|  |  |
| --- | --- |
| **Joint Collaborative Team on Video Coding (JCT-VC)**  **of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11**  7th Meeting: Geneva, CH, 21-30 November, 2011 | Document: JCTVC-G318 |

|  |  |  |  |
| --- | --- | --- | --- |
| *Title:* | **Tile groups** | | |
| *Status:* | Input Document to JCT-VC | | |
| *Purpose:* | Proposal | | |
| *Author(s) or Contact(s):* | Ye-Kui Wang Ying Chen Muhammed Coban Marta Karczewicz  5775 Morehouse Dr San Diego, CA 92121 USA | Tel: Email: | 1-858-651-8345 [yekuiw@qualcomm.com](mailto:yekuiw@qualcomm.com)  1-858-845-6589 [cheny@qualcomm.com](mailto:cheny@qualcomm.com)  1-858-658-3973 [mcoban@qualcomm.com](mailto:mcoban@qualcomm.com)  1-858-658-5673 [martak@qualcomm.com](mailto:martak@qualcomm.com) |
| *Source:* | Qualcomm Incorporated | | |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Abstract

This document proposes a tile grouping mechanism, wherein tiles are assigned to different tile groups. Each tile group is identified by a tile group ID value. The decoding order of the tile groups is in the ascending order of the group ID value. Inside each tile group, the tiles are decoded in a raster scan order. Six tile group types are defined, some of which may remind readers of some slice group types in AVC. It is asserted that tile groups may be used for improved parallel processing, improved error resilience and flexible region of interest (ROI) coding. No implementation of tile groups is available. No simulation results have been provided either.

# Introduction

The tiles technique as proposed in JCTVC-F335 was adopted at the previous JCT-VC meeting.

The decoding order of tiles is the tile raster scan order. Inside each tile, the decoding order of LCUs is the LCU raster scan order inside a tile. Each tile consists of an integer number of LCUs.

This ordering of tiles is associated with the following disadvantages:

1. As shown in Fig. 1, a picture is partitioned into 8 tiles by 7 non-picture-boundary horizontal tile boundaries, and tiles are counted from top to bottom and starting from 0. Thus the decoding order of the tiles is tile 0 (the top tile), tile 1, …, tile 7. In-picture prediction, including pixel value prediction, motion prediction, coding mode prediction, and entropy coding context prediction, across tile boundaries is disallowed to enable parallel processing (encoding and decoding). Tiles 0, 2, 4, 6 are processed by one processing (encoding or decoding) core, and tiles 1, 3, 5, 7 are processed by another processing core. It would thus be ideal to put coded bits for tiles 0, 2, 4, 6 continuously in decoding order, followed by tiles 1, 3, 5, 7.

However, this parallel proceeding friendly handling of tiles is not possible with the current design of tiles.

1. A picture partitioned the same way as shown in Fig. 1. Again, in-picture prediction across tile boundaries is disallowed. The coded picture is to be transmitted in an error prone environment. If the number of coded bits for tiles 0, 2, 4, 6 does not exceed the Maximum Transmission Unit (MTU) size, and the number of coded bits for tiles 1, 3, 5, 7 does not exceed the MTU size, it would be ideal to place the coded bits of tiles 0, 2, 4, 6 into one coded slice, which can then be encapsulated into one packet, and to place the coded bits of tiles 1, 3, 5, 7 into another coded slice, which can then be encapsulated into another packet. This is ideal because if either of the two packets gets lost, the regions covered by the tiles in the lost packet can be reasonably well reconstructed by using the decoded tiles in the received packet using error concealment from correctly decoded neighboring blocks, especially when the LCU size is not big and the each tile does not contains many LCU rows.

However, this error resilient handling of tiles is not possible with the current design of tiles.

**Figure 1 An example of a picture being partitioned into 8 tiles**

**Figure 2 An example of a picture being partitioned into a region of interest and other regions by tiles**

1. As shown in Fig. 2, each picture in a sequence of pictures is partitioned into more than one tile, and a subset of the tiles (the one in the center in Fig. 2) covers the same rectangular region in all the pictures in the sequence of pictures, and the region for all the pictures can be decoded independently of other region from the same picture and other pictures. Such a region is also referred to as an independently decodable sub-picture, which can be the only region of interest (ROI) to some clients, and due to restrictions such as decoding capability and network bandwidth as well as user preferences, the clients may choose to request the transmission of only coded bits that are sufficient to decode this ROI. In this case, it is ideal to place the coded bits of the tiles covering the ROI into one slice, and the coded bits of other tiles into a different slice.

