Compression Techniques

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

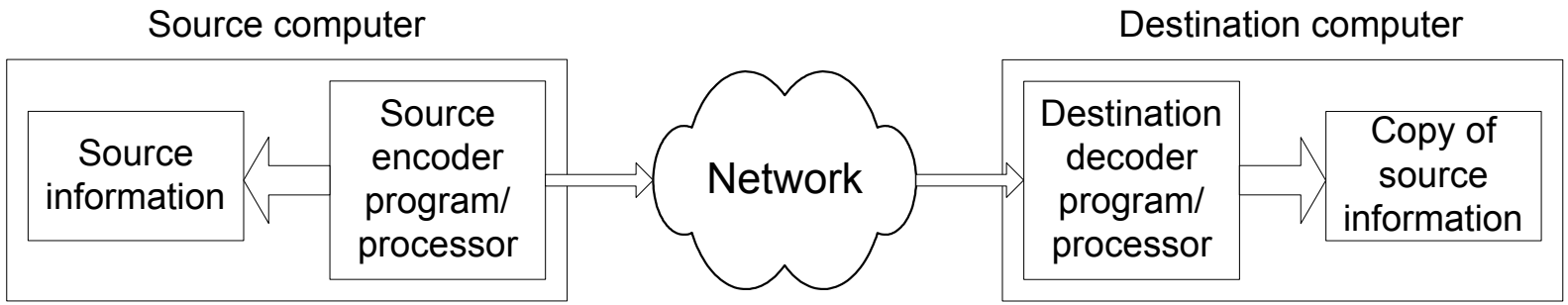
✓ Multimedia Objects Are Huge:

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

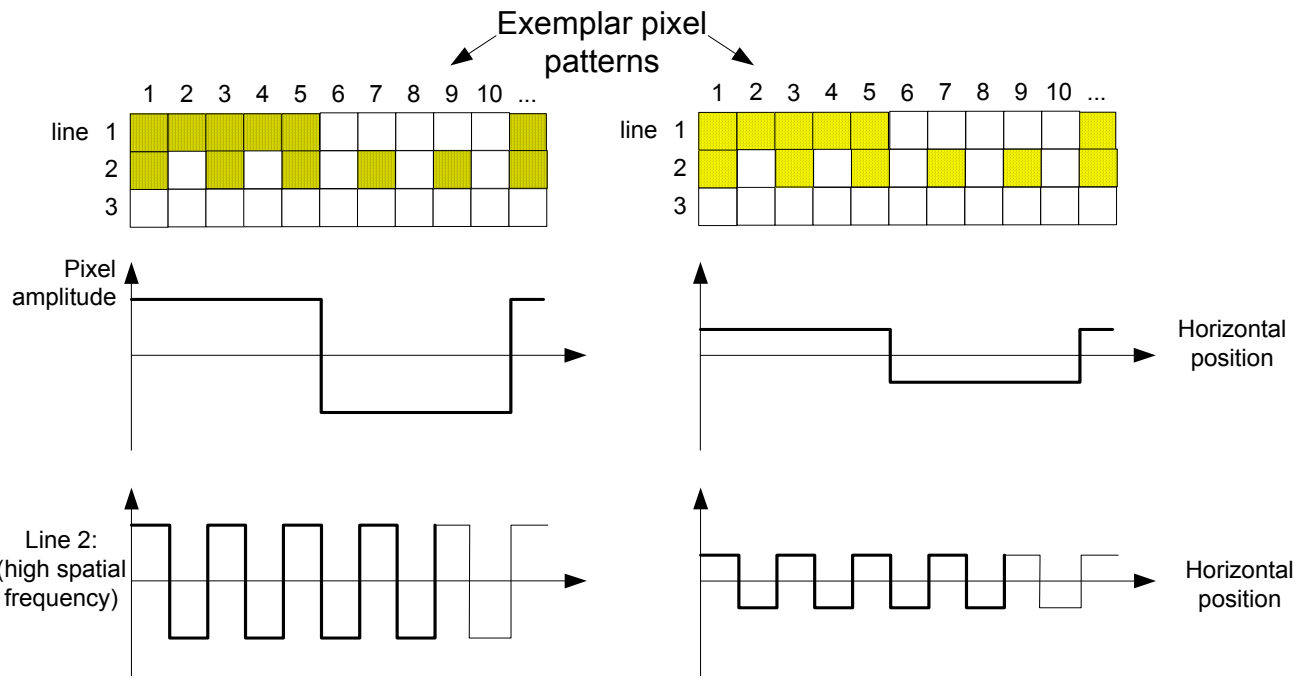
$\Rightarrow$ Transmission rate is: $640 \times 480 \times 2 \times 30 \approx 18$ MBytes/sec.

$\Rightarrow$ Storage requirement is: 18 MBytes/sec $\times 5$ min ≈ 5.4 GBytes.

Illustration



Information representation:



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) -

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

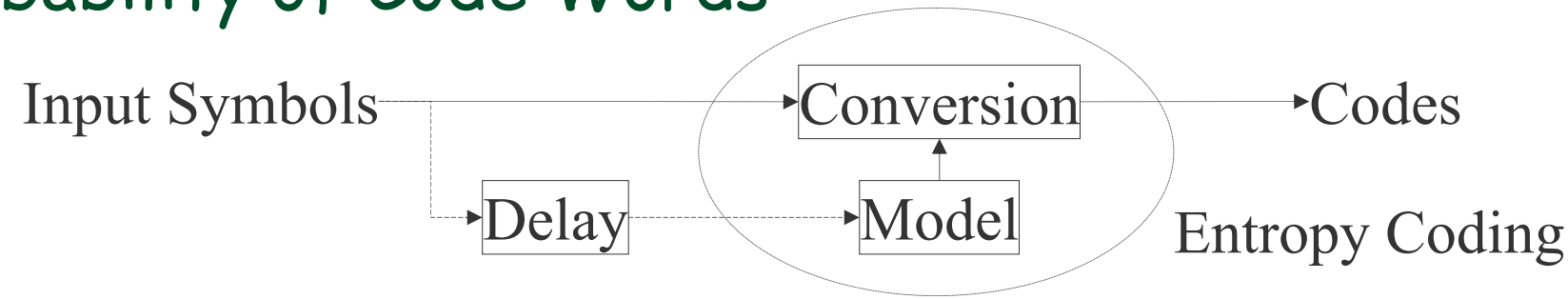
$$RMSE = \sqrt{\frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}[f(i,j) - \hat{f}(i,j)]^2}. \text{ 2D}$$

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

$$PSNR(dB) = 10\log_{10}\left(\frac{Q^2}{\sigma_r^2}\right).$$

$\boxed{Q = Maximum}$ =255, using 8 bits.

Probability of Code Words



- Delay of inputs are applied to approach the precision of statistical prediction model.
  - The Model keeps adjusting the mapping based on the conditional probability. $P(s_i \mid s_{i-r}, \cdots, s_{i-2}, s_{i-1})$
  - 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 ⇐ 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:

$$s_1 = 0, s_2 = 01, s_3 = 11, s_4 = 00 \quad \Rightarrow \quad \text{"0011"} = s_4 s_3, \quad s_1 s_1 s_3 \ ?$$

- 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".

$$s_1 = 0, s_2 = 01, s_3 = 011, s_4 = 111 \quad \Rightarrow \quad \text{"011}\cdots\text{1111"} = s_3 s_4, \quad s_4 s_4 \ ?$$

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.

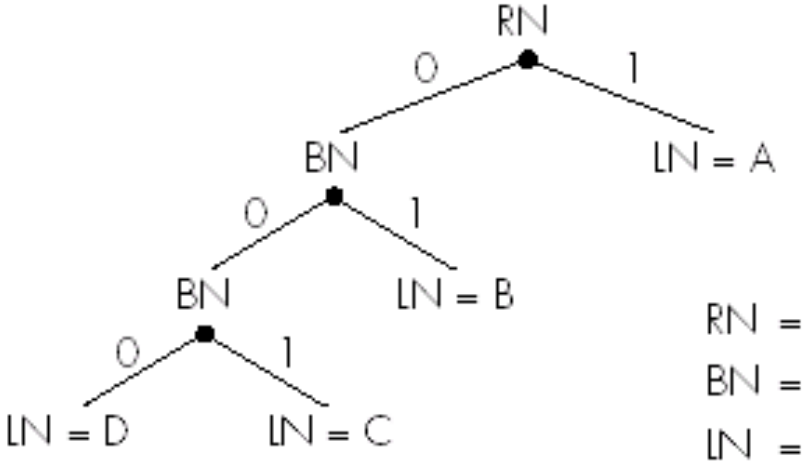
- NOTE:
  - "!2N" for NN would increase the code length by one byte.
  - "Byte stuffing": A real "!" (special flag) in the data is coded as "!!".
  - Longer sequences of different characters can be also coded in a similar way.
  - Indeed, different characters need not have to be encoded with a fixed 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) = 3/4, p(B) = 1/8, p(C) = p(D) = 1/16, or
    - p(ABCD) = 1, p(BCD) = 1/4, p(CD) = 1/8, p(D) = 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 x 3/4 + 2 x 1/8 + 3 x 1/16 x 2 = 1.375 bits instead of 2 bits.

Huffman code tree construction



A = 1
B = 0 1
C = 0 0 1
D = 0 0 0

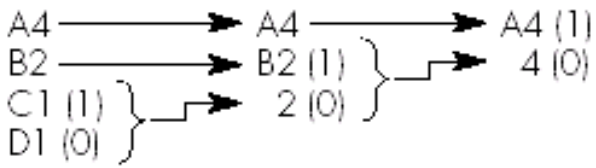
RN = root node
BN = branch node
LN = leaf node

Shannon's formula: Entropy H

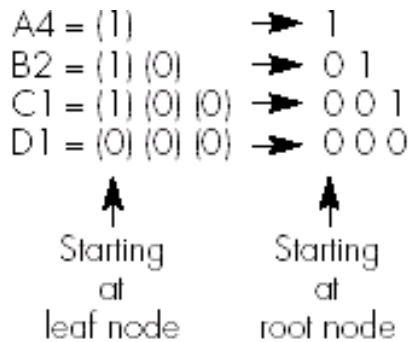
$$H = -\sum_{i=1}^{n} P_i \cdot \log_2 P_i$$

bits per codeword

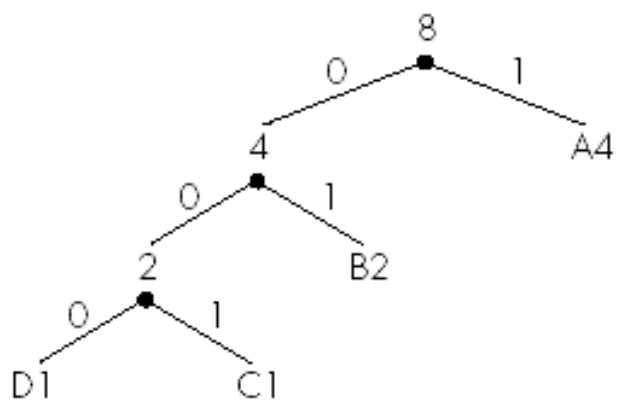
Begin



Frequency of occurrence

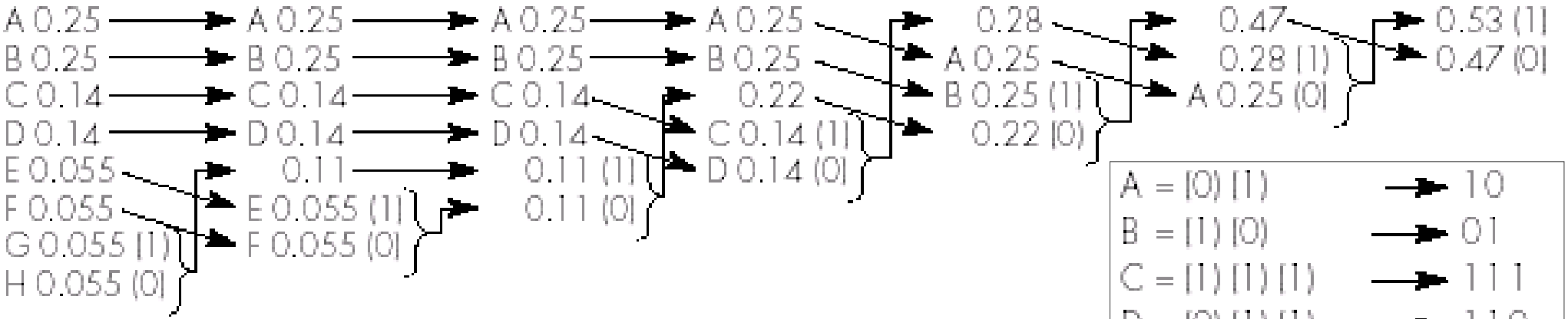


1. All nodes are free nodes;
2. Group two least-weight nodes (0 & 1).
3. Make a parent with the weights sum. (Branch node).
4. Repeat 2 & 3 until one node is left.

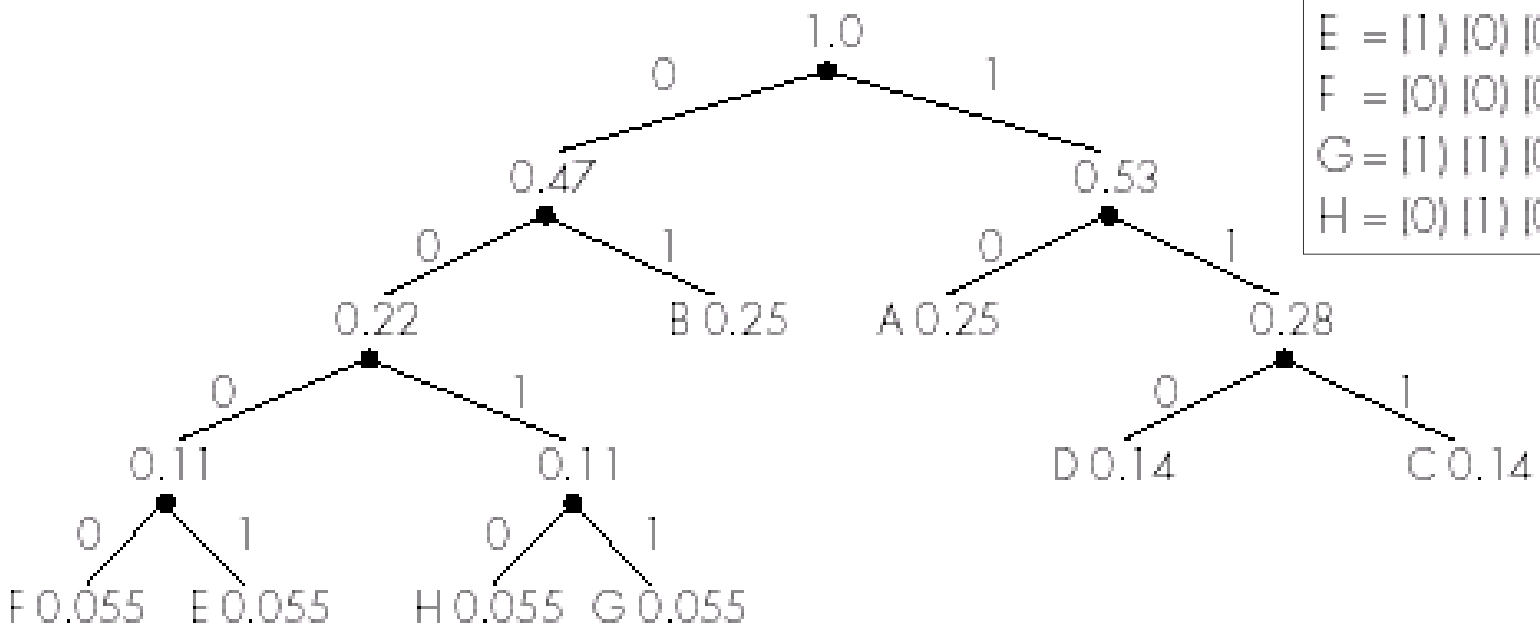


Weight order = D1 C1 2 B2 4 A4 8 ✔

Huffman encoding example



A = (0) (1)	→ 10
B = (1) (0)	→ 01
C = (1) (1) (1)	→ 111
D = (0) (1) (1)	→ 110
E = (1) (0) (0) (0)	→ 0001
F = (0) (0) (0) (0)	→ 0000
G = (1) (1) (0) (0)	→ 0011
H = (0) (1) (0) (0)	→ 0010



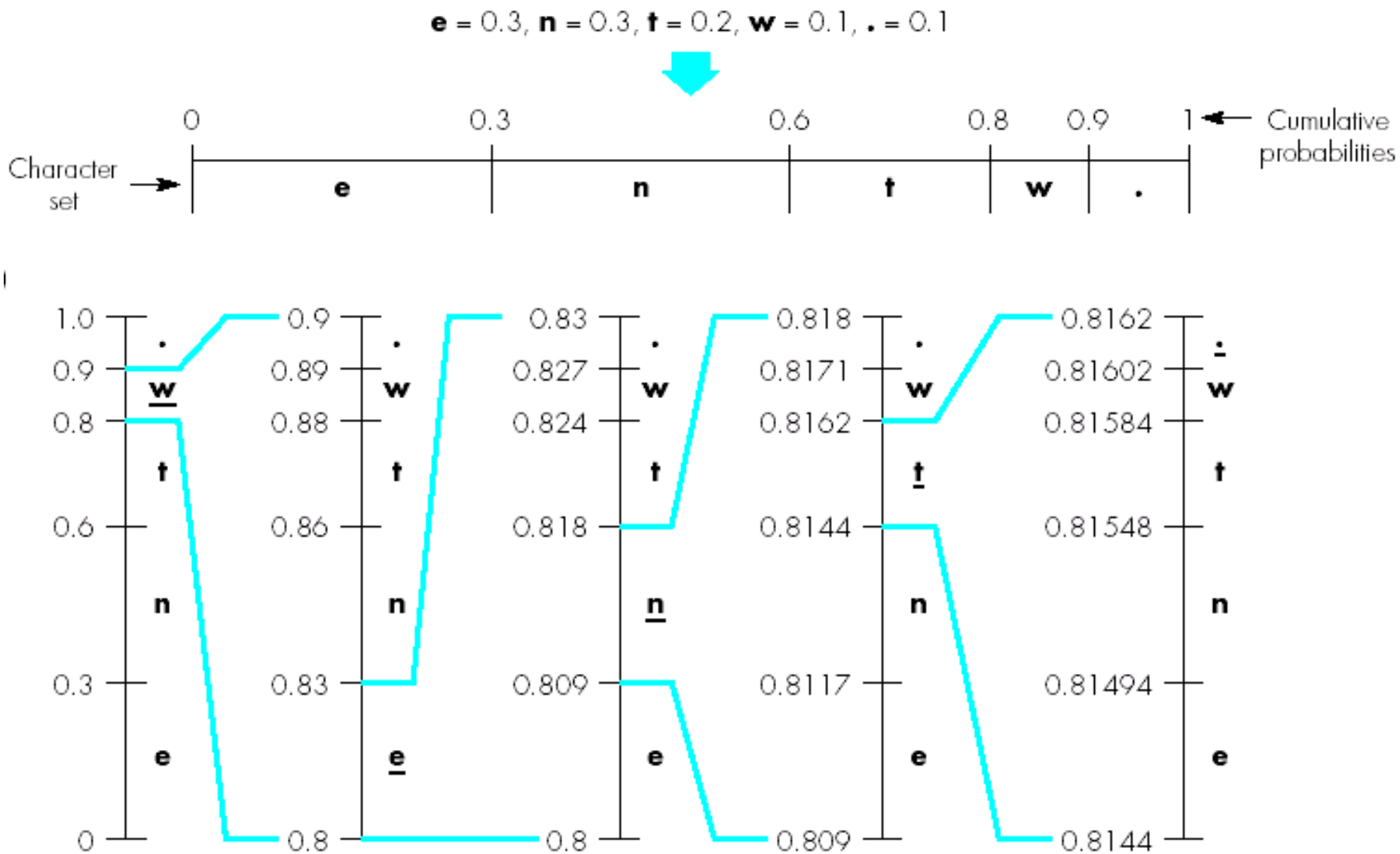
Weight order = 0.055 0.055 0.055 0.055 0.11 0.11 0.14 0.14 0.22 0.25 0.25 0.28 0.47 0.53 ✔

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,
    - ! [0.23354, 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.

