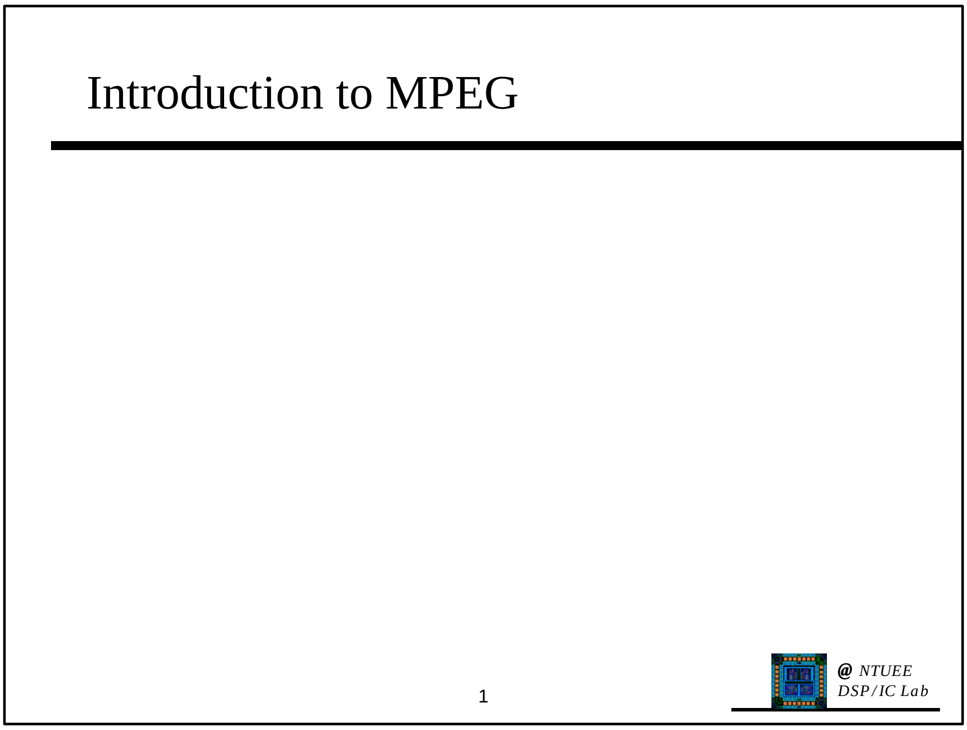
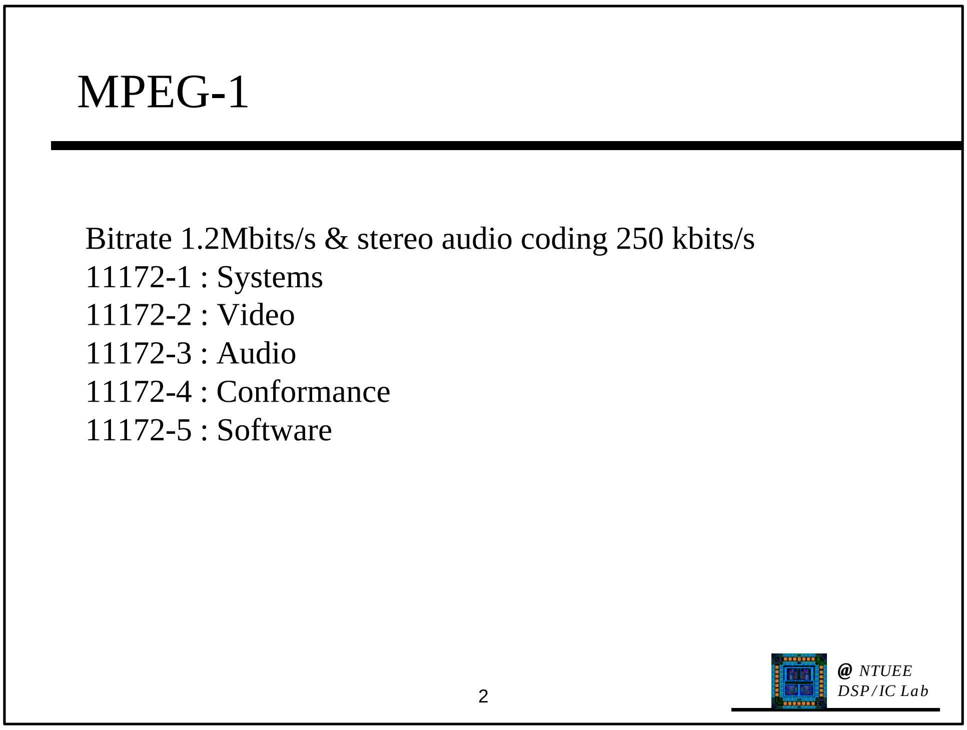