However, this ROI friendly handling of tiles is not possible with the current design of tiles.

This document proposes a tile grouping mechanism, as described in the following section, to enable the above ways for handling of tiles.

# Tile groups

## Syntax

The changed sequence parameter set syntax and picture parameter set syntax are as follows, wherein the changed parts are highlighted.

|  |  |
| --- | --- |
| seq\_parameter\_set\_rbsp( ) { | Descriptor |
| **profile\_idc** | u(8) |
| **reserved\_zero\_8bits** /\* equal to 0 \***/** | u(8) |
| **level\_idc** | u(8) |
| **seq\_parameter\_set\_id** | ue(v) |
| **...** |  |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| if (num\_tile\_columns\_minus1 != 0 || num\_tile\_rows\_minus1 != 0) { |  |
| **tile\_boundary\_independence\_idc** | u(1) |
| **uniform\_spacing\_idc** | u(1) |
| **tile\_group\_type** | ue(v) |
| if (uniform\_spacing\_idc != 1) { |  |
| for (i=0; i<num\_tile\_columns\_minus1 ; i++) |  |
| **column\_width[i]** | ue(v) |
| for (i=0; i <num\_tile\_rows\_minus1; i++) |  |
| **row\_height[i]** | ue(v) |
| } |  |
| if( tile\_group\_type >= 1 && tile\_group\_type <= 5 ) { |  |
| **num\_tile\_groups\_minus2** | ue(v) |
| if( tile\_group\_type = = 3 ) |  |
| for( i = 0; i < num\_tile\_groups\_minus2 + 1; i++ ) { |  |
| **top\_left[** i **]** | ue(v) |
| **bottom\_right[** i **]** | ue(v) |
| } |  |
| else if( tile\_group\_type = = 4 && tile\_group\_type = = 5 ) { |  |
| NumTiles = ( num\_tile\_columns\_minus1 + 1 ) \* ( num\_tile\_rows\_minus1 + 1 ) |  |
| for( i = 0, j = 0; j < NumTiles; i++ ) { |  |
| **tile\_group\_id**[ i ] | ue(v) |
| **run\_length\_minus1**[ i ] | ue(v) |
| j += run\_length\_minus1[ i ] |  |
| } |  |
| } |  |
| } |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

|  |  |
| --- | --- |
| pic\_parameter\_set\_rbsp( ) { | Descriptor |
| **pic\_parameter\_set\_id** | ue(v) |
| **seq\_parameter\_set\_id** | ue(v) |
| **entropy\_coding\_mode\_flag** | u(1) |
| **num\_ref\_idx\_l0\_default\_active\_minus1** | ue(v) |
| **num\_ref\_idx\_l1\_default\_active\_minus1** | ue(v) |
| **pic\_init\_qp\_minus26/**\* relative to 26 \*/ | se(v) |
| **constrained\_intra\_pred\_flag** | u(1) |
| **tile\_info\_present\_flag** | u(1) |
| if (tile\_info\_present\_flag == 1**) {** |  |
| **num\_tile\_columns\_minus1** | ue(v) |
| **num\_tile\_rows\_minus1** | ue(v) |
| if (num\_tile\_columns\_minus1 != 0 || num\_tile\_rows\_minus1 != 0) { |  |
| **tile\_boundary\_independence\_idc** | u(1) |
| **uniform\_spacing\_idc** | u(1) |
| **tile\_group\_type** | ue(v) |
| if (uniform\_spacing\_idc != 1) { |  |
| for (i=0; i<num\_tile\_columns\_minus1 ; i++) |  |
| **column\_width[i]** | ue(v) |
| for (i=0; i <num\_tile\_rows\_minus1; i++) |  |
| **row\_height[i]** | ue(v) |
| } |  |
| if( tile\_group\_type >= 1 && tile\_group\_type <= 5 ) { |  |
| **num\_tile\_groups\_minus2** | ue(v) |
| if( tile\_group\_type = = 3 ) |  |
| for( i = 0; i < num\_tile\_groups\_minus2 + 1; i++ ) { |  |
| **top\_left[** i **]** | ue(v) |
| **bottom\_right[** i **]** | ue(v) |
| } |  |
| else if( tile\_group\_type = = 4 && tile\_group\_type = = 5 ) { |  |
| NumTiles = (num\_tile\_columns\_minus1 + 1) \* (num\_tile\_rows\_minus1 + 1) |  |
| for( i = 0, j = 0; j < NumTiles; i++ ) { |  |
| **tile\_group\_id**[ i ] | u(v) |
| **run\_length\_minus1**[ i ] | ue(v) |
| j += run\_length\_minus1[ i ] |  |
| } |  |
| } |  |
| } |  |
| } |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