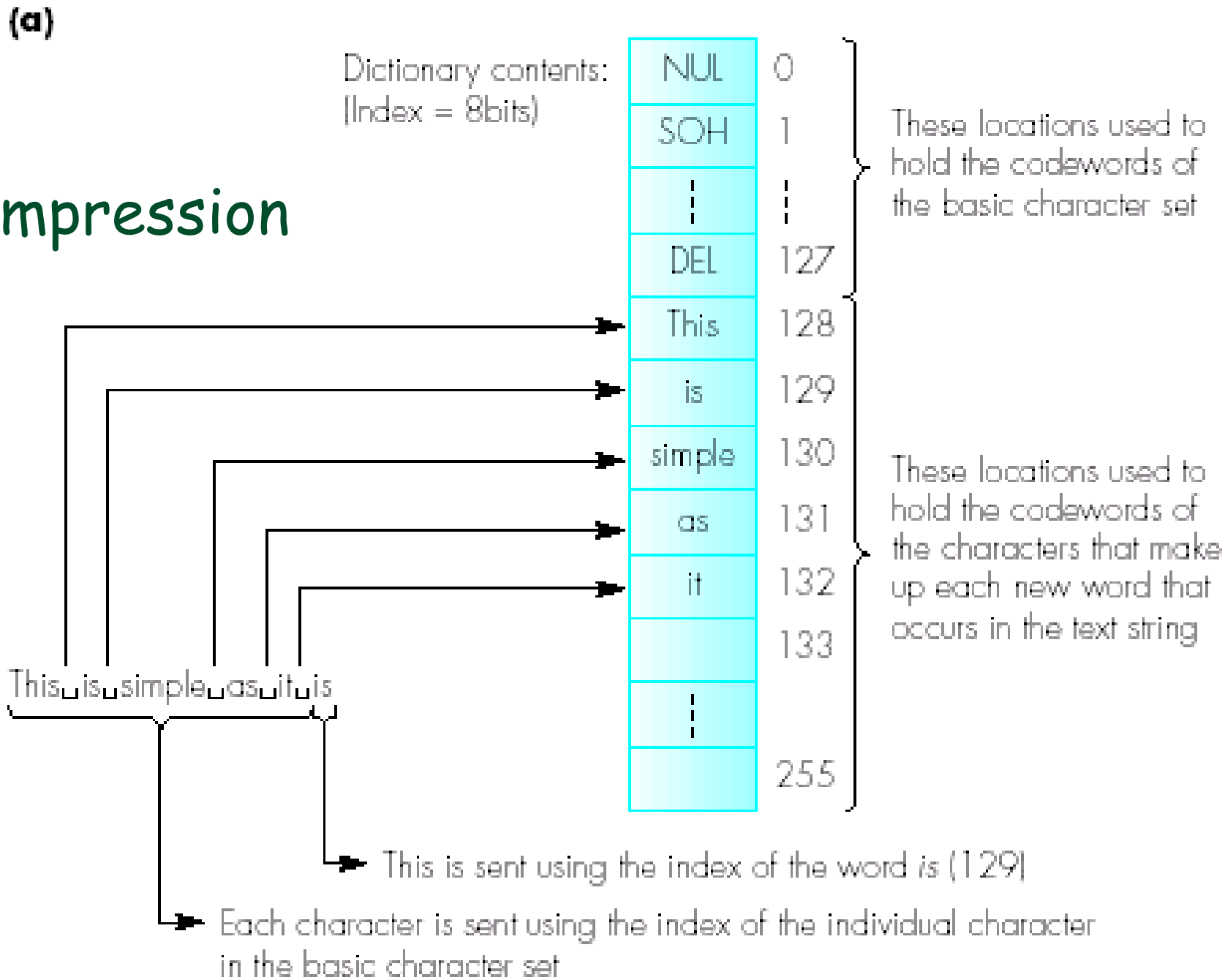
Arithmetic coding example

Example character set and their probabilities:



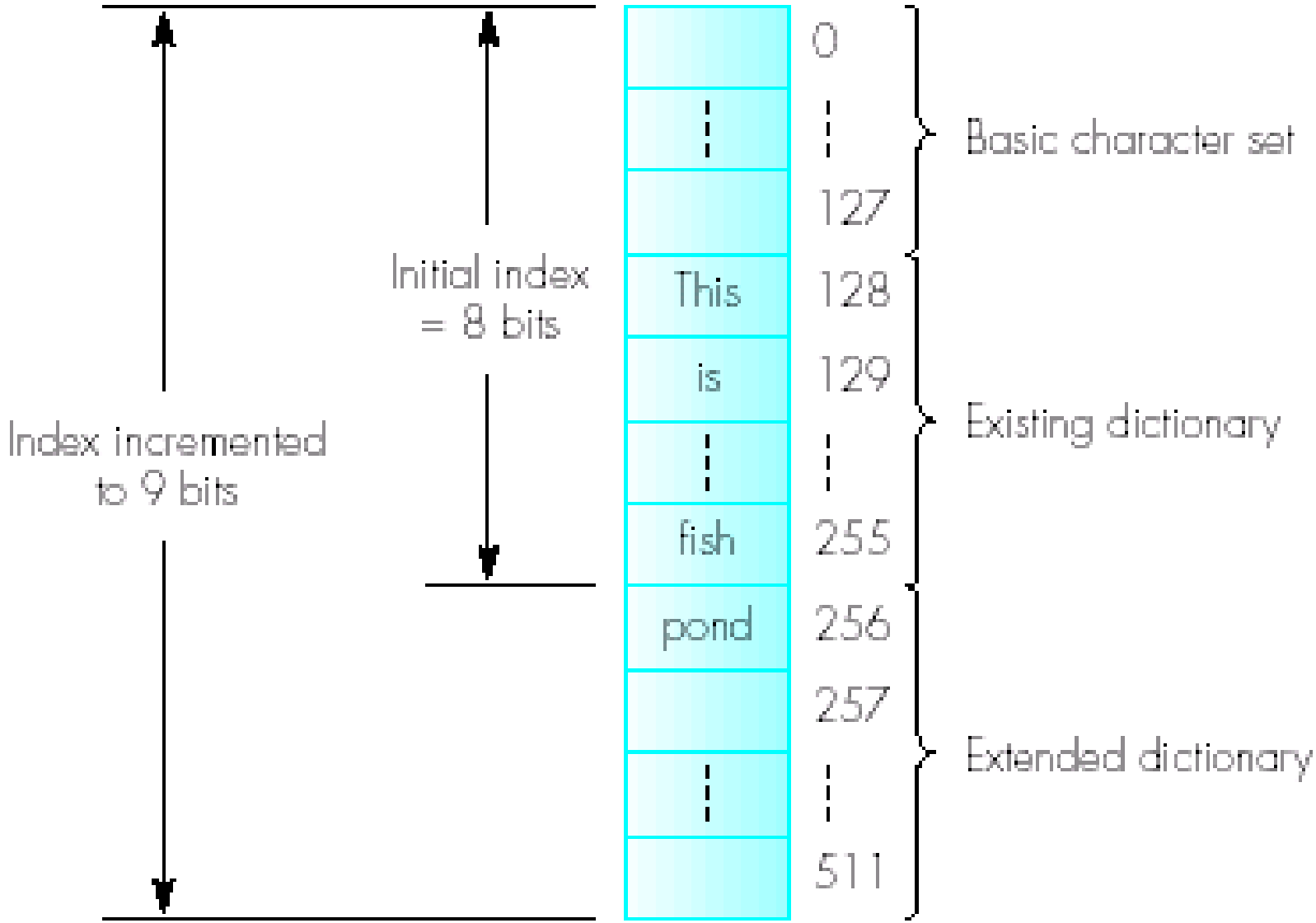
Encoded version of the character string **went.** is a single codeword in the range 0.816 02 ≤ codeword < 0.8162

LZW compression



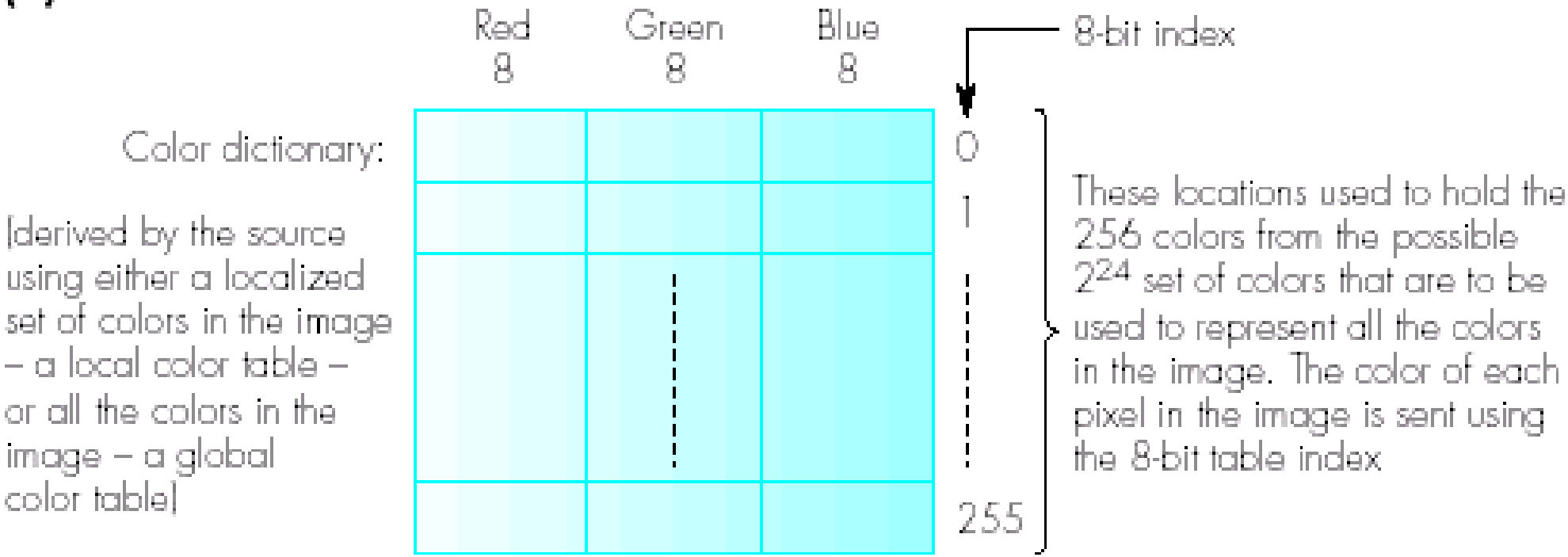
LZW compression

(b)



GIF compression

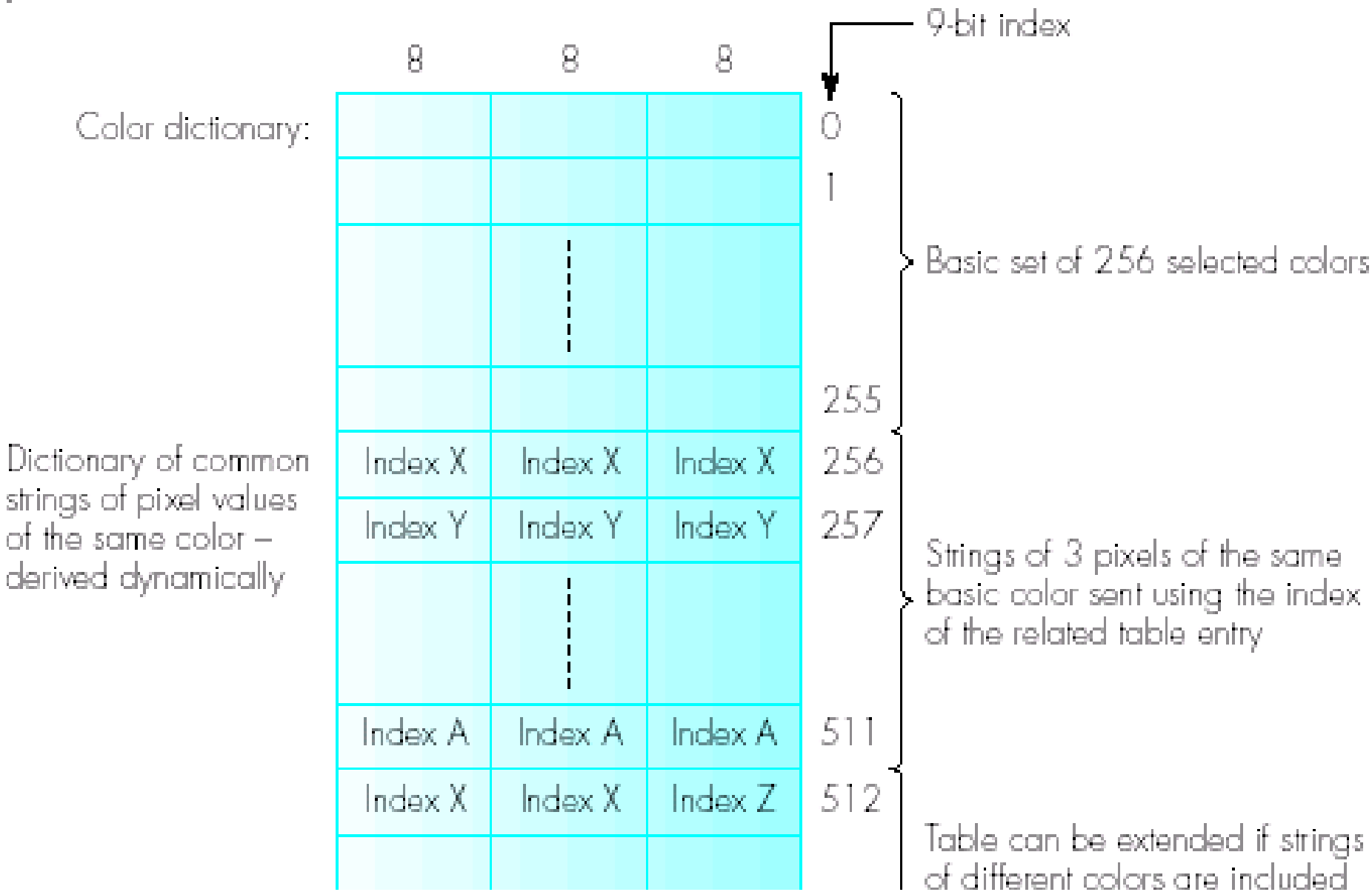
(a)



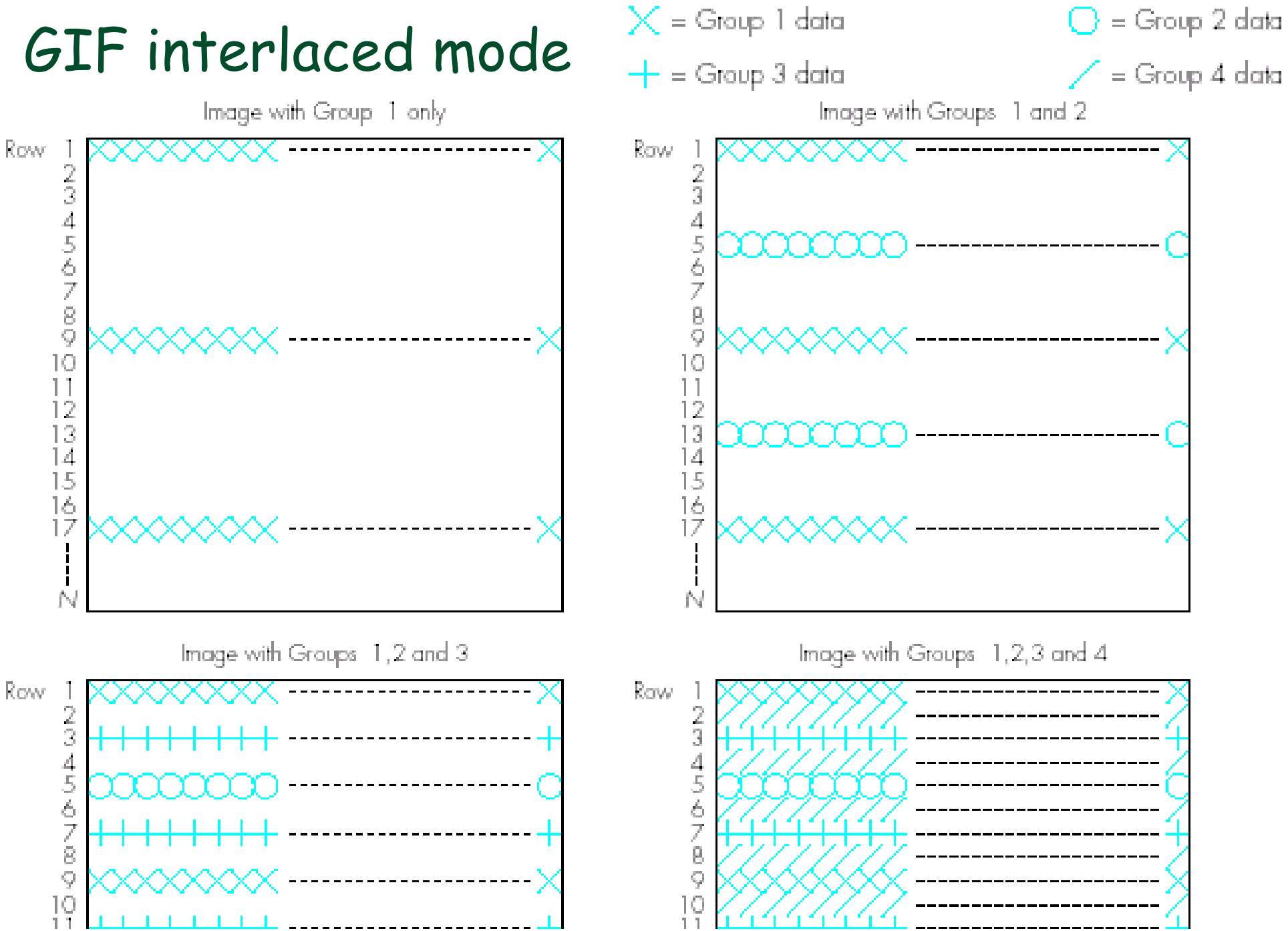
The color dictionary, screen size, and aspect ratio are sent with the set of indexes for the image.

