Compression Techniques

- Introduction to Compression Methods
- Basic Coding Methods
- Video Compression
- Audio Compression
- Interesting Web Sites & Tools
- Microsoft DirectShow

Multimedia Objects Are Huge:
- Consider a video sequence:
  - Frame size is 640x480 pixels, 16 bits per pixel, playback rate is 30 fps, and video length is five minutes.

  ⇒ Transmission rate is: \(640 \times 480 \times 2 \times 30 \approx 18 \text{ MBytes/sec.}\)
  ⇒ Storage requirement is: \(18 \text{ MBytes/sec} \times 5 \text{ min} \approx 5.4 \text{ GBytes.}\)
Illustration

Source computer

Source information

Source encoder program/processor

Network

Destination computer

Destination decoder program/processor

Copy of source information

Information representation:

Exemplar pixel patterns

Pixel amplitude

Line 2: (high spatial frequency)

Horizontal position

Horizontal position
Background of Compression

Several types of redundancy can be removed by compression:

- **Spatial redundancy**: Nearby pixels are strongly correlated in natural images.
- **Redundancy in scale**: Image features like straight edges and constant regions are invariant under re-scaling.
- **Redundancy in frequency**: An audio signal can completely mask a sufficiently weaker signal in its frequency-vicinity.
- **Temporal redundancy**: Adjacent video frames often show very little change.
- **Stereo redundancy**: The correlation's between stereo channels are high.

Characteristics of compression methods:

- **Lossless (& lossy)**: Original data can be precisely recovered (or not.)
- **Intraframe (& interframe)**: Frames are coded independently (or dep.).
- **Symmetrical (& asymmetrical)**: Encoding and decoding time are almost equal. (Encoding time considerably exceeds decoding time.)
- **Real-time**: Delay for encoding/decoding should not exceed 50 msec.
- **Scalable**: Frames are coded in different resolutions or quality levels.
Assessment of Compression Methods

- **Compression Ratio = (Source Bits)/(Compressed Bits).**
  - The ratio is higher when a more sophisticated method is used or under a higher amount of reconstructed errors.

- **Complexity:**
  - Computation Cost: execution time, required memory, etc.

- **Quality (of the Reconstructed):**
  - MSE (mean square error) -
  - RMSE (root MSE) -
  - SNR (Signal-to-Noise ratio) -
  - PSNR (Peak SNR) -

\[
RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^{N} [r(i)]^2}. \quad r(i) = f(i) - \hat{f}(i).
\]

\[
RMSE = \sqrt{\frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} [f(i, j) - \hat{f}(i, j)]^2}. \quad 2D
\]

\[
SNR(dB) = 10 \log_{10} \left( \frac{\sigma_f^2}{\sigma_r^2} \right). \quad \sigma_f^2 = \frac{1}{N} \sum_{i=1}^{N} [f(i)]^2.
\]

\[
PSNR(dB) = 10 \log_{10} \left( \frac{Q^2}{\sigma_r^2} \right). \quad Q = \text{Maximum} = 255, \text{ using 8 bits.}
\]
Delay of inputs are applied to approach the precision of statistical prediction model.

- The Model keeps adjusting the mapping based on the conditional probability.
  \[ P(s_i \mid s_{i-r}, \ldots, s_{i-2}, s_{i-1}) \]

- Fewer bits are used to denote the more frequent symbols, while using lengthier bits to distinguish the far seldom symbols.

- The quality of the Model determines the compression ratio.
Unique Decodable $\iff$ Instantaneous Code

- **Unique decodable:**
  - Each symbol in the output compressed codes can be uniquely identified to restore the original symbol.
  - For instance, this is not unique decodable:
    \[ s_1 = 0, s_2 = 01, s_3 = 11, s_4 = 00 \Rightarrow "0011" = s_4s_3, \ s_1s_1s_3 ? \]

- **Instantaneous Code:**
  - No code is the prefix of the others in the output.
  - The decoding tree can be constructed such that each received bit is used only once to direct the traversal of the tree from the root toward the leaf.
  - For instance, the following is not “Instantaneous Code”.
    \[ s_1 = 0, s_2 = 01, s_3 = 011, s_4 = 111 \Rightarrow "011\ldots1111" = s_3s_4, \ s_4s_4 ? \]

Extra delay is needed to first resolve the last symbol, and back to the previous ones!
Run-Length Coding (RLC)

A sequence of same bytes in multimedia data can be compressed (represented) in the number of occurrences and the repetitive byte pattern itself.

- Uncompressed data: UNNNNNNNIMANNHEIM
- Run-length coded: U!6NIMANNHEIM using an exclamation mark(!) as flag.

NOTE:

- “!2N” for NN would increase the code length by one byte.
- “Byte stuffing”: A real “!” (special flag) in the data is coded as “!!”.
- Longer sequences of different characters can be also coded in a similar way.
- Indeed, different characters need not have to be encoded with a fixed number of bits. (See next)
Huffman Coding: Instantaneous Code (Unique)

- Huffman ('52) developed a compression method to determine the optimal code for given data,
  - i.e., using the minimum number of bits given the probability distribution.

Example:
- Suppose the characters A, B, C, and D have the following probability of occurrence:
  - \( p(A) = \frac{3}{4}, \ p(B) = \frac{1}{8}, \ p(C) = p(D) = \frac{1}{16}, \) or
  - \( p(ABCD) = 1, \ p(BCD) = \frac{1}{4}, \ p(CD) = \frac{1}{8}, \ p(D) = \frac{1}{16}. \)

- Since \( p(A) \geq p(B) \geq p(C) \geq p(D) \) [skew probability], the Huffman table can be generated as
  - \( w(A) = 1, \ w(B) = 01, \ w(C) = 001, \ w(D) = 000. \)
  - The average code length will be \( 1 \times \frac{3}{4} + 2 \times \frac{1}{8} + 3 \times \frac{1}{16} \times 2 = 1.375 \) bits instead of 2 bits.
Huffman code tree construction

Shannon’s formula: Entropy $H$

$$H = - \sum_{i=1}^{n} P_i \cdot \log_2 P_i$$

bits per codeword

1. All nodes are free nodes;
2. Group two least-weight nodes (0 & 1).
3. Make a parent with the weights sum.
   (Branch node).
4. Repeat 2 & 3 until one node is left.

Frequency of occurrence

Weight order = D1  C1  2  B2  4  A4  8 ✔
Huffman encoding example

A = 01
B = 10
C = 111
D = 011
E = 00001
F = 00000
G = 0011
H = 0010

Weight order = 0.055 0.055 0.055 0.055 0.11 0.11 0.14 0.14 0.22 0.25 0.25 0.28 0.47 0.53
Arithmetic Coding

- Using floats instead of characters used in Huffman coding, Arithmetic coding achieves better compression for the sacrifice of more expensive computation. [always achieve the Shannon value]
  - A message is represented by an interval of real numbers between 0 and 1.
  - P(s)=1/3, the best size is 1.6 bits, but 1 or 2 bits are used in Huffman.

Example:

- Suppose p(A) = .2, p(B) = .3, p(C) = .1, p(D) = .2, and p(E) = p(!) = .1, relate to more precise probability ranges: A = [.0, .2), B = [.2, .5), C = [.5, .6), D = [.6, .8), E = [.8, .9) and ! (EOF symbol) = [.9, 1.).
- The encoded message of “BACC” builds up as follows:
  - Initially, [ 0, 1)
  - After seeing B [0.2, 0.5)
  - A [0.2, 0.26)
  - C [0.23, 0.236)
  - C [0.233, 0.2336)
  - ! [0.23354, 0.2336), therefore, a number in the range like 0.23355 can represent the code.
- The best single-character model for “BACC” can be computed as {A(0.2), B(0.2), C(0.4), !(0.2)} to consume only 3 digits.
Arithmetic coding example

Example character set and their probabilities:

\[ e = 0.3, \ n = 0.3, \ t = 0.2, \ w = 0.1, \ = 0.1 \]

Encoded version of the character string \textit{went}. is a single codeword in the range \( 0.81602 \leq \text{codeword} < 0.8162 \)
LZW compression

Dictionary contents:
[Index = 8bits]

These locations used to hold the codewords of the basic character set

These locations used to hold the codewords of the characters that make up each new word that occurs in the text string

This is sent using the index of the word is (129)

Each character is sent using the index of the individual character in the basic character set
LZW compression

(b)

Initial index = 8 bits

Index incremented to 9 bits

0

127

128

129

This

is

fish

pond

511

Basic character set

Existing dictionary

Extended dictionary
GIF compression

<table>
<thead>
<tr>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Color dictionary:

[derived by the source using either a localized set of colors in the image – a local color table – or all the colors in the image – a global color table]

These locations used to hold the 256 colors from the possible $2^{24}$ set of colors that are to be used to represent all the colors in the image. The color of each pixel in the image is sent using the 8-bit table index.

The color dictionary, screen size, and aspect ratio are sent with the set of indexes for the image.
### GIF compression

#### (b)

**Color dictionary:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index X</td>
<td>Index X</td>
<td>Index X</td>
</tr>
<tr>
<td>Index Y</td>
<td>Index Y</td>
<td>Index Y</td>
</tr>
<tr>
<td>Index A</td>
<td>Index A</td>
<td>Index A</td>
</tr>
<tr>
<td>Index X</td>
<td>Index X</td>
<td>Index Z</td>
</tr>
</tbody>
</table>

- **9-bit index**
  - Basic set of 256 selected colors
  - Strings of 3 pixels of the same basic color sent using the index of the related table entry
  - Table can be extended if strings of different colors are included

**Dictionary of common strings of pixel values of the same color — derived dynamically**
GIF interlaced mode

- X = Group 1 data
- O = Group 2 data
- + = Group 3 data
- / = Group 4 data

Image with Group 1 only

Image with Groups 1 and 2

Image with Groups 1, 2 and 3

Image with Groups 1, 2, 3 and 4
Image/block compression
Image/block compression
Transform Coding (one type of Source Coding)

**Spatial Domain**

<table>
<thead>
<tr>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>170</td>
</tr>
<tr>
<td>56</td>
<td>130</td>
</tr>
<tr>
<td>80</td>
<td>203</td>
</tr>
<tr>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>69</td>
<td>148</td>
</tr>
</tbody>
</table>

\[
\theta = \begin{bmatrix} \cos \phi & \sin \phi \\ -\sin \phi & \cos \phi \end{bmatrix} \cdot X
\]

**Frequency Domain**

<table>
<thead>
<tr>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>3</td>
</tr>
<tr>
<td>56</td>
<td>-4</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>-7</td>
</tr>
<tr>
<td>69</td>
<td>-9</td>
</tr>
</tbody>
</table>

Very small numbers

**Reconstructed Data**

<table>
<thead>
<tr>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>169</td>
</tr>
<tr>
<td>53</td>
<td>131</td>
</tr>
<tr>
<td>81</td>
<td>203</td>
</tr>
<tr>
<td>34</td>
<td>84</td>
</tr>
<tr>
<td>61</td>
<td>151</td>
</tr>
</tbody>
</table>

\[
W = 2.5H
\]

\[\Phi = \arctan 2.5\]

