

# mpegable Video SDK

MPEG-4 Video

Software Developers Kit

Last Update: 2007-07-22

Software Version: 1.24

© 2001-2007, **dicas** digital image coding GmbH

<http://www.dicas.de>



---

## Contents

<b>1</b>	<b>mpegable Video SDK</b>	<b>5</b>
<b>2</b>	<b>Input and Output formats</b>	<b>5</b>
2.1	Video data formats YUV 4:2:0 and YUV 4:2:2 . . . . .	6
2.2	Video data formats RGB and related . . . . .	6
<b>3</b>	<b>MPEG-4 Video Profiles</b>	<b>7</b>
<b>4</b>	<b>C++ Class Overview</b>	<b>7</b>
<b>5</b>	<b>Input and Output Streams</b>	<b>8</b>
5.1	Push data to input pin . . . . .	9
5.2	Pull data from output pin . . . . .	10
<b>6</b>	<b>Control Parameters for the Codec</b>	<b>11</b>
6.1	Attach input and output pins . . . . .	12
6.2	Parameters for the encoder . . . . .	13
6.2.1	Media format . . . . .	13
6.2.2	Frame width and height . . . . .	14
6.2.3	De-interlacing . . . . .	14
6.2.4	Resizing . . . . .	14
6.2.5	Decimation . . . . .	15
6.2.6	Pixel aspect ratio . . . . .	16
6.2.7	Frame rate . . . . .	17
6.2.8	Key-frame period . . . . .	18
6.2.9	B-frames . . . . .	18

---

6.2.10	Texture quantisation . . . . .	19
6.2.11	Bit-rate . . . . .	19
6.2.12	Frame skip probability . . . . .	20
6.2.13	VBV parameters . . . . .	21
6.2.14	User defined quantisation tables . . . . .	21
6.2.15	Error resilience . . . . .	22
6.2.16	GOV header period . . . . .	23
6.2.17	Motion estimation . . . . .	23
6.2.18	Interlaced Coding - ASP Level 4/5 . . . . .	24
6.2.19	High complexity mode . . . . .	25
6.2.20	MPEG-4 Version . . . . .	25
6.2.21	Short header mode . . . . .	25
6.3	Parameters for the decoder . . . . .	26
6.3.1	Media format . . . . .	26
6.3.2	Video object header . . . . .	27
6.3.3	DecoderSpecificInfo header information . . . . .	27
6.3.4	Post filter . . . . .	28
6.3.5	Resize . . . . .	29
6.4	Initializing the codec . . . . .	29
6.4.1	Video object header . . . . .	30
6.5	Error codes during initialization . . . . .	30
6.6	Configure hardware support . . . . .	31
<b>7</b>	<b>Processing the Data</b>	<b>31</b>
<b>8</b>	<b>Payload descriptor</b>	<b>32</b>

---

<b>9 H.263 Coding</b>	<b>33</b>
9.1 Limitations of the H.263 Baseline Profile . . . . .	34
9.2 New tools in H.263 Profile 3 . . . . .	34
<b>10 Codec Listener</b>	<b>35</b>
10.1 The VOPEncoderEvent class . . . . .	36
10.1.1 VOP statistics . . . . .	38
10.2 The VOPDecoderEvent class . . . . .	42
<b>11 Manual controls during encoding</b>	<b>44</b>
11.1 Key frames . . . . .	44
11.2 Frame skip . . . . .	44
<b>12 Using time stamps</b>	<b>45</b>
12.1 Manually setting of time stamps . . . . .	45
12.2 Time stamps and bitrate-control . . . . .	46
<b>13 Two-Pass Encoding</b>	<b>47</b>
13.1 General workflow in two-pass mode . . . . .	47
13.2 API Description . . . . .	49

## 1 mpegable Video SDK

The mpegable™ Video SDK enables application developers to use real-time MPEG-4 video encoding and decoding functionality for their Windows based digital video applications.

The SDK includes a dynamic link library (DLL) for MS Windows, a static library and the appropriate C++ header files to link your application with mpegable™. This documentation describes all classes and functions provided by the SDK. A simple example of an encoder application demonstrates usage of the mpegable™ Video SDK.

The mpegable™ MPEG-4 technology combines state-of-the-art video data compression with a very high processing speed on usual PC hardware. mpegable is available for Microsoft™ Windows (9x, NT 4.0, 2000, XP), for Linux (x86), for Mac OS X and for Sun Solaris.

mpegable™ is fully compliant to ISO/IEC 14496-2 Simple Profile and Advanced Simple Profile as well as to ITU-T Recommendation H.263 Profile 0 (Baseline). Furthermore, the encoder supports the Profile 3 (Interactive and streaming wireless) of H.263.

## 2 Input and Output formats

The encoder functions of mpegable™ accept different formats for the uncompressed video data. The preferred input formats are YUV 4:2:0, or YUVA for an additional alpha mask. Other supported formats are YUV 4:2:2, BGR (24 Bit), BGRA (32 Bit), RGB (24 Bit) and RGBA (32 Bit).

The frame width and frame height must be a multiple of 2. The input data are given to the encoder in data units, each of them representing a single frame.

The output data of the encoder is a MPEG-4 video stream (elementary stream). The output of the encoder is organised such that each data unit represents one encoded frame.

The decoder expects a MPEG-4 video stream. Each data unit pushed to the decoder must represent one complete access unit (frame). The decoder's output units are uncompressed frames. The output format can be one of YUV 4:2:0,

YUVA, YUV 4:2:2, BGR, BGRA, RGB, RGBA. The preferred format is YUV 4:2:0. When using an output format that has an alpha channel (YUVA, BGRA, RGBA), this component is not used.

## 2.1 Video data formats YUV 4:2:0 and YUV 4:2:2

The YUV format is a family of video formats where each frame is separated in three planes, one of them representing the luminance of each pixel (Y-plane) and the other two representing the chrominance (U- and V-plane). Each component of a pixel is represented by an 8 bit integer.

In the YUV 4:2:0 format, the chrominance components are downsampled in both, horizontal and vertical direction so U and V plane resolution is half of the Y plane. The Y, U and V planes are written one after the other in that order.

The planes are raw pixel bytes written row-by-row, top left to bottom right. If the image size is  $width \cdot height$ , then the size of the Y plane is  $width \cdot height$  bytes and the U/V planes are each  $width \cdot height/4$  bytes. The total amount of bytes for one frame is  $3 \cdot width \cdot height/2$ . There is no header information present, so frame size must be known in advance.

Additionally to the planar YUV 4:2:0 format, the codec also supports packed YUV 4:2:2 where luma and chrominance data are interleaved. The total amount of bytes for one YUV 4:2:2 frame is  $2 \cdot width \cdot height$ .

Internally, the MPEG-4 format uses the YUV 4:2:0 format. Any other input or output format is therefore converted to or from YUV 4:2:0 by mpegable™ Video SDK.

## 2.2 Video data formats RGB and related

The RGB formats also store three components for every pixel, representing the red, green and blue component. The three components for one pixel are written together. So, RGB 24 means that there are 24 bit (3 byte) for every pixel. This is 8 Bit for the red, green and blue component of one pixel. The RGBA32 takes 32 bit for every pixel reserving the last 8 bit for an alpha channel.

mpegable™ also supports the BGR and BGRA formats that are the same as RGB

and RGBA but with interchanged byte positions for the blue and red component.

### 3 MPEG-4 Video Profiles

The MPEG-4 standard is meant as a large toolbox that offers a common format for a wide range of applications. To avoid the danger of overloading the standard, the MPEG adopted the profile approach. A profile is a group of technologies capable of satisfying the needs for different applications.

The base for all profiles in the video part of the MPEG-4 standard is the *simple profile* that includes coding of rectangular video frames with I- and P-frames. The Advanced Simple Profile supports some more tools to improve the compression efficiency like bi-directionally predicted (B-) frames, quarter pixel estimation and global motion compensation. The *core profile* supports coding of arbitrarily shaped objects and B-frame coding as well as some further features. mpegable™ supports the MPEG-4 Video Simple Profile and Advanced Simple Profile. For support of Core and Main profile, please ask the dicas team for further assistance.

### 4 C++ Class Overview

The mpegable™ Video SDK enables encoding and decoding of MPEG-4 video data by providing several C++ classes.

**Mpeg4CodecIf:** An interface to the codec itself. Encoding and decoding is done by a call to an object's method.

**Mpeg4VidEncoderCtrlIf:** An interface to all parameters of the codec that controls encoding.

**Mpeg4VidDecoderCtrlIf:** An interface to all parameters of the codec that controls decoding.

**CBufferElem:** Any instance of this class represents a data unit that can be stored in a buffer queue. This class provides access to the stored data as well as to an additional payload descriptor.

