

Low Delay Picture Coding

Description

5 The present invention is concerned with low delay coding of pictures

In the current HEVC design Slices, Entropy Slices (former Light Weight Slices) Tiles and WPP (Wavefront Parallel Processing) are contained as tools for parallelization.

10 For parallelization of video encoders and decoders picture-level partitioning has several advantages compared to other approaches. In previous video codecs, like H.264/AVC [1], picture partitions were only possible with regular slices with a high cost in terms of coding efficiency. For scalable parallel H.264/AVC decoding it is necessary to combine macroblock-level parallelism for picture reconstruction and frame-level parallelism for
 15 entropy decoding. This approach, however, provides limited reduction in picture latencies and high memory usage. In order to overcome these limitations, new picture partition strategies have been included in the HEVC codec. Current reference software version (HM-6) contains 4 different approaches: regular or normal slices, entropy slices, wavefront parallel processing (WPP) sub-streams and tiles. Typically those picture partitions
 20 comprise a set of Largest Coding Units (LCUs), or, in a synonymous wording, Coding Tree Units (CTU), as defined in HEVC or even a subset of those.

Figure 1 shows as a picture 898 exemplarily positioned into regular slice 900 per row 902 of LCUs or macroblocks in a picture. Regular or normal slices (as defined in H.264 [1])
 25 have the largest coding penalty as they break entropy decoding and prediction dependencies.

Entropy slices, like slices, break entropy decoding dependencies but allow prediction (and filtering) to cross slice boundaries.

30 In WPP the picture partitions are row interleaved, and both entropy decoding and prediction are allowed to use data from blocks in other partitions. In this way coding losses are minimized while at the same time wavefront parallelism can be exploited. The interleaving, however, violates bitstream causality as a prior partition needs a next partition to decode.

35

Figure 2 exemplarily shows a picture 898 divided up into two rows 904, 904b of horizontally partitioning tiles 906. Tiles define horizontal 908 and vertical boundaries 910 that partition a picture 898 into tile columns 912a,b,c and rows 904a,b. Similar to regular

slices 900, tiles 906 break entropy decoding and prediction dependencies, but does not require a header for each tile.

For each of these techniques the number of partitions can be freely chosen by the encoder.

- 5 In general having more partitions leads to higher compression losses. However in WPP the loss propagation is not so high and therefore the number of picture partitions even can be fixed to one per row-. This leads also to several advantages. First, for WPP bitstream causality is guaranteed. Second, decoder implementations can assume that a certain amount of parallelism is available, which also increases with the resolution. And, finally,
10 none of the context selection and prediction dependencies have to be broken when decoding in wavefront order, resulting in relative low coding losses.

However, until now all parallel coding in transform concepts fail to provide an achievement of high compression efficiency in combination with keeping the delay low.

- 15 This is also true for the WPP concept. The slices are the smallest units of transportation, in the coding pipeline, and several WPP substreams still have to be transported serially.

Accordingly, it is the object of the present invention to provide a picture coding concept which allows for parallel decoding according to, for example, wavefront parallel
20 processing, with increased efficiency, such as with even further reducing the end-to-end delay or enhancing coding efficiency by reducing coding overhead to be spent.

This object is achieved by the subject matter of the independent claims.

- 25 It is a basic finding of the present invention that parallel processing concepts such as wavefront parallel processing, may be realized with a reduced end-to-end delay if the usual slice concept according to which slices are either coded/decoded completely independent from areas of the picture outside of the respective slice, or at least independent from areas outside the respective slice as far as the entropy coding is concerned is given up in favor
30 of slices of different modes, namely ones called dependent slices which allow for interdependencies across slice boundaries, and others which do not, called normal slices, for example.

- It is a further basic finding of the present invention which may be combined with the first
35 one or individually be used, that WPP processing concept may be made more efficiently if the slices' start syntax portions are used to locate WPP entry points.

Preferred embodiments of the present application are described below with regard to the figures, wherein advantageous embodiments are the subject of dependent claims. Among the figures,

- 5 Fig. 1 shows a picture exemplarily partitioned into a regular slice per row of LCUs or macroblocks in a picture;
- Fig. 2 shows a picture exemplarily divided up into two rows of horizontally partitioned tiles;
- 10 Fig. 3 exemplarily shows an assignment of parallel encoded partitions to a slice or network transport segment;
- Fig. 4 shows a schematic diagram illustrating a generic fragmentation of a frame with a tile coding approach for minimum end-to-end delay;
- 15 Fig. 5 shows a schematic diagram illustrating an exemplary fragmentation of a frame with WPP coding approach for minimum end-to-end delay;
- 20 Fig. 6 shows a schematic block diagram illustrating a scenario of a conversation using video services;
- Fig. 7 illustrates schematically a possible time scheduling of encoding, transmission and decoding for tiles with generic subsets with minimum end-to-end delay;
- 25 Fig. 8 schematically shows a timing schedule which commonly achieves an end-to-end delay;
- 30 Fig. 9 illustrates a picture exemplarily having 11x9 coding treeblocks, that is partitioned into two slices;
- Fig. 10 illustrates a picture exemplarily having 13x8 coding treeblocks, that is partitioned into three tiles;
- 35 Fig. 11 shows an example for a sequence parameter set syntax;
- Fig. 12 shows an example for a picture parameter set syntax;

	Fig. 13	shows an example for a slice header syntax;
5	Fig. 14	exemplifies a partition of a picture for WPP processing into a regular slice and, for a low delay processing, into dependent slices;
	Fig. 15	shows an example for a portion within a picture parameter set syntax;
10	Fig. 16	shows a possible slice-header syntax;
	Fig. 17	schematically illustrates coding interdependencies for normal slices (and dependent slices);
15	Fig. 18	shows a schematic diagram comparing an encoding for low delay transport of tiles (wavefront parallel processing using dependent slices);
20	Fig. 19	illustrates a timing schedule illustrating an exemplary WPP coding with pipeline low delay transmission when using wavefront parallel processing using dependent slices as shown at the right-hand side of Fig. 18;
	Fig. 20	shows a schematic diagram illustrating a robustness improvement by using regular slices as anchors;
25	Fig. 21	shows another embodiment for a slice header syntax;
	Fig. 22	shows another embodiment for a picture parameter set syntax;
30	Fig. 23	shows a schematic diagram illustrating a symbol probability initialization process for a dependent slice in case of beginning at the left picture boundary;
	Fig. 24	shows a schematic diagram of a decoder;
35	Fig. 25	schematically shows a block diagram of a decoder along with schematically illustrating the partitioning of a picture into coding blocks and slices;
	Fig. 26	schematically shows a block diagram of an encoder;

Fig. 27 schematically shows a picture partitioned into normal and dependent slices, here called slice segments;

Fig. 28a and 28b schematically shows a picture partitioned into normal and dependent slices here called slice segments, on the one hand, and tiles on the other hand;

Fig. 29 shows a flow diagram illustrating an context initialization process using dependent slices;

Fig. 30 shows a flow diagram illustrating a context storage process for using dependent slices; and

Fig. 31 schematically shows different possibilities of signaling WPP entry points.

In the following, the description starts with a description of today's concepts for enabling parallel picture processing and low delay coding, respectively. The problems occurring when wishing to have both abilities are outlined. In particular, as will turn out from the following discussion, the WPP substream concept as taught so far somehow conflicts with the wish to have a low delay due to the necessity to convey WPP substreams by grouping same into one slice. The following embodiments render parallel processing concepts such as the WPP concept, applicable to applications necessitating even less delay by broadening the slice concept, namely by introducing another type of slice, later on called dependent slices.

Minimization of the end to end video delay from capturing to display is one of the main aims in applications such as video conferencing and the like.

The signal processing chain for digital video transmission consists of camera, capturing device, encoder, encapsulation, transmission, demultiplexer, decoder, renderer and display. Each of these stages contributes to the end to end delay by buffering image data before its serial transmission to the subsequent stage.

Some applications require minimization of such delay, e.g. remote handling of objects in hazardous areas, without direct sight to the handled object, or minimal invasive surgery.

Even a short delay can result in severe difficulties of proper handling or even lead to catastrophic mistakes.

In many cases, a whole video frame is buffered within a processing stage, e.g. to allow for intra frame processing. Some stages gather data in order to form packets which are forwarded to the next stage. In general, there is a lower boundary for the delay which results from the requirements of the local processing. This is analyzed for each individual stage in more detail below.

The processing inside the camera does not necessarily require intra-frame signal processing, so the minimum delay is given by the integration time of the sensor, which is bounded by the frame rate, and some design choices by the hardware manufacturer. The camera output is typically related to the scan order which usually starts processing in the top left corner, moves over to the top right corner and continues line by line to the bottom right corner. Consequently, it takes about one frame duration until all data is transferred from the sensor to the camera output.

The capturing device could forward the camera data immediately after reception; however it will typically buffer some data and generate bursts in order to optimize data access to memory or storage. Furthermore, the connection between camera/capturer and memory of the computer is typically limiting the bitrate for forwarding the captured image data to the memory for further processing (encoding). Typically, cameras are connected via USB 2.0 or soon with USB 3.0, which will always include a partial transport of the image data to the encoder. This limits the parallelizability on the encoder side in extreme low-delay scenarios, i.e. the encoder will try to start encoding as soon as possible, when data becomes available from the camera, e.g. in a raster-scan order from top to bottom of the image.

In the encoder, there are some degrees of freedom which allow to trade-off encoding efficiency, in terms of data rate needed for a certain video fidelity, for a reduction of the processing delay.

The encoder uses data which has already been sent to predict the image to be subsequently encoded. In general, the difference between the actual image and the prediction can be encoded with fewer bits than would be needed without prediction. This prediction values need to be available at the decoder, thus prediction is based on previously decoded portions of the same image (intra-frame prediction) or on other images (inter-frame prediction) which have been processed earlier. Pre-HEVC video encoding standards use only the part

of the image above or in the same line, but left — which has been previously encoded — for intra-frame prediction, motion vector prediction and entropy coding (CABAC).

In addition to the optimization of the prediction structure, the influence of parallel processing can be considered. Parallel processing requires the identification of picture areas which can be processed independently. For practical reasons, contiguous regions such as horizontal or vertical rectangles are chosen which are often called “tiles”. In the case of low delay constraints, those regions should allow for parallelized coding of the data incoming from the capturer to the memory, as soon as possible. Assuming a raster-scan memory transfer, vertical partitions of the raw data make sense, in order to start encoding immediately. Inside such tiles, which divides the picture into vertical partitions (cf. figure below), intra-prediction, motion vector prediction and entropy coding (CABAC) can lead to reasonable coding efficiency. In order to minimize the delay, only part of the picture, starting from the top, would be transferred to the frame memory of the encoder, and parallel processing should be started in vertical tiles.

Another way of allowing parallel processing is to use WPP within a regular slice, that would be compared to tiles, a “row” of tiles included in a single slice. The data within that slice, could be also parallel encoded using WPP substreams , within the slice. The picture separation into slices 900 and tiles/WPP substreams 914 is shown in Fig. 3/1 form of examples.

Fig. 3, thus, shows the assignment of parallel encoded partitions such as 906 or 914 to a slice or network transport segment (a single network packet or multiple network 900 packets).

The encapsulation of the encoded data into Network Abstraction Layer (NAL) units, as defined in H.264 or HEVC, before transmission or during the encoding process adds some header to data blocks which allows for identification of each block and the reordering of blocks, if applicable. In the standard case, no additional signaling is required, since the order of coding elements is always in decoding order, that is an implicit assignment of the position of the tile or general coding fragment is given.

If parallel processing is considered with an additional transport layer for low delay parallel transport, i.e. the transport layer may re-order picture partitions for tiles in order to allow for low delay transmission, meaning to send out fragments as shown in Fig. 4 as they are encoded. Those fragments may be also not fully encoded slices, they may be a subset of a slice, , or may be contained in a dependent Slice.

In case of creating additional fragments, there is a trade-off between efficiency, which would be highest with large data blocks because the header information is adding a constant number of bytes, and delay, because large data blocks of the parallel encoders would need to be buffered before transmission. The overall delay can be reduced, if the encoded representation of vertical tiles 906 is separated in a number of fragments 916 which are transmitted as soon as a fragment is completely encoded. The size of each fragment can be determined in terms of a fixed image region, such as macroblocks, LCUs or in terms of a maximum data as shown in Fig. 4.