**Graph**

- Weight vs Height
- Very small numbers:
  - 65: 0
  - 56: 0
  - 80: 0
  - 40: 0
  - 69: 0

Do not store these numbers
Discrete Cosine Transform (DCT)

Apply DCT to each 8x8 block:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>136</td>
<td>138</td>
<td>140</td>
<td>144</td>
<td>145</td>
<td>147</td>
<td>155</td>
</tr>
<tr>
<td>136</td>
<td>140</td>
<td>140</td>
<td>147</td>
<td>140</td>
<td>148</td>
<td>155</td>
<td>156</td>
</tr>
<tr>
<td>140</td>
<td>143</td>
<td>144</td>
<td>148</td>
<td>150</td>
<td>152</td>
<td>154</td>
<td>155</td>
</tr>
<tr>
<td>144</td>
<td>144</td>
<td>146</td>
<td>145</td>
<td>149</td>
<td>150</td>
<td>153</td>
<td>160</td>
</tr>
<tr>
<td>150</td>
<td>152</td>
<td>155</td>
<td>156</td>
<td>150</td>
<td>145</td>
<td>144</td>
<td>140</td>
</tr>
<tr>
<td>144</td>
<td>145</td>
<td>146</td>
<td>148</td>
<td>143</td>
<td>158</td>
<td>150</td>
<td>140</td>
</tr>
<tr>
<td>150</td>
<td>156</td>
<td>157</td>
<td>156</td>
<td>140</td>
<td>146</td>
<td>156</td>
<td>145</td>
</tr>
<tr>
<td>148</td>
<td>145</td>
<td>146</td>
<td>148</td>
<td>156</td>
<td>160</td>
<td>140</td>
<td>145</td>
</tr>
</tbody>
</table>

Subtract $2^{p-1}$ from each pixel value to create $Spixel$, where $p$ is the number of bits used to represent each pixel.

\[
\text{DCT}(i,j) = \frac{1}{4} C_i C_j \sum_{x=0}^{7} \sum_{y=0}^{7} Spixel(x,y) \cdot \cos \frac{(2x+1)i \cdot \pi}{16} \cdot \cos \frac{(2y+1)j \cdot \pi}{16}
\]

where $C_i, C_j = \frac{1}{\sqrt{2}}$ for $x,y=0$, otherwise $C_i, C_j = 1$.

- The DC coefficient is some multiple of the average value in the 8x8 block.
- The lower-frequency coefficients in the top left corner of the table have larger values than the higher-frequency coefficients, except when there is substantial activity in the image block.
DCT computation features

\[ P[x, y] = 8 \times 8 \text{ matrix of pixel values} \]
\[ F[i, j] = 8 \times 8 \text{ matrix of transformed values/spatial frequency coefficients} \]

\[ \text{In } F[i, j]: \begin{cases} \square & \text{DC coefficient} \\ \square & \text{AC coefficients} \end{cases} \]
\[ f_H = \text{horizontal spatial frequency coefficient} \]
\[ f_V = \text{vertical spatial frequency coefficient} \]
Differential Pulse Code Modulation (DPCM)

- Source coding divides the original data into relevant and irrelevant information, thereby enabling consecutive processing steps to remove the irrelevant data.
  - Source coding can be lossy.

- DPCM, one of the simplest source coding methods, reduces the value range of numerical input characters s.t. successive entropy coding methods might achieve better results. (often applied to audio signals)

Example:
- The sequence, 10, 12, 14, 16, 18, 20, is no good for RLC.
- If DPCM is applied first to yield 10, 2, 2, 2, 2, 2, then RLC will work better to generate 10!52.
DPCM principles

\[ DPCM = PCM - R_0 \]
\[ R_1 = R_0 + DPCM \]

\( R_0 \) = current contents of register \( R \) and \( R_1 \) = new/updated contents
Predictive DPCM

$C_1, C_2, C_3 =$ predictor coefficients
Adaptive DPCM
Linear Predictive Coding (LPC)
Perceptual properties of the human ear
Frequency Masking

Temporal Masking
Motion-Compensated Prediction

\[
\begin{align*}
\text{distance} & \leq \text{threshold} \quad \Rightarrow \quad \text{Transmit the motion vector.} \\
\text{distance} & > \text{threshold} \quad \Rightarrow \quad \text{The block is encoded individually. (i.e., intra-coded)}
\end{align*}
\]
**JPEG Methodology - Image Compression**

- **Discrete Cosine Transform**: It removes data redundancy by transforming data from a *spatial representation* (or spatial domain) to a *spectral representation* (or frequency domain.)

- **Quantizer**: It reduces the precision of the integers, thereby reducing the number of bits required to store the data.

- **Entropy Encoder**: It compresses the quantized data more compactly based on their spatial characteristics (e.g., store the run length instead of 15 zeros.)
JPEG: Quantization

**DCT coefficients**

\[
L(i, j) = \frac{DCT(i, j)}{Quantum(i, j)}
\]

**Quantum matrix**

**DCT coefficients after quantization**
JPEG: Step Size

- The *quantum matrix* contains quantum values which are also called *step sizes*.

- The decision on the relative size of the step sizes is based on how errors in these coefficients will be perceived by the human visual system.

- Quantization errors in the DC and lower-frequency AC coefficients are more easily detectable than quantization errors in the higher-frequency AC coefficients.

  - We use larger step sizes for perceptually less important coefficients.

  - Applications may specify values which optimizes the desired quality according to the particular image characteristics.
Encoding DC Coefficients

- The DC coefficient is some multiple of the average value in the 8x8 block.
- The average pixel value in any 8x8 block will not substantially differ from the average value in the neighboring 8x8 block.
  - DC coefficient values will be quite close.
  - It makes sense to encode the difference between the DC coefficients of neighboring blocks rather than to encode the DC coefficients themselves.

- DC coefficient is encoded as the difference between the current DC coefficient and the previous one.

- The codeword has two fields.
  - The number of bits used to encode the difference.
  - The value of the difference.
Encoding AC Coefficients

- AC coefficients are encoded in the zig-zag order.

- The codeword for a non-zero AC coefficient has three fields:
  - The number of bits used for the presentation of the AC coefficient.
  - The value of the AC coefficient.
  - The number of subsequent zero AC coefficients.
JPEG: Decoding

- JPEG is a symmetrical method.
- Decompression is the exact reverse process of compression.
  - Perform de-quantization.
  - Take the inverse DCT transform.
  - Add $2^{p-1}$ to each pixel.

\[
\text{pixel}(x, y) = \frac{1}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} C_i C_j \text{DCT}(i, j) \cdot \cos \left( \frac{(2x+1)i \cdot \pi}{16} \right) \cdot \cos \left( \frac{(2y+1)j \cdot \pi}{16} \right)
\]

where $C_i, C_j = \frac{1}{\sqrt{2}}$ for $i, j = 0$; otherwise $C_i, C_j = 1$. 

Set of Huffman codewords for the block.

Start-of-frame

Frame header

Frame contents

End-of-frame

Scan header

Scan

---------

Scan

Segment header

Segment

---------

Segment header

Segment

Block

Block

---------

Block

DC

Skip, value

-----

Skip, value

End of block
Video Compression

- In most video sequences there is little change in the contents of the image from one frame to the next frame.
- A good video compression technique should take advantage of this temporal correlation, such as H.261, Motion-JPEG, MPEG(-1, 2, 4), etc.
- Make use of the temporal correlation to remove redundancy.
MPEG Video Standard

<table>
<thead>
<tr>
<th>Frame type</th>
<th>I</th>
<th>B</th>
<th>B</th>
<th>P</th>
<th>B</th>
<th>B</th>
<th>P</th>
<th>B</th>
<th>B</th>
<th>P</th>
<th>B</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>Bitstream order</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

Time to Display 2 3 4 5 6 7 8 9 10 11 12 13 14

- **I frame**: coded without any reference to other frames.
  - ⇒ Compression rate is relatively low.

- **P (predictive coded) frame**: coded using motion-compensated prediction from the last I or P frame, whichever happens to be closest.
  - ⇒ High level of compression.

- **B (bi-directionally predictive coded) frame**: coded using motion-compensated prediction from the most recent P or I frame and the closest future P or I frame.
  - ⇒ Very high level of compression.
Search Area in MPEG

- When searching for the best matching block in a neighboring frame, the region of search depends on the amount of motion.

  ⇒ *The search area grows with the distance between the frame being coded and the frame being used for prediction.*

- **Group of Pictures (GOP):**
  - The frames are organized together in a *group of pictures* (GOP).
  - A GOP is the smallest random access unit in the video sequence.

![Diagram](https://via.placeholder.com/150)

- Use smaller search area
- Use larger search area

I P P P P ... P P P ... P I ...

Need to decode all these frames. *Very long delay!*

I B B P B B P B P B B I ...

Need only decode these frames. *Essentially no delay!*

Playback starts here

One GOP

I P P P P ... P P P ... P I ...

Playback starts here
**MPEG Compression Standard**

- MPEG standard comprises system stream, video stream, audio stream syntactic structures:
  - At the top level, the system syntax offers the multiplexing capability of enveloping the underlying video/audio streams.
  - Video syntax defines how the frames are encoded in a packed bit-stream.
  - Audio syntax describes the similar data encapsulations for audio.

- The following are the byte-aligned delimiters (4 bytes) used across these 3 syntaxes for decoders to recognize each bit-stream segment.

<table>
<thead>
<tr>
<th>Start Code Name</th>
<th>Hexadecimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iso_11172_end_code</td>
<td>0x000001B9</td>
</tr>
<tr>
<td>Pack_start_code</td>
<td>0x000001BA</td>
</tr>
<tr>
<td>System_header_start_code</td>
<td>0x000001BB</td>
</tr>
<tr>
<td>Stream_id byte</td>
<td>(0x000001) BC ~ FF</td>
</tr>
<tr>
<td>Video stream # 0 ~ # 15</td>
<td>'1110 xxxx'</td>
</tr>
<tr>
<td>Audio stream # 0 ~ # 31</td>
<td>'110x xxxx'</td>
</tr>
<tr>
<td>Padding stream ID</td>
<td>'1011 1110'</td>
</tr>
<tr>
<td>Private stream 1</td>
<td>'1011 1101'</td>
</tr>
<tr>
<td>Private stream 2</td>
<td>'1011 1111'</td>
</tr>
<tr>
<td>Reserved stream</td>
<td>'1011 1100'</td>
</tr>
<tr>
<td>Reserved data stream # 0 ~ # 15</td>
<td>'1111 xxxx'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start Code Name</th>
<th>Hexadecimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension_start_code</td>
<td>0x000001B5</td>
</tr>
<tr>
<td>Group_start_code</td>
<td>0x000001B8</td>
</tr>
<tr>
<td>Picture_start_code</td>
<td>0x00000100</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x000001B0</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x000001B1</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x000001B6</td>
</tr>
<tr>
<td>Sequence_end_code</td>
<td>0x000001B7</td>
</tr>
<tr>
<td>Sequence_error_code</td>
<td>0x000001B4</td>
</tr>
<tr>
<td>Sequence_header_code</td>
<td>0x000001B3</td>
</tr>
<tr>
<td>Slice_start_code 1 ~ 175</td>
<td>0x000001 01 ~ AF</td>
</tr>
<tr>
<td>User_data_start_code</td>
<td>0x000001B2</td>
</tr>
</tbody>
</table>
MPEG-1 System Syntax (3 layers)

1. System: ISO 11172 stream = (pack)*((iso_11172_end_code)

2. Pack: pack = (pack header)(system header)* in 1st pack(packet)
   • pack header = (pack_start_code, 0x000001BA)(8 bytes for SCR & Mux_rate)
     • SCR, 33 bits: system clock reference specifying when stream bytes should enter
       the system target decoder (STD), in units of 1/90000 sec.
     • Mux_rate, 22 bits: in units of 50 bytes/sec, byte rate arriving at the STD.
   • system header = (sys_header_start_code)(streams related decoding spec.)