## Semantics

The semantics of the new syntax elements are as follows.

**tile\_group\_type** specifies how the mapping of tiles to tile groups is coded. The value of tile\_group\_type shall be in the range of 0 to 6, inclusive.

tile\_group\_type equal to 0 specifies that all tiles belong to one tile group, with the tile group identifier TileGroupId equal to 0. The number of tile groups, represented by the variable NumTileGroups, is derived as equal to 1.

tile\_group\_type equal to 1 specifies that each column of tiles belongs to one tile group, and the tile group identifier TileGroupId is equal to 0 for the left-most column of tiles, and increases by 1 for each column of tiles from left to right, modular NumTileGroups, which is derived as equal to num\_tile\_groups\_minus2 + 2.

tile\_group\_type equal to 2 specifies that each row of tiles belongs to one tile group, and the tile group identifier TileGroupId is equal to 0 for the top row of tiles, and increases by 1 for each row of tiles from top to bottom, modular NumTileGroups, which is derived as equal to num\_tile\_groups\_minus2 + 2.

tile\_group\_type equal to 3 specifies one or more "foreground" tile groups and the "leftover" tile group. The number of tile groups, NumTileGroups, is derived as equal to num\_tile\_groups\_minus2 + 2.

tile\_group\_type value equal to 4 or 5 specifies explicit assignment of each tile to a tile group, coded in a compact manner. The number of tile groups, NumTileGroups, is derived as equal to num\_tile\_groups\_minus2 + 2.

tile\_group\_type value equal to 6 specifies a checkerboard like assignment of tiles to two tile groups. The number of tile groups, represented by the variable NumTileGroups, is derived as equal to 2. The top-left tile is assigned to tile group 0. For any tile of tile group 0, a neighboring tile to the left, right, upper or lower, if present, is assigned to tile group 1. For any tile of tile group 1, a neighboring tile to the left, right, upper or lower, if present, is assigned to tile group 0.

**num\_tiles\_groups\_minus2** plus 2 specifies the number of tile groups for a picture. This syntax element is only present when tile\_group\_type is in the range of 1 to 5, inclusive. The value of this syntax element shall be in the range of 0 to 16, inclusive.

**top\_left[** i **]** and **bottom\_right[** i **]** specify the top-left and bottom-right corners of a rectangle, respectively. top\_left[ i ] and bottom\_right[ i ] are tile positions in the picture in tile raster scan order. For each rectangle i, all of the following constraints shall be obeyed by the values of the syntax elements top\_left[ i ] and bottom\_right[ i ]:

– top\_left[ i ] shall be less than or equal to bottom\_right[ i ] and bottom\_right[ i ] shall be less than NumTiles that is equal to ( num\_tile\_columns\_minus1 + 1 ) \* ( num\_tile\_rows\_minus1 + 1 ).

– ( top\_left[ i ] % num\_tile\_columns\_minus1 + 1 ) shall be less than or equal to the value of ( bottom\_right[ i ] % num\_tile\_columns\_minus1 + 1 ).

NOTE – The rectangles may overlap. Tile group 0 contains the tiles that are within the rectangle specified by top\_left[ 0 ] and bottom\_right[ 0 ]. A tile group having tile group ID greater than 0 and less than NumTileGroups – 1 contains the tiles that are within the specified rectangle for that tile group that are not within the rectangle specified for any tile group having a smaller tile group ID. The tile group with tile group ID equal to NumTileGroups – 1 contains the tiles that are not in the other tile groups.

**tile\_group\_id[** i **]** identifies a tile group. The length of the tile\_group\_id[ i ] syntax element is Ceil( Log2( num\_tile\_groups\_minus2 + 2 ) ) bits. The value of tile\_group\_id[ i ] shall be in the range of 0 to num\_tile\_groups\_minus2 + 2, inclusive.