Fig. 4, thus, shows a generic fragmentation of a frame with tile coding approach for minimum end-to-end delay.

Similarly, Fig. 5 shows a fragmentation of a frame with WPP coding approach for minimum end-to-end delay.

The transmission may add further delay, e.g., if additional block oriented processing is applied, such as Forward Error Correction codes increasing the robustness of the transmission. Besides, the network infrastructure (routers etc.) or the physical link can add delay, this is typically known as latency for a connection. In addition to the latency the transmission bitrate determines, the time (delay) for transferring the data from Party a to Party b, in a conversation as shown in Fig. 6 which uses video services.

If encoded data blocks are transmitted out of order, reordering delay has to be considered.

The decoding can start as soon as a data unit has arrived, assumed that other data units which have to be decoded prior to this are available.

In case of tiles, there are no dependencies between tiles, thus a tile can be decoded immediately. If fragments have been produced of a tile, such as separate slices per each fragment as shown in Fig 4, the fragments can be directly transported, as soon as they are encoded respectively their contained LCUs or CUs have been encoded.

The renderer assembles the outputs of parallel decoding engines and forwards the combined picture line by line to the display.

The display does not necessarily add any delay, but in practice may do some intra frame processing before the image data is actually displayed. This is up to design choices by the hardware manufacturer.

Summing up, we can influence the stages encoding, encapsulation, transmission and decoding in order to achieve minimum end-to-end delay. If we use parallel processing, tiles and fragmentation within the tiles, the total delay can be reduced significantly as shown in Fig. 7, compared to a commonly used processing chain that adds about one frame delay at each of these stages as shown in Fig. 8.

In particular, while Fig. 7 shows encoding, transmission and decoding for tiles with generic subsets with minimum end-to-end delay, Fig. 8 illustrates a commonly achieved end-to-end delay.

HEVC allows the use of slice partitioning, tile partitioning, and in the following way.

tile: An integer number of treeblocks co-occurring in one column and one row, ordered consecutively in treeblock raster scan of the tile. The division of each picture into tiles is a partitioning. Tiles in a picture are ordered consecutively in tile raster scan of the picture. Although a slice contains treeblocks that are consecutive in treeblock raster scan of a tile, these treeblocks are not necessarily consecutive in treeblock raster scan of the picture.

slice: An integer number of treeblocks ordered consecutively in the raster scan. The division of each picture into slices is a partitioning. The treeblock addresses are derived from the first treeblock address in a slice (as represented in the slice header).

raster scan: A mapping of a rectangular two-dimensional pattern to a one-dimensional pattern such that the first entries in the one-dimensional pattern are from the first top row of the two-dimensional pattern scanned from left to right, followed similarly by the second, third, etc., rows of the pattern (going down) each scanned from left to right.

treeblock: A $N \times N$ block of luma samples and two corresponding blocks of chroma samples of a picture that has three sample arrays, or a $N \times N$ block of samples of a monochrome picture or a picture that is coded using three separate colour planes. The division of a slice into treeblocks is a partitioning.

partitioning: The division of a set into subsets such that each element of the set is in exactly one of the subsets.

quadtree: A tree in which a parent node can be split into four child nodes. A child node may become parent node for another split into four child nodes.

In the following, the spatial subdivision of pictures, slices and tiles is explained. In particular, the following description specifies how a picture is partitioned into slices, tiles and coding treeblocks. Pictures are divided into slices and tiles. A slice is a sequence of coding treeblocks. Likewise, a tile is a sequence of coding treeblocks.

The samples are processed in units of coding treeblocks. The luma array size for each treeblock in samples in both width and height is CtbSize. The width and height of the chroma arrays for each coding treeblock are CtbWidthC and CtbHeightC, respectively.

For example, a picture may be divided into two slices as shown in the next figure. As another example, a picture may be divided into three tiles as shown in the second following figure.

Unlike slices, tiles are always rectangular and always contain an integer number of coding treeblocks in coding treeblock raster scan. A tile may consist of coding treeblocks contained in more than one slice. Similarly, a slice may comprise coding treeblocks contained in more than one tile.

Fig. 9 illustrates a picture 898 with 11 by 9 coding treeblocks 918 that is partitioned into two slices 900a,b.

Fig. 10 illustrates a picture with 13 by 8 coding treeblocks 918 that is partitioned into three tiles.

Each coding 898 treeblock 918 is assigned a partition signaling to identify the block sizes for intra or inter prediction and for transform coding. The partitioning is a recursive quadtree partitioning. The root of the quadtree is associated with the coding treeblock. The quadtree is split until a leaf is reached, which is referred to as the coding block. The coding block is the root node of two trees, the prediction tree and the transform tree.

The prediction tree specifies the position and size of prediction blocks. The prediction blocks and associated prediction data are referred to as prediction unit.

Fig. 11 shows an exemplary sequence parameter set RBSP syntax.

The transform tree specifies the position and size of transform blocks. The transform blocks and associated transform data are referred to as transform unit.

The splitting information for luma and chroma is identical for the prediction tree and may or may not be identical for the transform tree.

The coding block, the associated coding data and the associated prediction and transform units form together a coding unit.

A process for a conversion of a coding treeblock address in coding treeblock raster order to tile scan order could be as follows:

Outputs of this process are

- 10 – an array CtbAddrTS[ctbAddrRS], with ctbAddrRS in the range of 0 to PicHeightInCtbs * PicWidthInCtbs – 1, inclusive.
- an array TileId[ctbAddrTS], with ctbAddrTS in the range of 0 to PicHeightInCtbs * PicWidthInCtbs – 1, inclusive.

The array CtbAddrTS[] is derived as follows:

```

15 for( ctbAddrRS = 0; ctbAddrRS < PicHeightInCtbs * PicWidthInCtbs; ctbAddrRS++ ) {
    tbX = ctbAddrRS % PicWidthInCtbs
    tbY = ctbAddrRS / PicWidthInCtbs
    for( j = 0; j <= num_tile_columns_minus1; j++ )
        if( tbX < ColBd[ j + 1 ] )
            tileX = j
    for( i = 0; i <= num_tile_rows_minus1; i++ )
        if( tbY < RowBd[ i + 1 ] )
            tileY = i
    CtbAddrTS[ ctbAddrRS ] = ctbAddrRS - tbX
    for( i = 0; i < tileX; i++ )
        ctbAddrTS += RowHeight[ tileY ] * ColumnWidth[ i ]
    CtbAddrTS[ ctbAddrRS ] += ( tbY - RowBd[ tileY ] ) * ColumnWidth[ tileY ] + tbX - ColBd[ tileX ]
}

```

The array TileId[] is derived as follows:

```

    for( j = 0, tileId = 0; j <= num_tile_columns_minus1; j++ )
        for( i = 0; i <= num_tile_rows_minus1; i++, tileId++ )
            for( y = RowBd[ j ]; y < RowBd[ j + 1 ]; y++ )
                for( x = ColBd[ i ]; x < ColBd[ i + 1 ]; x++ )
                    TileId[ CtbAddrTS[ y*PicWidthInCtbs + x ] ] = tileId

```

A corresponding, exemplary syntax is shown in Figs. 11, 12 and 13, wherein Fig. 12 has an exemplary picture parameter set RBSP syntax. Fig. 13 shows an exemplary slice header syntax.

In the syntax example, the following semantics may apply:

entropy_slice_flag equal to 1 specifies that the value of slice header syntax elements not present is inferred to be equal to the value of slice header syntax elements in a proceeding slice, where a proceeding slice is defined as the slice containing the coding treeblock with location (SliceCtbAddrRS – 1). entropy_slice_flag shall be equal to 0 when SliceCtbAddrRS equal to 0.

tiles_or_entropy_coding_sync_idc equal to 0 specifies that there is only one tile in each picture in the coded video sequence, and no specific synchronization process for context variables is invoked before decoding the first coding treeblock of a row of coding treeblocks.

tiles_or_entropy_coding_sync_idc equal to 1 specifies that there may be more than one tile in each picture in the coded video sequence, and no specific synchronization process for context variables is invoked before decoding the first coding treeblock of a row of coding treeblocks.

tiles_or_entropy_coding_sync_idc equal to 2 specifies that there is only one tile in each picture in the coded video sequence, a specific synchronization process for context variables is invoked before decoding the first coding treeblock of a row of coding treeblocks, and a specific memorization process for context variables is invoked after decoding two coding treeblocks of a row of coding treeblocks.

The value of tiles_or_entropy_coding_sync_idc shall be in the range of 0 to 2, inclusive.

num_tile_columns_minus1 plus 1 specifies the number of tile columns partitioning the picture.

num_tile_rows_minus1 plus 1 specifies the number of tile rows partitioning the picture.

When num_tile_columns_minus1 is equal to 0, num_tile_rows_minus1 shall not be equal to 0.

One or both of the following conditions shall be fulfilled for each slice and tile:

- All coded blocks in a slice belong to the same tile.
- All coded blocks in a tile belong to the same slice.

NOTE – Within the same picture, there may be both slices that contain multiple tiles and tiles that contain multiple slices.

uniform_spacing_flag equal to 1 specifies that column boundaries and likewise row boundaries are distributed uniformly across the picture. uniform_spacing_flag equal to 0 specifies that column boundaries and likewise row boundaries are not distributed uniformly across the picture but signalled explicitly using the syntax elements column_width[i] and row_height[i].

column_width[i] specifies the width of the i-th tile column in units of coding treeblocks.

- 10 row_height[i] specifies the height of the i-th tile row in units of coding treeblocks. Values of ColumnWidth[i], specifying the width of the i-th tile column in units of coding treeblocks, and the values of ColumnWidthInLumaSamples[i], specifying the width of the i-th tile column in units of luma samples, are derived as follows:

```

15 for( i = 0; i <= num_tile_columns_minus1; i++ ) {
    if( uniform_spacing_flag )
        ColumnWidth[ i ] = ( ( i + 1 ) * PicWidthInCtbs ) / ( num_tile_columns_minus1 + 1 ) -
                           ( i * PicWidthInCtbs ) / ( num_tile_columns_minus1 + 1 )
    else
        ColumnWidth[ i ] = column_width[ i ]
        ColumnWidthInLumaSamples[ i ] = ColumnWidth[ i ] << Log2CtbSize
    }

```

Values of RowHeight[i], specifying the height of the i-th tile row in units of coding treeblocks, are derived as follows:

```

20 for( i = 0; i <= num_tile_rows_minus1; i++ )
    if( uniform_spacing_flag )
        RowHeight[ i ] = ( ( i + 1 ) * PicHeightInCtbs ) / ( num_tile_rows_minus1 + 1 ) -
                        ( i * PicHeightInCtbs ) / ( num_tile_rows_minus1 + 1 )
    else
        RowHeight[ i ] = row_height[ i ]

```

- 25 Values of ColBd[i], specifying the location of the left column boundary of the i-th tile column in units of coding treeblocks, are derived as follows:

```

for( ColBd[ 0 ] = 0, i = 0; i <= num_tile_columns_minus1; i++ )
    ColBd[ i + 1 ] = ColBd[ i ] + ColumnWidth[ i ]

```

Values of $\text{RowBd}[i]$, specifying the location of the top row boundary of the i -th tile row in units of coding treeblocks, are derived as follows:

```
for( RowBd[ 0 ] = 0, i = 0; i <= num_tile_rows_minus1; i++ )
    RowBd[ i + 1 ] = RowBd[ i ] + RowHeight[ i ]
```

$\text{num_substreams_minus1} + 1$ specifies the maximum number of subsets included in a slice when $\text{tiles_or_entropy_coding_sync_idc}$ is equal to 2. When not present, the value of $\text{num_substreams_minus1}$ is inferred to be equal to 0.

$\text{num_entry_point_offsets}$ specifies the number of $\text{entry_point_offset}[i]$ syntax elements in the slice header. When $\text{tiles_or_entropy_coding_sync_idc}$ is equal to 1, the value of $\text{num_entry_point_offsets}$ shall be in the range of 0 to $(\text{num_tile_columns_minus1} + 1) * (\text{num_tile_rows_minus1} + 1) - 1$, inclusive. When $\text{tiles_or_entropy_coding_sync_idc}$ is equal to 2, the value of $\text{num_entry_point_offsets}$ shall be in the range of 0 to $\text{num_substreams_minus1}$, inclusive. When not present, the value of $\text{num_entry_point_offsets}$ is inferred to be equal to 0.