**CInStrmPin:** This class provides functionality of a buffer queue to receive data units and buffer them until encoder or decoder are ready to process them.

**COutStrmPin:** Encoder and decoder write their output data to an instance of this class where the program can pull them out and receive the data.

**CVidStreamPayloadDesc:** Every data unit has a pointer to an object of this class. Informations such as frame size, composition time, frame type and other useful informations are stored in this class.

**MP4CodecListenerIf:** An application can derive its own classes from this abstract base class to receive events sent by the codec.

**MP4CodecEvent** The codec notifies the application in case certain events occur. These events are represented by objects of classes derived from MP4CodecEvent.

## 5 Input and Output Streams

A typical application for video coding has three parts that work in parallel: capturing video data, encoding the data and rendering it to the screen or writing it to a file or network device. As mpegable™ is prepared for real-time applications, it realizes a thread-safe concept of coding data in a one thread while input and output of the data is possibly done by other threads.

To use the codec, one has to create an object of the class CInStrmPin and another object of COutStrmPin, first. This must be done by a call to the static method createInstance().

```
CInStrmPin* pInPin = CInStrmPin::createInstance();  
COutStrmPin* pOutPin = COutStrmPin::createInstance();
```

You cannot create a CInStrmPin or COutStrmPin object by a direct call to its constructor. At the end, the application must free this objects by the static method destroyInstance().

```
CInStrmPin::destroyInstance(pInPin);  
COutStrmPin::destroyInstance(pOutPin);
```

## 5.1 Push data to input pin

Once you have created an object of `CInStrmPin`, you can push data units to its buffer queue. To do this, you have to request for a pointer to a data unit. Use the static method `allocateBufferElem()` or `getBufferElem()` and give the required size of the buffer in bytes as the second parameter.

```
CBufferElem *pUnit = NULL;
UInt32 uiYUVFrameSize = 352*288; // cif

bOk = CInStrmPin::allocateBufferElem(&pUnit, uiYUVFrameSize);
```

In the case of success, `pUnit` points to a `CBufferElem`'s object. To data unit gives access to a payload descriptor (see section 8) and to a pointer to the allocated memory.

In contrast to `allocateBufferElem()`, the `getBufferElem()` method does not automatically allocates new memory but tries to reuse memory from a buffer element that has been released before.

```
UInt8 *puiData = NULL;
UInt32 *uiDataSize = 0;
bOk = CInStrmPin::getData(pUnit, puiData, uiDataSize);
```

After the call of `getData()`, `puiData` points to the reserved memory block and `uiDataSize` gives the size of this memory block. If everything worked well this equals the value of the parameter previously given to the `allocateBufferElem()` method. If `puiDataSrc` is a pointer to the source data, simply copy to the reserved memory by

```
memcpy (puiData, puiDataSrc, uiDataSize);
```

The last step is to push the unit to the input pin

```
bOk = pInPin->push(pUnit);
```

After this push, the pUnit pointer is undefined and must not be destroyed.

After the last data unit has been pushed to the input stream pin, call the `setEndOfPin()` method to signalize the codec the end of the stream.

```
bOk = pInPin->setEndOfStream();
```

## 5.2 Pull data from output pin

The output pin works very similar to the input pin. Supposed that pOutPin points to a COutStrmPin's object still containing some data units. To get a data unit of this pin call the `pull()` method.

```
CBufferElem *pUnit = NULL;  
bOk = pOutPin->pull(pUnit);
```

If this call was successful pUnit points to a CBufferElem's object. To get a pointer to the data again use the `getData()` method.

```
UInt8 *puiData = NULL;  
UInt32 *uiDataSize = 0;  
bOk = CInStrmPin::getData(pUnit, puiData, uiDataSize);
```

After the call of `getData()`, puiData points to the reserved memory block and uiDataSize gives the size of this memory block. Maybe you simply want to write the data to a file, this is done by

```
fwrite(puiData, sizeof(UInt8), uiDataSize, f);
```

Finally the program has to free the memory allocated by the object pUnit points to. Do not use the destructor of the class CBufferElem but the static methods `freeBufferElem()` or `releaseBufferElem()` of the class COutStrmPin.

```
COutStrmPin::releaseBufferElem(pUnit);
```

The application must not release the memory which pData points to. This memory is released automatically by calling `freeBufferElem()`, or will be reused by the buffer queue later on.

---

## 6 Control Parameters for the Codec

Many of the parameters of the MPEG-4 video standard can be set individually in mpegable™. Some of them have to be fixed before starting the coding process, others can be changed also during the coding process. The parameter setting can be done by methods provided by the classes `Mpeg4VideoEncoderCtrlIf` or `Mpeg4VideoDecoderCtrlIf`. To access these functions, an object of the class `Mpeg4CodecIf` must be created first. This object provides the function `getCodecCtrl()` method that returns a pointer to an object of the class `Mpeg4VideoEncoderCtrlIf`, or `Mpeg4VideoDecoderCtrlIf` respectively. These two classes allow the application to set all parameters for the coding process.

The following example demonstrates the creation of the required objects on the encoder side.

```
Mpeg4CodecIf* pCodec = NULL;
Mpeg4VideoEncoderCtrlIf* pCtrl = NULL;

// create codec object
pCodec = Mpeg4CodecIf::createInstance();

// get pointer to the codec control object
// and decide for encoder or decoder functionality
pCtrl = pCodec->getCodecCtrl(MPEG4V_COMPRESSOR);

// set all parameters
....

// destroy codec object
Mpeg4CodecIf::destroyInstance(pCodec);
```

To set parameters to the encoder, use the methods of the class `Mpeg4VideoEncoderCtrlIf`. The implementation for the decoder is analogous.

If all parameters have been set, the codec must be initialized before processing the data. Some of the parameters cannot be changed after the codec has been initialized.

Please note that no object can be created by a direct call to its constructor via

the C++ new operator but only by a call to the static method provided by the class. The same is true for destroying an object. Do not use delete but use the appropriate static method.

## 6.1 Attach input and output pins

The encoder as well as the decoder have a buffered data input and output. The appropriate objects have to be created and attached to the codec control object.

```
CInStrmPin* pInPin = CInStrmPin::createInstance();
COutStrmPin* pOutPin = COutStrmPin::createInstance();

pCtrl->attachInput(pInPin);
pCtrl->attachOutput(pOutPin);
```

Note that the application is still owner of the pointers pInPin and pOutPin and is responsible to free this objects by calling the static function destroyInstance() after the coding process is finished.

It is also possible to get the encoded and reconstructed images from the encoder. To use this feature, attach another output pin to the codec. Use the optional second parameter of attachOutput() method to communicate the codec that it has to generate the reconstructed frames and to output them to this pin. To generate the reconstructed images call the codec control's setDumpRecOutput () method. You can also use this method to switch on and off the creation of the reconstructed images dynamically after the initialization.

Since version 1.44, it is also possible to get the original frames from the encoder. Again, you must first attach another output pin. To activate the feature call the codec control's setDumpOrigOutput () method. The encoder only outputs frames that are not skipped by the encoder.

```
COutStrmPin* pRecOutPin = COutStrmPin::createInstance();
COutStrmPin* pOrigOutPin = COutStrmPin::createInstance();

// get encoded and reconstructed frames
pCtrl->attachOutput(pRecOutPin, 1);
```

```
pCtrl->setDumpRecOutput(TRUE);

// get original frames if not skipped
pCtrl->attachOutput(pOrigOutPin, 2);
pCtrl->setDumpOrigOutput(TRUE);
```

## 6.2 Parameters for the encoder

### 6.2.1 Media format

The encoder supports different data formats for input data. These formats are defined in the header file `mediaformat.h`. Internally, the codec always works with YUV 4:2:0 or YUVA respectively. If possible use these formats to avoid performance loss by the necessary color conversions.

The input format can be set by a call to the `setInputFormat()` method.

```
Int32 iInputFormat = MF_VID_BGR24;    // for 24 Bit RGB

bOk = pCtrl->setInputFormat(iInputFormat, bIsBottomUp);
```

When using the optional output of reconstructed images or the original images it is also possible to set their media format by a call to the `setRecOutputFormat()`, or to `setOrigOutputFormat()` respectively.

The methods have an optional second parameter that can be `TRUE` or `FALSE`. If set to `TRUE` the function assumes the first bytes of the frame to represent the bottom line of the image. If set to `FALSE` the frame starts at the top of the image.

```
Int32 iRecFormat = MF_VID_YUV420;
Bool bIsBottomUp = TRUE;
```

```
bOk = pCtrl->setRecOutputFormat(iRecFormat, bIsBottomUp);
```

This version of mpegable™ Video SDK supports the following media formats:

```
MF_VID_YUV420, MF_VID_YUVA420, MF_VID_YUYV, MF_VID_UYVY,
MF_VID_RGB24, MF_VID_RGBA32, MF_VID_BGR24, MF_VID_BGRA32
```

### 6.2.2 Frame width and height

The parameter for frame resolution in horizontal and vertical direction must be set by the `setImageSize()` method

```
Int32 iWidth   = 176;    // qcif resolution
Int32 iHeight  = 144;
```

```
bOk = pCtrl->setImageSize(iWidth, iHeight);
```

Horizontal and vertical frame resolution have to be a multiple of 2. For some media formats other than YUV 4:2:0, the horizontal frame resolution should be a multiple of 8.