   • packet header = (packet_start_code_prefix, 0x000001)(stream_id, 1 byte)
     (packet_length, N, 2 bytes)( ... )(PTS, DTS, etc.)
     • PTS: presentation time stamp, the time for the stream_id’s data to be rendered.
     • DTS: decoding time stamp, the time for the stream_id’s data to be decoded.
     • stream_id, 0xb8 ~ 0xff, 68 streams: 16 video streams, 32 audio streams, 1
       padding stream, 2 private streams, 1 reserved stream, 16 reserved data
       streams.
   ✓ A typical MPEG stream consists of 1 video stream and 1 audio stream.
MPEG-1 Video Syntax (6 layers)

1. Sequence: video sequence = (sequence header)(GOP)* (seq_end_code)
   - sequence header: more can appear between GOP’s.
     - (seq_header_code, 32 bits)(width, 12 bits)(height, 12 bits)(aspect ratio, 4
       bits for table lookup)(frame rate, 4 bits for table lookup)(bit rate, 18 bits, x
       400bps)(buffer size, constrained flag, intra/inter quant. matrices,
       extension/user data)

2. GOP: GOP = (GOP header)(picture)*
   - GOP header: (GOP_start_code)(time_code, 25 bits)(closed_gop, 1 bit)
     (broken_link, 1 bit)(stuffing, 5 bits)(group_extension_data, 1 byte)*
     (user_data, 1 byte)*

3. Picture: picture = (picture header)(slice)*
   - picture header: (pic_start_code)(temporal_reference, 10 bits)(pic_type, 3 bits
     for table lookup)(vbv_delay for buffer control, motion vectors if P or B, extra
     info, MPEG-2 picture extension/user data)

4. Slice: slice = (slice header)(macroblock)*
   - slice header: (slice_start_code)(quantizer_scale, 5 bits)(extra info, 1 byte)*
   - The slice layer is the lowest layer that can be distinguished through byte-aligned
     start codes.
5. Macroblock:
   - macroblock = (macroblock header)(block 0)if coded... (block 5)if coded (end_of_macroblock)if D-picture
   - macroblock header: (11-bit stuffing)*(escape)*(address_increment, 1-11 bits)(type, 1-6 bits)(etc.)

6. Block: block 0, 1, 2, 3, 4, & 5 (8x8 to DCT).
   - block = (differential DC coef.)if intra(run-level VLC)*(end_of_block)
 Parsing MPEG System Streams

Motivation:

- A MPEG stream consists of individual audio and video streams interleaved in units of packets.
- The coded I-, P- and B-pictures in a video stream have different sensitivities to errors. When transmitted, as specified in RTP, a higher priority can be assigned to I or P pictures.
- With distinct start codes, slices are often used as the transmission units for video streaming.
- Identifying the positions of start codes (re-entry points for decoders) in system streams is the common step of various streaming techniques.
  - Layered transmission schemes deliver a system stream multiplexed in a set of N connections. An FEC code with higher redundancy is applied to the more important connections of greater priority.
  - Choosing a small slice size in I- and P-pictures and a large slice size in B pictures can also improve error resilience for better video playback quality.
Using Lex & Yacc for Stream Analyses

- Since byte-aligned, start codes can be treated as special byte sequences or delimiters in Lex (lexical analyzer).
  - Each start code can be positioned in the video file, and the difference between successive positions signifies the length in bytes.
- A higher-level semantic analysis can be performed by passing tokens from Lex to Yacc.
- According to the designs, a set of grammatical rules can be specified to yield the desirable statistics of stream analyses.
  - An alternative encoded stream output can also be generated by patching the passed tokens.

```c
1. video_sequence(){              // from ISO 11172-2 2.4.2.2
2.   next_start_code();           // find next byte aligned start code
3.   do{                          // do sequence(s)
4.     sequence_header();         // r/w sequence header
5.     do{                        // do group(s) of pictures (GOP)
6.       group_of_pictures();     // r/w group(s) of pictures
7.     }while(nextbits(32)==group_start_code);   //  0x 00 00 01 B8
8.   }while(nextbits(32)==sequence_header_code); //  0x 00 00 01 B3
9.   sequence_end_code(32);       // r/w 0x 00 00 01 B7
10. } /*  end video_sequence() function  */
```
**Algorithms for Parsing Video Sequence**

```
video_sequence()
  -> next_start_code()

sequence_header()
  -> group_of_pictures()
    YES
    -> nextbits(32)=0x000001B3?
      YES
      -> sequence_end_code(0x000001B7
      NO
    NO
    -> nextbits(32)=0x000001B8?
      YES
      -> sequence_end_code(0x000001B7
      NO
      -> sequence_header_code

sequence_header_code
  -> sequence_header_code
    vertical_size
    pel_aspect_ratio
    picture_rate
    bit_rate
    marker_bit
    vbv_buffer_size
    constrained_parameters_flag
    load_intra_quantizer_matrix
    load_intra_quantizer_matrix?
      intra_quantizer_matrix[0]
      intra_quantizer_matrix[63]
    load_non_intra_quantizer_matrix
    load_non_intra_quantizer_matrix?
      non_intra_quantizer_matrix[0]
      non_intra_quantizer_matrix[63]

extension_and_user_data()
  -> extension_and_user_data()
```

Default intra-Q mtx
<table>
<thead>
<tr>
<th>8</th>
<th>16</th>
<th>19</th>
<th>22</th>
<th>26</th>
<th>27</th>
<th>29</th>
<th>34</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>22</td>
<td>24</td>
<td>27</td>
<td>29</td>
<td>34</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>19</td>
<td>22</td>
<td>26</td>
<td>27</td>
<td>29</td>
<td>34</td>
<td>34</td>
<td>37</td>
<td>40</td>
</tr>
<tr>
<td>22</td>
<td>26</td>
<td>27</td>
<td>29</td>
<td>32</td>
<td>35</td>
<td>40</td>
<td>48</td>
<td>58</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>29</td>
<td>32</td>
<td>35</td>
<td>40</td>
<td>48</td>
<td>58</td>
<td>66</td>
</tr>
<tr>
<td>27</td>
<td>29</td>
<td>35</td>
<td>38</td>
<td>46</td>
<td>56</td>
<td>69</td>
<td>83</td>
<td></td>
</tr>
</tbody>
</table>

Default inter-Q mtx
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
Details of Sequence Header

1. sequence_header() { // from ISO 11172-2 2.4.2.3
2.     sequence_header_code(32); // r/w 0x 00 00 01 B3
3.     horizontal_size(12); vertical_size(12); // r/w picture width & height
4.     pel_aspect_ratio(4); picture_rate(4); // r/w aspect ratio, frame rate
5.     bit_rate(18); marker_bit(1); // r/w bit rate, '1',
6.     vbv_buffer_size(10); // r/w video buffer verifier buf.size
7.     constrained_parameters_flag(1); // r/w '1(0)' if (un)constrained
8.     load_intra_quantizer_matrix(1); // r/w flag for intra Q (up to 95 bits)
9.     if(load_intra_Q_matrix) intra_Q_matrix[0..63]; // if flag set, r/w 64 8-bit values
10.    load_inter_quantizer_matrix(1); // r/w flag for inter Q
11.    if(load_inter_Q_matrix) inter_Q_matrix[0..63]; // if flag set, r/w 64 8-bit values
12.    next_start_code(); // find next start code
13.    if(nextbits(32)==extension_start_code) { // if 0x 00 00 01 B5
14.        extension_start_code(32); // r/w extension start code
15.        while(nextbits(24)!= 0x 00 00 01) ext_data(8); // r/w byte of data w/o code prefix
16.        next_start_code(); // find next start code
17.    } /* sequence extension data end */
18.    if(nextbits(32)==user_data_start_code) { // if 0x 00 00 01 B2
19.        user_data_start_code(32); // r/w user data start code
20.        while(nextbits(24)!= 0x 00 00 01) user_data(8); // r/w byte of user data
21.        next_start_code(); // find next start code
22.    } /* user data done */
23. } /* end sequence_header() function */
Lookup Tables

Table 8.2: Ratio of height to width for the 16 pel_aspect_ratio codes

<table>
<thead>
<tr>
<th>Code</th>
<th>height/width</th>
<th>video source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Forbidden</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1.0000</td>
<td>computers (VGA)</td>
</tr>
<tr>
<td>0010</td>
<td>0.6735</td>
<td>16:9, 625-line</td>
</tr>
<tr>
<td>0011</td>
<td>0.7031</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0.7615</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>0.8055</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0.8437</td>
<td>16:9, 525-line</td>
</tr>
<tr>
<td>0111</td>
<td>0.8935</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0.9157</td>
<td>CCIR Rec. 601, 625-line</td>
</tr>
<tr>
<td>1001</td>
<td>0.9815</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>1.0255</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1.0695</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1.0950</td>
<td>CCIR Rec. 601, 525-line</td>
</tr>
<tr>
<td>1101</td>
<td>1.1575</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1.2015</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Picture rate in pictures per second and typical applications

<table>
<thead>
<tr>
<th>Code</th>
<th>Nominal picture rate</th>
<th>typical applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Forbidden</td>
<td>Movies on NTSC broadcast monitors</td>
</tr>
<tr>
<td>0001</td>
<td>23.976</td>
<td>Movies, commercial clips, animation</td>
</tr>
<tr>
<td>0010</td>
<td>24</td>
<td>PAL, SECAM, generic 625/50Hz component video</td>
</tr>
<tr>
<td>0011</td>
<td>25</td>
<td>Broadcast rate NTSC</td>
</tr>
<tr>
<td>0100</td>
<td>29.97</td>
<td>NTSC profession studio, 525/60Hz component video</td>
</tr>
<tr>
<td>0101</td>
<td>30</td>
<td>Non-interlaced PAL/SECAM/625 video</td>
</tr>
<tr>
<td>0110</td>
<td>50</td>
<td>Non-interlaced broadcast NTSC</td>
</tr>
<tr>
<td>0111</td>
<td>59.94</td>
<td>Non-interlaced studio 525 NTSC rate</td>
</tr>
<tr>
<td>1000</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.4: Constrained parameters bounds.

