**CONTENTS**

|  |
| --- |
| *Page* |
| 1 Scope 22 Normative references 22.1 Identical Recommendations | International Standards 22.2 Paired Recommendations | International Standards equivalent in technical content 22.3 Additional references 23 Definitions 24 Abbreviations 35 Conventions 45.1 General 45.2 Arithmetic operators 45.3 Logical operators 45.4 Relational operators 55.5 Bit-wise operators 55.6 Assignment operators 55.7 Range notation 55.8 Mathematical functions 65.9 Order of operation precedence 65.10 Mathematical functions, operators, and processes for floating-point approximations 75.10.1 Representation of floating-point approximations 75.10.2 Arithmetic operators and functions 85.10.3 Specification of arithmetic operations 95.10.4 Processes for solving linear equation systems using floating-point approximations 125.11 Variables, syntax elements and tables 145.12 Text description of logical operations 155.13 Processes 166 Bitstream and waveform signal format 177 Syntax and semantics 177.1 Method of specifying syntax in tabular form 177.2 Specification of syntax functions and descriptors 187.3 Syntax in tabular form 197.3.1 NAL unit syntax 197.3.2 Raw byte sequence payloads, trailing bits and byte alignment syntax 197.3.3 Frame data syntax 237.3.4 Annotation channel data syntax 297.4 Semantics 297.4.1 General 297.4.2 NAL unit semantics 297.4.3 Raw byte sequence payloads, trailing bits and byte alignment semantics 317.4.4 Frame data semantics 357.4.5 Annotation channel data semantics 398 Decoding process 408.1 General decoding process 408.1.1 General 408.1.2 Channel output index derivation process 408.1.3 Decoding process for independent frames (NAL units of type IF\_NUT) 408.1.4 Decoding process for dependent frames (NAL units of type DF\_NUT) 418.2 Frame data decoding process 418.3 Blockwise prediction decoding process 428.3.1 Linear extrapolation process of an array to the right 428.3.2 Linear extrapolation process of an array to the left 438.3.3 Zero prediction decoding process 438.3.4 DC prediction decoding process 438.3.5 Line fitting prediction decoding process 438.3.6 Cross channel prediction decoding process 448.3.7 Block matching prediction decoding process 478.4 Scaling and inverse transformation decoding process 498.4.1 Scaling process for transform coefficient levels 498.4.2 Inverse transformation process 508.5 Sample wise prediction decoding process 508.5.1 Overview 508.5.2 Sample wise one tap prediction decoding process 518.5.3 Sample wise full slope prediction decoding process 518.5.4 Sample wise half slope prediction decoding process 528.5.5 Filter coefficient decoding process for single channel linear predictive filtering 528.5.6 Filter coefficient decoding process for multi channel linear predictive filtering 538.5.7 Single channel linear predictive filtering prediction decoding process 538.5.8 Multi channel linear predictive filtering prediction decoding process 549 Parsing process 549.1 General 549.2 Parsing process for k-th order Exp-Golomb codes 549.2.1 General 549.2.2 Mapping process for signed Exp-Golomb codes 569.3 CABAC parsing process for frame data 569.3.1 General 569.3.2 Initialization process 569.3.3 Binarization process 599.3.4 Decoding process flow 64 Annex A Placeholder 73 Annex B Byte stream format 73B.1 General 73 |

**List of Figures**

|  Page |
| --- |
| [Figure 1 – Flowchart of the arithmetic decoding process for a single bin (informative) 68](#_Toc181199335)[Figure 2 – Flowchart of renormalization 70](#_Toc181199336)[Figure 3 – Flowchart of bypass decoding process 71](#_Toc181199337)[Figure 4 – Flowchart of decoding a decision before termination 72](#_Toc181199338) |

# Scope

TBA

# Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

## Identical Recommendations | International Standards

– None.

## Paired Recommendations | International Standards equivalent in technical content

– Rec. ITU-T H.274 | ISO/IEC 23002-7 (in force) *Versatile supplemental enhancement information messages for coded video bitstreams.*

## Additional references

– Rec. ITU-T T.35 (in force), *Procedure for the allocation of ITU-T defined codes for non standard facilities.*

– ISO/IEC 23001-11 (in force), *Information Technology – MPEG Systems technologies — Part 11: Energy-efficient media consumption (green metadata).*

– ISO/IEC 23090-13 (in force), *Information technology – Coded representation of immersive media – Part 13: Video decoding interface for immersive media.*

# Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply.

* 1. **coded waveform sequence (CWS)**:
	2. **context variable**: A variable specified for the *adaptive binary arithmetic decoding* *process* of a *bin* by an equation containing recently decoded *bins*.
	3. **decoder**: An embodiment of a *decoding process*.
	4. **decoding order**: The order in which *syntax elements* are processed by the *decoding process*.
	5. **decoding process**: The process specified in this Specification that reads a *bitstream* and derives *decoded* *pictures* from it.
	6. **dependent frame (DF)**: A *NAL unit* with nal\_unit\_type equal to DF\_NUT
	7. **emulation prevention byte**: A *byte* equal to 0x03 that is present within a *NAL unit* when the *syntax elements* of the *bitstream* form certain patterns of *byte* values in a manner that ensures that no sequence of consecutive *byte-aligned* *bytes* in the *NAL unit* can contain a *start code prefix*.
	8. **encoder**: An embodiment of an *encoding process*.
	9. **encoding process**: A process not specified in this Specification that produces a *bitstream* conforming to this Specification.
	10. **flag**: A variable or single-bit *syntax element* that can take one of the two possible values: 0 and 1.
	11. **frame**: Either an *independent frame* or a *dependent frame*.
	12. **frame sequence**: A sequence of *frames* that starts with an *independent frame* followed by zero or more *dependent frames*
	13. **independent frame (IF)**: A *NAL unit* with nal\_unit\_type equal to IF\_NUT.
	14. **network abstraction layer (NAL) unit**: A *syntax structure* containing an indication of the type of data to follow and *bytes* containing that data in the form of an *RBSP* interspersed as necessary with *emulation prevention bytes*.
	15. **network abstraction layer (NAL) unit stream**: A sequence of *NAL units*.
	16. **start code prefix**: A unique sequence of three *bytes* equal to 0x000001 embedded in the *byte stream* as a prefix to each *NAL unit*.

NOTE – The location of a start code prefix can be used by a decoder to identify the beginning of a new NAL unit and the end of a previous NAL unit. Emulation of start code prefixes is prevented within NAL units by the inclusion of emulation prevention bytes.

* 1. **string of data bits (SODB)**: A sequence of some number of bits representing *syntax elements* present within a *raw byte sequence payload* prior to the *raw byte sequence payload stop bit*, where the left-most bit is considered to be the first and most significant bit, and the right-most bit is considered to be the last and least significant bit.
	2. **syntax element**: An element of data represented in the *bitstream*.
	3. **syntax structure**: Zero or more *syntax elements* present together in the *bitstream* in a specified order*.*
	4. **transform**: A part of the *decoding process* by which a *block* of *transform coefficients* is converted to a *block* of spatial-domain values.
	5. **transform block**: A rectangular MxN *block* of samples resulting from a *transform* in the *decoding process*.
	6. **transform coefficient**: A scalar quantity, considered to be in a frequency domain, that is associated with a particular one-dimensional or two-dimensional *frequency index* in a *transform* in the *decoding process*.
	7. **transform coefficient level**: An integer quantity representing the value associated with a particular two‑dimensional frequency index in the *decoding process* prior to *scaling* for computation of a *transform coefficient* value.

# Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply.

AC Annotation Channel

AM Auxiliary Metadata

CG Channel Group

CRC Cyclic Redundancy Check

CWS Coded Waveform Sequence

DF Dependent Frame

EG Exponential-Golomb

EGk k-th order Exponential-Golomb

EOB End Of Bitstream

EOS End Of Sequence

FL Fixed-Length

IF Independent Frame

LPS Least Probable Symbol

LSB Least Significant Bit

MSB Most Significant Bit

NAL Network Abstraction Layer

QP Quantization Parameter

RBSP Raw Byte Sequence Payload

SODB String Of Data Bits

TR Truncated Rice

WPS Waveform Parameter Set

# Conventions

## General

The term "this Specification" is used to refer to this Proposed Draft Recommendation.

The word "shall" is used to express mandatory requirements for conformance to this Specification. When used to express a mandatory constraint on the values of syntax elements or the values of variables derived from these syntax elements, it is the responsibility of the encoder to ensure that the constraint is fulfilled.

The word "may" is used to refer to behaviour that is allowed, but not necessarily required.

The word "should" is used to refer to behaviour of an implementation that is encouraged to be followed under anticipated ordinary circumstances, but is not a mandatory requirement for conformance to this Specification.

Content of this Specification that is identified as "informative" does not establish any mandatory requirements for conformance to this Specification and is thus not considered an integral part of this Specification. Informative remarks in the text are, in some cases, set apart and prefixed with the word "note" or "NOTE".

The word "reserved" is used to specify that some values of a particular syntax element are for future use by ITU-T and shall not be used in syntax structures conforming to this version of this Specification, but could potentially be used in syntax structures conforming to future versions of this Specification by ITU‑T.

The word "unspecified" is used to describe some values of a particular syntax element to indicate that the values have no specified meaning in this Specification and are not expected to have a specified meaning in the future as an integral part of future versions of this Specification.

NOTE – The mathematical operators used in this Specification are similar to those used in the C programming language. However, the results of integer division and arithmetic shift operations are defined more precisely, and additional operations are defined, 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-st.

## 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 | exponentiationSpecifies x to the power of y. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation. |
| / | integer division with truncation of the result toward zero. For example, 7 / 4 and −7 / −4 are truncated to 1 and −7 / 4 and 7 / −4 are truncated to −1. |
| ÷ | division in mathematical equations where no truncation or rounding is intended |
| $$\frac{x}{y}$$ | division in mathematical equations where no truncation or rounding is intended |
| $$\sum\_{i = x}^{y}f( i )$$ | summation of f( i ) with i taking all integer values from x up to and including y |
| x % y | modulus. Remainder of x divided by y, defined only for integers x and y with x >= 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, evaluates to the value of y; otherwise, evaluates to the value of z

When evaluating a logical expression, the value 0 is interpreted as FALSE and any numerical value not equal to 0 is interpreted as TRUE. The result of any logical expression that evaluates as FALSE is the value 0, and the result of any logical expression that evaluates as TRUE is the value 1.

## Relational operators

> 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

& 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 most significant bits (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 least significant bits (LSBs) as a result of the left shift have a value equal to 0.

## Assignment operators

= assignment operator

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

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

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

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

## Range notation

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

## Mathematical functions

Abs( x ) = $\left\{ \begin{matrix}x&;&x >= 0\\-x&;&x < 0\end{matrix}\right.$ (1)

Ceil( x ) smallest integer greater than or equal to x. (2)

Clip1( x ) = Clip3( 0, ( 1 << BitDepth ) − 1, x ) (3)

Clip3( x, y, z ) = $\left\{ \begin{matrix}x&;&z < x\\y&;&z > y\\z&;&otherwise\end{matrix}\right.$ (4)

ClipH( v, w, x ) = $\left\{ \begin{matrix} x + v&;&x < 0\\x-v&;&x > w-1\\x&;&otherwise\end{matrix}\right.$ (5)

Floor( x ) largest integer less than or equal to x. (6)

Log2( x ) base-2 logarithm of x. (7)

Min( x, y ) = $\left\{ \begin{matrix}x&;&x <= y\\y&;&x > y\end{matrix}\right.$ (8)

Max( x, y ) = $\left\{ \begin{matrix}x&;&x >= y\\y&;&x < y\end{matrix}\right.$ (9)

Round( x ) = Sign( x ) \* Floor( Abs( x ) + 0.5 ) (10)

Sign( x ) = $\left\{\begin{matrix}1&;&x > 0\\0&;&x = = 0\\-1&;&x < 0\end{matrix}\right.$ (11)

Sqrt( x ) square root of x (12)

Swap( x, y ) = ( y, x ) (13)

BitWidth( x ) = $\left\{ \begin{matrix} 0&;&x = = 0\\1+Floor( Log2\left( x \right) )&;&x > 0\\unspecified&;&x < 0\end{matrix}\right.$ (14)

## Order of operation precedence

When 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", "$\frac{x}{y}$", "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" |

## Mathematical functions, operators, and processes for floating-point approximations

### Representation of floating-point approximations

The accuracy of all floating-point approximations is specified by the following constant parameter:

– FPANumDigitsVal specifies the number of significant binary digits in the floating-point approximation. Given the value of FPANumDigitsVal, all calculations for the significand of a floating-point approximation can be implemented using signed integer arithmetic with N = FPANumDigitsVal + 2 bits.

Division operations for floating-point approximations are specified based on a look-up table. The accuracy of division operations is specified by the following two constant parameters:

– FPALog2DivTabSize specifies the binary logarithm of the number of elements in the look-up table. The look-up table used for division operations has (1 << FPALog2DivTabSize) elements.

– FPANumDigitsDivTab specifies the number of significant binary digits for the values of the division look-up table. Given the value of FPANumDigitsDivTab, the elements of the look-up table can be represented as unsigned integers with N = FPANumDigitsDivTab bits or as signed integers with N = FPANumDigitsDivTab +1 bits.

The division look-up table is referred to as FPADivTab. It is determined as specified by the following pseudo-code:

nom = 1  <<  ( FPALog2DivTabSize + FPANumDigitsDivTab – 1 )
off    = 1  <<  FPALog2DivTabSize
for( k = 0; k < (1  <<  FPALog2DivTabSize ); k++ ) { (15)
 denom = off + k
 FPADivTab[ k ] = ( nom + (denom >> 1 ) ) / denom
}

where the division represents an integer division with truncation towards zero.

For the following specification, the values FPANumDigitsVal = 30, FPALog2DivTabSize = 8, and FPANumDigitsDivTab = 15 are used. All calculations for the significand of a floating-point approximation can be implemented in 32-bit signed integer arithmetic. The look-up table FPADivTab has 256 elements and each of these elements can be represented as 16-bit signed integer. The total size required for storing the look-up table is 512 bytes.

A floating point approximation is specified as a triple (val, exp, sgn) as follows:

– val is an integer number representing the significant digits of a floating-point approximation, also referred to as significand. The value of val is represented as signed integer number of at least N = FPANumDigitsVal + 2 bits. Even though the actual value of val can always be represented by an unsigned integer of FPANumDigitsVal bits, two additional bits and a signed integer representation is required for intermediate values in calculations with the floating-point approximations. With FPANumDigitsVal = 30, all calculations related to the number val representing the significand of a floating-point approximation can be done using 32-bit signed integers;

– exp is an integer number representing the exponent;

– sgn is a bit representing the sign, where negative numbers have sgn = 1 and positive numbers have sgn = 0.

With x being a floating point representation, the elements of the triple (val, exp, sgn) specifying x are also referred to as x.val, x.exp, and x.sgn, respectively.

The floating-point value fx that is represented by a floating-point approximation x is given by:

fx = (1 – 2 \* x.sgn) \* x.val \* 2^(x.exp). (16)

Note that a value of 0 for the floating-point approximation x is represented by x.val = 0. When x.val = 0, the floating-point approximation represents the value of 0, regardless of the values of x.exp and x.sgn (they don’t have any meaning when x.val is equal to 0).

### Arithmetic operators and functions

In the following x, y, and z represent floating-point approximations and a represents an integer. The following functions and arithmetic operators are specified:

– Conversion of an integer to a floating-point approximation:

x = FPApprox( a ) (17)

The floating-point approximation x is derived by invoking the process for converting an integer to a floating-point approximation as specified in clause 5.10.3.1 with a as input and the output is assigned to x.

– Conversion of a floating-point approximation to an integer:

a = FPAToInteger( x ) (18)

The integer a is derived by invoking the process for converting a floating-point approximation to an integer as specified in clause 5.10.3.2 with a as input and the output is assigned to a.

– Bit-shift to the left:

y = x << a (19)

The floating-point approximation y is derived by invoking the process for shifting a floating-point approximation to the left as specified in clause 5.10.3.3 with x and a as inputs and the output is assigned to y.

– Bit-shift to the right:

y = x >> a (20)

The floating-point approximation y is derived by invoking the process for shifting a floating-point approximation to the right as specified in clause 5.10.3.4 with x and a as inputs and the output is assigned to y.

– Negation:

y = –x (21)

The floating-point approximation y is derived by invoking the process for negating a floating-point approximation as specified in clause 5.10.3.5 with x as input and the output is assigned to y.

– Addition:

z = x + y (22)

The floating-point approximation z is derived by invoking the process for adding two floating-point approximations as specified in clause 5.10.3.6 with x and y as inputs and the output is assigned to z.

– Subtraction:

z = x – y (23)

The floating-point approximation z is derived by invoking the process for subtracting two floating-point approximations as specified in clause 5.10.3.7 with x and y as inputs and the output is assigned to z.

– Multiplication:

z = x \* y (24)

The floating-point approximation z is derived by invoking the process for multiplying two floating-point approximations as specified in clause 5.10.3.8 with x and y as inputs and the output is assigned to z.

– Reciprocal:

y = 1 / x (25)

The floating-point approximation y is derived by invoking the process for deriving the reciprocal of a floating-point approximation as specified in clause 5.10.3.9 with x as input and the output is assigned to y.

– Division:

z = x / y (26)

The floating-point approximation z is derived by invoking the process for dividing two floating-point approximations as specified in clause 5.10.3.10 with x and y as inputs and the output is assigned to z.

The operator precesence for floating-point approximations is the same as specified in clause 5.9.

### Specification of arithmetic operations

#### Process for converting an integer to a floating-point approximation

Input to this process is a signed or unsigned integer value a.

Output of this process is a floating-point approximation x = (x.val, x.exp, x.sgn) representing the integer a.

The output value y is initialized with y.val = 0, y.exp = 0, and y.sgn = 0.

When a is not equal to 0, the output value y is modified as specified by the following pseudo-code:

if( a < 0 ) {
 absVal = –a
 y.sgn = 1
} else
 absVal = a
y.exp = BitWidth( absVal ) – FPANumDigitsVal (27)
if( y.exp < 0 )
 y.val = absVal << ( –y.exp )
else
 y.val = absVal >> y.exp

#### Process for converting a floating-point approximation to an integer

Input to this process is a floating-point approximation x = (x.val, x.exp, x.sgn).

Output of this process is an integer a representing the floating-point approximation x rounded to an integer value.

The value of a is initially set to 0.

When x.val is not equal to 0, the value of a is modified as specified by the following pseudo-code:

if( x.exp >= 0 )
 a = x.val  <<  x.exp
else
 a = ( x.val + ( 1  <<  ( –x.exp – 1 ) ) )  >>  ( –x.exp ) (28)
if( x.sgn )
 a = –a

#### Process for shifting a floating-point approximation to the left

Inputs to this process are:

– a floating-point approximation x = (x.val, x.exp, x.sgn), and

– a non-negative integer number a representing a bit shift to the left.

Output of this process is a floating-point approximation y = (y.val, y.exp, y.sgn) representing the result of the bit shift.

The floating-point approximation y is determined as specified by the following pseudo-code:

y.val   = x.val
y.exp = x.exp + a (29)
y.sgn = x.sgn

#### Process for shifting a floating-point approximation to the right

Inputs to this process are:

– a floating-point approximation x = (x.val, x.exp, x.sgn), and

– a non-negative integer number a representing a bit shift to the right.

Output of this process is a floating-point approximation y = (y.val, y.exp, y.sgn) representing the result of the bit shift.

The floating-point approximation y is determined as specified by the following pseudo-code:

y.val  = x.val
y.exp = x.exp – a (30)
y.sgn = x.sgn

#### Process for negating a floating-point approximation

Input to this process is a floating-point approximation x = (x.val, x.exp, x.sgn).

Output of this process is a floating-point approximation y = (y.val, y.exp, y.sgn) representing the result of the negation of x.

The floating-point approximation y is determined as specified by the following pseudo-code:

y.val  = x.val
y.exp = x.exp (31)
y.sgn = 1 – x.sgn

#### Process for adding two floating-point approximations

Input to this process are:

– a floating-point approximation x = (x.val, x.exp, x.sgn) representing the first summand;

– a floating-point approximation y = (y.val, y.exp, y.sgn) representing the second summand.

Output of this process is a floating-point approximation z = (z.val, z.exp, z.sgn) representing the result of the addition of x and y.

If x.val is equal to 0, z is set equal to y, i.e., z.val = y.val, z.exp = y.exp, and z.sgn = y.sgn.

Otherwise, if y.val is equal to 0, z is set equal to x, i.e., z.val = x.val, z.exp = x.exp, and z.sgn = x.sgn.

Otherwise (x.val is not equal to 0 and y.val is not equal to 0), the value of z is first initialized with z.val = 0, z.exp = 0, and z.sgn = 0 and then updated as specified by the following pseudo-code:

if( x.exp >= y.exp ) {
 s = x.exp – y.exp
 a = ( y.val + ( ( 1  <<  s ) – 1 ) )  >>  s
 b = ( 1 – 2 \* x.sgn ) \* x.val + ( 1 – 2 \* y.sgn ) \* a
 c = x.exp
} else {
 s = y.exp – x.exp
 a = ( x.val + ( ( 1  <<  s ) – 1 ) )  >>  s
 b = ( 1 – 2 \* x.sgn ) \* a + ( 1 – 2 \* y.sgn ) \* y.val (32)
 c = y.exp
}
if( b < 0 ) {
 b = –b
 z.sgn = 1
}
if( b > 0 ) {
 d = BitWidth( b ) – FPANumDigitsVal
 if( d < 0 )
 z.val = b  <<  ( –d )
 else
 z.val = b  >>  d
 z.exp = c +d
}

#### Process for subtracting two floating-point approximations

Input to this process are:

– a floating-point approximation x = (x.val, x.exp, x.sgn) representing the minuend;

– a floating-point approximation y = (y.val, y.exp, y.sgn) representing the subtrahend.

Output of this process is a floating-point approximation z = (z.val, z.exp, z.sgn) representing the result of subtracting y from of x.

The floating point approximation yn is derived by invoking the process for negating a floating-point approximation as specified in clause 5.10.3.5 with y as input and the output is assigned to yn.

The floating point approximation z is derived by invoking the process for adding two floating-point approximations as specified in clause 5.10.3.6 with x and yn as inputs and the output is assigned to z.

#### Process for multiplying two floating-point approximations

Input to this process are:

– a floating-point approximation x = (x.val, x.exp, x.sgn) representing the multiplier;

– a floating-point approximation y = (y.val, y.exp, y.sgn) representing the multiplicand.

Output of this process is a floating-point approximation z = (z.val, z.exp, z.sgn) representing the result of the multiplication of x and y.

If x.val is equal to 0 or y.val is equal to 0, z is set equal to zero, i.e., z.val = 0.

Otherwise (x.val is not equal to 0 and y.val is not equal to 0), the floating-point approximation z is derived as specified by the following pseudo-code:

HSHIFT = FPANumDigitsVal  >>  1
LMASK = ( 1  <<  HSHIFT )  –  1
ah = x.val  >>  HSHIFT
bh = y.val  >>  HSHIFT (33)
al  = x.val  &  LMASK
bl  = y.val  &  LMASK
z.val  = ( ( ah \* bh )  <<  1 ) + ( ( ah \* bl + al \* bh + ( ( al \* bl )  >>  HSHIFT ) )  >>  ( HSHIFT – 1 ) )
z.exp = x.exp + y.exp + FPANumDigitsVal – 1
z.sgn = x.sgn ^ y.sgn
if( z.val >= ( 1  <<  FPANumDigitsVal ) ) {
 z.val  = z.val  >> 1
 z.exp = z.exp + 1
}

#### Process for deriving the reciprocal of a floating-point approximation

Input to this process is a floating-point approximation x = (x.val, x.exp, x.sgn) with x.val being not equal to 0.

Output of this process is a floating-point approximation y = (y.val, y.exp, y.sgn) representing the reciprocal of x.

The floating-point approximation y is derived as specified by the following pseudo-code:

DSHIFT = FPANumDigitsVal – FPALog2DivTabSize – 1
DADD   = ( 1  <<  ( DSHIFT – 1 ) )  –  ( 1  <<  ( FPANumDigitsVal  – 1 ) )
DMASK = ( 1  <<  FPALog2DivTabSize ) – 1
EXSUB  = FPANumDigitsDivTab + FPANumDigitsVal – 2
a = ( x.val + DADD )  >>  DSHIFT (34)
b = FPADivTab[ a & DMASK ]
c = BitWidth( b ) – FPANumDigitsVal
y.val = b  <<  ( –c )
y.exp = c – x.exp – ( a  >>  FPALog2DivTabSize ) – EXSUB
y.sgn = x.sgn

#### Process for dividing two floating-point approximations

Input to this process are:

– a floating-point approximation x = (x.val, x.exp, x.sgn) representing the dividend;

– a floating-point approximation y = (y.val, y.exp, y.sgn), with y.val being not equal to 0, representing the divisor.

Output of this process is a floating-point approximation z = (z.val, z.exp, z.sgn) representing the result of the division of x and y.

The floating point approximation yr is derived by invoking the process for deriving the reciprocal of a floating-point approximation as specified in clause 5.10.3.9 with y as input and the output is assigned to yr.

The floating point approximation z is derived by invoking the process for multiplying two floating-point approximations as specified in clause 5.10.3.10 with x and yr as inputs and the output is assigned to z.

### Processes for solving linear equation systems using floating-point approximations

#### Derivation process for a sub vector

Inputs to this process are:

– an N-dimensional vector vec of floating-point approximations, with N > 1;

– an index r.

Output of this process is an (N – 1)-dimensional vector subVec that is obtained by deleting the r-th element from the vector vec.