The minimum resolution is 22x18, the maximum resolution is 2048x2048.

The return value is TRUE if the codec is still uninitialized and the settings for width and height are allowed values. Otherwise the return value is FALSE.

### 6.2.3 De-interlacing

To process interlaced video mpegable™ Video SDK includes a de-interlacing filter to pre-process interlaced video material. Enabling this option can substantially help to reduce the bit-rate and to improve the picture quality.

```
Int32 iMode = 1;
```

```
pCtrl->setDeinterlace(iMode);
```

The setting `iMode = 1` enables the de-interlacing filter and `iMode = 0` (default) disables it. Use the filter only for interlaced input video data.

The de-interlacing filters always expects both fields to be in the same input unit.

### 6.2.4 Resizing

As an optional pre-processing step, the mpegable™ Video SDK allows to down-sample the dimension of the input video. The new video size can be of any width

and height that is a multiple of two and that is smaller than the actual input size.

For reasons of quality and processing speed, there are different options how the re-sizing is performed. There are two different interpolation modes, bi-linear (mode 2) and bi-cubic (mode 3). For most of the applications, we recommend to use the bi-linear interpolation mode (mode 2).

The re-sizing algorithm also knows two different quality modes where the high quality mode is slower but reaches a higher visual quality when the image scaling factor is greater or equal than 2.

Example: The input dimension is 720x576 and the target dimension after resizing is 320x240. Since the resizing factor is greater than 2 the high quality mode yields a higher visual quality. This example also demonstrates that the resizing may change the aspect ratio of the video. To avoid potential artefacts, a decoder must be informed about the scaling and must perform the necessary steps to rescale the image according to the display device.

```
Int32 iNewWidth  = 320;
Int32 iNewHeight = 240;
Int32 iMode      = 2;           // bi-linear
Int32 bHighQuality = FALSE;    // fast mode

bOk = pCtrl->setResize( iNewWidth, iNewHeight,
                       iMode, bHighQuality)
```

The resizing may change the ratio of the number of horizontal pixels and the number of vertical pixels of the video. To keep the original image aspect ratio we can re-define the pixel's aspect ratio not to be square anymore. The MPEG-4 video elementary stream syntax knows a respective field that can be set by the mpegable™ video SDK. The respective function is called `setPixelAspectRatio()` described below.

### 6.2.5 Decimation

For the special case that the input image has a PAL (720x576) or a NTSC (720x480) resolution, the mpegable Video SDK™ provides an option for a high-quality decimation to some common resolutions. To be more precise, the image

can be horizontally decimated from 720 pixels to one of 544, 528, 480 or 352. It can be decimated horizontally and vertically together from 720x576/480 to 352x288 (CIF), 320x240 (QVGA), 192x192 (HDPIP), 176x144 (QCIF) and 96x96 (PIP).

```
Bool bPerformDecimate = TRUE;
Int32 iNewWidth  = 352;  // PAL/2 resolution
Int32 iNewHeight = 576;

bOk = pCtrl->setDecimation (bPerformDecimate, iNewWidth, iNewHeight);
```

The values of the new width and height must be one of the above mentioned resolutions. The resolution of the input image must be 720x576 or 720x480. The decimation method must be called before initialization of the codec.

### 6.2.6 Pixel aspect ratio

To code the pixel aspect ratio use the method `setPixelAspectRatio()`. This method sets the pixel aspect ratio that is coded in the bitstream. The MPEG-4 parameter is defined in table 6-12 of ISO/IEC 14496-2. Please note that the pixel aspect ratio has no impact on the actual coding process. It gives the decoder an indication how to display the reconstructed video image.

The values for the pixel width and height must be between 1 and 255.

```
UInt32 uiPixelWidth = 1;
UInt32 uiPixelHight = 1;  // square pixel

bOk = setPixelAspectRatio (uiPixelWidth, uiPixelHight);
```

Typical values for the pixel width and height are:

- 1:1 (square pixel)
- 12:11 (625 type for 4:3 pictures)
- 10:11 (525 type for 4:3 pictures)
- 16:11 (625 type stretched for 16:9 pictures)
- 40:33 (525 type stretched for 16:9 pictures)

The default value for the pixel aspect ratio is 1:1 (square pixel).

The function return TRUE if the parameters are valid number and the codec ist still not initialized, it returns FALSE otherwise.

### 6.2.7 Frame rate

In previous versions, the frame rate was set by a call to the `setFrameRate()` method. This function is now supported only for backward compatibility. It is recommended to use the methods `setTimeResolution()` and `setVOPTimeIncrement()` instead.

A call to `setTimeResolution()` defines the number of ticks per second of the codec's internal clock. The `setVOPTimeIncrement()` method sets the constant time difference between two successive frames measured in ticks.

Let  $dFrameRate$  denote the frame rate measured in frames per second, and let  $iTimeResolution$  the internal clock rate defining the number of clock ticks per second and  $iVOPTimeIncrement$  the time difference between two successive frames measured in clock ticks. The following formula applies

$$dFrameRate = \frac{iTimeResolution}{iVOPTimeIncrement}$$

If, for example, the frame rate is  $dFrameRate = 14.96$  fps, set  $iTimeResolution = 374$  and  $iVOPTimeIncrement = 25$ .

```
Int32 iTimeResolution = 1000;
Int32 iVOPTimeIncrement = 40;    // 25 fps

Double dFrameRate = (Double)iTimeResolution / iVOPTimeIncrement;

bOk = pCtrl->setTimeResolution(iTimeResolution);
bOk = pCtrl->setVOPTimeIncrement(iVOPTimeIncrement);
```

The return value are TRUE if the codec is still uninitialized the setting for the time resolution and the time increment are allowed values, otherwise the return value is FALSE.

mpegable™ also allows the use of individual time stamps for each frame. Please refer to section 12 to enable this mode, the setting of `iVOPTimeIncrement` must be 0. Every data unit pushed to the encoder's input pin must have a well-defined composition time. The composition time can be set in the descriptor of the data unit by the `setVOPCompositionTime()` method.

### 6.2.8 Key-frame period

mpegable™ currently works with a constant Intra-frame (Key-frame) period that can be set with the `setKeyFramePeriod` method.

```
Int32 iKeyFramePeriod = 12;  
  
bOk = pCtrl->setKeyFramePeriod(iKeyFramePeriod);
```

If no B-frames are present the parameter given to the function exactly defines the distance between two key-frames. A setting of e.g. `iKeyFramePeriod = 3` and no additional B-frames, therefore means a sequence format IPPIPPIP...

It is possible to have no P-frames by setting `iKeyFramePeriod = 1` or to have only one first I-frame and then only P-frames and B-frames by setting `iKeyFramePeriod = 0`.

The return value is `TRUE` if the codec is still uninitialized and `FALSE` otherwise.

### 6.2.9 B-frames

The core profile of MPEG-4 video allows the use of bi-directionally predicted frames (B-frames). The number of successive B-frames can be set with the `setBFramePeriod` method.

```
Int32 iBCount = 2;  
  
bOk = pCtrl->setBFramePeriod(iBCount);
```

The return value is `TRUE` if the codec is still uninitialized and `FALSE` otherwise.

The maximum number of successive B-frames is not limited to a maximum of 5. Usually, you should not use a value other than 1, 2, or 0 respectively for no use of B-frames.

Please note that the use of B-frames, together with the setting of the key-frame period, determines the sequence format. A setting of e.g. `iKeyFramePeriod = 12` and `iBCount = 2` means a sequence format `IBBPBBPBBPBBIBBPBBPB... .` If `iKeyFramePeriod` is not a multiple of  $(iBCount + 1)$  the codec uses the nearest suitable key-frame period smaller than `iKeyFramePeriod`.

### 6.2.10 Texture quantisation

The default mode of the `mpegable™` encoder functions is a constant quantisation mode. In this mode the codec uses a constant quantisation step size for every block and every frame. As the video content may change the amount of bits can vary widely.

The quantisation parameters are in the range from 1 (nearly lossless quality) to 31 (extremely low bit-rate and bad quality). MPEG-4 allows the application to set this parameter separately for I-, P- and B-frames. These three parameters can be set by the method `setTextureQuant()`.

```
Int32 iQuantI = 6
Int32 iQuantP = 8;
Int32 iQuantB = 8;
```

```
bOk = pCtrl->setTextureQuant(iQuantI, iQuantP, iQuantB);
```

The return value is `TRUE` if all settings for the quantisation parameters are allowed values. The return value is `FALSE` otherwise.

