oldest reads are removed from the list crBuf until crBufNumReads is smaller than cr\_buf\_max\_reads.
2. If the variable crBufSize plus the size of the already decoded aligned read is greater than cr\_buf\_max\_size, oldest reads are removed from the list crBuf until crBufSize plus the size of the already decoded aligned read is smaller than or equal to cr\_buf\_max\_size.
3. The last decoded read is added to the list crBuf as newest read.

##### Process for the construction of the reference

Input to this process is a list crBuf containing at least one aligned read.

Output of this process is a list ref (the reference) containing a sequence of consensus symbols.

For each position covered by aligned reads in the list crBuf, the consensus symbol is derived as follows:

1. Collect all symbols mapping to the current position.
2. Count the occurrences of each symbol.
3. If two symbols have the same maximum number of occurrences, then:
   1. If the alphabet as specifid in clause 10.1 contains the symbol “N”, set “N” as the current consensus symbol.
   2. Otherwise, select the symbol “A” as the current consensus symbol.
4. Otherwise, select the symbol with the maximum number of occurrences as consensus symbol.
5. Append the consensus symbol to the list ref.

##### Decoding process for rftp and rftt

When **cr\_alg\_ID** ==3, if the optional descriptors **rftp** and **rftt** are present in the bitstream, they shall be used to reconstruct the original reference used for sequence alignment. The decoder shall apply a transformation to the reference sequence constructed according to the process described in clause 12.3.1.3.2 by replacing the symbols present in the reference sequence at the absolute position represented by each **rfppi** descriptor with the symbols conveyed by each corresponding **rftti** descriptor.

# Parsing process

## General

Input to this process are bits from the Block Payload.

Outputs of this process are decoded descriptors.

The following variables are specified:

* ***symVal*** is the value of one of the symbols used to reconstruct a genomic descriptor as specified in clause 10 for read identifiers, clause 11 for sequence reads and clause 13 for Quality Values.
* ***cMax*** is the largest possible binarized value. Larger values are truncated.
* ***cLength*** is the number of bits used to represent the binarized value.

This process is invoked when the data type of a syntax element in the syntax tables is equal to ue(v), se(v) as specified in clause 13.2.1.3, or ae(v) as specified in clause 13.2.

## CABAC parsing process

This clause describes the parsing process of **encoded\_descriptor** syntax elements carried by a Block Payload as specified in clause 7.8.

### Binarizations

The process of binarization converts a non-binary-valued symbol representing a genomic descriptor value (e.g. a mapping position, a mapped read length or a mismatch type) into a binary code prior to arithmetic coding. The following clauses describe the decoding process for the different binarizations adopted in this document.

#### Binary (BI) Binarization

Input to this process are bits from the Block Payload.

Output of this process is the variable symVal.

The parameter cLength indicates the length in bits of the binarized symVal. The decoding process is described below:

|  |
| --- |
| symVal = 0  for (i=0; i<cLength; i++) {  symVal += read\_bits(1)^(cLength-i)  } |

Table 42 illustrates explicitly the assignment of bit strings to symVal values, with cLength set to the length of the bit string.

Table 42. Binary bit strings and corresponding symVal (informative).

|  |  |  |
| --- | --- | --- |
| Bit string | cLength | symVal |
| 0 | 1 | 0 |
| 1 | 1 | 1 |
| 0 0 1 | 3 | 1 |
| 0 1 0 | 3 | 2 |
| 1 1 | 2 | 3 |

For the binary binarization, the ctxIdx of each encoded symbol is determined based on the decoded MSB such that the ctxIdx is unique for each different symbol. Specifically, for binVal bj, ctxIdx is calculated using the following process:

|  |
| --- |
| updateIdx(j){  j0 = round(j/cLength); // starting bin position of current symbol  ctxIdx = 0; // starting ctxIdx for first bin  ctx\_num = 1; // total ctxIdx number of each bin  s = 0; // partially decoded MSB value  for (; j0<j; j0++) {  ctxIdx += ctx\_num;  ctx\_num <= 1;  s = s\*2+binVal[j0];  }  ctxIdx += s;  } |

#### Truncated Unary (TU) Binarization

Input to this process are bits from the Block Payload.

Output of this process is the variable symVal.

The parameter cMax indicates the maximum value of symVal. The decoding process is described below:

|  |
| --- |
| symVal=0; while(read\_bits(1)==1 && symVal < cMax) {  symVal++ } |

Table 43 illustrates explicitly the assignment of bit strings to symVal values, with cLength set to the length of the bit string.

Table 43. Binary bit strings and corresponding symVal with cMax equal to 3 (informative).

|  |  |
| --- | --- |
| Bit string | symVal |
| 0 | 0 |
| 1 0 | 1 |
| 1 1 0 | 2 |
| 1 1 1 | 3 |

#### Exponential Golomb (EG) Binarization

##### General

Inputs to this process are bits from the Block Payload.

Output of this process is the variable symVal.

The decoding process is described below:

|  |
| --- |
| leadingZeroBits= −1  for( b = 0; !b; leadingZeroBits++ )  b = read\_bits( 1 )  symVal = 2leadingZeroBits − 1 + read\_bits( leadingZeroBits ) |

The value returned from read\_bits( leadingZeroBits ) is interpreted as a binary representation of an unsigned integer with the most significant bits written first.

Table 44 illustrates the structure of the Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed as specified above for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 44. The "suffix" bits are those bits that are parsed in the computation of symVal and are shown as xi in Table 44, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each xi is equal to either 0 or 1.

Table 44. Bit strings with "prefix" and "suffix" bits and assignment to symVal ranges (informative).

|  |  |
| --- | --- |
| Bit string form | Range of symVal |
| 1 | 0 |
| 0 1 x0 | 1..2 |
| 0 0 1 x1 x0 | 3..6 |
| 0 0 0 1 x2 x1 x0 | 7..14 |
| 0 0 0 0 1 x3 x2 x1 x0 | 15..30 |
| 0 0 0 0 0 1 x4 x3 x2 x1 x0 | 31..62 |
| ... | ... |

Table 45 illustrates explicitly the assignment of bit strings to symVal values.

Table 45. Exp-Golomb bit strings and symVal in explicit form and used as ue(v) (informative).

|  |  |
| --- | --- |
| Bit string | symVal |
| 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 2 |
| 0 0 1 0 0 | 3 |
| 0 0 1 0 1 | 4 |
| 0 0 1 1 0 | 5 |
| 0 0 1 1 1 | 6 |
| 0 0 0 1 0 0 0 | 7 |
| 0 0 0 1 0 0 1 | 8 |
| 0 0 0 1 0 1 0 | 9 |
| ... | ... |

Depending on the descriptor, the value of a syntax element is derived as follows:

* If the syntax element is coded as ue(v), the value of the syntax element is equal to symVal,
* Otherwise, if the syntax element is coded as se(v), the value of the syntax element is derived by invoking the mapping process for signed Exp-Golomb codes as specified in clause 13.2.1.3 with symVal as input.

##### Signed Exponential Golomb (SEG) Binarization

Input to this process is the output of an Exponential Golomb Binarization as specified in clause 13.2.1.3.1.

Output of this process is the variable symVal.

symVal for input value k is calculated as (−1)k + 1 Ceil( k ÷ 2 ) An example is provided in Table 46.

Table 46. Example of decoding process for SEG binarization.

|  |  |
| --- | --- |
| Input | symVal |
| 0 | 0 |
| 1 | 1 |
| 2 | −1 |
| 3 | 2 |
| 4 | −2 |
| 5 | 3 |
| 6 | −3 |

#### Truncated Exponential Golomb (TEG) Binarization

Input to this process are bits from the Block Payload.

Output of this process is the variable symVal.

Truncated Exponential Golomb is a concatenation of a Truncated Unary binarization (with cMax equal to cTruncExpGolParam, with cTruncExpGolParam signaled as described in clause 13.2.2.2.1) and an Exponential Golomb representation. The parsing process for these syntax elements are processed as follows:

1. Perform the Truncated Unary decoding process with cMax=cTruncExpGolParam (see 13.2.1.2)
2. If the output of step 1 is equal to cTruncExpGolParam
   1. Perform the Exponential Golomb decoding process specified in clause 13.2.1.3.

symVal is equal to the sum of step 1 and step 2a.

Table 47 illustrates the bin strings of this Truncated Exponential Golomb binarization with cTruncExpGolParam equal to 2.

Table 47 Bin string of the Truncated Exponential Golomb binarization with cTruncExpGolParam=*2.*

|  |  |
| --- | --- |
| **Bit string** | **symVal** |
| 0 | 0 |
| 1 0 | 1 |
| 1 1 1 | 2 |
| 1 1 0 1 0 | 3 |
| 1 1 0 1 1 | 4 |

#### Signed Truncated Exponential Golomb (STEG) Binarization

Input to this process are bits from the Block Payload.

Output of this process is the variable symVal.

Signed Truncated Exponential Golomb is a concatenation of a Truncated Unary binarization (with cMax equal to cTruncExpGolParam, with cSignedTruncExpGolParam signaled as described in 13.2.2.2.1), an Exponential Golomb representation and a 1-bit Binary representation (flag). The decoding process for these syntax elements is as follows:

1. Perform the Truncated Unary decoding process with cMax=cSignedTruncExpGolParam (see 13.2.1.2)
2. If the output of step 1 is equal to cSignedTruncExpGolParam
   1. Perform the Exponential Golomb decoding process specified in clause 13.2.1.3
3. Else if the output of step 1 is not equal to 0
   1. Perform the Binary decoding process specified in clause 13.2.1.1 with cLength set to 1.

symVal is equal to the sum of the output values of step 1 and step 2a. If the output of step 3a is 1, symVal= -1\*symVal.

Table 50 illustrates the bin strings of this Signed Truncated Exponential Golomb binarization with cSignedTruncExpGolParam equal to 2.

Table 48 Bin string of the Signed Truncated Exponential Golomb binarization with cSignedTruncExpGolParam=*2.*

|  |  |
| --- | --- |
| **Bit string** | **symVal** |
| 1 1 0 1 1 1 1 | -4 |
| 1 1 0 1 0 1 | -3 |
| 1 1 1 1 | -2 |
| 1 0 1 | -1 |
| 0 | 0 |
| 1 0 0 | 1 |
| 1 1 1 0 | 2 |
| 1 1 0 1 0 0 | 3 |
| 1 1 0 1 1 0 | 4 |

#### Split Unit-wise Truncated Unary (SUTU) Binarization

Inputs to this process are bits from the Block Payload and parameters for this binarization process: splitUnitSize and outputSymSize, where outputSymSize must always be a multiple of splitUnitSize.

Output of this process is the variable symVal.

The SUTU binary string is a concatenation of n TU binarizations (clause 13.2.1.2), where n = outputSymSize / splitUnitSize. Each TU binarization constructs a splitUnitSize bits long portion of symVal. The cMax parameter for each split unit is specified as cMax = (1<<splitUnitSize) – 1.

The decoding process for SUTU binarization is described below:

|  |
| --- |
| symVal=0 cMax = (1<<splitUnitSize) – 1 for (i=0; i<outputSymSize; i+=splitUnitSize) {  unitVal = 0  while(read\_bits(1)==1 && unitVal < cMax)   unitVal++  symVal |= unitVal<<i } |

Table 49 illustrates the bin strings of Split Unit-wise Truncated Unary binarization with splitUnitSize = 2, outputSymbSize = 8.