Introduction to MPEG



MPEG-1

Bitrate 1.2Mbits/s & stereo audio coding 250 kbits/s
11172-1 : Systems
11172-2 : Video
11172-3 : Audio
11172-4 : Conformance
11172-5 : Software

MPEG-1 Audio

Three layers
- I 128kbits/s
- II 128kbits/s
- III 64kbits/s the best

four models
- mono
- stereo
- dual (two separate channels)
- joint stereo

Picture Type and Structure

Picture, Frame, and Field ---MPEG-1 provides picture only (no field)

Source Input Format (SIF) ---each picture is divided into a series of macroblocks and the YCbCr color components are always interleaved.

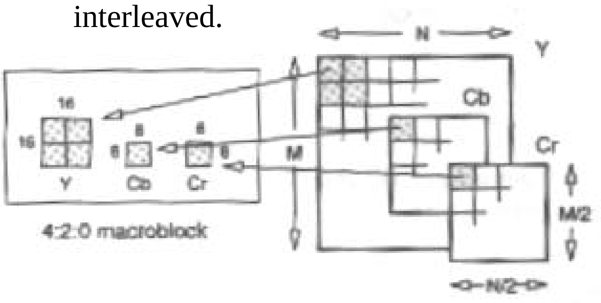
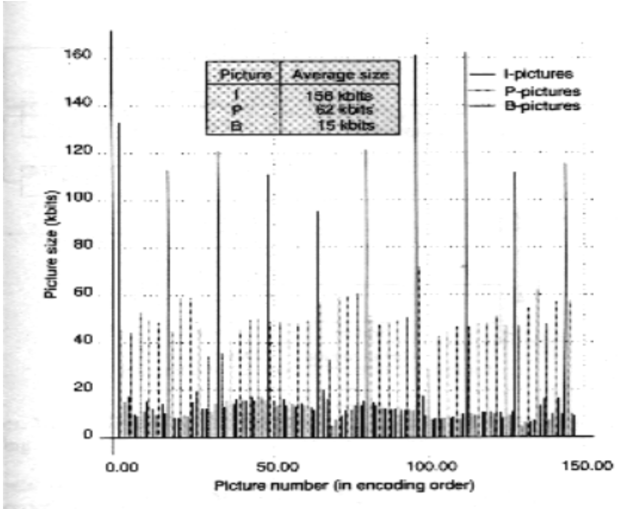
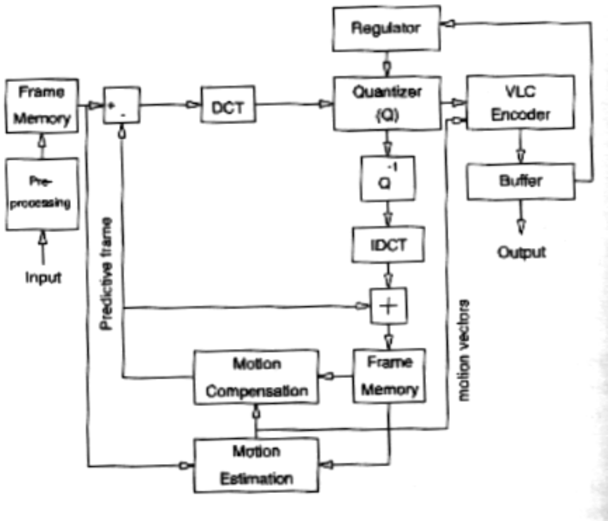


Figure 6.2 Definition of a macroblock in MPEG-1.