Since version 1.2, `mpegable™` allows to change the texture quantisation during the encoding process.

### 6.2.11 Bit-rate

Fixing the quantisation step size yields a undetermined bit-rate. As most users wish to fix a target bit-rate, `mpegable™` also supports bit-rate orientated encod-

ing modes. To set a target bit-rate use the `setBitrate()` method. The parameter fixes the amount of bits per second for the encoded bitstream.

```
Int32 iBitsPerSecond = 800000; // 800 kbps

bOk = pCtrl->setBitrate (iBitsPerSecond);
```

The bit-rate cannot be smaller than 15,000 bits per second and must not exceed 30 Mbits per second. The return value is `TRUE` if the parameter is in between these values and the codec is still uninitialized. If the codec already is initialized and a bit-rate orientated mode has been selected, the bit-rate will change dynamically and the return value is `TRUE`. If the codec is initialized but works in the constant quantisation mode the return value is `FALSE`.

### 6.2.12 Frame skip probability

To adapt the bit-rate to video material of different complexity, the codec controls the quantisations step-size and the frame-rate. While a high quantisation step size yields a poor picture quality, too many dropped frames give a jerky video. mpegable™ allows the application to decide on the preferred behaviour. To communicate the codec about how often frame shall be skipped, mpegable™ let the application define a frame skip probability. In the end, the codec itself decides whether to skip a frame or not, but compare with 11.2. The skip probability is a parameter value between 0.0 and 1.0 where 0.0 means that no frames will be skipped while 1.0 gives the highest possible skip probability. The default value is 0.5.

```
Double dFrameSkipProbability = 0.2;

bOk = pCtrl->setSkipProbability (dFrameSkipProbability);
```

5The possible values of `iRateControlType` have the following meanings:

### 6.2.13 VBV parameters

The video buffering verifier (VBV) is a virtual model that assumes a scenario where a video decoder receives the data with an constant bitrate. The decoder buffers the input data up to a maximum buffer size. An encoder that implements the VBV model as defined by the standard guarantees that the decoder buffer will never underflow.

A larger buffer size allows the encoder to vary the bitrate and to keep the visual quality more constant while a smaller buffer reduces the delay the decoder needs before playback can start. The best choice for the maximum buffer size depends on the application and the target (maximum) bitrate.

Example: For a streaming applications a delay of 2 seconds might be acceptable. We chose the buffer size to  $uiVbvBufferSize = uiBitrate * 2$ . The second parameter gives the initial fulness of the buffer (seen on the encoder side). A reasonable value for the initial buffer occupancy is half the maximum buffer size.

For an offline application a larger buffer of 10 times the bitrate, or even more, is recommended since the delay does not matter for file playback.

Both parameters of the `setVBVBufferParam()` method are measured in bits.

```
UInt32 uiVbvBufferSize = 20000;
UInt32 uiVbvBufferOccupancy = uiVbvBufferSize / 2;

bOk = pCtrl->setVBVBufferParam (uiVbvBufferSize,
                               uiVbvBufferOccupancy);
```

The return value is TRUE if all settings are allowed values and the codec is still uninitialized, otherwise the return value is FALSE.

### 6.2.14 User defined quantisation tables

Beside the definition of a quantisation step size, the MPEG-4 standards allows to code an user defined quantisation table. `mpegable™` supports this feature since version 1.2. The `setQuantMethod()` method chooses the quantisation type. Allowed parameters values are 0 or 1. If the quantisation mode equals 1 it is possible to set the quantisation table for the encoder. The `setIntraQuantizerMatrix()`

method defines the quantisation matrix for intra frames while the `setInterQuantizerMatrix()` method defines it for inter frames.

```
Int32 iQuantMethod = 1; // 0 = H.263, 1 = MPEG
Int32 piIntraQuantizerMatrix[64];
Int32 piInterQuantizerMatrix[64];

// fill the intra and inter table with quantisation coefficients
....

bOk = setQuantMethod (iQuantMethod);
bOk = setIntraQuantizerMatrix (piIntraQuantizerMatrix);
bOk = setInterQuantizerMatrix (piInterQuantizerMatrix);
```

The return value is `TRUE` if all settings are allowed values and the codec is still uninitialized, and `FALSE` otherwise.

### 6.2.15 Error resilience

The MPEG-4 standard provides several tools for error resilience in error prone environments. These tools allow a suitable decoder the resynchronisation, error detection, data recovery and error concealment. `mpegable™` gives developers the possibility to use these features by including the additional information during the encoding process. To divide a frame into more than one video packet use the `setVideoPacket()` method. To enable data partitioning mode and additionally the RVLC coding use the `setDataPartitioning()` method.

```
UInt32 uiVideoPacketLen = 1400;
Bool bUseRVLC = TRUE;
Bool bUseDataPartition = TRUE;

bOk = pCtrl->setVideoPacket(uiVideoPacketLen);
bOk = pCtrl->setDataPartitioning(bUseDataPartition, bUseRVLC);
```

The `uiVideoPacketLen` parameter sets the maximum number of bytes for a video packet. The first parameters of `setDataPartitioning()` enables or disables data

partitioning, i.e. separating motion data from texture data. The reversible length encoding (RVLC) can be configured by the second parameter. Note that RVLC can only be used if Data Partitioning is enabled.

To get the number of encoded video packets for a frame, and the byte offsets of the first byte for each video packet use the respective methods of the payload descriptor (see section 8).

The return value is TRUE if the codec is still uninitialized and FALSE otherwise.

### 6.2.16 GOV header period

The MPEG-4 video allows to set a so-called Group-of-Video-Object header (GOV header). mpegable™ allows the user to repeat this header periodically before an intra frame. This can be a nice feature to have non-linear access to the stream. The iGOVHeaderPeriod parameter gives the GOV header period as a multiplier of the key frame period.

```
Int32 iGOVHeaderPeriod = 1;  
  
bOk = pCtrl->setGOVHeaderPeriod(iGOVHeaderPeriod);
```

The return value is TRUE if the codec is still uninitialized and FALSE otherwise.

### 6.2.17 Motion estimation

There are some parameters concerning the motion estimation of the encoder. The most important one is the setting for the motion estimation accuracy. Possible values are 2 for half pel estimation and 4 for quarter pel estimation. The quarter pel estimation tool is part of the Advanced Simple Profile of MPEG-4 Visual. This profile was defined to further improve picture quality. Note that quarter pel estimation is often not useful for very low bit-rate encoding

```
Int32 iSearchRange = 32;  
Int32 iDirectModeRange = 0;  
Bool bUseOrigRefVop = TRUE;  
Int32 iMotEstAccuracy = 4; // quarter pel
```

```
bOk = pCtrl->setMotionEst(0
                        iSearchRange,
                        iDirectModeRange,
                        bUseOrigRefVop,
                        iMotEstAccuracy);
```

The return value is TRUE if the codec is still uninitialized and FALSE otherwise.

A simpler way to set the motion estimation accuracy is the method `setMotionEstAccuracy()`.

```
Int32 iMotEstAccuracy = 4;    // quarter pel

bOk = pCtrl->setMotionEstAccuracy(iMotEstAccuracy);
```

The return value is TRUE if the codec is still uninitialized and `iMotEstAccuracy` equals 1, 2 or 4; it is FALSE otherwise.

### 6.2.18 Interlaced Coding - ASP Level 4/5

In MPEG-4 Advanced Simple Profile the levels 4 and 5 allow the use of additional modes to improve the coding efficiency for interlaced source material. To enable this mode use the method `setInterlacedCoding()`. The method `setTopFieldFirst()` sets an indication into the bitstream about the order of top field and bottom field.

```
Bool bInterlacedCoding = TRUE;
Bool bTopFieldFirst = TRUE;

bOk = pCtrl->setInterlacedCoding(bInterlacedCoding);
bOk = pCtrl->setTopFieldFirst(bTopFieldFirst);
```

Do not use this mode when the input source is not interlaced but progressive. Do not use the de-interlacing or a vertical resizing together with the interlaced coding mode. The current version of mpegable™ Video SDK does not support interlaced coding together with B-frames or with the high complexity mode.

### 6.2.19 High complexity mode

The standard settings of the mpegable™ codec allow very fast encoding together with great compression efficiency. Sometimes the encoding process is not time critical and the only matter is the coding efficiency. The method `setHighComplexityMode ()` enables some more advanced tools that further reduce bit-rate up to 50%. Enabling this mode yields a much higher processor usage.

```
Bool bUseHighComplexityMode = TRUE;  
  
bOk = pCtrl->setHighComplexityMode(bUseHighComplexityMode);
```

The return value is TRUE if the codec is still uninitialized and FALSE otherwise.