The (N – 1)-dimensional vector subVec is obtained as specified by the following peudo-code:

for( k = 0; k < N – 1; k++ ) (35)
 subVec[ k ] = vec[ k + ( k >= r ? 1 : 0 ) ]

#### Derivation process for a sub-matrix

Inputs to this process are:

– an NxN matrix mat of floating-point approximations, with N > 1;

– a row index r; and

– a column index c.

Output of this process is an (N – 1)x(N – 1) matrix subMat that is obtained by deleting the r-th row and the c-th column from the matrix mat.

The (N – 1)x(N – 1) matrix subMat is obtained as specified by the following peudo-code:

for( k = 0; k < N – 1; k++ )
 for( n = 0; n < N – 1; n++ ) (36)
 subMat[ k ][ n ] = mat[ k + ( k >= c ? 1 : 0 ) ][ n + ( n >= r ? 1 : 0 ) ]

#### Derivation process for the determinant of a matrix

Inputs to this process is an NxN matrix mat of floating-point approximations, with N > 0.

Output of this process is a floating-point approximation det that represents the determinant of the matrix mat.

If N is equal to 1, the determinant det is set equal to det = x[ 0 ][ 0 ].

Otherwise (N is greater than 1), the determinant det is derived as specified by the following recursive process:

– The determinant is set equal to det = FPNum( 0 ).

– For k proceeding over the range from 0 to N – 1, inclusive, the following applies:

* The (N – 1)x(N – 1) sub-matrix subMat is derived by invoking the derivation process for a sub-matrix as specified in clause 5.10.4.2 with mat, the row index k, and the column index 0 as inputs and the output is assigned to the sub-matrix subMat.
* The determinant subDet of the (N – 1)x(N – 1) sub-matrix subMat is derived by invoking the derivation process for the determinant of a matrix as specified in this clause with subMat as input and the output is assigned to subDet.
* Depending on k, the determinant det is updated as specified by the following pseudo-code:

if( ( k & 1 )  = =  1 )
 det = det – mat[ 0 ][ k ] \* subDet (37)
else
 det = det + mat[ 0 ][ k ] \* subDet

#### Process for solving a linear equation system with floating-point matrices and vectors

Inputs to this process are:

– an NxN matrix mat of floating-point approximations, with N > 0;

– a N-dimensional vector vec of floating-point approximations;

– an unsigned integer bshift specifying the precision of the output.

Output of this process is an N-dimensional vector res of integer values.

The determinant detMat of the matrix mat is determined by invoking the derivation process for the determinant of a matrix as specified in clause 5.10.4.3 with mat as input and the output is assigned to detMat.

If detMat.val is equal to 0, the value of res[ 0 ] is set equal to res[ 0 ] = 0.

Otherwise (detMat.val is not zero), the value of res[ 0 ] is derived as follows:

– The NxN matrix matX of floating-point approximations is derived as specifified by the following pseudo-code:

for( k = 0; k < N; k++ )
 matX[ 0 ][ k ] = vec[ k ]
for( c = 1; c < N; c++ ) (38)
 for( k = 0; k < N; k++ )
 matX[ c ][ k ] = mat[ c ][ k ]

– The determinant detX of the matrix matX is determined by invoking the derivation process for the determinant of a matrix as specified in clause 5.10.4.3 with matX as input and the output is assigned to detX.

– The value of res[ 0 ] is derived by:

res[0] = FPAToInteger( ( detX / detMat )  <<  bshift ) (39)

When N is greater than 1, the values res[ k ] with k being in the range of 1 to N – 1, inclusive, are determined by the following recursive process:

– The (N – 1)x(N – 1) sub-matrix subMat is determined by invoking the derivation process for a sub-matrix as specified in clause 5.10.4.2 with mat, the row index 0, and the column index 0 as inputs and the output is assigned to the sub-matrix subMat.

– The (N – 1)-dimensional sub-vector subVec is determined by invoking the derivation process for a sub-vector as specified in clause 5.10.4.1 with vec and the index 0 as inputs and the output is assigned to the sub-vector subVec.

– The (N – 1)-dimensional sub-vector subVec is modified as specified by the following pseudo-code:

scale = FPNum( res[ 0 ] )  >>  bshift
for( k = 0; k < N – 1; k++ ) (40)
 subVec[ k ] = subVec[ k ] – scale \* mat[ 0 ][ 1 + k ]

– The process for solving a linear equation system with floating-point matrices and vectors as specified in this clause is invoked with the (N – 1)x(N – 1) matrix subMat, the (N – 1)-dimensional vector subVec, and the integer number bshift as input and the output is assigned to the (N – 1)-dimensional integer vector subRes.

– The values res[ k ] with k being in the range of 1 to N – 1, inclusive, are determined by:

for( k = 1; k < N; k++ ) (41)
 res[ k ] = subRes[ k – 1 ]

#### Process for solving a linear equation system in integer arithmetic

Inputs to this process are:

– an NxN matrix mat of integer values, with N > 0;

– a N-dimensional vector vec of integer values;

– an unsigned integer bshift specifying the precision of the output.

Output of this process is an N-dimensional vector res of integer values.

The NxN matrix fpaMat of floating-point approximations is derived as specified by the following pseudo-code:

for( c = 0; c < N; c++ )
 for( r = 0; r < N; r++ ) (42)
 fpaMat[ c ][ r ] = FPApprox( mat[ c ][ r ] )

The N-dimensional vector fpaVec of floating-point approximations is derived as specified by the following pseudo-code:

for( k = 0; k < N; k++ ) (43)
 fpaVec[ k ] = FPApprox ( vec[ k ] )

The N-dimensional output vector res is derived by invoking the process for solving a linear equation system with floating-point matrices and vectors as specified in clause 5.10.4.4 with fpaMat, fpaVec, and bshift as inputs and the output is assigned to res.

## 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 descriptor 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 and semantics use the values of other variables derived from the values of syntax elements. 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 could, in some cases, 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 could contain more upper case letters.

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

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in clause 7.2 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 5.8) 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 could be denoted either as s[ x ][ y ] or as syx. A single column of a matrix could 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 could be referred to as the list s[ x ].

A specification of values of the entries in rows and columns of an array could 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", is used in some cases 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 operations

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

is typically 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

is typically described in the following manner:

... 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

is typically 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 upper case variables that pertain to the current syntax structure and depending syntax structures are available in the process specification and invoking. A process specification might 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 is sometimes referred to by the variable name having a value equal to the address of the specific coding block.

# Bitstream and waveform signal format

# 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 might be specified, either directly or indirectly, in other clauses.

NOTE – An actual decoder is expected to implement some means for identifying entry points into the bitstream and some means to identify and handle non-conforming bitstreams. The methods for identifying and handling errors and other such situations are not specified in this Specification.

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.

|  | Descriptor |
| --- | --- |
| /\* A statement can be a syntax element with an associated descriptor 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(k) |
| 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 |  |

## Specification of syntax functions and descriptors

The functions presented in this clause are used in the specification of the syntax. 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 a byte-aligned position, i.e., the current position is an integer multiple of 8 bits from the position of the first bit in the bitstream, the return value of byte\_aligned( ) is equal to TRUE.

– Otherwise, the return value of byte\_aligned( ) is equal to FALSE.

more\_data\_in\_byte\_stream( ), which is used only in the byte stream NAL unit syntax structure specified in Annex B, is specified as follows:

– If more data follow in the byte stream, the return value of more\_data\_in\_byte\_stream( ) is equal to TRUE.

– Otherwise, the return value of more\_data\_in\_byte\_stream( ) is equal to FALSE.

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

– If byte\_aligned( ) is equal to TRUE and the current position in the sei\_payload( ) or vui\_payload( ) syntax structure is 8 \* payloadSize bits from the beginning of the syntax structure, the return value of more\_data\_in\_payload( ) is equal to FALSE.

– Otherwise, the return value of more\_data\_in\_payload( ) is equal to TRUE.

more\_rbsp\_data( ) is specified as follows:

– If there is no more data in the raw byte sequence payload (RBSP), the return value of more\_rbsp\_data( ) is equal to FALSE.

– Otherwise, the RBSP data are searched for the last (least significant, right-most) bit equal to 1 that is present in the RBSP. Given the position of this bit, which is the first bit (rbsp\_stop\_one\_bit) of the rbsp\_trailing\_bits( ) syntax structure, the following applies:

– If there is more data in an RBSP before the rbsp\_trailing\_bits( ) syntax structure, the return value of more\_rbsp\_data( ) is equal to TRUE.

– Otherwise, the return value of more\_rbsp\_data( ) is equal to FALSE.

The method for enabling determination of whether there is more data in the RBSP is specified by the application (or in Annex B for applications that use the byte stream format).

more\_rbsp\_trailing\_data( ) is specified as follows:

– If there is more data in an RBSP, the return value of more\_rbsp\_trailing\_data( ) is equal to TRUE.

– Otherwise, the return value of more\_rbsp\_trailing\_data( ) is equal to FALSE.

next\_bits( n ) provides the next bits in the bitstream for comparison purposes, without advancing the bitstream pointer. Provides a look at the next n bits in the bitstream with n being its argument. When used within the byte stream format as specified in Annex B and fewer than n bits remain within the byte stream, next\_bits( n ) returns a value of 0.

payload\_extension\_present( ) is specified as follows:

– If the current position in the sei\_payload( ) or vui\_payload( ) syntax structure is not the position of the last (least significant, right-most) bit that is equal to 1 that is less than 8 \* payloadSize bits from the beginning of the syntax structure (i.e., the position of the sei\_payload\_bit\_equal\_to\_one or vui\_payload\_bit\_equal\_to\_one syntax element), the return value of payload\_extension\_present( ) is equal to TRUE.

– Otherwise, the return value of payload\_extension\_present( ) is equal to FALSE.

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

The following descriptors specify the parsing process of each syntax element:

– ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in clause 9.3.

– b(8): byte having any pattern of bit string (8 bits). The parsing process for this descriptor is specified by the return value of the function read\_bits( 8 ).

– f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this descriptor 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 descriptor is specified by the return value of the function read\_bits( n ) interpreted as a two's complement integer representation with most significant bit written first.

– se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k equal to 0.

– u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this descriptor is specified by the return value of the function read\_bits( n ) interpreted as a binary representation of an unsigned integer with most significant bit written first.

– ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this descriptor is specified in clause 9.2 with the order k equal to 0.

## Syntax in tabular form

### NAL unit syntax

#### General NAL unit syntax

|  |  |
| --- | --- |
| nal\_unit( NumBytesInNalUnit ) { | Descriptor |
| nal\_unit\_header( ) |  |
|  NumBytesInRbsp = 0 |  |
|  for( i = 1; i < NumBytesInNalUnit; i++ ) |  |
|  if( i + 2 < NumBytesInNalUnit && next\_bits( 24 ) = = 0x000003 ) { |  |
|  **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
|  **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
|  i += 2 |  |
|  **emulation\_prevention\_three\_byte** /\* equal to 0x03 \*/ | f(8) |
|  } else |  |
|  **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| } |  |

#### NAL unit header syntax

|  |  |
| --- | --- |
| nal\_unit\_header( ) { | Descriptor |
|  **nal\_unit\_type** | u(8) |
| } |  |

### Raw byte sequence payloads, trailing bits and byte alignment syntax

#### Waveform parameter set RBSP syntax

|  |  |
| --- | --- |
| waveform\_parameter\_set\_rbsp( ) { | Descriptor |
|  **wps\_waveform\_parameter\_set\_id** | u(4) |
|  NumChannelGroups = 0 |  |
|  TotalNumChannels = 0 |  |
|  do { |  |
|  **wps\_num\_channels\_in\_next\_group\_minus1** | ue(v) |
|  **wps\_num\_channel\_group\_repetitions** | ue(v) |
|  for( j = 0; j <= wps\_num\_channel\_group\_repetitions; j++ ) { |  |
|  NumChannels[ NumChannelGroups ] = wps\_num\_channels\_in\_next\_group\_minus1 + 1 |  |
|  ChannelGroupStartingPos[ NumChannelGroups++ ]= TotalNumChannels |  |
|  TotalNumChannels += wps\_num\_channels\_in\_next\_group\_minus1 + 1 |  |
|  } |  |
|  **wps\_more\_channel\_groups\_present\_flag** | u(1) |
|  } while( wps\_more\_channel\_groups\_present\_flag ) |  |
|  **wps\_channel\_reordering\_flag** | u(1) |
|  if( wps\_channel\_reordering\_ flag ) { |  |
|  **wps\_num\_channel\_swaps\_minus1** | ue(v) |
|  for( i = 0; i <= wps\_num\_channel\_swaps\_minus1; i++ ) { |  |
|  **wps\_swap\_frst\_idx**[ i ] | ue(v) |
|  **wps\_swap\_scnd\_idx\_min\_frst\_idx\_min1**[ i ] | ue(v) |
|  } |  |
|  } |  |
|  **wps\_num\_annotation\_channels** | ue(v) |
|  for( j = 0; j < wps\_num\_annotation\_channels; j++ ) |  |
|  AnnotationChannelNumSamples[ j ] = 0 | ue(v) |
|  rbsp\_trailing\_bits( ) |  |
| } |  |

#### Independent frame RBSP syntax

|  |  |
| --- | --- |
| independent\_frame\_rbsp( ) { | Descriptor |
|  **if\_waveform\_parameter\_set\_id** | u(4) |
|  if( NumChannelGroups > 1 ) |  |
|  **if\_channel\_group\_id** /\* Consider changing to fixed length depending on NumChannelGroups\*/ | ue(v) |
|  **if\_length\_signal\_mode\_flag** | u(1) |
|  **if\_frame\_length\_shift** | u(2) |
|  **if\_max\_min\_block\_size** | u(6) |
|  **if\_perceptual\_mode** | u(2) |
|  **if\_max\_min\_bit\_depth** | u(6) |
|  **if\_allow\_cross\_channel\_pred\_flag** | u(1) |
|  if( if\_allow\_cross\_channel\_pred\_flag ) { |  |
|  **if\_cc\_pred\_filtering\_mode** | u(2) |
|  **if\_allow\_cc\_pred\_mult\_hyp\_flag** | u(1) |
|  for( ch = 1; ch < NumChannels[ if\_channel\_group\_id ]; ch++ ) { |  |
|  for( n = 0; n <= if\_allow\_cc\_pred\_mult\_hyp\_flag; n++ ) { |  |
| CrossChannelPredInputChDistMinus1[ ch ][ n ] = 0 |  |
| } |  |
| } |  |
| } |  |
|  **if\_allow\_block\_matching\_pred\_flag** | u(1) |
|  if( if\_allow\_block\_matching\_pred\_flag ) { |  |
|  **if\_bm\_pred\_filtering\_mode** | u(2) |
|  **if\_allow\_bm\_pred\_mult\_hyp\_flag** | u(1) |
|  **if\_allow\_bm\_offset\_pred\_prev\_ch\_flag** | u(1) |
|  for( ch = 0; ch < NumChannels[ if\_channel\_group\_id ]; ch++ ) { |  |
|  for( n = 0; n <=if\_allow\_bm\_pred\_mult\_hyp\_flag; n++ ) { |  |
| BlockMatchingPredOffsetMinusBlocksSize[ ch][ n ] = 0 |  |
|  Log2BlockMatchingPredBlockSize[ ch][ n ] = 0 |  |
| } |  |
| } |  |
|  } |  |
|  **if\_allow\_lpf** | u(1) |
|  if( if\_ allow\_lpf ){ |  |
|  **if\_lpf\_allow\_prev\_ch\_flag** | u(1) |
|  LPFMaxNumWeightsNoPrevCh = 16 |  |
|  for( ch = 0; ch < NumChannels[ if\_channel\_group\_id ]; ch++ ) { |  |
|  for( n = 0; n <=LPFMaxNumWeightsNoPrevCh; n++ ) { |  |
|  LPFWeightsNoPrevChPred[ ch ] [ n ] = 0 |  |
|  } |  |
|  } |  |
|  } |  |
|  **if\_residual\_quant\_mode** | u(2) |
|  **if\_ch\_indep\_interval\_idx** | u(4) |
|  DepChMask = ( 2 << if\_ch\_indep\_interval\_idx ) – 1 |  |
|  **if\_max\_abs\_delta\_qp\_idx** | u(3) |
|  MaxAbsDeltaQP = ( 1 << if\_max\_abs\_delta\_qp\_idx ) – 1 |  |
|  if( if\_length\_signal\_mode\_flag ) |  |
|  **if\_indep\_num\_samples\_per\_channel\_minus1** | u(32) |
|  **if\_indep\_init\_block\_qp** | u(8) |
|  **if\_ctx\_init\_flag** | u(1) |
|  if( if\_ctx\_init\_flag ) |  |
|  **if\_ctx\_init\_mode** | u(1) |
|  for( i = 0; i < NumChannels[ if\_channel\_group\_id ]; i++ ) { |  |
|  CurrBlockQP[ i ] = if\_indep\_init\_block\_qp |  |
|  CurrZeroLSB[ i ] = 0 |  |
|  } |  |
|  byte\_alignment( ) |  |
|  frame\_data( NumChannels[ if\_channel\_group\_id ] ) |  |
|  rbsp\_trailing\_bits( ) |  |
| } |  |

#### Dependent frame RBSP syntax

|  |  |
| --- | --- |
| dependent\_frame\_rbsp( ) { | Descriptor |
|  **df\_waveform\_parameter\_set\_id** | u(4) |
|  if( NumChannelGroups < 16 ) |  |
|  **df\_channel\_group\_id\_plus1** | u(4) |
|  else if( NumChannelGroups < 4096 ) |  |
|  **df\_channel\_group\_id\_plus1** | u(12) |
|  else |  |
|  **df\_channel\_group\_id\_plus1** | u(20) |
| frame\_data( NumChannels[ df\_channel\_group\_id\_plus1 - 1 ] ) |  |
|  rbsp\_trailing\_bits( ) |  |
| } |  |

#### Annotation channel RBSP syntax

|  |  |
| --- | --- |
| annotation\_channel\_rbsp( ) { | Descriptor |
|  **ac\_waveform\_parameter\_set\_id** | u(4) |
|  **ac\_annotation\_channel\_id** | ue(v) |
|  **ac\_num\_annotation\_bytes\_div2\_minus1** | ue(v) |
|  byte\_alignment() |  |
|  annotation\_channel\_data( ) |  |
| } |  |

#### Auxiliary metadata RBSP syntax

|  |  |
| --- | --- |
| auxiliary\_metadata\_rbsp( ) { | Descriptor |
|  **am\_fourcc\_id\_last\_three\_bytes** /\* Equal to 0x415743\*/ | u(24) |
|  **am\_header\_crc32** | u(32) |
|  **am\_reserved\_flag** | u(1) |
|  **am\_waveform\_type** | u(2) |
|  **am\_length\_signal\_mode** | u(1) |
|  **am\_allow\_reconfig\_flag** | u(1) |
|  **am\_copyright\_flag** | u(1) |
|  **am\_original\_flag** | u(1) |
|  **am\_private\_flag** | u(1) |
|  **am\_stream\_max\_sampling\_rate\_minus1** | u(24) |
|  **am\_stream\_max\_num\_channels\_minus1** | u(16) |
|  if( am\_length\_signal\_mode ) |  |
|  **am\_stream\_num\_samples\_per\_ch** | u(32) |
|  if( am\_waveform\_type = = WT\_BS2088 ) { |  |
|  **am\_metadata\_reserved\_flag** | u(1) |
|  **am\_metadata\_num\_bytes\_minus1** | u(31) |
|  for( i = 0; i <= am\_metadata\_num\_bytes\_minus1; i++ ) |  |
|  **am\_metadata\_payload\_bytes**[ i ] | u(8) |
|  } else if( am\_waveform\_typt = = WT\_EDF\_PLUS ) { |  |
|  **am\_num\_channels\_edf** | u(16) |
|  for( i = 0; i < 256 \* ( am\_num\_channels\_edf + 1); i++ ) |  |
|  **am\_edf\_header\_payload\_bytes**[ i ] | u(8) |
|  } |  |
| } |  |

#### RBSP trailing bits syntax

|  |  |
| --- | --- |
| rbsp\_trailing\_bits( ) { | Descriptor |
|  **rbsp\_stop\_one\_bit** /\* equal to 1 \*/ | f(1) |
|  while( !byte\_aligned( ) ) |  |
|  **rbsp\_alignment\_zero\_bit** /\* equal to 0 \*/ | f(1) |
| } |  |

#### Byte alignment syntax

|  |  |
| --- | --- |
| byte\_alignment( ) { | Descriptor |
|  **byte\_alignment\_bit\_equal\_to\_one** /\* equal to 1 \*/ | f(1) |
|  while( !byte\_aligned( ) ) |  |
|  **byte\_alignment\_bit\_equal\_to\_zero** /\* equal to 0 \*/ | f(1) |
| } |  |

### Frame data syntax

#### General frame data syntax

|  |  |
| --- | --- |
| frame\_data( numChannels ) { | Descriptor |
|  FrameNumSamplesPerChannel = 0 |  |
|  do { |  |
|  if( FrameNumSamplesPerChannel > 0 ) |  |
|  **end\_of\_frame\_sequence\_flag** | ae(v) |
|  if( !end\_of\_frame\_sequence\_flag ) { |  |
|  if( MaxSplitDepth > 0 ) |  |
|  **block\_split\_log2** | ae(v) |
|  Log2BlockSize = Log2MaxBlockSize - block\_split\_log2 |  |
|  for( ch = 0; ch < numChannels; ch++ ) { |  |
|  if( if\_allow\_block\_matching\_pred\_flag | | ( if\_allow\_cross\_channel\_pred\_flag && ( ch &DepChMask ) > 0 ) ) |  |
|  **block\_matching\_or\_cross\_channel\_pred\_flag** | ae(v) |
|  if( block\_matching\_or\_cross\_channel\_pred\_flag) { |  |
|  if( if\_allow\_block\_matching\_pred\_flag && if\_allow\_cross\_channel\_pred\_flag && ( ch & DepChMask ) > 0 ) |  |
|  **cross\_channel\_pred\_flag**  | ae(v) |
|  if( cross\_channel\_pred\_flag ) |  |
|  cross\_channel\_prediction\_data( ch ) |  |
|  else |  |
|  block\_matching\_prediction\_data( ch ) |  |
|  }else |  |
|  **block\_pred\_mode** | ae(v) |
|  if( block\_matching\_or\_cross\_channel\_pred\_flag | | ( block\_pred\_mode = = BPM\_OFF ) ) { |  |
|  sample\_pred\_mode( ) |  |
|  if( spred\_lpf\_flag ) |  |
|  linear\_predictive\_filtering\_data( ch ) |  |
|  } |  |
|  if( if\_max\_abs\_delta\_qp\_idx > 0 ) { |  |
|  **block\_abs\_delta\_qp** | ae(v) |
|  if( block\_abs\_delta\_qp > 0 ) |  |
|  **block\_delta\_qp\_sign\_flag** | ae(v) |
|  blockDeltaQP = block\_delta\_qp\_sign\_flag ? –block\_abs\_delta\_qp : block\_abs\_delta\_qp  |  |
|  CurrBlockQP[ ch ] = Clip3( 0, 255, CurrBlockQP[ ch ] + blockDeltaQP ) |  |
|  } |  |
|  if( spred\_lpf\_or\_diff\_flag | | ( spred\_rem\_mode\_idx = = 2 ) ) { |  |
|  if( spred\_lpf\_or\_diff\_flag && CurrBlockQP[ ch ] = = 0 ) { |  |
|  **block\_delta\_zlsb\_present\_flag** | ae(v) |
|  if(block\_delta\_zlsb\_present\_flag ){ |  |
|  **block\_delta\_zlsb\_sign\_flag** | ae(v) |
|  deltaLSB = block\_delta\_zlsb\_sign\_flag ? -1 : 1 |  |
|  CurrZeroLSB[ ch ] = Clip3( 0, BitDepthMax - BitDepthMin, CurrZeroLSB[ ch ] + deltaLSB ) |  |
|  } |  |
|  } else { |  |
|  **transform\_skip\_flag** | ae(v) |
|  if ( !transform\_skip\_flag ) |  |
|  **transform\_dst\_flag** | ae(v) |
|  TransformMode = transform\_skip\_flag ? TM\_OFF :  ( transform\_dst\_flag = = 1 ? TM\_DST : TM\_DCT ) |  |
|  } |  |
|  } |  |
|  } |  |
|  quant\_res\_sample\_data( ) |  |
|  } |  |
|  FrameNumSamplesPerChannel += 1 << Log2BlockSize |  |
|  **end\_of\_truncated\_frame\_sequence\_flag** | ae(v) |
|  if( end\_of\_truncated\_frame\_sequence\_flag ) |  |
|  **num\_samples\_per\_channel\_to\_discard** | u(v) |
|  if( !end\_of\_truncated\_frame\_sequence\_flag && FrameNumSamplesPerChannel = = ( 1 << Log2FrameLength ) ) |  |
|  **end\_of\_frame\_one\_bit**  /\* equal to 1 \*/ | ae(v) |
|  } |  |
|  } while( !end\_of\_frame\_sequence\_flag && !end\_of\_truncated\_frame\_sequence\_flag && !end\_of\_frame\_one\_bit ) |  |
| } |  |