Table 49 Bin string of the Split Unit-wise Truncated Unary binarization with splitUnitSize = 2, outputSymSize = 8.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **symVal** | **TU Instance 1** cMax==3 | | | **TU Instance 2** cMax==3 | | | **TU Instance 3** cMax==3 | | | **TU Instance 4** cMax==3 | | |
| 0 | 0 | - | - | 0 | - | - | 0 | - | - | 0 | - | - |
| 1 | 1 | 0 | - | 0 | - | - | 0 | - | - | 0 | - | - |
| 3 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 0 | - | - |
| 15 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 0 | - | - |
| 31 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | 0 | - | - |
| 63 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

#### Signed Split Unit-wise Truncated Unary (SSUTU) Binarization

Inputs to this process are bits from the Block Payload and parameters for this binarization process: splitUnitSize and outputSymSize, where outputSymSize must always be a multiple of splitUnitSize.

Output of this process is the variable symVal.

The SSUTU bin string is extension of the SUTU binarization (clause 13.2.1.6) with sign of symVal coded as a separate flag. The decoding process for this binarization is as follows:

1. The SUTU binarization produces the absolute value of symVal. The actual sign of symVal is calculated in the second step.
2. If symVal !=0, a one-bit flag equal to 1 (if symVal<0) or equal to 0 (if symVal>0) is added

Table 50 illustrates the bin strings of signed split unit-wise truncated unary binarization with splitUnitSize = 2, outputSymbSize = 8.

Table 50. Bin string of the Split Unit-wise Truncated Unary binarization with splitUnitSize = 2, outputSymSize = 8.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **symVal** | **TU Instance 1** cMax==3 | | | **TU Instance 2** cMax==3 | | | **TU Instance 3** cMax==3 | | | **TU Instance 4** cMax==3 | | | **Sign** |
| -63 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 1 |
| -31 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | 0 | - | - | 1 |
| -15 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 1 |
| -3 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 0 | - | - | 1 |
| -1 | 1 | 0 | - | 0 | - | - | 0 | - | - | 0 | - | - | 1 |
| 0 | 0 | - | - | 0 | - | - | 0 | - | - | 0 | - | - | - |
| 1 | 1 | 0 | - | 0 | - | - | 0 | - | - | 0 | - | - | 0 |
| 3 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 0 | - | - | 0 |
| 15 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 0 |
| 31 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | 0 | - | - | 0 |
| 63 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 0 |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

#### Double Truncated Unary (DTU) Binarization

Inputs to this process are bits from the Block Payload and parameters for this binarization process: cMax, splitUnitSize and outputSymSize.

Output of this process is the variable symVal.

The DTU bin string is a concatenation of two binarizations, namely TU binarization (clause 13.2.1.2) and SUTU binarization (clause 13.2.1.6). The parameter cMax is used for TU binarization, and parameters splitUnitSize and outputSymSize are used for a SUTU binarization (where its cMax is derived internally).

|  |
| --- |
| if(cMax > 0) {  symVal = decode\_cabac\_TU(cMax)  if(symVal >= cMax)  symVal += decode\_cabac\_SUTU(splitUnitSize, outputSymSize)  else  symVal = decode\_cabac\_SUTU(splitUnitSize, outputSymSize)  } |

**decode\_cabac\_TU** specifies the decoding process specified in clause 13.2.1.2.

**decode\_cabac\_SUTU** specifies the decoding process specified in clause 13.2.1.6.

Table 51 illustrates the bin strings of the Double Truncated Unary binarization with cMax = 1, splitUnitSize = 2, outputSymSize = 8.

Table 51. Bin string of the Double Truncated Unary binarization with cMax = 1, splitUnitSize = 2, outputSymSize = 8.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **symVal** | **TU Instance** cMax=1 | **SUTU Instance: splitUnitSize = 2, outputUnitSize = 8** | | | | | | | | | | | |
| **TU Instance 1** cMax=3 | | | **TU Instance 2** cMax=3 | | | **TU Instance 3** cMax=3 | | | **TU Instance 4** cMax=3 | | |
| 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - |
| 1 | 1 | 0 | - | - | 0 | - | - | 0 | - | - | 0 | - | - |
| 3 | 1 | 1 | 1 | 0 | 0 | - | - | 0 | - | - | 0 | - | - |
| 15 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | - | - | 0 | - | - |
| 31 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | - | 0 | - | - |
| 63 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

#### Signed Double Truncated Unary (SDTU) Binarization

Inputs to this process are bits from the Block Payload and parameters for this binarization process: cMax, splitUnitSize and outputSymSize, where outputSymSize must always be a multiple of splitUnitSize.

Output of this process is the variable symVal.

The SDTU bin string is an extension of the DTU binarization with sign of symVal coded as a flag. It is obtained as follows:

1. The DTU binarization produces the absolute value of symVal. The actual sign of symVal is calculated in the second step.
2. If symVal !=0, a one-bit flag equal to 1 (if symVal<0) or equal to 0 (if symVal>0).

Table 52 illustrates the bin strings of the Double Truncated Unary binarization with cMax = 1, splitUnitSize = 2, outputSymSize = 8.

Table 52. Bin string of the Signed Double Truncated Unary binarization with cMax = 1, splitUnitSize = 2, outputSymSize = 8.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **symVal** | **TU Instance** cMax=1 | **SUTU Instance: splitUnitSize = 2, outputUnitSize = 8** | | | | | | | | | | | | **Sign** |
| **TU Instance 1** cMax=3 | | | **TU Instance 2** cMax=3 | | | **TU Instance 3** cMax=3 | | | **TU Instance 4** cMax=3 | | |
| -63 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 1 |
| -31 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | - | 0 | - | - | 1 |
| -15 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 1 |
| -3 | 1 | 1 | 1 | 0 | 0 | - | - | 0 | - | - | 0 | - | - | 1 |
| -1 | 1 | 0 | - | - | 0 | - | - | 0 | - | - | 0 | - | - | 1 |
| 0 | 0 | - | - | - | - | - | - | - | - | - | - | - | - | - |
| 1 | 1 | -0 | - | - | 0 | - | - | 0 | - | - | 0 | - | - | 0 |
| 3 | 1 | 1 | 1 | 0 | 0 | - | - | 0 | - | - | 0 | - | - | 0 |
| 15 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | - | - | 0 | - | - | 0 |
| 31 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | - | 0 | - | - | 0 |
| 63 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | - | - | 0 |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

### Decoder Configuration

The decoder configuration syntax is specified below.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| decoder\_configuration(encoding\_mode\_ID){ |  |
| **dependency\_mask** | u(16) |
| if (encoding\_mode\_ID == 0){ /\* CABAC \*/ |  |
| **num\_descriptor\_subsequences\_minus1** | u(8) |
| for (i = 0; i <=num\_descriptor\_subsequences\_minus1; i++){ |  |
| **descriptor\_subsequence\_ID** | u(v) |
| support\_values() | As specified in 13.2.2.1 |
| cabac\_binarizations() | As specified in 13.2.2.2 |
| transformation\_parameters() | As specified in 13.2.2.3 |
| } |  |
| } else if(encode\_mode\_ID >= 1){ |  |
| **/**\* reserved for future use \*/ |  |
| } |  |
| } |  |

**dependency\_mask** signals which dependencies are enabled as specified in

|  |  |
| --- | --- |
| **dependency\_mask** | **Semantics** |
| 0x0001 | The dependency specified in 13.2.6.2.2 is enabled |
| all other bits | reserved for future use |

**num\_descriptor\_subsequences\_minus1** specifies the number of subsequences the genomic descriptor has been split into for encoding minus 1

**descriptor\_subsequence\_ID[i]** indicates the id of the ith subsequence. This table is only available if the descriptor sequence consists of multiple subsequences (for **mmpos**, **mmtype**, **clips,** and **pair** descriptors as specified in clauses 11.4.4, 11.4.5, 11.4.6 and 11.4.9 respectively). This id is used to signal which binarizations, transformations and contexts will need to be used to decode a specific subsequence. The descriptor\_subsequence\_ID is represented as u(ceil(log2(num\_descriptor\_subsequences\_minus1)).

**descriptor\_subsequence\_ID** can only contain values as indicated in Table 53 to Table 56 (**mmpos**, **mmtype**, **clips,** and **pair** descriptors as specified in clauses 11.4.4, 11.4.5, 11.4.6 and 11.4.9 respectively). If subsequences are not identified in the descriptor\_subsequence\_ID array, they are assumed to be empty.

Table 53: Subsequences for descriptor\_ID = 9 (pair descriptor).

|  |  |
| --- | --- |
| **subsequence\_ID** | **Semantics** |
| 0 | Sequence of unsigned integers identifying:   * the next subsequence symbol required for the decoding process when values range from 1 to 6. * R1\_unpaired decoding case as specified in 11.4.9 when the value is equal to 7. * R2\_unpaired decoding case as specified in 11.4.9 when the value is equal to 8. |
| 1 | same\_rec decoding case as specified in 11.4.9. Sequence of signed integers representing the distance between the mapping position of read 1 and the mapping position of read 2 on the reference sequence. The value is comprised between -32767 and 32767. |
| 2 | R1\_split decoding case as specified in 11.4.9. Sequence of unsigned integers representing the position of read 1 on the reference sequence. The maximum value can be 2^ **max\_bits\_pos** -1. |
| 3 | R2\_split decoding case as specified in11.4.9. Sequence of unsigned integers representing the position of read 2 on the reference sequence. The maximum value can be 2^ **max\_bits\_pos** -1. |
| 4 | R1\_diff\_ref\_seq decoding case as specified in 11.4.9. Sequence of unsigned integers representing the identifier of the reference sequence read 1 is mapped to. The maximum value is 2^16-1. |
| 5 | R2\_diff\_ref\_seq decoding case as specified in 11.4.9. Sequence of unsigned integers representing the identifier of the reference sequence read 2 is mapped to. The maximum value is 2^16-1. |
| 6 | more\_align decoding case as specified in 11.4.9. Sequence of unsigned integers representing the identifier of the reference sequence a secondary alignment of read 1 is mapped to. The maximum value is 2^16-1. |
| 7 | R1\_diff\_ref\_seq decoding case as specified in 11.4.9. Sequence of unsigned integers representing the position of read 1 on the reference sequence. The maximum value is 2^ **max\_bits\_pos** -1. |
| 8 | R2\_diff\_ref\_seq decoding case as specified in 11.4.9. Sequence of unsigned integers representing the position of read 2 on the reference sequence. The maximum value is 2^ **max\_bits\_pos** -1. |
| 9 | more\_align decoding case as specified in 11.4.9. Sequence of unsigned integers representing a secondary alignment mapping position of read 1 on the reference sequence. The maximum value is 2^32-1. |

Table 54: Subsequences for descriptor\_ID = 6 (clips descriptor)

|  |  |  |
| --- | --- | --- |
| **subsequence\_ID** | **Semantics** | **Value** |
| 0 | Record identifier | uint |
| 1 | Position flag | uint |
| 2 | Nucleotides strings with terminators | ASCII string terminated by a reserved value as specified in Table 31 |
| 3 | Hard clips length | uint |

Table 55: Subsequences for descriptor\_ID = 3 (mmpos descriptor)

|  |  |  |
| --- | --- | --- |
| **subsequence\_ID** | **Semantics** | **Subsequence representation** |
| 0 | Terminator flag | 1 bit |
| 1 | Position value | Unsigned integer with max value equal to 2^(terminator\_size\_minus1 + 1) - 1 |

Table 56: Subsequences for descriptor\_ID = 4 (mmptype descriptor)

|  |  |  |
| --- | --- | --- |
| **subsequence\_ID** | **Semantics** | **Subsequence representation** |
| 0 | Symbol type flag | 1 bit |
| 1 | Substitution type | uint |
| 2 | Indels | uint |

Table 57: Subsequences for descriptor\_ID = 11 (mmap descriptor)

|  |  |  |
| --- | --- | --- |
| **subsequence\_ID** | **Semantics** | **Subsequence representation** |
| 0 | First read alignments | uint |
| 1 | Second read alignments | uint |

**support\_values()** specifies the parsing of a set of configuration parameters used to parse the descriptor subsequence identified by **descriptor\_subsequence\_ID**. It is described in clause 13.2.2.1.