$\text{offset_len_minus1} + 1$ specifies the length, in bits, of the $\text{entry_point_offset}[i]$ syntax elements.

$\text{entry_point_offset}[i]$ specifies the i -th entry point offset, in bytes and shall be represented by $\text{offset_len_minus1} + 1$ bits. The coded slice NAL unit consists of $\text{num_entry_point_offsets} + 1$ subsets, with subset index values ranging from 0 to $\text{num_entry_point_offsets}$, inclusive. Subset 0 consists of bytes 0 to $\text{entry_point_offset}[0] - 1$, inclusive, of the coded slice NAL unit, subset k , with k in the range of 1 to $\text{num_entry_point_offsets} - 1$, inclusive, consists of bytes $\text{entry_point_offset}[k - 1]$ to $\text{entry_point_offset}[k] + \text{entry_point_offset}[k - 1] - 1$, inclusive, of the coded slice NAL unit, and the last subset (with subset index equal to $\text{num_entry_point_offsets}$) consists of the remaining bytes of the coded slice NAL unit.

NOTE – The NAL unit header and the slice header of a coded slice NAL unit are always included in subset 0.

When $\text{tiles_or_entropy_coding_sync_idc}$ is equal to 1 and $\text{num_entry_point_offsets}$ is greater than 0, each subset shall contain all coded bits of one or multiple complete tiles, and the number of subsets shall be equal to or less than the number of tiles in the slice.

When `tiles_or_entropy_coding_sync_idc` is equal to 2 and `num_entry_point_offsets` is greater than 0, subset `k`, for each of all the possible `k` values, shall contain all bits to be used during the initialization process for the current bitstream pointer `k`.

- 5 Regarding the slice data semantics, the following may apply.

`end_of_slice_flag` equal to 0 specifies that another macroblock is following in the slice.
`end_of_slice_flag` equal to 1 specifies the end of the slice and that no further macroblock follows.

10

`entry_point_marker_two_3bytes` is a fixed-value sequence of 3 bytes equal to 0x000002. This syntax element is called an entry marker prefix.

- 15 `tile_idx_minus_1` specifies the TileID in raster scan order. The first tile in the picture shall have a TileID of 0. The value of `tile_idx_minus_1` shall be in the range of 0 to $(\text{num_tile_columns_minus1} + 1) * (\text{num_tile_rows_minus1} + 1) - 1$.

A CABAC parsing process for slice data could be as follows:

- 20 This process is invoked when parsing syntax elements with descriptor `ae(v)`.

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

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

- When starting the parsing of the slice data of a slice, the initialization process of a CABAC parsing process is invoked. When `tiles_or_entropy_coding_sync_idc` is equal to 2 and `num_substreams_minus1` is greater than 0, a mapping table `BitStreamTable` with
 30 `num_substreams_minus1 + 1` entries specifying a bitstream pointer table to use for later current bitstream pointer derivation is derived as follows.

- `BitStreamTable[ 0 ]` is initialized to contain the bitstream pointer.
- For all indices `i` greater than 0 and less than `num_substreams_minus1+1`, `BitStreamTable[ i ]` contains a bitstream pointer to `entry_point_offset[ i ]` bytes after
 35 `BitStreamTable[ i - 1 ]`.

The current bitstream pointer is set to `BitStreamTable[ 0 ]`.

The minimum coding block address of the coding treeblock containing the spatial neighbor block T, `ctbMinCbAddrT`, is derived using a location (`x0`, `y0`) of the top-left luma sample of the current coding treeblock such so, for example, as follows.

$x = x0 + 2 \ll \text{Log2CtbSize} - 1$

5 $y = y0 - 1$

$\text{ctbMinCbAddrT} = \text{MinCbAddrZS}[x \gg \text{Log2MinCbSize}][y \gg \text{Log2MinCbSize}]$

The variable `availableFlagT` is obtained by invoking an appropriate coding block availability derivation process with `ctbMinCbAddrT` as input.

10 When starting the parsing of a coding tree and `tiles_or_entropy_coding_sync_idc` is equal to 2 and `num_substreams_minus1` is greater than 0 , the following applies.

– If `CtbAddrRS % PicWidthInCtbs` is equal to 0, the following applies.

– When `availableFlagT` is equal to 1, the synchronization process of the CABAC parsing process is invoked as specified in subclause “Synchronization process for context variables”.

15

– A decoding process for binary decisions before termination is invoked, followed by an initialisation process for the arithmetic decoding engine.

– The current bitstream pointer is set to indicate `BitStreamTable[i]` with the index `i` derived as follows.

20

$i = (\text{CtbAddrRS} / \text{PicWidthInCtbs}) \% (\text{num_substreams_minus1} + 1)$

– Otherwise, if `CtbAddrRS % PicWidthInCtbs` is equal to 2, the memorization process of the CABAC parsing process is invoked as specified in subclause “Memorization process for context variables”.

25

An initialization process could be as follows:

Outputs of this process are initialised CABAC internal variables.

30

Special processes thereof are invoked when starting the parsing of the slice data of a slice or when starting the parsing of the data of a coding tree and the coding tree is the first coding tree in a tile.

35 Memorization process for context variables could be as follows:

Inputs of this process are the CABAC context variables indexed by `ctxIdx`.

Output of this process are variables TableStateSync and TableMPSSync containing the values of the variables m and n used in the initialization process of context variables that are assigned to syntax elements except for the end-of-slice flag.

For each context variable, the corresponding entries n and m of tables TableStateSync and TableMPSSync are initialized to the corresponding pStateIdx and valMPS.

Synchronization process for context variables could be as follows:

Inputs of this process are variables TableStateSync and TableMPSSync containing the values of the variables n and m used in the memorization process of context variables that are assigned to syntax elements except for the end-of-slice flag.

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

For each context variable, the corresponding context variables pStateIdx and valMPS are initialized to the corresponding entries n and m of tables TableStateSync and TableMPSSync.

In the following, a low delay coding and transport using WPP is explained. In particular, the below discussion reveals as to how the low-delay transport as described in Fig. 7 can be also applied to WPP.

First of all, it is important that a subset of the picture can be sent, before completion of the whole picture. Normally, this is achievable using slices, as already shown in Fig. 5.

In order to reduce the delay compared to tiles, as shown in the following figures, there is the need to apply a single WPP substream per row of LCUs and further allow the separate transmission of each of those rows. In order to keep the coding efficiency high, slices per each row/sub stream cannot be used. Therefore, below, a so-called Dependent Slice as defined in the next section is introduced. This slice, for example, has not all fields of the full HEVC Slice Header, but the fields used for Entropy Slices. Furthermore, there may be a switch to turn off the break of CABAC between rows. In case of WPP, usage of CABAC context (arrows in Fig. 14) and prediction of rows shall be allowed to keep the coding efficiency gain of WPP over tiles.

In particular, Fig. 14 exemplifies a picture 10 for WPP into a regular slice 900 (reg. SL), and, for low delay processing, into dependent slices(OS) 920.

Currently the upcoming HEVC standard offers two types of partitioning in terms of slices. There are regular (normal) slice and entropy slice. The regular slice is completely independent picture partition except some dependencies that may be available due to deblocking filter process on slice boundaries. The entropy slice is also independent but only in terms of entropy coding. The idea of Fig. 14 is to generalize the slicing concept. Thus the upcoming HEVC standard should offer two general types of slices: independent (regular) or dependent. Therefore, a new type of slice, a Dependent Slice, is introduced.

Dependent Slice is a slice that has dependencies to previous slice. The dependencies are a particular data that can be utilized between slices in entropy decoding process and/or pixel reconstruction process.

In Figure 14 a concept of dependent slices is exemplarily presented. The picture starts, for example, always with a regular slice. Note, in this concept the regular slice behavior is slightly changed. Typically, in the standards like H264/AVC or HEVC, regular slice is completely independent partition and do not have to keep any data after decoding except some data for deblocking filter process. But the processing of forthcoming dependent slice 920 is only possible by referencing the data of slice above, here in the first row: regular slice 900. To establish that, regular slices 900 should keep the data of the last CU-row.

This data comprises:

- CABAC coding engine data (context model states of one CU from that the entropy decoding process of dependent slice can be initialized),
- all decoded syntax elements of CUs for regular CABAC decoding process of dependent CUs,
- Data of intra and motion vector prediction.

Consequently each dependent slice 920 shall do the same procedure – keep data for forthcoming dependent slice in the same picture.

In practice, these additional steps should not be an issue, because the decoding process in generally is forced always to store some data like syntax elements.

In the sections below possible changes for the HEVC standard syntax that are required to enable the concept of dependent slices, are presented.

Fig. 5, for example, illustrates possible changes in picture parameter set RBSP syntax

Picture parameter set semantics for dependent slices could be as follows:

dependent_slices_present_flag equal to 1 specifies that the picture is containing dependent slices and the decoding process of each (regular or dependent) slice shall store states of entropy decoding and data of intra and motion vector prediction for next slice that may be a dependent slice that may also follow the regular slice. The following dependent slice may reference that stored data.

Fig. 16 shows a possible slice_header syntax with the changes relative to HEVC's current status.

dependent_slice_flag equal to 1 specifies that the value of slice header syntax elements not present is inferred to be equal to the value of slice header syntax elements in a proceeding (regular) slice, where a proceeding slice is defined as the slice containing the coding treeblock with location ($\text{SliceCtbAddrRS} - 1$). **dependent_slice_flag** shall be equal to 0 when SliceCtbAddrRS equal to 0.

no_cabac_reset_flag equal to 1 specifies CABAC initialization from saved state of previously decoded slice (and not with initial values). Otherwise, i.e. if 0, CABAC initialization independent from any state of previously decoded slice, i.e. with initial values.

last_ctb_cabac_init_flag equal to 1 specifies CABAC initialization from saved state of last coded treeblock of previously decoded slice (e.g. for tiles always equal to 1). Otherwise (equals to 0), initialization data is referenced from saved state of second coded treeblock of last (neighboring) ctb-row of previously decoded slice, if the first coded treeblock of current slice is first coded treeblock in row (i.e. WPP mode), otherwise CABAC initialization is preformed from saved state of last coded treeblock of previously decoded slice.

A comparison of Dependent Slices and other partitioning schemes(informative) is provided below.

In Fig. 17, the difference between normal and dependent slices is shown.

A possible coding and transmission of WPP substreams in dependent slices (DS) as illustrated with respect to Fig. 18 compares an encoding for low delay transport of tiles (left) and WPP/DS (right). The bold continuously drawn crosses in Fig. 18 show the same

time point of time for the two methods assuming that the encoding of the WPP row take the same time as the encoding of a single tile. Due to the coding dependencies, only the first line of WPP is ready, after all tiles have been encoded. But using the dependent slice approach allows the WPP approach to send out the first row once it is encoded. This is different from earlier WPP substream assignments, “substream” is defined for WPP as a concatenation of CU rows of slice to be WPP decoded by the same decoder thread, i.e. the same core/processor. Although, a substream per row and per entropy slice would also have been possible before, the entropy slice breaks the entropy coding dependencies and has therefore lower coding efficiency, i.e. the WPP efficiency gain is lost.

Additionally the delay difference between both approaches may be really low, assuming a transmission as shown in Fig. 19. In particular, Fig. 19 illustrates a WPP coding with pipelined low delay transmission.

Assuming that the encoding of the latter two CUs of DS #1.1 in the WPP approach in Fig. 18, does not take longer than the transmission of the first row SL #1, there is no difference between Tiles and WPP in the low delay case. But the coding efficiency of WP/DS outperforms the tile concept.

In order to increase the robustness for WPP low delay mode, Fig. 20 illustrates that robustness improving is achieved by using Regular Slices (RS) as anchors. In the picture shown in Fig. 20 a (regular) slice (RS) is followed by dependent slices (DS). Here, the (regular) slice acts as an anchor to break the dependencies to preceding slices, hence more robustness is provided at such insertion point of a (regular) slice. In principal, this is not different from inserting (regular) slices anyway.

The concept of dependent slices could also be implemented as follows.

Here, Fig. 21 shows a possible slice header syntax.