#### Cross channel prediction data syntax

|  |  |
| --- | --- |
| cross\_channel\_prediction\_data( ch ) { | Descriptor |
|  **cc\_pred\_offset\_only\_flag** | ae(v) |
|  if( if\_cc\_pred\_filtering\_mode > 0 **)** |  |
|  **cc\_pred\_filter\_flag** | ae(v) |
|  if( if\_cc\_pred\_filtering\_mode = = 2 && cc\_pred\_filter\_flag ) |  |
|  **cc\_pred\_filter\_idx** | ae(v) |
|  if( (ch & DepChMask) > 1 ) { |  |
|  if( if\_allow\_cc\_pred\_mult\_hyp\_flag ) |  |
|  **cc\_pred\_mult\_hyp\_flag** | ae(v) |
|  if( !cc\_pred\_mult\_hyp\_flag | | ( ch & DepChMask ) > 2 ) { |  |
|  for( n = 0; n <= cc\_pred\_mult\_hyp\_flag; n++ ) { |  |
|  **cc\_pred\_abs\_chd\_greater0\_flag**[ n ] | ae(v) |
|  if( cc\_pred\_abs\_chd\_greater0\_flag[ n ] ) { |  |
|  **cc\_pred\_abs\_chd\_minus1**[ n ] | ae(v) |
|  **cc\_pred\_chd\_sign\_flag**[ n ] | ae(v) |
|  } |  |
|  cCPredInputChDistDiffSign = cc\_pred\_chd\_sign\_flag[ n ] ? –1 : 1 |  |
|  cCPredInputChDistDiff = cc\_pred\_abs\_chd\_greater0\_flag[ n ] ? cCPredInputChDistDiffSign \* (cc\_pred\_abs\_chd\_minus1[ n ] +1 ) : 0 |  |
|  CrossChannelPredInputChDistMinus1[ ch ][ n ] = max( 0, min ( ( ch & DepChMask ) – 1, cCPredInputChDistDiff + CrossChannelPredInputChDistMinus1[ ch ][ n ] ) ) |  |
|  } |  |
|  } |  |
|  } |  |
| } |  |

#### Block matching prediction data syntax

|  |  |
| --- | --- |
| block\_matching\_prediction\_data( ch ) { | Descriptor |
|  if( if\_allow\_bm\_pred\_mult\_hyp\_flag **)** |  |
|  **bm\_pred\_mult\_hyp\_flag** | ae(v) |
|  **bm\_pred\_add\_offset\_flag** | ae(v) |
|  for( n = 0; n <= bm\_pred\_mult\_hyp\_flag; n++ ) { |  |
|  if( if\_bm\_pred\_filtering\_mode > 0 **)** |  |
|  **bm\_pred\_filter\_flag**[ n ] | ae(v) |
|  if( bm\_pred\_filter\_flag[ n ] && if\_bm\_pred\_filtering\_mode = = 2 ) |  |
|  **bm\_pred\_filter\_idx**[ n ] | ae(v) |
|  if( if\_allow\_bm\_offset\_pred\_prev\_ch\_flag && ( ch & DepChMask ) > 0 ) |  |
|  **bm\_pred\_off\_pred\_prev\_ch\_flag**[ n ] | ae(v) |
|  **bm\_pred\_abs\_offd\_greater0\_flag**[ n ] | ae(v) |
|  if( bm\_pred\_abs\_offd\_greater0\_flag[ n ] ) { |  |
|  **bm\_pred\_abs\_offd\_minus1**[ n ] | ae(v) |
|  **bm\_pred\_offd\_sign\_flag**[ n ] | ae(v) |
|  } |  |
|  bmOffsetMinusBlockSizeDiffSign = bm\_pred\_offd\_sign\_flag[ n ] ? –1 :1  |  |
|  bmOffsetMinusBlockSizeDiff = bm\_pred\_abs\_offd\_greater0\_flag[ n ] ? bmOffsetMinusBlockSizeDiffSign \*( bm\_pred\_abs\_offd\_minus1 + 1 ) : 0 |  |
|  bmOffsetMinusBlockSizePred = bm\_pred\_off\_pred\_prev\_ch\_flag[ n ] ? BlockMatchingPredOffsetMinusBlocksSize [ ch – 1 ][ n ] : BlockMatchingPredOffsetMinusBlocksSize [ ch ][ n ] |  |
|  log2BMBlockSizePred = bm\_pred\_off\_pred\_prev\_ch\_flag[ n ] ? Log2BlockMatchingPredBlockSize[ ch – 1 ][ n ] : Log2BlockMatchingPredBlockSize[ ch ][ n ]  |  |
|  if( ( log2BMBlockSizePred = = Log2BlockSize ) | | ( log2BMBlockSizePred = = 0 ) ) { |  |
|  BlockMatchingPredOffsetMinusBlocksSize[ ch ][ n ] = bmOffsetMinusBlockSizePred + bmOffsetMinusBlockSizeDiff  |  |
|  } else { |  |
|  bmOffsetPred = bmOffsetMinusBlockSizePred + ( 1 << log2BMBlockSizePred ) |  |
|  if( bmOffsetPred < ( 1 << Log2BlockSize ) ) |  |
|  bmOffsetPred = bmOffsetPred << (Log2BlockSize – log2BMBlockSizePred) |  |
|  BlockMatchingPredOffsetMinusBlocksSize[ ch ][ n ] = bmOffsetPred – ( 1 << Log2BlockSize ) + bmOffsetMinusBlockSizeDiff |  |
|  } |  |
|  BlockMatchingPredOffsetMinusBlocksSize[ ch ][ n ] = Clip3( 0, (1 << 16) – 1, BlockMatchingPredOffsetMinusBlocksSize[ ch ][ n ] )  |  |
|  Log2BlockMatchingPredBlockSize[ ch ][ n ] = Log2BlockSize |  |
|  } |  |
| } |  |

#### Sample pred mode syntax

|  |  |
| --- | --- |
| sample\_pred\_mode( ) { | Descriptor |
|  **spred\_lpf\_or\_diff\_flag** | ae(v) |
|  if( if\_ allow\_lpf && spred\_lpf\_or\_diff\_flag ) |  |
|  **spred\_lpf\_flag** | ae(v) |
|  if( !spred\_lpf\_or\_diff\_flag ) |  |
|  **spred\_rem\_mode\_idx** | ae(v) |
| } |  |

#### Linear predictive filtering data syntax

|  |  |
| --- | --- |
| linear\_predictive\_filtering\_data( ch ) { | Descriptor |
|  if**(** !block\_matching\_or\_cross\_channel\_pred\_flag && ( ch & DepChMask ) > 0 && if\_lpf\_allow\_prev\_ch\_flag **)** |  |
|  **lpf\_prev\_ch\_flag** | ae(v) |
|  if( !lpf\_prev\_ch\_flag ) |  |
|  **lpf\_delta\_coding\_flag** | ae(v) |
|  **lpf\_num\_weights\_idx** | ae(v) |
|  LPFNumWeightsCurr = 1 << ( lpf\_num\_weights\_idx + 1 ) + lpf\_prev\_ch\_flag ? min( 3, ch & DepChMask ) + 1 : 0 |  |
|  for( n = 0; n < LPFNumWeightsCurr; n++ ) { |  |
|  **abs\_lpf\_weight\_greater0\_flag**[ n ] | ae(v) |
| if( abs\_lpf\_weight\_greater0\_flag[ n ] ) { |  |
|  **abs\_lpf\_weight\_minus1**[ n ] | ae(v) |
|  **lpf\_weight\_sign\_flag**[ n ] | ae(v) |
| } |  |
|  lpfWeightSign = lpf\_weight\_sign\_flag[ n ] ? – 1 : 1 |  |
|  currentVal = abs\_lpf\_weight\_greater0\_flag[ n ] ? lpfWeightSign \* (abs\_lpf\_weight\_minus1[ n ] + 1 ) : 0 |  |
|  if(lpf\_delta\_coding\_flag**)** |  |
|  LPFWeightsCurr[ n ] = Clip3(– 64, 64,  currentVal + LPFWeightsNoPrevChPred[ ch ] [ n ] ) |  |
|  else |  |
|  LPFWeightsCurr[ n ] = currentVal |  |
|  } |  |
|  if( !lpf\_prev\_ch\_flag ){ |  |
|  for( n = 0; n <LPFMaxNumWeightsNoPrevCh; n++ ){ |  |
|  LPFWeightsNoPrevChPred[ ch ] [ n ] = (n < LPFNumWeightsCurr) ?  LPFWeightsCurr[ n ] : 0  |  |
|  } |  |
|  } |  |
| } |  |

#### Quant res sample data syntax

|  |  |
| --- | --- |
| quant\_res\_sample\_data( ) { | Descriptor |
|  NumQuantIndices = 1  <<  Log2BlockSize |  |
|  if( block\_pred\_mode = = BPM\_OFF && TransformMode = = TM\_OFF && sample\_pred\_mode = = SPM\_OFF ) { |  |
|  IntBitDepth = BitDepthMax – CurrZeroLSB[ ch ] |  |
|  offset = (1 << IntBitDepth ) >> 1 |  |
|  for( k = 0; k < NumQuantIndices; k = k + 1 ) { |  |
|  **coeff\_bypass\_value**[ k ] | ae(v) |
|  QuantIndices[ k ] = coeff\_bypass\_value[ k ] – offset  |  |
|  } |  |
|  } else if( TransformMode = = TM\_OFF ) { |  |
|  numLevels = 0 |  |
|  sumAbsLevels = 0 |  |
|  TSkipRiceParameter = 1 |  |
|  for( k = NumQuantIndices – 1; k >= 0; k = k – 1 ) { |  |
|  **abs\_tskip\_coeff\_gt0\_flag**[ k ] | ae(v) |
|  QuantIndices[ k ] = abs\_tskip\_coeff\_gt0\_flag[ k ] |  |
|  if( QuantIndices[ k ] > 0 ) { |  |
|  **abs\_tskip\_coeff\_offset**[ k ] | ae(v) |
|  QuantIndices[ k ] += abs\_tskip\_coeff\_offset[ k ] |  |
|  } |  |
|  if( QuantIndices[ k ] = = NumTSkipGtxFlags + 1 ) { |  |
|  **abs\_tskip\_coeff\_rem\_prefix**[ k ] | ae(v) |
|  QuantIndices[ k ] += ( abs\_tskip\_coeff\_rem\_prefix[ k ]  <<  RiceParameter ) |  |
|  if( abs\_tskip\_coeff\_rem\_prefix[ k ] < MaxTSkipRemPrefix ) |  |
|  if( TSkipRiceParameter > 0 ) { |  |
|  **abs\_tskip\_coeff\_rem\_fl\_suffix**[ k ] | ae(v) |
|  QuantIndices[ k ] += abs\_tskip\_coeff\_rem\_rice\_suffix[ k ] |  |
|  } |  |
|  else { |  |
|  **abs\_tskip\_coeff\_rem\_eg0\_suffix**[ k ] | ae(v) |
|  QuantIndices[ k ] += abs\_tskip\_coeff\_rem\_eg0\_suffix[ k ] |  |
|  } |  |
|  } |  |
|  if( QuantIndices[ k ] > 0 ) { |  |
|  **tskip\_coeff\_sign\_flag**[ k ] | ae(v) |
|  if( tskip\_coeff\_sign\_flag[ k ] ) |  |
|  QuantIndices[ k ] = –QuantIndices[ k ] |  |
|  } |  |
|  numLevels += 1 |  |
|  sumAbsLevels += Abs( QuantIndices[ k ] ) |  |
|  if( sumAbsLevels > 15 \* numLevels ) |  |
|  TSkipRiceParameter = 4 |  |
|  else if( sumAbsLevels > 5 \* numLevels ) |  |
|  TSkipRiceParameter = 3 |  |
|  else if( sumAbsLevels > 2 \* numLevels ) |  |
|  TSkipRiceParameter = 2 |  |
|  else |  |
|  TSkipRiceParameter = 1 |  |
|  } |  |
|  } else { |  |
|  **last\_sbb\_index\_gt0\_flag** | ae(v) |
|  if( last\_sbb\_index\_gt0\_flag ) { |  |
|  **last\_sbb\_index\_rem** | ae(v) |
|  last\_sbb\_index = 1 + last\_sbb\_index\_rem |  |
|  } else |  |
|  last\_sbb\_index = 0 |  |
|  **last\_index\_offset** | ae(v) |
|  last\_scan\_pos = ( last\_sbb\_index  <<  Log2SbbSize ) + last\_index\_offset |  |
|  for( k = 0; k < NumQuantIndices; k = k + 1 ) |  |
|  QuantIndices[ k ] = 0 |  |
|  QState = 0 |  |
|  for( k = last\_scan\_pos; k >= 0; k = k – 1 ) {  |  |
|  if( k  = =  last\_scan\_pos  &&  k > 0 ) |  |
|  QuantIndices[ k ] = 1 |  |
|  else { |  |
|  **abs\_trafo\_coeff\_gt0\_flag**[ k ] | ae(v) |
|  QuantIndices[ k ] = abs\_trafo\_coeff\_gt0\_flag[ k ] |  |
|  } |  |
|  if( QuantIndices[ k ] > 0 ) { |  |
|  **abs\_trafo\_coeff\_offset**[ k ] | ae(v) |
|  QuantIndices[ k ] += abs\_trafo\_coeff\_offset[ k ] |  |
|  } |  |
|  if( QuantIndices[ k ] = = NumTCoeffGtxFlags + 1 ) { |  |
|  **abs\_trafo\_coeff\_remainder**[ k ] | ae(v) |
|  QuantIndices[ k ] += abs\_trafo\_coeff\_remainder[ k ] |  |
|  } |  |
|  if( QuantIndices[ k ] > 0 ) { |  |
|  **trafo\_coeff\_sign\_flag**[ k ] | ae(v) |
|  if( trafo\_coeff\_sign\_flag[ k ] > 0 ) |  |
|  QuantIndices[ k ] = –QuantIndices[ k ]  |  |
|  } |  |
|  QState = QStateTransTab[ QState ][ QuantIndices[ k ] & 1 ] |  |
|  } |  |
| } |  |

### Annotation channel data syntax

|  |  |
| --- | --- |
| annotation\_channel\_data( ) { | Descriptor |
|  offset = AnnotationChannelNumSamples[ ac\_annotation\_channel\_id ] |  |
|  for( i = 0; i < 2 \* ( ac\_num\_annotation\_bytes\_div2\_minus1 + 1 ); i++ ) { |  |
|  **am\_annotaion\_byte** | u(8) |
|  AnnotationChannelBytes[ ac\_annotation\_channel\_id ][ offset + i ] = am\_annotation\_byte |  |
|  AnnotationChannelNumSamples[ ac\_annotation\_channel\_id ]++ |  |
|  } |  |
| } |  |

## Semantics

### General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in clause 7.4. 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 Specification.

### NAL unit semantics

#### General NAL unit semantics

NumBytesInNalUnit specifies the size of the NAL unit in bytes. This value is required for decoding of the NAL unit. Some form of demarcation of NAL unit boundaries is necessary to enable inference of NumBytesInNalUnit. One such demarcation method is specified in Annex B for the byte stream format. Other methods of demarcation could be specified outside of this Specification.

**rbsp\_byte**[ i ] is the i-th byte of an RBSP. An RBSP is specified as an ordered sequence of bytes as follows:

The RBSP contains a string of data bits (SODB) as follows:

– If the SODB is empty (i.e., zero bits in length), the RBSP is also empty.

– Otherwise, the RBSP contains the SODB as follows:

1) The first byte of the RBSP contains the first (most significant, left-most) eight bits of the SODB; the next byte of the RBSP contains the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain.

2) The rbsp\_trailing\_bits( ) syntax structure is present after the SODB as follows:

i) The first (most significant, left-most) bits of the final RBSP byte contain the remaining bits of the SODB (if any).

ii) The next bit consists of a single bit equal to 1 (i.e., rbsp\_stop\_one\_bit).

iii) When the rbsp\_stop\_one\_bit is not the last bit of a byte-aligned byte, one or more zero-valued bits (i.e., instances of rbsp\_alignment\_zero\_bit) are present to result in byte alignment.

3) One or more rbsp\_cabac\_zero\_word 16-bit syntax elements equal to 0x0000 could be present in some RBSPs after the rbsp\_trailing\_bits( ) at the end of the RBSP.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "\_rbsp" suffix. These structures are carried within NAL units as the content of the rbsp\_byte[ i ] data bytes. The association of the RBSP syntax structures to the NAL units is as specified in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

NOTE 1 – When the boundaries of the RBSP are known, the decoder could extract the SODB from the RBSP by concatenating the bits of the bytes of the RBSP and discarding the rbsp\_stop\_one\_bit, which is the last (least significant, right-most) bit equal to 1, and discarding any following (less significant, farther to the right) bits that follow it, which are equal to 0. The data necessary for the decoding process is contained in the SODB part of the RBSP.

**emulation\_prevention\_three\_byte** is a byte equal to 0x03. When an emulation\_prevention\_three\_byte is present in the NAL unit, it shall be discarded by the decoding process.

The last byte of the NAL unit shall not be equal to 0x00.

Within the NAL unit, the following three-byte sequences shall not occur at any byte-aligned position:

– 0x000000;

– 0x000001;

– 0x000002.

Within the NAL unit, any four-byte sequence that starts with 0x000003 other than the following sequences shall not occur at any byte-aligned position:

– 0x00000300;

– 0x00000301;

– 0x00000302;

– 0x00000303.

#### NAL unit header semantics

**nal\_unit\_type** specifies the NAL unit type, i.e., the type of RBSP data structure contained in the NAL unit as specified in Table 2.

NAL units that have nal\_unit\_type in the range of UNSPEC\_5..UNSPEC\_85, inclusive, and UNSPEC\_87..UNSPEC\_255, inclusive, for which semantics are not specified, shall not affect the decoding process specified in this Specification.

Table 2 - NAL unit types

| **nal\_unit\_type** | **Name of nal\_unit\_type** | **Content of NAL unit and RBSP syntax structure** |
| --- | --- | --- |
| 0 | FORBIDDEN\_NUT | Forbidden nal unit type for start code emulation prevention |
| 1 | WPS\_NUT | Waveform parameter setwaveform\_parameter\_set\_rbsp( ) |
| 2 | IF\_NUT | Independent frameindependent\_frame\_rbsp( ) |
| 3 | DF\_NUT | Dependent framedependent\_frame\_rbsp( ) |
| 4 | AC\_NUT | Annotation channelannotation\_channel\_rbsp( ) |
| 5..85 | UNSPEC\_5..UNSPEC\_85 | Unspecified NAL unit types |
| 86 | AM\_NUT | Auxiliary metadataauxiliary\_metadata\_rbsp( ) |
| 87..255 | UNSPEC\_87..UNSPEC\_255 | Unspecified NAL unit types |

#### Encapsulation of an SODB within an RBSP (informative)

This clause does not form an integral part of this Specification.

The form of encapsulation of an SODB within an RBSP and the use of the emulation\_prevention\_three\_byte for encapsulation of an RBSP within a NAL unit is described for the following purposes:

– To prevent the emulation of start codes within NAL units while allowing any arbitrary SODB to be represented within a NAL unit,

– To enable identification of the end of the SODB within the NAL unit by searching the RBSP for the rbsp\_stop\_one\_bit starting at the end of the RBSP,

– To enable a NAL unit to have a size greater than that of the SODB under some circumstances (using one or more rbsp\_cabac\_zero\_word syntax elements).

The encoder can produce a NAL unit from an RBSP by the following procedure:

1. The RBSP data are searched for byte-aligned bits of the following binary patterns:

 '00000000 00000000 000000xx' (where 'xx' represents any two-bit pattern: '00', '01', '10', or '11'),

and a byte equal to 0x03 is inserted to replace the bit pattern with the pattern:

 '00000000 00000000 00000011 000000xx',

and finally, when the last byte of the RBSP data is equal to 0x00 (which can only occur when the RBSP ends in a rbsp\_cabac\_zero\_word), a final byte equal to 0x03 is appended to the end of the data. The last zero byte of a byte‑aligned three-byte sequence 0x000000 in the RBSP (which is replaced by the four-byte sequence 0x00000300) is taken into account when searching the RBSP data for the next occurrence of byte-aligned bits with the binary patterns of the form '00000000 00000000 000000xx'.

1. The resulting sequence of bytes is then prefixed with the NAL unit header, within which the nal\_unit\_type indicates the type of RBSP data structure in the NAL unit.

This procedure results in the construction of the entire content of the NAL unit that follows the NAL unit header.

This process can allow any SODB to be represented in a NAL unit while ensuring both of the following:

– No byte-aligned start code prefix is emulated within the NAL unit.

* No sequence of 8 zero-valued bits followed by a start code prefix, regardless of byte-alignment, is emulated within the NAL unit.

### Raw byte sequence payloads, trailing bits and byte alignment semantics

#### Waveform parameter set RBSP semantics

A WPS RBSP shall be available to the decoding process prior to it being referenced by either of the following:

* an IF NAL with if\_waveform\_parameter\_set\_id equal to the value of wps\_waveform\_parameter\_set\_id in the WPS RBSP,
* a DF NAL with df\_waveform\_parameter\_set\_id equal to the value of wps\_waveform\_parameter\_set\_id in the WPS RBSP,
* a AC NAL with ac\_waveform\_parameter\_set\_id equal to the value of wps\_waveform\_parameter\_set\_id in the WPS RBSP.

All WPS NAL units with a particular value of wps\_waveform\_parameter\_set\_id in a CWS shall have the same content.

**wps\_waveform\_parameter\_set\_id** provides an identifier for the WPS for reference by other syntax elements.

**wps\_num\_channels\_in\_next\_group\_minus1** plus 1 specifies the number of channels in the next channel group in the sequence of channel groups.

**wps\_num\_channel\_group\_repetitions** specifies the number of channel groups that follow the previous channel group. Each of these channel groups has the same number of channels as the previous channel group.

**wps\_more\_channel\_groups\_present\_flag** equal to 1 specifies that more channel groups are specified on the WPS.

**wps\_channel\_reordering\_flag** equal to 1 specifies that syntax elements for reordering the channels in the decoded waveform sequence is present.

**wps\_num\_channel\_swaps\_minus1** plus 1 specifies the number of channel swaps to be carried out in order to perform channel reordering on the decoded waveform sequence.

**wps\_swap\_frst\_idx**[ i ] specifies the first channel of channel pair i to be swpped.

**wps\_swap\_scnd\_idx\_min\_frst\_idx\_min1**[ i ] plus 1 plus wps\_swap\_frst\_idx[ i ] specifies the second channel of channel pair i to be swapped.

**wps\_num\_annotation\_channels** specifies the number of annotation channels present in the bitstream.

#### Independent frame RBSP semantics

**if\_waveform\_parameter\_set\_id** specifies the value of wps\_waveform\_parameter\_set\_id for the WPS in use.

**if\_channel\_group\_id** identifies the channel group to which the current independent frame belongs. When if\_channel\_group\_id is not present, it is inferred to be equal to 0.

**if\_length\_signal\_mode\_flag** equal to 1 specifies that a syntax element if\_indep\_num\_samples\_per\_channel\_minus1 is present.

**if\_frame\_length\_shift** specifies an offset for deriving the variable Log2FrameLength as follows:

Log2FrameLength = Log2MaxBlockSize + if\_frame\_length\_shift (44)

**if\_max\_min\_block\_size** specifies an index for deriving variable Log2MaxBlockSize as follows:

Log2MaxBlockSize = LutBlockSizeMaxLog2[ if\_max\_min\_block\_size ] (45)

The value of if\_max\_min\_block\_size shall be in the range of 0 to 62, inclusive.

The array LutBlockSizeMaxLog2[ ] is specified as follows:

LutBlockSizeMaxLog2[ ] = (46)

{

 4, 5, 5, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 8, 9,

 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11,

 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13,

 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 14

}

The array LutBlockSizeMinLog2[ ] is specified as follows:

LutBlockSizeMinLog2[ ] = (47)

{

 4, 4, 5, 4, 5, 6, 4, 5, 6, 7, 4, 5, 6, 7, 8, 4,

 5, 6, 7, 8, 9, 4, 5, 6, 7, 8, 9, 10, 4, 5, 6, 7,

 8, 9, 10, 11, 4, 5, 6, 7, 8, 9, 10, 11, 12, 5, 6, 7,

 8, 9, 10, 11, 12, 13, 6, 7, 8, 9, 10, 11, 12, 13, 14

}

The variable MaxSplitDepth is derived as follows:

MaxSplitDepth = Log2MaxBlockSize - LutBlockSizeMinLog2[ if\_max\_min\_block\_size ] (48)

**if\_perceptual\_mode** specifies ... TBD

**if\_max\_min\_bit\_depth** specifies an index for deriving the variables BitDepthMax and BitDepthMin as follows:

BitDepthMax = LutBitDepthMax[ if\_max\_min\_bit\_depth ] (49)

BitDepthMin = LutBitDepthMin[ if\_max\_min\_bit\_depth ] (50)

The value of if\_max\_min\_bit\_depth shall be in the range of 0 to 62, inclusive.

The array LutBitDepthMax[ ] is specified as follows:

LutBitDepthMax[ ] = (51)

{

 3, 4, 4, 8, 8, 8, 8, 8, 8, 12, 12, 12, 12, 12, 12, 12,

 12, 12, 12, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20, 20,

 20, 20, 20, 20, 20, 20, 20, 20, 20, 24, 24, 24, 24, 24, 24, 24,

 24, 24, 24, 24, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28

}

The array LutBitDepthMin[ ] is specified as follows:

LutBitDepthMin[ ] = (52)

{

 2, 2, 3, 2, 3, 4, 5, 6, 7, 2, 3, 4, 5, 6, 7, 8,

 9, 10, 11, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 9, 10,

 11, 12, 13, 14, 15, 16, 17, 18, 19, 13, 14, 15, 16, 17, 18, 19,

 20, 21, 22, 23, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27

}

**if\_allow\_cross\_channel\_pred\_flag** equal to 1 specifies that the cross channel prediction mode is allowed.

**if\_cc\_pred\_filtering\_mode** specifies the allowed filtering options that may be applied to the cross channel prediction signal as follows:

* If if\_cc\_pred\_filtering\_mode is equal to 0, no filtering may be applied to the cross channel prediction signal.
* If if\_cc\_pred\_filtering\_mode is equal to 1, a half-pel filtering of the cross channel prediction signal is allowed.
* If if\_cc\_pred\_filtering\_mode is equal to 2, a half-pel filtering and a full-pel filtering of the cross channel prediction signal are allowed.