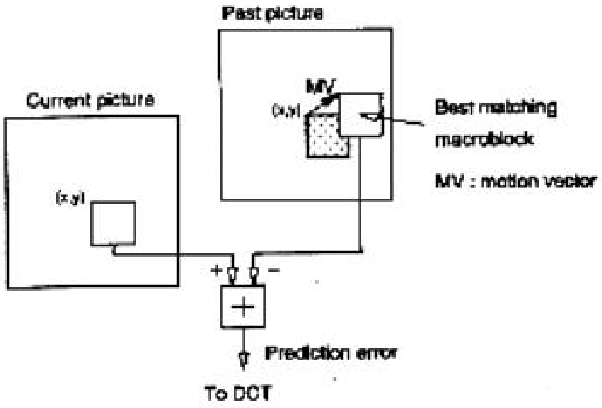
Picture Structure



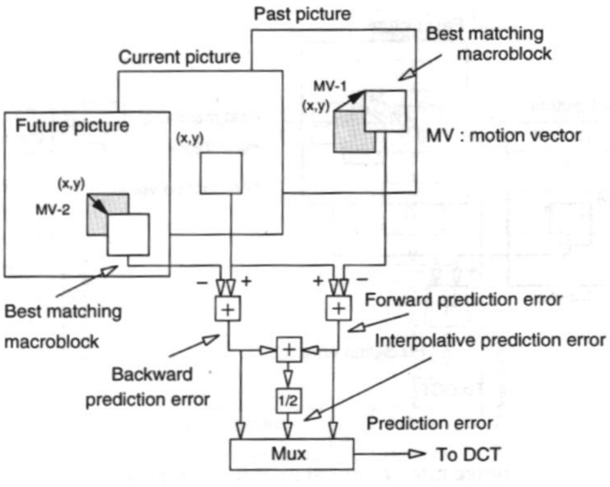
Block Diagram of an MPEG-1 Encoder



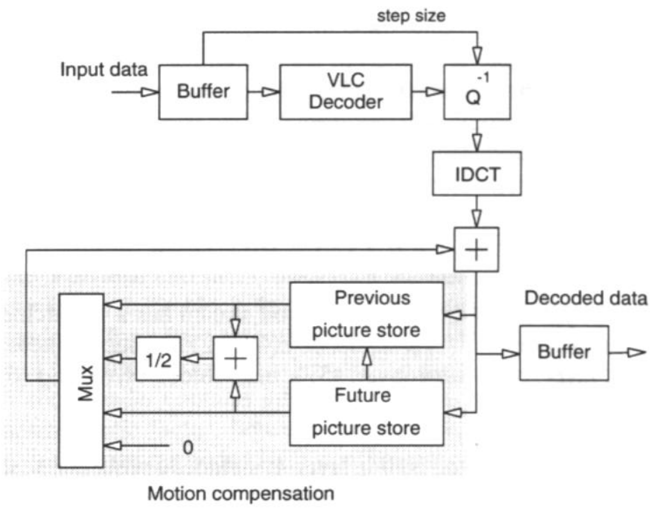
Forward Motion Compensation



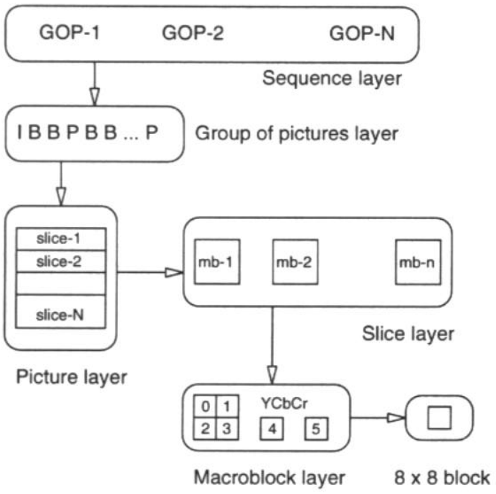
Bidirectional Motion Compensation



Block Diagram of an MPEG Decoder



Syntax layers in MPEG-1 Video Coding



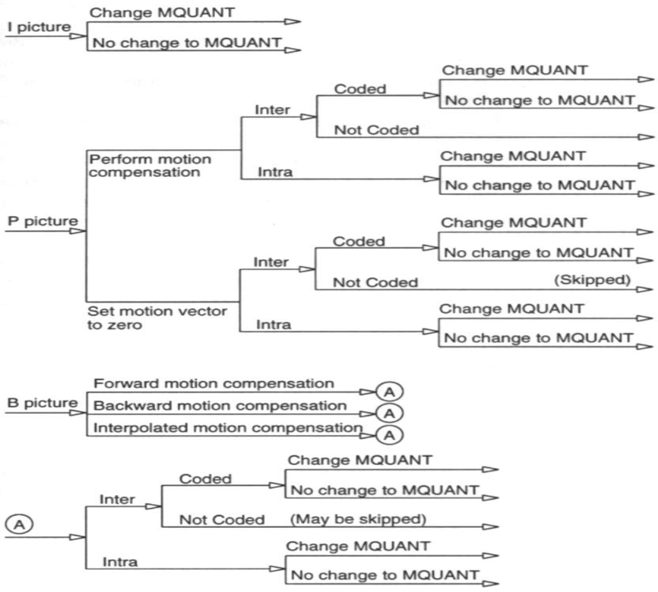


Figure 6.10 Decision trees for coding macroblocks in I-, P-, and B-pictures.

MPEG-1 System Structure

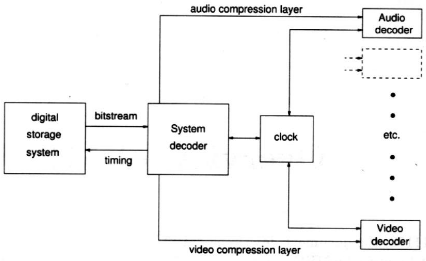