The slice header semantics is as follows:

dependent_slice_flag equal to 1 specifies that the value of each slice header syntax element not present is inferred to be equal to the value of corresponding slice header syntax element in the preceding slice containing the coding tree block for which the coding tree block address is $\text{SliceCtbAddrRS} - 1$. When not present, the value of **dependent_slice_flag** is inferred to be equal to 0. The value of **dependent_slice_flag** shall be equal to 0 when SliceCtbAddrRS equal to 0.

slice_address specifies the address in slice granularity resolution in which the slice starts. The length of the slice_address syntax element is $(\text{Ceil}(\text{Log}_2(\text{PicWidthInCtbs} * \text{PicHeightInCtbs})) + \text{SliceGranularity})$ bits.

5

The variable SliceCtbAddrRS, specifying the coding tree block in which the slice starts in coding tree block raster scan order, is derived as follows.

$$\text{SliceCtbAddrRS} = (\text{slice_address} \gg \text{SliceGranularity})$$

10

The variable SliceCbAddrZS, specifying the address of first coding block in the slice in minimum coding block granularity in z-scan order, is derived as follows.

$$\text{SliceCbAddrZS} = \text{slice_address}$$

15

$$\ll ((\text{log2_diff_max_min_coding_block_size} - \text{SliceGranularity}) \ll 1)$$

The slice decoding starts with the largest coding unit possible, or, in other terms, CTU, at the slice starting coordinate.

20

first_slice_in_pic_flag indicates whether the slice is the first slice of the picture. If first_slice_in_pic_flag is equal to 1, the variables SliceCbAddrZS and SliceCtbAddrRS are both set to 0 and the decoding starts with the first coding tree block in the picture.

pic_parameter_set_id specifies the picture parameter set in use. The value of pic_parameter_set_id shall be in the range of 0 to 255, inclusive.

25

num_entry_point_offsets specifies the number of entry_point_offset[i] syntax elements in the slice header. When tiles_or_entropy_coding_sync_idc is equal to 1, the value of num_entry_point_offsets shall be in the range of 0 to $(\text{num_tile_columns_minus1} + 1) * (\text{num_tile_rows_minus1} + 1) - 1$, inclusive. When tiles_or_entropy_coding_sync_idc is equal to 2, the value of num_entry_point_offsets shall be in the range of 0 to $\text{PicHeightInCtbs} - 1$, inclusive. When not present, the value of num_entry_point_offsets is inferred to be equal to 0.

30

35

offset_len_minus1 plus 1 specifies the length, in bits, of the entry_point_offset[i] syntax elements.

entry_point_offset[i] specifies the i-th entry point offset, in bytes and shall be represented by offset_len_minus1 plus 1 bits. The coded slice data after the slice header consists of num_entry_point_offsets + 1 subsets, with subset index values ranging from 0 to num_entry_point_offsets, inclusive. Subset 0 consists of bytes 0 to entry_point_offset[0] - 1, inclusive, of the coded slice data, subset k, with k in the range of 1 to num_entry_point_offsets - 1, inclusive, consists of bytes entry_point_offset[k - 1] to entry_point_offset[k] + entry_point_offset[k - 1] - 1, inclusive, of the coded slice data, and the last subset (with subset index equal to num_entry_point_offsets) consists of the remaining bytes of the coded slice data.

10

When tiles_or_entropy_coding_sync_idc is equal to 1 and num_entry_point_offsets is greater than 0, each subset shall contain all coded bits of exactly one tile, and the number of subsets (i.e., the value of num_entry_point_offsets + 1) shall be equal to or less than the number of tiles in the slice.

15

NOTE – When tiles_or_entropy_coding_sync_idc is equal to 1, each slice must include either a subset of one tile (in which case signalling of entry points is unnecessary) or an integer number of complete tiles.

20

When tiles_or_entropy_coding_sync_idc is equal to 2 and num_entry_point_offsets is greater than 0, each subset k with k in the range of 0 to num_entry_point_offsets - 1, inclusive, shall contain all coded bits of exactly one row of coding tree blocks, the last subset (with subset index equal to num_entry_point_offsets) shall contain all coded bits of the remaining coding blocks included in the slice, wherein the remaining coding blocks consist of either exactly one row of coding tree blocks or a subset of one row of coding tree blocks, and the number of subsets (i.e., the value of num_entry_point_offsets + 1) shall be equal to the number of rows of coding tree blocks in the slice, wherein a subset of one row of coding tree blocks in the slice is also counted..

25

30

NOTE When tiles_or_entropy_coding_sync_idc is equal to 2, a slice may include a number of rows of coding tree blocks and a subset of a row of coding tree blocks. For example, if a slice include two and a half rows of coding tree blocks, the the number of subsets (i.e., the value of num_entry_point_offsets + 1) shall be equal to 3.

35

The corresponding picture parameter set RBSP syntax could be selected as shown in Fig. 22.

The picture parameter set RBSP semantics could be as follows:

dependent_slice_enabled_flag equal to 1 specifies the presence of the syntax element **dependent_slice_flag** in the slice header for coded pictures referring to the picture parameter set. **dependent_slice_enabled_flag** equal to 0 specifies the absence of the syntax element **dependent_slice_flag** in the slice header for coded pictures referring to the picture parameter set. When **tiles_or_entropy_coding_sync_idc** is equal to 3, the value of **dependent_slice_enabled_flag** shall be equal to 1.

tiles_or_entropy_coding_sync_idc equal to 0 specifies that there shall be only one tile in each picture referring to the picture parameter set, there shall be no specific synchronization process for context variables invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of **cabac_independent_flag** and **dependent_slice_flag** for coded pictures referring to the picture parameter set shall not be both equal to 1.

Note, when **cabac_independent_flag** and **dependent_slice_flag** are both equal to 1 for a slice, the slice is an entropy slice.

tiles_or_entropy_coding_sync_idc equal to 1 specifies that there may be more than one tile in each picture referring to the picture parameter set, there shall be no specific synchronization process for context variables invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of **cabac_independent_flag** and **dependent_slice_flag** for coded pictures referring to the picture parameter set shall not be both equal to 1.

tiles_or_entropy_coding_sync_idc equal to 2 specifies that there shall be only one tile in each picture referring to the picture parameter set, a specific synchronization process for context variables shall be invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set and a specific memorization process for context variables shall be invoked after decoding two coding tree blocks of a row of coding tree blocks in each picture referring to the picture parameter set, and the values of **cabac_independent_flag** and **dependent_slice_flag** for coded pictures referring to the picture parameter set shall not be both equal to 1.

tiles_or_entropy_coding_sync_idc equal to 3 specifies that there shall be only one tile in each picture referring to the picture parameter set, there shall be no specific synchronization process for context variables invoked before decoding the first coding tree block of a row of coding tree blocks in each picture referring to the picture parameter set,

and the values of `cabac_independent_flag` and `dependent_slice_flag` for coded pictures referring to the picture parameter set may both be equal to 1.

When `dependent_slice_enabled_flag` shall be equal to 0, `tiles_or_entropy_coding_sync_idc` shall not be equal to 3.

It's a requirement of bitstream conformance that the value of `tiles_or_entropy_coding_sync_idc` shall be the same for all picture parameter sets that are activated within a coded video sequence.

For each slice referring to the picture parameter set, when `tiles_or_entropy_coding_sync_idc` is equal to 2 and the first coding block in the slice is not the first coding block in the first coding tree block of a row of coding tree blocks, the last coding block in the slice shall belong to the same row of coding tree blocks as the first coding block in the slice slice.

`num_tile_columns_minus1` plus 1 specifies the number of tile columns partitioning the picture.

`num_tile_rows_minus1 plus 1` specifies the number of tile rows partitioning the picture.

When `num_tile_columns_minus1` is equal to 0, `num_tile_rows_minus1` shall not be equal to 0. `uniform_spacing_flag` equal to 1 specifies that column boundaries and likewise row boundaries are distributed uniformly across the picture. `uniform_spacing_flag` equal to 0 specifies that column boundaries and likewise row boundaries are not distributed uniformly across the picture but signalled explicitly using the syntax elements `column_width[ i ]` and `row_height[ i ]`.

`column_width[ i ]` specifies the width of the *i*-th tile column in units of coding tree blocks.

`row_height[ i ]` specifies the height of the *i*-th tile row in units of coding tree blocks.

The vector `colWidth[ i ]` specifies the width of the *i*-th tile column in units of CTBs with the column *i* ranging from 0 to `num_tile_columns_minus1`, inclusive.

The vector `CtbAddrRStoTS[ ctbAddrRS ]` specifies the conversion from a CTB address in raster scan order to a CTB address in tile scan order with the index `ctbAddrRS` ranging from 0 to $(\text{picHeightInCtbs} * \text{picWidthInCtbs}) - 1$, inclusive.

The vector `CtbAddrTstoRS[ ctbAddrTS ]` specifies the conversation from a CTB address in tile scan order to a CTB address in raster scan order with the index `ctbAddrTS` ranging from 0 to $(\text{picHeightInCtbs} * \text{picWidthInCtbs}) - 1$, inclusive.

- 5 The vector `TileId[ ctbAddrTS ]` specifies the conversation from a CTB address in tile scan order to a tile id with `ctbAddrTS` ranging from 0 to $(\text{picHeightInCtbs} * \text{picWidthInCtbs}) - 1$, inclusive.

10 The values of `colWidth`, `CtbAddrRstoTS`, `CtbAddrTstoRS` and `TileId` are derived by invoking a CTB raster and tile scanning conversation process with `PicHeightInCtbs` and `PicWidthInCtbs` as inputs and the output is assigned to `colWidth`, `CtbAddrRstoTS` and `TileId`.

15 The values of `ColumnWidthInLumaSamples[ i ]`, specifying the width of the *i*-th tile column in units of luma samples, are set equal to `colWidth[ i ] << Log2CtbSize`.

20 The array `MinCbAddrZS[ x ][ y ]`, specifying the conversation from a location (*x*, *y*) in units of minimum CBs to a minimum CB address in z-scan order with *x* ranging from 0 to $\text{picWidthInMinCbs} - 1$, inclusive, and *y* ranging from 0 to $\text{picHeightInMinCbs} - 1$, inclusive, is dervied by invoking a Z scanning order array initialization process with `Log2MinCbSize`, `Log2CtbSize`, `PicHeightInCtbs`, `PicWidthInCtbs`, and the vector `CtbAddrRstoTS` as inputs and the output is assigned to `MinCbAddrZS`.

25 **loop_filter_across_tiles_enabled_flag** equal to 1 specifies that in-loop filtering operations are performed across tile boundaries. **loop_filter_across_tiles_enabled_flag** equal to 0 specifies that in-loop filtering operations are not performed across tile boundaries. The in-loop filtering operations include the deblocking filter, sample adaptive offset, and adaptive loop filter operations. When not present, the value of **loop_filter_across_tiles_enabled_flag** is inferred to be equal to 1.

30

cabac_independent_flag equal to 1 specifies that CABAC decoding of coding blocks in a slice is independent from any state of the previously decoded slice. **cabac_independent_flag** equal to 0 specifies that CABAC decoding of coding blocks in a slice is dependent from the states of the previously decoded slice. When not present, the value of **cabac_independent_flag** is inferred to be equal to 0.

35

A derivation process for the availability of a coding block with a minimum coding block address could be as follows:

Inputs to this process are

- a minimum coding block address minCbAddrZS in z-scan order
- the current minimum coding block address currMinCBAddrZS in z-scan order

5

Output of this process is the availability of the coding block with minimum coding block address cbAddrZS in z-scan order cbAvailable.

NOTE 1 – The meaning of availability is determined when this process is invoked.

10 NOTE 2 – Any coding block, regardless of its size, is associated with a minimum coding block address, which is the address of the coding block with the minimum coding block size in z-scan order.

- If one or more of the following conditions are true, cbAvailable is set to FALSE.
 - 15 – minCbAddrZS is less than 0
 - minCbAddrZS is greater than currMinCBAddrZS
 - the coding block with minimum coding block address minCbAddrZS belongs to a different slice than the coding block with the current minimum coding block address currMinCBAddrZS and the dependent_slice_flag of the slice containing the coding block with the current minimum coding block address currMinCBAddrZS is equal to 0.
 - 20 – the coding block with minimum coding block address minCbAddrZS is contained in a different tile than the coding block with the current minimum coding block address currMinCBAddrZS.

25

- Otherwise, cbAvailable is set to TRUE.

A CABAC parsing process for slice data could be as follows:

30 This process is invoked when parsing certain syntax elements with descriptor ae(v).

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

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

When starting the parsing of the slice data of a slice, the initialization process of the CABAC parsing process is invoked.