The value of if\_cc\_pred\_filtering\_mode shall lie in the range from 0 to 2 inclusively.

**if\_allow\_cc\_pred\_mult\_hyp\_flag** equal to 1 specifies that cross channel prediction with two input channels is allowed.

**if\_allow\_block\_matching\_pred\_flag** equal to 1 specifies that the block matching prediction mode is allowed.

**if\_bm\_pred\_filtering\_mode** specifies the allowed filtering options that may be applied to the block matching prediction signal as follows:

* If if\_bm\_pred\_filtering\_mode is equal to 0, no filtering may be applied to the block matching prediction.
* If if\_bm\_pred\_filtering\_mode is equal to 1, a half-pel filtering of the block matching prediction signal is allowed.
* If if\_bm\_pred\_filtering\_mode is equal to 2, a half-pel filtering and a full-pel filtering of the block matching prediction signal are allowed.

The value of if\_bm\_pred\_filtering\_mode shall lie in the range from 0 to 2 inclusively.

**if\_allow\_bm\_pred\_mult\_hyp\_flag** equal to 1 specifies that block matching prediction with two hypothesis is allowed.

**if\_allow\_bm\_offset\_pred\_prev\_ch\_flag** equal to 1 specifies that offsets for the block matching prediction can be predicted from offsets for the block matching prediction of the previous channel.

**if\_allow\_lpf** equal to 1 specifies that linear predictive filtering is allowed.

**if\_lpf\_allow\_prev\_ch\_flag** equal to 1 specifies that linear predictive filtering using input samples from up to 3 previous channels is allowed.

**if\_residual\_quant\_mode** specifies the quantization mode.

**if\_ch\_indep\_interval\_idx** specifies the variable DepChMask = ( 2 << if\_ch\_indep\_interval\_idx ) – 1. For the decoding process of clause 8, the channels can be grouped into consecutive groups of channels, each consisting of at most DepChMask +1 many channels, such that each group of channels can be processed independently from each other group of channels.

**if\_max\_abs\_delta\_qp\_idx** specifies the Variable MaxAbsDeltaQP = ( 1 << if\_max\_abs\_delta\_qp\_idx ) – 1.

**if\_indep\_num\_samples\_per\_channel\_minus1** plus 1 specifies the number of samples per channel present in the current frame sequence.

**if\_indep\_init\_block\_qp** specifies the quantization parameter.

**if\_ctx\_init\_flag** equal to 1 specifies that if\_ctx\_init\_mode is present.

**if\_ctx\_init\_mode** indicates which parameters are used for context initialization.

#### Dependent frame RBSP semantics

A DP RBSP unit shall only occur in the bitstream if at least one IF RBSP occurs before for which all of the following conditions are fulfilled:

* Syntax element if\_waveform\_parameter\_set\_id of the IF RBSP is equal to the value of df\_waveform\_parameter\_set\_id of the current DP RBSP,
* Syntax element if\_channel\_group\_id of the IF RBSP is equal to the value of df\_channel\_group\_id\_plus1 minus 1 of the current DP RBSP.

**df\_waveform\_parameter\_set\_id** specifies the value of wps\_waveform\_parameter\_set\_id for the WPS in use.

**df\_channel\_group\_id\_plus1** minus 1 identifies the channel group to which the current independent frame belongs.

#### Annotation channel RBSP semantics

**ac\_waveform\_parameter\_set\_id** specifies the value of wps\_waveform\_parameter\_set\_id for the WPS in use.

**ac\_annotation\_channel\_id** specifies the annotation channel index.

**ac\_num\_annotation\_bytes\_div2\_minus1** is used to determine the number of syntax elements am\_annotaion\_bytes present in the current AC RBSP as 2 \* ( ac\_num\_annotation\_bytes\_div2\_minus1 + 1).

#### Auxiliary metadata RBSP semantics

An AM RBSP unit shall only occur as the first NAL unit in the bitstream.

**am\_fourcc\_id\_last\_three\_bytes** must equal 0x415743.

**am\_header\_crc32** is the CRC calculated over the byte sequence starting with the byte containing the am\_reserved\_flag until the end of auxiliary\_metadata\_rbsp( ).

**am\_reserved\_flag** shall be ignored.

**am\_waveform\_type** specifies the waveform type according to Table 3.

Table 3 – Name association to am\_waveform\_type and type description

|  |  |  |
| --- | --- | --- |
| am\_waveform\_type | Name of am\_waveform\_type | Type of waveform |
| 0 | WT\_GENERIC | None, not signalled |
| 1 | WT\_EDF\_PLUS | EDF+, www.edfplus.info  |
| 2 | WT\_BS2088 | BW64, ITU-R BS.2088-1  |
| 3 | WT\_RESERVED | reserved, astro- or geophysical signal |

**am\_length\_signal\_mode** equal to 1 indicates that syntax element am\_stream\_num\_samples\_per\_ch is present in the bitstream.

**am\_allow\_reconfig\_flag**

**am\_copyright\_flag**

**am\_original\_flag**

**am\_private\_flag**

**am\_stream\_max\_sampling\_rate\_minus1** plus 1 specifies the maximum sampling rate present in the bitstream.

**am\_stream\_max\_num\_channels\_minus1** plus 1 specifies the maximum number of channels in the bitstream.

**am\_stream\_num\_samples\_per\_ch** specifies the number of samples per channel present in the bitstream.

**am\_metadata\_reserved\_flag** shall be ignored.

**am\_metadata\_num\_bytes\_minus1** plus 1 specifies the number of metadata payload bytes present in the AM RBSP.

**am\_metadata\_payload\_bytes**[ i ] specifies the i-th metadata payload byte. The array am\_metadata\_payload\_bytes is a bitstream according to ITU-R BS.2088-1

**am\_num\_channels\_edf** is used to determine the number of syntax elements am\_edf\_header\_payload\_bytes present in the current AM RBSP as 256 \* ( am\_num\_channels\_edf + 1).

**am\_edf\_header\_payload\_bytes**[ i ] specifies the i-th byte of an EDF header structure.

#### RBSP trailing bits semantics

**rbsp\_stop\_one\_bit** shall be equal to 1.

**rbsp\_alignment\_zero\_bit** shall be equal to 0.

#### Byte alignment semantics

**byte\_alignment\_bit\_equal\_to\_one** shall be equal to 1.

**byte\_alignment\_bit\_equal\_to\_zero** shall be equal to 0.

### Frame data semantics

#### General frame data semantics

**end\_of\_frame\_sequence\_flag** equal to 1 indicates that that the current frame sequence is terminated. When end\_of\_frame\_sequence\_flag is not present, it is inferred to be equal to 0.

**block\_split\_log2** specifies a parameter for calculating Log2BlockSize. When block\_split\_log2 is not present, it is inferred to be equal to 0.

**block\_matching\_or\_cross\_channel\_pred\_flag** equal to 1 indicates that the predicition is generated by invoking either the block-matching or the linear model prediction mode. When block\_matching\_or\_cross\_channel\_pred\_flag is not present, it is inferred to be 0.

**cross\_channel\_pred\_flag** equal to 1 indicates that the prediction is generated by invoking the cross channel prediction mode.

When cross\_channel\_pred\_flag is not present, it is inferred as follows:

– If if\_allow\_cross\_channel\_pred\_flag is equal to 1 and if block\_matching\_or\_cross\_channel\_pred\_flag is equal to 1, cross\_channel\_pred\_flag is inferred to be 1.

– Otherwise (if\_allow\_cross\_channel\_pred\_flag is equal to 0 or block\_matching\_or\_cross\_channel\_pred\_flag is equal to 0), the value of cross\_channel\_pred\_flag is inferred to be 0.

**block\_pred\_mode** specifies the block prediction mode according to Table 4. When block\_pred\_mode is not present, it is inferred to be equal to BPM\_OFF.

Table 4 – Name association to block\_pred\_mode and mode description

|  |  |  |
| --- | --- | --- |
| block\_pred\_mode | Name of block\_pred\_mode | Mode description |
| 0 | BPM\_DC | Average value prediction |
| 1 | BPM\_LF | Linear-function predictor |
| 2 | BPM\_OFF | No block prediction |

**block\_abs\_delta\_qp** specifies the absolute value of the current QP-value-difference associated with the current block of the current channel. When block\_abs\_delta\_qp is not present, it is inferred to be equal to 0.

**block\_delta\_qp\_sign\_flag** specifies the sign of the current QP-value-difference as follows:

– When block\_delta\_qp\_sign\_flag is equal to 0, blockDeltaQP has a positive sign.

– Otherwise (block\_delta\_qp\_sign\_flag), blockDeltaQP has a negative sign.

When block\_delta\_qp\_sign\_flag is not present, it is inferred to be 0.

The variable BlockDeltaQP is derived from the syntax elements block\_abs\_delta\_qp and block\_delta\_qp\_sign\_flag by setting BlockDeltaQP = block\_delta\_qp\_sign\_flag ? – block\_abs\_delta\_qp : block\_abs\_delta\_qp.

**block\_delta\_zlsb\_present\_flag** equal to 1 indicates that the current number CurrZeroLSB[ ch ] of zero least significant bits has to be increased or decreased by one.

**block\_delta\_zlsb\_sign\_flag** specifies the change of the value of CurrZeroLSB[ ch ] as follows:

* If block\_delta\_zlsb\_sign\_flag is equal to 1, the value of CurrZeroLSB[ ch ] is decreased by one.
* Otherwise ( block\_delta\_zlsb\_sign\_flag is not equal to 1 ) , the value of CurrZeroLSB[ ch ] is increased by one.

**transform\_skip\_flag** equal to 1 indicates that the identity transform is used on the current block.

**transform\_dst\_flag** equal to 1 indicates that the discrete sine transform is used on the current block.

The syntax elements transform\_skip\_flag and transform\_dst\_flag determine the blockwise transform TransformMode for the current block. The possible values of TransformMode are delineated in Table 5.

Table 5 – Name association to TransformMode and mode description

|  |  |
| --- | --- |
| Name of transform\_mode | Mode description |
| TM\_OFF | Identity transform |
| TM\_DCT | Discrete cosine transform |
| TM\_DST | Discrete sine transform |

**end\_of\_truncated\_frame\_sequence\_flag** equal to 1 indicates that the current block is the last block of the frame sequence. When end\_of\_truncated\_frame\_sequence\_flag is not present, it is inferred to be equal to 0.

**num\_samples\_per\_channel\_to\_discard** specifies the number of samples per channel to be discarded at the end of the current block. Discarding of samples is only done if end\_of\_truncated\_frame\_sequence\_flag equals 1. The length of this syntax element is Log2BlockSize bits. The value of num\_samples\_per\_channel\_to\_discard shall not be equal to 0.

**end\_of\_frame\_one\_bit** equal to 1 indicates the end of a frame. When end\_of\_frame\_one\_bit is present, it shall be equal to 1. When end\_of\_frame\_one\_bit is not present, it is inferred to be equal to 0.

#### Cross channel prediction data semantics

**cc\_pred\_offset\_only\_flag** equal to 1 indicates that the cross channel prediction mode is to be applied without a scaling factor.

**cc\_pred\_filter\_flag** equal to 1 indicates that the output of the cross channel prediction mode is to be filtered, where the set of filter coefficients is determined by the syntax element cc\_pred\_filter\_idx. When cc\_pred\_filter\_flag is not present, it is inferred to be 0.

**cc\_pred\_filter\_idx** specifies the index filterIdx to derive the array CCFiltCoeffs as specified in Table 6 for filtering the output of the cross channel prediction.

Table 6 – Name association to filterIdx

|  |  |
| --- | --- |
| filterIdx | filter coefficients |
| 0 | {-3, 0, 19, 32, 19, 0, -3, 0} |
| 1 | {-1, -4, 8, 29, 29, 8, -4, -1} |

When cc\_pred\_filter\_idx is not present, it is inferred to be 1:

**cc\_pred\_mult\_hyp\_flag** equal to 1 indicates that the cross channel prediction mode with two hypotheses is used. When cc\_pred\_mult\_hyp\_flag is not present, it is inferred to be 0.

**cc\_pred\_abs\_chd\_greater0\_flag**[ n ] equal to 1 indicates that the channel difference to the predicted input channel difference minus 1 for the n-th hypothesis of the cross channel prediction is not 0.

**cc\_pred\_abs\_chd\_minus1**[ n ] plus 1 specifies the absolute value of the channel difference to the predicted input channel difference minus 1 for the n-th hypothesis of the cross channel prediction.

**cc\_pred\_chd\_sign\_flag**[ n ] specifies the sign of the channel difference to the predicted input channel difference minus 1 for the n-th hypothesis of the cross channel prediction as follows:

– When cc\_pred\_chd\_sign\_flag[ n ] is equal to 0, the corresponding channel difference has a positive sign.

– When cc\_pred\_chd\_sign\_flag[ n ] is equal to 1, the corresponding channel difference has a negative sign.

When cc\_pred\_chd\_sign\_flag[ n ] is not present, it is inferred to be 0.

#### Block matching prediction data semantics

**bm\_pred\_mult\_hyp\_flag** equal to 1 indicates that the block matching prediction mode with two hypotheses is used. When bm\_pred\_mult\_hyp\_flag is not present, it is inferred to be 0.

**bm\_pred\_add\_offset\_flag** equal to 1 indicates that an offset, derived from previous reconstructed samples, is added to the block matching prediction.

**bm\_pred\_filter\_flag**[ n ]equal to 1 indicates that the reference samples used for the n-th hypothesis of the block matching prediction are to be filtered, where the set of filter coefficients is determined by the syntax element bm\_pred\_filter\_idx[ n ]. When bm\_pred\_filter\_flag[ n ] is not present, it is inferred to be 0.

**bm\_pred\_filter\_idx**[ n ]specifies the index filterIdx used to derive the array BMFiltCoeffs[ n ][ i ], with 0 <=i < 7, of filter coefficients according to Table 6 for filtering the reference samples of the n-th hypothesis of the block matching prediction. When bm\_pred\_filter\_idx[ n ] is not present, it is inferred to be 1:

**bm\_pred\_off\_pred\_prev\_ch\_flag**[ n ] equal to 1 indicates that the value of offset minus block size for the n-th block matching prediction hypothesis is predicted from the value of offset minus block size of the n-th hypothesis of the previous channel. When bm\_pred\_off\_pred\_prev\_ch\_flag[n] is not present, it is inferred to be 0.

**bm\_pred\_abs\_offd\_greater0\_flag**[ n ] equal to 1 indicates that the offset difference to the predicted value of offset minus block size for the n-th hypothesis of the block matching prediction is not 0.

**bm\_pred\_abs\_offd\_minus1**[ n ] plus 1 specifies the absolute value of the offset difference to the predicted value of offset minus blocksize for the n-th hypothesis of the block matching prediction.

**bm\_pred\_offd\_sign\_flag**[ n ]specifies the sign of the offset difference to the predicted value of offset minus blocksize for the n-th hypothesis of the block matching prediction as follows:

– When bm\_pred\_offd\_sign\_flag[ n ] is equal to 0, the corresponding offset difference has a positive sign.

– Otherwise (bm\_pred\_offd\_sign\_flag[ n ]is not equal to 0), the corresponding offset difference has a negative sign.

When bm\_pred\_offd\_sign\_flag[ n ] is not present, it is inferred to be 0.

#### Sample pred mode data semantics

The sample pred mode data syntax specifes the mode SamplePredMode for the sample wise prediction on a block. The supported options for SamplePredMode and their name and meaning are delinated inTable 7.

Table 7 – Name association to SamplePredMode and mode description

|  |  |  |
| --- | --- | --- |
| SamplePredMode | Name of sample\_pred\_mode | Mode description |
| 0 | SPM\_SLOPE | Full-slope delta coding |
| 1 | SPM\_HALF\_SLOPE | Half-slope delta coding |
| 2 | SPM\_OFF | No sample prediction |
| 3 | SPM\_LPC | Linear predictive filtering |
| 4 | SPM\_DIFFS | Sample-wise delta coding |

**spred\_lpf\_or\_diff\_flag** equal to 1 specifies that either linear predictive filtering or sample-wise delta coding is to be used as a sample wise prediction mode.

**spred\_lpf\_flag** equal to 1 specifies that linear predictive filtering is to be used as a sample wise prediction mode. When spred\_lpf\_flag is not present, it is inferred to be 0.

**spred\_rem\_mode\_idx** specifies the sample wise prediction mode to be used among the first three options in Table 7.

The variable SamplePredMode is derived as follows:

– When spred\_lpf\_or\_diff\_flag is equal to 1, the following applies:
– If spred\_lpf\_flag is equal to 0, SamplePredMode is set to 4.
– Otherwise (spred\_lpf\_flag is equal to 1), SamplePredMode is set to 3.

– Otherwise (spred\_lpf\_or\_diff\_flag is not equal to 0), SamplePredMode is set to spred\_rem\_mode\_idx.

#### Linear predictive filtering data semantics

The array LPFWeightsCurr[  ] represents the filter coefficients used for the linear linear predictive filtering on the current block.

**lpf\_prev\_ch\_flag** equal to 1 specifies that the samples from up to 3 previous channels contribute to the input of the linear predictive filtering. When lpf\_prev\_ch\_flag is not present, it is inferred to be 0.

**lpf\_delta\_coding\_flag** equal to 1 specifies that the current filter coefficients for the linear predictive filtering process are to be reconstructed predictively by using the filter coefficients LPFWeightsNoPrevChPred[  ] as a prediction input. When lpf\_delta\_coding\_flag is not present, it is inferred to be 0.

**lpf\_num\_weights\_idx** determines the number of filter coefficients for the linear predictive filtering process.

**abs\_lpf\_weight\_greater0\_flag**[ n ] equal to 1 specifies that the current coded filter coefficient value is not 0.

**abs\_lpf\_weight\_minus1**[ n ] plus 1 specifies the absolute value of the current coded filter coefficient value.

**lpf\_weight\_sign\_flag**[ n ] specifies the sign of the current coded filter coefficient value as follows:

– When lpf\_weight\_sign\_flag[ n ] is equal to 0, the current coded filter coefficient value has a positive sign.

– When lpf\_weight\_sign\_flag[ n ] is equal to 1, the current coded filter coefficient value has a negative sign.

When lpf\_weight\_sign\_flag[ n ] is not present, it is inferred to be 0.

#### Quant res sample data semantics

The array QuantIndices[ k ] represents an array of quantization indices for the current block. The array index k specifies the location of the quantization index within the current block.

The variables NumTSkipGtxFlags, MaxTSkipRemPrefix, Log2SbbSize, and NumTCoeffGtxFlags are specified as follows:

NumTSkipGtxFlags = 4 (53)

MaxTSkipRemPrefix = 32 (54)

Log2SbbSize = 1 (55)

NumTCoeffGtxFlags = 20 (56)

The array QStateTransTable is specified as follows:

– If if\_residual\_quant\_mode is equal to 0, QStateTransTable is given by:

QStateTransTable[ ][ ] = { { 0, 0 } } (57)

– Otherwise, if if\_residual\_quant\_mode is equal to 1, QStateTransTable is given by:

QStateTransTable[ ][ ] = { { 0, 1 }, { 2, 3 }, { 1, 0 }, { 3, 2 } } (58)

– Otherwise (if\_residual\_quant\_mode is equal to 2), QStateTransTable is given by:

QStateTransTable[ ][ ] = { { 0, 2 }, { 5, 7 }, { 1, 3 }, { 6, 4 }, { 2, 0 }, { 4, 6 }, { 3, 1 }, { 7, 5 } } (59)

**coeff\_bypass\_value**[ k ] represents, when the current block is coded in bypass mode, the quantization index at location k within the current block as an unsigned integer. The value of coeff\_bypass\_value[ k ] shall be in the range of 0 to (1 << IntBitDepth) – 1, inclusive.

**abs\_tskip\_coeff\_gt0\_flag**[ k ] specifies, when the current block is coded with a sample-wise prediction mode, whether the quantization index at location k within the current block is non-zero as follows:

– If abs\_tskip\_coeff\_gt0\_flag[ k ] is equal to 0, the quantization index at location k is set equal to 0.

– Otherwise (abs\_tskip\_coeff\_gt0\_flag[ k ] is equal to 1), the quantization index at location k has non-zero value.

**abs\_tskip\_coeff\_offset**[ k ] specifies, when the current block is coded with a sample-wise prediction mode, an offset, coded using truncated unary binarization, for the absolute value of the quantization index at location k. The value of abs\_tskip\_coeff\_offset[ k ] shall be in the range of 0 to NumTSkipGtxFlags, inclusive. When abs\_tskip\_coeff\_offset[ k ] is not present, it shall be inferred to be equal to 0.

**abs\_tskip\_coeff\_rem\_prefix**[ k ] specifies, when the current block is coded with a sample-wise prediction mode, a prefix for the remainder of the absolute quantization index at location k within the current block. The value of abs\_tskip\_coeff\_rem\_prefix[ k ] shall be in the range of 0 to MaxTSkipRemPrefix, inclusive. When abs\_tskip\_coeff\_rem\_prefix[ k ] is not present, it shall be inferred to be equal to 0.

**abs\_tskip\_coeff\_rem\_fl\_suffix**[ k ] specifies, when the current block is coded with a sample-wise prediction mode, a suffix, coded using a fixed-length binarization, for the remainder of the absolute quantization index at location k within the current block. The value of abs\_tskip\_coeff\_rem\_fl\_suffix[ k ] shall be in the range of 0 to (1 << RiceParameter) – 1, inclusive. When abs\_tskip\_coeff\_rem\_fl\_suffix[ k ] is not present, it shall be inferred to be equal to 0.

**abs\_tskip\_coeff\_rem\_eg0\_suffix**[ k ] specifies, when the current block is coded with a sample-wise prediction mode, a suffix, coded using an exponential Golomb code of order zero, for the remainder of the absolute quantization index at location k within the current block. When abs\_tskip\_coeff\_rem\_eg0\_suffix[ k ] is not present, it shall be inferred to be equal to 0.

**tskip\_coeff\_sign\_flag**[ k ] specifies, when the current block is coded with a sample-wise prediction mode, the sign of the quantization index at location k as follows:

– If tskip\_coeff\_sign\_flag[ k ] is equal to 0, the quantization index at location k a non-negative value.

– Otherwise (tskip\_coeff\_sign\_flag[ k ] is equal to 1), the quantization index at location k has a negative value.

When tskip\_coeff\_sign\_flag[ k ] is not present, it shall be inferred to be equal to 0.

**last\_sbb\_index\_gt0\_flag** specifies, when the current block is coded with a block-wise transform, whether the last non-zero quantization index within the current block is located in a subblock with subblock index greater than 0:

– If abs\_tskip\_coeff\_gt0\_flag is equal to 0, all subblocks with subblock index greater than 0 contain only non-zero quantization indexes.

– Otherwise (abs\_tskip\_coeff\_gt0\_flag is equal to 1), one or more subblocks with subblock index greater than 0 contain non-zero quantization index.

**last\_sbb\_index\_rem** specifies, when the current block is coded with a block-wise transform, the remainder of the subblock index of the subblock that contains the last non-zero quantization index within the current block. The value of last\_sbb\_index\_rem shall be in the range of 0 to (NumQuantIndices >> Log2SbbSize) – 2, inclusive. When last\_sbb\_index\_rem is not present, it shall be inferred to be equal to 0.

**last\_index\_offset** specifies, when the current block is coded with a block-wise transform, the location of the last non-zero quantization index within the subblock with index last\_sbb\_index\_gt0\_flag + last\_sbb\_index\_rem as follows:

– If last\_sbb\_index\_gt0\_flag is greater than 0 or last\_index\_offset is greater than 0, the location of the last non-zero quantization index is given by ( (last\_sbb\_index\_gt0\_flag + last\_sbb\_index\_rem) << Log2SbbSize) + last\_index\_offset.

– Otherwise (last\_sbb\_index\_gt0\_flag is equal to 0 and last\_index\_offset is equal to 0), the current block does either contain only zero quantization indexes or only the quantization index at location k = 0 is non-zero. Whether or not the quantization index at location k = 0 is non-zero is specified by the value of abs\_trafo\_coeff\_gt0\_flag[ 0 ].

The value of last\_index\_offset shall be in the range of 0 to (1 << Log2SbbSize) – 1, inclusive. When last\_index\_offset is not present, it shall be inferred to be equal to 0.

**abs\_trafo\_coeff\_gt0\_flag**[ k ] specifies, when the current block is coded with a block-wise transform, whether the quantization index at location k within the current block is non-zero as follows:

– If abs\_trafo\_coeff\_gt0\_flag[ k ] is equal to 0, the quantization index at location k is set equal to 0.

– Otherwise (abs\_trafo\_coeff\_gt0\_flag[ k ] is equal to 1), the quantization index at location k has non-zero value.

When abs\_trafo\_coeff\_gt0\_flag[ k ] is not present, it is inferred as follows:

– If k is equal to ( (last\_sbb\_index\_gt0\_flag + last\_sbb\_index\_rem)  <<  Log2SbbSize ) + last\_index\_offset and k is greater than zero, the value of abs\_trafo\_coeff\_gt0\_flag[ k ] is inferred to be equal to 1.

– Otherwise (k is not equal to ( (last\_sbb\_index\_gt0\_flag + last\_sbb\_index\_rem)  <<  Log2SbbSize ) + last\_index\_offset or k is equal to 0), the value of abs\_trafo\_coeff\_gt0\_flag[ k ] is inferred to be equal to 0.

**abs\_trafo\_coeff\_offset**[ k ] specifies, when the current block is coded with a block-wise transform, an offset, coded using truncated unary binarization, for the absolute value of the quantization index at location k. The value of abs\_trafo\_coeff\_offset[ k ] shall be in the range of 0 to NumTCoeffGtxFlags, inclusive. When abs\_trafo\_coeff\_offset[ k ] is not present, it shall be inferred to be equal to 0.