GIF compression

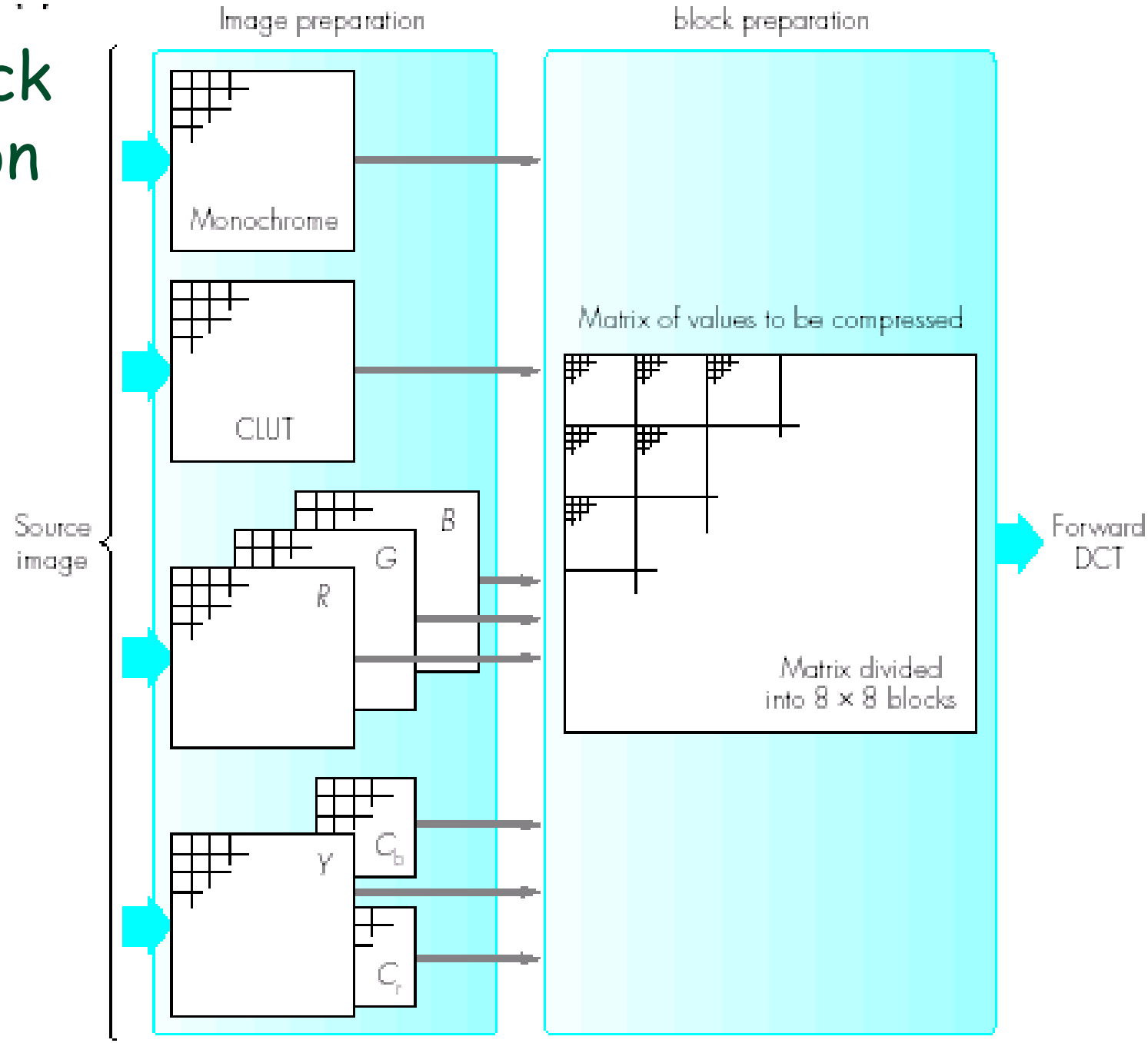
(b)



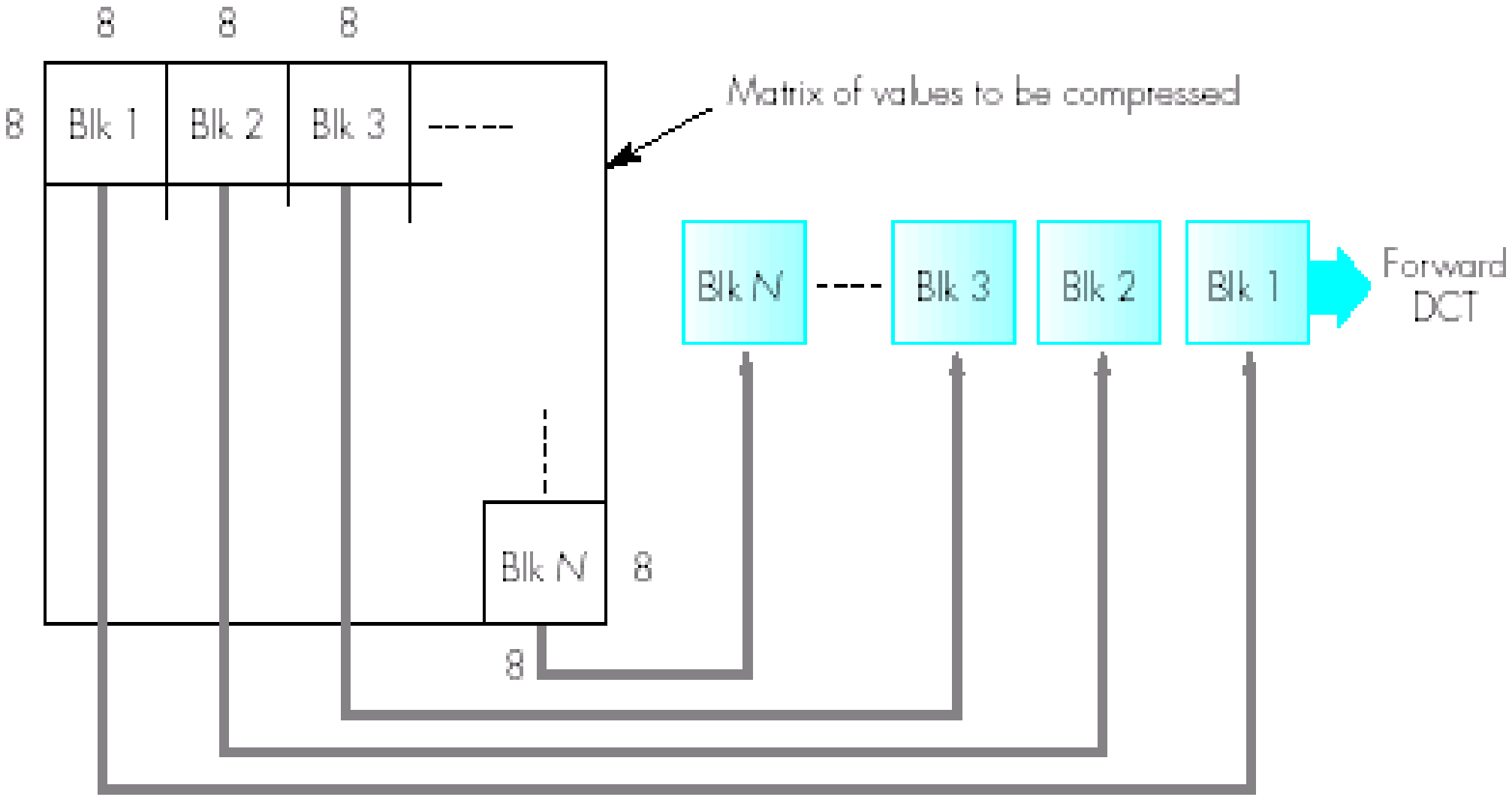
GIF interlaced mode



Image/block compression



Image/block compression



Transform Coding (one type of Source Coding)

SPATIAL DOMAIN

Height	Weight
65	170
56	130
80	203
40	80
69	148

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

Height	Weight
65	3
56	-4
80	1
40	-7
69	-9

Very small numbers

FREQUENCY DOMAIN

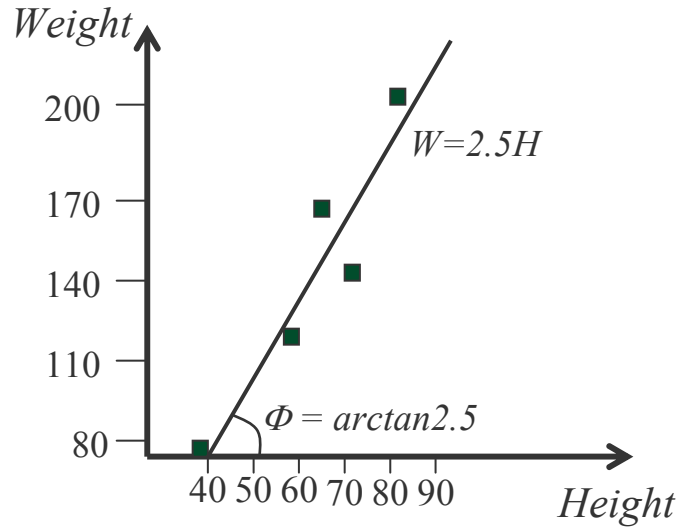
Height	Weight
65	0
56	0
80	0
40	0
69	0

Do not store these numbers

$$X' = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \cdot \theta'$$

RECONSTRUCTED DATA

Height	Weight
68	169
53	131
81	203
34	84
61	151



Discrete Cosine Transform (DCT)

Apply DCT to each 8x8 block:

132	136	138	140	144	145	147	155
136	140	140	147	140	148	155	156
140	143	144	148	150	152	154	155
144	144	146	145	149	150	153	160
150	152	155	156	150	145	144	140
144	145	146	148	143	158	150	140
150	156	157	156	140	146	156	145
148	145	146	148	156	160	140	145

Input matrix *pixel*

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

$$DCT(i,j)=\frac{1}{4}C_iC_j\sum_{x=0}^{7}\sum_{y=0}^{7}Spixel(x,y)\cdot\cos\frac{(2x+1)i\cdot\pi}{16}\cdot\cos\frac{(2y+1)j\cdot\pi}{16}$$

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

DC coefficient

Output matrix *DCT*

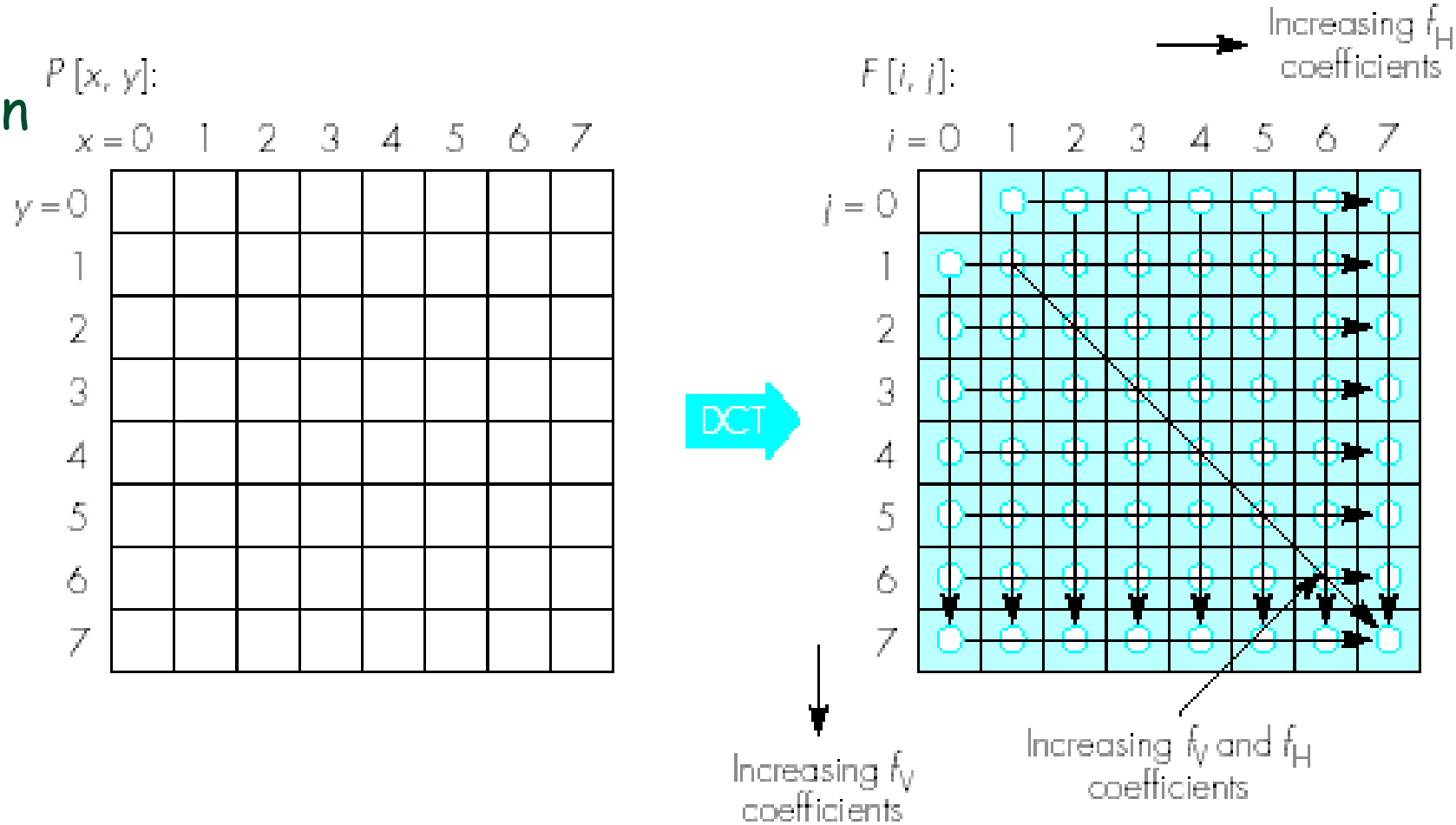
AC coefficients

172	-18	15	-8	23	-9	-14	19
21	-34	24	-8	-10	11	14	7
-9	-8	-4	6	-5	4	3	-1
-10	6	-5	4	-4	4	2	1
-8	-2	-3	5	-3	3	4	6
4	-2	-4	6	-4	4	2	-1
4	-3	-4	5	6	3	1	1
0	-8	-4	3	2	1	4	0

- 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 × 8 matrix of pixel values

$F[i, j]$ = 8 × 8 matrix of transformed values/spatial frequency coefficients

In $F[i, j]$: ☐ = DC coefficient ▭ = AC coefficients

f_H = horizontal spatial frequency coefficient

f_V = vertical spatial frequency coefficient

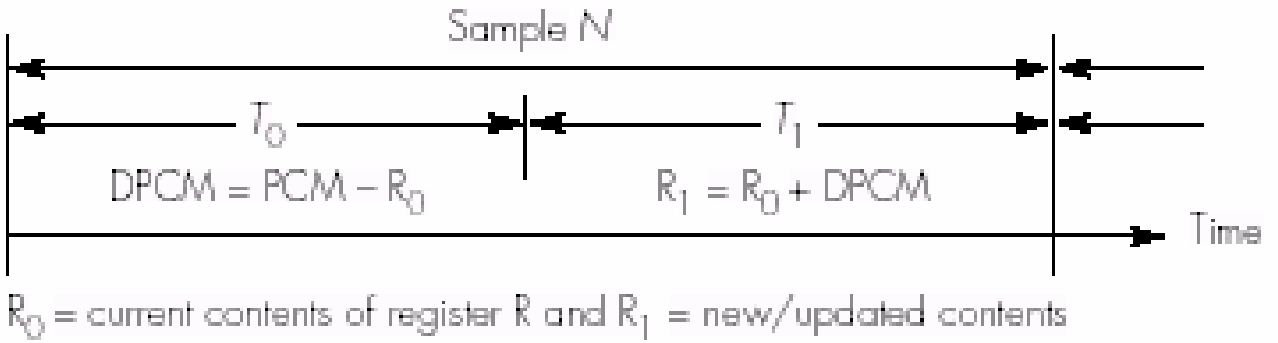
Differential Pulse Code Modulation (DPCM)

- 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.

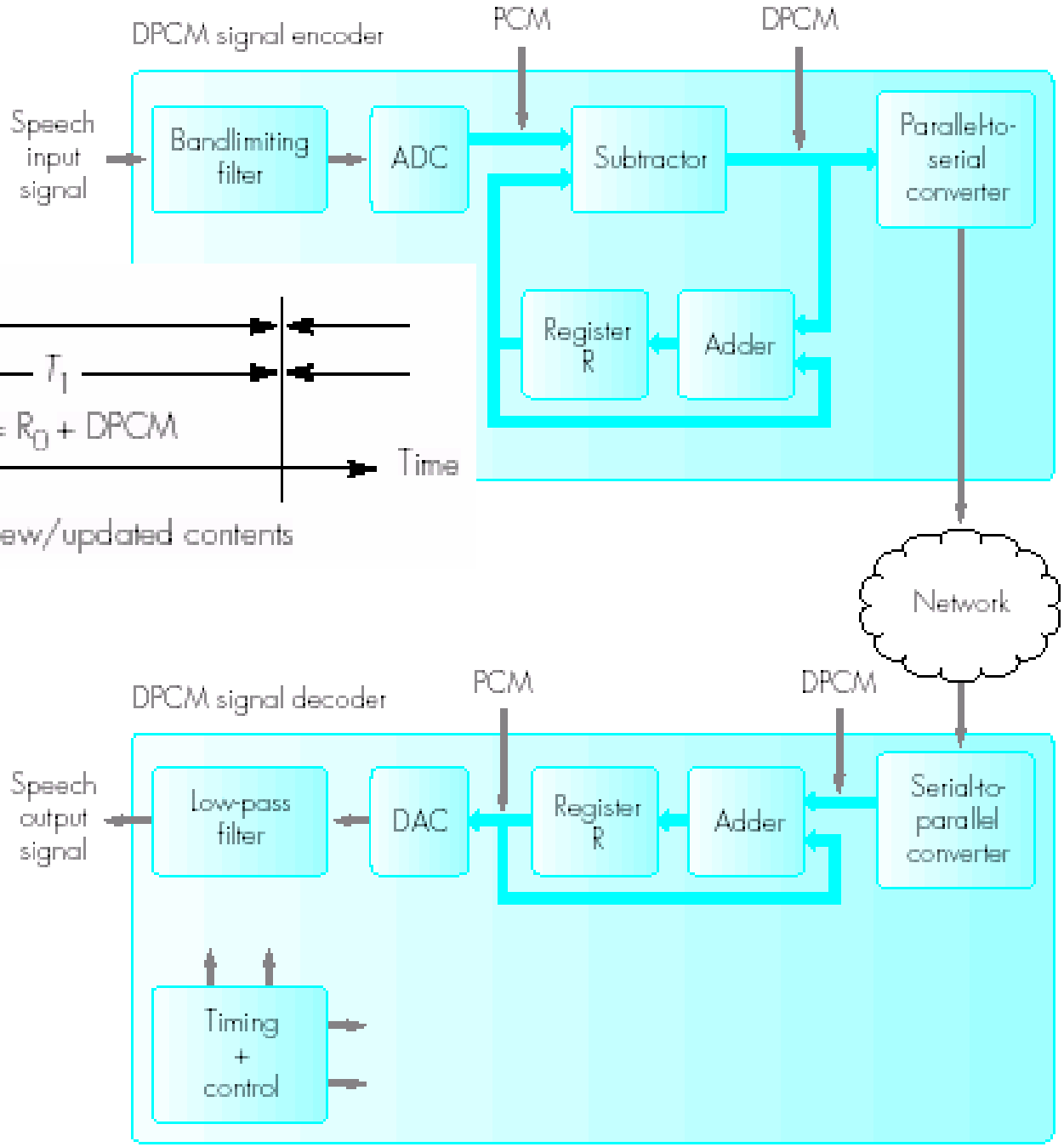
- DPCM, one of the simplest source coding methods, reduces the value range of numerical input characters s.t. successive entropy coding methods might achieve better results. (often applied to audio signals)

- 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.