### 6.2.20 MPEG-4 Version

The current version of the MPEG-4 standard is version 2. To create bitstreams compliant to version 1 syntax use the `setVersionId ()` method. The default settings are for version 2.

```
Int32 iVersionId = 1;  
  
bOk = pCtrl->setVersionID (iVersionId);
```

The function return TRUE if the codec is still not initialized and `iVersionId` is 1 or 2, otherwise it return FALSE.

### 6.2.21 Short header mode

The short video header mode defined by the MPEG-4 standard is a compatibility mode to H.263 Baseline. If enabled not all of the encoding features are available. This includes quarter pixel estimation, B-VOPs, MPEG quantization method and some more.

```
Bool bUseShortHeader = TRUE;  
  
bOk = pCtrl->setShortVideoHeader(bUseShortHeader);
```

The short video header mode assumes a fixed time scale of 30,000/1001 which is approximately 29.97 Hz. The `setTimeResolution()` method has no effect if short headers are used. To define a constant frame rate use the `setVOPTimeIncrement()` method. If e.g. the frame rate is about 7.5 Hz you must set a constant VOP time increment of 4, for a frame rate of about 15 Hz set it to 2. To encode video data that has no constant frame set the VOP time increment to 0 and use the descriptor's method `setVOPCompositionTime()` (see 12) for every input frame instead.

The return value of `setShortVideoHeader()` is `TRUE` if the codec is still not initialized, and `FALSE` otherwise.

## 6.3 Parameters for the decoder

### 6.3.1 Media format

The decoder supports different data formats for output data. These formats are defined in the header file `mediaformat.h`. Internally, the codec always works with YUV 4:2:0. If possible use this format or YUV 4:2:2 (YUYV) to avoid performance loss by the necessary color conversions.

The output format can be set by a call to the `setOutputFormat()` method.

The method has an optional second parameter that can be `TRUE` or `FALSE`. If set to `TRUE` the function exchanges top and bottom of every image.

```
Int32 iOutputFormat = MF_VID_YUV420;  
Bool bIsBottomUp = TRUE;
```

```
bOk = pCtrl->setOutputFormat(iOutputFormat, bIsBottomUp);
```

This version of mpegable™ Video SDK supports the following media formats:

```
MF_VID_YUV420, MF_VID_YUVA420, MF_VID_YUYV, MF_VID_UYVY  
MF_VID_RGB24, MF_VID_RGBA32, MF_VID_BGR24, MF_VID_BGRA32
```

### 6.3.2 Video object header

Before initialization, the decoder needs information about the Video Object header and the Video Object Layer header defined in ISO/IEC 14496-2. These headers are part of the DecoderSpecificInfo field of the DecoderConfigDescriptor defined in ISO/IEC 14496-1 Chap. 8.6.

The header data must be set to an object of CBufferElem and pushed to the decoder before initializing the codec.

```
// decoderSpecificInfo is a pointer to data
// uiHeaderSize is size of decoderSpecificInfo field

CBufferElem* pUnit;
Int8* pData;

COutStrmPin::allocateBufferElem (pUnit, uiHeaderSize);
COutStrmPin::getData (pUnit, pData, uiHeaderSize);
memcpy (pData, decoderSpecicInfo, uiHeaderSize);

bOk = pCtrl->pushHeader (pUnit);
```

### 6.3.3 DecoderSpecificInfo header information

After the decoder has received the DecoderSpecificInfo header and has been initialized the API allows to receive certain information e.g. the pixel aspect ratio and the VBV parameters coded in the video bitstream.

```
UInt32 uiVBVBitrate; // VBV
UInt32 uiVBVBufferSize;
UInt32 uiVBVInitialOccupancy;

UInt32 uiPARWidth; // pixel aspect ratio
UInt32 uiPARHeight;
UInt32 uiPARMode;

bOk = pCtrl->getVBVParam (
```

```
        &uiVBVBitrate,  
        &uiVBVBufferSize,  
        &uiVBVInitialOccupancy);  
  
bOk = pCtrl->getPixelAspectRatio(  
        &uiPARWidth,  
        &uiPARHeight,  
        &uiPARMode);
```

#### 6.3.4 Post filter

To increase the subjectively measured quality of the decoded video, there are different post-filters that can be applied to the reconstructed frames. We suggest the use of this filter especially for very low bit-rate video. Please note that use of this filter increases substantially processor usage of the decoder.

Currently, two filter are implemented: a deblocking filter and a deringing filter.

```
Int32 iPFMode = 3;    // deringing + deblocking  
  
bOk = pCtrl->setPostfilterMode(iPFMode);
```

The parameter values have the following meaning:

0 is for no post-filter

1 is for deblocking

2 is for deringing

3 is deblocking + deringing

It is possible to change the post filter mode after having initialized the coded control object and also during the decoding process. The return value of the method is TRUE.

Please note that in the current version of the codec the post filter only work if the horizontal resolution of the image is a multiple of 8.

### 6.3.5 **Resize**

Optionally, the decoded image can be resized before its output. This can be useful if the image shall have a certain dimension or a certain number of pixels, and no hardware accelerated resizing option is available. Usually the resizing of the output means an upsampling of the picture i.e. the number of pixels shall be enlarged. Note that almost all modern graphic cards have a resizing function implemented in hardware which should be used preferably.

There are two modes indicating how the resizing shall be performed: bi-linear or bi-cubic. While bi-cubic results usually in a higher quality the bi-linear resizing is much faster. The default mode is bi-linear. Note that the resizing still can be more time consuming than the actual decoding.

```
Int32 iNewWidth  = 720;  // upscale to PAL resolution
Int32 iNewHeight = 576;

bOk = pCtrl->setResize (iNewWidth, iNewHeight);
```

The values of the new width and height of the frame must be a multiple of 4.

The resize method can be called before initialization of the codec, or at any time during the decoding process.

## 6.4 **Initializing the codec**

When all parameters for the encoder or the decoder are set, the codec must be initialized calling the initialize method.

```
bOk = pCtrl->initialize();
```

If the return value is TRUE all parameters are set correctly and the codec is now initialized. A return value FALSE means that there is a unallowed combination of parameter settings.

### 6.4.1 Video object header

After initialization, the encoder provides access to the Video Object header and the Video Object Layer header defined in ISO/IEC 14496-2. These headers are part of the DecoderSpecificInfo field of the DecoderConfigDescriptor defined in ISO/IEC 14496-1 Chap. 8.6. This header contains exactly the data that must be pushed to the decoder before it's initialisation as described in section 6.3.2.

The data can be pulled from the encoder after having initialized the codec.

```
// decoderSpecificInfo is a pointer to a data buffer
// uiHeaderSize returns the size of decoderSpecificInfo field

CBufferElem* pUnit;
Int8* pData;

bOk = pCtrl->pullHeader (pUnit);

COutStrmPin::getData (pUnit, pData, uiHeaderSize);
memcpy (decoderSpecicInfo, pData, uiHeaderSize);

COutStrmPin::freeBufferElem (pUnit);
```

## 6.5 Error codes during initialization

In addition to the boolean return value all methods of the codec control class generate an error value in case they fail. This allows the application a better control about conflicting or not-supported settings. To reset the last error call the function clearLastError(). The error codes are defined in m4verr.h.

```
Int32 iErr = M4VNoErr;
char* pcErr = NULL;

// a quantization step of 32 is not allowed
bOk = pCtrl->setTextureQuant (16,16,32);

if (bOk == FALSE)
```

```
{
    iErr    = pCtrl->getLastErrNo();    // return last error number
    pcChar  = pCtrl->getLastErrText();  // return last error text

    clearLastError(); // clear last error
}
```

## 6.6 Configure hardware support

Although mpegable™ Video SDK is designed to work on different platforms the codec makes extensive use of processor specific optimizations. To enable or disable a certain hardware support use the function `setMMXSupport()`. For x86 processors, the value `TRUE` means to use MMX support while `FALSE` disables any processor specific hardware accelerations.

In the current version, only the parameter values `TRUE` and `FALSE` are supported but future versions will allow an integer parameter to enable different types of processor optimization.

The function `setMMXSupport()` returns `TRUE` if the platform supports the requested hardware and the codec is still not initialized, and `FALSE` otherwise.

```
Bool bUseMMXSupport = TRUE;

bOk = pCtrl->setMMXSupport (bUseMMXSupport);
```

## 7 Processing the Data

It has been described how parameters are set in the codec control object and how input and output of the mpegable codec works. The last step is to instruct the codec to process data. This is done by a call to the codec's object method `process()`. This function processes all data from the input pin that are available at this time, writing the output data units to the output pin. If there are no more input data the function returns.