**abs\_trafo\_coeff\_remainder**[ k ] specifies, when the current block is coded with a block-wise transform, the remainder of the absolute quantization index at location k within the current block. When abs\_trafo\_coeff\_remainder[ k ] is not present, it shall be inferred to be equal to 0.

**trafo\_coeff\_sign\_flag**[ k ] specifies, when the current block is coded with a block-wise transform, the sign of the quantization index at location k as follows:

– If trafo\_coeff\_sign\_flag[ k ] is equal to 0, the quantization index at location k a non-negative value.

– Otherwise (trafo\_coeff\_sign\_flag[ k ] is equal to 1), the quantization index at location k has a negative value.

When trafo\_coeff\_sign\_flag[ k ] is not present, it shall be inferred to be equal to 0.

### Annotation channel data semantics

**am\_annotaion\_byte** specifie an annotation channel byte.

# Decoding process

## General decoding process

### General

Input to this process is a bitstream BitstreamToDecode. Output of this process are a list of decoded channels, each having an associated waveform parameter set identifyer and an associated channel index.

The decoding process is specified such that all decoders that conform to the specification will produce numerically identical decoded output channels when invoking the decoding process. Any decoding process that produces identical decoded channels conforms to the decoding process requirements of this Specification.

### Channel output index derivation process

Input of this process are:

* The syntax elements of a waveform parameter set according to clause 7.3.2.1,
* a channel group index chGrpIdx
* a channel index chIdxInChGroup in a channel group.

Output of this process is an output channel index outChIdx.

This index is derived as follows:

The variable chIdxBRd which specifies the channel index before reordering ( if applicable ) is set to ChannelGroupStartingPos[chGrpIdx] + chIdxInChGroup.

If wps\_channel\_reordering\_flag is equal to zero, chIdxBdR is set equal to outChIdx.

Otherwise ( wps\_channel\_reordering\_flag is not equal to zero ), the following process is invoked:

 Set outChIdx = chIdxBRd.

 Set i = 0

 do

 if ( outChIdx == wps\_swap\_frst\_idx[ i ])

 outChIdx += wps\_swap\_scnd\_idx\_min\_frst\_idx\_min1[ i ] + 1

 else if (outputChIx == wps\_swap\_frst\_idx[ i ] + wps\_swap\_scnd\_idx\_min\_frst\_idx\_min1[ i ] + 1 )

 outChIdx = wps\_swap\_frst\_idx[ i ]

 i = i + 1

 while ( i <= wps\_num\_channel\_swaps\_minus1)

### Decoding process for independent frames (NAL units of type IF\_NUT)

The syntax elements of the last previously decoded waveform parameter set in the bitstream in decoding order that has the same waveform parameter set identifier as the current independent frame (waveform\_parameter\_set\_id == if\_waveform\_parameter\_set\_id) are inferred. It is a requirement of bitstream conformance that such a waveform parameter set exists. The set of syntax elements of this waveform parameter set shall be referred to as wpsCurr.

The variable wPId is set to if\_waveform\_parameter\_set\_id and the variable chGId is set to if\_channel\_group\_id.

The frame decoding process of clause 8.2 is invoked with the variable frameStartPos set to 0 and the variable frameNumChannels set to NumChannels[ if\_channel\_group\_id ].

The output of this process is assigned to the number of reconstructed samples numRecSmpls and the array of reconstructed sample values Rec[ wPId ][ chGId ][ c ][ i ] with 0 <= c < NumChannels[ if\_channel\_group\_id ] and 0 <= i < numRecSmpls.

The variable NumSpls[ wPId ][ chGId ] which specifies the number of decoded samples corresponding to the current waveform parameter set id and to the current channel group id is set to numRecSmpls.

For each c with 0 <= c < NumChannels[ if\_channel\_group\_id ], the following applies:

* The decoded output channel outCh[ c ] is specified as the channel of reconstructed sample values Rec[ wPId ][ chGId ][ c ][ i ] with 0 <= i < numRecSmpls.
* The waveform parameter set identifier associated to outCh[ c ] is set to wpId.
* The process of clause 8.1.2 is invoked with the set of waveform parameter set syntax elements set to wpsCurr, the variable chGrpIdx set to if\_channel\_group\_id and the variable chIdxInChGroup set to c. The output of this process is assigned to the output channel index outputIdx associated to outCh[ c ].

### Decoding process for dependent frames (NAL units of type DF\_NUT)

The syntax elements of the independent frame header of the last previously decoded independent frame in the bitstream in decoding order that has the same waveform parameterset id and the same channel group id as the current dependent frame ( if\_waveform\_parameter\_set\_id == df\_waveform\_parameter\_set\_id and if\_channel\_group\_id == df\_channel\_group\_id ) are inferred. It is a requirement of bitstream conformance that such an independent frame exists. The set of syntax elements of the last previously decoded waveform parameter set that has the same waveform parameter set identifier as the current dependent frame (waveform\_parameter\_set\_id == df\_waveform\_parameter\_set\_id) shall be referred to as wpsCurr.

The variable wPId is set to df\_waveform\_parameter\_set\_id and the variable chGId is set to df\_channel\_group\_id.

The frame decoding process of clause 8.2 is invoked with the variable frameStartPos set to NumSpls[ wPId ][ chGId ], the variable frameNumChannels set to NumChannels[ if\_channel\_group\_id ] and the input array sample values recSamples[ c ][ i ] set to Rec[ wPId ][ chGId ][ c ][ i ] for 0 <= c < NumChannels[ df\_channel\_group\_id ] and 0 < =i < NumSpls[ wPId ][ chGId ].

The output of this process is assigned to the number of currently reconstructed samples numRecSmpls and the reconstructed sample values rec [ c ][ i ] with 0 <= c < NumChannels[ df\_channel\_group\_id ] and numSpls[ wPId ][ chGId ] < = i < numSpls[ wPId ][ chGId ] + numRecSmpls.

The values rec[ c ][ i ] are assigned to the variables Rec[ wPId ][ chGId ][ c ][ i ] for 0 <= c < NumChannels[ df\_channel\_group\_id ] and numSpls[ wPId ][ chGId ] < = i < numSpls[ wPId ][ chGId ] + numRecSmpls.

The variable NumSpls[ wPId ][ chGId ] is set to NumSpls[ wPId ][ chGId ] + numRecSmpls.

For each c with 0 <= c < NumChannels[ df\_channel\_group\_id ], the following applies:

* The decoded output channel outCh[ c ] is specified as the channel of reconstructed sample values Rec[ wPId ][ chGId ][ c ][ i ] with numSpls[ wPId ][ chGId ]< = i < numSpls[ wPId ][ chGId ] + numRecSmpls.
* The waveform parameter set identifier associated to outCh[ c ] is set to wpId.
* The process of clause 8.1.2 is invoked with the set of waveform parameter set syntax elements set to wpsCurr, the variable chGrpIdx set to df\_channel\_group\_id and the variable chIdxInChGroup set to c to specify the output channel index outputIdx associated to outCh[ c ].

## Frame data decoding process

Input to this process are:

* a starting position startPos.
* a number of channels numCh
* if starPos is not equal to zero: An array of reconstructed sample values rec[ c ][ i ] with 0 <= c < numCh and
0 <= i < startPos.

Output of this process are the number of reconstructed samples numRecSmpls and the newly reconstructed sample values rec[ c ][ i ] with 0 <= c <numCh and startPos <= i < startPos + numRecSmpls.

The ordered steps in the the decoding process of this clause follow the corresponding orderd steps in the syntax of clause 7.3.3.1, where at each step, the associated syntax elements are inferred from clause 7.3.3.

The variable currLog2TSize is set to 4 and the variable currTSize is set to ( 1 << currLog2Tsize ).

The variable maxVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1 and the variable minVall is set to – maxVal- 1.

The variable currBlockPos is initialized to 0.

For each step in the enclosing while loop of clause 7.3.3.1, the following ordered steps apply:

* The variable blockSize is set equal to ( 1 << Log2BlockSize ).
* For each channel with channel index currCh, 0 <= currCh <numCh, the following ordered steps apply:
	1. The blockwise prediction decoding process of clause 8.3 is applied. The output of this process is assigned to the prediction sample values predCurr[ i ] with 0 <= i < blockSize and the extended left adjacent residual samples resiLeftCurr[ j ] with 0 <= j < currTSize.
	2. The scaling and inverse blockwise transform process from clause 8.4 is applied. The output of this process is assigned to the intermediate reconstructed residual sample values resImdCurr[ i ] with 0 <= i < blockSize.
	3. The sample wise prediction decoding process of clause 8.5 is applied. The output of this process is assigned to the final residual sample values resCurr[ i ] with 0 <= i < blockSize.
	4. For 0 <= i < blockSize, the value Clip3( minVal, maxVal, predCurr[ i ] + resCurr[ i ]) is assigned to the reconstructed sample values rec[ currCh ][ i ]
* The variable currBlockPos is incremented as follows:
	+ If end\_of\_truncated\_frame\_sequence\_flag is equal to 1, the variable currBlockPos is set equal to currBlockPos + blockSize – num\_samples\_per\_channel\_to\_discard
	+ Otherwise ( end\_of\_truncated\_frame\_sequence\_flag is not equal to 1 ), the variable currBlockPos is set equal to currBlockPos + blockSize.

The variable currBlockPos is set equal to numRecSmpls.

## Blockwise prediction decoding process

Input to this process are:

* the current channel index currCh,
* the current block position currBlockPos,
* the array of reconstructed samples of previous channels rec[ c ][ i ] with
max( currCh – ( DepChMask & currCh ), 0 ) <= c < currCh and with 0 <= i < currBlockPos + ( 1 << Log2BlockSize ).
* the array of reconstructed samples of the current channel rec[ currCh ][ i ] with 0 < = i < currBlockPos.
* the parameter currLog2TSize which determines the size of the adjacent left residual samples to be computed and the size of the template for the parameter computation of the cross-channel and the block-matching prediction.

Output to this process are the prediction sample values pred [ i ] with 0 <= i < ( 1 << Log2BlockSize ) and the extended residual sample values resiLeft[ j ] with 0 <= j < (1 << currLog2TSize ). These values are derived as follows:

* If block\_matching\_or\_cross\_channel\_pred\_flag is equal to 1, the following applies:
	+ If cross\_channel\_pred\_flag is equal to 1, the cross channel prediction decoding process of clause 8.3.6 is invoked with chIdx set equal to currCh, blockPos set equal to currBlockPos, log2BlockSize set equal to Log2BlockSize, the reference sample values of previous channels ref[ c ][ i ] set equal to rec[ c ][ i ] for max( currCh – ( DepChMask & currCh ), 0 ) <= c < currCh and 0 <= i < currBlockPos + ( 1 << Log2BlockSize), the reference sample values of the current channel refCurr[ i ] set equal to rec[ currCh ][ i ] with 0 < = i < currBlockPos and log2TSize set equal to currLog2TSize.
	+ Otherwise ( cross\_channel\_pred\_flag is not equal to 1 ), the block matching prediction decoding process of clause 8.3.7 is invoked with chIdx set equal to currCh, blockPos set equal to currBlockPos, log2BlockSize set equal to Log2BlockSize, the reference sample values ref[ i ] set equal to rec[ currCh ][ i ] with 0 < = i < currBlockPos and log2TSize set equal to currLog2TSize.
* Otherwise (block\_matching\_or\_cross\_channel\_pred flag is not equal to 1), the following applies:
	+ If block\_pred\_mode is equal to BPM\_DC, the DC prediction decoding process of clause 8.3.4 is invoked with blockPos set equal to currBlockPos, log2BlockSize set equal to Log2BlockSize, the reference sample values ref[ i ] set equal to rec[ currCh ][ i ] with 0 < = i < currBlockPos and log2TSize set equal to currLog2TSize.
	+ If block\_pred\_mode is equal to BPM\_LF, the line fitting decoding process of clause 8.3.5 is invoked with blockPos set equal to currBlockPos, log2BlockSize set equal to Log2BlockSize, the reference sample values ref[ i ] set equal to rec[ currCh ][ i ] with 0 < = i < currBlockPos and log2TSize set equal to currLog2TSize.
	+ If block\_pred\_mode is equal to BPM\_OFF, the zero prediction decoding process of clause 8.3.3 is invoked with log2BlockSize set equal to Log2BlockSize and log2TSize set equal to currLog2TSize.

### Linear extrapolation process of an array to the right

Input to this process are:

* an input array size szArr >= 4,
* an input array of sample values p[ i ] with 0 <= i < szArr,
* an extrapolation size log2SzExt > 0

Output to this process are the array values extrapolated to the right p[ szArr + j ] with 0 <= j < (1 << log2SzExt) and an extrapolated mean value meanValExtr.

The variable szExt is set to 1 << log2SzExt.

The variable slope is set to 15\*( p[ szArr – 1 ] – p[ szArr –4 ] ) +5\*( p[ szArr – 2 ] – p[ szArr – 3 ] ).

The variable offset is set to ( ( p[ szArr – 4 ] + p[ szArr – 3 ] + p[ szArr – 2] + p[ szArr – 1 ] +2 ) << 7 ).

For 0 <= j < szExt, the extrapolated array values p[ szArr + j ] are defined as follows:

* One sets stepCurr = 25 + j\*10.
* One sets slopeCurr = slope \* ( 2\*szExt – j ).
* One sets p[ szArr + j ] = ( ( offset<<( log2SzExt +1 ) )+ slopeCurr\*stepCurr ) >> (log2SzExt + 10).

One sets meanValExtr = (offset + 20 \* slope ) >> 9.

### Linear extrapolation process of an array to the left

Input to this process are:

* an input array starting position startPos
* an input array size szArr with szArr – startPos >= 4,
* an input array of sample values p[ i ] with startPos <= i < szArr,
* an extrapolation size szExt > 0

Output to this process are the array values extrapolated to the left p[ startPos – 1– j ] with 0 < = j < szExt.

The mirrored input array sample values pMirror[ i ] with 0 <= i < szArr – startPos are defined as pMirror[ i ] = p[ szArr – 1– i ].

The extrapolation process to the right from clause 8.3.1 is invoked with the input array size szArr – startPos, the input array pMirror and the extensions size set to Ceil( Log2 ( szExt ) ) as input to obtain the array values pMirror[ szArr + j ] with 0 <= j < ( 1 << Ceil( Log2 ( szExt ) ) ).

For 0 <= j < szExt the value pMirror[ szArr + j ] is assigned to p[ startPos – 1 – j ].

### Zero prediction decoding process

Input to this process are:

* a variable log2BlockSize that determines the size of the current block,
* the parameter log2TSize which determines the size of the adjacent left residual samples to be computed.

This process specifies as an output the arrays pred[ i ] = 0 with 0 <= i < (1 <<log2BlockSize) and the array of adjacent left residual samples resiLeft [ j ] = 0 with 0 <= j < ( 1 << log2TSize ).

### DC prediction decoding process

Input to this process are:

* a variable blockPos specifying the position of the first sample of the current block,
* a variable log2BlockSize that determinesthe size of the current block,
* an array of reconstructed samples ref [ i ] with 0 < = i < blockPos.
* a parameter log2TSize which determines the size of the adjacent left residual samples to be computed.

Output of this process are the array of DC prediction sample values pred[ i ] with 0 <= i < (1 <<log2BlockSize) and the array of adjacent left residual samples resiLeft [ j ] with 0 <= j < ( 1 << log2TSize ).

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The variable blockSize is set to 1 << log2BlockSize.

* If blockPos < 4, the values pred[ i ] with 0 <= i < blockSize and the values resiLeft[ j ] with 0 <=. j < tSize are set to 0.
* Otherwise (blockPos > 4 ), the following applies:
	+ The variable mean is set as
	mean = Clip3(minPredVal, maxPredVal, $\left(\sum\_{k=0}^{3}ref\left[ blockPos-4+k \right]+2)\gg 2\right)$.
	+ One sets pred[ i ] = mean for all i with 0 <= i < blockSize.
	+ If blockPos >= tSize, resiLeft[ j ] is set to ( ref[ – blockPos + j ] – mean ) for all j with 0 <=. j < tSize.
	+ Otherwise ( blockPos < tSize ), resiLeft[ j ] is set to 0 for all j with 0 <= j < tSize.

### Line fitting prediction decoding process

Input to this process are:

* a variable blockPos specifying the position of the first sample of the current block,
* a variable log2BlockSize that determinesthe size of the current block,
* the array of reconstructed samples of the current channel ref[ i ] with 0 < = i < blockPos.
* the parameter log2TSize which determines the size of the adjacent left residual samples to be computed.

Output of this process are the array of line fitting prediction sample values pred[ i ] with 0 <= i < (1 << log2BlockSize) and the array of adjacent left residual samples resiLeft [ j ] with 0 <= j < ( 1 << log2TSize ).

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The variable blockSize is set to 1 << log2BlockSize.

The variable tSize is set to 1 << log2TSize.

* If blockPos < 4, the values pred[ i ] with 0 <= i < blockSize and the values resiLeft[ j ] with 0 <=. j < tSize are set to 0.
* Otherwise (blockPos > 4 ), the following applies:
	+ The array q[ j ] with 0 <=j <4 is defined as q[ j ] = ref[ blockPos – 4 + j ].
	+ The linear extrapolation process to the right from clause 8.3.1 is invoked with input the input array size 4, the input array q and the extension size log2BlockSize to obtain the array values q[ 4 + i ] with 0 <= i < blockSize and the extrapolated mean value meanValExtr.
	+ One sets pred[ i ] = Clip3( minPredVal, maxPredVal, q[ 4+i ] ) for all 0 <=i <blockSize.
	+ If blockPos >= tSize, resiLeft[ j ] is set to ( ref[ – blockPos + j ] – meanValExtr ) for all j with 0 <=. j < tSize.
	+ Otherwise ( blockPos < tSize ), resiLeft[ j ] is set to 0 for all j with 0 < = j < tSize.

### Cross channel prediction decoding process

Input to this process are:

* a variable chIdx specifying the current channel,
* a variable blockPos specifying the position of the first sample of the current block,
* a variable log2BlockSize that determinesthe size of the current block,
* the array of reference samples of previous channels ref[ c ][ i ] with
max( chIdx – ( DepChMask & chIdx ), 0 ) <= c < chIdx and 0 <= i < blockPos + (1 << log2BlockSize).
* the array of reference samples of the current channel refCurr[ i ] with 0 < = i < blockPos.
* the parameter log2TSize which determines the size of the adjacent left residual samples to be computed and which determines tze size of the template used for parameter derivation.

Output of this process are the array of cross channel prediction sample values pred[ i ] with 0 <= i < (1 << log2BlockSize) and the array of adjacent left residual samples resiLeft [ j ] with 0 <= j < ( 1 << log2TSize ).

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The variable blockSize is set to 1 << log2BlockSize.

The variable firstPrevCh which indexes the first input channel is set to
chIdx – 1 – CrossChannelPredInputChDistMinus1[ chIdx ][ 0 ].

If cc\_pred\_mult\_hyp\_flag is not equal to zero, the variable secondPrevCh, which indexes the second input channel, is set to chIdx – 1– CrossChannelPredInputChDistMinus1[ chIdx ][ 1 ].

The variable tSize is set to 1 << log2TSize.

The variable fPdL which specifies the length to the left for the prediction filters of clause 7.3.3.2 is set to 3.

The variable log2FPdR which which determines the length to the right for the prediction filters of clause 7.3.3.2 is set to 2.

The variable fPdR is set to 1 << log2FPdR.

The variable fSz which specifies the filter size for the filters of clause 7.3.3.2 is set to fPdL + fPdR.

* If blockPos < tSize, the following applies:
	+ The intermediate cross channel prediction samples before filtering and extrapolation p[ i ] with 0 <= i < blockSize are defined as follows:
		- If cc\_pred\_mult\_hyp\_flag is equal to zero, one pust p[ i ] = ref[ firstPrevCh ][ blockPos + i ].
		- Otherwise ( cc\_pred\_mult\_hyp\_flag is not equal to zero), one set
		p[ i ] = ( ref[ firstPrevCh ][ blockPos + i ] + ref[ secondPrevCh ][ blockPos + i ] + 1 ) >> 1.
	+ If cc\_pred\_filter\_flag is equal to zero, the final cross channel prediction sample values pred[ i ] are set to Clip3(minPredVal, maxPredVal, p[ i ]) for 0 <= i < blockSize.
	+ Otherwise ( cc\_pred\_filter\_flag is not equal to zero ) , the following applies:
		- If blockPos < fPdL, the extrapolation process to the left from clause 8.3.2 is applied with the input starting position 0, the input array p, the input array size blockSize and the extension size fPdL to obtain the sample values p[ –fPdL + i ] with 0 <= i < fPdL.
		- Otherwise ( blockPos >= fPdL) the following applies:
			* If cc\_pred\_mult\_hyp\_flag is equal to zero, for 0 <= i < fPdL the value ref[ firstPrevCh ][ blockPos –fPdL + i ] is assigned to p[ –fPdL + i ].
			* Otherwise ( cc\_pred\_mult\_hyp\_flag is not equal to zero ) , for 0 <= i < fPdL the value ( ref[ firstPrevCh ][ blockPos –fPdL + i ] + ref[ secondPrevCh ][ blockPos –fPdL + i ]+ 1 ) >>1 is assigned to p[ –fPdL + i ].
		- The extrapolation process to the right from clause 8.3.1 is applied with the input array p, the input array size blockSize and the extension size log2FPdR. The output of this process is assigned to the sample values p[ blockSize +i  ] with 0 <= i < fPdR.
		- The final cross channel prediction sample values pred[ i ] with 0 <= i < blockSize are set to Clip3( minPredVal, maxPredVal, $\left(\sum\_{k=0}^{fSz}p\left[-fPdL+ i+k\right]⋅CCFiltCoeffs\left[k\right]+32\right)\gg 6$)
	+ The extended residual samples resiLeft[ i ] are set to 0 for 0 <=i < tSize.
* Otherwise (blockPos >= tSize), the following applies:

For 0 <= i < tSize, the previous reconstructed sample value currChTpl[ i ] in the current channel on the left template is set to refCurr[ blockPos – tSize + i ], with 0 <= i < tSize.

The array firstPrevChTpl[  ] of previous reconstructed sample values in the first input channel on the left template is defined as

firstPrevChTpl [ i ] = ref[ firstPrevCh ][ blockPos – tSize + i ], with 0 <= i < tSize.

If cc\_pred\_mult\_hyp\_flag is not equal to zero, the array secondPrevChTpl[  ] of previous reconstructed sample values in the second input channel on the left template is defined as

secondPrevChTpl [ i ] = ref[ secondPrevCh ][ blockPos – tSize + i ], with 0 <= i< tSize.