Figure 2.1: MPEG system structure.

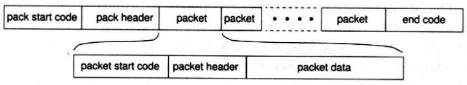
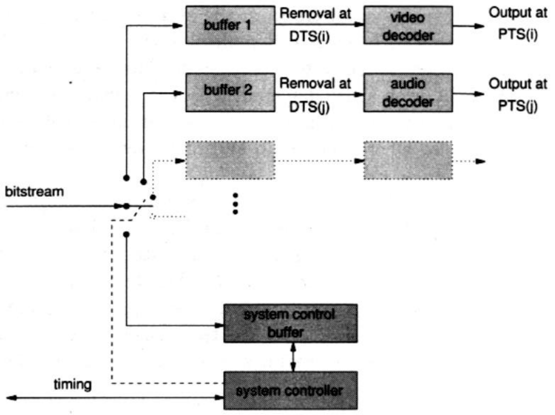


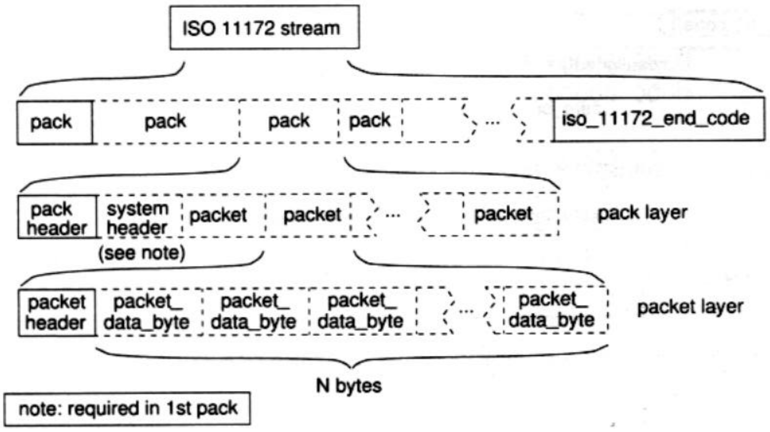
Figure 2.2: System layer pack and packet structure.

System Target Decoder

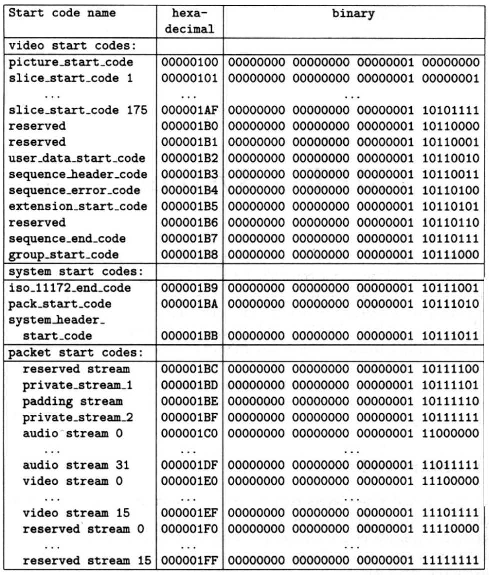
MPEG system users an idealized decoder called STD to interpret the pack and packet headers and deliver the elementary bitstreams to audio and video decoders. DTS: the bits for an access unit are removed from the buffer at decoding time stamp(DTS) in the bitstream.