Fig. 23 illustrates as to how a spatial neighbor T is used to invoke the coding tree block availability derivation process relative to the current coding tree block (informative).

- 5 The minimum coding block address of the coding tree block containing the spatial neighbor block T (Fig. 23), `ctbMinCbAddrT`, is derived using the location (`x0`, `y0`) of the top-left luma sample of the current coding tree block as follows.

$x = x0 + 2 \ll \text{Log2CtbSize} - 1$

10 $y = y0 - 1$

$\text{ctbMinCbAddrT} = \text{MinCbAddrZS}[x \gg \text{Log2MinCbSize}][y \gg \text{Log2MinCbSize}]$

The variable `availableFlagT` is obtained by invoking a coding block availability derivation process with `ctbMinCbAddrT` as input.

15

When starting the parsing of a coding tree as specified, the following ordered steps apply.

The arithmetic decoding engine is initialised as follows.

- 20 If `CtbAddrRS` is equal to `slice_address`, `dependent_slice_flag` is equal to 1 and `entropy_coding_reset_flag` is equal to 0, the following applies.

The synchronization process of the CABAC parsing process is invoked with `TableStateIdxDS` and `TableMPSValDS` as input.

25

A decoding process for binary decisions before termination is invoked, followed by an initialization process for the arithmetic decoding engine.

- 30 Otherwise if `tiles_or_entropy_coding_sync_idc` is equal to 2, and `CtbAddrRS % PicWidthInCtbs` is equal to 0, the following applies.

When `availableFlagT` is equal to 1, a synchronization process of the CABAC parsing process is invoked with `TableStateIdxWPP` and `TableMPSValWPP` as input.

35

A decoding process for binary decisions before termination is invoked, followed by the process for the arithmetic decoding engine.

When `cabac_independent_flag` is equal to 0 and `dependent_slice_flag` is equal to 1, or when `tiles_or_entropy_coding_sync_idc` is equal to 2, the memorization process is applied as follows.

5 When `tiles_or_entropy_coding_sync_idc` is equal to 2 and `CtbAddrRS % PicWidthInCtbs` is equal to 2, the memorization process of the CABAC parsing process is invoked with `TableStateIdxWPP` and `TableMPSValWPP` as output.

10 When `cabac_independent_flag` is equal to 0, `dependent_slice_flag` is equal to 1, and `end_of_slice_flag` is equal to 1, the memorization process of the CABAC parsing process is invoked with `TableStateIdxDS` and `TableMPSValDS` as output.

The parsing of syntax elements proceeds as follows:

15

For each requested value of a syntax element a binarization is derived.

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow.

20

For each bin of the binarization of the syntax element, which is indexed by the variable `binIdx`, a context index `ctxIdx` is derived.

For each `ctxIdx` the arithmetic decoding process is invoked.

25

The resulting sequence (`b0..bbinIdx`) of parsed bins is compared to the set of bin strings given by the binarization process after decoding of each bin. When the sequence matches a bin string in the given set, the corresponding value is assigned to the syntax element.

30 In case the request for a value of a syntax element is processed for the syntax element `pcm_flag` and the decoded value of `pcm_flag` is equal to 1, the decoding engine is initialised after the decoding of any `pcm_alignment_zero_bit`, `num_subsequent_pcm`, and all `pcm_sample_luma` and `pcm_sample_chroma` data.

35 Thus, the above description reveals a decoder as shown in Fig. 24. This decoder, which is generally indicated by reference sign 5, reconstructs a picture 10 from a data stream 12 into which the picture 10 is coded in units of slices 14 into which the picture 10 is partitioned, wherein the decoder 5 is configured to decode the slices 14 from the data stream 12 in

accordance with a slice order 16. Naturally, decoder 5 is not restricted to serially decode the slices 14. Rather, the decoder 5 may use wavefront parallel processing in order to decode the slices 14, provided the pictures 10 partitioning into slices 14 is appropriate for wavefront parallel processing. Accordingly, decoder 5 may, for example, be a decoder
 5 which is able to decode slices 14 in parallel in a staggered manner with starting the decoding of the slices 14 by taking the slice order 16 into account so as to allow wavefront processing as it has been described above and will be described below, too.

Decoder 5 is responsive to a syntax element portion 18 within a current slice of the slices
 10 14 so as to decode the current slice in accordance with one of at least two modes 20 and 22. In accordance with a first of the at least two modes, namely mode 20, the current slice is decoded from the data stream 12 using context adaptive entropy decoding including a derivation of context across slice boundaries, i.e. across the dashed lines in Fig. 24, i.e. by using information stemming from coding/decoding of other “in slice order 16 preceding
 15 slices”. Further, decoding the current slice from the data stream 12 using the first mode 20 comprises a continuous update of symbol probabilities of the codec and an initialization of the symbol probabilities at the beginning of the decoding of the current slice, which depends on saved states of the symbol probabilities of a previously decoded slice. Such dependency was described above, for example, in connection with the “synchronization
 20 process for codec variables”. Finally, the first mode 20 also involves predictive decoding across the slice boundaries. Such predictive decoding across slice boundaries may, for example, involve intra-prediction across slice boundaries, i.e. predicting sample values within the current slice on the basis of already reconstructed sample values of an “in slice order 16”, preceding slice, or a prediction of coding parameters across slice boundaries
 25 such as a prediction of motion vectors, prediction modes, coding modes or the like.

In accordance with the second mode 22, the decoder 5 decodes the current slice, i.e. the slice currently to be decoded, from the data stream 12 using context adaptive entropy decoding with restricting, however, the derivation of the contexts so as to not cross the
 30 slice boundaries. If ever, for example, a template of neighboring positions used for deriving the context for a certain syntax element relating to a block within the current slice extends into a neighboring slice, thereby crossing the slice boundary of the current slice, the corresponding attribute of the respective portion of the neighboring slice, such as the value of the corresponding syntax element of this neighboring portion of the neighboring
 35 slice, is set to the default value in order to inhibit interdependencies between the current slice and the neighboring slices. While a continuous update of symbol probabilities of the contexts may take place just as it is the case in the first mode 20, the initialization of the symbol probabilities in the second mode 22 is independent of any previously decoded

slice. Further, the predictive decoding is performed with restricting the predictive decoding so as to not cross the slice boundaries.

In order to ease the understanding of the description of Fig. 24 and the following description, reference is made to Fig. 25, which shows a possible implementation of decoder 5 in a more structural sense than compared to Fig. 24. As it is the case in Fig. 24, the decoder 5 is a predictive decoder using context adaptive entropy decoding for decoding the data stream so as to obtain, for example, the prediction residual and prediction parameters.

As shown in Fig. 25, the decoder 5 may comprise an entropy decoder 24, a dequantization and inverse transform module 26, a combiner 28 implemented, as shown in Fig. 25, for example, as an adder and a predictor 28. Entropy decoder 24, module 26 and adder 27 are serially connected between input and output of decoder 5 in the order of their mentioning, and predictor 28 is connected between an output of adder 28 and a further input thereof in order to form the prediction loop along with combiner 27. Hence, the decoder 24 has its output additionally connected to a coding parameter input of predictor 28.

Although Fig. 25 provides the impression that the decoder serially decodes the current picture, the decoder 5 may, for example, be implemented so as to decode picture 10 in parallel. The decoder 5 may, for example, comprise multiple cores each operating according to elements 24-28 in Fig. 25. The parallel processing, however, is optional and a serially operating decoder 5 is also able to decode the data stream inbound at the input of entropy decoder 24.

In order to efficiently achieve the just-mentioned ability of either serially or in parallel decoding the current picture 10, the decoder 5 operates in units of coding blocks 30 in order to decode picture 10. Coding blocks 30 are, for example, leaf blocks into which coding tree blocks or largest coding blocks 32 are partitioned by recursive multitree partitioning such as quadtree partitioning. The code treeblocks 32, in turn, may be regularly arranged in columns and rows so as to form a regular partitioning of picture 10 into these code treeblocks 32. In Fig. 25, the code treeblocks 32 are shown with continuous lines, whereas coding blocks 30 are shown with dashed lines. For illustration purposes, merely one code treeblock 32 is shown to be further partitioned into coding blocks 30, while the other code treeblocks 32 are shown to be not further partitioned so as to directly form a coding block, instead. The data stream 12 may comprise a syntax portion signaling as to how picture 10 is partitioned into the code blocks 30.

The data stream 12 conveys, for each coding block 30, syntax elements which reveal as to how modules 24 to 28 are to recover the picture content within that coding block 30. For example, these syntax elements comprise:

- 5 1) optionally, partitioning data further partitioning coding block 30 into prediction blocks,
- 2) optionally, partitioning data further partitioning the coding block 30 into residual and/or transform blocks,
- 10 3) a prediction mode signaling as to which prediction mode is to be used for deriving the prediction signal for the coding block 30, wherein the granularity at which this prediction mode is signaled may depend on coding blocks 30 and/or prediction block.
- 15 4) prediction parameters may be signaled per coding block or, if present, per prediction block with a kind of prediction parameters sent depending, for example, on the prediction mode. Possible prediction modes may, for example, comprise intra-prediction and/or inter-prediction.
- 20 5) Other syntax elements may also be present such as filtering information for filtering picture 10 at coding block 30 so as to obtain the prediction signal and/or the reconstructed signal to be reproduced.
- 25 6) Finally, residual information in form of, inter alia, transform coefficients may be comprised in a data stream for coding block 30; in units of residual blocks, residual data may be signaled; per residual block, the spectral decomposition may , for example, performed in units of the aforementioned transform blocks, if present.
- 30 The entropy decoder 24 is responsible for obtaining the just-mentioned syntax elements from the data stream. To this end, the entropy decoder 24 uses context adaptive entropy decoding. That is, the entropy decoder 24 provides several context. In order to derive a certain syntax element from the data stream 12, the entropy decoder 24 selects a certain context among the possible contexts. The selection among the possible contexts is
- 35 performed depending on an attribute of a neighborhood of the portion of picture 10 to which the current syntax element belongs. For each of the possible contexts, the entropy decoder 24 manages symbol probabilities, i.e., a probability estimate for each possible symbol of the symbol alphabet based on which the entropy decoder 24 operates. The

“managing” involves the aforementioned continuous updates of the symbol probabilities of the contexts so as to adapt the symbol probabilities associated with each context to the actual picture content. By this measure, the symbol probabilities are adapted to the actual probability statistics of the symbols.

5

Another circumstance where attributes of a neighborhood influence the reconstruction of a current portion of picture 10 such as a current coding block 30, is the predictive decoding within predictor 28. The prediction is restricted not only to prediction content within the current coding block 30, but may also encompass prediction of parameters contained within the data stream 12 for the current coding block 30 such as prediction parameters, partitioning data, or even transform coefficients. That is, predictor 28 may predict picture content or such parameters from the aforementioned neighborhood so as to obtain the written signal which is then combined with the prediction residual as obtained by module 26 from data stream 12. In case of predicting parameters, predictor 28 may use syntax elements contained within the data stream as prediction residuals so as to obtain the actual value of the prediction parameter. Predictor 28 uses the latter prediction parameter value in order to obtain the just-mentioned prediction signal to be combined with the prediction residual in combiner 27.

20 The aforementioned “neighborhood” primarily covers the upper left-hand part of the circumference of the current portion to which the syntax element currently to be entropy-decoded or the syntax element to be currently predicted belongs. In Fig. 25, such neighborhood is illustrated at 34 exemplarily for one coding block 30.

25 A coding/decoding order is defined among the coding blocks 30: at a coarsest level, the code treeblocks 32 of picture 10 are scanned in a scan order 36, here illustrated as a raster scan leading row-wise from top to bottom. Within each code treeblock, the coding blocks 30 are scanned in a depth first traversal order such that, in each hierarchy level, the code treeblock 32 is substantially scanned also in a raster scan leading row-wise from top to bottom.

30

The coding order defined among the coding blocks 30 harmonizes with the definition of the neighborhood 34 used for deriving an attribute in the neighborhood so as to select contexts and/or perform spatial prediction in that the neighborhood 34 mostly covers portions of picture 10 which have already been subject to decoding in accordance with the coding order. Whenever a portion of neighborhood 34 covers non-available portions of picture 10, default data is, for example, being used instead. For example, a neighborhood

35

template 34 may extend outside picture 10. Another possibility is, however, that the neighborhood 34 extends into a neighboring slice.