The intermediate cross channel prediction signal samples before filtering and extrapolation to the right p[ i ] with 0 <= i < blockSize + tSize +fPdL are defined as follows:

* If cc\_pred\_offset\_only\_flag is equal to one, the following applies:
	+ The variable sum Diff is defined as follows:
		- If cc\_pred\_mult\_hyp\_flag is equal to zero, one sets
		sumDiff = $\sum\_{i=0}^{tSize-1}(currChTpl\left[i\right]-firstPrevChTpl[i]).$
		- Otherwise (cc\_pred\_mult\_hyp flag is not equal to zero), one sets
		sumDiff = $\sum\_{i=0}^{tSize-1}(currChTpl\left[i\right]- ((firstPrevChTpl\left[i\right]+secondPrevChTpl\left[i\right]+1)\gg 1))$
	+ The variable offset is defined as offset = (sumDiff + (1 << (log2TSize – 1 ) ) ) >> log2TSize
	+ If cc\_pred\_mult\_hyp\_flag is equal to zero, the following applies:
		- If tSize + fPdL <= blockPos, for 0 <= i < blockSize + tSize +fPdL one sets:
		p[ i ] = ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + offset.
		- Otherwise ( tSize + fPdL > blockPos), the following applies:
			* For fPdL <= i <blockSize + tSize +fPdL one sets p[ i ] = ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + offset.
			* The extrapolation process to the left fom clause 8.3.2 is invoked with the input starting position fPdL, the input array size blockSize + tSize, the input array p and the extension size fPdL to obtain the values p[ i ] with 0 <= i < fPdL.
	+ Otherwise ( cc\_pred\_mult\_hyp\_flag is not equal to zero), the following applies:
		- If tSize + fPdL <= blockPos, for 0 <= i < blockSize + tSize +fPdL one sets:
		p[ i ] = ( ( ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + ref[ chIdxScnd ][ blockPos – tSize – fPdL + i ] +1 ) >> 1 )+ offset.
		- Otherwise ( tSize + fPdL > blockPos), the following applies:
			* For fPdL <= i <blockSize + tSize +fPdL one sets p[ i ] = ( ( ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + ref[ chIdxScnd ][ blockPos – tSize – fPdL + i ] +1 ) >> 1 )+ offset.
			* The extrapolation process to the left fom clause 8.3.2 is invoked with the input starting position fPdL, the input array size blockSize + tSize, the input array p and the extension size fPdL to obtain the values p[ i ] with 0 <= i < fPdL.
* Otherwise ( cc\_pred\_offset\_only flag is not equal to one ), if cc\_pred\_mult\_hyp\_flag is equal to zero, the following applies:
	1. The symmetric 2x2 matrix A with the following entries A[ 0 ][ 0 ], A[ 0 ][ 1 ], A[ 1 ][ 0 ], A[ 1 ][ 1 ] is defined:
		+ A[ 0 ][ 0 ] = $\sum\_{i=0}^{tSize-1}firstPrevChTpl\left[ i \right]⋅firstPrevChTpl\left[ i \right]$
		+ A[ 0 ][ 1 ] $=\sum\_{i=0}^{tSize}firstPrevChTpl\left[ i \right]$
		+ $A[ 1 ][ 0 ]= A[ 0 ][ 1 ]$
		+ A[ 1 ][ 1 ] = tSize
	2. The 2 dimensional vector b with the following entries b$[ 0 ]]$and b$[ 1 ]]$s defined:
		+ b[ 0 ]= $\sum\_{i=0}^{tSize-1}currChTpl\left[ i \right]⋅firstPrevChTpl\left[ i \right]$
		+ b[ 1 ]= $\sum\_{i=0}^{tSize-1}currChTpl\left[ i \right]$
	3. The cross component precision ccShift is set to 16
	4. The process of clause 5.10.4 is invoked to obtain the 2 dimensional vector v with entries v[ 0 ] and v[ 1 ] which solves the equation Ax=b in precision ccShift according to that section.
	5. The variable ccShiftOffst is set to ( (1 << ccShift) – 1 ).
	6. The following is applied:
		+ If tSize + fPdL <= blockPos, for 0 <= i < blockSize + tSize +fPdL one sets:
		p[ i ] = ( v[ 0 ]\*ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + v[ 1 ] + ccShiftOffst) ) >> ccShift.
		+ Otherwise ( tSize + fPdL > blockPos), the following applies:
			- For fPdL <= i <blockSize + tSize +fPdL one sets p[ i ] = (v[ 0 ]\*ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + v[ 1 ] + ccShiftOffst) ) >> ccShift.
			- The extrapolation process to the left fom clause 8.3.2 is invoked with the input starting position fPdL, the input array size blockSize + tSize, the input array p and the extension size fPdL to obtain the values p[ i ] with 0 <= i < fPdL.
* Otherwise ( cc\_pred\_offset\_only flag is not equal to one and if\_cc\_pred\_mult\_hyp\_flag is equal to one ), the following applies:
	+ The symmetric 3x3 matric A with the entries A[ k ][ l ] with 0 <= k < 3 and 0 <= l < 3 is defined as follows:
		- A[ 0 ][ 0 ] = $\sum\_{i=0}^{tSize-1}firstPrevChTpl\left[ i \right]⋅firstPrevChTpl\left[ i \right]$
		- A[ 0 ][ 1 ] = $\sum\_{i=0}^{tSize-1}firstPrevChTpl\left[ i \right]⋅secondPrevChTpl\left[ i \right]$
		- A[ 0 ][ 2 ] = $\sum\_{i=0}^{tSize-1}firstPrevChTpl\left[ i \right]⋅$
		- A[ 1 ][ 0 ] = A[ 0 ][ 1 ]
		- A[ 1 ][ 1 ] = $\sum\_{i=0}^{tSize-1}secondPrevChTpl\left[ i \right]⋅secondPrevChTpl\left[ i \right]$
		- A[ 1 ][ 2 ] = $\sum\_{i=0}^{tSize-1}secondPrevChTpl\left[ i \right]⋅$
		- A[ 2 ][ 0 ] = A[ 0 ][ 2 ]
		- A[ 2 ][ 1 ] = A[1 ][ 2 ]
		- A[ 2 ][ 2 ] = tSize
	+ The 3 dimensional vector b with the following entries b[ 0 ], b[ 1 ] and b[ 2 ] is defined as follows:
		- b[ 0 ] = $\sum\_{i=0}^{tSize-1}currChTpl\left[ i \right]⋅firstPrevChTpl\left[ i \right]$
		- b[ 1 ] = $\sum\_{i=0}^{tSize-1}currChTpl\left[ i \right]⋅secondPrevChTpl\left[ i \right]$
		- b[ 2 ] = $\sum\_{i=0}^{tSize-1}currChTpl\left[ i \right]$
	+ The cross component precision ccShift is set to 16.
	+ The process of clause 5.10.4 which solves the equation Ax=b in precision ccShift is invoked to obtain the 3 dimensional vector v with entries v[ 0 ], v[ 1 ] and v[ 2 ].
	+ The following is applied:
		- If tSize + fPdL <= blockPos, for 0 <= i < blockSize + tSize +fPdL one sets
		p[ i ] = (v[ 0 ]\*ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + v[ 1 ]\* ref[ chIdxSecond ][ blockPos – tSize – fPdL + i ] + v[ 2 ]+ccShiftOffst) >> ccShift.
		- Otherwise (tSize + fPdL > blockPos), the following applies:
			* For fPdL <= i < blockSize + tSize +fPdL, one sets
			p[ i ] = (v[ 0 ]\*ref[ chIdxFirst ][ blockPos – tSize – fPdL + i ] + v[ 1 ]\* ref[ chIdxSecond ][ blockPos – tSize – fPdL + i ] + v[ 2 ]+ccShiftOffst) >> ccShift
			* The extrapolation process to the left fom clause 8.4.2 is invoked with the input starting position fPdL, the input array size blockSize + tSize, the input array p and the extension size fPdL to obtain the values p[ i ] with 0 <= i < fPdL.
	+ If cc\_pred\_filter\_flag is equal to zero, the following applies:
		- For 0 <= i < blockSize one sets pred[ i ] = Clip3( minPredVal, maxPredVal, p[ tSize +fPdL + i ] ).
		- For 0 <=i < tSize one sets
		resiLeft[ i ] = currChTpl[ i ] – Clip3( minPredVal, maxPredVal, p[ fPdL +i ] ).
	+ Otherwise (cc\_pred\_filter\_flag is not equal to zero), the following applies:
		- The extrapolation process to the right from clause 8.3.1 is invoked with input array size
		fPdL + tSize +blockSize, input array p and extrapolation size log2FPdR to obtain the values p[ fPdL +tSize + blockSize + k ] with 0 <= k < fPdR.
		- For 0 <= i < blockSize one sets pred[ i ] = Clip3( minPredVal, maxPredVal, $((\sum\_{k=0}^{fSz}p\left[tSize+i+k\right]·CCFiltCoeffs\left[k\right])+32)\gg 6$).
		- For 0 <= j < tSize one sets resiLeft[ i ] = currChTpl[ i ] – Clip3( minPredVal, maxPredVal, $((\sum\_{k=0}^{fSz}p\left[i+k\right]·CCFiltCoeffs\left[k\right])+32)\gg 6$).

### Block matching prediction decoding process

Input to this process are:

* a variable chIdx specifying the current channel,
* a variable blockPos specifying the position of the first sample of the current block,
* a variable log2BlockSize which determines the size of the current block,
* the array of reconstructed samples of the current channel ref[ i ] with 0 < = i < blockPos.
* the parameter log2TSize which determines the size of the adjacent left residual samples to be computed.

Output of this process are the array of block matching prediction sample values pred[ i ] with 0 <= i < (1 <<log2BlockSize) and the array of adjacent left residual samples resiLeft [ j ] with 0 <= j < ( 1 << log2TSize ).

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The variable blockSize is set to 1 << log2BlockSize

The variable tSize is set to 1 << log2TSize.

If blockPos < blockSize, one sets pred[ i ] = 0 for all i with 0 <= i < blockSize and resiLef[ j ] = 0 for all j with 0 <= j < tSize.

Otherwise (blockPos >= blockSize), the following applies:

The variable fPdL which specifies the padding length to the left for the prediction filters of clause 7.3.3.2 is set to 3.

The variable log2FPdR which which determines the length to the right for the prediction filters of clause 7.3.3.2 is set to 2.

The variable fPdR is set to 1 << log2FPdR.

The variable fSz which specifies the filter size for the filters of clause 7.3.3.2 is set as fSz = fPdL + fPdR.

The variable maxBMOffMinusBS is set to min( ( 1 << 16) – 1, (1 << ( Log2MaxBlockSize + 6 ) ).

The variables offsetMinusBSFirst and blockOffsetFirst are derived as follows:

offsetMinusBSFirst =
min(maxBMOffMinusBS, BlockMatchingPredOffsetMinusBlocksSize[ chIdx ][ 0 ])

blockOffsetFirst = min ( blockPos, blockSize + offsetMinusBSFirst ).

If bm\_pred\_mult\_hyp\_flag is equal to 1, the variables offsetMinusBSScnd and blockOffsetScnd are derived as follows:

offsetMinusBSScnd =
min ( maxBMOffMinusBS, BlockMatchingPredOffsetMinusBlocksSize[ currCh ][ 1 ] ).

blockOffsetScnd =min( blockPos, blockSize + offsetMinusBSScnd).

If blockOffsetFirst < blockSize + fPdR or if bm\_pred\_mult\_hyp\_flag is equal to 1 and blockOffsetScnd < blockSize + fPdR , the exptrapolation process to the right from clause 8.3.1is invoked with input array size blockPos, input array ref and extension size log2FPdR to obtain the reference sample values ref[ i ] with blockPos <= i < blockPos + fPdR.

The variable minPos is set to max( 0, blockPos – maxBMOffMinusBS – blockSize ).

If blockPos – blockOffsetFirst –fPdL < minPos or if bm\_pred\_mult\_hyp\_flag is equal to 1 and
blockPos – blockOffsetScnd –fPdL < minPos, the extrapolation process to the left from clause 8.3.2 is invoked with input starting position minPos, input array size blockPos, input array ref and extension size fPdL to obtain the reference sample values ref [ minPos – fPdL + i  ] with 0 <= i < fPdL.

The intermediate prediction sample values of the first hypothesis pFirst[ i ] with 0 <= i < blockSize are derived as follows:

* If bm\_pred\_filter\_flag[ 0 ] is equal to 0, one puts
pFirst[ i ] = ref[ blockPos – blockOffsetFirst +  i ].
* Otherwise ( bm\_pred\_filter\_flag[ 0 ] is not equal to 0 ), one puts
pFirst[ i ] =
 $(\left(\sum\_{k=0}^{fSz}ref\left[blockPos – blockOffsetFirst-fPdL + i+k\right]⋅BMFiltCoeffs[0][k] )+32\right)\gg 6.$

If bm\_pred\_mult\_hyp\_flag is equal to 1, the intermediate prediction values of the second hypothesis pScnd[ i ] with 0 <= i < blockSize are derived as follows:

* If bm\_pred\_filter\_flag[ 1 ] is equal to 0, one puts pScnd[ i ] = ref[ currCh ][ blockPos– blockOffsetScnd +  i ].
* Otherwise ( bm\_pred\_filter\_flag[ 1 ] is not equal to 0 ), one puts
pScnd[ i ]
= $(\left(\sum\_{k=0}^{fSz}ref\left[ blockPos – blockOffsetScnd -fPdL+i+k\right]⋅BMFiltCoeffs[1][k] )+32\right)\gg 6.$

The extended first left prediction sample values pFirstLeftExt[ i ] with 0 <= i < tSize are derived as follows:

* If bm\_pred\_filter\_flag[ 0 ] is equal to 0, the following applies:
	+ If blockPos – blockOffsetFirst –tSize < minPos – fPdL, one sets pFirstLeftExt[ i ] = 0.
	+ Otherwise (blockPos – blockOffsetFirst– tSize >= minPos – fPdL ), one sets
	pFirstLeftExt[ i ] = ref[ blockPos –tSize – blockOffsetFirst + i ] .
* Otherwise ( bm\_pred\_filter\_flag[ 0 ] is not equal to 0), the following applies:
	+ If blockPos – blockOffsetFirst – tSize < minPos, one sets pFirstLeftExt[ i ] = 0.
	+ Otherwise (blockPos – blockOffsetFirst – tSize >= minPos), one sets
	pFirstLeftExt[ i ] = $(\left(\sum\_{k=0}^{fSz}ref\left[ blockPos – tSize- blockOffsetFirst -fPdL+i+k\right]⋅BMFiltCoeffs[0][k] )+32\right)\gg $6.

If bm\_pred\_mult\_hyp\_flag is equal to 1, the extended second left prediction sample values pScndLeftExt[ i ] with 0 <= i < tSize are derived as follows:

* If bm\_pred\_filter\_flag[ 1 ] is equal to 0, the following applies:
	+ If blockPos – blockOffsetScnd – tSize < minPos – fPdL, one sets pScndLeftExt[ i ] = 0.
	+ Otherwise (blockPos – blockOffsetScnd– tSize >= minPos – fPdL), one sets
	pScndLeftExt[ i ] = ref[ blockPos –tSize – blockOffsetScnd + i ] .
* Otherwise ( bm\_pred\_filter\_flag[ 1 ] is not equal to 0), the following applies:
	+ If blockPos – blockOffsetScnd – tSize < minPos, one sets pScndLeftExt[ i ] = 0.
	+ Otherwise (blockPos – blockOffsetScnd – tSize >= minPos), one sets
	pScndLeftExt[ i ] = $(\left(\sum\_{k=0}^{fSz}ref\left[ blockPos – tSize- blockOffsetScnd -fPdL+i+k\right]⋅BMFiltCoeffs[1][k] )+32\right)\gg $6.

The variable diffTpl is derived as follows:

* If bm\_pred\_add\_offset\_flag is equal to zero, one sets diffTpl = 0
* Otherwise if blockPos < tSize, one sets diffTpl = 0.
* Otherwise (bm\_pred\_add\_offset\_flag is not equal to zero and blockPos >= tSize), the following applies:
	+ If bm\_pred\_mult\_hyp\_flag is equal to 0, one sets
	diffTpl = $\sum\_{i=0}^{tSize-1}(ref\left[blockPos-tSize+i\right]-pFirstLeftExt\left[i\right])$.
	+ Otherwise (bm\_pred\_mult\_hyp\_flag is not equal to 0), one set
	diffTpl = $\sum\_{i=0}^{tSize-1}(ref\left[blockPos-tSize+i\right]-(\left(pFirstLeftExt\left[i\right]+pScndLeftExt\left[i\right]+1\right)\gg 1).$

The variable offsetAdd is derived as offsetAd = ( diffTpl + ( 1 << ( log2TSize – 1 ) )) >> log2TSize.

The final block matching prediction sample values pred[ i ] with 0 <= i < blockSize are derived as follows:

* If bm\_pred\_mult\_hyp\_flag is equal to 0, one sets pred[ i ] = Clip3( minPredVal, maxPredVal, pFirst[ i ] + offsetAd ).
* Oterwise (bm\_pred\_mult\_hyp\_flag is not equal to 0), one sets
pred[ i ] = Clip3( minPredVal, maxPredVal, ((pFirst[ i ] + pSecond[ i ] + 1 ) >> 1 ) + offsetAd).

The final adjacent left residual sample values resiLeft[ i ] with 0 <= i < tSize are derived as follows:

* If blockPos < tSize, one sets reiLeft[ i  = 0
* Otherwise ( blockPos >= tSize ), the following applies:
	+ If bm\_pred\_mult\_hyp\_flag is equal to 0, one sets
	resiLeft[ i ] = ref[blockPos $–$ tSize + i] – Clip3( minPredVal, maxPredVal,pFirst$LeftExt$ [ i ] + offsetAd).
	+ Otherwise (bm\_pred\_mult\_hyp\_flag is not equal to 0), one sets
	resiLeft[ i ] = ref[blockPos $–$ tSize + i] – Clip3(minPredVal, maxPredVal, ((pFirst$LeftExt$ [ i ] + pSecond$LeftExt$ [ i ] + 1 ) >> 1 ) + offsetAd.

## Scaling and inverse transformation decoding process

This process specifies the computation of the reconstructed residual sample values res[ i ] with 0 <= i < blockSize.

The following steps are applied:

* The process of Section 8.4.1 is invoked. The output of this process is assigned to the residual transform coefficients tCoeff [ i ] with 0 <= i < Log2BlockSize.
* The process of Section 8.4.2 is invoked with the variable blockSize set to ( 1 << Log2BlockSize ) and the residual transform coefficients tCoeffCurr[ i ] set to tCoeff[ i ] for 0 <= i < ( 1 << Log2BlockSize ). The output sample values of this process are assigned to the intermediate residual values resImd[ i ] with 0 <= i < ( 1 << Log2BlockSize ).

### Scaling process for transform coefficient levels

Output of this process are the intermediate reconstructed residual transform coefficients tCoeff[ i ] with 0 <= i < (1 << Log2BlockSize).

The variable ssShift is set to be equal to max(0, BitDepthMax – 17 ) .

The variable ssDvdr is set to be equal to 1<<( max(0, 17 – BitDepthMax ) ).

The variable leftShiftTrigTrafo is set to be equal to 30 – BitDepthMax – ( ( Floor( Log2 (Max (1, 1 << ( Log2BlockSize – 4) ) ) ) + 1) >> 1).

The variable maxCoeffVal is set to be equal to (1<<31) – 1 and the variable minCoeffVal is set to be equal to – maxCoeffCal – 1.

The values of the variables qScale and qShift are determined as follows:

* If CurrBlockQP[ch] is equal to 1, qScale is set to be equal to ssDvdr<<2 and qShift is set to be equal to max(0, 17-bitDepthMax)+2.
* Otherwise (CurrBlockQP[ch] is not equal to 1), qScale is set to be equal to ( ( ( CurrBlockQP[ch]\* CurrBlockQP[ch]) >> 2) << ssShift ) + ssDvdr and qShift is set to be equal to max( 0, 17 – BitDepthMax )

The reconstructed residual transform coefficients tCoeff[ i ] with 0 <= i < ( 1 << Log2BlockSize ) are derived as follows:

* If transform\_skip\_flag is equal is equal to 1, the following applies:
	+ If qShift > 0, the value (Abs( QuantIndices[ i ] ) \* qScale + (1 << (qShift – 1) ))>>qShift is assigned to the variable absValCurr and the value ( QuantIndices[ i ] >= 0 ? 1: –1 )\*absValCurr is assigned to tCoeff [ i ].
	+ Otherwise, (qShift <= 0), the value Clip3( minCoeffVal, maxCoeffVal, QuantIndices[ i ] \* ( qScale << ( – qShift ) ) ) is assigned to the variable tCoeff [ i ].
	+ Otherwise (transform\_skip\_flag is not equal to 1) the following applies:
	+ The variable qShift is modified to qShift – leftShiftTrigTrafo
	+ If qShift < 0, the variable qScale is modified to ( qScale << ( – qShift) ) and the variable qShift is modified to 0
	+ The variable offset is set to be equal to (1<<(qShift – 1 ))
	+ The variable tCoeff [ i ] is determined by the following pseudo-code process:

 nextState = 0
 for( i = last\_scan\_pos, i >=0 , i = i – 1 ){
 offsetCurr = ( QuantIndices[ i ] < 0 ) ? nextState & 1 : – ( nextState & 1 )
 absValCurr =( ( Abs( QuantIndices[ i ] ) << 1) – ( nextState & 1 ) ) \* qScale + offset ) >> qShift
 tCoeff[ i ] = QuantIndices[ i ] < 0 ? max( – absValCurr, minCoeffVal, ) : min (absValCurr, maxCoeffVal )
 nextState = QStateTransTabel[ nextState ][ QuantIndices[ i ] & 1 ]
 }

### Inverse transformation process

Input to this process are a block size bSizeCurr and reconstructed transform coefficients tCoeffCurr[ i ] with 0 <= i < bSizeCurr. Output of this process are the intermediate residual sample values resImdCurr[ i ] with 0 <= i < bSizeCurr.

The following is applied

* If transform\_skip\_flag is equal to 1, the values tCoeffCurr[ i ] are assigned to the residual samples resImd[ i ].
* Otherwise ( transform\_skip\_flag is not equal to 1), the following applies:
	+ If transform\_dst\_flag is equal to 1, a DST-II transform process which will be specified in a later version of this draft is applied where the values tCoeffCurr[ i ] are assigned to the input of this process. The output of this process is assigned to the residual sample values resImd[ i ].
	+ Otherwise ( transform\_dst\_flag is not equal to 1), a DCT-II transform process which will be specified in a later version of this draft is applied where the values tCoeffCurr[ i ] are assigned to the input of this process. The output of this process is assigned to the residual sample values resImd[ i ].

## Sample wise prediction decoding process

### Overview

Input to this process are:

* the current channel index currCh,
* the current block position currBlockPos,
* the array of reconstructed samples of previous channels rec[ c ][ i ] with
max( currCh – ( DepChMask & currCh ), 0 ) <= c < currCh and with 0 <= i < currBlockPos + ( 1 << Log2BlockSize ),
* the array of reconstructed samples of the current channel rec[ currCh ][ i ] with 0 <= i < currBlockPos,
* the value maxTemplateSize,
* the array of left residual values resiLeftCurr[ j ] with 0 <= j < maxTemplateSize,
* the array of reconstructed intermediate residual sample values resImdCurr[ i ] with 0 <= i <
( 1 << Log2BlockSize ).

Output to this process are the residual sample values res[ i ] with 0 <= i < ( 1 << Log2BlockSize ). These values are derived as follows:

* If SamplePredMoed is equal SPM\_OFF, for 0 <= i < ( 1 << Log2BlockSize ) one sets res[ i ] = resImdCurr[ i ].
* If SamplePredMode is equal to SPM\_DIFFS, the process of clause 8.5.2 is invoked with blockSize set to
(1 << Log2BlockSize), resiLeftVal set to resiLeftCurr[ maxTemplateSize – 1 ] and resImd set to resImdCurr .
* If SamplePredMode is equal to SPM\_SLOPE, the process of clause 8.5.3 is invoked with blockSize set to
(1 << Log2BlockSize), resiLeftValFirst set to resiLeftCurr[ maxTemplateSize – 1 ], resiLeftValSecond set to resiLeftCurr[ maxTemplateSize – 2 ] and resImd set to resImdCurr .
* If SamplePredMode is equal to SPM\_HALF\_SLOPE, the process of clause 8.5.4 is invoked with blockSize set to (1 << Log2BlockSize), resiLeftValFirst set to resiLeftCurr[ maxTemplateSize – 1 ], resiLeftValSecond set to resiLeftCurr[ maxTemplateSize – 2 ] and resImd set to resImdCurr .
* If SamplePredMode is equal to SPM\_LPC, the following applies:
	+ The variable fltrSzCurrCh is set to 1 << (lpf\_num\_weights\_idx+1).
	+ If lpc\_use\_prev\_ch\_flag is equal 1, the following applies:
		- The process from clause 8.5.6 is invoked with fSCurrCh set to fltrSzCurrCh and numPrevCh set to min( 3, currCh & DepChMask ) to obtain the filter weights for the current channel wghtCurrCh[ k ] with 0 <= k < fSCurrCh and the filter weights for the previous channel wghtPrevCh[ k ] with 0 <= k <= numPrevCh.
		- The process from clause 8.5.8 is invoked with blockSize set to (1 << Log2BlockSize), fSCurrCh set to fltrSzCurrCh, numPrevCh set to min( 3, currCh & DepChMask ), wCurrCh set to wghtCurrCh, wPrevCh set to wghtPrevCh, refPrev[ c ][ l ] set to rec[ currCh – 1 – c ][ l ], where 0 <= c < min( 3, currCh & DepChMask ) and 0 <= l < ( 1 << Log2BlockSize), resiLeft[ k ] set to resiLeftCurr[ maxTemplateSize – fltrSzCurrCh + k ] for 0 <= k < fltrSzCurrCh and resImd set to resImdCurr.
	+ Otherwise (lpc\_use\_prev\_ch\_flag is not equal to 1), the following applies:
		- The process from clause 8.5.5 is invoked with fltrSz set to fltrSzCurrCh
		to obtain the filter weights for the current channel wghtCurr[ k ] with 0 <= k < fSCurrCh.
		- The process from clause 8.5.7 is invoked with blockSize set to (1 << Log2BlockSize), fltrSz set to fltrSzCurrCh, w set to wghtCurr, resiLeft[ k ] set to resiLeftCurr[ maxTemplateSize – fltrSzCurrCh + k ] for 0 <= k < fltrSzCurrCh and resImd set to resImdCurr.

### Sample wise one tap prediction decoding process

Input to this process are:

* a variable blockSize which determines the size of the current block,
* an adjacent left residual value resiLeftVal
* an array of intermediate reconstructed residual samples resImd[ j ] with 0 <= j < blockSize.

Output of this process are the reconstructed residual samples res[ i ] with 0 <= i < blockSize.

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The array of input values q[ k ] with 0 <= k < blockSize + 1 is initialized with 0.

One sets q[ 0 ] = resiLeftVal

The following process is invoked:

 Set j = 1

 do

 q[ j ] = q[ j – 1 ] + resImd[ j – 1  ]

 j = j+1

 while( j < blockSize + 1)

For 0 <= i < blockSize one sets res[ i ] =Clip3( minResVal, maxResVal, q[ 1 + i ] ).

###  Sample wise full slope prediction decoding process

Input to this process are:

* a variable blockSize which determines the size of the current block,
* a directly adjacent left residual value resiLeftValFirst
* a penultimately adjacent left residual value resiLeftValSecond
* an array of intermediate reconstructed residual samples resImd[ j ] with 0 <= j < blockSize.

Output of this process are the reconstructed residual samples res[ i ] with 0 <= i < blockSize.

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The array of input values q[ k ] with 0 <= k < blockSize + 2 is initialized with 0.

One sets q[ 0 ] = resiLeftValSecond and q[ 1 ] = resiLeftValFirst.

The following process is invoked:

 Set j = 2.

 do

 q[ j ] = ( (q[ j – 1 ] – q[ j – 2 ] ) <<1 ) + resImd[ j – 2 ].

 j = j+1

 while( j < blockSize + 2)

For 0 <= i < blockSize one sets res[ i ] =Clip3( minResVal, maxResVal, q[ 2 + i ] ).

### Sample wise half slope prediction decoding process

Input to this process are:

* a variable blockSize which determines the size of the current block,
* a directly adjacent left residual value resiLeftValFirst
* a penultimately adjacent left residual value resiLeftValSecond
* an array of intermediate reconstructed residual samples resImd[ j ] with 0 <= j < blockSize.

Output of this process are the reconstructed residual samples res[ i ] with 0 <= i < blockSize.

The variable maxPredVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minPredVal is set to – maxPredVal – 1.

The array of input values q[ k ] with 0 <= k < blockSize + 2 is initialized with 0.

One sets q[ 0 ] = resiLeftValSecond and q[ 1 ] = resiLeftValFirst.