**cabac\_binarizations()** specifies the parsing of information about the binarization used for CABAC coding of the descriptor subsequence identified by **descriptor\_subsequence\_ID**. It is described in clause 13.2.2.2.

**transform\_values()** specifies the parsing of transformation parameters for the descriptors subsequence identified by **descriptor\_subsequence\_ID**. It is specified in clause 13.2.2.3.

#### Support values

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| support\_values(){ |  |
| **output\_symbol\_size** | u(5) |
| **coding\_symbol\_size[]** | u(5) |
| **coding\_order** | u(2) |
| **num\_output\_symbols** | u(32) |
| } |  |

**output\_symbol\_size** represents the size in bits of each reconstructed symbol of the descriptor subsequence.

**coding\_symbol\_size** signals the length of each symbol in the encoded bitstream during decoding.

**coding\_order** signals the number previously decoded symbols internally maintained as state variables and may be used in decoding the next symbol. The default value is 0.

**num\_output\_symbols** signals the number of symbols encoded in the subsequence and to be reconstructued by the decoding process.

#### CABAC binarizations

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| cabac\_binarizations(){ |  |
| **binarization\_ID** | u(5) |
| **bypass\_flag** | u(1) |
| cabac\_binarization\_parameters(binarization\_ID) | 13.2.2.2.1 |
| if(!**bypass\_flag**){ |  |
| cabac\_context\_parameters(descriptor\_subsequence\_ID) | 13.2.2.2.2 |
| } |  |
| } |  |

**binarization\_ID** indicates the binarization method to be used for decoding. The list of binarizations is shown in Table 58.

**bypass\_flag** if equal to 1, all bins of the binarization are decoded using bypass mode.

Table 58: Values of binarization\_ID and associated binarizations

|  |  |
| --- | --- |
| **binarization\_ID** | **Type of binarization** |
| 0 | Binary Coding as specified in clause 13.2.1.1 |
| 1 | Truncated Unary as specified in clause 13.2.1.2 |
| 2 | Exponential Golomb as specified in clause 13.2.1.3 |
| 3 | Signed Exponential Gomb as specified in clause 13.2.1.3.2 |
| 4 | Truncated Exponential Golomb as specified in clause 13.2.1.4 |
| 5 | Signed Truncated Exponential Golomb as specified in clause 13.2.1.5 |
| 6 | Split Unit-wise Truncated Unary as specified in clause 13.2.1.6 |
| 7 | Signed Split Unit-wise Truncated Unary as specified in clause 13.2.1.7 |
| 8 | Double Truncated Unary as specified in clause in 13.2.1.8 |
| 9 | Signed Double Truncated Unary as specified in clause in 13.2.1.9 |
| 10 .. 31 | Reserved for future use. |

##### CABAC binarizations parameters

The **cabac\_binarization\_parameters** data structure contains the parameters used for binarization for the descriptor subsequence identified by **descriptor\_subsequence\_ID**. **binarization\_ID** is specified in clause 13.2.2.2.

Table 59. CABAC binarization parameters.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| binarization\_parameters(binarization\_ID){ |  |
| if(binarization\_ID==0){ |  |
| **cLength** | u(5) |
| } else if ( binarization\_ID==1 || binarization\_ID==8 || binarization\_ID==9){ |  |
| **cMax** | u(8) |
| } else if (binarization\_ID==4){ |  |
| **cTruncExpGolParam** | u(5) |
| } else if (binarization\_ID==5){ |  |
| **signedTruncExpGolParam** | u(5) |
| } |  |
| if (binarization\_ID==6 || binarization\_ID==7 || binarization\_ID==8 || binarization\_ID==9){ |  |
| **splitUnitSize** | u(5) |
| **outputSymSize** | u(5) |
| } |  |
| } |  |

**cMax** represents the cMax variable specified in clause 13.2.1.2.

**cLength** represents the cLength variable specified in clause 13.2.1.1.

**truncExpGolParam** represents the truncExpGolParam variable specified in clause 13.2.1.4.

**signedTruncExpGolParam** represents the signedTruncExpGolParam variable specified in clause 13.2.1.5.

**splitUnitSize** represents the splitUnitSize variable as specified in clause 13.2.1.6.

**outputSymSize** represents the outputSymSizevariable as specified in clause 13.2.1.6.

##### CABAC context parameters

The **cabac\_context\_parameters** data structure contains the parameters used for context value adaptation and initialization for the descriptor subsequence identified by **descriptor\_subsequence\_ID** specified in clause 13.2.2.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| cabac\_context\_parameters(){ |  |
| **num\_contexts** | u(16) |
| **adaptive\_mode\_flag** | u(1) |
| **default\_context\_init\_flag** | u(1) |
| if(!default\_context\_init\_flag){ |  |
| for (i=0; i<num\_contexts; i++){ |  |
| **context\_initialization\_value[i]** | u(7) |
| } |  |
| } |  |
| if(coding\_order\_> 0) { |  |
| **context\_offset\_luts** | u(16) |
| for (i=0; i<coding\_order; i++){ |  |
| **context\_offset\_value[i]** | u(16) |
| } |  |
| } |  |
| } |  |

**num\_contexts** specifies the size of the table ctxTable[ ] specified in 13.2.4 containing the list of context values needed for the decoding of the descriptor subsequence (including any LUTs).

**adaptive\_mode\_flag** set to 1 the arithmetic decoding engine specified in clause 13.2.5uses contexts adaptation otherwise contexts adaptation is disabled.

**default\_context\_init\_flag** if set to 1 the contexts are initialized with a default value of 64 (equiprobability) otherwise contexts are initialized with values specified in **context\_initialization\_value**.

**context\_offset\_luts** specifies the number of contexts used for decoding of LUTs present in the descriptor subsequence. The range of context variables in ctxTable[ ] for LUTs is [0, context\_offset\_luts) and the range for context variables in ctxTable[ ] for descriptor tokens is [context\_offset\_luts, num\_contexts).

**context\_initialization\_value[i]** specifies the initialization values for the ith context variable.

**context\_offset\_value[i]** specifies the list of offset values needed to select the contexts (described in 13.2.6.2.5) when decoding the descriptor subsequence with a coding\_order > 0, where coding\_order is signaled in support\_values() specified in 13.2.2.1.

#### Transformation Parameters

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| transformation\_parameters(descriptor\_subsequence\_ID){ |  |
| **transform\_count** | u(4) |
| for(i = 0; i < transform\_count; i++){ |  |
| **transform\_ID** | u(8) |
| if(transform\_ID == match\_coding\_ID){ |  |
| **buffer\_size\_in\_bytes** | u(16) |
| **pointer\_subsequence\_ID** | u(v) |
| **length\_subsequence\_ID** | u(v) |
| **raw\_value\_subsequence\_ID** | u(v) |
| } else if(transform\_ID == equality\_ID) { |  |
| **equality\_flag\_subsequence\_ID** | u(v) |
| **equality\_values\_subsequence\_ID** | u(v) |
| } else if(transform\_ID == scaling\_ID) { |  |
| **scaling\_opcode\_ID** | u(8) |
| if(scaling\_opcode\_ID > 0x0) |  |
| **scaling\_value** | u(16) |
| } |  |
| } |  |
| } |  |

**transform\_count** signals the number of transformations that are applied to the descriptor subsequence identified with descriptor\_subsequence\_ID.

**transform\_ID** signals the applied transformation according to Table 60.

Table 60. Values of transformation\_ID and associated transformations.

|  |  |
| --- | --- |
| **transform\_ID** | **name** |
| 0 | match\_coding\_ID |
| 1 | equality\_ID |
| 2 | scaling\_ID |
| 3 | fixed\_transform\_ID |
| 4 | diff\_coding |
| 5 .. 255 | Reserved for future use |

**pointer\_subsequence\_ID** is the subsequence\_ID of the subsequence that contains all pointer information for the match\_coding transformation process. The type of the pointer\_subsequence\_ID syntax element is equal to the type of the descriptor\_subsequence\_ID in clause 13.2.2.

**length\_subsequence\_ID** is the subsequence\_ID of the subsequence that contains all length information for the match\_coding transformation process. The type of the length\_subsequence\_ID syntax element is equal to the type of the descriptor\_subsequence\_ID in clause 13.2.2.

**raw\_value\_subsequence\_ID** is the subsequence\_ID of the subsequence that contains all raw values for the match\_coding transformation process. The type of the raw\_value\_subsequence\_ID syntax element is equal to the type of the descriptor\_subsequence\_ID in clause 13.2.2.

**equality\_flag\_subsequence\_ID** is the subsequence\_ID of the subsequence that contains all flag information for the equality transformation process. The type of the equality\_flag\_subsequence\_ID syntax element is equal to the type of the descriptor\_subsequence\_ID in clause 13.2.2.

**equality\_values\_subsequence\_ID** is the subsequence\_ID of the subsequence that contains all value information for the equality transformation process. The type of the equality\_values\_subsequence\_ID syntax element is equal to the type of the descriptor\_subsequence\_ID in clause 13.2.2.

**scaling\_opcode\_ID** indicates the ID of the opcode which specifies how the decoded symbol may be scaled with **scaling\_value**. The list of support opcodes and their operation is specified in Table 61.

Table 61: Description of scaling operations as transformations

|  |  |  |
| --- | --- | --- |
| **scaling\_opcode\_ID** | **Description** | |
| 0 | NOT | Bitwise NOT of the decoded symbol. |
| 1 | AND | Bitwise AND of the scaling value and the decoded symbol. |
| 2 | OR | Bitwise OR of the scaling value and the decoded symbol. |
| 3 | XOR | Bitwise XOR of the scaling value and the decoded symbol. |
| 4 | LSHIFT | Left shifting of the decoded symbol by scaling\_value. |
| 5 | RSHIFT | Right shifting of the decoded symbol by scaling\_value. |
| 6 | ADD | Addition of the scaling\_value to the decoded symbol. |
| 7 | SUB | Subtraction of the scaling\_value from the decoded symbol. |
| 8 | rSUB | Subtraction of the decoded symbol from the scaling\_value. |
| 9 | MUL | Multiplication of the scaling\_value with decoded\_symbol. |
| 10 | DIV | Integer division of the decoded symbol by the scaling value. |
| 11 | rDIV | Integer division of the scaling\_value by the decoded symbol. |
| 12 .. 255 | Reserved | Reserved for future use. |

### Decoder configuration for read identifiers

The decoder configuration syntax for read identifiers is specified below.