Slices divide-up, for example, picture 10 along a coding/decoding order defined along the coding blocks 30, i.e. each slice is a continuous non-interrupted sequence of coding blocks 30 along the aforementioned coding block order. In Fig. 25, the slices are indicated with dash-dot-lines 14. The order defined among the slices 14 results from their composition of runs of sequential coding blocks 30 as outlined above. If the syntax element portion 18 of a certain slice 14 indicates that same is to be decoded in the first mode, then entropy-decoder 24 allows the context adaptive entropy decoding to derive contexts across slice boundaries. That is, the spatial neighborhood 34 is used in order to select contexts in entropy decoding data concerning the current slice 14. In case of Fig. 25, for example, slice number 3 may be the currently decoded slice, and in entropy-decoding syntax element concerning coding block 30 or some portion contained therein, the entropy decoder 24 may use attributes stemming from decoding portions within the neighboring slice such as slice number 1. The predictor 28 behaves the same: for slices being of the first mode 20, predictor 28 uses spatial prediction across the slice boundary encircling the current slice.

For slices, however, having the second mode 22 associated therewith, i.e. for which syntax element portion 18 indicates the second mode 22, entropy decoder 24 and predictor 28 restrict the derivation of entropy contexts and predictive decoding to depend on attributes relating to portions lying within the current slice only. Obviously, the coding efficiency suffers from this restriction. On the other hand, slices of the second mode 22 allow for disrupting interdependencies between the sequence of slices. Accordingly, slices of the second mode 22 may be interspersed within picture 10 or within a video to which picture 10 belongs in order to allow for resynchronization points. It is not necessary, however, that each picture 10 has at least one slice in the second mode 22.

As already mentioned above, first and second mode 20 and 22 also differ in their initialization of the symbol probabilities. Slices coded in the second mode 22, result in the entropy decoder 24 re-initializing the probabilities independent of any previously decoded slice, i.e. previously decoded in the sense of the order defined among the slices. The symbol probabilities are, for example, set to default values known to both encoder and decoder side, or initialization values are contained within slices coded in a second mode 22.

That is, for slices being coded/decoded in a second mode 22, the adaptation of the symbol probabilities always starts immediately from the beginning of these slices. Accordingly, the adaptation accuracy is bad for these slices at the beginning of these slices.

- 5 Things are different in slices coded/decoded in the first mode 20. For the latter slices, the initialization of the symbol probabilities performed by entropy decoder 24 depends on saved states of symbol probabilities of a previously decoded slice. Whenever a slice coded/decoded in a first mode 20, has its beginning, for example, positioned other than the left-hand side of picture 10, i.e. not at the side from where the raster scan 36 starts running
- 10 row-wise before stepping to the next row bottom-wards, the symbol probabilities as resulting at the end of entropy decoding the immediately preceding slice are adopted. This is illustrated, for example, in Fig. 2 by an arrow 38 for slice no. 4. Slice no. 4 has its beginning somewhere in between the right-hand side and left-hand side of picture 10 and accordingly, in initializing the symbol probabilities, entropy decoder 24 adopts, in
- 15 initializing the symbol probabilities, the symbol probabilities which were obtained in entropy decoding the immediately preceding slice, i.e. slice no. 3, until the end thereof, i.e. including the continuous update of symbol probabilities during the entropy decoding of slice 3 until its end.
- 20 Slices having the second mode 22 associated therewith, which have, however, the beginning at the left-hand side of picture 10 such as, for example, slice no. 5, do not adapt the symbol probabilities as obtained after finishing the entropy decoding of the immediately preceding slice no. 4, because this would prevent decoder 5 from parallelly decoding picture 10 by use of wavefront processing. Rather, as outlined above, the entropy
- 25 decoder 24 adapts the symbol probabilities as obtained after finishing entropy decoding the second - in encoding/decoding order 36 – code treeblock 32 in the immediately preceding –in encoding/decoding order 36 – code treeblock row as illustrated by an arrow 40.

In Fig. 25, for example, picture 10 was exemplarily partitioned into three rows of code treeblocks and four columns of coding tree root blocks 32 and each code treeblock row was sub-divided into two slices 14, so that the beginning of every second slice coincides with the first coding unit in coding unit order of a respective code treeroot block row. The entropy decoder 24 would, accordingly, be able to use wavefront processing in decoding picture 10, by decoding each code tree root block row in parallel, with commencing

30 decoding these code tree root block rows in a staggered manner, starting with a first or

35 topmost code tree root block row, then the second, and then the third.

Naturally, the partitioning of blocks 32 in a recursive manner into further coding blocks 30 is optional and accordingly, in a more general sense, blocks 32 could be called “coding blocks” as well. That is, more generally speaking, picture 10 may be partitioned into coding blocks 32 arranged in rows and columns and having a raster scan order 36 defined among each other, and the decoder 5 may be considered to associate each slice 14 with a continuous subset of the coding blocks 32 in the raster scan order 36 so that the subsets follow each other along the raster scan order 36 in accordance with the slice order.

As became also clear from the above discussion, the decoder 5 or, more specifically, the entropy decoder 24 may be configured to save symbol probabilities as obtained in context adaptive entropy decoding any slice up to a second coding block in a coding block row in accordance with the raster scan order 36. In initializing the symbol probabilities for the context adaptive entropy decoding of a current slice having the first mode 20 associated therewith, the decoder 5, or, more specifically, the entropy decoder 24, checks as to whether a first coding block 32 of the continuous subset of coding blocks 32 associated with the current slice is a first coding block 32 in a coding block row in accordance with the raster scan order 36. If so, the symbol probabilities for the context adaptive entropy decoding of the current slice is initialized as explained with respect to arrow 40, namely depending on the saved symbol probabilities as obtained in context entropy decoding the previously decoded slice up to the second coding block in coding block row in accordance with the raster scan order 36. If not, initialization of the symbol probabilities for the context adaptive entropy decoding of the current slice is performed depending on symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to the end of the previously decoded slice, i.e. according to arrow 38. Again, in case of initialization according to 38, the saved state at the end of entropy decoding the immediately preceding slice in slice order 36 is meant, whereas in case of initialization 40, it is that previously decoded slice comprising the end of the second block of the immediately preceding row of block 32 in block order 36.

As illustrated by dashed lines in Fig. 24, the decoder may be configured to be responsive to the syntax element portion 18 within the current slice of the slices 14 so as to decode the current slice in accordance with one of at least three modes. That is, there may be a third mode 42 beside the other ones 20 and 22. The third mode 42 may differ from the second mode 22 in that prediction across slice boundaries is allowed, whereas entropy coding/decoding is still restricted so as to not cross slice boundaries.

Above, two embodiments were presented regarding the syntax element portion 18. The table below summarizes these two embodiments.

	embodiment 1	embodiment 2
syntax element portion	dependent_slice_flag, no_cabac_reset_flag	dependent_slice_flag
mode 1	dependent_slice_flag = 1, no_cabac_reset_flag = 1	dependent_slice_flag = 1,
mode 2	dependent_slice_flag = 0	dependent_slice_flag = 0
mode 3	dependent_slice_flag = 1, no_cabac_reset_flag = 0	dependent_slice_flag = 1, cabac_independent_flag = 1, tiles_or_entropy_coding_sync_idc = 3

In the one embodiment, the syntax element portion 18 is formed by dependent_slice_flag individually, while in the other embodiment, the combination of dependent_slice_flag and no_cabac_reset_flag forms the syntax element portion. Reference is made to the synchronization process for context variables as far as the initialization of symbol probabilities depending on saved states of symbol probabilities of a previously decoded slice is concerned. In particular, the decoder may be configured to, if last_ctb_cabac_init_flag=0 and tiles_or_entropy_coding_sync_idc=2, save symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to a second coding block in a row in accordance with the raster scan order, and, in initializing the symbol probabilities for the context adaptive entropy decoding of the current slice in accordance with the first mode, check as to whether a first coding block of the continuous subset of coding blocks associated with the current slice is a first coding block in a row in accordance with the raster scan order, and, if so, initialize the symbol probabilities for the context adaptive entropy decoding of the current slice depending on the saved symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to a second coding block in a row in accordance with the raster scan order, and, if not, initialize the symbol probabilities for the context adaptive entropy decoding of the current slice depending on symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to the end of the previously decoded slice.

Thus, in other words, in accordance with the second embodiment for a syntax, the decoder would reconstruct picture 10 from datastream 12 into which the picture is coded in units of

slices 14 into which the picture (10) is partitioned, wherein the decoder is configured to decode the slices 14 from the datastream 12 in accordance with a slice order 16 and the decoder is responsive to a syntax element portion 18, namely `dependent_slice_flag` within a current slice of the slices, so as to decode the current slice in accordance with one of at least two modes 20, 22. In accordance with a first 20 of the at least two modes, namely if `dependent_slice_flag=1`, the decoder decodes the current slice from the datastream 12 using context adaptive entropy decoding 24 including a derivation of contexts across slice boundaries, a continuous update of symbol probabilities of the contexts and an initialization 38, 40 of the symbol probabilities depending on saved states of symbol probabilities of a previously decoded slice, and predictive decoding across the slice boundaries, and in accordance with a second 22 of the at least two modes, namely if `dependent_slice_flag=0`, the decoder decodes the current slice from the datastream 12 using context adaptive entropy decoding with restricting the derivation of the contexts so as to not cross the slice boundaries, a continuous update of symbol probabilities of the contexts and an initialization of the symbol probabilities independent on any previously decoded slice, and predictive decoding with restricting the predictive decoding so as to not cross the slice boundaries. The picture 10 may be partitioned in coding blocks 32 arranged in rows and columns and having a raster scan order 36 defined among each other, and the decoder is configured to associate each slice 14 with a continuous subset of the coding blocks 32 in the raster scan order 36 so that the subsets follow each other along the raster scan order 36 in accordance with the slice order. The decoder may be configured to, namely responsive to `tiles_or_entropy_coding_sync_idc=2`, save symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to a second coding block 32 in a row in accordance with the raster scan order 36, and, in initializing the symbol probabilities for the context adaptive entropy decoding of the current slice in accordance with the first mode, check as to whether a first coding block of the continuous subset of coding blocks 32 associated with the current slice is a first coding block 32 in a row in accordance with the raster scan order, and, if so, initialize 40 the symbol probabilities for the context adaptive entropy decoding of the current slice depending on the saved symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to a second coding block in a row in accordance with the raster scan order 36, and, if not, initialize 38 the symbol probabilities for the context adaptive entropy decoding of the current slice depending on symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to the end of the previously decoded slice. The decoder may be configured to be responsive to the syntax element portion (18) within the current slice of the slices 14, so as to decode the current slice in accordance with one of at least three modes, namely in the one of the first 20 and a third mode 42 or a second mode 22, wherein the decoder is configured to, in accordance

with the third mode 42, namely if **dependent_slice_flag**=1 and tiles_or_entropy_coding_sync_idc=3, decode the current slice from the datastream using context adaptive entropy decoding with restricting the derivation of the contexts so as to not cross the slice boundaries, a continuous update of symbol probabilities of the contexts and an initialization of the symbol probabilities independent on any previously decoded slice, and predictive decoding across the slice boundaries, wherein the one of the first and third modes is selected depending on a syntax element, namely cabac_independent_flag. The decoder may further be configured to, namely if tiles_or_entropy_coding_sync_idc=0,1, and 3 ("3" when cabac_independent flag=0), save symbol probabilities as obtained in context adaptive entropy decoding the previously decoded slice up to an end of the previously decoded slice, and, in initializing the symbol probabilities for the context adaptive entropy decoding of the current slice in accordance with the first mode, initialize the symbol probabilities for the context adaptive entropy decoding of the current slice depending on symbol probabilities saved. The decoder may be configured to, namely if tiles_or_entropy_coding_sync_idc=1, in the first and second mode, restrict the predictive decoding within tiles into which the picture is sub-divided.