The following process is invoked:

 Set j = 2.

 do

 q[ j ] = q[ j – 1 ] + ( ( q[ j – 1 ] – q[ j – 2 ] +1) >> 1 ) + resImd[ j – 2 ].

 j = j+1

 while( j < blockSize + 2 )

For 0 <= i < blockSize one sets res[ i ] =Clip3( minResVal, maxResVal, q[ 2 + i ] ).

### Filter coefficient decoding process for single channel linear predictive filtering

Input to this process is a filter size fltrSz.

Output of this process are the reconstructed filter coefficients w[ k ] with 0 < =k < fltrSz.

The variable weightPrec is set to 14.

The intermediate temporal coefficient values tempImdCoeff[ k ] with 0 <= k < fltrSz are defined as

tempImdCoeff[ k ] =( 64 – Abs( LPFWeightsCurr[ k ] ) ) << 1.

The temporal coefficient values tempCoeff[ k ] with 0 <= k < fltrSz are defined as

tempCoeff[ k ] =( ( 1 << 14 ) – tempImdCoeff[ k ]\* tempImdCoeff[ k ] )\*

( lpf\_weight\_sign\_flag[ k ] > 0 ? – 1: 1 )

The following process is invoked:

 Set i = 0

 do

 a[ i ] = tempCoeff[ i ]

 set j = 0

 do

 b [ j ] = a[ j ] – tempCoeff[ i ]\*a[ i – j – 1 ]

 while( j < i)

 set j = 0

 do

 a[ j ] = b[ j ]

 while ( j < i )

 while( i < fltrSz )

### Filter coefficient decoding process for multi channel linear predictive filtering

Input to this process are:

* a variable fSCurrCh which specifies the number of prediction filter coefficients for the current channel,
* a variable numPrevCh which specifiec the number of previous channels.

Output of this process are the filter coeffcients for the current channel wCurrCh[ k ] with 0 <= k < fSCurrCh and the filter coefficients for the previous channel wPrevCh[ c ] with 0 <= c <= numPrevCh.

For 0 <= k < fSCurrCh, one sets wCurrCh[ k ] = LPFWeightsCurr[ fsCurrCh + numPrevCh – k ].

For 0 <= c <= numPrevCh, one sets wPrevCh[ c ] = LPFWeightsCurr[ numPrevCh – c ].

### Single channel linear predictive filtering prediction decoding process

Input to this process are:

* a variable blockSize which specifies the size of the current block,
* a variable fltrSz which specifies the number of prediction filter coefficients,
* an array of prediction filter coefficients w[ k ] with 0 <= k < fltrSz,
* an array of left adjacent residual samples resiLeft[ j ] with 0 <= j <fltrSz,
* an array of intermediate reconstructed residual samples resImd[ j ] with 0 <= j < blockSize.

Output of this process are the reconstructed residual samples res[ i ] with 0 <= i < blockSize.

The variable maxResVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minResVal is set to – maxPredVal – 1.

The array of input values q[ k ] with 0 <= k < blockSize + fltrSz is initialized with 0.

For 0 <= j < fltrSz one sets q[ j ] = resiLeft[ j ].

One sets leftShiftFltr = 14.

One sets offsetLeftShift = 1 << ( leftShiftFltr – 1 ).

The following process is invoked:

 Set j = fltrSz.

 do

 q[ j ] = ( ( ( $\sum\_{k=0}^{fltrSz-1}w\left[k\right]⋅q[j-fltrSz+k]$ ) + offsetLeftShift) ) >>leftShiftFltr )+ resImd[ j – fltrSz ].

 j = j+1

 while( j < blockSize + fltrSz)

For 0 <= i < blockSize one sets res[ i ] =Clip3( minResVal, maxResVal, q[ fltrSz + i ] ).

### Multi channel linear predictive filtering prediction decoding process

Input to this process are:

* a variable blockSize which specifies the size of the current block,
* a variable fSCurrCh which specifies the number of prediction filter coefficients for the current channel,
* a variable numPrevCh, 0 < numPrevCh <= 3 which specifies the number of previous channels,
* an array of prediction filter coefficients for the current channel wCurrCh [ k ] with 0 <= k < fSCurrCh,
* an array of prediction filter coefficients for the previous channel wPrevCh[ c ] with 0 <= c <= numPrevCh,
* an arry of reconstructed sample values of previous channels refPrev[ c ][ l ], where 0 <= c < numPrevCh and
0 <= l < blockSize.
* an array of left adjacent residual samples resiLeft[ j ] with 0 <= j < fSCurrCh,
* an array of intermediate reconstructed residual samples resImd[ j ] with 0 <= j < blockSize.

Output of this process are the reconstructed residual samples res[ i ] with 0 <= i < blockSize.

The variable maxResVal is set to ( 1 << ( BitDepthMax – 1 ) ) – 1.

The variable minResVal is set to – maxResVal – 1.

The array of input values q[ k ] with 0 <= k < blockSize + fltrSz is initialized with 0.

For 0 <= j < fSCurrCh one sets q[ j ] = resiLeft[ j ].

One sets leftShiftFltr = 14.

One sets offsetLeftShift = 1 << ( leftShiftFltr – 1 ).

The following process is invoked:

 Set j = fSCurrCh.

 do

 q[ j ] = ( ( ( $\sum\_{k=0}^{fSCurrCh-1}wCurrCh\left[k\right]⋅q[j-fSCurrCh+k]$ )

 + $\sum\_{c=0}^{numPrevCh-1}( wPrevCh\left[c\right]⋅refPrev\left[c\right][j-fSCurrCh]$ + wPrevCh[ numPrevCh ])

 +offsetLeftShift )>> leftShiftFltr) + resImd[ j – fSCurrCh ].

 j = j+1

 while( j < blockSize + fSCurrCh )

For 0 <= i < blockSize one sets res[ i ] =Clip3 ( minResVal, maxResVal, q[ fSCurrCh + i ] ).

# Parsing process

## General

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), se(v), or ae(v) (see clause 9.3).

## Parsing process for k-th order Exp-Golomb codes

### General

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v) or se(v).

Inputs to this process are bits from the RBSP.

Outputs of this process are syntax element values.

Syntax elements coded as ue(v) or se(v) are Exp-Golomb-coded with order k equal to 0. The parsing process for these syntax elements begins with reading the bits starting at the current location in the bitstream up to and including the first non-zero bit, and counting the number of leading bits that are equal to 0. This process is specified as follows:

leadingZeroBits = −1
for( b = 0; !b; leadingZeroBits++ ) (60)
 b = read\_bits( 1 )

The variable codeNum is then assigned as follows:

codeNum = ( 2leadingZeroBits − 1 ) \* 2k + read\_bits( leadingZeroBits + k ) (61)

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

Table 8 illustrates the structure of the 0-th order Exp-Golomb code by separating the bit string into "prefix" and "suffix" bits. The "prefix" bits are those bits that are parsed for the computation of leadingZeroBits, and are shown as either 0 or 1 in the bit string column of Table 8. The "suffix" bits are those bits that are parsed in the computation of codeNum and are shown as xi in Table 8, with i in the range of 0 to leadingZeroBits − 1, inclusive. Each xi is equal to either 0 or 1.

Table 8 – Bit strings with "prefix" and "suffix" bits and assignment to codeNum ranges (informative)

|  |  |
| --- | --- |
| **Bit string form** | **Range of codeNum** |
| 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 9 illustrates explicitly the assignment of bit strings to codeNum values.

Table 9 – Exp-Golomb bit strings and codeNum in explicit form and used as ue(v) (informative)

|  |  |
| --- | --- |
| **Bit string** | **codeNum** |
| 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 codeNum.
* Otherwise (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 9.2.2 with codeNum as input.

### Mapping process for signed Exp-Golomb codes

Input to this process is codeNum as specified in clause 9.2.1.

Output of this process is a value of a syntax element coded as se(v).

The syntax element is assigned to the codeNum by ordering the syntax element by its absolute value in increasing order and representing the positive value for a given absolute value with the lower codeNum. Table 10 provides the assignment rule.

Table 10 – Assignment of syntax element to codeNum for signed Exp-Golomb coded syntax elements se(v)

|  |  |
| --- | --- |
| **codeNum** | **syntax element value** |
| 0 | 0 |
| 1 | 1 |
| 2 | −1 |
| 3 | 2 |
| 4 | −2 |
| 5 | 3 |
| 6 | −3 |
| k | (−1)k + 1 \* Ceil( k ÷ 2 ) |

## CABAC parsing process for frame data

### General

Inputs to this process are a request for a value of a syntax element and values of prior parsed syntax elements.

Output of this process is the value of the syntax element.

The initialization process as specified in clause 9.3.2 is invoked when starting the parsing of the general frame data syntax specified in clause 7.3.3.1.

The parsing of syntax elements proceeds as follows:

For each requested value of a syntax element a binarization is derived as specified in clause 9.3.3.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in clause 9.3.4.

### Initialization process

#### General

Outputs of this process are initialized CABAC internal variables.

The context variables of the arithmetic decoding engine are initialized as follows:

– The initialization process for context variables is invoked as specified in clause 9.3.2.2.

The decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision are initialized by invoking the initialization process for the arithmetic decoding engine as specified in clause 9.3.2.3.

#### Initialization process for context variables

Outputs of this process are the initialized CABAC context variables indexed by ctxTable and ctxIdx.

For each context variable, the three variables pBinCount, pStateIdx0 and pStateIdx1 are initialized as follows:

* Variable pBinCount is set to 63.
* Table 12 to Table 13 contain the values of the 12 bit variable initValue used in the initialization of context variables that are assigned to all syntax elements in clauses 7.3.3.1 through **Fehler! Verweisquelle konnte nicht gefunden werden.**, except end\_of\_frame\_sequence\_flag, end\_of\_truncated\_frame\_sequence\_flag, num\_samples\_per\_channel\_to\_discard, and end\_of\_frame\_one\_bit.
* From the 12 bit table entry initValue, the two 2 bit variables qpPosProb0 and log2qpRange and the two 4 bit variables probStart and probEnd are derived as follows:

qpPosProb0 = ( 2 << ( initValue & 3 ) ) − 2
log2qpRange = ( ( initValue >> 2 ) & 3 ) + 3
probStart = ( ( initValue >> 4 ) & 15 ) \* 8
probEnd = ( ( initValue >> 8 ) & 15 ) \* 8 (62)

* The variables slopeMul and valAdd, used in the initialization of context variables, are derived from probStart, probEnd and log2qpRange as follows:

slopeMul = probEnd − probStart
valAdd = ( ( 1 << log2qpRange ) >> 1 (63)

* The two values assigned to pStateIdx0 and pStateIdx1 for the initialization are derived from if\_indep\_init\_block\_qp. Given the variables m and n, the initialization is specified as follows:

currQP = if\_indep\_init\_block\_qp − qpPosProb0
preCtxState = ( currQP \* slopeMul + ( probStart << log2qpRange ) + add ) >> log2qpRange (64)

* The two values assigned to pStateIdx0 and pStateIdx1 for the initialization are derived as follows:

pStateIdx0 = Clip3( 32, 992, preCtxState << 3 ) − 512
pStateIdx1 = Clip3( 256, 7936, preCtxState << 6 ) − 4096 (65)

NOTE – The variables pStateIdx0 and pStateIdx1 correspond to the probability state indices as further described in clause 9.3.4.3.

Table 11 lists the range of ctxIdx values for which initialization is needed for each of the three initialization types, specified by the variable initType. It also lists the table number that includes the values of initValue needed for the initialization for each value of ctxIdx. The derivation of initType depends on the value of the if\_ctx\_init\_flag syntax element and on the value of the if\_ctx\_init\_mode syntax element. The variable initType is derived as follows:

if( !if\_ctx\_init\_flag )
 initType = 0
else if( if\_ctx\_init\_mode == 0 )
 initType = 1 (66)
else
 initType = 2

| Table 11 – Association of ctxIdx and syntax elements for each initializationTypein the initialization process |
| --- |
| **Syntax structure** | **Syntax element** | **ctxTable** | **initType** |
| **0** | **1** | **2** |
| frame\_data( ) | block\_split\_log2 | Table XX | 0..8 | 9..17 | 18..26 |
| block\_matching\_or\_cross\_channel\_pred\_flag | Table XX | 0 |  |  |
| cross\_channel\_pred\_flag | Table XX | 0 |  |  |
| block\_pred\_mode | Table XX | 0 |  |  |
| block\_abs\_delta\_qp | Table XX | 0..7 |  |  |
| block\_delta\_zlsb\_present\_flag | Table XX | 0 |  |  |
| transform\_skip\_flag | Table XX | 0 |  |  |
| cross\_channel\_prediction\_data( ) | cc\_pred\_offset\_only\_flag | Table XX | 0 |  |  |
| cc\_pred\_filter\_flag | Table XX | 0 |  |  |
| cc\_pred\_filter\_idx | Table XX | 0 |  |  |
| cc\_pred\_mult\_hyp\_flag | Table XX | 0 |  |  |
| cc\_pred\_abs\_chd\_greater0\_flag[ ] | Table XX | 0 |  |  |
| cc\_pred\_abs\_chd\_minus1[ ] | Table XX | 0..15 |  |  |
| block\_matching\_prediction\_data( ) | bm\_pred\_mult\_hyp\_flag | Table XX | 0 |  |  |
| bm\_pred\_add\_offset\_flag | Table XX | 0 |  |  |
| bm\_pred\_filter\_flag[ ] | Table XX | 0 |  |  |
| bm\_pred\_filter\_idx[ ] | Table XX | 0 |  |  |
| bm\_pred\_off\_pred\_prev\_ch\_flag[ ]  | Table XX | 0 |  |  |
| bm\_pred\_abs\_offd\_greater0\_flag[ ]  | Table XX | 0 |  |  |
| bm\_pred\_abs\_offd\_minus1[ ] | Table XX | 0..15 |  |  |
| sample\_pred\_mode( ) | spred\_lpf\_or\_diff\_flag | Table XX | 0 |  |  |
| spred\_lpf\_flag | Table XX | 0. |  |  |
| spred\_rem\_mode\_idx | Table XX | 0. |  |  |
| linear\_predictive\_filtering\_data( ) | lpf\_prev\_ch\_flag  | Table XX | 0 |  |  |
| lpf\_delta\_coding\_flag | Table XX | 0 |  |  |
| lpf\_num\_weights\_idx | Table XX | 0..9 |  |  |
| abs\_lpf\_weight\_greater0\_flag[ ] | Table XX | 0 |  |  |
| abs\_lpf\_weight\_minus1[ ] | Table XX | 0 |  |  |
| quant\_res\_sample\_data( ) | abs\_tskip\_coeff\_gt0\_flag[ ] | Table XXX | 0 |  |  |
| abs\_tskip\_coeff\_offset[ ] | Table XXX | 0..3 |  |  |
| abs\_tskip\_coeff\_rem\_prefix[ ] | Table XXX | 0..7 |  |  |
| abs\_tskip\_coeff\_rem\_eg0\_suffix[ ] | Table XXX | 0..30 |  |  |
| last\_sbb\_index\_gt0\_flag | Table XXX | 0 |  |  |
| last\_sbb\_index\_rem | Table XXX | 0..14 |  |  |
| abs\_trafo\_coeff\_gt0\_flag[ ] | Table XXX | 0..53 |  |  |
| abs\_trafo\_coeff\_offset[ ] | Table XXX | 0..8 |  |  |
| abs\_trafo\_coeff\_remainder[ ] | Table XXX | 0..30 | 31..61 | 62..92 |

Table 12 – Specification of initValue for ctxIdx of block\_split\_log2

|  |  |
| --- | --- |
| **Initialization variable** | **ctxIdx of block\_split\_log2** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **initValue** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** |  |  |  |  |  |
| **initValue** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Add init tables later.

Table 13 – Specification of initValue for ctxIdx of abs\_trafo\_coeff\_remainder

|  |  |
| --- | --- |
| **Initialization variable** | **ctxIdx of abs\_trafo\_coeff\_remainder** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| **initValue** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

#### Initialization process for the arithmetic decoding engine

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 the 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 9.3.4.3. The arithmetic decoding process for a binary decision (DecodeDecision) as specified in clause 9.3.4.3.2 and the decoding process for a binary decision before termination (DecodeTerminate) as specified in clause 9.3.4.3.5 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 9.3.4.3.4 requires a minimum register precision of 10 bits for the variable ivlOffset and a minimum register precision of 9 bits for the variable ivlCurrRange.

### Binarization process

#### General

Input to this process is a request for a syntax element.

Output of this process is the binarization of the syntax element.

Table 14 specifies the type of binarization process associated with each syntax element and corresponding inputs.

The specification of the truncated Rice (TR) binarization process, the truncated binary (TB) binarization process, the k-th order Exp-Golomb (EGk) binarization process, the limited k-th order Exp-Golomb (LEGk) binarization process, and the fixed-length (FL) binarization process are given in clauses 9.3.3.2 through 9.3.3.6, respectively.

| Table 14 – Syntax elements and associated binarizations |
| --- |
| **Syntax structure** | **Syntax element** | **Binarization** |
| **Process** | **Input parameters** |
| frame\_data( ) | end\_of\_frame\_sequence\_flag | FL | cMax = 1 |
| block\_split\_log2 | TR | cMax = MaxSplitDepth, cRiceParam = 0 |
| block\_matching\_or\_cross\_channel\_pred\_flag | FL | cMax = 1 |
| cross\_channel\_pred\_flag | FL | cMax = 1 |
| block\_pred\_mode | FL | cMax =2 |
| block\_abs\_delta\_qp  | LEG0 | maxPreExtLen = if\_max\_abs\_delta\_qp\_idx  |
| block\_delta\_qp\_sign\_flag | FL | cMax = 1 |
| block\_delta\_zlsb\_present\_flag | FL | cMax = 1 |
| block\_delta\_zlsb\_sign\_flag | FL | cMax = 1 |
| transform\_skip\_flag | FL | cMax = 1 |
| transform\_dst\_flag | FL | cMax = 1 |
| end\_of\_truncated\_frame\_sequence\_flag | FL | cMax = 1 |
| end\_of\_frame\_one\_bit | FL | cMax = 1 |
| cross\_channel\_prediction\_data( ) | cc\_pred\_offset\_only\_flag | FL | cMax = 1 |
| cc\_pred\_filter\_flag | FL | cMax = 1 |
| cc\_pred\_filter\_idx | FL | cMax = 1 |
| cc\_pred\_mult\_hyp\_flag | FL | cMax = 1 |
| cc\_pred\_abs\_chd\_greater0\_flag[ ] | FL | cMax = 1 |
| cc\_pred\_abs\_chd\_minus1[ ] | LEG0 | maxPreExtLen = 15 |
| cc\_pred\_chd\_sign\_flag[ ] | FL | cMax = 1 |
| block\_matching\_prediction\_data( ) | bm\_pred\_mult\_hyp\_flag | FL | cMax = 1 |
| bm\_pred\_add\_offset\_flag | FL | cMax = 1 |
| bm\_pred\_filter\_flag[ ] | FL | cMax = 1 |
| bm\_pred\_filter\_idx[ ] | FL | cMax = 1 |
| bm\_pred\_off\_pred\_prev\_ch\_flag[ ] | FL | cMax = 1 |
| bm\_pred\_abs\_offd\_greater0\_flag[ ] | FL | cMax = 1 |
| bm\_pred\_abs\_offd\_minus1[ ] | LEG0 | maxPreExtLen = 15 |
| bm\_pred\_offd\_sign\_flag[ ] | FL | cMax = 1 |
| sample\_pred\_mode( ) | spred\_lpf\_or\_diff\_flag | FL | cMax = 1 |
| spred\_lpf\_flag | FL | cMax = 1 |
| spred\_rem\_mode\_idx | FL | cMax = 2 |
| linear\_predictive\_filtering\_data( ) | lpf\_prev\_ch\_flag  | FL | cMax = 1 |
| lpf\_delta\_coding\_flag | FL | cMax = 1 |
| lpf\_num\_weights\_idx | TR | cMax = 6, cRiceParam = 1 |
| abs\_lpf\_weight\_greater0\_flag[ ] | FL | cMax = 1 |
| abs\_lpf\_weight\_minus1[ ] | 9.3.3.7 | lpf\_prev\_ch\_flag, lpf\_delta\_coding\_flag , filter coefficient index i, number of filter weights LPFNumWeightsCurr |
| lpf\_weight\_sign\_flag[ ] | FL | cMax = 1 |
| quant\_res\_sample\_data( ) | coeff\_bypass\_value[ ] | FL | cMax = ( 1  <<  IntBitDepth ) – 1 |
| abs\_tskip\_coeff\_gt0\_flag[ ] | FL | cMax = 1 |
| abs\_tskip\_coeff\_offset[ ] | TR | cMax = NumTSkipGtxFlags, cRiceParam = 0 |
| abs\_tskip\_coeff\_rem\_prefix[ ] | TR | cMax = MaxTSkipRemPrefix, cRiceParam = 0 |
| abs\_tskip\_coeff\_rem\_fl\_suffix[ ] | FL | cMax = TSkipRiceParameter |
| abs\_tskip\_coeff\_rem\_eg0\_suffix[ ] | EG0 | - |
| tskip\_coeff\_sign\_flag[ ] | FL | cMax = 1 |
| last\_sbb\_index\_gt0\_flag | FL | cMax = 1 |
| last\_sbb\_index\_rem | LEG0 | maxPreExtLen = Log2BlockSize – Log2SbbSize – 1 |
| last\_index\_offset | FL | cMax = ( 1  <<  Log2SbbSize ) – 1 |
| abs\_trafo\_coeff\_gt0\_flag[ ] | FL | cMax = 1 |
| abs\_trafo\_coeff\_offset[ ] | TR | cMax = NumTCoeffGtxFlags, cRiceParam = 0 |
| abs\_trafo\_coeff\_remainder[ ] | EG0 | - |
| trafo\_coeff\_sign\_flag[ ] | FL | cMax = 1 |

#### Truncated Rice binarization process

Input to this process is a request for a truncated Rice (TR) binarization, cMax and cRiceParam.

Output of this process is the TR binarization associating each value symbolVal with a corresponding bin string.

A TR bin string is a concatenation of a prefix bin string and, when present, a suffix bin string.

For the derivation of the prefix bin string, the following applies:

* The prefix value of symbolVal, prefixVal, is derived as follows:

prefixVal = symbolVal  >>  cRiceParam (67)

* The prefix of the TR bin string is specified as follows:
* If prefixVal is less than cMax  >>  cRiceParam, the prefix bin string is a bit string of length prefixVal + 1 indexed by binIdx. The bins for binIdx less than prefixVal are equal to 1. The bin with binIdx equal to prefixVal is equal to 0. Table 15 illustrates the bin strings of this unary binarization for prefixVal.
* Otherwise, the bin string is a bit string of length cMax  >>  cRiceParam with all bins being equal to 1.

Table 15 – Bin string of the unary binarization (informative)

|  |  |
| --- | --- |
| **prefixVal** | **Bin string** |
| 0 | 0 |  |  |  |  |  |
| 1 | 1 | 0 |  |  |  |  |
| 2 | 1 | 1 | 0 |  |  |  |
| 3 | 1 | 1 | 1 | 0 |  |  |
| 4 | 1 | 1 | 1 | 1 | 0 |  |
| 5 | 1 | 1 | 1 | 1 | 1 | 0 |
| ... |  |  |  |  |  |  |
| binIdx | 0 | 1 | 2 | 3 | 4 | 5 |

When cMax is greater than symbolVal and cRiceParam is greater than 0, the suffix of the TR bin string is present and it is derived as follows:

* The suffix value suffixVal is derived as follows:

suffixVal = symbolVal − ( prefixVal  <<  cRiceParam ) (68)

* The suffix of the TR bin string is specified by invoking the fixed-length (FL) binarization process as specified in clause 9.3.3.6 for suffixVal with a cMax value equal to ( 1  <<  cRiceParam ) − 1.

NOTE – For the input parameter cRiceParam = 0, the TR binarization is exactly a truncated unary binarization and it is always invoked with a cMax value equal to the largest possible value of the syntax element being decoded.

#### Truncated binary (TB) binarization process

Input to this process is a request for a TB binarization for a syntax element with value synVal and cMax. Output of this process is the TB binarization of the syntax element.The bin string of the TB binarization process of a syntax element synVal is specified as follows:

n = cMax + 1
k = Floor( Log2( n ) ) (69)
u = ( 1  <<  ( k + 1 ) ) − n

* If synVal is less than u, the TB bin string is derived by invoking the FL binarization process specified in clause 9.3.3.6 for synVal with a cMax value equal to ( 1  <<  k ) − 1.
* Otherwise (synVal is greater than or equal to u), the TB bin string is derived by invoking the FL binarization process specified in clause 9.3.3.6 for ( synVal + u ) with a cMax value equal to ( 1  <<  ( k + 1 ) ) − 1.

#### k-th order Exp-Golomb binarization process

Inputs to this process is a request for a k-th order Exp-Golomb (EGk) binarization.

Output of this process is the EGk binarization associating each value symbolVal with a corresponding bin string.

The bin string of the EGk binarization process for each value symbolVal is specified as follows, where each call of the function put( X ), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

absV = Abs( symbolVal )
stopLoop = 0
do
 if( absV >= ( 1 << k ) ) {
 put( 1 )
 absV = absV − ( 1 << k )
 k++
 } else {
 put( 0 ) (70)
 while( k− − )
 put( ( absV >> k ) & 1 )
 stopLoop = 1
 }
while( !stopLoop )

NOTE – The specification for the k-th order Exp-Golomb (EGk) code uses 1s and 0s in reverse meaning for the unary part of the Exp-Golomb code of k-th order as specified in clause 9.2.

#### Limited k-th order Exp-Golomb binarization process

Inputs to this process is a request for a limited k-th order Exp-Golomb (EGk) binarization, the order k, the variables maxPreExtLen and truncSuffixLen.