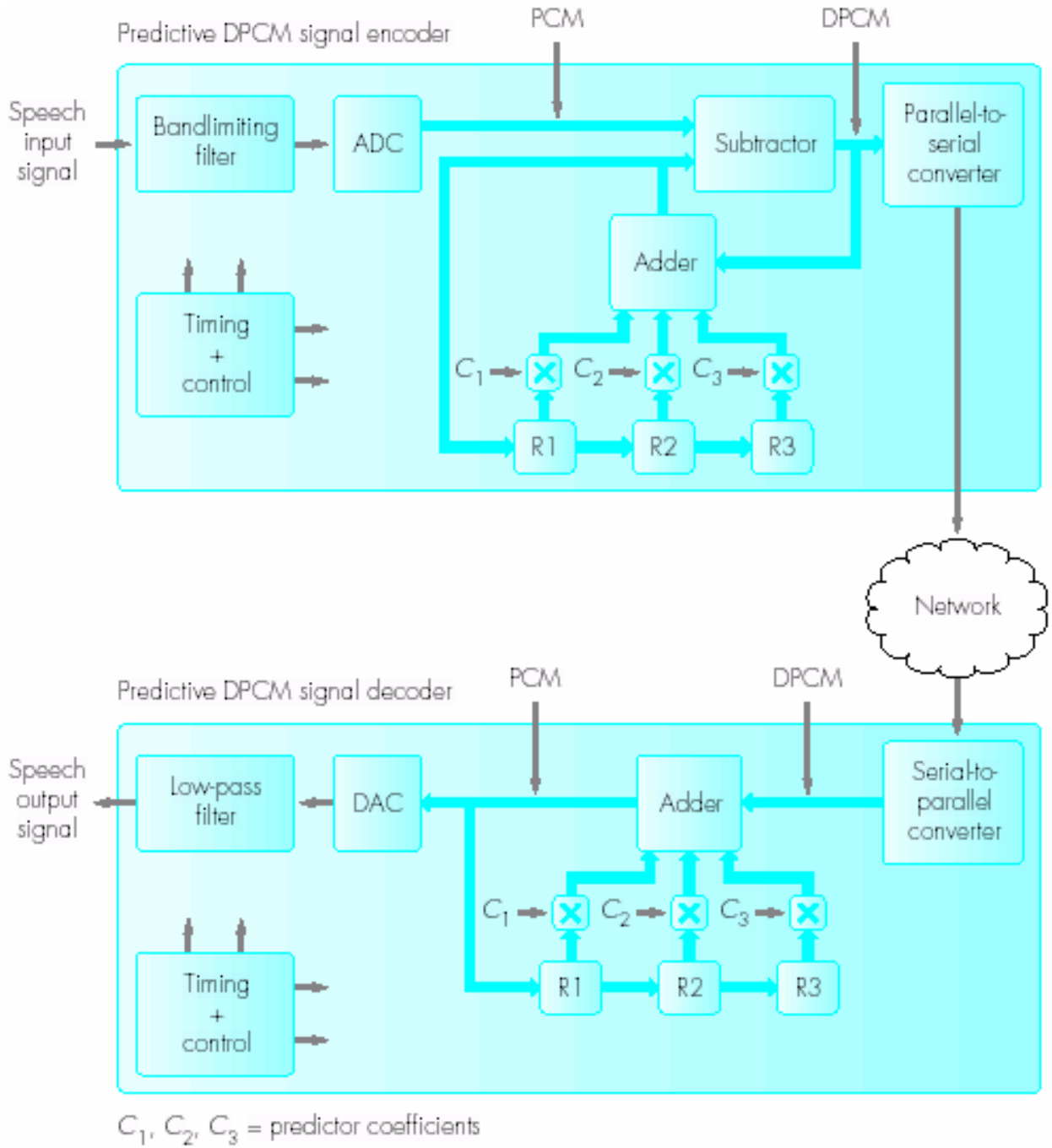
DPCM principles



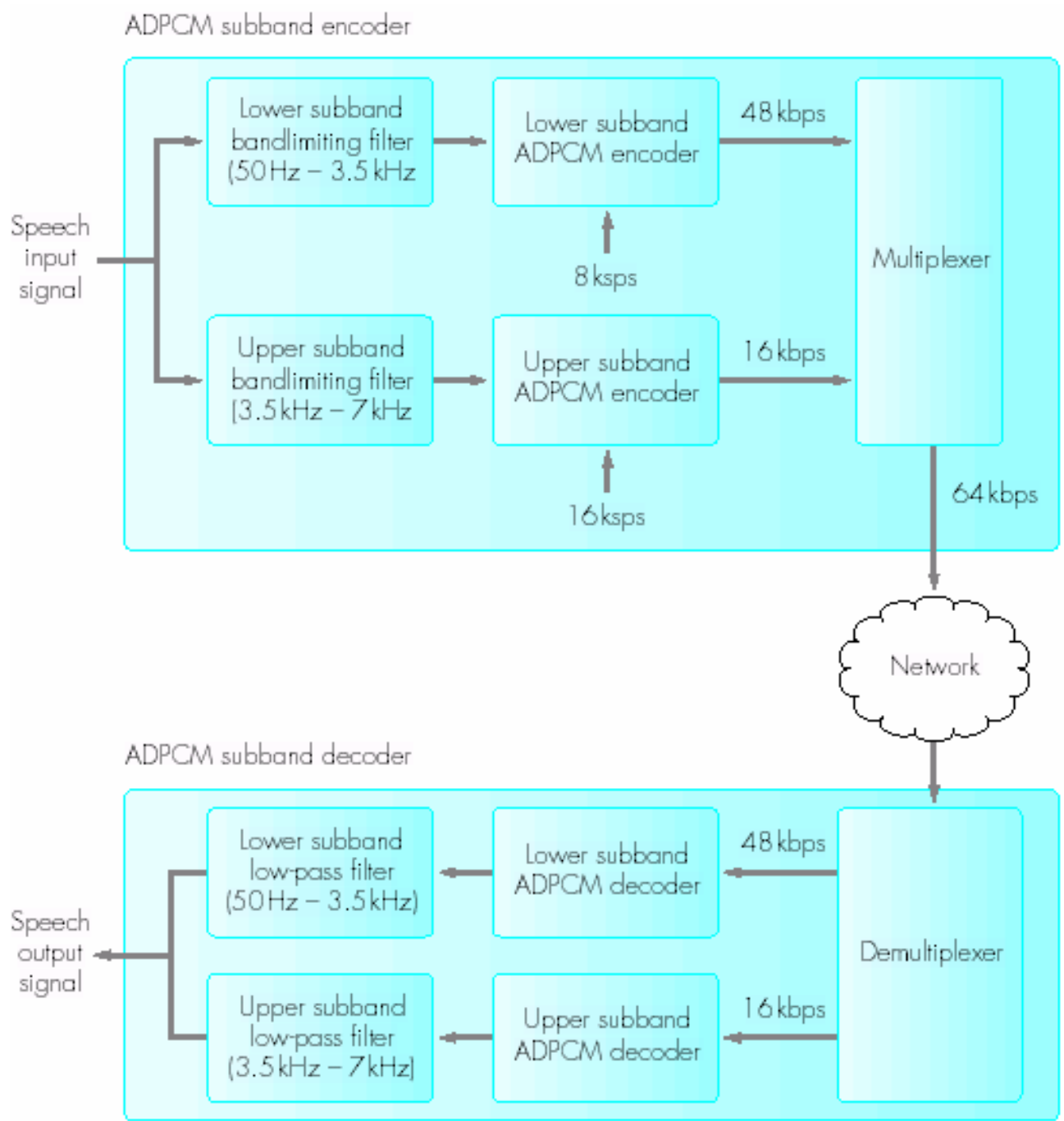
R_0 = current contents of register R and R_1 = new/updated contents



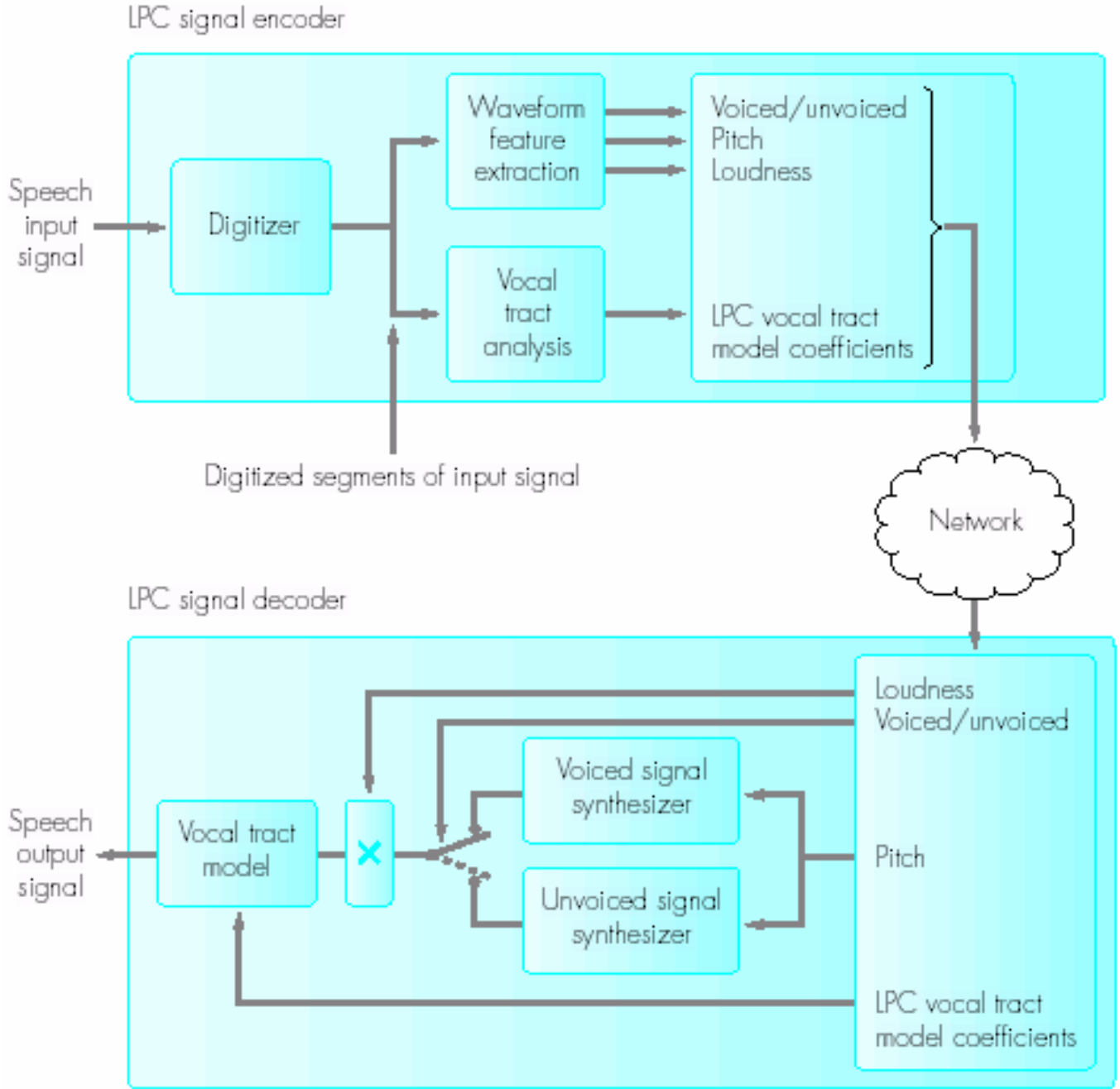
Predictive DPCM



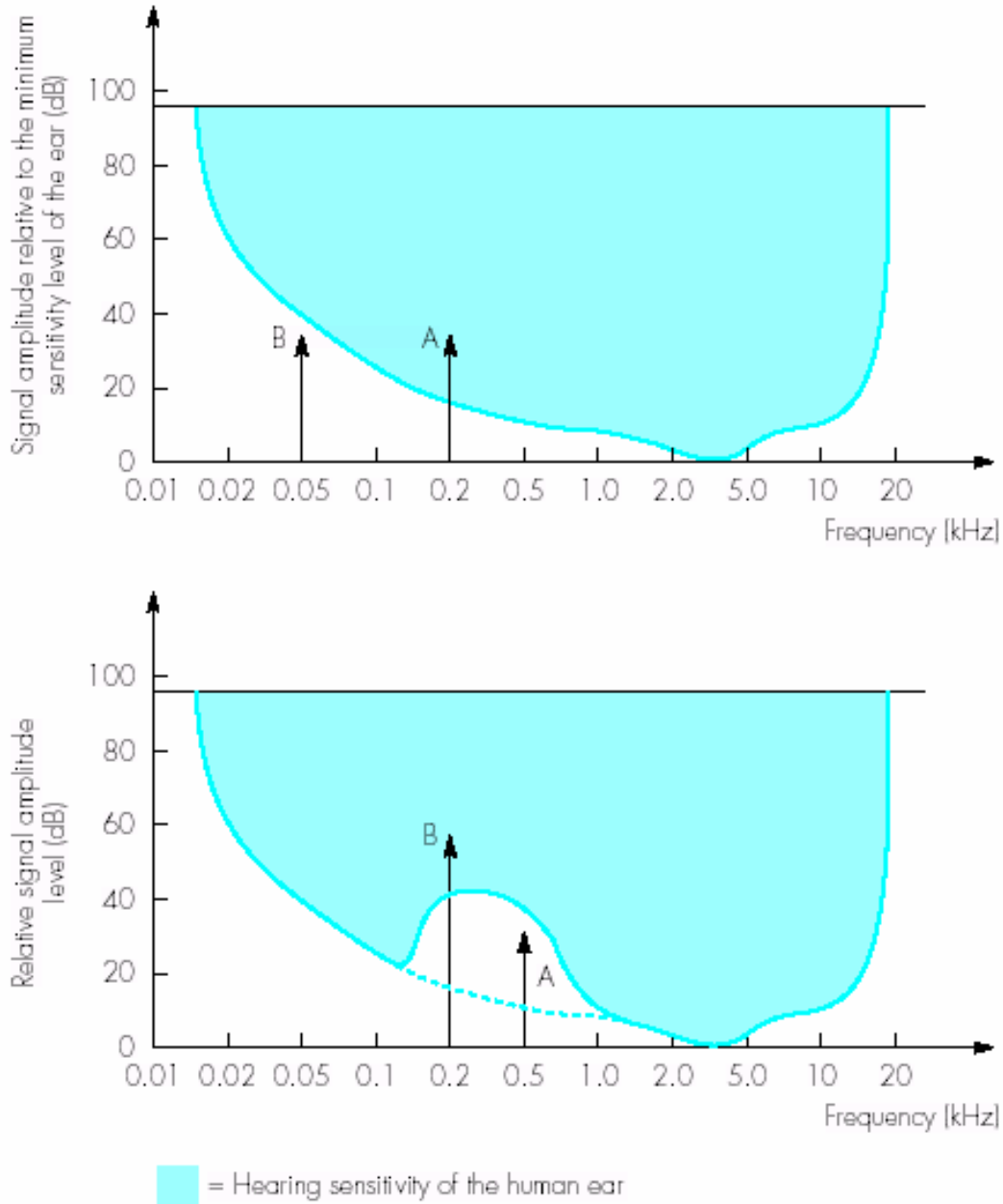
Adaptive DPCM



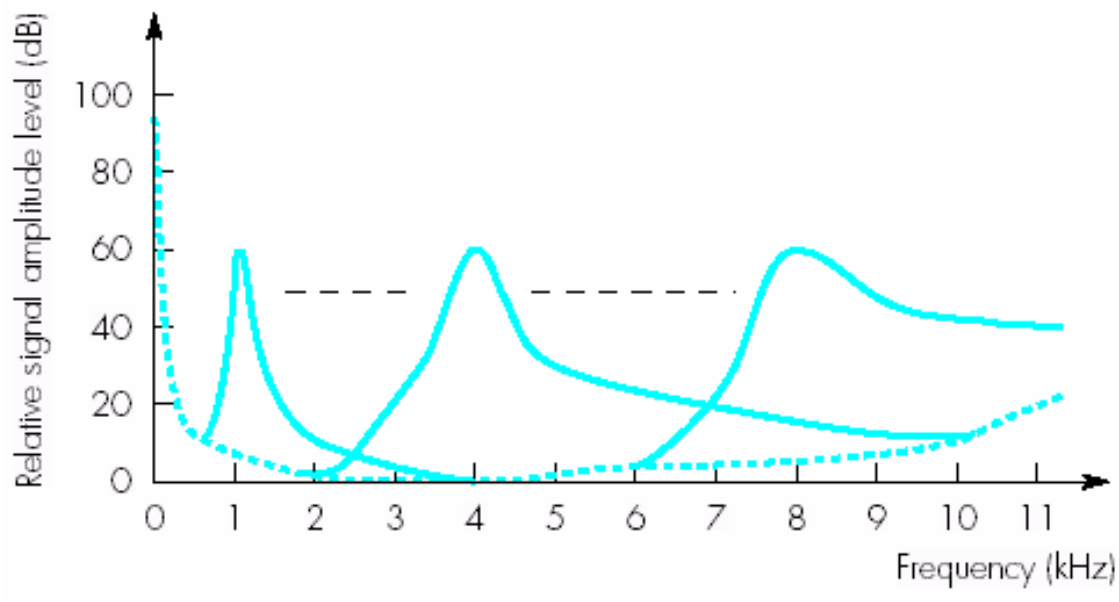
Linear Predictive Coding (LPC)

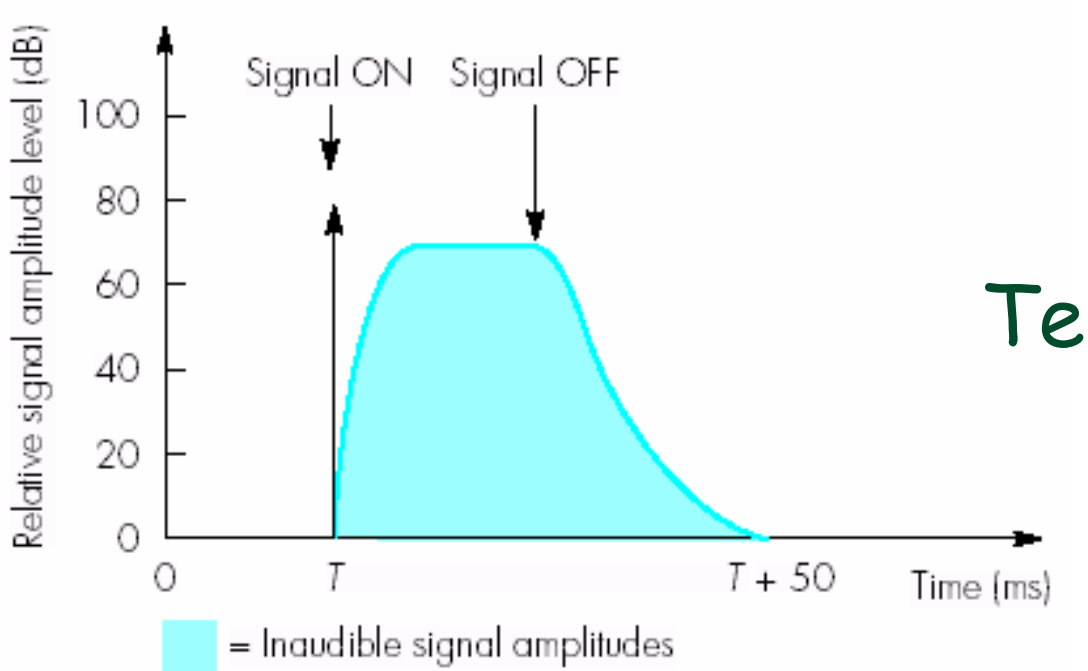


Perceptual properties of the human ear



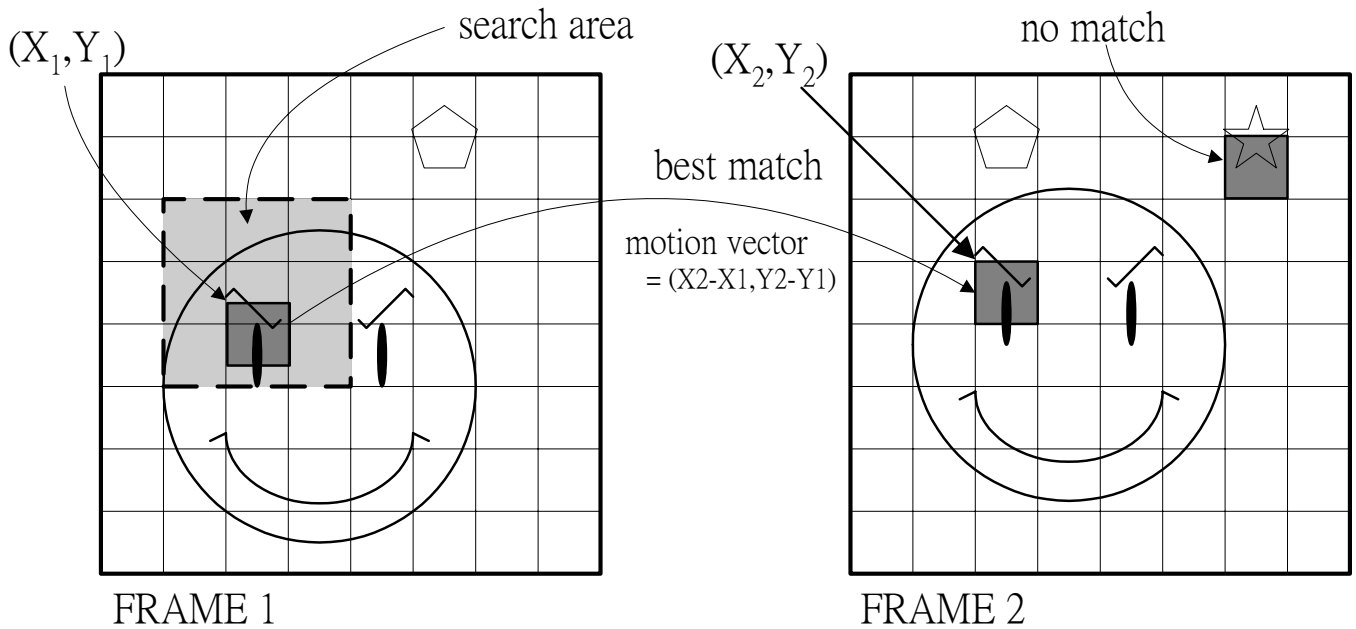
Frequency Masking





Temporal Masking

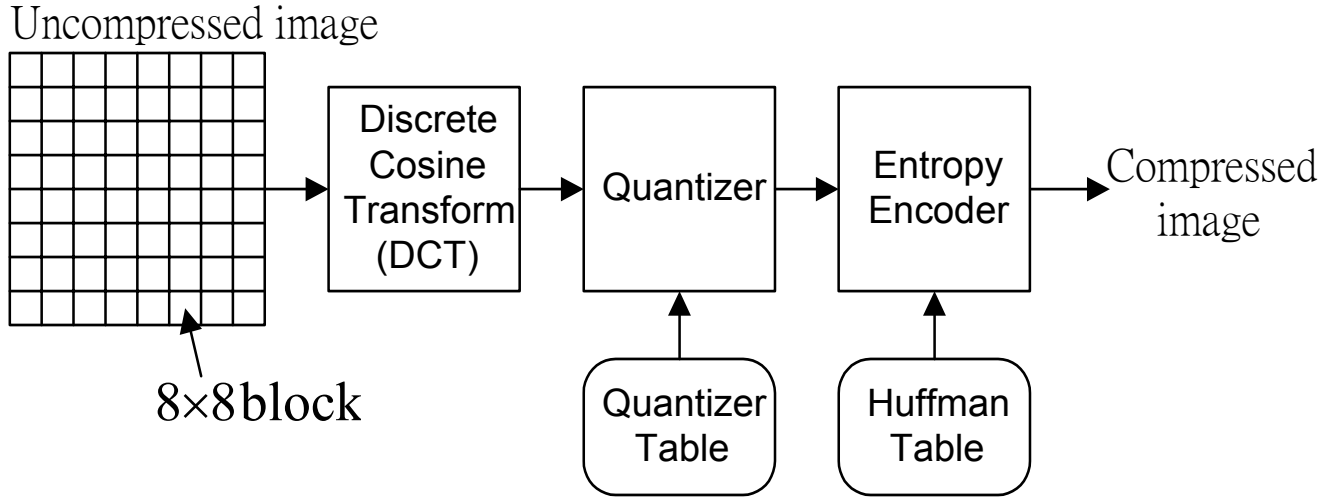
Motion-Compensated Prediction



distance $\leq$ threshold $\Rightarrow$ Transmit the *motion vector*.