The `process` method expects a parameter. If this parameter is a positive number the codec tries to process the given number of frames. If set to zero the codec

processes all data units available from the input pin.

After the process() method has been returned, this parameter holds the number of processed data units.

```
Int32 iProcessedUnits = 0;

pCodec->process(iProcessedUnits);
```

The function returns as soon as no more data are available from the input pin or the given number of frames are written to the output pin.

## 8 Payload descriptor

For every data unit that the codec writes to the attached output pin or read from the appropriate input pin, there exists an object of the class CVidStreamPayloadDesc. This object holds some additional information of the frame.

The encoder sets a flag indicating a key-frame and the time stamp encoded in the bitstream. The decoder sets frame resolution, frame type (0 = I-frame, 1 = P-frame, 2 = B-frame) and the timestamp of the decoded frame.

To access this information, the class CVidStreamPayloadDesc provides the required methods.

```
CBufferElem* pUnit;
CVidStreamPayloadDesc* pDesc;

// pull a data unit from the output pin

// Get pointer to unit descriptor
COutStrmPin::getDesc(pUnit, pDesc);

// Read and print frame info
Int32 iWidth = pDesc->getVOPWidth();
Int32 iHeight = pDesc->getVOPHeight();
Int32 iCompTime = pDesc->getVOPCompositionTime();
```

```
Int32 iDecodeTime = pDesc->getVOPCodingTime();
Int32 iVOPType = pDesc->getVOPType();
Char cVOPType;
switch (iVOPType)
{
    case 0: cVOPType = 'I'; break;
    case 1: cVOPType = 'P'; break;
    case 2: cVOPType = 'B'; break;
    default: assert(false);
}
printf("%3d x %3d Time: %4d %4d (%c-VOP)\n",
        iWidth, iHeight, iCompTime, iDecodeTime, cVOPType);
```

The application must not delete the CVidStreamPayloadDesc object. It will be removed automatically when destroying the CBufferElem object.

## 9 H.263 Coding

In addition to the MPEG-4 video format, the mpegable™ Video SDK also supports the ITU-T standard H.263. The Baseline profile is already part of MPEG-4 and is called Short Header in this context. Other profiles of H.263 include more advanced tools and are not part of MPEG-4. but are referenced by other standards like e.g. 3GPP Version 5.

Before decoding or encoding H.263 video data the codec state must be switched accordingly by using the setH263Mode() method of the codec control class. This method must be called before initializing the codec.

When configured for H.263 the decoder does not require any header information since all necessary data are part of the stream. The decoder of version 1.9 of the SDK supports the Baseline profile of H.263.

The encoder implements additional tools which are not part of the Baseline Profile but of more advanced Profile 3. To configure the encoder and to enable or disable tools in H.263 there are extra methods in the codec's API.

## 9.1 Limitations of the H.263 Baseline Profile

The level 0 of H.263 (also known as Short Header) only supports a fixed set of picture resolutions. The predefined resolutions are 120x96, 176x144, 352x288, 704x576 and 1408x1152. No other resolutions are supported in Level 0 of H.263.

In a similar way the time resolution is fixed and therefore the possible frame rates. The default time resolution in H.263 is 30000/1001 which is approximately 29.97. Picture time stamps are measured in the default time scale when H.263 mode is enabled and no custom time scale is set.

To encode different resolution or time scales the PLUSTYPE header is added to the H.263 picture header. The PLUSTYPE header is able to indicate user defined resolutions and time scales but is not conformant to H.263 Level 0.

The mpegable Video SDK™ inserts the PLUSTYPE header automatically always when the user settings require it. In this case the resulting H.263 bitstream is not longer compatible to H.263 Level 0, or MPEG-4 Short Header respectively.

The use of a custom time scale requires two parameters. A time scale and a clock conversion factor which is either 1000 or 1001 (default). The default value for the time scale is 30000. The actual time resolution in which all picture time stamps are measured finally results from the formula  $timescale/clockconversionfactor$ . All common frame rates can be presented easily due to this formula.

Base frame rate	Time scale	Clock conversion factor	Time increment
25 Hz	25000	1000	1
29,97 Hz	30000	1001	1
15 Hz	15000	1000	1
14,985 Hz	30000	1001	2

## 9.2 New tools in H.263 Profile 3

The mpegable Video SDK™ supports further tools of H.263 that are not part of the Baseline Profile but of Profile 3. This is

- Deblocking
- Modified quant
- Four motion vectors

- Advanced intra prediction

All tools can be enabled and disabled separately by the corresponding method of the codec control class.

## 10 Codec Listener

Since version 1.2, mpegable™ implements a mechanism to send events to the application. To use that feature, the application has to define a new class that inherits from the class `MP4CodecListenerIf` defined in `codeclisten.h`. An object of that class must be added to the codec control by a call of `addListener()` before having initialized the codec control.

```
class CMyListener : public MP4CodecListenerIf
{
    ...
};

CMyListener* pListener = new CMyListener();
Int32 iEventType = MP4CodecEvent::EncoderVOP;
Int32 iLevel = 5;

pCtrl->addListener (pListener, iEventType, iLevel);

...

pCtrl->removeListener(pListener, iEventType);
delete pListener;
```

At the end, the listener object must be removed by a call to the `removeListener()` method. The listener object must not be destroyed before it has been removed from the codec control.

The derived class must overwrite the `notify()` method of the class `MP4CodecListenerIf`. The prototype declaration of this method is

```
void MP4CodecListenerIf::notify (MP4CodecEvent& rcEvt);
```

The `MP4CodecEvent` class is declared in `codecevent.h`. There are different types of codec events, each represented by an own class derived from `MP4CodecEvent`. In this version of `mpegable™`, the only event types are `MP4CodecEvent::EncoderVOP` and `MP4CodecEvent::DecoderVOP`. A typical implementation of a derived `notify()` method can be

```
Void CMP4VideoCodec::notify (MP4CodecEvent& rcEvt)
{
    switch (rcEvt.getType ())
    {
        case MP4CodecEvent::EncoderVOP:
            notifyEnc(static_cast<VOPEncoderEvent*> (rcEvt.getEventIf()));
            break;
        case MP4CodecEvent::DecoderVOP:
            notifyDec(static_cast<VOPDecoderEvent*> (rcEvt.getEventIf()));
            break;
        default:
            break;
    }
}
```

Always use the `getEventIf()` method to get a pointer to the event object of the appropriate event type. The return value has to be casted to a pointer of this class. The codec calls this `notify()` method whenever an event occurs. An event is defined by an event context, an event message and further informations depending on the event type.

## 10.1 The `VOPEncoderEvent` class

The codec sends events of type `MP4CodecEvent::EncoderVOP` while encoding a frame. This version of `mpegable™` defines five different contextes in which the `notify()` function can be called.

```
enum EventContext
{
    SevereError = 0,
```

```
    InitVop = 1,  
    BeginOfVop = 2,  
    EndOfVop = 3,  
    EndOfSeq = 4,  
};
```

The method `getEventContext()` returns the context in which the event has occurred. Currently, this is only `MP4CodecEvent::InitVop` or `MP4CodecEvent::EndOfVop`.

```
Void CMP4VideoCodec::notifyEnc (VOPEncoderEvent& rcEncoderEvt)  
{  
    Int32 iContext = rcEncoderEvt.getEventContext();  
  
    switch (iContext)  
    {  
    case MP4CodecEvent::InitVop:  
        ...  
        break;  
    case MP4CodecEvent::EndOfVop:  
        ...  
        break;  
    default:  
        // unknown context  
    }  
}
```

The method `getVopType()` returns the type of frame. That is 0 for I-frames, 1 for P-frames and 2 indicating a B-frame.

The method `getMessage()` returns the message that belongs to the event. The following messages are defined

```
enum EventMessage  
{  
    NoMsg = 0,  
    SendVolHeaderMsg = 1,  
    CodedVopMsg = 2,  
};
```

```
    NotCodedVopMsg = 3,  
    SkippedVopMsg = 4,  
};
```

In the `MP4CodecEvent::InitVop` context, the `MP4CodecEvent::SendVolHeaderMsg` message is sent if a video object layer header has been encoded. The `getVolHeaderIf()` method returns a pointer to an interface to access the parameters defined in the MPEG-4 visual video object layer header. The interface is defined in `volheaderif.h`.