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| decoder\_configuration\_read\_ids(encoding\_mode\_id){ |  |
| if (encoding\_mode\_id == 0){ |  |
| **output\_symbol\_size** | u(5) |
| default\_cabac\_binarizations() | 11.3.10.2.4 11.3.10.2.5 |
| for (s=k=0; s<output\_symbol\_size; k++){ |  |
| **coding\_symbol\_size**[k] | u(5) |
| **use\_default\_cabac\_binarization\_flag** | u(1) |
| if (!use\_default\_cabac\_binarization\_flag) |  |
| cabac\_binarizations() | 11.3.10.2.4 11.3.10.2.5 |
| s += coding\_symbol\_size[k] |  |
| } |  |
| **rle\_guard** | u(8) |
| } else if(encode\_mode\_id >= 1){ |  |
| /\* reserved for future use \*/ |  |
| } |  |
| } |  |
|  |  |

**output\_symbol\_size** represents the size (in bits) of each output symbol when decoding the **token\_type** descriptor sequences. The default value is 8.

**coding\_symbol\_size** represents the internal coding size in which each was coded into encoded bitstream. Note that **coding\_symbol\_size** must always be multiple of **output\_symbol\_size** and cannot be greater than **output\_symbol\_size**. The default value is 4.

**use\_default\_cabac\_binarization\_flag** signals if the current coding symbol segment use defaulat CABAC binarization scheme, or use its own specific binarization scheme.

**rle\_guard** represents the guard value used in the decoding process of RLE method (listed in Table 24 and specified in clause 11.3.10.2.3) for coding of **token\_type** descriptor sequences.

**default\_cabac\_binarizations(),** **cabac\_binarizations()** represent the data structure containing the parameters used for CABAC\_ORDER\_0 and CABAC\_ORDER\_1 methods (listed in Table 24 specified in clauses 11.3.10.2.4 and 11.3.10.2.5, respectively) for coding of **token\_type** descriptor sequences. It is described in clause 13.2.2.2.

### Initialization process for context variables

Input to this process are a context variable table ctxTable and variables ctxIdx and initValue.

ctxTable[ ] is the data structure containing all context variables needed to decode a descriptor sequence or subsequence. Each element of the ctxTable[ ] represents one context variable and consists of two state variables: pStateIdx and valMps. The variable pStateIdx represents a probability state index and the variable valMps represents the value of the most probable symbol as further described in clause 13.2.5.2.

The output of this process is an initialized context variable in ctxTable at index ctxIdx.

The state variables pStateIdx and valMps corresponding to index ctxIdx are initialized based on a 7-bit initValue as described below:

|  |
| --- |
| **Syntax** |
| context\_initialize\_states(ctxTable[ ], ctxIdx, initValue) { |
| ctxTable[ctxIdx].valMps = ( initValue <= 63 ) ? 0 : 1 |
| ctxTable[ctxIdx].pStateIdx = ctxTable[i].valMps ? ( initValue − 64 ) : ( 63 − initValue ) |
| } |

where

ctxTable[ctxIdx].valMps represents the variable valMps associated to the element in ctxTable at index ctxIdx

ctxTable[ctxIdx].pStateIdx represents the variable pStateIdx associated to the element in ctxTable at index ctxIdx

### Arithmetic decoding engine

#### Initialization

Outputs of this process are the initialized decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision.

The status of the arithmetic decoding engine is represented by the variables ivlCurrRange and ivlOffset. In the initialization procedure of the arithmetic decoding process, ivlCurrRange is set equal to 510 and ivlOffset is set equal to the value returned from read\_bits( 9 ) interpreted as a 9 bit binary representation of an unsigned integer with the most significant bit written first.

The bitstream shall not contain data that result in a value of ivlOffset being equal to 510 or 511.

NOTE – The description of the arithmetic decoding engine in this Specification utilizes 16 bit register precision. However, a minimum register precision of 9 bits is required for storing the values of the variables ivlCurrRange and ivlOffset after invocation of the arithmetic decoding process (DecodeBin) as specified in clause 13.2.5.2. The arithmetic decoding process for a binary decision (DecodeDecision) as specified in clause 13.2.5.2.2 and the decoding process for a binary decision before termination (DecodeTerminate) as specified in clause 13.2.5.2.5 require a minimum register precision of 9 bits for the variables ivlCurrRange and ivlOffset. The bypass decoding process for binary decisions (DecodeBypass) as specified in clause 13.2.5.2.4 requires a minimum register precision of 10 bits for the variable ivlOffset and a minimum register precision of 9 bits for the variable ivlCurrRange.

#### Arithmetic decoding process

##### General

Inputs to this process are ctxTable, ctxIdx, and bypass\_flag, as specified in clause 13.2.6.2.6, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

Figure 8 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table ctxTable and the ctxIdx are passed to the arithmetic decoding process DecodeBin( ctxTable, ctxIdx ), which is specified as follows:

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in clause 13.2.5.2.4 is invoked.

– Otherwise, if bypassFlag is equal to 0, ctxTable is equal to 0, and ctxIdx is equal to 0, DecodeTerminate( ) as specified in clause 13.2.5.2.5 is invoked.

– Otherwise (bypassFlag is equal to 0 and ctxTable is not equal to 0), DecodeDecision( ) as specified in clause 13.2.5.2.2 is invoked.



Figure 8 – Overview of the arithmetic decoding process for a single bin (informative)

NOTE – Arithmetic coding is based on the principle of recursive interval subdivision. Given a probability estimation p( 0 ) and p( 1 ) = 1 − p( 0 ) of a binary decision ( 0, 1 ), an initially given code sub-interval with the range ivlCurrRange will be subdivided into two sub-intervals having range p( 0 ) \* ivlCurrRange and ivlCurrRange − p( 0 ) \* ivlCurrRange, respectively. Depending on the decision, which has been observed, the corresponding sub-interval will be chosen as the new code interval, and a binary code string pointing into that interval will represent the sequence of observed binary decisions. It is useful to distinguish between the most probable symbol(MPS) and the least probable symbol(LPS), so that binary decisions have to be identified as either MPS or LPS, rather than 0 or 1. Given this terminology, each context is specified by the probability pLPS of the LPS and the value of MPS (valMps), which is either 0 or 1. The arithmetic core engine in this Specification has three distinct properties:

– The probability estimation is performed by means of a finite-state machine with a table-based transition process between 64 different representative probability states { pLPS( pStateIdx ) | 0 <= pStateIdx < 64 } for the LPS probability pLPS. The numbering of the states is arranged in such a way that the probability state with indexpStateIdx = 0 corresponds to an LPS probability value of 0.5, with decreasing LPS probability towards higher state indices.

– The range ivlCurrRange representing the state of the coding engine is quantized to a small set {Q1,...,Q4} of pre-set quantization values prior to the calculation of the new interval range. Storing a table containing all 64x4 pre-computed product values of Qi \* pLPS( pStateIdx ) allows a multiplication-free approximation of the product ivlCurrRange \* pLPS( pStateIdx ).

– For syntax elements or parts thereof for which an approximately uniform probability distribution is assumed to be given a separate simplified encoding and decoding bypass process is used.

##### Arithmetic decoding process for a binary decision

###### General

Inputs to this process are the variables ctxTable, ctxIdx, ivlCurrRange, and ivlOffset.

Outputs of this process are the decoded value binVal, and the updated variables ivlCurrRange and ivlOffset.

Figure 9 shows the flowchart for decoding a single decision (DecodeDecision):

1. The value of the variable ivlLpsRange is derived as follows:

– Given the current value of ivlCurrRange, the variable qRangeIdx is derived as follows:

qRangeIdx =( ivlCurrRange >> 6 ) & 3 (13‑1)

– Given qRangeIdx and pStateIdx associated with ctxTable and ctxIdx, the value of the variable rangeTabLps as specified in Table 62 is assigned to ivlLpsRange:

ivlLpsRange = rangeTabLps[ pStateIdx ][ qRangeIdx ] (13‑2)

1. The variable ivlCurrRange is set equal to ivlCurrRange − ivlLpsRange and the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 − valMps, ivlOffset is decremented by ivlCurrRange, and ivlCurrRange is set equal to ivlLpsRange.

– Otherwise, the variable binVal is set equal to valMps.

Given the value of binVal, the state transition isperformed as specified in clause 13.2.5.2.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in clause 13.2.5.2.3.



Figure 9 – Flowchart for decoding a decision

###### State transition process

Inputs to this process are the current pStateIdx, the decoded value binVal and valMps values of the context variable associated with ctxTable and ctxIdx.

Outputs of this process are the updated pStateIdx and valMps of the context variable associated with ctxIdx.

Depending on the decoded value binVal, the update of the two variables pStateIdx and valMps associated with ctxIdx is derived as follows:

|  |
| --- |
| If (adaptive\_mode\_flag) {  if( binVal = = valMps )   pStateIdx = transIdxMps( pStateIdx )  else {  if( pStateIdx = = 0 )  valMps = 1 − valMps  pStateIdx = transIdxLps( pStateIdx )  }  } |

Table 63 specifies the transition rules transIdxMps( ) and transIdxLps( ) after decoding the value of valMps and 1 − valMps, respectively.

Table 62 – Specification of rangeTabLps depending on the values of pStateIdx and qRangeIdx.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pStateIdx** | **qRangeIdx** | | | | **pStateIdx** | **qRangeIdx** | | | |
| **0** | **1** | **2** | **3** | **0** | **1** | **2** | **3** |
| **0** | 128 | 176 | 208 | 240 | **32** | 27 | 33 | 39 | 45 |
| **1** | 128 | 167 | 197 | 227 | **33** | 26 | 31 | 37 | 43 |
| **2** | 128 | 158 | 187 | 216 | **34** | 24 | 30 | 35 | 41 |
| **3** | 123 | 150 | 178 | 205 | **35** | 23 | 28 | 33 | 39 |
| **4** | 116 | 142 | 169 | 195 | **36** | 22 | 27 | 32 | 37 |
| **5** | 111 | 135 | 160 | 185 | **37** | 21 | 26 | 30 | 35 |
| **6** | 105 | 128 | 152 | 175 | **38** | 20 | 24 | 29 | 33 |
| **7** | 100 | 122 | 144 | 166 | **39** | 19 | 23 | 27 | 31 |
| **8** | 95 | 116 | 137 | 158 | **40** | 18 | 22 | 26 | 30 |
| **9** | 90 | 110 | 130 | 150 | **41** | 17 | 21 | 25 | 28 |
| **10** | 85 | 104 | 123 | 142 | **42** | 16 | 20 | 23 | 27 |
| **11** | 81 | 99 | 117 | 135 | **43** | 15 | 19 | 22 | 25 |
| **12** | 77 | 94 | 111 | 128 | **44** | 14 | 18 | 21 | 24 |
| **13** | 73 | 89 | 105 | 122 | **45** | 14 | 17 | 20 | 23 |
| **14** | 69 | 85 | 100 | 116 | **46** | 13 | 16 | 19 | 22 |
| **15** | 66 | 80 | 95 | 110 | **47** | 12 | 15 | 18 | 21 |
| **16** | 62 | 76 | 90 | 104 | **48** | 12 | 14 | 17 | 20 |
| **17** | 59 | 72 | 86 | 99 | **49** | 11 | 14 | 16 | 19 |
| **18** | 56 | 69 | 81 | 94 | **50** | 11 | 13 | 15 | 18 |
| **19** | 53 | 65 | 77 | 89 | **51** | 10 | 12 | 15 | 17 |
| **20** | 51 | 62 | 73 | 85 | **52** | 10 | 12 | 14 | 16 |
| **21** | 48 | 59 | 69 | 80 | **53** | 9 | 11 | 13 | 15 |
| **22** | 46 | 56 | 66 | 76 | **54** | 9 | 11 | 12 | 14 |
| **23** | 43 | 53 | 63 | 72 | **55** | 8 | 10 | 12 | 14 |
| **24** | 41 | 50 | 59 | 69 | **56** | 8 | 9 | 11 | 13 |
| **25** | 39 | 48 | 56 | 65 | **57** | 7 | 9 | 11 | 12 |
| **26** | 37 | 45 | 54 | 62 | **58** | 7 | 9 | 10 | 12 |
| **27** | 35 | 43 | 51 | 59 | **59** | 7 | 8 | 10 | 11 |
| **28** | 33 | 41 | 48 | 56 | **60** | 6 | 8 | 9 | 11 |
| **29** | 32 | 39 | 46 | 53 | **61** | 6 | 7 | 9 | 10 |
| **30** | 30 | 37 | 43 | 50 | **62** | 6 | 7 | 8 | 9 |
| **31** | 29 | 35 | 41 | 48 | **63** | 2 | 2 | 2 | 2 |