- horizontal_size \(\leq 768\) pels; vertical_size \(\leq 576\) lines.
- number of macroblocks \(\leq 396\).
- \((\text{number of macroblocks}) \times \text{picture_rate} \leq 396 \times 25\).
- picture_rate \(\leq 30\) pictures per second.
- vbv_buffer_size \(\leq 160\); bit_rate \(\leq 4640\).
- forward_f_code \(\leq 4\); backward_f_code \(\leq 4\).
Algorithms for Ext/User Data & GOP

extension_and_user_data()

next_start_code()

nextbits(32)=0x 00 00 01 B5?

Yes

extension_start_code

nextbits(24)=0x000001?

byte of extension data

No

32 [0x 00 00 01 B5]

nextbits(32)=0x 00 00 01 B2?

Yes

user_data_start_code

nextbits(24)=0x000001?

byte of user data

No

next_start_code()

group_of_pictures()

group_start_code

drop_frame_flag

't' if 29.97 Hz

drop1/1000

1

5

6

6

6

1

1

extension_and_user_data()

picture

picture data

nextbits(32)=0x 00 00 01 00?

Yes

Netxbits(32)=0x 00 00 01 B8

No

Done

Netxbits(32)=0x 00 00 01 B5?

Yes

extension_start_code

nextbits(24)=0x000001?

byte of extension data

No

32 [0x 00 00 01 B5]

nextbits(32)=0x 00 00 01 B2?

Yes

user_data_start_code

nextbits(24)=0x000001?

byte of user data

No

next_start_code()

32 [0x 00 00 01 B8]

drop1/1000

1

5

6

6

6

1

1

extension_and_user_data()

picture

picture data

nextbits(32)=0x 00 00 01 00?

Yes

Netxbits(32)=0x 00 00 01 B5

No

Done
Group of Pictures

1. `group_of_pictures(){` // from ISO 11172-2 2.4.2.4
2. `group_start_code(32);` // r/w 0x 00 00 01 B8
3. `time_code(25);` // r/w SMPTE time code
4. `closed_gop(1);` // r/w '1' if closed, '0' if open
5. `broken_link(1);` // r/w normally '0', '1' if broken
6. `next_start_code();` // find next start code
7. `if(nextbits(32)==0x 00 00 01 B5) { extension_start_code(32); ` // r/w this code
8. `while(nextbits(24)!= 0x 00 00 01) {` // while not start code prefix
9. `group_extension_data(8);` // r/w byte of data
10. `}` /* group extension data done */
11. `next_start_code();` // find next start code
12. `}
13. `if(nextbits(32)==0x 00 00 01 B2) { use_data_start_code(32); ` // r/w this code
14. `while(nextbits(24)!= 0x 00 00 01) {` // while not start code prefix
15. `user_data(8);` // r/w byte of data
16. `}` /* group user data done */
17. `next_start_code();` // find next start code
18. `}
19. `do{` // do picture(s)
20. `picture();` // encode/decode picture
21. `}while(nextbits(32)==picture_start_code) // while 0x 00 00 01 00
22. `}` /* end group_of_pictures function */
Pictures from ISO 11172-2 2.4.2.5

1. picture(){ picture_start_code(32); // r/w 0x 00 00 01 00
2.   temporal_reference(10); // r/w picture count modulo 1024
3.   type(3); vbv_delay(16); // r/w picture type & VBV buffer delay
4.   if(type==2||type==3){ // if P or B type, need forward motion vector
5.     full_pel_forward_vector(1); // r/w 1=full pel, 0=half pel
6.     forward_f_code(3); } // r/w fwd motion vector range (scaling)
7.   if(type==3){ full_pel_backward_vector(1); backward_f_code(3); } // B: need bkwd mv
8.   while(nextbits(1)=='1'){ // while '1', extra information
9.     extra_bit_picture(1); extra_information_picture(8); } // r/w '1' & byte of info
10. extra_bit_picture(1); // r/w '0' to end extra information
11. next_start_code(); // find next start code
12. if(nextbits(32)==extension_start_code){ // if 0x 00 00 01 B5
13.   extension_start_code(32); // r/w extension start code
14.   while(nextbits(24)!=0x000001){ picture_extension_data(8); } // r/w byte of data
15.   next_start_code(); // find next start code
16. }
17. if(nextbits(32)==user_data_start_code){ // if 0x 00 00 01 B2
18.   user_data_start_code(32); // r/w user data start code
19.   while(nextbits(24)!=0x000001){ user_data(8); } // r/w byte of user data
20.   next_start_code(); // find next start code
21. }
22. do{ slice(); }while(nextbits(32)==slice_start_code) // slice(s): 0x 00 00 01 01~AF
23. } /* end picture() function */
Details of Pictures

Table 8.5: Picture type codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Picture type</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>forbidden</td>
</tr>
<tr>
<td>001</td>
<td>I-picture</td>
</tr>
<tr>
<td>010</td>
<td>P-picture</td>
</tr>
<tr>
<td>011</td>
<td>B-picture</td>
</tr>
<tr>
<td>100</td>
<td>D-picture</td>
</tr>
<tr>
<td>101</td>
<td>reserved</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>111</td>
<td>reserved</td>
</tr>
</tbody>
</table>

- forbidden
- I-picture
- P-picture
- B-picture
- D-picture
- reserved

- Picture type codes:
  - I-picture: 001
  - P-picture: 010
  - B-picture: 011
  - D-picture: 100
  - reserved: 101
  - reserved: 111

- Extra information picture:
  - extra_bit_picture '1': 0x 00 01
  - extra_bit_picture '0': 0x 00 00

- Picture coding type:
  - I=1, P, B, D
  - VBV delay (decoding delay)
  - Full pel forward vector (full=1, half)
  - Forward f code (scaling)
  - Full pel backward vector
  - Backward f code

- Slice:
  - Picture start code
  - Temporal reference (% 1024)
  - Picture coding type
  - MPEG-II

- Extension and user data:
  - Nextbits(32)=0x 00 00 01 01~AF?
  - Done
Slices

1. slice(){ // from ISO 11172-2 2.4.2.6
2.  slice_start_code(32); // r/w 0x 00 00 01 01~AF [1~175th macroblock row]
3.  quantizer_scale(5); // r/w quantizer scale
4.  while(nextbits(1)=='1') { // while '1', extra slice info [never in MPEG-I]
5.      extra_bit_slice(1); // r/w '1'
6.      extra_information_slice(8); // r/w byte of extra information
7.  } /* end - extra slice info */
8.  extra_bit_slice(1); // r/w '0' to end extra slice info
9. do{ macroblock(); // process a macroblock
10. } while(nextbits(23)!=0) // do while not 23 zeros
11. next_start_code(); // find next start code
12. } /* end - slice() function */
Macroblocks

1. macroblock(){ // ISO 11172-2 2.4.2.7
2. while(nextbits(11)=='00000001111') m_stuffing(11); // r/w macroblock stuffing
3. while(nextbits(11)=='00000001000') m_escape(11); // r/w macroblock escape
4. macroblock_address_increment(1-11); // r/w VLC for mb address
5. macroblock_type(1-6); // r/w VLC for mb type
6. if(macroblock_quant) quantizer_scale(5); // for changing quantizer scale
7. if(macroblock_motion_forward){ // if forward motion vector
8. motion_horizontal_forward_code(1-11); // r/w VLC for fwd h. mv
9. if(forward_f>1)&&(motion_horizontal_forward_code!=0) // if fwd. h. mv
10. motion_horizontal_forward_r(1-6); // r/w residual of h. mv
11. motion_vertical_forward_code(1-11); // r/w VLC for fwd. v. mv
12. if(forward_f>1)&&(motion_vertical_forward_code!=0) /* if fwd. v. mv
13. motion_vertical_forward_r(1-6); // r/w residual of v. mv
14. } /* end if forward motion vect */
15. if(macroblock_motion_backward){ /* if backward motion vector
16. motion_horizontal_backward_code(1-11); // r/w VLC for bkwd h. mv
17. if(backward_f!=1)&&(motion_horizontal_backward_code!=0) // if bkwd. h. mv
18. motion_horizontal_backward_r(1-6); // r/w residual of h. mv
19. motion_vertical_backward_code(1-11); // r/w VLC for bkwd. v. mv
20. if(backward_f!=1)&&(motion_vertical_backward_code!=0) // if bkwd. v. mv
21. motion_vertical_backward_r(1-6); // r/w residual of v. mv
22. } /* end If backward motion vect */
23. if(macro_block_pattern) code_block_pattern(3-9); // r/w coded block pattern, if any
24. for(i=0; i<6; i++) block(i); // r/w block data for possible of the 6 blocks
25. if(picture_coding_type==4) end_of_macroblock(1); // if D-picture, r/w '1'-end of mb
26. } /* end macroblock() function */
Algorithm for Macroblocks

```
macroblock()
  Yes
  netxbits(11)=B'00000001111'?
  No
  macroblock_stuffing
    Yes
    netxbits(11)=B'00000001000'?
    No
    macroblock_escape
      Yes
      macroblock_address_increment
      Yes
      macroblock_address_increment
      No
      1-6
      0
      1
      5
    No
    11
  No
  11
  1-11
  motion_vectors()
    2
    3
    1-9
    0
    1
    3
    4
    i=0
    i<6?
    No
    Yes
    block(i)
      5
      j=i+1
      not 4
    4
    1
    end_of_macroblock '1'
```

Table 8.6: Variable length codes (VLC) for macroblock_address_increment.

<table>
<thead>
<tr>
<th>increment value</th>
<th>macroblock_address_increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>macroblock_escape</td>
<td>0000 0001 000 (+= 33)</td>
</tr>
<tr>
<td>macroblock_stuffing</td>
<td>0000 0001 111</td>
</tr>
<tr>
<td>33</td>
<td>0000 0011 000</td>
</tr>
<tr>
<td>32</td>
<td>0000 0011 001</td>
</tr>
<tr>
<td>31</td>
<td>0000 0011 010</td>
</tr>
<tr>
<td>30</td>
<td>0000 0011 011</td>
</tr>
<tr>
<td>29</td>
<td>0000 0011 100</td>
</tr>
<tr>
<td>28</td>
<td>0000 0011 101</td>
</tr>
<tr>
<td>27</td>
<td>0000 0011 110</td>
</tr>
<tr>
<td>26</td>
<td>0000 0011 111</td>
</tr>
<tr>
<td>25</td>
<td>0000 0100 000</td>
</tr>
<tr>
<td>24</td>
<td>0000 0100 001</td>
</tr>
<tr>
<td>23</td>
<td>0000 0100 010</td>
</tr>
<tr>
<td>22</td>
<td>0000 0100 011</td>
</tr>
<tr>
<td>21</td>
<td>0000 0100 100</td>
</tr>
<tr>
<td>20</td>
<td>0000 0100 101</td>
</tr>
<tr>
<td>19</td>
<td>0000 0101 000</td>
</tr>
<tr>
<td>18</td>
<td>0000 0101 001</td>
</tr>
<tr>
<td>17</td>
<td>0000 0101 010</td>
</tr>
<tr>
<td>16</td>
<td>0000 0101 011</td>
</tr>
<tr>
<td>15</td>
<td>0000 0101 100</td>
</tr>
<tr>
<td>14</td>
<td>0000 0101 101</td>
</tr>
<tr>
<td>13</td>
<td>0000 0110 000</td>
</tr>
<tr>
<td>12</td>
<td>0000 0110 001</td>
</tr>
<tr>
<td>11</td>
<td>0000 0110 010</td>
</tr>
<tr>
<td>10</td>
<td>0000 0110 100</td>
</tr>
<tr>
<td>9</td>
<td>0000 0110 101</td>
</tr>
<tr>
<td>8</td>
<td>0000 0111 000</td>
</tr>
<tr>
<td>7</td>
<td>0000 0111 001</td>
</tr>
<tr>
<td>6</td>
<td>0000 0111 010</td>
</tr>
<tr>
<td>5</td>
<td>0000 0111 011</td>
</tr>
<tr>
<td>4</td>
<td>0000 0111 100</td>
</tr>
<tr>
<td>3</td>
<td>0000 0111 101</td>
</tr>
<tr>
<td>2</td>
<td>0000 0111 110</td>
</tr>
<tr>
<td>1</td>
<td>0000 0111 111</td>
</tr>
</tbody>
</table>
Table 8.7: VLC for macroblock_type in I-, P-, B-, and D-pictures.