distance $>$ threshold $\Rightarrow$ The block is encoded individually. (i.e., intra-coded)

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

172	-18	15	-8	23	-9	-14	19
21	-34	24	-8	-10	11	14	7
-9	-8	-4	6	-5	4	3	-1
-10	6	-5	4	-4	4	2	1
-8	-2	-3	5	-3	3	4	6
4	-2	-4	6	-4	4	2	-1
4	-3	-4	5	6	3	1	1
0	-8	-4	3	2	1	4	0

$$L(i,j) = \left\lfloor \frac{DCT(i,j)}{Quantum(i,j)} \right\rfloor$$

DCT coefficients after quantization

43	3	2	0	0	0	0	0
3	3	2	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantum matrix

4	7	10	13	16	19	22	25
7	10	13	16	19	22	25	28
10	13	16	19	22	25	28	31
13	16	19	22	25	28	31	34
16	19	22	25	28	31	34	37
19	22	25	28	31	34	37	40
22	25	28	31	34	37	40	43
25	28	31	34	37	40	43	46

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.

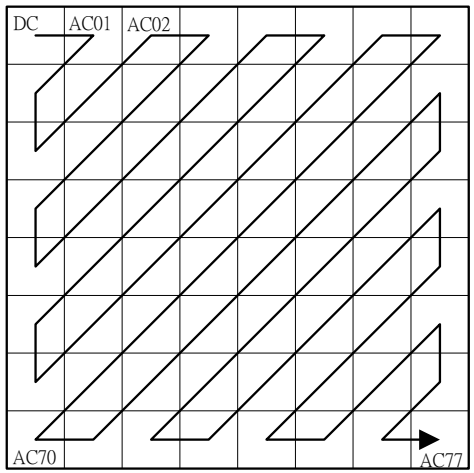
- Quantization errors in the DC and lower-frequency AC coefficients are more easily detectable than quantization errors in the higher-frequency AC coefficients.
  - We use larger step sizes for perceptually less important coefficients.
  - Applications may specify values which optimizes the desired quality 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.



8×8 block

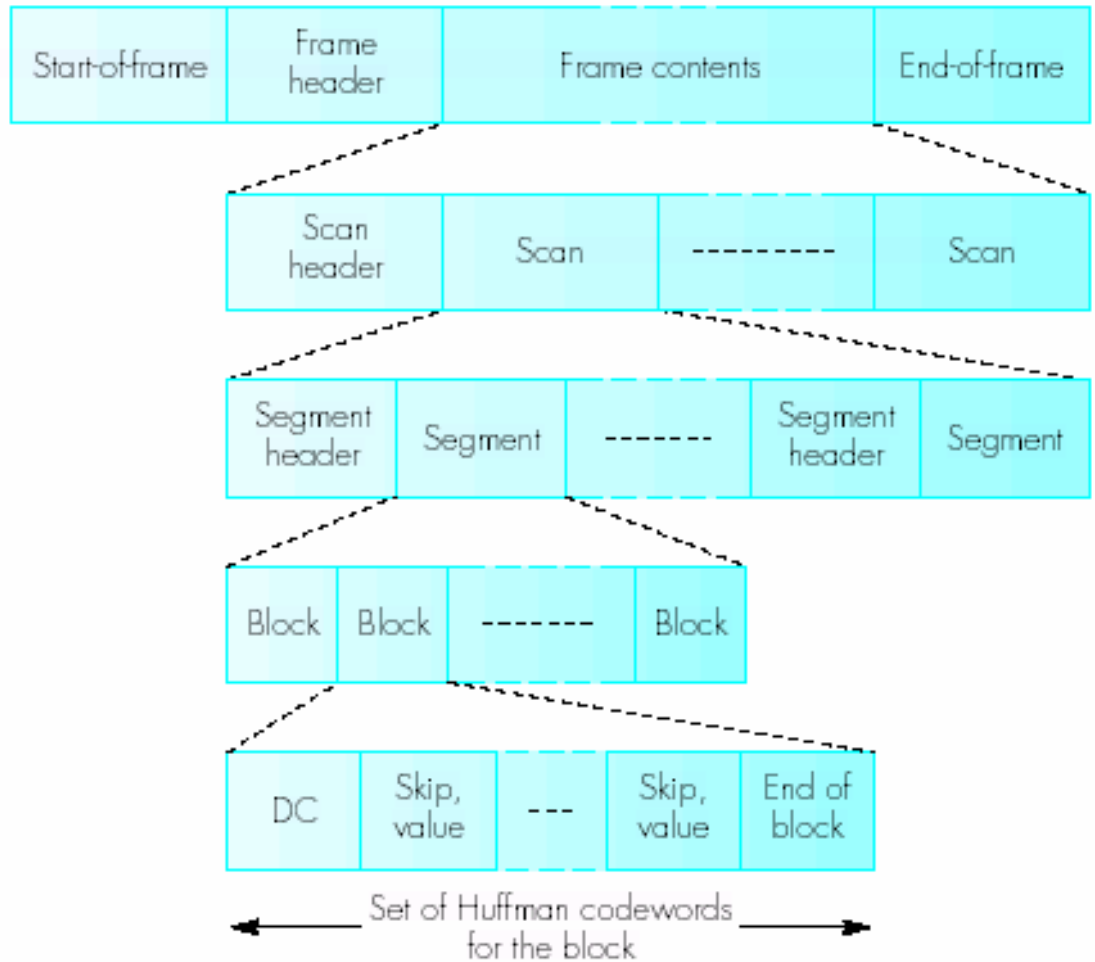
- The codeword for a non-zero AC coefficient has three fields:
  - The number of bits used for the presentation of the AC coefficient.
  - The value of the AC coefficient.
  - The number of subsequent zero AC coefficients.

JPEG: Decoding

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

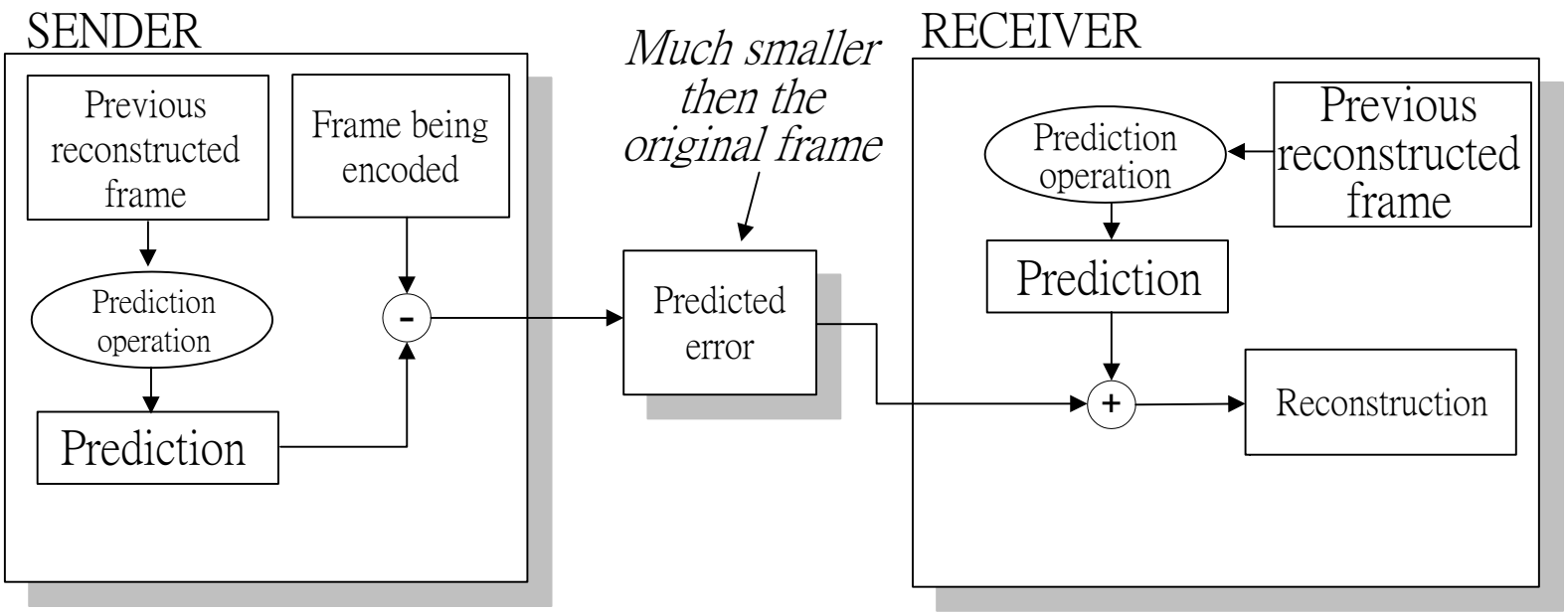
where $C_i, C_j = \frac{1}{\sqrt{2}}$ for $i, j = 0$; otherwise $C_i, C_j = 1$.

- 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.



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

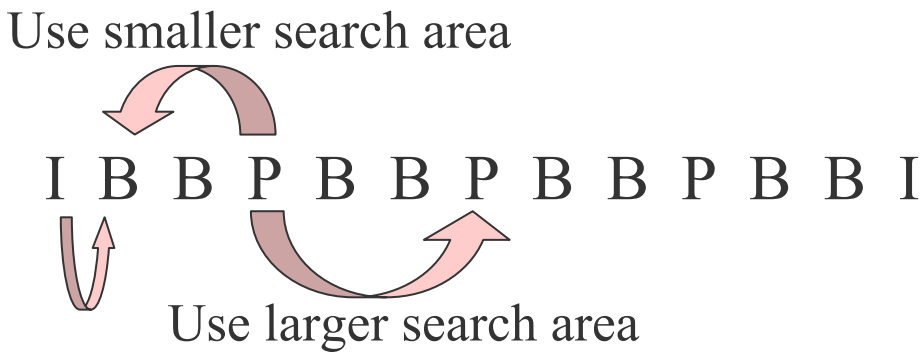
Frame type	I	B	B	P	B	B	P	B	B	P	B	B	I
Display order	1	2	3	4	5	6	7	8	9	10	11	12	13
Bitstream order	1	3	4	2	6	7	5	9	10	8	12	13	11
Time to Display	2	3	4	5	6	7	8	9	10	11	12	13	14

- **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.

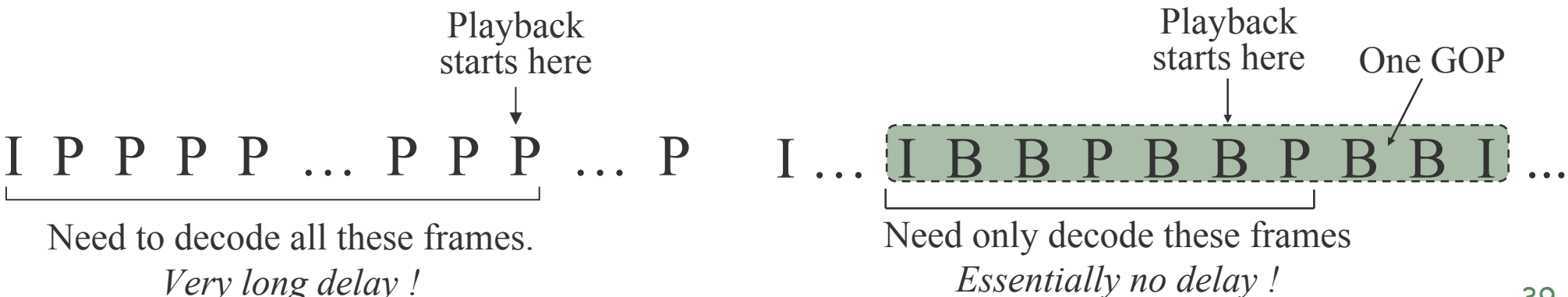
Search Area in MPEG

- 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.



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.

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

Start Code Name	Hexadecimal value
Extension_start_code	0x000001B5
Group_start_code	0x000001B8
Picture_start_code	0x00000100
Reserved	0x000001B0
Reserved	0x000001B1
Reserved	0x000001B6
Sequence_end_code	0x000001B7
Sequence_error_code	0x000001B4
Sequence_header_code	0x000001B3
Slice_start_code 1 ~ 175	0x000001 01 ~ AF
User_data_start_code	0x000001B2

MPEG-1 System Syntax (3 layers)

1. System: ISO 11172 stream = (pack)$^{+}$(iso_11172_end_code)

2. Pack: pack = (pack header)(system header)$^{\text{if in 1st pack}}$(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.)

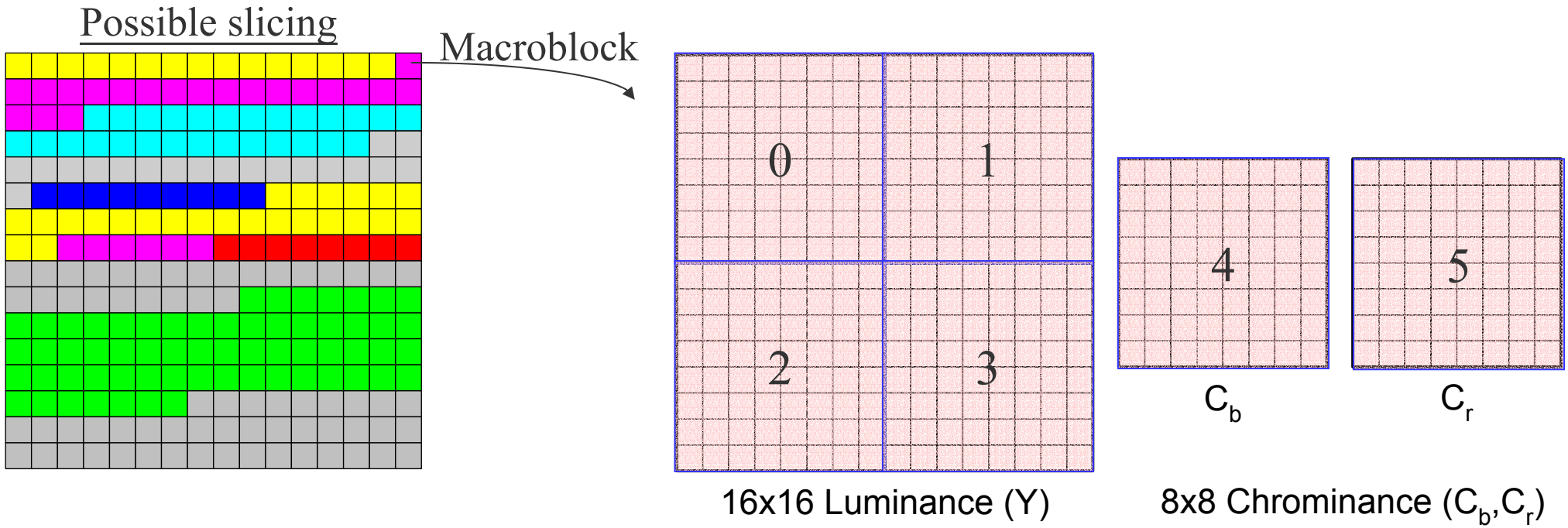
3. Packet: packet = (packet header)(packet_data_byte)^{N}
   - packet header = (packet_start_code_prefix, 0x000001)(stream_id, 1 byte)
     (packet_length, N, 2 bytes)(…)(PTS, DTS, etc.)
     - PTS: presentation time stamp, the time for the stream_id's data to be rendered.
     - DTS: decoding time stamp, 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.