Table 63. – State transition table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **pStateIdx** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **transIdxLps** | 0 | 0 | 1 | 2 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 11 | 11 | 12 |
| **transIdxMps** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **pStateIdx** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** |
| **transIdxLps** | 13 | 13 | 15 | 15 | 16 | 16 | 18 | 18 | 19 | 19 | 21 | 21 | 22 | 22 | 23 | 24 |
| **transIdxMps** | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| **pStateIdx** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** | **41** | **42** | **43** | **44** | **45** | **46** | **47** |
| **transIdxLps** | 24 | 25 | 26 | 26 | 27 | 27 | 28 | 29 | 29 | 30 | 30 | 30 | 31 | 32 | 32 | 33 |
| **transIdxMps** | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| **pStateIdx** | **48** | **49** | **50** | **51** | **52** | **53** | **54** | **55** | **56** | **57** | **58** | **59** | **60** | **61** | **62** | **63** |
| **transIdxLps** | 33 | 33 | 34 | 34 | 35 | 35 | 35 | 36 | 36 | 36 | 37 | 37 | 37 | 38 | 38 | 63 |
| **transIdxMps** | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 62 | 63 |

##### Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from Block Payload data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset.

A flowchart of the renormalization is shown in Figure 10. The current value of ivlCurrRange is first compared to 256 and then the following applies:

– If ivlCurrRange is greater than or equal to 256, no renormalization is needed and the RenormD process is finished;

– Otherwise (ivlCurrRange is less than 256), the renormalization loop is entered. Within this loop, the value of ivlCurrRange is doubled, i.e., left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ).

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.



Figure 10 – Flowchart of renormalization.

##### Bypass decoding process for binary decisions

Inputs to this process are bits from Block Payload data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variable ivlOffset and the decoded value binVal.

The bypass decoding process is invoked when bypassFlag is equal to 1. Figure 11 shows a flowchart of the corresponding process.

First, the value of ivlOffset is doubled, i.e., left-shifted by 1 and a single bit is shifted into ivlOffset by using read\_bits( 1 ). Then, the value of ivlOffset is compared to the value of ivlCurrRange and then the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 and ivlOffset is decremented by ivlCurrRange.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0*.*

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.



Figure 11 – Flowchart of bypass decoding process.

##### Decoding process for binary decisions before termination

Inputs to this process are bits from Block Payload data and the variables ivlCurrRange and ivlOffset.

Outputs of this process are the updated variables ivlCurrRange and ivlOffset, and the decoded value binVal.

This decoding process applies to decoding of end\_of\_descriptor\_subsequence\_terminatecorresponding to ctxTable equal to 0 and ctxIdx equal to 0. Figure 12 shows the flowchart of the corresponding decoding process, which is specified as follows:

First, the value of ivlCurrRange is decremented by 2. Then, the value of ivlOffset is compared to the value of ivlCurrRange and then the following applies:

– If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated. The last bit inserted in register ivlOffset is equal to 1. When decoding end\_of\_descriptor\_subsequence\_terminate, this last bit inserted in register ivlOffset is interpreted as rbsp\_stop\_one\_bit. When decoding end\_of\_subset\_one\_bit, this last bit inserted in register ivlOffset is interpreted as alignment\_bit\_equal\_to\_one.

– Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0 and renormalization is performed as specified in clause 13.2.5.2.3.

NOTE – This procedure may also be implemented using DecodeDecision( ctxTable, ctxIdx, bypassFlag ) with ctxTable = 0, ctxIdx = 0 and bypassFlag = 0. In the case where the decoded value is equal to 1, seven more bits would be read by DecodeDecision( ctxTable, ctxIdx, bypassFlag ) and a decoding process would have to adjust its bitstream pointer accordingly to properly decode following syntax elements.



Figure 12 – Flowchart of decoding a decision before termination.

##### Alignment process prior to aligned bypass decoding

Input to this process is the variable ivlCurrRange.

Output of this process is the updated variable ivlCurrRange.

ivlCurrRange is set equal to 256.

NOTE – When ivlCurrRange is 256, ivlOffset and the bit-stream can be considered as a shift register, and binVal as the register's second most significant bit (the most significant bit is always 0 due to the restriction of ivlOffset being less than ivlCurrRange).

### Decoding process for sequence descriptors

This clause describes the decoding process for descriptors specified in clause 10.4 and 10.5.

#### General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in clause 13.2.1.

Output of this process is the value of the syntax element.

This process specifies how each bin of a bin string is parsed for each syntax element. After parsing each bin, the resulting bin string is compared to all bin strings of the binarization of the syntax element and the following applies:

* If the bin string is equal to one of the bin strings, the corresponding value of the syntax element is the output.
* Otherwise (the bin string is not equal to one of the bin strings), the next bit is parsed.

While parsing each bin, the variable binIdx is incremented by 1 starting with binIdx being set equal to 0 for the first bin.

The parsing of each bin is specified by the following two ordered steps:

1. The context selection process as specified in clause 13.2.6.2.5.

2. The arithmetic decoding process as specified in clause 13.2.5.2.1 is invoked with ctxTable, ctxIdx, and bypassFlag as inputs and the value of the bin as output.

#### Block Payload decoding process

Inputs to this process are a Block Payload as specified in clause 7.8 and a decoder configuration (as specified in clause 13.2.2.2) for each descriptor sequence identified by a **descriptor\_ID** specified in the Block Header (clause 7.7).

Output of this process is one bi-dimensional array of the decoded\_symbols data structure[descriptor\_ID] (the reconstructed genomic descriptors of type **descriptor\_ID**).

##### General decoding process for descriptors of sequence reads

A block of encoded descriptors is decoded as follows:

For each descriptor\_subsequence\_ID associated to the current descriptor\_ID:

* Initialize an array prv\_values to all 0.
* Set prv\_values as specified in 13.2.6.2.2.
* if dependency\_enabled\_flag is enabled for this descriptor\_substream\_ID, loop up their dependencies and update the prv\_values based on their dependencies as specified in 13.2.6.2.2.
* For each expected output symbol for the subsequence identified by descriptor\_subsequence\_ID and while more\_data\_in\_block\_payload( ) is true
* Initialize ctxTable with the CABAC context variables needed to decode each descriptor subsequence as specified in clause 13.2.6.2.3
* Retrieve the Look-Up Tables lut\_values[][][] needed for CABAC decoding as specified in 13.2.6.2.4
  + For each coding\_symbol\_size bits sub-portion of the decoded symbol and repeated until output\_symbol\_size bits of the decoded symbol are decoded, where coding\_symbol\_size is multiple of output\_symbol\_size
    - Select the context index ctxIdx as specified in 13.2.6.2.5
    - Perform CABAC decoding of decodedCabacSymbol as specified in 14.2.6.2.6
    - Calculate invTransfSym by performing an inverse transformation of decodedCabacSymbol as specified in 14.2.6.2.7
    - Update the jth decoded symbol of the ith descriptor subsequence as decoded\_symbol |= (invTransfSym <<s)
    - Update the state variables values as specified in 14.2.6.2.8
* if equality\_ID is present among the identifiers of the transformations associated to this descriptor\_subsequence\_ID apply the equality transformation as specified in 14.2.6.2.10.1
* if match\_coding\_ID is present among the identifiers of the transformations associated to this descriptor\_subsequence\_ID apply the equality transformation as specified in 14.2.6.2.10.2
* multiplex the subsequences as specified in 14.2.6.2.9

The process described above is implemented in pseudo code as follows:

|  |  |
| --- | --- |
| **Decoding process** | |
| encoded\_descriptor\_sequences(descriptor\_ID) { |  |
| /\* Initializations \*/ |  |
| decoded\_symbols[descriptor\_ID][num\_descriptor\_subsequences\_minus1 + 1][ ] = {0} |  |
| for (i=0; i<=num\_descriptor\_subsequences\_minus1; i++) { |  |
| prv\_values[ ] = {0} |  |
| dependency\_reference\_lookup(descriptor\_ID, descriptor\_subsequence\_ID, prv\_values) | 13.2.6.2.2 |
| j = 0 |  |
| do { |  |
| ctxTable[ ] = contexts\_creation\_initialization(i) | 13.2.6.2.3 |
| lut\_values[ ][ ][ ] = decode\_LUTs(i, ctxTable, 0) | 13.2.6.2.4 |
| for (s=0; s<output\_symbol\_size[i]; s+= coding\_symbol\_size[i]) { |  |
| ctxIdx = context\_selection(i, prv\_values) | 13.2.6.2.5 |
| /\* CABAC decoding \*/ |  |
| decodedCabacSymbol | 13.2.6.2.6 |
| /\* Inverse-transform \*/ |  |
| invTransfSym = inverse\_transform\_symbol(i, decodedCabacSymbol , prv\_values, lut\_values, bypassFlag) | 13.2.6.2.7 |
| decoded\_symbols[descriptor\_ID][i][j] |= ( invTransfSym <<s) |  |
| /\* Update state values \*/ |  |
| update\_state\_values(i, invTransfSym , prv\_values) | 13.2.6.2.8 |
| } |  |
| j++ |  |
| invTransfSym = inverse\_transform\_symbol(i, decodedCabacSymbol , prv\_values, lut\_values, bypassFlag) | 13.2.6.2.7 |
| } while(j < num\_output\_symbols[i] && more\_data\_in\_block\_payload( )) |  |
| **end\_of\_descriptor\_subsequence\_terminate** | av(t) |
| } |  |
| // multiplex subsequences |  |
| if( transform\_present(equality\_ID)){ |  |
| equality\_transformation\_multiplexing(decoded\_symbols[descriptor\_ID][][]) | 13.2.6.2.10.1 |
| } |  |
| if( transform\_present(match\_coding\_ID)){ |  |
| match\_transformation\_multiplexing(decoded\_symbols[descriptor\_ID][][]) | 13.2.6.2.10.2 |
| } |  |
|  |  |
| } |  |

**num\_descriptor\_subsequences\_minus1** specifies the number of descriptor subsequences signaled in decoder configuration (clause 13.2.2.2) for the descriptor sequence identified with **descriptor\_ID**, minus 1.