Naturally, an encoder is able to set the above presented syntax accordingly in order to enable the decoder to gain the above outlined advantages. The encoder may a parallel processing, such as a multi-core, encoder, but does not need to be. For encoding picture 10 into datastream 12 in units of slices 14, the encoder would be configured to encode the slices 14 into the datastream 12 in accordance with slice order 16. The encoder would determine syntax element portion 18 for, and code same into, a current slice of the slices so that the syntax element portion signals the current slice to be coded in accordance with one of the at least two modes 20, 22, and if the current slice is to be coded in accordance with a first 20 of the at least two modes, encode the current slice into the datastream 12 using context adaptive entropy encoding 24 including a derivation of contexts across slice boundaries, a continuous update of symbol probabilities of the contexts and an initialization 38, 40 of the symbol probabilities depending on saved states of symbol probabilities of a previously encoded slice, and predictive encoding across the slice boundaries, and if the current slice is to be coded in accordance with a second 22 of the at least two modes, encode the current slice into the datastream 12 using context adaptive entropy encoding with restricting the derivation of the contexts so as to not cross the slice boundaries, a continuous update of symbol probabilities of the contexts and an initialization of the symbol probabilities independent on any previously encoded slice, and predictive encoding with restricting the predictive encoding so as to not cross the slice boundaries. While the picture 10 may be partitioned in coding blocks 32 arranged in rows and columns and having a raster scan order 36 defined among each other, the encoder may

be configured to associate each slice 14 with a continuous subset of the coding blocks 32 in the raster scan order 36 so that the subsets follow each other along the raster scan order 36 in accordance with the slice order. The encoder may be configured to save symbol probabilities as obtained in context adaptive entropy encoding the previously encoded slice up to a second coding block 32 in a row in accordance with the raster scan order 36, and, in initializing the symbol probabilities for the context adaptive entropy encoding of the current slice in accordance with the first mode, check as to whether a first coding block of the continuous subset of coding blocks 32 associated with the current slice is a first coding block 32 in a row in accordance with the raster scan order, and, if so, initialize 40 the symbol probabilities for the context adaptive entropy encoding of the current slice depending on the saved symbol probabilities as obtained in context adaptive entropy encoding the previously encoded slice up to a second coding block in a row in accordance with the raster scan order 36, and, if not, initialize 38 the symbol probabilities for the context adaptive entropy encoding of the current slice depending on symbol probabilities as obtained in context adaptive entropy encoding the previously decoded slice up to the end of the previously encoded slice. The encoder may be configured to be code the syntax element portion (18) into the current slice of the slices (14) so that the current slice is signaled to be coded thereinto in accordance with one of at least three modes, namely in the one of the first (20) and a third mode (42) or a second mode (22), wherein the encoder is configured to ** in accordance with the third mode (42), encode the current slice into the datastream using context adaptive entropy encoding with restricting the derivation of the contexts so as to not cross the slice boundaries, a continuous update of symbol probabilities of the contexts and an initialization of the symbol probabilities independent on any previously encoded slice, and predictive coding across the slice boundaries, wherein the encoder distinguishes between the one of the first and third modes being using a syntax element, namely `cabac_independent_flag`, for example. The encoder may be configured to determine a generic syntax element such as `dependent_slices_present_flag` and write same into the datastream with operating in one of at least two generic operating modes depending on the generic syntax element, namely, with, according to a first generic operating mode, performing coding the syntax element portion for each slice, and, according to a second generic operating mode, inevitably using a different one of the at least two modes other than the first mode. The encoder may be configured to according to the first and second modes, inevitably und uninterruptedly continue continuously updating the symbol probabilities from a beginning to an end of the current slice. The encoder ay be configured to save symbol probabilities as obtained in context adaptive entropy encoding the previously encoded slice up to an end of the previously encoded slice, and, in initializing the symbol probabilities for the context adaptive entropy encoding of the current slice in accordance with the first mode, initialize the symbol probabilities for the

context adaptive entropy encoding of the current slice depending on symbol probabilities saved. And the encoder may, in the first and second mode, restrict the predictive encoding within tiles into which the picture is sub-divided.

- 5 A possible structure of an encoder is depicted in Fig. 26 for sake of completeness. Predictor 70 operates almost the same as predictor 28, i.e. performs the prediction, but also determines, by optimization, for example, the coding parameters including the prediction parameters and modes. Modules 26 and 27 also occur in the decoder. Subtractor 72 determines the lossless prediction residual which is then, lossy - by use of quantization and, optionally, using spectrally decomposing transformation -, coded in the transformation and quantization module 74. Entropy coder 76 performs the context adaptive entropy encoding.

- 15 In addition to the above concrete syntax examples, a different example is outlined below with presenting the concordance between terms used hereinafter and terms used above.

- In particular, without having particularly outlined above, dependent slices are not only “dependent” in that same allow for exploiting knowledge known from outside its boundary, such as, as outlined above, have the entropy contexts adapted faster, or achieve a better spatial prediction due to allowance of crossing its boundary. Rather, in order to save rate cost having to be spent for defining the slice headers by splitting-up the picture into slices, dependent slices adopt a part of the slice header syntax from previous slices, i.e. this slice syntax header portion is not transmitted again for dependent slices. This is shown for example, in Fig. 16 at 100 and in Fig. 21 at 102, according to which the slice type, for example, is adopted from the previous slice. By this measure, the sub-division of a picture into slices, such as an independent slice and dependent slices, as less expensive in terms of bit-consumption expensive.

- It is the just-mentioned dependency which leads, in the below outlined example, to a slightly different wording: slices are defined as unit portions of a picture at which slice header syntax is individually settable. Accordingly, slices are composed of one – using the nomenclature above - independent/regular/normal slice, now called independent slice segment and no, one or more – using the nomenclature above - dependent slices, now called dependent slice segments.

- 35 Fig. 27, for example, shows a picture to be partitioned into two slices, one formed by slice segments 14₁ to 14₃ and the other solely formed by slice segment 14₄. The indices 1 to 4 show the slice order in coding order. Fig. 28a and b show different example in case of a

sub-division of picture 10 into two tiles, with, in case of Fig. 28a, one slice formed by all five slice segments 14, covering both tiles 50₁ and 50₂ – the index again raising in coding order –, and, in case of Fig. 28a, two slices formed by slice segments 14₁ and 14₂ and 14₃ and 14₄, respectively, subdividing tile 50₁, and another slice formed by slice segments 14₅-14₆ covering tile 50₂.

The definitions could be as follows:

dependent slice segment: A *slice segment* for which the values of some *syntax elements* of the *slice segment header* are inferred from the values for the preceding *independent slice segment* in *decoding order* – formerly, in the above embodiments- called a *dependent slice*.

independent slice segment: A *slice segment* for which the values of the *syntax elements* of the *slice segment header* are not inferred from the values for a preceding *slice segment* – formerly, in the above embodiments- called a *normal slice*.

slice: An integer number of *coding tree units* contained in one *independent slice segment* and all subsequent *dependent slice segments* (if any) that precede the next *independent slice segment* (if any) within the same *access unit/picture*.

slice header: The *slice segment header* of the *independent slice segment* that is a current *slice segment* or is the *independent slice segment* that precedes a current *dependent slice segment*.

slice segment: An integer number of *coding tree units* ordered consecutively in the *tile scan* and contained in a single *NAL unit*; the division of each *picture* into slice segments is a *partitioning*.

slice segment header: A part of a coded *slice segment* containing the data elements pertaining to the first or all *coding tree units* represented in the *slice segment*.

The signaling of the “modes” 20 and 22, i.e. “dependent slice segment” und “independent slice segment” could be as follows:

In some extra NAL units such as PPS, a syntax element could be used to signal as to whether usage of dependent slices is made or not for a certain picture of even a sequence for certain pictures:

- 5 **dependent_slice_segments_enabled_flag** equal to 1 specifies the presence of the syntax element `dependent_slice_segment_flag` in the slice segment headers. `dependent_slice_segments_enabled_flag` equal to 0 specifies the absence of the syntax element `dependent_slice_segment_flag` in the slice segment headers.
- 10 **dependent_slice_segments_enabled_flag** is similar in scope to the formerly described **dependent_slices_present_flag**.

Similarly, **dependent_slice_flag** could be called **dependent_slice_segment_flag** so as to account for the different nomenclature with respect to slices.

- 15 **dependent_slice_segment_flag** equal to 1 specifies that the value of each slice segment header syntax element that is not present in the current slice segment's header is inferred to be equal to the value of the corresponding slice segment header syntax element in the slice header, i.e. the slice segment header of the preceding independent slice segment.

- 20 In the same level, such as picture level, the following syntax element could be included:

- entropy_coding_sync_enabled_flag** equal to 1 specifies that a specific synchronization process for context variables is invoked before decoding the coding tree unit which includes the first coding tree block of a row of coding tree blocks in each tile in each picture referring to the PPS, and a specific storage process for context variables is invoked after decoding the coding tree unit which includes the second coding tree block of a row of coding tree blocks in each tile in each picture referring to the PPS. `entropy_coding_sync_enabled_flag` equal to 0 specifies that no specific synchronization process for context variables is required to be invoked before decoding the coding tree unit which includes the first coding tree block of a row of coding tree blocks in each tile in each picture referring to the PPS, and no specific storage process for context variables is required to be invoked after decoding the coding tree unit which includes the second coding tree block of a row of coding tree blocks in each tile in each picture referring to the PPS.
- 35 PPS.

It is a requirement of bitstream conformance that the value of `entropy_coding_sync_enabled_flag` shall be the same for all PPSs that are activated within a CVS.

When `entropy_coding_sync_enabled_flag` is equal to 1 and the first coding tree block in a slice is not the first coding tree block of a row of coding tree blocks in a tile, it is a requirement of bitstream conformance that the last coding tree block in the slice shall belong to the same row of coding tree blocks as the first coding tree block in the slice.

- 5 When `entropy_coding_sync_enabled_flag` is equal to 1 and the first coding tree block in a slice segment is not the first coding tree block of a row of coding tree blocks in a tile, it is a requirement of bitstream conformance that the last coding tree block in the slice segment shall belong to the same row of coding tree blocks as the first coding tree block in the slice segment.

10

As already described, the coding/decoding order among the CTBs 30 leads in raster manner row-wise from top to bottom starting with scanning the first tile with then visiting the next tile, if more than one tile is present in the picture.

15

The decoder 5 – and the encoder accordingly - acts as follows in entropy decoding (coding) the slice segments 14 of the picture:

- 20 A1) Whenever a currently decoded/coded syntax element `synEI` is the first syntax element of a tile 50, slice segment 14 or row of CTB, an initialization process of Fig. 29 is commenced.

A2) Otherwise, decoding of this syntax element takes place using the current entropy contexts.

- 25 A3) If the current syntax element was the last syntax element in a CTB 30, then an entropy context storage process as shown in Fig. 30 is commenced.

A4) The process proceeds at A1) with the next syntax element.

- 30 In the initialization process, it is checked 200 whether `synEI` is the first syntax element of a slice segment 14 or tile 50. If yes, the contexts are initialized independent from any previous slice segment in step 202. If no, is checked 204 whether `synEI` is the first syntax element of a row of CTBs 30 and **`entropy_coding_sync_enabled_flag`** equals one. If yes, it is checked 206 whether in the previous line of CTBs 30 of equal tile, the second CTB 30 is available (see Fig. 23). If yes, a context adoption according to 40 is performed in step 210 using the currently stored context probabilities for type 40 adoptions. If not, contexts are initialized independent from any previous slice segment in step 202. If the ccheck 204 reveals no, then it is checked in step 212, whether `synEI` is the first syntax element in the first CTB of a dependent slice segment 14 and whether `dependent_slice_segement_flag` equals one, and if yes, context adoption according to 38 is performed in step 214 using the
- 35

currently stored context probabilities for type 38 adoptions. After any of steps 214, 212, 210 and 202, the decoding/coding is actually commenced.

Dependent slice segments with `dependent_slice_segement_flag` equaling one, thus help to further decrease coding/decoding delay with almost no coding efficiency penalty.

In the storage process of Fig. 30, it is checked in step 300 whether the coded/decoded `synEl` is the last syntax element of a second CTB 30 of a row of CTBs 30, and **`entropy_coding_sync_enabled_flag`** equals one. If yes, the current entropy contexts are stored in step 302, i.e. the entropy coding probabilities of the contexts, in a storage which is specific for adoptions à la 40. Similarly, it is checked in step 304, in addition to steps 300 or 302, whether the coded/decoded `synEl` is the last syntax element of a slice segment 14, and `dependent_slice_segement_flag` equals one. If yes, the current entropy contexts are stored in step 306, i.e. the entropy coding probabilities of the contexts, in a storage which is specific for adoptions à la 38.

It is noted that any check querying as to whether a syntax element is the first `synEl` of a CTB row, exploits for example, the syntax element `slice_adress` 400 within the slice segment's headers, i.e. a start syntax element revealing a position of a begin of the respective slice segment along the decoding order.