MPEG-1 Video Syntax – layers 5 & 6



5. Macroblock:
   - macroblock = (macroblock header)(block 0)$^{\text{if coded}}$...(block 5)$^{\text{if coded}}$ (end_of_macroblock)$^{\text{if D-picture}}$
   - 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.)$^{\text{if intra}}$(run-level VLC)*(end_of_block)

Parsing MPEG System Streams

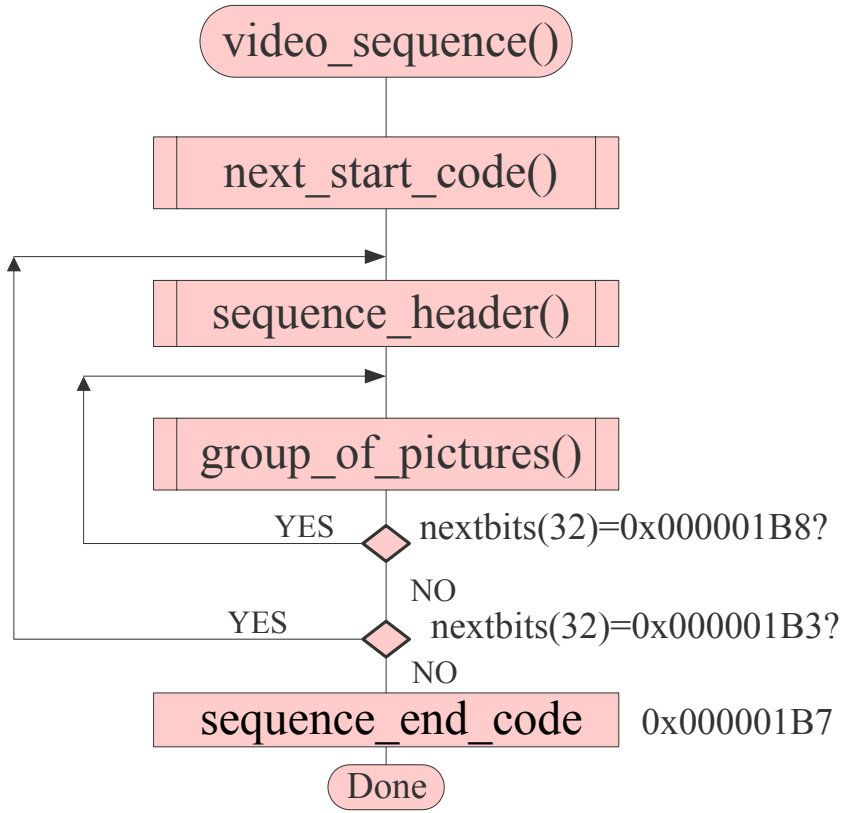
- Motivation:
  - A MPEG stream consists of individual audio and video streams interleaved in units of packets.
  - The coded I-, P- and B-pictures in a video stream have different sensitivities to errors. When transmitted, as specified in RTP, a higher priority can be assigned to I or P pictures.
  - 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.

```
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

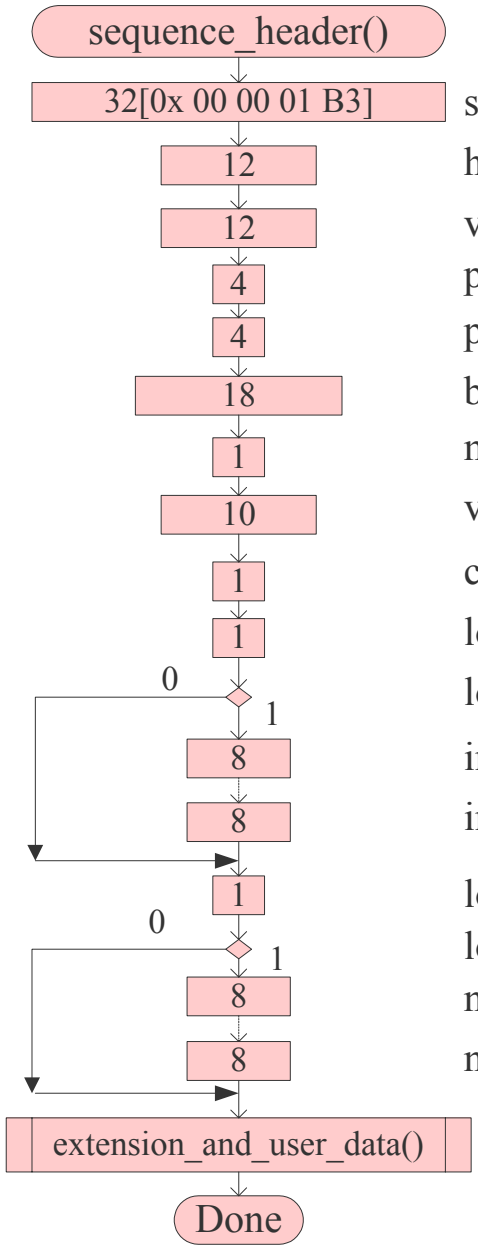


Default intra-Q mtx

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

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



sequence_header_code
horintal_size
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]

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

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

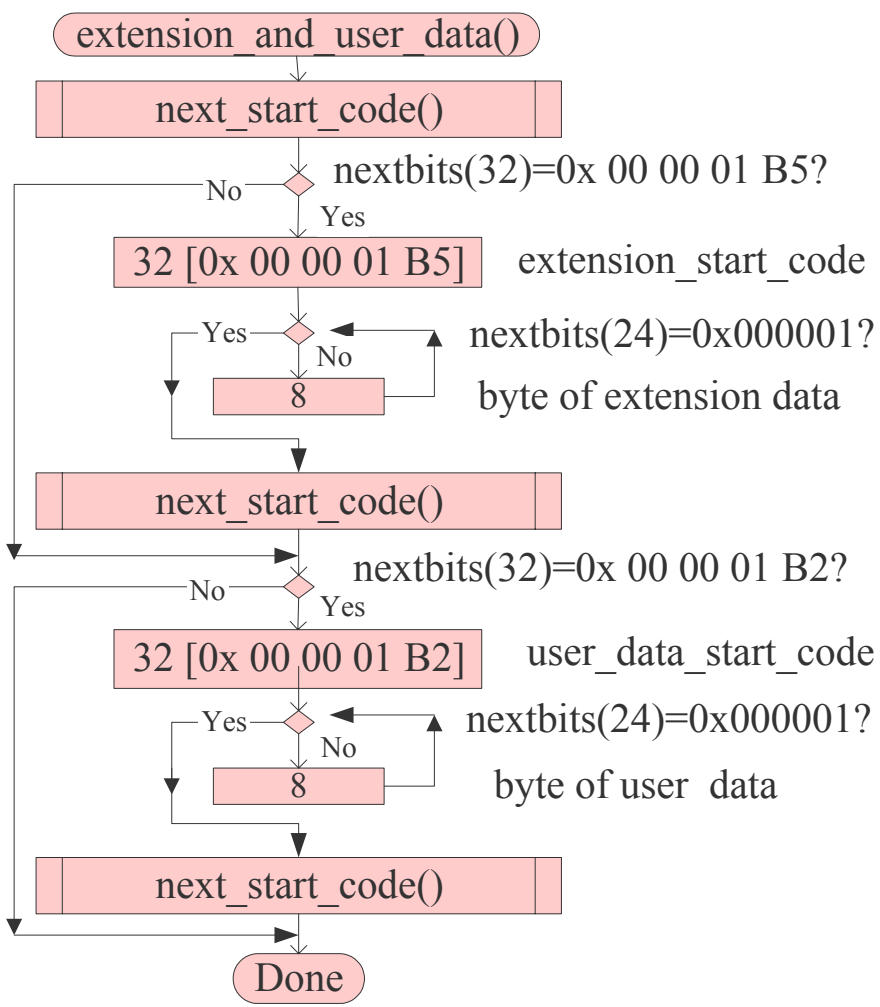
Table 8.3: Picture rate in pictures per second and typical applications

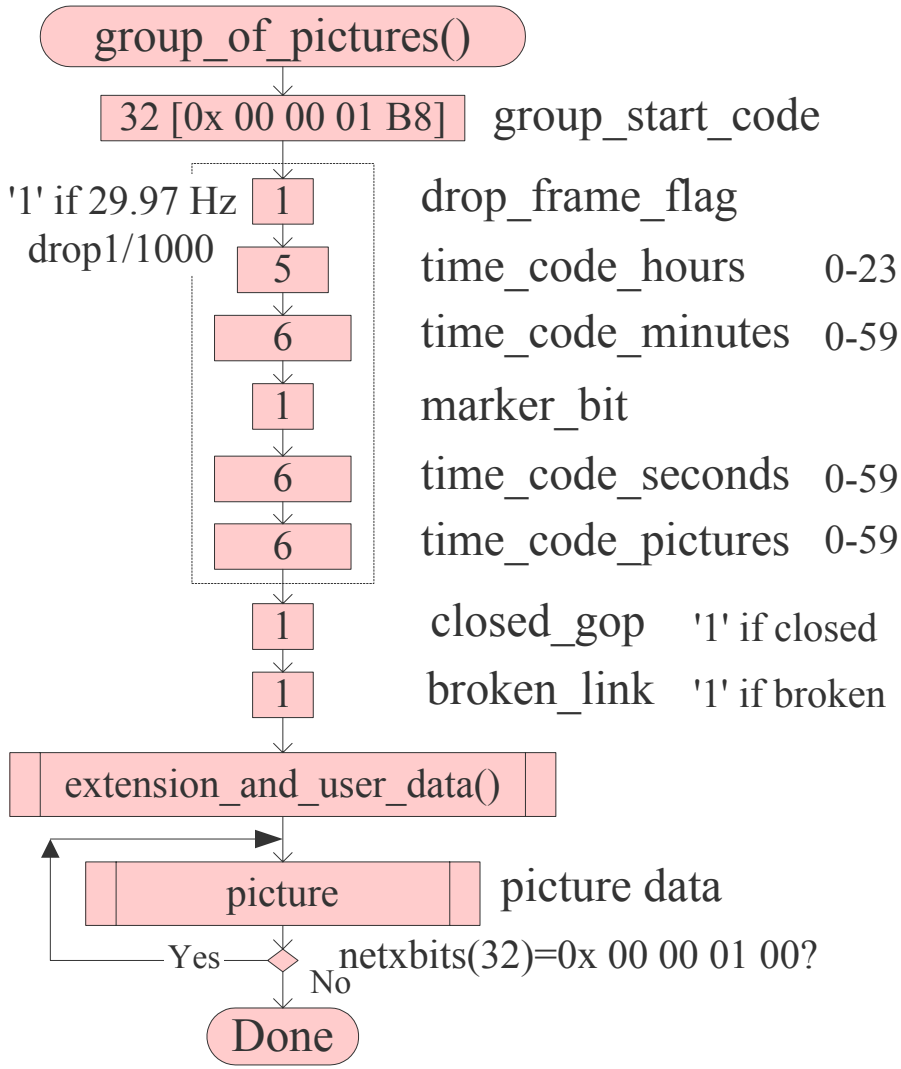
Code	Nominal picture rate	typical applications
0000	Forbidden	
0001	23.976	Movies on NTSC broadcast monitors
0010	24	Movies, commercial clips, animation
0011	25	PAL, SECAM, generic 625/50Hz component video
0100	29.97	Broadcast rate NTSC
0101	30	NTSC profession studio, 525/60Hz component video
0110	50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

Table 8.4: Constrained parameters bounds.

horizontal_size $\leq$ 768 pels; vertical_size $\leq$ 576 lines.
number of macroblocks $\leq$ 396.
(number of macroblocks) x picture_rate $\leq$ 396 x 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





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 */
```

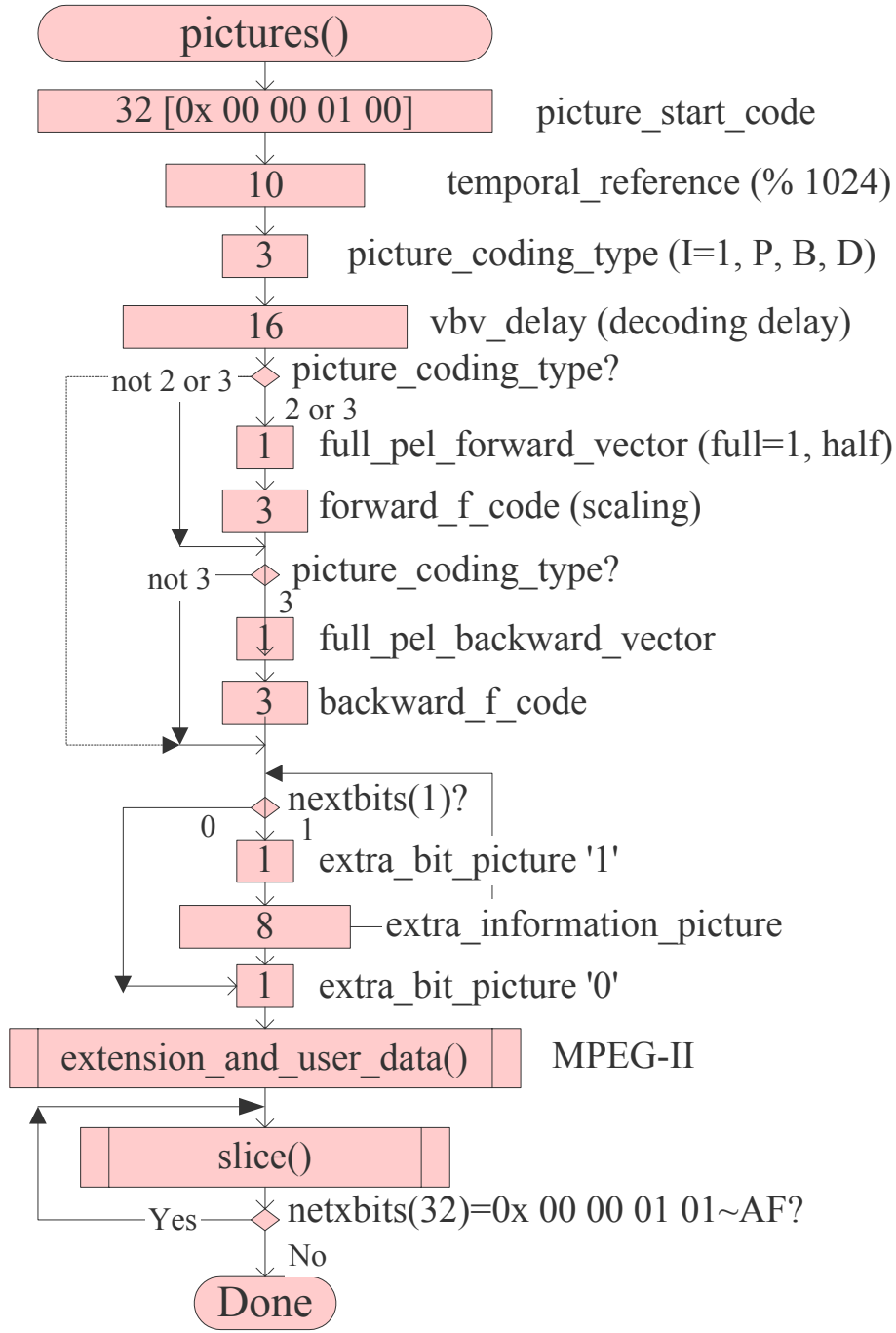
Pictures from ISO 11172-2 2.4.2.5

```
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

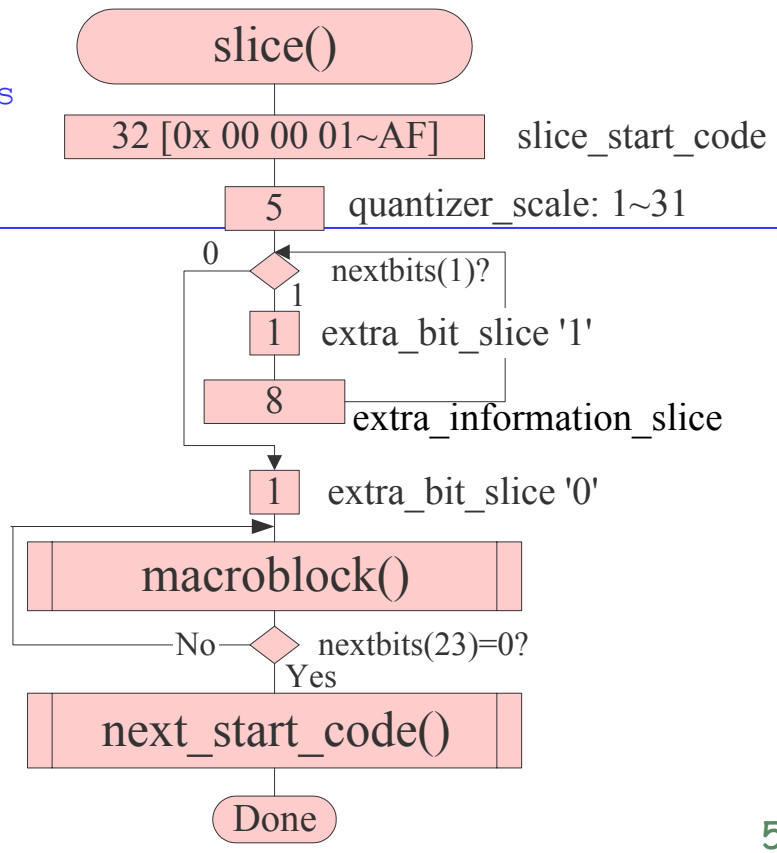
Table 8.5: Picture type codes.

Code	Picture type
000	forbidden
001	I-picture
010	P-picture
011	B-picture
100	D-picture
101	reserved
...	...
111	reserved



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.  1 if(macroblock_quant) quantizer_scale(5); // for changing quantizer scale
7.  2 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. 3 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. 4 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 */
```

5: macro_block_intra

Algorithm for Macroblocks

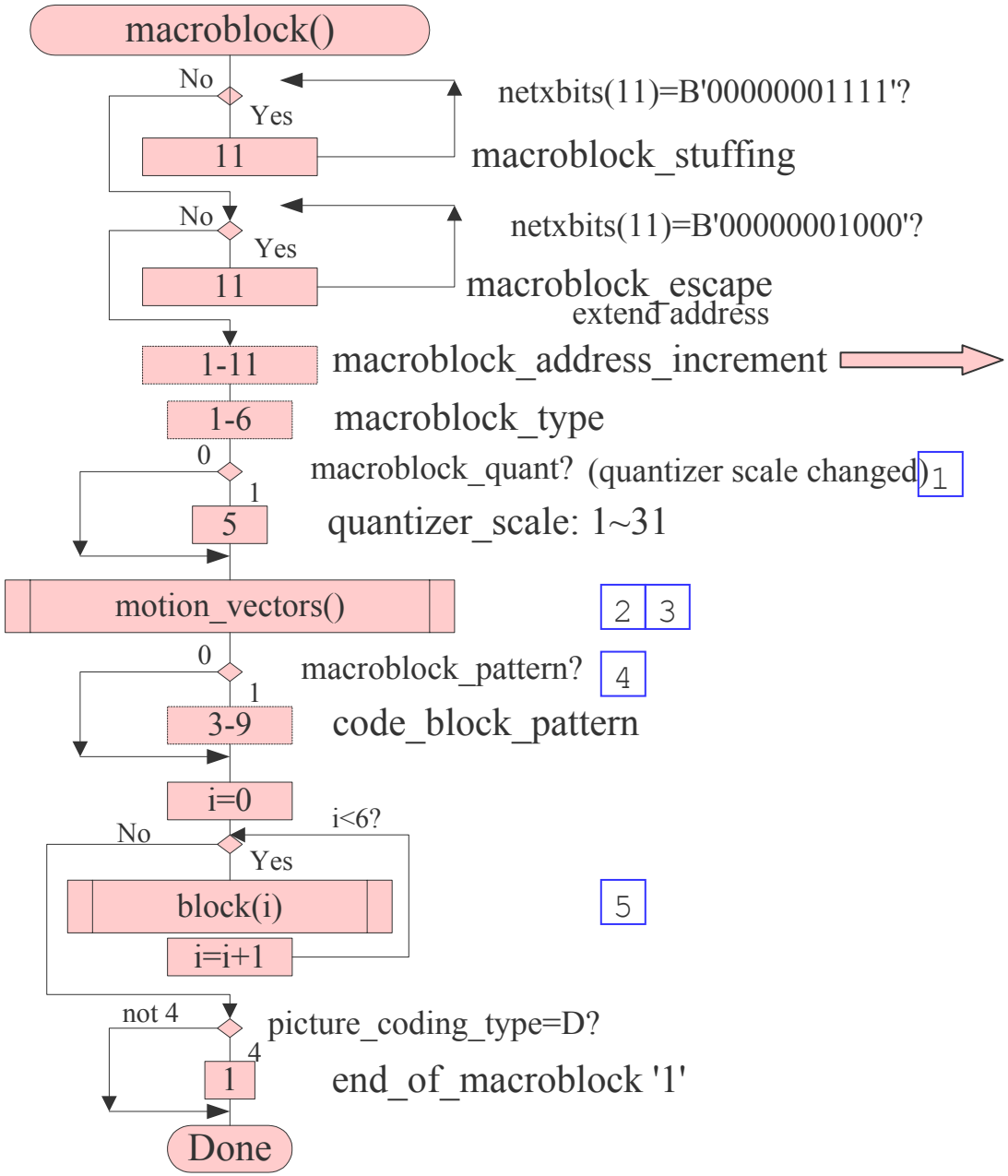


Table 8.6: Variable length codes (VLC) for macroblock_address_increment.

increment value	macroblock_address_increment
macroblock_escape	0000 0001 000 (+= 33)
macroblock_stuffing	0000 0001 111
33	0000 0011 000
32	0000 0011 001
31	0000 0011 010
30	0000 0011 011
29	0000 0011 100
28	0000 0011 101
27	0000 0011 110
26	0000 0011 111
25	0000 0100 000
24	0000 0100 001
23	0000 0100 010
22	0000 0100 011
21	0000 0100 10
20	0000 0100 11
19	0000 0101 00
18	0000 0101 01
17	0000 0101 10
16	0000 0101 11
15	0000 0110
14	0000 0111
13	0000 1000
12	0000 1001
11	0000 1010
10	0000 1011
9	0000 110
8	0000 111
7	0001 0
6	0001 1
5	0010
4	0011
3	010
2	011
1	1

Table 8.7: VLC for macroblock_type in I-, P-, B-, and D-pictures.