**num\_output\_symbols[i]** specifies the number of output symbols encoded for the ith descriptor subsequence, which is signaled in support\_values() specified clause 13.2.2.1.

**decoded\_symbols[descriptor\_ID][i][]** is the data structure containing the list of output symbols decoded for the ith descriptor subsequence of the descriptor identified by descriptor\_ID.

**prv\_values[]** is the data structure specifying the internal state of this process and contains the list of previously decoded symbols. Index 0 represents the most recent value.

**output\_symbol\_size[i]** specifies the size of the output symbol (in bits) for each encoded (output) symbol of the ith descriptor subsequence. It is signaled in support\_values() specified clause 13.2.2.1.

**coding\_symbol\_size[i]** specifies the size of the symbol (in bits) used for encoding each encoded symbol of the ith descriptor subsequence. It is signaled in support\_values() specified clause 13.2.2.1.

**bypassFlag** signals if the bypass mode of arithmetic decoding engine (clause 13.2.5.2.4) is enabled and is signaled in cabac\_binarizations() specified in clause13.2.2.2.

**dependency\_reference\_lookup()**specifies the process for lookup of reference values from the dependency streams and is specified in clause 13.2.6.2.2.

**context\_selection()** specifies the process for context selection and is specified in clause 13.2.6.2.5.

**decodedCabacSymbol** specifies the decoded symbols as specified in clause 13.2.6.2.6. Input to the process are variables i, ctxTable, ctxIdx and bypassFlag.

**inverse\_transform\_symbol()**specifies the process for inverse transformation of each decoded symbol and is specified in clause 13.2.6.2.7.

**update\_state\_values()**specifies the process to update the state values and is specified in clause 13.2.6.2.8.

**more\_data\_in\_block\_payload( )** is specified in clause 6.3.

**transform\_present(transform\_ID)** signals the presence of transform\_ID in the vector of transforms associated to this descriptor\_ID.

##### Dependency reference lookup

Input to this process is the descriptor\_subsequence\_ID, and decoder configuration related to dependency descriptor sequences (specified in clause 13.2.2).

Output of this process is a vector prv\_values containing updated state variables based on the lookup from the dependency descriptor sequence.

The updated state variables are calculated as follows:

|  |
| --- |
| if(dependency\_enabled\_flag && num\_dependencies > 0) {  /\* mmtype dependency on mmpos \*/  if(dependency\_ID[0] == 3 && descriptor\_ID == 4 && descriptor\_subsequence\_ID == 1)  prv\_values[0] = reference\_genomic\_record[decoded\_tokens[dependency\_ID[0][i]]]  } |

**reference\_genomic\_record[i]** specifies the base value at the ith position of the reference genomic record.

**decoded\_tokens[descriptor\_ID][i]** specifies the reconstructed ith descriptor value of a given descriptor sequence identified by descriptor\_ID.

##### Contexts creation and initialization

Input to this process is the **descriptor\_subsequence\_ID** and decoder configuration corresponding to context variables, signaled via syntax cabac\_context\_parameters(descriptor\_subsequence\_ID) specified in 13.2.2.2.1.

The initialization process of a context variable is specified in clause 13.2.4.

Outputs of this process is a **ctxTable[ ]** data structure containing the initialized CABAC context variables needed to decode each descriptor subsequence (as specified in clause 13.2.2.2.2). The size of ctxTable[] is signaled as **num\_contexts** as part of cabac\_context\_parameters(descriptor\_subsequence\_ID) specified in 13.2.2.2.2, where ctxTable[i] represents the ith context variable.

If bypassFlag is equal to 0 the ctxTable is calculated as follows:

|  |
| --- |
| **ctxTable**[num\_contexts] = { {0,0} }  if( default\_context\_init\_flag ) {  for (ctxIdx=0; ctxIdx<num\_contexts; ctxIdx++) {  context\_initialize\_states(**ctxTable**, ctxIdx, 64)  }  } else {  for (ctxIdx=0; ctxIdx<num\_contexts; ctxIdx++) {  context\_initialize\_states(**ctxTable**, ctxIdx, context\_initialization\_value[ctxIdx])  }  } |

**num\_contexts** specifies the size of the ctxTable[ ] containing the list of context values needed for the decoding of the descriptor subsequence (including any LUTs). It is signaled in cabac\_context\_parameters(descriptor\_subsequence\_ID) for each descriptor subsequence (clause 13.2.2.2.2).

**context\_initialization\_value[i]** specifies the explicit initialization value for the ith context variable. It is signaled in cabac\_context\_parameters(descriptor\_subsequence\_ID) for each descriptor subsequence (clause 13.2.2.2.2).

Note: this initialization step must precede the decoding process of the descriptor subsequence.

##### Decoding LUTs

Input to this process are the descriptor\_subsequence\_ID, ctxTable and ctxIdx identifying the contexts where CABAC decoding of LUTs is to be performed.

A lookup table (i.e. lists of key-value pairs) is signaled as list of values, where the keys are assumed to be equal to the 0-based index of a value in the list.

This process is executed when fixed\_transform\_ID is present among the identifiers of the transformations associated to this descriptor\_subsequence\_ID as specified in clause 13.2.2.3.

Output of this process is

* a collection of 2-dimensional tables when coding\_order is equal to 1
* a collection of 3-dimensional tables when coding\_order is equal to 2

where coding\_order is signaled in support\_values specified in 13.2.2.1.

If coding\_order is 1 then the Look-Up Table LUT\_values is calculated as follows:

|  |  |
| --- | --- |
| **Decoding Process** | **Type** |
| decode\_LUTs\_Order\_1(descriptor\_subsequence\_ID, ctxTable, ctxIdx) { |  |
| numAlpha = 1<< coding\_symbol\_size |  |
| i = 0, j=0 |  |
| do { |  |
| LUT\_values[i][j] = 0 |  |
| **numElems** | ae(v) |
| do { |  |
| **LUT\_values**[i][j++] | ae(v) |
| } while(j < numElems && more\_data\_in\_block\_payload( )) |  |
| i++ |  |
| } while(i < numAlpha && more\_data\_in\_block\_payload( )) |  |
| } |  |

**numElems** specifies the number of elements in the LUT and is calculated as specified in 13.2.6.2.6.

**LUT\_values**[i][j++] are the values of the LUT and are calculated as specified in 13.2.6.2.6. Set a limit on the size.

If coding\_order is 2 then the Look-Up Table LUT\_values is calculated as follows:

|  |  |
| --- | --- |
| **Decoding Process** | **Type** |
| decode\_LUTs\_Order\_2(descriptor\_subsequence\_ID, ctxTable, ctxIdx) { |  |
| numAlpha = 1<< coding\_symbol\_size |  |
| i = 0 |  |
| do { |  |
| LUT\_values[i++][ ][ ] = decode\_LUTs\_Order\_1(descriptor\_subsequence\_ID, ctxTable, ctxIdx) |  |
| } while(i < numAlpha && more\_data\_in\_block\_payload( )) |  |
| } |  |

##### Context selection

Input to this process is the descriptor\_subsequence\_ID and decoder configuration corresponding to context variables, signaled via syntax cabac\_context\_parameters() specified in 13.2.2.2.1.

The initialization process of a context variable is specified in clause 13.2.4.

Outputs of this process is the index ctxIdx of the element of **ctxTable** containing the initialized CABAC context variables needed to decode each descriptor subsequence (as specified in clause 13.2.6.2.1).

|  |
| --- |
| **Decoding process** |
| context\_selection(descriptor\_subsequence\_ID, prv\_values[]) { |
| if(!bypass\_flag && coding\_order > 0){ |
| ctxIdx = context\_offset\_luts |
| for (i=0; i<coding\_order; i++) { |
| ctxIdx += prv\_value[i] \* context\_offset\_values[i] |
| } |
| } |
| } |

**bypass\_flag** specifies if the bypass mode of the arithmetic decoding engine (clause 13.2.5.2.4) is enabled and is signaled in cabac\_binarizations() specified in clause13.2.2.2.

**coding\_order** specifies the coding order signaled in support\_values() for each descriptor subsequence (clause 13.2.2.1).

**context\_offset\_luts** specifies the number of context variables used in decoding of LUTs. It is signaled in cabac\_context\_parameters() for each descriptor subsequence (clause 13.2.2.2.2).

**context\_offset\_values[]** specifies the list of context offset values signaled in cabac\_context\_parameters() for each descriptor subsequence (clause 13.2.2.2.2).

##### Decoding descriptor symbols

Input to this process is the descriptor\_subsequence\_ID, ctxTable the index ctxIdx of the context selected as specified in clause 13.2.6.2.5, and **bypass\_flag**.

Output of this process is a variable decodedCabacSymbol calculated as specified in clause 13.2.5.2 using the binarization identified by **binarization\_ID** as specified in Table 58. Binarizations are listed in clause 13.2.1.

##### Inverse transformation

Input to this process are the descriptor\_subsequence\_ID, the symbol decoded according to the process specified in clause 13.2.6.2.6, a list of previously decoded symbols, the decoded LUTs (13.2.6.2.4) and the decoder configuration corresponding to inverse transformation, signaled via syntax transformation\_parameters (descriptor\_subsequence\_ID) as specified in 13.2.2.3.

Output of this process is the inverse transformed symbol **invTransfSym** calculated as follows:

|  |
| --- |
| inverse\_transform\_symbol(descriptor\_subsequence\_ID, decodedCabacSymbol, prv\_values[], lut\_values[][][]) {  if(transform\_present(diff\_coding\_ID)) {  **invTransfSym** = decodedCabacSymbol+prv\_values[0]  }  else if(transform\_present(fixed\_transform\_ID)) {  if(coding\_order == 2)  **invTransfSym** = lut\_values[prv\_values[1]][prv\_values[0]][decodedCabacSymbol]  else if(coding\_order == 1)  **invTransfSym** = lut\_values[0][prv\_values[0]][decodedCabacSymbol]  }  else if(transform\_present(scale\_opcode\_enabled\_ID)){  if(scaling\_opcode\_ID == 0x00)  **invTransfSym** = !decodedCabacSymbol  else if(scaling\_opcode\_ID == 0x01)  **invTransfSym** = decodedCabacSymbol & scaling\_value  else if(scaling\_opcode\_ID == 0x02)  **invTransfSym** = decodedCabacSymbol| scaling\_value  else if(scaling\_opcode\_ID == 0x03)  **invTransfSym** = decodedCabacSymbol^ scaling\_value  else if(scaling\_opcode\_ID == 0x04)  **invTransfSym** = decodedCabacSymbol<< scaling\_value  else if(scaling\_opcode\_ID == 0x05)  **invTransfSym** = decodedCabacSymbol>> scaling\_value  else if(scaling\_opcode\_ID == 0x06)  **invTransfSym** = decodedCabacSymbol+ scaling\_value  else if(scaling\_opcode\_ID == 0x07)  **invTransfSym** = decodedCabacSymbol- scaling\_value  else if(scaling\_opcode\_ID == 0x08)  **invTransfSym** = scaling\_value - decodedCabacSymbol  else if(scaling\_opcode\_ID == 0x09)  **invTransfSym** = decodedCabacSymbol\* scaling\_value  else if(scaling\_opcode\_ID == 0x0a)  **invTransfSym** = decodedCabacSymbol/ scaling\_value  else if(scaling\_opcode\_ID == 0x0b)  **invTransfSym** = scaling\_value / decodedCabacSymbol  else  **invTransfSym** = decodedCabacSymbol  }  else{  **invTransfSym** = decodedCabacSymbol  }  } |

**lut\_values[i][j][k]** specifies the look-up table decoded in clause 13.2.6.2.4.

**transform\_present(transform\_ID)** specifies the presence of transform\_ID in the vector of transforms associated to this descriptor\_ID as specified in clause 13.2.2.3.