<table>
<thead>
<tr>
<th>macro-block_type</th>
<th>macro-block_pattern</th>
<th>macro-block_motion_backward</th>
<th>macro-block_motion_forward</th>
<th>macro-block_quant</th>
<th>VLC code (1~6 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0000 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0001 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0000 01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0000 11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0000 10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0001 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0000 01</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

56
Algorithm for Motion Vectors

Table 8.9: Variable length codes for the motion codes.
- "s" is 0 for positive motion values and 1 for negative motion values.

<table>
<thead>
<tr>
<th>motion code VLC</th>
<th>motion value magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>01s</td>
</tr>
<tr>
<td>2</td>
<td>001s</td>
</tr>
<tr>
<td>3</td>
<td>0001 s</td>
</tr>
<tr>
<td>4</td>
<td>0000 11s</td>
</tr>
<tr>
<td>5</td>
<td>0000 101s</td>
</tr>
<tr>
<td>6</td>
<td>0000 100s</td>
</tr>
<tr>
<td>7</td>
<td>0000 011s</td>
</tr>
<tr>
<td>8</td>
<td>0000 0101 1s</td>
</tr>
<tr>
<td>9</td>
<td>0000 0101 0s</td>
</tr>
<tr>
<td>10</td>
<td>0000 0100 1s</td>
</tr>
<tr>
<td>11</td>
<td>0000 0100 01s</td>
</tr>
<tr>
<td>12</td>
<td>0000 0100 00s</td>
</tr>
<tr>
<td>13</td>
<td>0000 0011 11s</td>
</tr>
<tr>
<td>14</td>
<td>0000 0011 10s</td>
</tr>
<tr>
<td>15</td>
<td>0000 0011 01s</td>
</tr>
<tr>
<td>16</td>
<td>0000 0011 00s</td>
</tr>
</tbody>
</table>

/* read/write horizontal forward MV VLC */
/* if forward_f=1, no residue bits */
/* if code is 0, no residue bits */
/* read/write complement of residue */
/* vertical calculation identical to horizontal */
/* if backward motion vector */
/* read/write horizontal backward MV VLC */
/* if backward_f=1, no residue bits */
/* if code is 0, no residue bits */
/* read/write complement of residue */
/* vertical calculation identical to horizontal */
Table 8.8: MPEG-1 code_block_pattern (cbp) VLC codes.

- Blocks labeled “.” (bit=0) are skipped, whereas blocks labeled “c” (bit=1) are coded.

<table>
<thead>
<tr>
<th>cbp</th>
<th>Block #</th>
<th>code_block_pattern</th>
<th>VLC code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>012345</td>
<td>YYYY CrCr</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000000</td>
<td>. . . . C</td>
<td>Forbidden</td>
</tr>
<tr>
<td>1</td>
<td>000001</td>
<td>. . . C</td>
<td>0101 1</td>
</tr>
<tr>
<td>2</td>
<td>000010</td>
<td>. . C</td>
<td>0100 1</td>
</tr>
<tr>
<td>3</td>
<td>000011</td>
<td>. C C</td>
<td>0011 01</td>
</tr>
<tr>
<td>4</td>
<td>000100</td>
<td>C . .</td>
<td>1101</td>
</tr>
<tr>
<td>5</td>
<td>000101</td>
<td>C . C</td>
<td>0010 111</td>
</tr>
<tr>
<td>6</td>
<td>000110</td>
<td>C C .</td>
<td>0010 011</td>
</tr>
<tr>
<td>7</td>
<td>000111</td>
<td>C C C</td>
<td>0001 1111</td>
</tr>
<tr>
<td>8</td>
<td>001000</td>
<td>. C .</td>
<td>1100</td>
</tr>
<tr>
<td>9</td>
<td>001001</td>
<td>. C C</td>
<td>0010 110</td>
</tr>
<tr>
<td>10</td>
<td>001010</td>
<td>. C C</td>
<td>0010 010</td>
</tr>
<tr>
<td>11</td>
<td>001011</td>
<td>. C C</td>
<td>0001 1110</td>
</tr>
<tr>
<td>12</td>
<td>001100</td>
<td>. C C</td>
<td>1001 1</td>
</tr>
<tr>
<td>13</td>
<td>001101</td>
<td>. C C</td>
<td>0001 1011</td>
</tr>
<tr>
<td>14</td>
<td>001110</td>
<td>. C C C</td>
<td>0001 0111</td>
</tr>
<tr>
<td>15</td>
<td>001111</td>
<td>. C . C</td>
<td>0001 0011</td>
</tr>
<tr>
<td>16</td>
<td>010000</td>
<td>. C . C .</td>
<td>1011</td>
</tr>
<tr>
<td>17</td>
<td>010001</td>
<td>. C . C .</td>
<td>0010 101</td>
</tr>
<tr>
<td>18</td>
<td>010010</td>
<td>. C . C .</td>
<td>0010 001</td>
</tr>
<tr>
<td>19</td>
<td>010011</td>
<td>. C . C .</td>
<td>0001 1101</td>
</tr>
<tr>
<td>20</td>
<td>010100</td>
<td>. C . C .</td>
<td>1000 1</td>
</tr>
<tr>
<td>21</td>
<td>010101</td>
<td>. C . C .</td>
<td>0001 1001</td>
</tr>
<tr>
<td>22</td>
<td>010110</td>
<td>. C . C .</td>
<td>0001 0101</td>
</tr>
<tr>
<td>23</td>
<td>010111</td>
<td>. C . C .</td>
<td>0001 0001</td>
</tr>
<tr>
<td>24</td>
<td>011000</td>
<td>. C . C . .</td>
<td>0011 1</td>
</tr>
<tr>
<td>25</td>
<td>011001</td>
<td>. C . C . .</td>
<td>0000 1111</td>
</tr>
<tr>
<td>26</td>
<td>011010</td>
<td>. C . C . .</td>
<td>0000 1101</td>
</tr>
<tr>
<td>27</td>
<td>011011</td>
<td>. C . C . .</td>
<td>0000 0001</td>
</tr>
<tr>
<td>28</td>
<td>011100</td>
<td>. C . C . . .</td>
<td>0111 1</td>
</tr>
<tr>
<td>29</td>
<td>011101</td>
<td>. C . C . . .</td>
<td>0000 1011</td>
</tr>
<tr>
<td>30</td>
<td>011110</td>
<td>. C . C . . .</td>
<td>0000 0111</td>
</tr>
<tr>
<td>31</td>
<td>011111</td>
<td>. C . C . . .</td>
<td>0000 0011 1</td>
</tr>
</tbody>
</table>
1. block(i){ // from ISO 1172-2 2.4.2.8
2.    if(pattern_code[i]){ // if ith block coded [Intra-coded macroblock has all blocks]
3.        if(macroblock_intra){ // intra-coded macroblock in I, D, P or B
4.            if(i<4){ // luminance blocks
5.                dct_dc_size_luminance(2-7) // r/w VLC for Y size: Table 5.3
6.                if(dc_size_luminance!=0) // if Y size not zero
7.                    dct_dc_differential(1-8); // r/w size bits of diff. DC
8.            } else { // chrominance blocks
9.                dct_dc_size_chrominance(2-7) // r/w VLC for Cb or Cr size: Table 5.3
10.               if(dc_size_chrominance!=0) // if Cb or Cr size not zero
11.                  dct_dc_differential(1-8); // r/w size bits of diff. DC
12.            }
13.        } else { // inter-coded macroblock in P or B
14.            dct_coeff_first(2-28); // r/w VLC 1st run-level
15.        }
16.    } else { // if not D-picture
17.        if (picture_coding_type!=4){ // if not D-picture
18.            while (nextbits(2)!='10') // while not end-of-block
19.                dct_coeff_next(3-28); // r/w VLC next run-level → zigzag scan
20.        }
21.    }
22. } /* end block(i) function */
Algorithm for Blocks

macroblock_int

i>=4
Table 8.10: Video syntax data element summary.