Output of this process is the limited EGk binarization associating each value symbolVal with a corresponding bin string.

When truncSuffixLen is not specified as input to this process, the value of truncSuffixLen is inferred to be equal to maxPreExtLen + k.

The bin string of the limited EGk binarization process for each value symbolVal is specified as follows, where each call of the function put( X ), with X being equal to 0 or 1, adds the binary value X at the end of the bin string:

codeValue = symbolVal >> k
preExtLen = 0
while( ( preExtLen < maxPreExtLen ) && ( codeValue > ( ( 2 << preExtLen ) − 2 ) ) ) {
 preExtLen++
 put( 1 )
}
if( preExtLen = = maxPreExtLen ) (71)
 escapeLength = truncSuffixLen
else {
 escapeLength = preExtLen + k
 put( 0 )
}
symbolVal = symbolVal − ( ( ( 1 << preExtLen ) − 1 ) << k )
while( ( escapeLength− − ) > 0 )
 put( ( symbolVal >> escapeLength ) & 1 )

#### Fixed-length binarization process

Inputs to this process are a request for a fixed-length (FL) binarization and cMax.

Output of this process is the FL binarization associating each value symbolVal with a corresponding bin string.

FL binarization is constructed by using the fixedLength‑bit unsigned integer bin string of the symbol value symbolVal, where fixedLength = Ceil( Log2( cMax + 1 ) ). The indexing of bins for the FL binarization is such that the binIdx = 0 relates to the most significant bit with increasing values of binIdx towards the least significant bit.

#### Binarization process for abs\_lpf\_weight\_minus1

Input to this process is a request for the binarization of abs\_lpf\_weight\_minus1[  ], the number of filter coefficients LPFNumWeightsCurr, the filter coefficient index i and the syntax elements lpf\_prev\_ch\_flag, lpf\_delta\_coding\_flag.

Output of this process is a binarization of abs\_lpf\_weight\_minus1[ i ].

The variable shiftStartVal is derived as follows :

* If lpf\_prev\_ch\_flag is equal to 0, shiftStartVal is set equal to 6.
* Otherwise ( lpf\_prev\_ch\_flag is not equal to 0 ), shiftStartVal is set equal to 13.

The variable cMax is derived as follows:

* If lpf\_prev\_ch\_flag is not equal to 0, cMax is set equal to 1<<17.
* Otherwise ( lpf\_prev\_ch\_flag is equal to 0 ), the following applies:
	+ If lpf\_delta\_coding\_flag is equal to 0, cMax is set equal to 64.
	+ Otherwise (lpf\_delta\_coding\_flag is not equal to 0), cMax is set equal to 128.

The variable rightShift is derived as follows:

* If lpf\_prev\_ch\_flag is equal to 0, rightShift is set equal to 2.
* Otherwise ( lpf\_prev\_ch\_flag is not equal to 0), rightShift is set to 4.

The variable cRiceParam is derived as

 cRiceParam = max (2, shftStartVal - ((i + 3) >> rightShift))-1.

The binarization of abs\_lpf\_weight\_minus1[ i ] is specified by invoking the TR binarization process as specified in clause 9.3.3.2 with the variables cMax and cRiceParam as inputs.

### Decoding process flow

#### General

Inputs to this process are all bin strings of the binarization of the requested syntax element as specified in clause 9.3.3.

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 derivation process for ctxTable, ctxIdx, and bypassFlag as specified in clause 9.3.4.2 is invoked with binIdx as input and ctxTable, ctxIdx and bypassFlag as outputs.

2. The arithmetic decoding process as specified in clause 9.3.4.3 is invoked with ctxTable, ctxIdx and bypassFlag as inputs and the value of the bin as output.

#### Derivation process for ctxTable, ctxIdx and bypassFlag

##### General

Input to this process is the position of the current bin within the bin string, binIdx.

Outputs of this process are ctxTable, ctxIdx and bypassFlag.

The values of ctxTable, ctxIdx and bypassFlag are derived as follows based on the entries for binIdx of the corresponding syntax element in Table 16:

* If the entry in Table 16 is not equal to any of "bypass", "terminate" and "na", the values of binIdx are decoded by invoking the DecodeDecision process as specified in clause 9.3.4.3.2 and the following applies:
* ctxTable is specified in Table 11
* The variable ctxInc is specified by the corresponding entry in Table 16 and when more than one value is listed in Table 16 for a binIdx, the assignment process for ctxInc for that binIdx is further specified in the clauses given in parenthesis.
* The variable ctxIdxOffset set equal to the smallest value of ctxIdx is specified in Table 11 for the current value of initType and the current syntax element.
* ctxIdx is set equal to the sum of ctxInc and ctxIdxOffset.
* bypassFlag is set equal to 0.
* Otherwise, if the entry in Table 16 is equal to "bypass", the values of binIdx are decoded by invoking the DecodeBypass process as specified in clause 9.3.4.3.4 and the following applies:
* ctxTable is set equal to 0.
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 1.
* Otherwise, if the entry in Table 16 is equal to "terminate", the values of binIdx are decoded by invoking the DecodeTerminate process as specified in clause 9.3.4.3.5 and the following applies:
* ctxTable is set equal to 0.
* ctxIdx is set equal to 0.
* bypassFlag is set equal to 0.
* Otherwise (the entry in Table 16 is equal to "na"), the values of binIdx do not occur for the corresponding syntax element.

| Table 16 – Assignment of ctxInc to syntax elements with context coded bins |
| --- |
| **Syntax element** | **binIdx** |
| **0** | **1** | **2** | **3** | **4** | **>= 5** |
| end\_of\_frame\_sequence\_flag | terminate | na | na | na | na | na |
| block\_split\_log2 | min( binIdx, 8 ) |
| block\_matching\_or\_cross\_channel\_pred\_flag | 0 | na | na | na | na | na |
| cross\_channel\_pred\_flag | 0 | na | na | na | na | na |
| block\_pred\_mode | 0 | bypass | na | na | na | na |
| block\_abs\_delta\_qp  | min( binIdx, 7 ) |
| block\_delta\_qp\_sign\_flag | bypass | na | na | na | na | na |
| block\_delta\_zlsb\_present\_flag | 0 | na | na | na | na | na |
| block\_delta\_zlsb\_sign\_flag | bypass | na | na | na | na | na |
| transform\_skip\_flag | 0 | na | na | na | na | na |
| transform\_dst\_flag | bypass | na | na | na | na | na |
| end\_of\_truncated\_frame\_sequence\_flag | terminate | na | na | na | na | na |
| end\_of\_frame\_one\_bit  | terminate | na | na | na | na | na |
| cc\_pred\_offset\_only\_flag | 0 | na | na | na | na | na |
| cc\_pred\_filter\_flag | 0 | na | na | na | na | na |
| cc\_pred\_filter\_idx | 0 | na | na | na | na | na |
| cc\_pred\_mult\_hyp\_flag | 0 | na | na | na | na | na |
| cc\_pred\_abs\_chd\_greater0\_flag[ ] | 0 | na | na | na | na | na |
| cc\_pred\_abs\_chd\_minus1[ ](if the bin with binIdx is part of the prefix code of the limited EG0 binarization) | min( binIdx, 15 ) |
| cc\_pred\_abs\_chd\_minus1[ ](if the bin with binIdx is part of the fixed-length suffix code of the limited EG0 binarization) | bypass | bypass | bypass | bypass | bypass | bypass |
| cc\_pred\_chd\_sign\_flag[ ] | bypass | na | na | na | na | na |
| bm\_pred\_mult\_hyp\_flag | 0 | na | na | na | na | na |
| bm\_pred\_add\_offset\_flag | 0 | nan | nan | nan | nan | nan |
| bm\_pred\_filter\_flag[ ] | 0 | na | na | na | na | na |
| bm\_pred\_filter\_idx[ ] | 0 | na | na | na | na | na |
| bm\_pred\_off\_pred\_prev\_ch\_flag[ ] | 0 | na | na | na | na | na |
| bm\_pred\_abs\_offd\_greater0\_flag[ ] | 0 | na | na | na | na | na |
| bm\_pred\_abs\_offd\_minus1[ ](if the bin with binIdx is part of the prefix code of the limited EG0 binarization) | min( binIdx, 15 ) |
| bm\_pred\_abs\_offd\_minus1[ ](if the bin with binIdx is part of the fixed-length suffix code of the limited EG0 binarization) | bypass | bypass | bypass | bypass | bypass | bypass |
| bm\_pred\_offd\_sign\_flag[ ] | na | na | na | na | na | na |
| spred\_lpf\_or\_diff\_flag | 0 | Na | na | na | na | na |
| spred\_lpf\_flag | 0 | Na | na | na | na | na |
| spred\_rem\_mode\_idx | 0 | bypass | na | na | na | na |
| lpf\_prev\_ch\_flag  | 0 | na | na | na | na | na |
| lpf\_delta\_coding\_flag  | 0 | na | na | na | na | na |
| lpf\_num\_weights\_idx | 0 | 1 if bin at index 0 is equal to 1, otherwise bypass  | bypass | bypass | na | na |
| abs\_lpf\_weight\_greater0\_flag[ ] | 0 | na | na | na | na | na |
| abs\_lpf\_weight\_minus1[ ](if the bin with binIdx is part of the fixed length suffix code of the Truncated Rice binarization) | min( binIdx, 9 ) |
| abs\_lpf\_weight\_minus1[ ](if the bin with binIdx is part of the fixed length suffix code of the Truncated Rice binarization) | bypass | bypass | bypass | bypass | bypass | bypass |
| lpf\_weight\_sign\_flag[ ] | bypass | na | na | na | na | na |
| coeff\_bypass\_value[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| abs\_tskip\_coeff\_gt0\_flag[ ] | 0 | na | na | na | na | na |
| abs\_tskip\_coeff\_offset[ ] | Min( binIdx, NumTSkipGtxFlags ) |
| abs\_tskip\_coeff\_rem\_prefix[ ] | Min( binIdx, MaxTSkipRemPrefix ) |
| abs\_tskip\_coeff\_rem\_fl\_suffix[ ] | bypass | bypass | bypass | bypass | bypass | bypass |
| abs\_tskip\_coeff\_rem\_eg0\_suffix[ ](if the bin with binIdx is part of the unary prefix code of the EG0 binarization) | Min( binIdx, 30 ) |
| abs\_tskip\_coeff\_rem\_eg0\_suffix[ ](if the bin with binIdx is part of the fixed-length suffiox code of the EG0 binarization) | bypass | bypass | bypass | bypass | bypass | bypass |
| tskip\_coeff\_sign\_flag[ ] | bypass | na | na | na | na | na |
| last\_sbb\_index\_gt0\_flag | 0 | na | na | na | na | Na |
| last\_sbb\_index\_rem(if the bin with binIdx is part of the unary prefix code of the limited EG0 binarization) | Min( binIdx, 14 ) |
| last\_abb\_index\_rem(if the bin with binIdx is part of the fixed-length suffiox code of the limited EG0 binarization) | bypass | bypass | bypass | bypass | bypass | bypass |
| last\_index\_offset | bypass | na | na | na | na | na |
| abs\_trafo\_coeff\_gt0\_flag[ ] | 0..53(clause 9.3.4.2.2) | na | na | na | na | na |
| abs\_trafo\_coeff\_offset[ ] | 0..8(clause 9.3.4.2.3) |
| abs\_trafo\_coeff\_remainder[ ](if the bin with binIdx is part of the unary prefix code of the EG0 binarization) | Min( binIdx, 30 ) |
| abs\_trafo\_coeff\_remainder[ ](if the bin with binIdx is part of the fixed-length suffiox code of the EG0 binarization) | bypass | bypass | bypass | bypass | bypass | bypass |
| trafo\_coeff\_sign\_flag[ ] | bypass | na | na | na | na | na |

##### Derivation process of ctxInc for the syntax element abs\_trafo\_coeff\_gt0\_flag

Inputs to this process are the variables is the location k of the current quantization index.

Output of this process is the variable ctxInc.

The variable templateClass is derived as specified by the following pseudo-code:

templateSum = 0 (72)
for( n = k + 1; n < Min( NumQuantIndices – 1, k + 4 ; n = n + 1 )
 templateSum += Abs( QuantIndices[ n ] )
templateClass = Min( 2, templateSum )

The variable posClass is derived as specified by the following pseudo-code:

posThresholds[ ] = { 0, 1, 2, 3, 7, 11, 15, 23 } (73)
posClass = 0
for( n = 0; n < 8 ; n = n + 1 )
 if( k > posThresholds[ n ] )
 posClass += 1

The variable ctxInc is derived by

ctxInc = 27 \* ( QState & 1 ) + 3 \* posClass + templateClass (74)

##### Derivation process of ctxInc for the syntax element abs\_trafo\_coeff\_offset

Inputs to this process are the variables is the location k of the current quantization index.

Output of this process is the variable ctxInc.

The variable posClass is derived as specified by the following pseudo-code:

posThresholds[ ] = { 0, 1, 2, 3, 7, 11, 15, 23 } (75)
posClass = 0
for( n = 0; n < 8 ; n = n + 1 )
 if( k > posThresholds[ n ] )
 posClass += 1

The variable ctxInc is set equal to posClass.

#### Arithmetic decoding process

##### General

Inputs to this process are ctxTable, ctxIdx, and bypassFlag, as derived in clause 9.3.4.2, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

Output of this process is the value of the bin.

Figure 1 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table ctxTable, the ctxIdx and the bypassFlag are passed to the arithmetic decoding process DecodeBin( ctxTable, ctxIdx, bypassFlag ), which is specified as follows:

– If bypassFlag is equal to 1, DecodeBypass( ) as specified in clause 9.3.4.3.4 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 9.3.4.3.5 is invoked.

– Otherwise (bypassFlag is equal to 0 and ctxTable is not equal to 0), DecodeDecision( ctxTable, ctxIdx ) as specified in clause 9.3.4.3.2 is invoked.



Figure 1 – Flowchart 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 would 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 would be chosen as the new code interval, and a binary code string pointing into that interval would 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 the 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 the following two distinct properties:

– The range ivlCurrRange representing the state of the coding engine and the probability estimate are quantized to reduced-precision values to allow for a reduced-precision multiplication to determine the product ivlCurrRange and the probability estimate.

– 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.

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 − 256 ) >> 5 (76)

– Given qRangeIdx, pStateIdx0 and pStateIdx1 associated with ctxTable and ctxIdx, valMps and ivlLpsRange are derived as follows:

rangeTabLps[ ][ ] =
{
 { 128, 142, 156, 171, 185, 199, 213, 228 }, { 112, 125, 137, 150, 162, 175, 187, 200 },
 { 97, 108, 119, 130, 141, 152, 163, 174 }, { 84, 93, 103, 112, 121, 131, 140, 150 },
 { 74, 82, 90, 99, 107, 115, 123, 132 }, { 65, 72, 79, 87, 94, 101, 108, 116 },
 { 57, 63, 70, 76, 82, 89, 95, 102 }, { 50, 56, 61, 67, 73, 78, 84, 90 },
 { 45, 50, 55, 60, 65, 70, 75, 80 }, { 39, 43, 48, 52, 56, 61, 65, 70 },
 { 34, 38, 42, 46, 50, 54, 58, 62 }, { 30, 33, 37, 40, 43, 47, 50, 54 },
 { 27, 30, 33, 36, 39, 42, 45, 48 }, { 23, 26, 28, 31, 34, 36, 39, 42 },
 { 20, 22, 24, 27, 29, 31, 33, 36 }, { 18, 20, 22, 24, 26, 28, 30, 32 },
 { 15, 17, 19, 21, 22, 24, 26, 28 }, { 14, 16, 17, 19, 21, 22, 24, 26 },
 { 12, 13, 15, 16, 17, 19, 20, 22 }, { 11, 12, 13, 15, 16, 17, 18, 20 },
 { 10, 11, 12, 13, 14, 15, 16, 18 }, { 9, 10, 11, 12, 13, 14, 15, 16 },
 { 7, 8, 9, 10, 11, 12, 13, 14 }, { 7, 8, 9, 10, 11, 12, 13, 14 },
 { 5, 6, 6, 7, 8, 8, 9, 10 }, { 5, 6, 6, 7, 8, 8, 9, 10 },
 { 4, 5, 5, 6, 6, 7, 7, 8 }, { 4, 5, 5, 6, 6, 7, 7, 8 },
 { 3, 3, 4, 4, 4, 5, 5, 6 }, { 3, 3, 4, 4, 4, 5, 5, 6 },
 { 2, 2, 2, 3, 3, 3, 3, 4 }, { 2, 2, 2, 3, 3, 3, 3, 4 }
}
pState = ( pStateIdx1 + 8 \* pStateIdx0 ) >> 8
valMps = pState >= 0 ? 1 : 0
ivlLpsRange = rangeTabLps[ Abs( pState ) ][ qRangeIdx ] (77)

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 9.3.4.3.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in clause 9.3.4.3.3.

###### State transition process

Inputs to this process are the current pBinCount, pStateIdx0, and pStateIdx1, and the decoded value binVal.

Outputs of this process are the updated pBinCount , pStateIdx0, and pStateIdx1 of the context variable associated with ctxTable and ctxIdx.

Depending on the decoded value binVal, the update of the three variables pBinCount, pStateIdx0, and pStateIdx1 associated with ctxTable and ctxIdx is derived as follows:

transitionTable[ ] =
{
 157, 143, 129, 115, 101, 87, 73, 59, 45, 35, 29, 23, 17, 13, 9, 5,
 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0
}
signVal = 2 \* binVal − 1
if( pBinCount > 0) {
 pStateIdx0 += signVal \* ( transitionTable[ 16 + ( ( signVal \* pStateIdx0 ) >> 5 ) ] << 1 )
 pStateIdx1 = pStateIdx0 << 3
 pBinCount− −
} else {

 pStateIdx0 += signVal \* transitionTable[ 16 + ( ( signVal \* pStateIdx0 ) >> 5 ) ]
 pStateIdx1 += signVal \* transitionTable[ 16 + ( ( signVal \* pStateIdx1 ) >> 8 ) ]
}

##### Renormalization process in the arithmetic decoding engine

Inputs to this process are bits from slice 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 2. The current value of ivlCurrRange is first compared to 256 and further steps are specified as follows:

– 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 2 – Flowchart of renormalization

##### Bypass decoding process for binary decisions

Inputs to this process are bits from slice 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 3 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 further steps are specified as follows:

– 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 3 – Flowchart of bypass decoding process

##### Decoding process for binary decisions before termination

Inputs to this process are bits from slice 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\_frame\_sequence\_flag, end\_of\_truncated\_frame\_sequence\_flag, and end\_of\_frame\_one\_bit corresponding to ctxTable equal to 0 and ctxIdx equal to 0. Figure 4 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 further steps are specified as follows:

– 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\_slice\_one\_bit, this last bit inserted in register ivlOffset is interpreted as rbsp\_stop\_one\_bit. When decoding end\_of\_tile\_one\_bit or end\_of\_subset\_one\_bit, this last bit inserted in register ivlOffset is interpreted as byte\_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 9.3.4.3.3.

NOTE – This procedure could 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, 7 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 4 – Flowchart of decoding a decision before termination

1. Annex A

Placeholder

Placeholder

1. Annex B

Byte stream format
	1. General

This annex specifies syntax and semantics of a byte stream format specified for use by applications that deliver some or all of the NAL unit stream as an ordered stream of bytes or bits within which the locations of NAL unit boundaries need to be identifiable from patterns in the data, such as Rec. ITU-T H.222.0 | ISO/IEC 13818-1 systems or Rec. ITU‑T H.320 systems. For bit-oriented delivery, the bit order for the byte stream format is specified to start with the MSB of the first byte, proceed to the LSB of the first byte, followed by the MSB of the second byte, etc.

The byte stream format consists of a sequence of byte stream NAL unit syntax structures. Each byte stream NAL unit syntax structure contains one start code prefix followed by one nal\_unit( NumBytesInNalUnit ) syntax structure. It could (and under some circumstances, it shall) also contain an additional zero\_byte syntax element. It could also contain one or more additional trailing\_zero\_8bits syntax elements. When it is the first byte stream NAL unit in the bitstream, it could also contain one or more additional leading\_zero\_8bits syntax elements.

* 1. **Byte stream NAL unit syntax and semantics**
		1. **Byte stream NAL unit syntax**

|  |  |
| --- | --- |
| byte\_stream\_nal\_unit( NumBytesInNalUnit ) { | **Descriptor** |
| while( next\_bits( 24 ) != 0x000001 && next\_bits( 32 ) != 0x00000001 ) |  |
|  **leading\_zero\_8bits** /\* equal to 0x00 \*/ | f(8) |
|  if( next\_bits( 24 ) != 0x000001 ) |  |
|  **zero\_byte** /\* equal to 0x00 \*/ | f(8) |
|  **start\_code\_prefix\_one\_3bytes** /\* equal to 0x000001 \*/ | f(24) |
| nal\_unit( NumBytesInNalUnit ) |  |
|  while( more\_data\_in\_byte\_stream( ) && next\_bits( 24 ) != 0x000001 && next\_bits( 32 ) != 0x00000001 ) |  |
|  **trailing\_zero\_8bits** /\* equal to 0x00 \*/ | f(8) |
| } |  |

* + 1. **Byte stream NAL unit semantics**

The order of byte stream NAL units in the byte stream shall follow the decoding order of the NAL units contained in the byte stream NAL units.

**leading\_zero\_8bits** is a byte equal to 0x00.

NOTE – The leading\_zero\_8bits syntax element could only be present in the first byte stream NAL unit of the bitstream, because (as shown in the syntax diagram of clause B.2.1) any bytes equal to 0x00 that follow a NAL unit syntax structure and precede the four-byte sequence 0x00000001 (which is to be interpreted as a zero\_byte followed by a start\_code\_prefix\_one\_3bytes) would be considered to be trailing\_zero\_8bits syntax elements that are part of the preceding byte stream NAL unit.

**zero\_byte** is a single byte equal to 0x00.

**start\_code\_prefix\_one\_3bytes** is a fixed-value sequence of 3 bytes equal to 0x000001. This syntax element is called a start code prefix.

**trailing\_zero\_8bits** is a byte equal to 0x00.

* 1. **Byte stream NAL unit decoding process**

Input to this process consists of an ordered stream of bytes consisting of a sequence of byte stream NAL unit syntax structures.

Output of this process consists of a sequence of NAL unit syntax structures.

At the beginning of the decoding process, the decoder initializes its current position in the byte stream to the beginning of the byte stream. It then extracts and discards each leading\_zero\_8bits syntax element (when present), moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next four bytes in the bitstream form the four-byte sequence 0x00000001.

The decoder then performs the following step-wise process repeatedly to extract and decode each NAL unit syntax structure in the byte stream until the end of the byte stream has been encountered (as determined by unspecified external means) and the last NAL unit in the byte stream has been decoded:

1. When the next four bytes in the bitstream form the four-byte sequence 0x00000001, the next byte in the byte stream (which is a zero\_byte syntax element) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this discarded byte.
2. The next three-byte sequence in the byte stream (which is a start\_code\_prefix\_one\_3bytes) is extracted and discarded and the current position in the byte stream is set equal to the position of the byte following this three‑byte sequence.
3. NumBytesInNalUnit is set equal to the number of bytes starting with the byte at the current position in the byte stream up to and including the last byte that precedes the location of one or more of the following conditions:

– A subsequent byte-aligned three-byte sequence equal to 0x000000,

– A subsequent byte-aligned three-byte sequence equal to 0x000001,

– The end of the byte stream, as determined by unspecified external means.

1. NumBytesInNalUnit bytes are removed from the bitstream and the current position in the byte stream is advanced by NumBytesInNalUnit bytes. This sequence of bytes is nal\_unit( NumBytesInNalUnit ) and is decoded using the NAL unit decoding process.
2. When the current position in the byte stream is not at the end of the byte stream (as determined by unspecified external means) and the next bytes in the byte stream do not start with a three-byte sequence equal to 0x000001 and the next bytes in the byte stream do not start with a four byte sequence equal to 0x00000001, the decoder extracts and discards each trailing\_zero\_8bits syntax element, moving the current position in the byte stream forward one byte at a time, until the current position in the byte stream is such that the next bytes in the byte stream form the four-byte sequence 0x00000001 or the end of the byte stream has been encountered (as determined by unspecified external means).
	1. **Decoder byte-alignment recovery (informative)**

This clause does not form an integral part of this Specification.

Many applications provide data to a decoder in a manner that is inherently byte aligned, and thus have no need for the bit-oriented byte alignment detection procedure described in this clause.

A decoder is said to have byte alignment with a bitstream when the decoder has determined whether or not the positions of data in the bitstream are byte-aligned. When a decoder does not have byte alignment with the bitstream, the decoder may examine the incoming bitstream for the binary pattern '00000000 00000000 00000000 00000001' (31 consecutive bits equal to 0 followed by a bit equal to 1). The bit immediately following this pattern is the first bit of an aligned byte following a start code prefix. Upon detecting this pattern, the decoder will be byte-aligned with the bitstream and positioned at the start of a NAL unit in the bitstream.

Once byte aligned with the bitstream, the decoder can examine the incoming bitstream data for subsequent three-byte sequences 0x000001 and 0x000003.

When the three-byte sequence 0x000001 is detected, this is a start code prefix.

When the three-byte sequence 0x000003 is detected, the third byte (0x03) is an emulation\_prevention\_three\_byte to be discarded as specified in clause 7.4.2.

When an error in the bitstream syntax is detected (e.g., a non-zero value of the forbidden\_zero\_bit or one of the three‑byte or four-byte sequences that are prohibited in clause 7.4.2), the decoder may consider the detected condition as an indication that byte alignment may have been lost and may discard all bitstream data until the detection of byte alignment at a later position in the bitstream in the manner described in this clause.

**Bibliography**