##### Internal State update

Input to this process is the descriptor\_subsequence\_ID, list of state variables, like previously decoded symbols for coding\_order > 0, and the current value currVal to be updated.

Output of this process is an update of the state variables contained in prv\_values.

|  |
| --- |
| **Syntax** |
| update\_state\_values(descriptor\_subsequence\_ID, currVal, prv\_values[]) { |
| for (i=coding\_order-1; i>0; i--) { |
| prv\_values[i] = prv\_values[i-1] |
| } |
| prv\_values[0] = currVal |
| } |

##### Transformation multiplexing

Input to these processes is a set of decoded values: decoded\_values[][]. The subsequences used as input streams for these processes are identified as specified in clause 13.2.2.3.

Output of these processes is the multiplexed descriptor subsequence, which stands as the reconstructed encoded stream for the descriptor subsequence identified by **descriptor\_subsequence\_ID**.

###### Equality transformation

In the case that transform\_ID is equal to equality\_ID, this process shall be used to multiplex the input streams that represent the equality flags (input stream 0, i.e. decoded\_values [equality\_flag\_subsequence\_ID ][]) and the values (input stream 1, i.e. decoded\_values [equality\_values\_subsequence\_ID ][]).

The subsequences shall be multiplexed as follows:

1. previous\_value=0
2. for each symbol from input stream 0 (equality\_flag)
   1. if equality\_flag is equal to 1
      1. The output\_value is equal to the previous\_value
      2. Write output\_value to output
   2. else
      1. The output\_value is equal to the next N bits in input stream 1 (coding\_value)
      2. If output\_value is greater or equal to previous\_value
         1. output\_value = output\_value+1
      3. write output\_value to output
   3. previous\_value = output\_value

###### Match transformation

In the case that transform\_ID is equal to match\_coding\_ID, this process shall be used to multiplex the input streams that represent the pointers (input stream 0, i.e. decoded\_values[**pointer\_subsequence\_ID**][]), lengths (input stream 1, i.e. decoded\_values[**length\_subsequence\_ID**][]), and the raw values (input stream 2, i.e. decoded\_values[**raw\_value\_subsequence\_ID**][]).

The pointers are coded in input stream 0, identified by subsequence\_ID[0], i.e. **pointer\_subsequence\_ID**. Each pointer consists of K bits, represented by the output\_symbol\_size syntax element for this subsequence (see 13.2.2.1).

The lengths are coded in input stream 1, identified by subsequence\_ID[1], i.e. **length\_subsequence\_ID**. Each pointer consists of L bits, represented by the output\_symbol\_size syntax element for this subsequence (see 13.2.2.1).

The pointers are coded in input stream 2, identified by subsequence\_ID[2], i.e. **raw\_value\_subsequence\_ID**. Each pointer consists of M bits, represented by the output\_symbol\_size syntax element for this subsequence (see 13.2.2.1).

During this transformation process, a buffer is assumed that contains the last Min(number of decoded output values, buffer\_size) decoded output values, where the output value at position 0 is assumed to be the output value that has been decoded first and the output value at position N is assumed to be the most recently decoded output\_value.

The input streams shall be multiplexed as follows:

For each length from subsequence 1 (L bits)

1. if length is equal to 0
   1. Read raw\_value from subsequence 2 (M bits) as output\_value
   2. Write output\_value to output
   3. Append output\_value to the buffer
   4. Clear the oldest value if the buffer exceeds buffer\_size
2. Else
   1. Read pointer from subsequence 1 (K bits) as pointer
   2. Reach buffer starting at position n-pointer-length to position n-pointer as output\_value[]
   3. Append output\_value[] to the buffer
   4. If the buffer exceeds buffer\_size clear as many oldest values as the size of output\_value[]

# MPEG-G output format

This clause specifies the output of the normative decoding process specified in clause 0 when **dataset\_type** is equal to 0 or 1 as specified in clause 8.

The format allows:

* saving the decoded content in a structured sequence of data
* providing a normative reference output content to be used for the Conformance test

When **dataset\_type** is equal to 2 the normative output is specified in Clause 7.2.

Table 64. MPEG-G Record syntax

|  |  |
| --- | --- |
| **Syntax** | **Type** |
| mpegg\_record() { |  |
| **global\_ID** | u(64) |
| **read\_name** | st(v) |
| **class\_type** | u(8) |
| **seq\_ID** | u(16) |
| **read\_group** | u(8) |
| **read 1\_first** | u(8) |
| **flags** | u(8) |
| **number\_of\_template\_segments** | u(8) |
| **number\_of\_record\_segments** | u(8) |
| **number\_of\_alignments** | u(16) |
| for (tSeg=0; tSeg<number\_of\_record\_segments; tSeg++) { |  |
| **read\_len[tSeg]** | u(32) |
| **quality\_values[tSeg]** | c(read\_len) |
| **sequence[tSeg]** | c(read\_len) |
| } |  |
| for (noa=0; noa<number\_of\_alignments; noa++) { |  |
| **mapping\_pos[noa]** | u(64) |
| **ecigar\_len[noa][0]** | u(16) |
| **ecigar\_string[noa][0]** | c(ecigar\_size) |
| **reverse\_comp[noa][0]** | u(8) |
| for (tSeg=1; tSeg<number\_of\_template\_segments; tSeg++) { |  |
| **split\_alignment[noa][tSeg]** | u(8) |
| if(split\_alignment[noa][tSeg] == 0) {  /\* paired segment in same record \*/ |  |
| **delta[noa][tSeg]** | i(32) |
| **ecigar\_len[noa][tSeg]** | u(16) |
| **ecigar\_string[noa][tSeg]** | c(cigar\_size) |
| **reverse\_comp[noa][tSeg]** | u(8) |
| } else if (split\_alignment[noa][tSeg] == 1) {  /\* paired segment split in other record \*/ |  |
| **split\_pos[noa][tSeg]** | u(64) |
| **split\_seq\_ID[noa][tSeg]** | u(16) |
| } else if (split\_alignment[noa][tSeg] == 2) { |  |
| /\* unpaired read \*/ |  |
| }else{/\* reserved value for split\_alignment[noa][tSeg] \*/ |  |
| } |  |
| } |  |
| } |  |
| } |  |
| **more\_alignments** | u(8) |
| if (more\_alignments == 0) {/\* no more alignments \*/ |  |
| } else if (more\_alignements == 1) {  /\* more alignments in other records \*/ |  |
| **next\_pos** | u(64) |
| **next\_seq\_ID** | u(16) |
| } else { /\* reserved value for more\_alignments \*/ |  |
| } |  |
| } |  |

This data structure is intended to allow quick data access by browsers of MPEG-G decoded information and efficient file comparison.

The following clauses specify how each attribute of the mpegg\_record data structure is computed.

## global\_ID

**global\_ID** is a unique 64-bit unsigned integer used as unique identifier for the MPEG-G record. Scope of the global\_ID is the Genomic Dataset containing the Access Unit the encoded genomic record belongs to. The global\_ID is calculated as follow:

* The 32 MSB are equal to the Access Unit ID the genomic record belongs to
* The 32 LSB is a counter of the genomic records in the Access Unit. Genomic records are considered ordered by their respective **pos** descriptor.

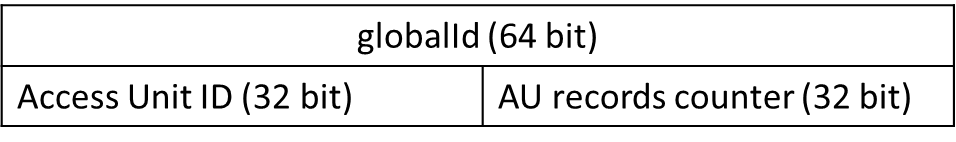


Figure 13. Composition of global\_ID

## read\_name

**read\_name** contains the decoded read identifier as a null terminated string. In case read names are not encoded in the Access Unit, each **read\_name** is empty and only contains the string terminator character (eight zero bits).

## class\_type

**class\_type** is an unsigned integer identifying the data class the Genomic Record belongs to among those specified in clause 10.3.

## seq\_ID

**seq\_ID** contains the unique identifier of the Reference Sequence each segment in the Genomic Record is aligned to among those listed in the Dataset Header as specified in ISO/IEC 23092-1. In case of unmapped sequences (either raw reads or the unmapped portion of aligned reads), this field is empty. The association between this unique identifier and the textual name of the Reference Sequence shall be performed by a decoder compliant with ISO/IEC 23092-1.

## read\_group

**read\_group** is an unsigned integer identifying the read group the MPEG-G record belongs to. Allowed values are between 0 and **num\_groups** - 1. The presence of read groups in an Access Unit is signaled by **num\_groups** > 0 in the Parameter Set as specified in clause 8.

## read 1\_first

**read 1\_first** is a flag signaling if read 1 is the left-most mapped read in the Genomic Record.

This field is always set to 1 in case of:

1. unmapped sequences (either raw reads or the unmapped portion of aligned reads)
2. single (non paired-end) sequence reads
3. genomic record containing only read 1 of a split read

This field is always set to 0 in case of:

1. genomic record containing only read 2 of a split read

## flags

**flags** is an 8-bit unsigned integer obtained by setting the flags related to the Genomic Record as specified in clause 11.4.3.

## number\_of\_template\_segments

**number\_of\_template\_segments** specifies the number of nucleotide segments in the template. This number is the same for all the records within the same access unit.

## number\_of\_record\_segments

**number\_of\_record\_segments** specifies the number of nucleotide segments coded in current record. It is comprised between 1 and **number\_of\_template\_segments**.

## read\_len

**read\_len** contains the 32-bit unsigned integer representation of the length of each read in the Genomic Record expressed as number of nucleotides.

## quality\_values

The **quality\_values** vectors contain the Quality Values associated to each nucleotide contained in the **sequence** vectors. When present each **quality\_values** vector shall have the same size as the corresponding **sequence** vector.

## sequence

The sequence vectors contain the ASCII representation of the nucleotides contained in the Genomic Record according to one of the alphabets specified in clause 10.1.

## number\_of\_alignments

**number\_of\_alignments** is the number of alignments of the first read in the current Genomic Record. It is set to 0 for raw or unmapped reads.

## mapping\_pos

**mapping\_pos** contains the Genomic Record position as 0-based coordinate from the beginning of the Reference Sequence the segment is mapped to. In case of unmapped sequences (either raw reads or the unmapped portion of aligned reads), this field is empty.

## split\_alignment

**split\_alignment** flag indicates whether current segment of the current alignment is coded in another record (value = 1) or not (value = 0).