In the `MP4CodecEvent::EndOfVop` context, the `MP4CodecEvent::CodedVopMsg` message or the `MP4CodecEvent::NotCodedVopMsg` is sent if a frame has been encoded. The `getVopStatisticIf()` method returns a pointer to an interface to access some statistic parameters resulting from encoding the frame.

```
    Int32 iMessage = rcEncoderEvt.getMessage();  
    VOPStatisticIf* pStat = NULL;  
  
    assert(MP4CodecEvent::EndOfVop == rcEncoderEvt.getEventContext());  
  
    if (MP4CodecEvent::CodedVopMsg == iMessage)  
    {  
        pStat = rcEncoderEvt.getVopStatisticIf();  
        printf("VOP total bits %d\n", pStat->getBitsTotal());  
        printf("VOP statistic\n");  
        pStat->print(TRUE);  
    }  
    else if (MP4CodecEvent::NotCodedVopMsg == iMessage)  
    {  
        printf("VOP not coded\n");  
    }  
}
```

### 10.1.1 VOP statistics

The class `VOPStatisticIf` is defined in `vopstatistic.h` and provides many functions to access internal statistics about the encoded frame. The following section describes the statistic functions of the SDK.

```
virtual UInt32 getBitsHead() const = 0;
```

returns the number of bits for coding the VOP header.

```
virtual UInt32 getBitsY() const = 0;
```

returns the number of bits for coding the luminance component.

```
virtual UInt32 getBitsCr() const = 0;
```

returns the number of bits for coding the first chrominance component.

```
virtual UInt32 getBitsCb() const = 0;
```

returns the number of bits for coding the second chrominance component.

```
virtual UInt32 getBitsA() const = 0;
```

returns the number of bits for coding the grayscale alpha mask.

```
virtual UInt32 getBitsShapeMode() const = 0;
```

returns the number of bits for coding the shape (does not work correct).

```
virtual UInt32 getBitsTexture() const = 0;
```

returns the number of bits for coding the texture.

The following functions return the number of bits used for coding the appropriate property defined in the MPEG-4 Visual standard

```
virtual UInt32 getBitsCOD() const = 0;
```

```
virtual UInt32 getBitsCBPY() const = 0;
```

```
virtual UInt32 getBitsMCBPC() const = 0;
```

```
virtual UInt32 getBitsDQUANT() const = 0;
```

```
virtual UInt32 getBitsMODB() const = 0;
```

```
virtual UInt32 getBitsCBPB() const = 0;
```

```
virtual UInt32 getBitsMBTYPE() const = 0;
```

```
virtual UInt32 getBitsIntraPred() const = 0;
```

```
virtual UInt32 getBitsNoDCT() const = 0;
```

```
virtual UInt32 getBitsCODA() const = 0;
```

```
virtual UInt32 getBitsCBPA() const = 0;
virtual UInt32 getBitsMODBA() const = 0;
virtual UInt32 getBitsCBPBA() const = 0;
virtual UInt32 getBitsStuffing() const = 0;
```

The following functions return the number of macro blocks (MB) coded in the specified way

```
virtual UInt32 getSkipMB() const = 0;
```

returns the number of skipped macro blocks

```
virtual UInt32 getInterMB() const = 0;
```

returns the number of inter-coded predicted macro blocks

```
virtual UInt32 getInter4VMB() const = 0;
```

returns the number of predicted macro blocks using 4 motion vectors

```
virtual UInt32 getIntraMB() const = 0;
```

returns the number of intra coded macro blocks

```
virtual UInt32 getDirectMB() const = 0;
```

returns the number of predicted macro block using the direct mode (B-VOP only)

```
virtual UInt32 getForwardMB() const = 0;
```

returns the number of predicted macro block using the forward mode (B-VOP only)

```
virtual UInt32 getBackwardMB() const = 0;
```

returns the number of predicted macro block using the backward mode (B-VOP only)

```
virtual UInt32 getInterpolateMB() const = 0;
```

returns the number of predicted macro block using the bidirectional prediction

interpolation mode (B-VOP only)

```
virtual UInt32 getVOPs() const = 0;
```

returns always 1

```
virtual UInt32 getBitsMV() const = 0;
```

returns the number of bits used for coding the motion vectors

```
virtual UInt32 getBitsShape() const = 0;
```

returns the number of bits for coding the shape

```
virtual UInt32 getBitsTotal() const = 0;
```

returns the total number of bits bits used for the current for coding VOP

The following functions return information about the difference between the original and the reconstructed image. PSNR here means Peak Signal to Noise Relation

```
virtual Double getSNRY() const = 0;
```

returns the average PSNR of the luminance component (Y-signal)

```
virtual Double getSNRU() const = 0;
```

returns the average PSNR of the first chrominance component (U-signal)

```
virtual Double getSNRV() const = 0;
```

returns the average PSNR of the second chrominance component (V-signal)

```
virtual Double getSNRA() const = 0;
```

returns the average PSNR of the grayscale alpha mask

```
virtual UInt32 getQMB() const = 0;
```

returns the number of quantized macro blocks

```
virtual UInt32 getQp() const = 0;
```

returns the sum of the quantisation step sizes of all quantized macro blocks. The average quantization parameter of one VOP is given by `getQP() / getQMB()` assuming not all macro blocks are skipped.

```
virtual Int32 getPeakErrY() const = 0;
```

returns the maximum peak error of the luminance cY-component

```
virtual Int32 getPeakErrU() const = 0;
```

returns the the maximum peak error of the first chrominance U-component

```
virtual Int32 getPeakErrV() const = 0;
```

returns the maximum peak error of the second chrominance V-component

```
virtual Int32 getPeakErrA() const = 0;
```

returns the maximum peak error of the grayscale alpha mask

## 10.2 The VOPDecoderEvent class

The codec sends events of type `MP4CodecEvent::DecoderVOP` while decoding a frame. This version of mpegable™ defines five different contextes in which the `notify()` function can be called.

```
enum EventContext
{
    SevereError = 0,
    InitVop = 1,
    BeginOfVop = 2,
    EndOfVop = 3,
    EndOfSeq = 4,
};
```

The method `getEventContext()` returns the context in which the event has ocured. Currently, this is only `MP4CodecEvent::InitVop` or `MP4CodecEvent::EndOfVop`.

```
Void CMP4VideoCodec::notifyDec (VOPDecoderEvent& rcDecoderEvt)
```

```
{
    Int32 iContext = rcDecoderEvt.getEventContext();

    switch (iContext)
    {
    case MP4CodecEvent::InitVop:
        ...
        break;
    case MP4CodecEvent::EndOfVop:
        ...
        break;
    default:
        // unknown context
    }
}
```

The method `getMessage()` returns the message that belongs to the event. The following messages are defined

```
enum EventMessage
{
    NoMsg = 0,
    ReceivedVolHeaderMsg = 1,
    CodedVopMsg = 2,
    NotCodedVopMsg = 3,
};
```

In the `MP4CodecEvent::InitVop` context, the `MP4CodecEvent::ReceivedVolHeaderMsg` message is sent if a video object layer header has been decoded. The `getVolHeaderIf()` method returns a pointer to an interface to access the parameters defined in the MPEG-4 visual video object layer header. The interface is defined in `volheaderif.h`.

In the `MP4CodecEvent::EndOfVop` context, the `MP4CodecEvent::CodedVopMsg` message or the `MP4CodecEvent::NotCodedVopMsg` is sent if a video object plane has been encoded. The method `getVopType()` returns the type of frame that has been decoded. That is 0 for an I-frames, 1 for a P-frames and 2 indicating a decoded B-frame.

---

## 11 Manual controls during encoding

As described above, it is possible to change some parameters also after having initialized the codec and during the encoding or decoding process. Beside bit-rate, quantisation step-size and post-filter mode, there are some more possibilities how to control the coding process.

### 11.1 Key frames

As a new feature in version 1.24, mpegable<sup>TM</sup> allows the manual setting of key-frames. This is done by the unit descriptor of the data unit pushed to the encoder's input queue. Use the `setKeyVop()` method of the descriptor class.

```
CBufferElem* pUnit;
CVidStreamPayloadDesc* pDesc;
CInStrmPin* pInBuf;

// allocate buffer unit and fill with data
...

CInStrmPin::getDesc(pUnit, pDesc);

pDesc->setKeyVop();

pInBuf->push(pUnit);
```

### 11.2 Frame skip

Normally the bit-rate control implemented in mpegable<sup>TM</sup> decides whether to skip a frame or not. But sometimes there are reasons outside the codec to skip one or more frames. To communicate this to the codec an application can call the `setSkipNFrames()` method of the codec control every time during the encoding process.

```
Int32 iSkipCount = 1;
```

```
bOk = pCtrl->setFramesToSkip(iSkipCount);
```

The return value is TRUE if iSkipCount is a positive number and the codec is already initialized, and FALSE otherwise.

## 12 Using time stamps

As mpegable<sup>TM</sup> expects raw video frames without header as input data, it needs additional information about the time line. Usually the time line of a video starts at 0 with the first frame and then every two successive frames have a constant time difference. The time can be measured in parts of a second. The smallest time unit in mpegable<sup>TM</sup> is called a "tick" and is defined by the setTimeResolution() method of the codec control. The parameter given to this method counts the number of ticks elapsing in one second.