<table>
<thead>
<tr>
<th>video data element name</th>
<th>set</th>
<th>used</th>
<th>dt</th>
<th># of bits</th>
<th>value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>backward_f_code</td>
<td>.</td>
<td>.</td>
<td></td>
<td>3</td>
<td>1..7</td>
</tr>
<tr>
<td>backward_f</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit_rate</td>
<td>.</td>
<td>.</td>
<td>U</td>
<td>18</td>
<td>1...0x3FFF</td>
</tr>
<tr>
<td>broken_link</td>
<td>U</td>
<td>0</td>
<td></td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>closed_gop</td>
<td>U</td>
<td>0</td>
<td></td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>coded_block_pattern</td>
<td>V</td>
<td>3-9</td>
<td></td>
<td>1...63</td>
<td></td>
</tr>
<tr>
<td>pattern_code[0]</td>
<td>.</td>
<td>.</td>
<td>V</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>pattern_code[5]</td>
<td>.</td>
<td>.</td>
<td>U</td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>constrained_parameters_flag</td>
<td>U</td>
<td>1</td>
<td></td>
<td>0,1</td>
<td></td>
</tr>
<tr>
<td>dct_coeff_first</td>
<td>V</td>
<td>2-28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dct_coeff_next</td>
<td>V</td>
<td>3-28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dct_dc_differential</td>
<td>U</td>
<td>1-8</td>
<td></td>
<td>-255...1...255</td>
<td></td>
</tr>
<tr>
<td>dct_dc_size_chrominance</td>
<td>V</td>
<td>2-8</td>
<td></td>
<td>0,8</td>
<td></td>
</tr>
<tr>
<td>dct_dc_size_luminance</td>
<td>V</td>
<td>2-7</td>
<td></td>
<td>0...8</td>
<td></td>
</tr>
<tr>
<td>end_of_block</td>
<td>V</td>
<td>2</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>end_of_macroblock</td>
<td>V</td>
<td>3</td>
<td></td>
<td>1...7</td>
<td></td>
</tr>
<tr>
<td>extension_start_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001B5</td>
<td></td>
</tr>
<tr>
<td>extra_bit_picture</td>
<td>U</td>
<td>1</td>
<td></td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>extra_bit_slice</td>
<td>U</td>
<td>1</td>
<td></td>
<td>0</td>
<td>reserved</td>
</tr>
<tr>
<td>extra_information_picture</td>
<td></td>
<td>8</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>extra_information_slice</td>
<td></td>
<td>8</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>forward_f_code</td>
<td>U</td>
<td>3</td>
<td></td>
<td>1...7</td>
<td></td>
</tr>
<tr>
<td>full_pel_backward_vector</td>
<td>.</td>
<td>.</td>
<td>U</td>
<td>1</td>
<td>0,1</td>
</tr>
<tr>
<td>full_pel_forward_vector</td>
<td>.</td>
<td>.</td>
<td>U</td>
<td>1</td>
<td>0,1</td>
</tr>
<tr>
<td>group_extension_data</td>
<td>U</td>
<td>8</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>group_start_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001B8</td>
<td></td>
</tr>
<tr>
<td>horizontal_size</td>
<td>U</td>
<td>12</td>
<td></td>
<td>1...4095</td>
<td></td>
</tr>
<tr>
<td>intra_quantizer_matrix[0]</td>
<td>U</td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>intra_quantizer_matrix[63]</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>load_intra_quantizer_matrix</td>
<td>U</td>
<td>8</td>
<td></td>
<td>1...255</td>
<td></td>
</tr>
<tr>
<td>load_intra_quantizer_matrix[0]</td>
<td>U</td>
<td>8</td>
<td></td>
<td>1...255</td>
<td></td>
</tr>
<tr>
<td>load_non_intra_quantizer_matrix</td>
<td>U</td>
<td>8</td>
<td></td>
<td>1</td>
<td>reserved</td>
</tr>
<tr>
<td>macroblock_address_increment</td>
<td>V</td>
<td>1-11</td>
<td></td>
<td>1...33</td>
<td></td>
</tr>
<tr>
<td>macroblock_escape</td>
<td>V</td>
<td>11</td>
<td></td>
<td>00000001000</td>
<td></td>
</tr>
<tr>
<td>macroblock_data</td>
<td>V</td>
<td>11</td>
<td></td>
<td>00000001111</td>
<td></td>
</tr>
<tr>
<td>macroblock_type</td>
<td>V</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>macroblock_intra</td>
<td>V</td>
<td>1-6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>macroblock_motion_backward</td>
<td>V</td>
<td>1-6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>macroblock_motion_forward</td>
<td>V</td>
<td>1-6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>macroblock_pattern</td>
<td>V</td>
<td>1-6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>macroblock_quant</td>
<td>V</td>
<td>1-6</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>marker_bit</td>
<td>B</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>motion_horizontal_backward_code</td>
<td>V</td>
<td>1-11</td>
<td></td>
<td>-16...16</td>
<td></td>
</tr>
<tr>
<td>motion_horizontal_r_backward</td>
<td>U</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>motion_horizontal_forward</td>
<td>U</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>motion_horizontal_forward_r</td>
<td>U</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>motion_vertical_backward_code</td>
<td>V</td>
<td>1-11</td>
<td></td>
<td>-16...16</td>
<td></td>
</tr>
<tr>
<td>motion_vertical_r_backward</td>
<td>U</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>motion_vertical_forward</td>
<td>U</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>motion_vertical_forward_r</td>
<td>U</td>
<td>1-6</td>
<td></td>
<td>0..63</td>
<td></td>
</tr>
<tr>
<td>non_intra_quantizer_matrix[0]</td>
<td>U</td>
<td>1</td>
<td></td>
<td>1255</td>
<td></td>
</tr>
<tr>
<td>non_intra_quantizer_matrix[63]</td>
<td>U</td>
<td>1</td>
<td></td>
<td>1255</td>
<td></td>
</tr>
<tr>
<td>pel_aspect_ratio</td>
<td>U</td>
<td>1</td>
<td></td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>picture_coding_type</td>
<td>U</td>
<td>3</td>
<td></td>
<td>1</td>
<td>reserved</td>
</tr>
<tr>
<td>picture_extension_data</td>
<td>U</td>
<td>3</td>
<td></td>
<td>1.4</td>
<td>reserved</td>
</tr>
<tr>
<td>picture_rate</td>
<td>U</td>
<td>4</td>
<td></td>
<td>1</td>
<td>reserved</td>
</tr>
<tr>
<td>picture_start_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x00000100</td>
<td></td>
</tr>
<tr>
<td>quantizer_scale</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001B7</td>
<td></td>
</tr>
<tr>
<td>sequence_end_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001B3</td>
<td></td>
</tr>
<tr>
<td>sequence_extension_data</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x00000101</td>
<td></td>
</tr>
<tr>
<td>sequence_header_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001xx</td>
<td></td>
</tr>
<tr>
<td>slice_start_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001AF</td>
<td></td>
</tr>
<tr>
<td>slice_start_code_175</td>
<td></td>
<td>10</td>
<td></td>
<td>0...1023</td>
<td></td>
</tr>
<tr>
<td>temporal_reference</td>
<td>U</td>
<td>10</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>time_code</td>
<td>U</td>
<td>25</td>
<td></td>
<td>0...255</td>
<td></td>
</tr>
<tr>
<td>user_data</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001B2</td>
<td></td>
</tr>
<tr>
<td>user_data_start_code</td>
<td>B</td>
<td>32</td>
<td></td>
<td>0x000001AF</td>
<td></td>
</tr>
<tr>
<td>vbv_buffer_size</td>
<td>U</td>
<td>16</td>
<td></td>
<td>0..0xFFF</td>
<td></td>
</tr>
<tr>
<td>vbv_delay</td>
<td>U</td>
<td>12</td>
<td></td>
<td>2.4...4094</td>
<td></td>
</tr>
</tbody>
</table>
An Example

A simple source image comprising two macroblocks, with $\gamma_{b,c}$ values = 128, was compressed to create a simple MPEG-1 video stream.

1. Compressed data (hexadecimal format): 46 bytes
   000001B302001014FFFFE0A0
   000001B888080040000001
   0000FFFFFF800000101FA96529488AA2529488800001B7

2. Compressed data (binary format):
   00000000 00000000 00000001 10110011 00000100 00000000 00100000 00101000
   11111111 11111111 11100000 10100000 00000000 00000000 00100000 00100000
   00000000 00001111 11111111 11110000 00000000 00000000 00000001 00000001
   11111010 10010110 01010010 10010100 10001000 10101010 00100101 00101001
   01001010 10001000 00000000 00000000 00000001 10110111

3. Sequence header: 0x000001B302001014FFFFE0A0

   sequence_header_code
   horizontal_size = 32 pels
   vertical_size = 16 pels
   pel_aspect_ratio = 1
   picture_rate = 4
   bit_rate = 0x3fff (variable)
   marker bit = 1
   vbv_buffer_size = 20
   constrained_parameters_flag = 0
   load_intra_quantizer_matrix = 0
   load_inter_quantizer_matrix = 0

Parsed...
Parsing GOP

1. Compressed data (hexadecimal format): 46 bytes
   000001B302001014FFFE0A000001B888080040000001
   00000FFFF80000101FA96529488AA252948880000001B7

2. Compressed data (binary format):
   00000000 00000000 00000001 10110011 00000010 00000000 00010000 00010100
   1111111 1111111 11100000 10100000 00000000 00000000 00000001 10111000
   1000000 00001000 00000000 01000000 00000000 00000000 00000001 00000000
   11111010 10010110 01010010 10010100 10001000 10101010 00100101 00101001
   01001000 10001000 00000000 00000000 00000001 10110111

1. Group_of_pictures header 0x000001B880080040

2. 0000000 00000000 00000001 10111000
   group_start_code

3. time_code:
   1
   drop_frame_flag
   00000
   time_code_hours = 0

5. 00 0000
   time_code_minutes = 0

6. 1
   marker_bit

7. 000 000
   time_code_seconds = 0

8. 0000 0
   time_code_pictures = 0

9. 1
   closed_gop = 1

10. 0
    broken_link = 0

12. 00000
    stuffed bits to byte boundary
# Parsing Picture & Slice

1. Compressed data (hexadecimal format): 46 bytes
2. 000001B302001014FFFFE0A0000001B888080040000001
3. 0000FFFF80000101FA96529488AA25294888000001B7

4. Compressed data (binary format):
5. 00000000 00000000 00000001 10110011 00000010 00000000 00100000 00101000
6. 11111111 11111111 11100000 10100000 00000000 00000000 00000001 10111000
7. 10000000 00001000 00000000 01000000 00000000 00000000 00000000 00000000
8. 00000000 00001111 11111111 11110000 00000000 00000000 00000001 00000001
9. 11111010 10010110 01010010 10010100 10001000 10101010 00100101 00101001
10. 01001000 10001000 00000000 00000000 00000001 10110111

<table>
<thead>
<tr>
<th>Picture Header: 0x00000010000000FFFF8</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 00000000 00000001 00000000</td>
</tr>
<tr>
<td>00000000 00</td>
</tr>
<tr>
<td>001</td>
</tr>
<tr>
<td>111 11111111 11111</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slice Header: 0x00000101FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 00000000 00000001 00000001</td>
</tr>
<tr>
<td>slice_vertical_position = 1</td>
</tr>
<tr>
<td>macroblock_address = -1 (reset)</td>
</tr>
<tr>
<td>11111</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
### Parsing the others

1. Compressed data (hexadecimal format): 46 bytes
2. 000001B302001014FFFFFFE0A0000001B888080040000001
3. 00000FFFFFF800000101FA_96529488AA25294888000001B7

<table>
<thead>
<tr>
<th>Macroblock, Block and Sequence End Code</th>
<th>Compressed Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.  10 10010110 01010010 10010100 10001000 10101010 00100101 00101001 00000000 00000000 00000001 10110111</td>
<td></td>
</tr>
</tbody>
</table>

#### Macroblock 1

- **Macroblock Address Increment**: 1
- **Macroblock Type**: i+q
- **Macroblock Intra**: 1
- **Macroblock Quant**: 1
- **CPB (I-Picture)**: '1111 11'
- **Quantizer Scale**: 5
- **DCT DC Size Luminance**: Y0
- **End of Block Y0**: 0
- **DCT DC Size Luminance**: Y1
- **End of Block Y1**: 0
- **DCT DC Size Luminance**: Y2
- **End of Block Y2**: 0
- **DCT DC Size Luminance**: Y3
- **End of Block Y3**: 0
- **DCT DC Size Chrominance**: Cb4
- **End of Block Cb4**: 0
- **DCT DC Size Chrominance**: Cr5
- **End of Block Cr5**: 0
- **Stuffed Bits to Byte Boundary**

#### Macroblock 2

- **Macroblock Address Increment**: 1
- **Macroblock Type**: i+q
- **Macroblock Intra**: 1
- **Macroblock Quant**: 1
- **CPB (I-Picture)**: '1111 11'
- **Quantizer Scale**: 8
- **DCT DC Size Luminance**: Y0
- **End of Block Y0**: 0
- **DCT DC Size Luminance**: Y1
- **End of Block Y1**: 0
- **DCT DC Size Luminance**: Y2
- **End of Block Y2**: 0
- **DCT DC Size Luminance**: Y3
- **End of Block Y3**: 0
- **DCT DC Size Chrominance**: Cb4
- **End of Block Cb4**: 0
- **DCT DC Size Chrominance**: Cr5
- **End of Block Cr5**: 0
- **Sequence End Code**
**Motion Compensation**