For split alignments (split\_alignment greater than 0) the following cases have to be identified:

1. split pair with mate on the same reference sequence
2. split pair with mate on another reference sequence
3. unpaired read

|  |  |  |
| --- | --- | --- |
| case | split\_alignment value |  |
| 1 | 1 | seqId == split \_seq\_ID  mapping\_pos != split \_pos |
| 2 | 1 | seqId != split\_seq\_ID  mapping\_pos != split \_pos |
| 3 | 2 | no split\_seq\_ID and no split\_seq\_pos |

## split\_pos

**split\_pos** contains the absolute position of the paired segment in the split alignment.

## split\_seq\_ID

**split\_seq\_ID** contains the unique identifier of the Reference Sequence to which the paired segment in the split alignment is aligned, among those listed in the Dataset Header as specified in ISO/IEC 23092-1.

## delta

**delta** contains the position of the paired segment, relative to the position of the first segment, within the same Reference Sequence of the first segment. It is calculated as specified in clause 11.4.9.

## ecigar\_len

**ecigar\_len** contains the size in characters of the **ecigar\_string** field specified in clause 14.20.

## ecigar\_string

This section specifies an extended CIGAR (E-CIGAR) syntax for strings to be computed from sequences and related mismatches, indels, clipped bases and information on multiple alignments and spliced reads.

Alignments are described as a sequence of consecutive edit operations between the Reference Sequence and a sequence mapped onto the Reference Sequence.

Edit operations might involve skipping or replacing part of the sequence of either reference or read; due to this reason one has to keep track of a pointer *R* to the current position within the reference, and a pointer *r* to the current position within the read. They are both set to 0 at the beginning of the alignment process, the 0 of the reference being the position of the match.

Edit operations specified in this document are listed in Table 65.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Semantics** | **E-CIGAR representation** | **Equivalent SAM CIGAR representation** |
| Increment both pointer-to-reference *R* and pointer-to-read *r* by *n* positions (match) | n matching bases | n= | nM in older versions (not equivalent),  = in recent versions |
| Replace nucleotide in the read with base C from the reference, increment pointer-to-reference *R* and pointer-to-read *r* by 1 | substitution of character C (C is present in the read and not in the reference) | C | M in older versions,  X in recent versions (not equivalent) |
| Increment pointer-to-read *r* by *n* positions (insert from the read) | n bases are inserted in the read (not present in the reference) | n+ | nI |
| Increment pointer-to-reference *R* by *n* positions (deletion of sequence S in the read) | n bases are deleted in the read (but present in the reference) | n- | nD |
| Increment pointer-to-reference *R* by *n* positions (insertion in the read). Can only occur at beginning or end of read | n soft clips | (n) | nS |
| Hard trim. Can only occur at beginning or end of read | n hard clips | [n] | nH |
| Increment pointer-to-reference *R* by *n* positions, splice consensus observed (splice in the read) | An undirected splice of n bases | n\* | nN |
| Increment pointer-to-reference *R* by *n* positions, splice consensus observed on the forward strand (forward splice in the read) | A forward splice of n bases | n/ | Not existing |
| Increment pointer-to-reference *R* by *n* positions, splice consensus observed on the reverse strand (reverse splice in the read) | A reverse splice of n bases | n% | Not existing |

Table 65. Syntax of the MPEG-G E-CIGAR string

The general framework is illustrated in Figure 14.

|  |
| --- |
| 0000000000111111111122222222223333333 Position in the reference  0123456789012345678901234567890123456  ACAGATATATCAGAGACCATACAGGAACATAACAGAC Reference  AAAGATCTAT+++++++++++CAGGTACATA Read  0000000000 1111111111 Position in the read  0123456789 0123456789  CIGAR=(2)4=C3=11+4=T5= |

Figure 14. Alignment with soft clips, deletions and substitutions

According to the syntax specified in Table 65, the alignment depicted in Figure 14 the figure would result in this CIGAR string:

(2)4=C3=11+4=T5=

## reverse\_comp

**reverse\_comp** is a flag associated to each alignment in the Genomic Record signaling if the alignment is on the forward (reverse\_comp = 0) or reverse (reverse\_comp = 1) strand. In case of unmapped sequences (either raw reads or the unmapped portion of aligned reads), this field is empty.

## more\_alignments

**more\_alignments** flag indicates whether more alignments, paired with the segments in current record, are recorded in other records (value = 1) or not (value = 0).

## next\_pos

**next\_pos** contains the absolute position of the paired segment in the next record in the **more\_alignments** list.

## next\_seq\_ID

**next\_seq\_ID** contains the unique identifier of the Reference Sequence to which the paired segment in the next record in the **more\_alignments** list is aligned, among those listed in the Dataset Header as specified in ISO/IEC 23092-1.

# Annex A. Tokenization of Reads Identifiers (informative)

This clause provides an example of tokenization process for two reads identifiers encoded as a sequence of tokens specified in clause 0 and related parameters.

The following read identifiers

HS25\_09827:2:2215:4133:00216#49

HS25\_09827:2:2227:15822:00246#49

are tokenized into token values and parameters as shown in Table 66.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| HS25\_09827:2:2215:4133:00216#49 | | | | HS25\_09827:2:2227:15822:00246#49 | | |
| Token position | Token  values |  | size in bytes | Token values |  | size in bytes |
| 0 | DIFF  0 | First identifier in the AU | 1  4 | DIFF  1 | This identifier is encoded as delta on the previously decoded | 1  4 |
| 1 | ALPHA  72, 83(HS), 0 | null-terminated string  two ASCII characters | 1  3 | MATCH | Matches token at pos. 1 in previous identifier | 1 |
| 2 | DIGITS  25 | digit  uint32 value | 1  4 | MATCH | Matches token at pos. 2 in previous identifier | 1 |
| 3 | CHAR  95 | character  ASCII code for “\_” | 1  1 | MATCH | Matches token at pos. 3 in previous identifier | 1 |
| 4 | DIGITS0  5  9827 | digit of length 5 starting with 0  uint8 length  uint32 value excluding the leading 0 | 1  1  4 | MATCH | Matches token at pos. 4 in previous identifier | 1 |
| 5 | CHAR  58 | character  ASCII code for “:” | 1  1 | MATCH | Matches token at pos. 5 in previous identifier | 1 |
| 6 | DIGITS  2 | digit  uint32 value | 1  4 | MATCH | Matches token at pos. 6 in previous identifier | 1 |
| 7 | CHAR  58 | character  ASCII code for “:” | 1  1 | MATCH | Matches token at pos. 7 in previous identifier | 1 |
| 8 | DIGITS  2215 | digit  uint32 value | 1  4 | DELTA  12 | delta encoding on token at pos. 8 in previous identifier | 1  1 |
| 9 | CHAR  58 | character  ASCII code for “:” | 1  1 | MATCH | Matches token at pos. 9 in previous identifier | 1 |
| 10 | DIGITS  4133 | digit  uint32 value | 1  4 | DIGITS  15822 | digit  uint32 value | 1  4 |
| 11 | CHAR  58 | character  ASCII code for “:” | 1  1 | MATCH | Matches token at pos. 11 in previous identifier | 1 |
| 12 | DIGITS0  5  216 | digit of length 5 starting with 0  uint8 length  uint32 value excluding the leading 0 | 1  1  4 | DELTA0  30 | digit  uint32 value | 1  4 |
| 13 | CHAR  35 | character  ASCII code for “#” | 1  1 | MATCH | Matches token at pos. 13 in previous identifier | 1 |
| 14 | DIGITS  49 | digit  uint32 value | 1  4 | MATCH | Matches token at pos. 14 in previous identifier | 1 |
| 15 | END | End of identifier | 1 | END | End of identifier | 1 |

Table 66. Example of the first two encoded read identifiers in an Access Unit

# Annex B. Examples of decoding processes (informative)

## clips descriptor

An example of **clips** descriptor representing two set of soft clips in a paired-end read is provided in Table 67.

|  |  |  |
| --- | --- | --- |
| **byte** | **value** | **semantics** |
| N | 0x00 | This is the unique identifier in this Access Unit (a monotonically increasing counter) of the next Genomic Record containing soft clips. |
| N+1 | 0xf1 |
| N+2 | 0x31 |
| N+3 | 0xe3 |
| N+4 | 0x01 | This flag indicates the position of the next soft clips in the read or read pair. In this case the soft clips are located at the end of read 1. |
| N+5 | ‘A’ | This is a string of 10 ASCII characters which represent the clipped bases at the end of read 1 |
| ... | ... |
| N+14 | ‘C’ |
| N+15 | 0xfe | More soft clips for this Genomic Record are present. |
| N+16 | 0x02 | This flag indicates the position of the next soft clips in the read or read pair. In this case the soft clips are located at the beginning of read 2. |
| N+17 | ‘G’ | This is a string of 6 ASCII characters which represent the clipped bases at the end of read 1. |
| ... | ... |
| N+22 | ‘T’ |
| N+23 | 0xff | Descriptor terminator. |

Table 67. Example of clips descriptor.

## pair descriptor

Table 68 shows an example of 16 decoded bytes from the **pair** stream for three possible cases:

1. the signed distance between the two mapped reads is encoded in 2 bytes
2. the absolute mapping position on the current Reference Sequence is encoded for one of the two reads
3. the absolute mapping position on a different Reference Sequence is encoded for one of the two reads

Given this sequence of 4 SAM records

|  |  |
| --- | --- |
| 1 | QNAME 163 1 11000 0 100M = 11578 -578 SEQ QS |
| 2 | QNAME 65 1 10006 0 100M = 19800651 0 SEQ QS |
| 3 | QNAME 129 1 10006 0 100M X 19800651 0 SEQ QS |
| 4 | QNAME 1161 1 10010 0 100M = 10010 0 SEQ QS |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SAM record** | **byte** | **value** | **descriptor** | **semantics** |
| 1 | N | 0x0a | pairi | This represents a distance of -578 positions between the left-most mapped base of the read 1 and the left-most mapped base of read 2. The negative distance implies that read 2 is mapped at a smaller coordinate than read 1. |
| N+1 | 0xf1 |
| 2 | N+2 | 0x7f | pairi+1 | read 2 in pair is on the same Reference Sequence but coded separately. This implies that:   * the pos descriptor corresponding to this pair descriptor refers to the mapping position of read 1 * the following 4 bytes encode the absolute mapping position of read 2 on the current Reference Sequence |
| N+3 | 0xfd |
| N+4 | 0x01 | the absolute mapping position of read 2 on the current Reference Sequence is 19,800,651 |
| N+5 | 0x2e |
| N+6 | 0x22 |
| N+7 | 0x4b |
| 3 | N+8 | 0x80 | pairi+2 | read 1 is on a different Reference Sequence |
| N+9 | 0x02 |
| N+10 | 0x00 | Reference Sequence identifier for chromosome X (in this case Ref\_ID = 22) |
| N+11 | 0x16 |
| N+12 | 0x01 | the absolute mapping position of read 1 on the Reference Sequence identified by Ref\_ID = 22 is 19,800,651 |
| N+13 | 0x2e |
| N+14 | 0x22 |
| N+15 | 0x4b |
| 4 | N+16 | 0x80 | pairi+3 | read 2 unpaired |
| N+17 | 0x01 |

Table 68. Examples of decoded pair descriptors

# Annex C. Mapping quality (informative)

Some aligners produce a score for each found alignment which is usually stored in the MAPQ field of the SAM format. MPEG-G supports the representation of a mapping quality value by means of the optional **mscore** descriptor as described in clause 11.4.10. When several alignments scores are needed per each alignment, the parameter **as\_score** in the Parameter Set shall signal how many scores are associated to each alignment in the Access Unit.