In many applications the time difference between two successive frames is a constant. In that case, the setVOPTimeIncrement() method sets the time difference measured in ticks. The codec automatically computes the time stamp for every frame beginning from 0 for the first frame.

### 12.1 Manually setting of time stamps

To set the time stamps individually for every frame, the application must set the time increment explicitly to 0 to switch off the automatic setting of time stamps.

```
Int32 iTimeResolution = 1000;

bOk = pCtrl->setTimeResolution(iTimeResolution);
bOk = pCtrl->setVOPTimeIncrement(0); // manually setting
```

The only property of the payload descriptor currently used to add information to a input data unit is the composition time. Supposed the time increment is set to 0, the encoder expects a composition time stamp for every frame. A call to the setVOPCompositionTime() method sets the composition time in the descriptor that belongs to the frame.

```
CBufferElem* pUnit = NULL;
UInt32 uiFrameSize = 352*288; // cif
CVidStreamPayloadDesc* pDesc = NULL;
Int32 iCompTime = 0;

// allocate data unit
CInStrmPin::allocateBufferElem(&pUnit, uiFrameSize);

// get data pointer and copy data to the buffer
....

// get pointer to the unit descriptor
COutStrmPin::getDesc(pUnit, pDesc);

// set composition time
pDesc->setVOPCompositionTime(iCompTime++);

pInPin->push(pUnit);
```

The composition time is measured in time ticks and depends on the time resolution that has been set by the `setTimeResolution()` method called before.

## 12.2 Time stamps and bitrate-control

To enable the constant bitrate (CBR) mode of mpegable™ the application determines the number of available bits per second. To work properly the rate control also needs to know how many frames per second share these bits. Therefore, it needs at least an average frame rate. When setting the time stamps manually, the application must use an additional parameter of the `setBitrate()` method that indicates the average frame rate.

```
Int32 iBitsPerSecond = 500000;
Double dAvgFrameRate = 25.0;

bOk = pCtrl->setBitrate (iBitsPerSecond,
    dAvgFrameRate);
```

## 13 Two-Pass Encoding

Starting with version 1.6, the mpegable™ Video SDK includes a two-pass encoding mode. The general framework for two-pass encoding is already available in version 1.46.

### 13.1 General workflow in two-pass mode

In 2-pass encoding mode, two separate codec objects must be instantiated. We call the first one the *Pre-Encoder* and the second one the *Main-Encoder*. Both encoder objects must be initialized as in the usual single-pass mode with exactly the same set of parameters. By a new method `set2PassEncoding()` of the codec control class the objects are configured as Pre-Encoder and Main-Encoder.

In the first phase of encoding, the Pre-Encoder analyzes the input video and collects statistical data about the video content. The Pre-Encoder expects the uncompressed video frames at its input pin, as it is in the single-pass mode. Each time the `process()` method is called, the Pre-Encoder processes all data at the input pin and sends a data packet for every frame to the output pin. The data packets contain VOP meta data that can be used by the Main-Encoder for the actual encoding.

In the second phase the VOP meta data of all analyzed frames are evaluated and the actual encoding takes place. The Main-Encoder has two input pins, one for the uncompressed video data (pin 0), and one for the VOP meta data packets, which have been created by the Pre-Encoder (pin 3). Each time the `process()` method of the Main-Encoder object is called, the Main-Encoder evaluates all VOP meta-data at its input pin 3, and then reads and processes the uncompressed video frames from its input pin 0 considering the information from the VOP meta data. The resulting output packets are pushed to the Main-Encoder's output pin in the same way as in single-pass mode.

Using two separate encoder objects allows a flexible usage of the two-pass encoding mode. On one hand, you can process the two phases strictly separated after each other. The Pre-encoder analyzes the complete input stream and sends all VOP meta data. The application then pushes all VOP meta data packets to the Main-Encoder's input pin, reads the input stream again starting with the first frame. The Main-Encoder has the VOP meta data of all frames of the in-

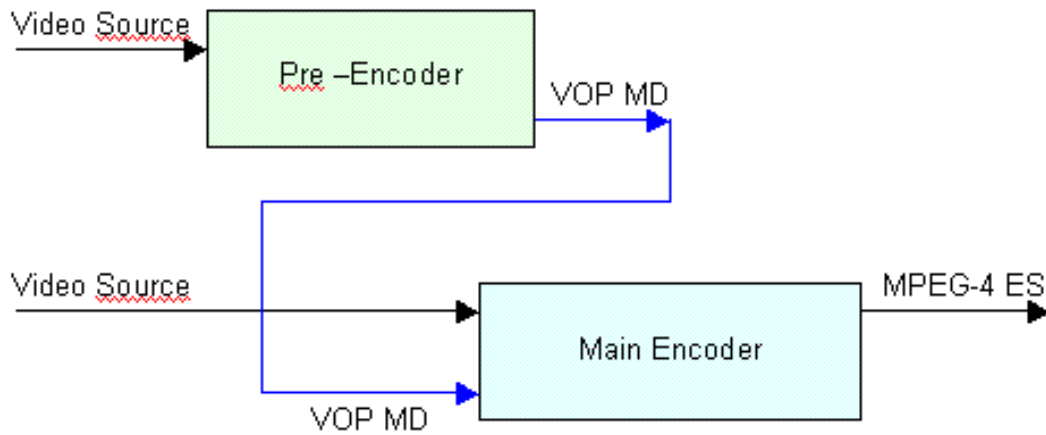


Figure 1: Two-pass encoding workflow

put stream before processing the first frame. This mode is the classical two-pass encoding mode.

On the other hand, it is possible to have both phases at the same time. In this case the Pre-Encoder runs in advance. Every VOP meta data packet is sent immediately from the Pre-Encoder's output pin to the Main-Encoder's input pin. If the Main-Encoder has a certain amount of VOP meta data packets, the first uncompressed video frame is sent to the Main-Encoder's input pin 0. When the application calls the `process()` method the Main-Encoder processes the video frames from the input pin 0 considering all information collected from the VOP meta data packets that are processed until this point. This mode is also called Look-ahead. The size of the Look-ahead interval, i.e. the offset of processed frame by the Pre-Encoder and the Main-Encoder can be arbitrarily chosen and can also vary over time during the encoding.

As a larger Look-ahead interval means more information for the Main-Encoder at the point of encoding, a larger offset between Pre-Encoder and Main-Encoder can also mean more quality. On the other hand, for many real-time applications it is desirable to have small delay. The above described architecture meets both requirements.

## 13.2 API Description

There is a new function `set2PassEncoding()` in the API to configure a codec object work in single pass mode, or to be a Pre-Encoder or a Main-Encoder in two-pass encoding mode. A parameter value 0 means to run in single pass mode. This is the default. A parameter 1 indicates to run as Pre-Encoder, and 2 means to run as Main-Encoder.

```
Int32 iTwoPassEncMode = 1; // run as pre-encoder

bOk = pCtrl->set2PassEncoding (iTwoPassEncMode);
```

The API has functions to configure a codec control object with a certain set of parameters. This shall ensure that both encoders are configured with exactly the same parameters.

The `getInitString()` receives an initialization string from the codec control that includes all default values or previously changed setting of the codec control object. The codec allocates the memory for the initialization string but the application is responsible to free the memory by a call to `freeInitString ()`. The return value is TRUE

```
char* pcInitString = NULL;

// get codec control object of the Pre-Encoder
pCtrl = (Mpeg4VidEncoderCtrlIf*)
        (pPreEncoder->getCodecCtrl(MPEG4V_COMPRESSOR));

// set all parameters here
// ....

// get the initialization string
bOk = pCtrl->getInitString (&pcInitString);

// configure the object to run as pre-encoder
bOk = pCtrl->set2PassEncoding(1);

// initialize the codec and run the first pass
```

```
// ...
```

The `setInitString()` method configures a the codec control with the initialization string that is given as parameter. The char pointer must point to a null terminated string. The return value is TRUE if the codec is still not initialized and the parameter points to a valid initialization string. The application is responsible for allocation and release of the memory used for the string.

```
// get codec control object of the Pre-Encoder
pCtrl = (Mpeg4VidEncoderCtrlIf*)
        (pMainEncoder->getCodecCtrl(MPEG4V_COMPRESSOR));

// set the initialization string to configure the codec
bOk = pCtrl->setInitString (pcInitString);

// free the initialization string to configure the codec
bOk = pCtrl->freeInitString (pcInitString);
pcInitString = NULL;

// configure the object to run as main-encoder
bOk = pCtrl->set2PassEncoding(2);

// initialize the codec and run the first pass
// ...
```

The `freeInitString()` method releases the memory of an initialization string previously created by `getInitString()`. The return value is TRUE.