System Layer Overview of MPEG-1

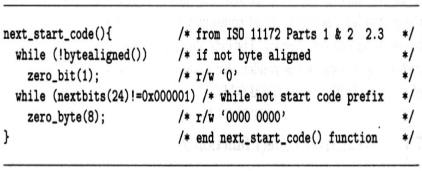


MPEG start code



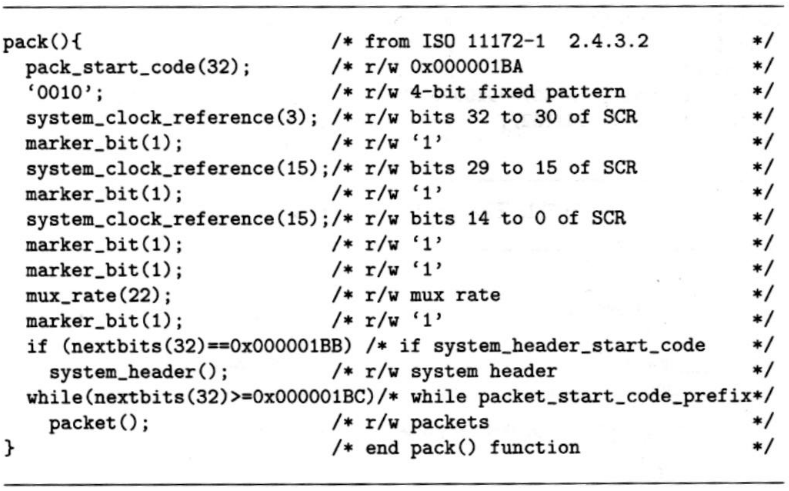
Start code name	hexa- decimal	binary
video start codes:		
picture_start_code	00000100	00000000 00000000 00000001 00000000
slice_start_code 1	00000101	00000000 00000000 00000001 00000001
...	...	...
slice_start_code 175	000001AF	00000000 00000000 00000001 10101111
reserved	000001B0	00000000 00000000 00000001 10110000
reserved	000001B1	00000000 00000000 00000001 10110001
user_data_start_code	000001B2	00000000 00000000 00000001 10110010
sequence_header_code	000001B3	00000000 00000000 00000001 10110011
sequence_error_code	000001B4	00000000 00000000 00000001 10110100
extension_start_code	000001B5	00000000 00000000 00000001 10110101
reserved	000001B6	00000000 00000000 00000001 10110110
sequence_end_code	000001B7	00000000 00000000 00000001 10110111
group_start_code	000001B8	00000000 00000000 00000001 10111000
system start codes:		
iso_11172_end_code	000001B9	00000000 00000000 00000001 10111001
pack_start_code	000001BA	00000000 00000000 00000001 10111010
system_header_ start_code	000001BB	00000000 00000000 00000001 10111011
packet start codes:		
reserved stream	000001BC	00000000 00000000 00000001 10111100
private_stream_1	000001BD	00000000 00000000 00000001 10111101
padding stream	000001BE	00000000 00000000 00000001 10111110
private_stream_2	000001BF	00000000 00000000 00000001 10111111
audio stream 0	000001C0	00000000 00000000 00000001 11000000
...	...	...
audio stream 31	000001DF	00000000 00000000 00000001 11011111
video stream 0	000001E0	00000000 00000000 00000001 11100000
...	...	...
video stream 15	000001EF	00000000 00000000 00000001 11101111
reserved stream 0	000001F0	00000000 00000000 00000001 11110000
...	...	...
reserved stream 15	000001FF	00000000 00000000 00000001 11111111

Table 7.1: MPEG start codes in numeric order.



```
next_start_code(){             /* from ISO 11172 Parts 1 & 2  2.3   */
  while (!bytealigned())       /* if not byte aligned               */
    zero_bit(1);               /* r/w '0'                           */
  while (nextbits(24)!=0x000001) /* while not start code prefix     */
    zero_byte(8);              /* r/w '0000 0000'                   */
}                              /* end next_start_code() function    */
```