	macro-block_ intra	macro-block_ pattern	macro-block_ motion_backward	macro-block_ motion_forward	macro-block_ quant	macro-block_**type** VLC code (1~6 bits)
I	**1**	0	0	0	0	1
	1	0	0	0	1	01
P	0	0	0	1	0	001
	0	1	0	0	0	01
	0	1	0	0	1	0000 1
	0	1	0	1	0	1
	0	1	0	1	1	0001 0
	1	0	0	0	0	0001 1
	1	0	0	0	1	0000 01
B	0	0	0	1	0	0010
	0	0	1	0	0	010
	0	0	1	1	0	10
	0	1	0	1	0	0011
	0	1	0	1	1	0000 11
	0	1	1	0	0	011
	0	1	1	0	1	0000 10
	0	1	1	1	0	11
	0	1	1	1	1	0001 0
	1	0	0	0	0	0001 1
	1	0	0	0	1	0000 01
D	**1**	0	0	0	0	1

5 4 3 2 1 ← VLC **type** code

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.

motion value magnitude	motion_code VLC
0	1
1	01s
2	001s
3	0001 s
4	0000 11s
5	0000 101s
6	0000 100s
7	0000 011s
8	0000 0101 1s
9	0000 0101 0s
10	0000 0100 1s
11	0000 0100 01s
12	0000 0100 00s
13	0000 0011 11s
14	0000 0011 10s
15	0000 0011 01s
16	0000 0011 00s

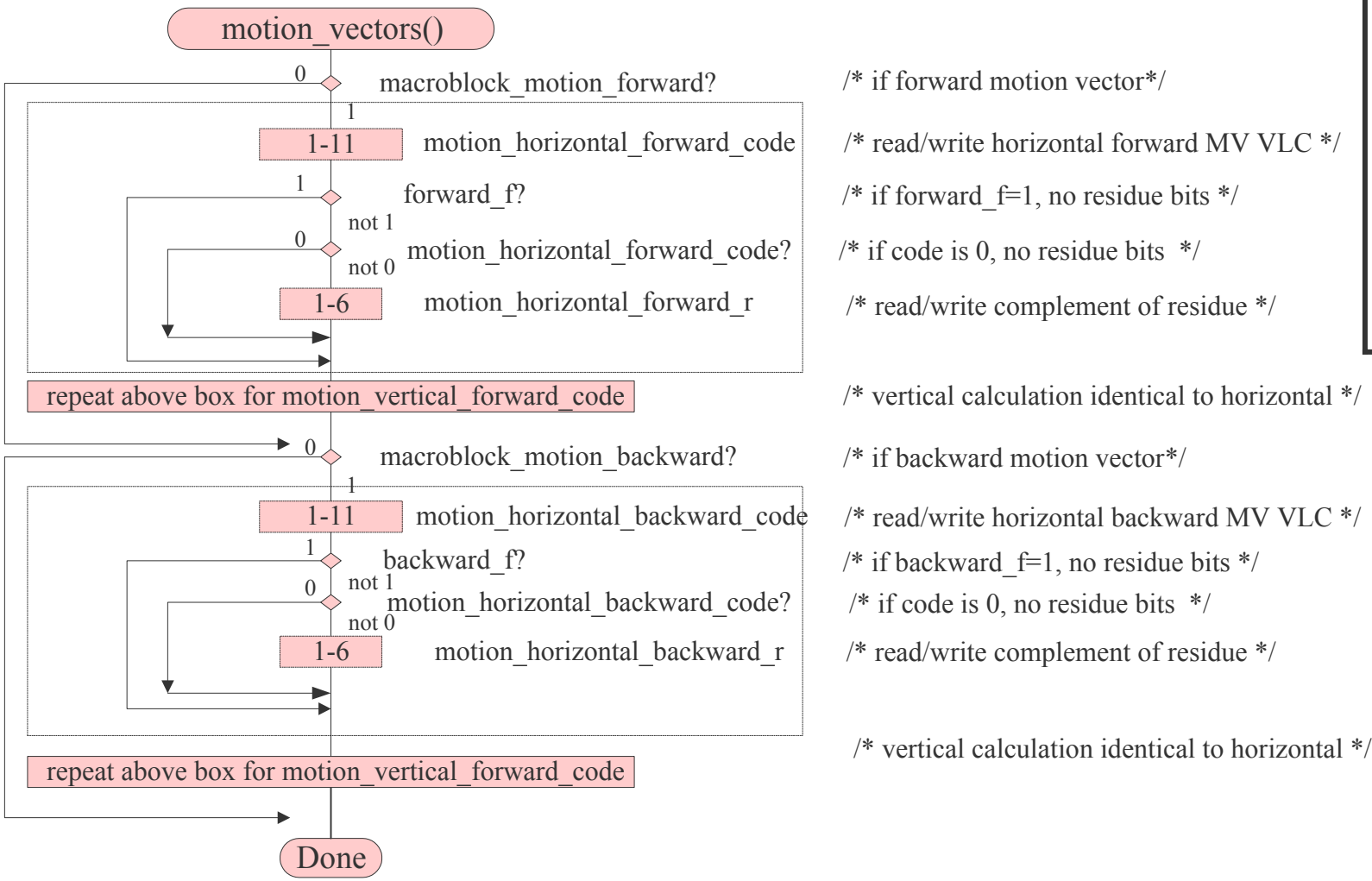


Table 8.8: MPEG-1 code_block_pattern (cbp) VLC codes.

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

cbp		Block # 0123 4 5	code_block_ pattern
decimal	binary	YYYY C_bC_r	VLC code
0	000000	.... . .	Forbidden
1	000001	 c	0101 1
2	000010	.... c .	0100 1
3	000011	.... c c	0011 01
4	000100	...c . .	1101
5	000101	...c . c	0010 111
6	000110	...c c .	0010 011
7	000111	...c c c	0001 1111
8	001000	..c. . .	1100
9	001001	..c. . c	0010 110
10	001010	..c. c .	0010 010
11	001011	..c. c c	0001 1110
12	001100	..cc . .	1001 1
13	001101	..cc . c	0001 1011
14	001110	..cc c .	0001 0111
15	001111	..cc c c	0001 0011
16	010000	.c.. . .	1011
17	010001	.c.. . c	0010 101
18	010010	.c.. c .	0010 001
19	010011	.c.. c c	0001 1101
20	010100	.c.c . .	1000 1
21	010101	.c.c . c	0001 1001
22	010110	.c.c c .	0001 0101
23	010111	.c.c c c	0001 0001
24	011000	.cc. . .	0011 11
25	011001	.cc. . c	0000 1111
26	011010	.cc. c .	0000 1101
27	011011	.cc. c c	0000 0001
28	011100	.ccc . .	0111 1
29	011101	.ccc . c	0000 1011
30	011110	.ccc c .	0000 0111
31	011111	.ccc c c	0000 0011 1
32	100000	c... . .	1010
33	100001	c... . c	0010 100
34	100010	c... c .	0010 000
35	100011	c... c c	0001 1100
36	100100	c..c . .	0011 10
37	100101	c..c . c	0000 1110
38	100110	c..c c .	0000 1100
39	100111	c..c c c	0000 0001 0
40	101000	c.c. . .	1000 0
41	101001	c.c. . c	0001 1000
42	101010	c.c. c .	0001 0100
43	101011	c.c. c c	0001 0000
44	101100	c.cc . .	0111 0
45	101101	c.cc . c	0000 1010
46	101110	c.cc c .	0000 0110
47	101111	c.cc c c	0000 0011 0
48	110000	cc.. . .	1001 0
49	110001	cc.. . c	0001 1010
50	110010	cc.. c .	0001 0110
51	110011	cc.. c c	0001 0010
52	110100	cc.c . .	0110 1
53	110101	cc.c . c	0000 1001
54	110110	cc.c c .	0000 0101
55	110111	cc.c c c	0000 0010 1
56	111000	ccc. . .	0110 0
57	111001	ccc. . c	0000 1000
58	111010	ccc. c .	0000 0100
59	111011	ccc. c c	0000 0010 0
60	111100	cccc . .	111
61	111101	cccc . c	0101 0
62	111110	cccc c .	0100 0
63	111111	cccc c c	0011 00

Blocks

Table 5.3

Y code	C code	size	magnitude range
100	00	**0**	0
00	01	**1**	-1, 1
01	10	**2**	-3…-2, 2…3
101	110	**3**	-7…-4, 4…7
110	1110	**4**	-15…-8, 8…15
1110	11110	**5**	-31…-16, 16…31
11110	111110	**6**	-63…-32, 32…63
111110	1111110	**7**	-127…-64, 64…127
1111110	11111110	**8**	-255…-128, 128…255

difference		size	additional bits
decimal	binary		
+5	…00101	**3**	101
+4	…00100	**3**	100
+3	…00011	**2**	11
+2	…00010	**2**	10
+1	…00001	**1**	1
0	…00000	**0**	-
-1	…11111	**1**	0
-2	…11110	**2**	01
-3	…11101	**2**	00
-4	…11100	**3**	011
-5	…11011	**3**	010

```
1.  block(i){  // from ISO 1172-2 2.4.2.8
2.    if(pattern_code[i]){ // if i^th  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.     if(picture_coding_type!=4){ // if not D-picture
17.       while (nextbits(2)!='10') // while not end-of-block
18.         dct_coeff_next(3-28);   // r/w VLC next run-level → zigzag scan
19.       end_of_block(2);          // r/w '01' EOB
20.     }
21.   }
22. }  /* end block(i) function */
```

DC or First

After Q-ed

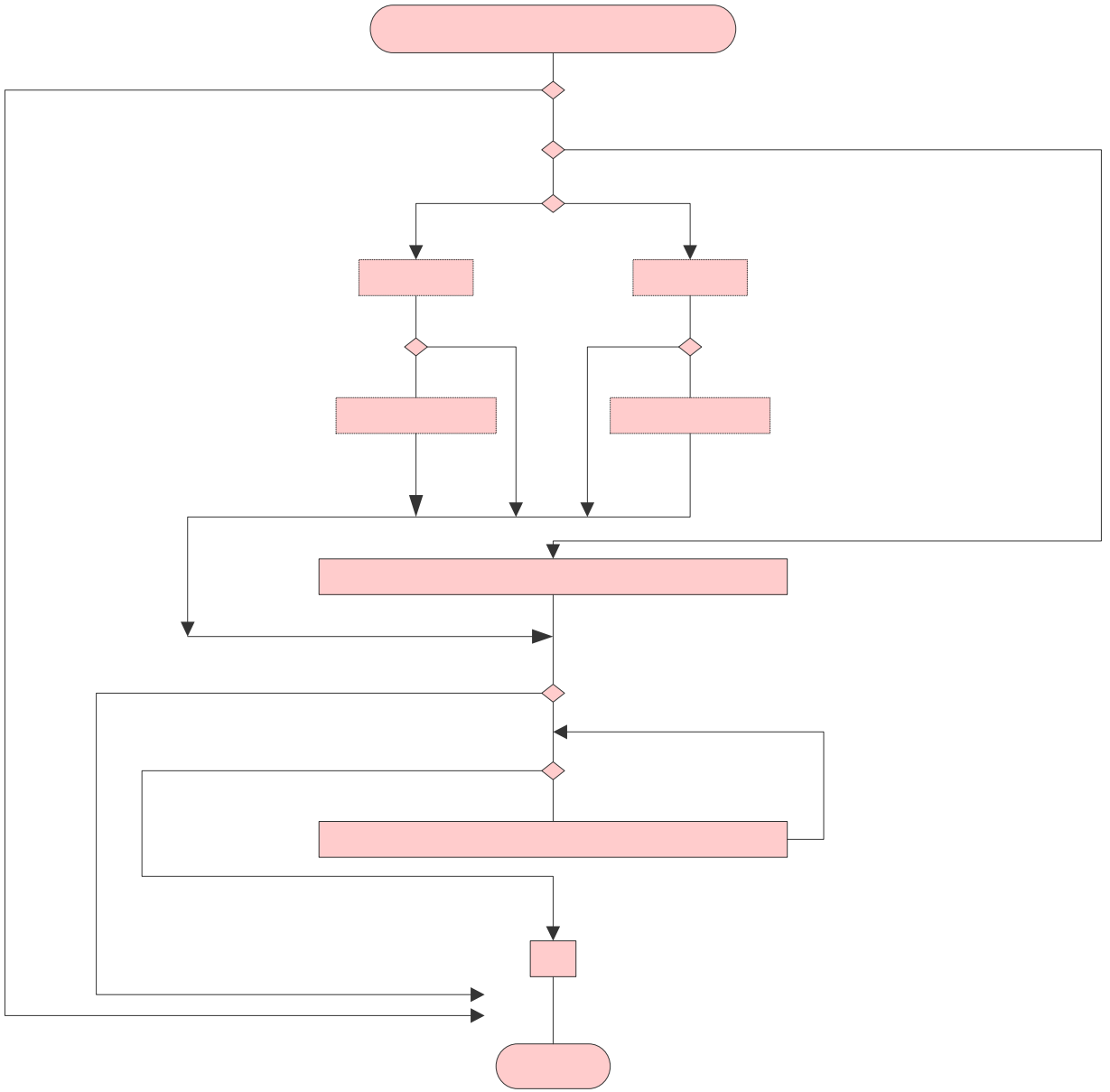
132	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

132 0 0 -1 2 0 0 0 1 EOB

(132-p) 2:-1 1:2 3:1 EOB

Table 5.5 run:level

Algorithm for Blocks



macroblock_i

i>=4

Table 8.10: Video syntax data element summary.

video data element name	set vgpsmb	used vgpsmb	dt	# of bits	value range
backward_f_code	..●...	.....	U	3	1...7
backward_f	..○...	....●.			1,2,4,8,16,32,64
bit_rate	●.....	.....	U	18	1...0x3FFFF
broken_link	.●....	.....	U	1	0,1
closed_gop	.●....	.....	U	1	0,1
coded_block_pattern	....●.	.....	V	3-9	1...63
pattern_code[0]	....○.	.....●			0,1
...	....○.	.....●			0,1
pattern_code[5]	....○.	.....●			0,1
constrained_parameters_flag	●.....	.....	U	1	0,1
dct_coeff_first	.....●	.....	V	2-28	
dct_coeff_next	.....●	.....	V	3-28	
dct_dc_differential	.....●	.....	U	1-8	-255...-1,1...255
dct_dc_size_chrominance	.....●	.....	V	2-8	
dc_size_chrominance	.....○	.....●			0...8
dct_dc_size_luminance	.....●	.....	V	2-7	
dc_size_luminance	.....○	.....●			0...8
end_of_block	.....●	.....○	V	2	'10'
end_of_macroblock	....●.	.....	B	1	'1'
extension_start_code	●●●...	●●●...	B	32	0x000001B5
extra_bit_picture	..●...	..○...	U	1	'0'
extra_bit_slice	...●..	...○..	U	1	'0'
extra_information_picture	..●...	.....	-	8	reserved
extra_information_slice	...●..	.....	-	8	reserved
forward_f_code	..●...	.....	U	3	1...7
forward_f	..○...	....●.			1,2,4,8,16,32,64
full_pel_backward_vector	..●...	.....	U	1	0,1
full_pel_forward_vector	..●...	.....	U	1	0,1
group_extension_data	.●....	.....	-	8	reserved
group_start_code	.●....	●.....	B	32	0x000001B8
horizontal_size	●.....	.....	U	12	1...4095
intra_quantizer_matrix[0]	●.....	.....	U	8	8
...	●.....	.....	U	8	1...255
intra_quantizer_matrix[63]	●.....	.....	U	8	1...255
load_intra_quantizer_matrix	●.....	●.....	U	1	0,1
load_non_intra_quantizer_matrix	●.....	●.....	U	1	0,1
macroblock_address_increment	....●.	.....	V	1-11	1...33
macroblock_escape	....●.	....○.	V	11	'00000001000'

video data element name	set vgpsmb	used vgpsmb	dt	# of bits	value range
macroblock_stuffing	....●.	....○.	V	11	'00000001111'
macroblock_type	....●.	.....	V	1-6	0...63
macroblock_intra	....○.	.....●	-	1	0,1
macroblock_motion_backward	....○.	....●.	-	1	0,1
macroblock_motion_forward	....○.	....●.	-	1	0,1
macroblock_pattern	....○.	....●.	-	1	0,1
macroblock_quant	....○.	....●.	-	1	0,1
marker_bit	●.....	.....	B	1	'1'
motion_horizontal_backward_code	....●.	....●.	V	1-11	-16...16
motion_horizontal_backward_r	....●.	.....	U	1-6	0...63
motion_horizontal_forward_code	....●.	....●.	V	1-11	-16...16
motion_horizontal_forward_r	....●.	.....	U	1-6	0...63
motion_vertical_backward_code	....●.	....●.	V	1-11	-16...16
motion_vertical_backward_r	....●.	.....	U	1-6	0...63
motion_vertical_forward_code	....●.	....●.	V	1-11	-16...16
motion_vertical_forward_r	....●.	.....	U	1-6	0...63
non_intra_quantizer_matrix[0]	●.....	.....	U	8	1...255
...	●.....	.....	U	8	1...255
non_intra_quantizer_matrix[63]	●.....	.....	U	8	1...255
pel_aspect_ratio	●.....	.....	*U*	*4*	*1...14*
picture_coding_type	..●...	..●.●●	U	3	1...4
picture_extension_data	..●...	.....	-	8	reserved
picture_rate	●.....	.....	U	4	1...8
picture_start_code	..●...	.●....	B	32	0x00000100
quantizer_scale	...●●.	.....	U	5	1...31
sequence_end_code	●.....	●.....	B	32	0x000001B7
sequence_extension_data	●.....	.....	-	8	reserved
sequence_header_code	●.....	●.....	B	32	0x000001B3
slice_start_code 1	...●..	..●...	B	32	0x00000101
...	...●..	..●...	B	32	0x000001xx
slice_start_code 175	...●..	..●...	B	32	0x000001AF
temporal_reference	..●...	.....	U	10	0...1023
time_code	.●....	.....	-	25	
user_data	●●●...	.....	-	8	0...255
user_data_start_code	●●●...	●●●...	B	32	0x000001B2
vbv_buffer_size	●.....	.....	U	10	0...1023
vbv_delay	..●...	.....	U	16	0...0xFFFF
vertical_size	●.....	.....	U	12	2,4,...,4094

An Example

- A simple source image comprising two macroblocks, with YC_bC_r values =128, was compressed to create a simple MPEG-1 video stream.

```
1.  Compressed data (hexadecimal format): 46 bytes
2.  000001B302001014FFFFE0A0000001B888080040000001
3.  00000FFFF800000101FA96529488AA25294888000001B7

4.  Compressed data (binary format):
5.  00000000  00000000  00000001  10110011  00000010  00000000  00010000  00010100
6.  11111111  11111111  11100000  10100000  00000000  00000000  00000001  10111000
7.  10000000  00001000  00000000  01000000  00000000  00000000  00000001  00000000
8.  00000000  00001111  11111111  11111000  00000000  00000000  00000001  00000001
9.  11111010  10010110  01010010  10010100  10001000  10101010  00100101  00101001
10. 01001000  10001000  00000000  00000000  00000001  10110111
```

```
1.  Sequence header: 0x000001B302001014FFFFE0A0

2.  00000000  00000000  00000001  10110011  sequence_header_code
3.  00000010  0000                          horizontal_size  = 32 pels
4.  0000  00010000                          vertical_size    = 16 pels
5.  0001                                    pel_aspect_ratio = 1
6.  0100                                    picture_rate     = 4
7.  11111111  11111111  11                  bit_rate         = 0x3ffff (variable)
8.  1                                       marker bit       = 1
9.  00000  10100                            vbv_buffer_size  = 20
10. 0                                       constrained_parameters_flag = 0
11. 0                                       load_intra_quantizer_matrix = 0
12. 0                                       load_inter_quantizer_matrix = 0
```

Parsed…

Parsing GOP

```
1.  Compressed data (hexadecimal format): 46 bytes
2.  000001B302001014FFFFE0A0000001B888080040000001
3.  00000FFFF800000101FA96529488AA25294888000001B7

4.  Compressed data (binary format):
5.  00000000  00000000  00000001  10110011  00000010  00000000  00010000  00010100
6.  11111111  11111111  11100000  10100000  00000000  00000000  00000001  10111000
7.  10000000  00001000  00000000  01000000  00000000  00000000  00000001  00000000
8.  00000000  00001111  11111111  11111000  00000000  00000000  00000001  00000001
9.  11111010  10010110  01010010  10010100  10001000  10101010  00100101  00101001
10. 01001000  10001000  00000000  00000000  00000001  10110111
```

```
1.  Group_of_pictures header 0x000001B880080040

2.  0000000  00000000  00000001  10111000    group_start_code
3.                                           time_code:
4.  1                                          drop_frame_flag
5.  00000                                      time_code_hours    = 0
6.  00  0000                                   time_code_minutes  = 0
7.  1                                          marker_bit
8.  000  000                                   time_code_seconds  = 0
9.  00000  0                                   time_code_pictures = 0
10. 1                                        closed_gop  = 1
11. 0                                        broken_link = 0
12. 00000                                    stuffed bits to byte boundary
```