In reconstructing the picture 10 from the datastream 12 using WPP processing, the decoder is able to exploit exactly the latter start syntax portion 400 so as to retrieve the WPP substream entry points. Since each slice segment comprises a start syntax portion 400 indicating the position of a decoding begin of the respective slice segment within the picture 10, the decoder is able to identify the entry points of the WPP substreams into which the slice segments are grouped, by identifying, using the slice segments' start syntax portions 400, slice segments starting at a left hand side of the picture. The decoder may then, in parallel, decode the WPP substreams in a staggered manner with sequentially commencing the decoding of the WPP substreams in accordance with the slice order. The slice segments may even be smaller than one picture width, i.e. one row of CTBs, so that their transmission may be interleaved among the WPP substreams so as to further reduce the overall transmission end-to-end delay. The Encoder provides each slice (14) with a start syntax portion (400) indicating a position of a coding begin of the respective slice within the picture (10) and would group the slices into WPP substreams so that for each WPP substream, the first slice in slice order starts at a left hand side of the picture. The encoder may even, by itself, use WPP processing in encoding the picture: the encoder

parallel encodes the WPP substreams in a staggered manner with sequentially commencing the encoding of the WPP substreams in accordance with the slice order.

- 5 By the way, the latter aspect of using the slice segment's start syntax portions as a means for locating the WPP substream's entry points, may be used without the dependent slice concept.

It would be feasible to all for parallel processing picture 10, by setting above variables as follows:

10

	tiles_enabled_flag==0	tiles_enabled_flag==1	entropy_coding_sync_enabled_flag==1	tiles_enabled_flag==1	entropy_coding_sync_enabled_flag==1
dependent_slice_segment_flag	0/1	0/1	0/1	0/1	0/1
Picture subdivision into tiles	No	Yes	No	Yes	No
Slice subdivision with entry points to indicate start of WPP substream or Tile.	No	Yes	Yes	No	No
parallel processing possible	in slices	in Tiles	WPP	in Tiles	WPP

- 15 It would even feasible to mix WPP with tile partitioning. In that case, one could treat tiles as individual pictures: each which uses WPP would be composed of a slice having one or more dependent slice segments, and the check in step 300 and 208 would refer to the second CTB in the above CTB row in the same tile, just as steps 204 and A1 would refer to the first CTB 30 in the CTB 30 row of the current tile! In that case, above table could be extended:

	tiles_enabled_flag	tiles_enabled_flag==1	entropy_coding_sync_enabled_flag	tiles_enabled_flag==1	entropy_coding_sync_enabled_flag	entropy_coding_sync_enabled_flag	entropy_coding_sync_enabled_flag

			ag ==1		ag ==1	ag ==1	ag ==1
dependent_slice_segment_flag	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Picture subdivision into tiles	No	Yes	No	Yes	No	Yes	Yes
Slice subdivision with entry points to indicate start of WPP substream or Tile.	No	Yes	Yes	No	No	No (each substream of a tile in its own dependent slice)	Yes (each tile in a separate slice and entry points for WPP substreams within the slice of the tile)
parallel processing possible	in slices	in Tiles	WPP	in Tiles	WPP	WPP substreams within Tiles	WPP substreams within Tiles

As a brief note, the latter extension would have also been possible with embodiment 2. Embodiment 2 allows the following processing:

5

	Syntax level				
tiles_or_entropy_coding_sync_idc	per picture	0	1	2	3

	slice								
Picture subdivision into tiles		No	Yes	No	No	Yes	No	Yes	Yes
Slice subdivision with entry points to indicate start of WPP substream or Tile.		No	Yes	Yes	No	No	No	No (each substream of a tile in its own dependent slice)	Yes (each tile in a separate slice and entry points for WPP substreams within the slice of the tile)
parallel processing possible		in slices	in Tiles	WPP	Entropy coding of slices	in Tiles	WPP	WPP substreams within Tiles	WPP substreams within Tiles

With regard the above embodiments, it should be noted that the decoder could be configured to, for example, responsive to tiles_or_entropy_coding_sync_idc=1,2, in the first and second mode, read information from current slice revealing a subdivision of the current slice into parallel subsections, wherein parallel subsections could be WPP substreams or tiles, cease the context adaptive entropy decoding at the end of the first parallel subsection and resume the context adaptive entropy decoding anew at a beginning of any subsequent parallel subsection including, in the first mode, an initialization of the symbol probabilities depending on saved states of symbol probabilities of the preceding

parallel subsection and, in the second mode, an initialization of the symbol probabilities independent from any previously decoded slice and any previously decoded parallel subsection.

5

Thus, above description revealed methods for low delay encoding, decoding, encapsulation and transmission of structured video data as provided by the new HEVC coding standard, such as structured in tiles, Wavefront Parallel Processing (WPP) substreams, slices or entropy slices.

10

Especially, it has been defined how to transport parallel encoded data in a conversational scenario in order to gain minimum latency in the encoding, decoding and transmission process. Therefore it has been described a pipelined parallel coding, transmission and decoding approach in order to allow minimal delay applications like gaming, remote surgery, etc.

15

Furthermore, the above embodiments closed the gap of Wavefront Parallel Processing (WPP) to make it useable in low delay transmission scenarios. Therefore, a new encapsulation format for WPP substreams 0 has been presented, a dependent slice. This dependent slice may contain Entropy Slice data, a WPP substream, a full row of LCUs, just a fragment of a slice, where the prior transmitted slice header also applies to the contained fragment data. The contained data is signaled in the sub slice header.

20

It is finally noted, that the naming for the new slices could also be “Subset / Light weight slices”, but the name “Dependent Slice” has been found to be better.

25

A signaling has been presented which describes the level of parallelization in coding and transport.

30

Although some aspects have been described in the context of an apparatus, it is clear that these aspects also represent a description of the corresponding method, where a block or device corresponds to a method step or a feature of a method step. Analogously, aspects described in the context of a method step also represent a description of a corresponding block or item or feature of a corresponding apparatus. Some or all of the method steps may be executed by (or using) a hardware apparatus, like for example, a microprocessor, a programmable computer or an electronic circuit. In some embodiments, some one or more of the most important method steps may be executed by such an apparatus.

35

Depending on certain implementation requirements, embodiments of the invention can be implemented in hardware or in software. The implementation can be performed using a digital storage medium, for example a floppy disk, a DVD, a Blu-Ray, a CD, a ROM, a
5 PROM, an EPROM, an EEPROM or a FLASH memory, having electronically readable control signals stored thereon, which cooperate (or are capable of cooperating) with a programmable computer system such that the respective method is performed. Therefore, the digital storage medium may be computer readable.

10 Some embodiments according to the invention comprise a data carrier having electronically readable control signals, which are capable of cooperating with a programmable computer system, such that one of the methods described herein is performed.

15 Generally, embodiments of the present invention can be implemented as a computer program product with a program code, the program code being operative for performing one of the methods when the computer program product runs on a computer. The program code may for example be stored on a machine readable carrier.

20 Other embodiments comprise the computer program for performing one of the methods described herein, stored on a machine readable carrier.

In other words, an embodiment of the inventive method is, therefore, a computer program having a program code for performing one of the methods described herein, when the
25 computer program runs on a computer.

A further embodiment of the inventive methods is, therefore, a data carrier (or a digital storage medium, or a computer-readable medium) comprising, recorded thereon, the computer program for performing one of the methods described herein. The data carrier,
30 the digital storage medium or the recorded medium are typically tangible and/or non-transitory.

A further embodiment of the inventive method is, therefore, a data stream or a sequence of signals representing the computer program for performing one of the methods described
35 herein. The data stream or the sequence of signals may for example be configured to be transferred via a data communication connection, for example via the Internet.

A further embodiment comprises a processing means, for example a computer, or a programmable logic device, configured to or adapted to perform one of the methods described herein.

- 5 A further embodiment comprises a computer having installed thereon the computer program for performing one of the methods described herein.

10 A further embodiment according to the invention comprises an apparatus or a system configured to transfer (for example, electronically or optically) a computer program for performing one of the methods described herein to a receiver. The receiver may, for example, be a computer, a mobile device, a memory device or the like. The apparatus or system may, for example, comprise a file server for transferring the computer program to the receiver .

- 15 In some embodiments, a programmable logic device (for example a field programmable gate array) may be used to perform some or all of the functionalities of the methods described herein. In some embodiments, a field programmable gate array may cooperate with a microprocessor in order to perform one of the methods described herein. Generally, the methods are preferably performed by any hardware apparatus.

20 The above described embodiments are merely illustrative for the principles of the present invention. It is understood that modifications and variations of the arrangements and the details described herein will be apparent to others skilled in the art. It is the intent, therefore, to be limited only by the scope of the impending patent claims and not by the specific details presented by way of description and explanation of the embodiments herein.

25

References

- [1] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, Ajay Luthra, "Overview of the H.264/AVC Video Coding Standard", IEEE Trans. Circuits Syst. Video Technol., vol. 13, N7, July 2003.
- [2] JCT-VC, "High-Efficiency Video Coding (HEVC) text specification Working Draft 6", JCTVC-H1003, February 2012.
- [3] ISO/IEC 13818-1: MPEG-2 Systems specification.

Claims

1. Decoder for reconstructing a picture from a datastream into which the picture is coded in units of slices into which the picture is partitioned, using WPP processing, wherein the decoder is configured to decode the slices from the datastream in accordance with a slice order, wherein each slice comprises a start syntax portion indicating a position of a decoding begin of the respective slice within the picture and the decoder is configured to

identifying entry points of WPP substreams into which the slices are grouped, by identifying, using the slices' start syntax portions, slices starting at a left hand side of the picture,

parallel decoding the WPP substreams in a staggered manner with sequentially commencing the decoding of the WPP substreams in accordance with the slice order.
2. Decoder of claim 1, wherein the data stream has splitting information which is different for luma and chroma.
3. Encoder for coding a picture into a datastream into which the picture is coded in units of slices into which the picture is partitioned, using WPP processing, wherein the encoder is configured to encode the slices into the datastream in accordance with a slice order, wherein the encoder is configured to provide each slice with a start syntax portion indicating a position of a coding begin of the respective slice within the picture and the encoder is configured to

group the slices into WPP substreams so that for each WPP substream, the first slice in slice order starts at a left hand side of the picture, and

parallel encode the WPP substreams in a staggered manner with sequentially commencing the encoding of the WPP substreams in accordance with the slice order.
4. Encoder of claim 3, wherein the data stream has splitting information which is different for luma and chroma.

5. Method for reconstructing a picture from a datastream into which the picture is coded in units of slices into which the picture is partitioned, using WPP processing, wherein the method comprises decoding the slices from the datastream in accordance with a slice order, wherein each slice comprises a start syntax portion indicating a position of a decoding begin of the respective slice within the picture and the method further comprises

identifying entry points of WPP substreams into which the slices are grouped, by identifying, using the slices' start syntax portions, slices starting at a left hand side of the picture,

parallel decoding the WPP substreams in a staggered manner with sequentially commencing the decoding of the WPP substreams in accordance with the slice order.

6. Method for coding a picture into a datastream into which the picture is coded in units of slices into which the picture is partitioned, using WPP processing, wherein the method comprises encoding the slices into the datastream in accordance with a slice order, and providing each slice with a start syntax portion indicating a position of a coding begin of the respective slice within the picture and further

grouping the slices into WPP substreams so that for each WPP substream, the first slice in slice order starts at a left hand side of the picture, and

parallel encoding the WPP substreams in a staggered manner with sequentially commencing the encoding of the WPP substreams in accordance with the slice order.

7. Computer program having a program code for performing, when running on a computer, a method according to claim 5 or 6.

8. Digital storage medium storing a datastream into which a picture is coded in units of slices into which the picture is partitioned, wherein the slices are decodable, using WPP processing, from the datastream in accordance with a slice order, wherein each slice comprises a start syntax portion indicating a position of a decoding begin of the respective slice within the picture so that a decoder is able to

identify entry points of WPP substreams into which the slices are grouped, by identifying, using the slices' start syntax portions, slices starting at a left hand side of the picture,

- 5 in parallel decode the WPP substreams in a staggered manner with sequentially commencing the decoding of the WPP substreams in accordance with the slice order.
- 9. Digital storage medium of claim 8, wherein the data stream has splitting
- 10 information which is different for luma and chroma.