**run\_length\_minus1[**i**]** is used to specify the number of consecutive tiles to be assigned to the tile group identified by tile\_group\_id[ i ]. The value of run\_length\_minus1[ i ] shall be in the range of 0 to NumTiles− 1, inclusive.

## Decoding process for tile to tile group map

Inputs to this process are the active sequence parameter set and the active picture parameter set.

Output of this process is a tile to tile group map TileGroupMap, which consists of NumTiles values, each corresponding to the tile group ID value of one tile, indexed in tile raster scan order.

This process is invoked at the start of every slice.

NOTE – The output of this process is equal for all slices of a picture.

The tile to tile group map TileGroupMap is derived by the following pseudo-code:

numRows = num\_tiles\_rows\_minus1 + 1  
numCols = num\_tiles\_columns\_minus1 + 1  
if( tile\_group\_type = = 0)  
 for( i = 0; i < NumTiles; i++ )  
 TileGroupMap[ i ] = 0  
else if(tile\_group\_type = = 1)  
 for( y = 0; y < numRows ; y++ )  
 for( x = 0; x < numCols; x++ )  
 TileGroupMap[ y \* numCols + x ] = x % NumTileGroups  
else if(tile\_group\_type = = 2)  
 for( y = 0; y < numRows; y++ )  
 for( x = 0; x < numCols; x++ )  
 TileGroupMap[ y \* numCols + x ] = y % NumTileGroups  
else if(tile\_group\_type = = 3) {  
 for( i = 0; i < NumTiles ; i++ )  
 TileGroupMap[ i ] = NumTileGroups − 1  
 for( i = NumTileGroups − 2; i >= 0; i− − ) {  
 yTopLeft = top\_left[ i ] / numCols  
 xTopLeft = top\_left[ i ] % numCols  
 yBottomRight = bottom\_right[ i ] / numCols  
 xBottomRight = bottom\_right[ i ] % numCols  
 for( y = yTopLeft; y <= yBottomRight; y++ )  
 for( x = xTopLeft; x <= xBottomRight; x++ )  
 TileGroupMap[ y \* numCols + x ] = i  
 }  
}  
else if( tile\_group\_type = = 4 | | tile\_group\_type = = 5 ) {  
 for( i = 0, j = 0; j < NumTiles; i++ )   
 for( k = 0; k <= run\_length\_minus1[ i ]; k++, j++ )   
 tileGroupMap[ j ] = tile\_group\_id[ i ]  
  
 if( tile\_group\_type = = 4 )  
 for( i = 0; i < NumTiles; i++ ) {  
 y = i / numCols  
 x = i % numCols  
 TileGroupMap[ i ] = tileGroupMap[ x \* numRows + y ]  
 }  
 else // tile\_group\_type = = 5  
 for( i = 0; i < NumTiles; i++ )   
 TileGroupMap[ i ] = tileGroupMap[ i ]  
}  
else // tile\_group\_type = = 6  
 for( y = 0; y < numRows; y++ )  
 for( x = 0; x < numCols ; x++ )  
 TileGroupMap[ y \* numCols + x ] = (y % 2 ) ? (x+1) % 2 : x % 2

## Examples of tile groups

Some examples of tile groups of tile grouping types 1, 2, 3 and 6 are shown in the figures below.

**Figure 3 8 tiles, 2 tile groups, tile\_group\_type = 2**

**Figure 4 9 tiles, 3 tile groups, tile\_group\_type = 2**

**Figure 5 8 tiles, 2 tile groups, tile\_group\_type = 1**

**Figure 6 9 tiles, 3 tile groups, tile\_group\_type = 1**

**Figure 7 9 tiles, 2 tile groups, tile\_group\_type = 3**

**Figure 8 12x9 tiles, 2 tile groups, tile\_group\_type = 6**

# Patent rights declaration(s)

**Qualcomm Incorporated may have current or pending patent rights relating to the technology described in this contribution and, conditioned on reciprocity, is prepared to grant licenses under reasonable and non-discriminatory terms as necessary for implementation of the resulting ITU-T Recommendation | ISO/IEC International Standard (per box 2 of the ITU-T/ITU-R/ISO/IEC patent statement and licensing declaration form).**