Parsing Picture & Slice

```
1.  Compressed data (hexadecimal format): 46 bytes
2.  000001B302001014FFFFE0A0000001B888080040000001
3.  00000FFFF800000101FA96529488AA25294888000001B7

4.  Compressed data (binary format):
5.  00000000  00000000  00000001  10110011  00000010  00000000  00010000  00010100
6.  11111111  11111111  11100000  10100000  00000000  00000000  00000001  10111000
7.  10000000  00001000  00000000  01000000  00000000  00000000  00000001  00000000
8.  00000000  00001111  11111111  11111000  00000000  00000000  00000001  00000001
9.  11111010  10010110  01010010  10010100  10001000  10101010  00100101  00101001
10. 01001000  10001000  00000000  00000000  00000001  10110111
```

```
1.  picture header: 0x000001000000FFFF8

2.  00000000  00000000  00000001  00000000   picture_start_code
3.  00000000  00                             temporal_reference  = 0
4.  001                                      picture_coding_type = 1 (I-picture)
5.  111  11111111  11111                     vbv_delay = 0xFFFF (variable rate)
6.  0                                        extra_bit_picture   = 0
7.  00                                       stuffing bits to byte boundary
```

```
1.  Slice header: 0x00000101FA

2.  00000000  00000000  00000001  00000001   slice_start_code
3.                                             slice_vertical_position = 1
4.                                             macroblock_address      = -1 (reset)
5.  11111                                    quantizer_scale = 31
6.  0                                        extra_bit_slice = 0 (reserved 1 for future)
7.  10                                       (belong to macroblock layer)
```

Parsing the others

```
1.  Compressed data (hexadecimal format): 46 bytes
2.  000001B302001014FFFFE0A0000001B888080040000001
3.  00000FFFF800000101FA96529488AA25294888000001B7
```

```
1.  macroblock, block and sequence_end_code compressed data:
2.        10  10010110  01010010  10010100  10001000  10101010  00100101  00101001
3.  01001000  10001000  00000000  00000000  00000001  10110111
```

```
4.  macroblock1
5.  1       macroblock_address_increment = 1
6.  0  1    macroblock_type     = i+q
7.            macroblock_intra = 1
8.            macroblock_quant = 1
9.            cpb(∵ i-picture)= '1111  11'
10. 00101   quantizer_scale          = 5
11. 10   0  dct_dc_size_luminance    = 0
12. 10      end_of_block Y0
13. 100     dct_dc_size_luminance    = 0
14. 10      end_of_block Y1
15. 100     dct_dc_size_luminance    = 0
16. 10      end_of_block Y2
17. 100     dct_dc_size_luminance    = 0
18. 10      end_of_block Y3
19. 00      dct_dc_size_chrominance = 0
20. 10      end_of_block Cb4
21. 00      dct_dc_size_chrominance = 0
22. 10      end_of_block Cr5
```

```
23. macroblock2
24. 1         macroblock_address_increment=1
25. 01        macroblock_type     = i+q
26.             macroblock_intra = 1
27.             macroblock_quant = 1
28.             cpb(∵ i-picture)= '1111  11'
29. 010  00   quantizer_scale          = 8
30. 100       dct_dc_size_luminance    = 0
31. 10        end_of_block Y0
32. 1  00     dct_dc_size_luminance    = 0
33. 10        end_of_block Y1
34. 100       dct_dc_size_luminance    = 0
35. 1  0      end_of_block Y2
36. 100       dct_dc_size_luminance    = 0
37. 10        end_of_block Y3
38. 00        dct_dc_size_chrominance = 0
39. 10        end_of_block Cb4
40. 00        dct_dc_size_chrominance = 0
41. 10        end_of_block Cr5
42. 00        stuffed bits to byte boundary

43. 00000000  00000000  00000001  10110111
44.           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.

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

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

$$MSE(x,y) = \frac{1}{256}\sum_{i=0}^{15}\sum_{j=0}^{15}\left(V_n(x+i,y+j) - V_m(x+dx+i,y+dy+j)\right)^2$$

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, **MD**, is predicted from the one of the preceding macroblock, **PMD**, in the slice. → only the difference, **dMD**, is coded.
    - `dMD = MD – PMD`.
- The predicted 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 MD falling outside of the reference picture.
  - The prediction (B-pictures):

```
pel[i][j] = (pel_for[i][j]+ pel_back[i][j])//2    → 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. → **r_size = f_code −1.**
    - f (forward_f/…): 1, 2, 4, 8, 16, 32, 64. → **$f = 2^{r\_size} = 2^{f\_code-1}$.**
  - 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 → r = motion_r − f + 1.**
- Principal **$dMD_p = motion\_code \times f = motion\_code \times 2^{f\_code-1}$.**
- Residual **$r = |\, dMD_p \,| - |\, dMD \,| > 0$.**
  - **motion_code × f = dMD + Sign(dMD) × (f-1).**
    - ✓ **dMD = motion_code × f − Sign(motion_code) × r.**
  - **Current MD = PMD + dMD = [(PMD+dMD+16×f)%(32×f)]−16×f.**
    - **MD & dMD ∈ [min = −16×f, max = 16×f−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 or CHF = (MD/2) & 1

right_for = YMD = MD>>1 or CMD = (MD/2) >>1

Reconstruction

```
1.  invQ_intra(){
2.    for(m=0; m<8, m++)
3.      for(n=0; n<8, n++){
4.        i=scan[m][n]; // zigzag order index
5.        dct_recon[m][n]=(2*dct_zz[i]*quantizer_scale*intra_quant[m][n])/16;
6.        if((dct_recon[m][n]&1)==0) dct_recon[m][n]-= Sign(dct_recon[m][n]); // oddify
7.        if(dct_recon[m][n]>2047) dct_recon[m][n]=2047;   // clamp to max
8.        if(dct_recon[m][n]<-2048) dct_recon[m][n]=-2048; // clamp to min
9.      }
10.  dct_recon[0][0]=dct_zz[0]*8; // overrule calculation for DC coefficient
11. }
```

```
1.  invQ_intra_Y0(){ // similar for Cb [dct_dc_Cb_past], Cr [dct_dc_Cr_past]
2.    invQ_intra();
3.    if((macroblock_address - past_intra_address)>1) // macroblocks skipped
4.      dct_recon[0][0] += (128*8);
5.    else dct_recon[0][0] += dct_dc_y_past;          // previous block is prediction
6.    dct_dc_y_past = dct_recon[0][0];              // set prediction for the next Y blocks
7.  }
```

```
1.  invQ_intra_Ynext(){ // for Y1, Y2, & Y3
2.    invQ_intra();
3.    dct_recon[0][0] += dct_dc_y_past;             // previous block is prediction
4.    dct_dc_y_past = dct_recon[0][0];              // set prediction for the next Y blocks
5.  }
```

```
1.  invQ_inter(){
2.    for(m=0; m<8, m++)
3.      for(n=0; n<8, n++){
4.        i=scan[m][n]; // zigzag order index
5.     dct_recon[m][n]=(2*dct_zz[i]+Sign(dct_zz[i]))*quantizer_scale*intra_quant[m][n]/16;
6.        if((dct_recon[m][n]&1)==0) dct_recon[m][n]-= Sign(dct_recon[m][n]); // oddify
7.        if(dct_recon[m][n]>2047) dct_recon[m][n]=2047;   // clamp to max
8.        if(dct_recon[m][n]<-2048) dct_recon[m][n]=-2048; // clamp to min
9.        if(dct_zz[i]==0) dct_recon[m][n]=0; // force zeroed dequantized coefficient
10.     }                                     // in case of skipped macroblocks
11. }
```

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.

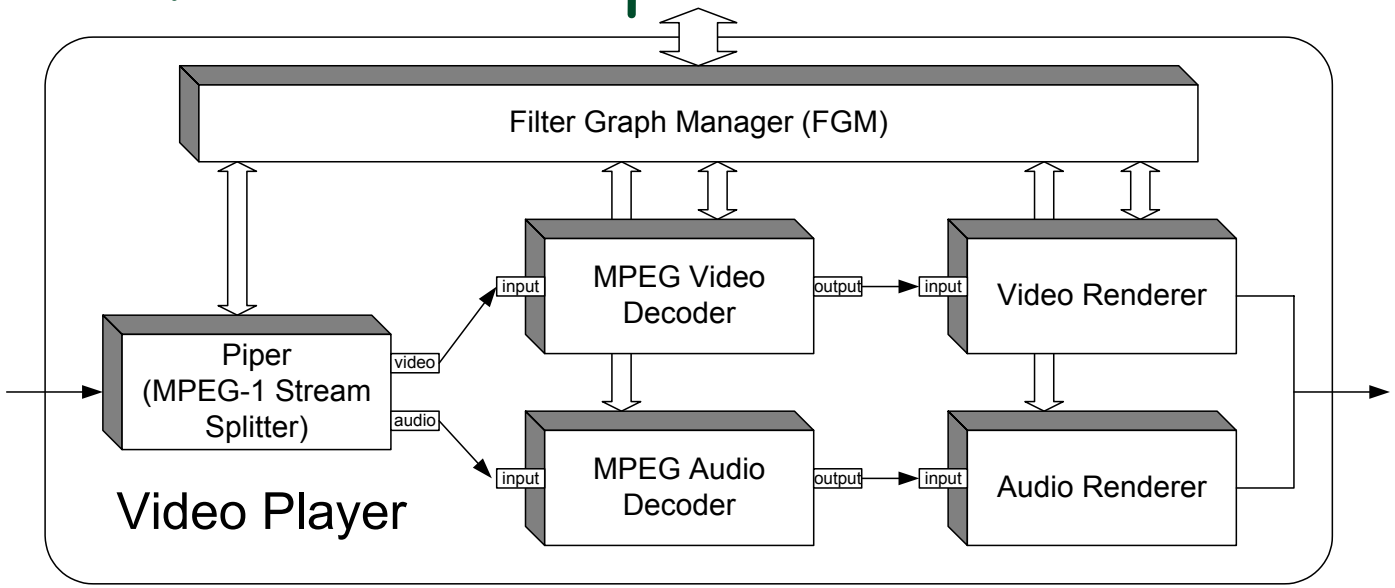
Perceptual Audio Coder

- 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…

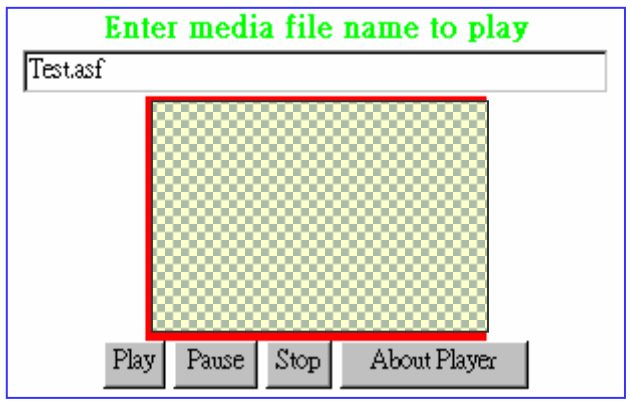
- Filter Graph Editor (graphedt.exe) from DirectX SDK 8.0 provides visual configuration of filter graphs.
- PlayWnd.exe (only 28KBytes) from SDK can playback most media files.

```
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. } // …
```

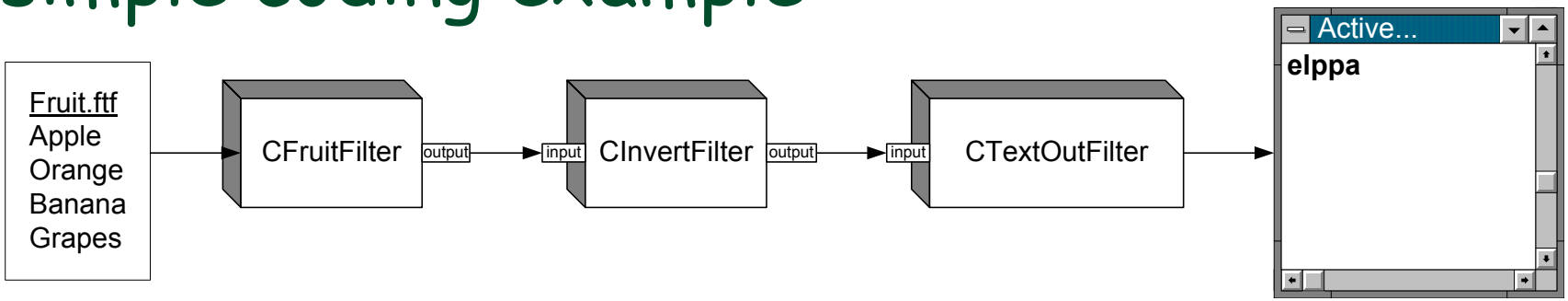
```
1.  void PauseClip(void){ HRESULT hr;  if (!pMC) return;
2.    if((g_psCurrent == Paused) || (g_psCurrent == Stopped)){
3.      hr = pMC->Run();  g_psCurrent = Running;
4.    }else{  hr = pMC->Pause(); g_psCurrent = Paused; }
5.    UpdateMainTitle();
6.  }
```

Embedded Media Player ActiveX control

```
1.  <HTML><HEAD><TITLE>Embedding Media Player Control</TITLE></HEAD>
2.  <BODY BGCOLOR="#FFFFFF" TEXT="#00FF00"><CENTER><BR>
3.  <STRONG>Enter media file name to play</STRONG><BR>
4.  <INPUT TYPE="TEXT" ID=MediaFile name="MyField" VALUE="Test.asf" SIZE=40><BR>
5.  <OBJECT ID="MyObj" classid="CLSID:22D6F312-B0F6-11D0-94AB-0080C74C7E95"
6.    CODEBASE="http://activex.microsoft.com/…/nsmp2inf.cab"
7.    type="application/x-oleobject">
8.    <PARAM NAME="AnimationAtStart" VALUE="0">
9.    <PARAM NAME="ShowControls" VALUE="0">
10.   <PARAM NAME="AutoStart" VALUE="0">
11.   <PARAM NAME="VideoBorderWidth" VALUE="5">
12.   <PARAM NAME="VideoBorderColor" VALUE="255">
13. </OBJECT><BR>
14. <INPUT TYPE="BUTTON" NAME="PlayBtn"  VALUE="Play" OnClick="PlayStream()">
15. <INPUT TYPE="BUTTON" Name="PauseBtn" VALUE="Pause" OnClick="PauseStream()">
16. <INPUT TYPE="BUTTON" NAME="StopBtn"  VALUE="Stop" OnClick="StopStream()">
17. <INPUT TYPE="BUTTON" NAME="AboutBtn" VALUE="About Player" OnClick="MyObj.AboutBox()">
18. <BR></CENTER>
19. <SCRIPT LANGUAGE="JavaScript">
20. <!--
21. function PlayStream(){ MyObj.FileName=MyField.value; MyObj.Play(); }
22. function PauseStream(){ if(MyObj.PlayState==1){ MyObj.Play(); } else
23.   if(MyObj.PlayState==2) { MyObj.Pause(); } }
24. function StopStream(){ MyObj.Stop(); MyObj.CurrentPosition=0; }
25. -->
26. </SCRIPT> </BODY> </HTML>
```

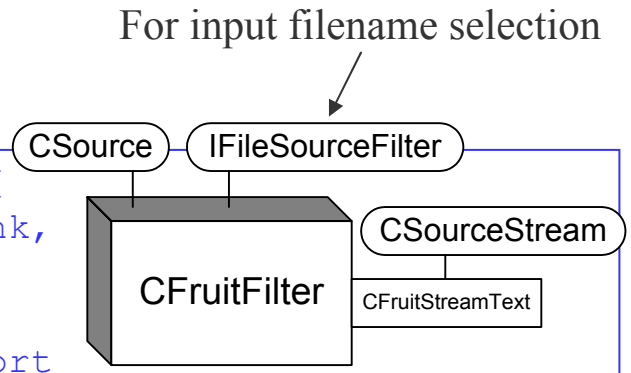


A simple coding example



- 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 CFruitFilter




```
class CFruitFilter: public CSource, public IFileSourceFilter {
public: static CUnknown * WINAPI CreateInstance(LPUNKNOWN lpunk,
          HRESULT *phr); // for creating an object
private:
  DECLARE_IUNKNOWN     // Required for IFileSourceFilter support
  STDMETHODIMP GetCurFile(LPOLESTR * ppszFileName,AM_MEDIA_TYPE *pmt);          // added
  STDMETHODIMP Load(LPCOLESTR pszFileName, const AM_MEDIA_TYPE __RPC_FAR *pmt); // added
  STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv); // customized
  // It is only allowed to to create these objects with CreateInstance
  CFruitFilter(LPUNKNOWN lpunk, HRESULT *phr);   }; // private constructor
  OLECHAR m_szFileName[_MAX_PATH];  // added
};
```

- "**static**" CreateInstance() 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 g_Templates[] and g_cTemplates 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
CFactoryTemplate g_Templates[] = {
  { L"ABC - Fruit Source Filter" , &CLSID_FruitFilter,
    CFruitFilter::CreateInstance, NULL , &sudFruitFilter }
};
int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

- For instance, FGM uses the 3rd item to get a pointer to the CreateInstance() function, which in turns is called for object creation.

More about CFruitFilter

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

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

```
// Initialise a FruitStreamText object so that we have a pin.
CFruitFilter::CFruitFilter(LPUNKNOWN lpunk, HRESULT *phr) :
 CSource(NAME("Fruit Source Filter"), lpunk, CLSID_FruitFilter){
  CAutoLock cAutoLock(&m_cStateLock);
  new CFruitStreamText(phr, this, L"Text!");
  if (m_paStreams[1] == NULL) *phr = E_OUTOFMEMORY;    return;
}
```

- FGM then checks the exposed interfaces of **CFruitFilter** by IUnKnown::NonDelegatingQueryInterfact().

```
STDMETHODIMP CFruitFilter::NonDelegatingQueryInterface(REFIID riid, void ** ppv){
  CheckPointer(ppv,E_POINTER);
  if (riid == IID_IFileSourceFilter)
    return GetInterface((IFileSourceFilter *) 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.

Source Stream Class

```
class CFruitStreamText: public CSourceStream {
public:
  HRESULT OnThreadDestroy();
  CFruitStreamText(HRESULT *phr, CFruitFilter *pParent, LPCWSTR pPinName);
  ~CFruitStreamText();
  HRESULT FillBuffer(IMediaSample *pms); // to fill the buffer with data
  // Ask for buffers of the size appropriate to the agreed media type
  HRESULT DecideBufferSize(IMemAllocator *pIMemAlloc, ALLOCATOR_PROPERTIES *pProperties);
  HRESULT GetMediaType(int iPosition, CMediaType *pmt);
  HRESULT CheckMediaType(const CMediaType *pMediaType); // Verify if supported
  HRESULT SetMediaType(const CMediaType *pMediaType); // Set the agreed media type
  HRESULT OnThreadCreate(void);
  HRESULT Init(void) { return CallWorker(CMD_INIT); }
  HRESULT Exit(void) { return CallWorker(CMD_EXIT); }
  HRESULT Run(void) { return CallWorker(CMD_RUN); }
  HRESULT Pause(void) { return CallWorker(CMD_PAUSE); }
  HRESULT Stop(void) { return CallWorker(CMD_STOP); }
private:
  CRefTime m_rtSampleTime;                  // The time stamp for each sample
  ifstream m_inFile;   int m_index;
}; // CFruitStreamText
```

- 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.

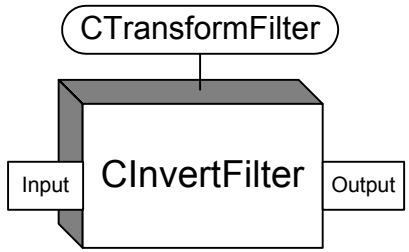
- Stopping:
  - When a "Stop()" command of **IMediaControl** is called, OnThreadDestroy() will close the input file and end the thread.

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 "RBG555" 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 CFruitFilter, 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.