- **Main issues in motion compensation:**
  - The precision of the motion vectors,
  - The size of regions assigned to a single motion vector, and
  - The criteria used to select the best motion vector value.
    - A full search of the predicting image is used to find the best motion vector.
    - Optical flow technique can be used to derive the motion vectors.
    - Criteria: MAD (mean absolute distortion), MSE (mean square error), etc.
  - MAD is defined for a 16×16 pel macroblock:
    - $V_n(x+i, y+j)$ at macroblock position $(x, y)$ in source picture $n$.
    - $V_m(x+dx+i, y+dy+j)$ at macroblock position $(x+dx, y+dy)$ in reference picture $m$.
      - $(x, y)$ refers to the upper left corner of the macroblock, $(i, j)$ refers to the values to the right and down, and $(dx, dy)$ are the displacement.

\[
MAD(x, y) = \frac{1}{256} \sum_{i=0}^{15} \sum_{j=0}^{15} |V_n(x+i, y+j) - V_m(x+dx+i, y+dy+j)|
\]

- SAD (sum of absolute distortions): MAD × 256.
- MSE:

\[
MSE(x, y) = \frac{1}{256} \sum_{i=0}^{15} \sum_{j=0}^{15} (V_n(x+i, y+j) - V_m(x+dx+i, y+dy+j))^2
\]
Motion-Compensation Prediction

- Each macroblock has an associated motion vector.
- The vectors of adjacent macroblocks are highly correlated.
  - The horizontal or vertical motion vector displacement, \( \mathbf{MD} \), is predicted from the one of the preceding macroblock, \( \mathbf{PMD} \), in the slice. \( \Rightarrow \) only the difference, \( \mathbf{dMD} \), is coded.
    - \( \mathbf{dMD} = \mathbf{MD} - \mathbf{PMD} \).
- The predicted \( \mathbf{MD} \)'s are reset to 0 in P-pictures, when:
  - at the start of a slice, after a macroblock is intra-coded, upon skipping a macroblock, upon a zero macro-block_motion_forward.
- There are some other rules applied to P or B-pictures.
  - For instance, it is not permitted to have \( \mathbf{MD} \) falling outside of the reference picture.
  - The prediction (B-pictures):
    \[
    \text{pel}[i][j] = (\text{pel}_{\text{for}}[i][j]+ \text{pel}_{\text{back}}[i][j])//2 \quad \Rightarrow \text{odd} \ #
    \]
Motion Displacements

- **MD** = principal + residual
  - **full_pel_vector** (full_pel_forward_vector/...backward...): 0, 1.
  - **f_code** (forward_f_code/...backward...): 1~7, [range] in picture headers.
    - r_size (forward_r_size/...): 0~6.  \( \rightarrow \ r\_size = f\_code -1. \)
    - f (forward_f/...): 1, 2, 4, 8, 16, 32, 64.  \( \rightarrow \ f = 2^{f\_size} = 2^{f\_code-1}. \)
  - **motion_code** (motion_h.../v...f.../b...code): -16~+16. [Table 8.9, Slide 36]
  - motion_r (motion_h.../v...f.../b...r), r (compliment_h.../v...f.../b...r): 0, ... , f-1.
    - motion_r = (f-1) - r  \( \rightarrow \ r = motion\_r - f + 1. \)
  - **Principal**  \( dMD_p = motion\_code \times f = motion\_code \times 2^{f\_code-1}. \)
  - **Residual**  \( r = \mid dMD_p \mid - \mid dMD \mid > 0. \)
    - motion_code \times f = dMD + Sign(dMD) \times (f-1).
      \( \checkmark \) dMD = motion_code \times f - Sign(motion_code) \times r.
  - Current MD = PMD + dMD = [(PMD+dMD+16xf)%32xf]-16xf.
    - MD & dMD \in [min = -16xf, max = 16xf-1]
  - All MD's, recon_right(_half)_for/back, recon_down (_half)_for/back, are thus computed.
    - For horizontal forward/backward motion vectors, the displacement to the right from the macroblock in the current picture to the predictive area in the reference picture is:
      \( right\_half\_for \) (Half_pel Flag) = YHF = MD\&1  \( \text{ or } \)  CHF = (MD/2) & 1
      right_for = YMD = MD>>1  \( \text{ or } \)  CMD = (MD/2) >>1
invQ_intra()
{
    for(m=0; m<8, m++)
        for(n=0; n<8, n++)
            i=scan[m][n]; // zigzag order index
            dct_recon[m][n]=(2*dct_zz[i]*quantizer_scale*intra_quant[m][n])/16;
        if((dct_recon[m][n]&1)==0) dct_recon[m][n]=Sign(dct_recon[m][n]); // oddify
        if(dct_recon[m][n]>2047) dct_recon[m][n]=2047; // clamp to max
        if(dct_recon[m][n]<-2048) dct_recon[m][n]=-2048; // clamp to min
    }
}

dct_recon[0][0]=dct_zz[0]*8; // overrule calculation for DC coefficient

invQ_intra_Y0()
{
    invQ_intra();
    if((macroblock_address - past_intra_address)>1) // macroblocks skipped
        dct_recon[0][0] += (128*8);
    else dct_recon[0][0] += dct_dc_Y_past; // previous block is prediction
    dct_dc_Y_past = dct_recon[0][0]; // set prediction for the next Y blocks
}

invQ_intra_Ynext()
{
    invQ_intra();
    dct_recon[0][0] += dct_dc_Y_past; // previous block is prediction
    dct_dc_Y_past = dct_recon[0][0]; // set prediction for the next Y blocks
}

invQ_inter()
{
    for(m=0; m<8, m++)
        for(n=0; n<8, n++)
            i=scan[m][n]; // zigzag order index
            dct_recon[m][n]=(2*dct_zz[i]+Sign(dct_zz[i]))*quantizer_scale*intra_quant[m][n]/16;
        if((dct_recon[m][n]&1)==0) dct_recon[m][n]=Sign(dct_recon[m][n]); // oddify
        if(dct_recon[m][n]>2047) dct_recon[m][n]=2047; // clamp to max
        if(dct_recon[m][n]<-2048) dct_recon[m][n]=-2048; // clamp to min
        if(dct_zz[i]==0) dct_recon[m][n]=0; // force zeroed dequantized coefficient
    }
} // in case of skipped macroblocks
Audio Compression

- Amplitude and time are the only two dimensions used in audio coding methods.
  - However, the human auditory system is more sensitive to quality degradation than is the human visual system.
    - The amount of redundancy that can be removed for digital audio is relatively small.
    - De-correlation on audio information requires much computation (more complexity).

- Variants of Pulse Code Modulation (PCM)
  - Logarithmic transformation:
    - A-law (µ-law): mapping 13 (14) to 8 bits of linearly quantized PCM values.
    - Adaptive Differential PCM can achieve better lossy compression using a small number of bits (say, 4) by changing the step size of the quantizer, the predictor, or both.

- MPEG audio addresses different compliance points as layers 1, 2, 3 (most complex.)
  - If a decoder can decode layer n, it can also decode all lesser layers.
    - MPEG-1 Audio defines 1 or 2 audio channels: a single channel, 2 indep. channels, or 1 stereo signal.
      - MPEG-1 Layer 3 is often abbreviated as MP3.
    - MPEG-2 Audio extends it as up to 5 channels (left, right, center, two surrounds), plus a low-frequency enhancement channel, and/or up to 7 commentary/multilingual channels.
  - Sub-band decomposition divides audio signal into multiple frequency bands, each then scaled and quantized. Further frequency domain analysis is used to select the quantizer step size.
  - Decoding reverse such sequence to reconstruct the signal.
Perceptual Audio Coder

- Trend of audio signal compression:
  - **Simultaneous masking**: MPEG-1 Audio exploits masking in the frequency domain.
  - **Non-simultaneous masking**: Recently developed methods use masking in the time domain to better the compression rate.

- Rationale:
  - The strategy is to analyze whether a strong signal in a given time block can mask an adequately lower distortion in a previous or subsequent block (backward/forward masking.)
  - As an example, PAC (Perceptual Audio Coder) from AT&T Bell Lab. demonstrates the best decoded quality of any algorithm at 320 kbps for 5-channel audio.
    - PAC is not backward compatible since it cannot decode MPEG-1 audio streams.

- More information for compression:
  - Through Web search engines:
    - Yahoo (www.yahoo.com), Lycos (lycos.cs.cmu.edu), Alta Vista (altavista.digital.com), etc.
    - A keyword like “MPEG” can find the related research development sizes to date.
  - Interesting Web sites:
    - MPEG Home Page: http://drogo.cselt.stet.it/mpeg/
    - MPEG-4 Structured Audio: http://sound.media.mit.edu/mpeg4/
    - Berkeley Continuous Media Toolkit: http://bmrc.berkeley.edu/frame/research/cmt/
    - Emphasis project: http://www.fzi.de/esm/projects/emphasis/
    - ISO online: http://www.iso.ch/
    - Wavelets: http://www.mat.sbg.ac.at/~uhl/wav.html
    - MIDI: http://www.midi.org/
Microsoft DirectShow

- DirectShow is the part of DirectX Media for decoding and rendering multimedia streams, such as MPEG, AVI, QuickTime from local files or over the Internet.
  - **DirectShow** objects are the **COM** objects as like other **DirectX** components.
    - DirectShow is the part of DirectX Media that make use of **COM** technique for the interconnection of components.
  - **Filters** are logical **DirectShow** objects to perform a specific operation on data and produce an altered output.
    - Filters are connected through their input and output **pins**.
    - Pins are objects created (defined) by the hosting filters.
  - The connected filters form a collaborating entity called **Filter Graph**.
    - Stream playback can be done through a series of filters' operations, from retrieving media data to outputting on the hardware devices.

- Since filters of diverse functionality (say, decoding MPEG video/audio streams) are available in the MS Windows, only a limited coding is necessary to develop any specific multimedia application.
  - Component reusing is possible; components can be downloaded on demand.
Example of Filter Graph

Three major types of filters:

- **Source**: It has no input pins and has one or more output pins.
  - Typically, a source filter is responsible for reading the raw data from a source file, network, or any other media (audio/video hardware's).

- **Transform**: It has one or more pins each for input and output.
  - Typically, a transform filter converts the input data from the upstream filter(s) before sending it further to the downstream filter (like decoding).

- **Rendering**: It has one (or more) input pin and no output pin.
  - A rendering filter accepts data on its input pin and delivers it to the final destination (screen, audio card, file and so on.)
Minimal Coding...

- Filter Graph Editor (graphedt.exe) from DirectX SDK 8.0 provides visual configuration of filter graphs.
- PlayWnd.exe (only 28KBytes) from SDK can playback most media files.

```c
1. #include <dshow.h>
2. IGraphBuilder *pGB = NULL; IMediaControl *pMC = NULL; IMediaEventEx *pME = NULL;
3. IVideoWindow *pVW = NULL; IBasicAudio *pBA = NULL; IBasicVideo *pBV = NULL;
4. IMediaSeeking *pMS = NULL; // DirectShow interfaces
5. HRESULT PlayMovieInWindow(LPTSTR szFile){  WCHAR wFile[MAX_PATH];   HRESULT hr; ...
6.   // Get the interface for DirectShow's GraphBuilder
7.   CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
8.     IID_IGraphBuilder, (void **)&pGB));
9.   pGB->RenderFile(wFile, NULL); // Have it construct the proper graph automatically
10.  // QueryInterface for DirectShow interfaces
11.  pGB->QueryInterface(IID_IMediaControl, (void **)&pMC);
12.  pGB->QueryInterface(IID_IMediaEventEx, (void **)&pME);
13.  pGB->QueryInterface(IID_IMediaSeeking, (void **)&pMS);
14.  pGB->QueryInterface(IID_IVideoWindow, (void **)&pVW);
15.  pGB->QueryInterface(IID_IBasicVideo, (void **)&pBV);
16.  pGB->QueryInterface(IID_IBasicAudio, (void **)&pBA); pVW->put_Owner((OAHWND)ghApp);
17.  pVW->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS | WS_CLIPCHILDREN));
18.  pME->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY, 0); InitVideoWindow(1, 1); ...
19.  // Run the graph to play the media file
20.  pMC->Run();  g_psCurrent=Running;  SetFocus(ghApp);
21.  return hr;
22. } // ...
```
Enter media file name to play

---

```
function PlayStream(){ MyObj.FileName=MyField.value; MyObj.Play(); }
function PauseStream(){ if(MyObj.PlayState==1){ MyObj.Play(); } else if(MyObj.PlayState==2) { MyObj.Pause(); } }
function StopStream(){ MyObj.Stop(); MyObj.CurrentPosition=0; }
```
A simple coding example

Naïve codes:
- The source filter `CFruitFilter` reads one line at a time from the text file "Fruit.ftf" and passes it to the next filter.
- The transform filter `CInvertFilter` accepts a string on its input pin and delivers an inverted string further down to its output pin.
- The rendering filter `CTextOutFilter` displays the string presented at the input pin to a text window.

DirectShow is extensible:
- Many built-in classes are available for further derivation.
  - `CSource`, `CSourceStream`, etc. can be subclassed by all source filters.
    - Handling data movement from the upstream to the downstream filter is implemented in the base class by default operations.
  - Particular functionality is customized through overriding.
Details of \textbf{C\text{F}ruit\text{F}ilter}

class C\text{F}ruit\text{F}ilter: public C\text{S}ource, public IFileSource\text{F}ilter {
public: static CUnknown * WINAPI Create\text{I}nstance(LPUNKNOWN lpunk,
 HRESULT *phr); // for creating an object

private:
DECLARE_IUNKNOWN // Required for IFileSource\text{F}ilter support
STDMETHODIMP Get\text{C}ur\text{F}ile(LPOLESTR * ppsz\text{F}ilename, AM.MEDIA\_\text{TYPE} *pmt); // added
STDMETHODIMP Load(LPCOLESTR psz\text{F}ilename, const AM.MEDIA\_\text{TYPE} __RPC_FAR*pmt); // added
STDMETHODIMP NonDelegatingQuery\text{I}nterface(REFIID riid, void ** ppv); // customized
// It is only allowed to create these objects with Create\text{I}nstance
C\text{F}ruit\text{F}ilter(LPUNKNOWN lpunk, HRESULT *phr); // private constructor
OLECHAR m\_sz\text{F}ilename[_MAX\_\text{PATH}]; // added
}

- "\text{S}tatic" Create\text{I}nstance() is the only way to construct an instance, and exists even
  before the filter is created by FGM.
- When FGM loads a filter into a filter graph, the variables \text{g}\_\text{T}emplates[] and
  \text{g}\_\text{c}\text{T}emplates in the filter file "*.ax" is examined to figure out which objects exist
  and how to create them.

  // COM global table of objects in this dll
  C\text{F}actory\text{T}emplate g_Templates[] = {
    \{ L"ABC - Fruit Source Filter", &\text{CLS}ID\_Fruit\text{F}ilter,
    C\text{F}ruit\text{F}ilter::Create\text{I}nstance, NULL, &\text{s}ud\text{F}ruit\text{F}ilter \}
  };
  int g\_\text{c}\text{T}emplates = sizeof(g_Templates) / sizeof(g_Templates[0]);

- For instance, FGM uses the 3\text{rd} item to get a pointer to the Create\text{I}nstance() function,
  which in turns is called for object creation.
More about CFruitFilter

- Then, the constructor `CFruitFilter()` is called, followed by the output pin creation.

```cpp
CUnknown * WINAPI CFruitFilter::CreateInstance(LPUNKNOWN lpunk, HRESULT *phr){
    CUnknown *punk=new CFruitFilter(lpunk, phr);
    if (punk == NULL) *phr = E_OUTOFMEMORY;
    return punk;
}
```

- FGM then checks the exposed interfaces of `CFruitFilter` by `IUnknown::NonDelegatingQueryInterfact()`.

```cpp
STDMETHODIMP CFruitFilter::NonDelegatingQueryInterface(REFIID riid, void ** ppv){
    CheckPointer(ppv,E_POINTER);
    if (riid == IID_IListSourceFilter)
        return GetInterface((IListSourceFilter *) this, ppv);
    return CSource::NonDelegatingQueryInterface(riid, ppv);
}
```

- Due to having `IFileSourceFilter` interface, FGM further initializes `CFruitFilter` by calling its `Load()` function. It prompts the user for a filename to fill up the member `m_szFileName[]` in UNICODE.
  - Subsequent call to `GetCurFile()` will open the file for reading.
As an output pin, *CFruitStreamText* bases on *CSourceStream* to handle the connection process with the downstream filter, buffer allocation, and the data movement.

Besides, it also processes the **Start**, **Pause**, **Stop** and other commands coming from the host application through FGM and the parent filter.
**Connection Process**

* FGM configures the filter graph as follows:
  1. A source filter is first added into a filter graph.
  2. For each output pin not connected yet, FGM tries to connect it to the input pin of another filter downstream using one of the supported media types.
    * There is a series of underlying negotiations.
  3. Once connected, the new added filter joins the graph. If it has output pins, FG further links this filter further as Step 2.
    * The rendering filters have no output pins, thus completing the filter graph.

* Negotiations:
  - **Media type**: The output pin queries the input pin for a list of media types it supports by repeatedly calling the input pin’s `GetMediaType()`.
    * For each type in a list, the output pin calls its own `CheckMediaType()` to see if it is supported.
    * Upon agreement, the `SetMediaType()` is called to confirm the selection.
    * Then, the negotiation moves on next for the shared memory buffer.
      - An MPEG file `MEDIATYPE_MPEGVideo` type.
Negotiation for Connection (cont’d)

- **Shared buffer:**
  - To allocate the shared buffer for pieces of data movement between the pins, the output pin calls its `DecideBufferSize()` to determine the buffer size.
    - It depends on the media type and the header information of the data (such as, the picture width and height).
  
  - Then, the buffer allocator function `SetProperties()` is called to check if the memory is available.
    - The actual buffer is allocated later when the filter graph is running (or in pause state.)
Start and Stop of Filter Graph

- When the filter graph is complete, the host application can start the graph.
  - For instance, the application can first obtain from FGM an `IMediaControl` interface, by which a "Run()" command can be issued.
  - As starting, a new thread for each output pin of the source filter is created to pump the corresponding data downstream.
    - In this example, the source file is opened (by `OnThreadCreate()` function) and read.

- Stopping:
  - When a "Stop()" command of `IMediaControl` is called, `OnThreadDestroy()` will close the input file and end the thread.
Moving the data

HRESULT CFruitStreamText::FillBuffer(IMediaSample *pms){
    BYTE *pData;  long lDataLen;
    pms->GetPointer(&pData);
lDataLen = pms->GetSize();
    // Read one line at a time till end of file..
    if (m_inFile.getline(pData, lDataLen)) {
        pms->SetActualDataLength(strlen((char*)pData)+1);
        // The current time is the sample's start
        CRefTime rtStart = m_rtSampleTime;
        // Increment to find the finish time
        m_rtSampleTime += (LONG)1000;
        pms->SetTime((REFERENCE_TIME *) &rtStart, (REFERENCE_TIME *) &m_rtSampleTime);
    } else return S_FALSE;
    return S_OK;
} // FillBuffer

- As long as the filter graph is running,
  - each thread repeatedly calls the FillBuffer() function to fill the shared buffer
    with the data retrieved from the input file.
      - SetActualDataLength() is called to set the size of valid bytes in the shared
        buffer.
      - The data buffer in terms of media sample objects is automatically delivered
        to the downstream filter for each successful call to FillBuffer().
Creating a Transform Filter

- **CInvertFilter**, a transform filter,
  - accepts the data from its input pin,
  - applies some transformation on the data,
    - In this example, a text string is inverted.
    - E.g., MPEG video decoder converts the input data of major type “Video”, sub type “MPEG1Payload” with format “RGB 320x240, 0 bits”, to the output data of major type “Video”, sub type “RGB555” with format “RGB 320x240, 16 bits”.
  - then sends it out to the next filter.

- The base CTransformFilter is commonly used to create a transform filter.
  - Additional pins can be added to carry different output streams.
    - An MPEG-1 stream splitter can configure two output pins when the input stream is an MPEG-1 system stream.
Data transformation

- Like CFilter, CInverFilter has the same functions on its output pin.
  - GetMediaType(), CheckMediaType(), SetMediaType(), etc.
  - These functions can be overridden to specify the distinct operations.

- There are some new functions:
  - CheckInputType():
  - CheckTransform():
  - Receive():
    - Accepts an IMediaSample as an input,
    - Calls IMediaSample::GetPointer() to retrieve a pointer to the input buffer.
    - Calls the output pin’s GetDeliveryBuffer() to get a pointer to the shared output buffer.
    - Inverts the input string and inserts it in the output buffer.
    - Calls the output pin’s Deliver() function to deliver the data downstream.
The Rendering Filter

- The base **CBaseRenderer** is used to develop rendering filters, like **CTextOutFilter**.
  - It accepts data from an upstream filter and renders it to
    - a dump file, screen, audio device, the Internet, and so on.
  - It is the last stop for the data in the filter graph.
  - In this example, a text string is displayed to a text window on the screen.

- New functions:
  - **CompleteConnect()**: called to affirm the connection on pins at last.
  - **BreakConnect()**: called when the input pin is disconnected.
    - Upon “Stop”, connections of pins are broken. This function is used to handle this situation. (say, hide/destroy the output window)
  - **OnReceiveFirstSample()**: called to render the 1st sample of data right after the “Pause” or “Run” commands are issued to the filter graph.
    - In motion video, it is necessary to show the last video frame when paused.
  - **DoRenderSample()**: repeatedly called upon data arrival to perform the rendering action.