Figure 7.1: The next_start_code() function.



```
pack(){                         /* from ISO 11172-1  2.4.3.2         */
  pack_start_code(32);          /* r/w 0x000001BA                    */
  '0010';                       /* r/w 4-bit fixed pattern           */
  system_clock_reference(3);    /* r/w bits 32 to 30 of SCR          */
  marker_bit(1);                /* r/w '1'                           */
  system_clock_reference(15);/* r/w bits 29 to 15 of SCR             */
  marker_bit(1);                /* r/w '1'                           */
  system_clock_reference(15);/* r/w bits 14 to 0 of SCR              */
  marker_bit(1);                /* r/w '1'                           */
  marker_bit(1);                /* r/w '1'                           */
  mux_rate(22);                 /* r/w mux rate                      */
  marker_bit(1);                /* r/w '1'                           */
  if (nextbits(32)==0x000001BB) /* if system_header_start_code       */
    system_header();            /* r/w system header                 */
  while(nextbits(32)>=0x000001BC)/* while packet_start_code_prefix*/
    packet();                   /* r/w packets                       */
}                               /* end pack() function               */
```

Figure 7.6: The pack() function.

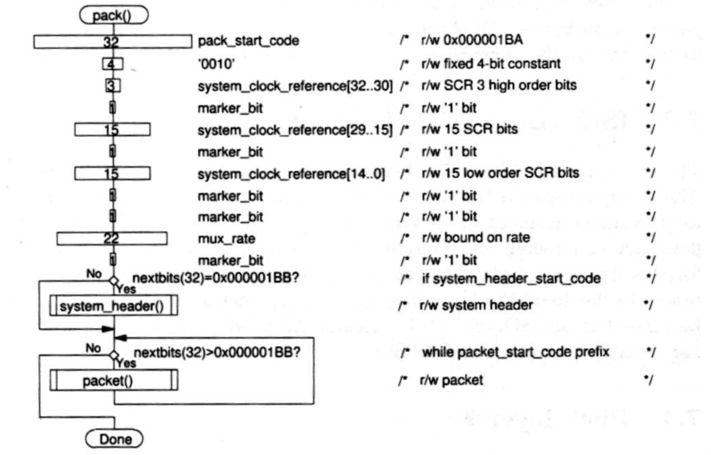

pack()
32 pack_start_code /* r/w 0x000001BA */
4 '0010' /* r/w fixed 4-bit constant */
3 system_clock_reference[32..30] /* r/w SCR 3 high order bits */
1 marker_bit /* r/w '1' bit */
15 system_clock_reference[29..15] /* r/w 15 SCR bits */
1 marker_bit /* r/w '1' bit */
15 system_clock_reference[14..0] /* r/w 15 low order SCR bits */
1 marker_bit /* r/w '1' bit */
1 marker_bit /* r/w '1' bit */
22 mux_rate /* r/w bound on rate */
1 marker_bit /* r/w '1' bit */
No nextbits(32)=0x000001BB? Yes /* if system_header_start_code */
system_header() /* r/w system header */
No nextbits(32)>0x000001BB? Yes /* while packet_start_code prefix */
packet() /* r/w packet */
Done

Figure 7.7: Flowchart for **pack()** function.

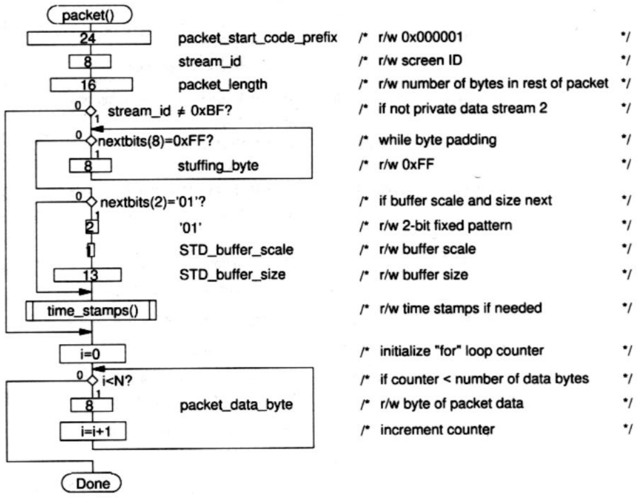

packet()
24 packet_start_code_prefix /* r/w 0x000001 */
8 stream_id /* r/w screen ID */
16 packet_length /* r/w number of bytes in rest of packet */
0 1 stream_id ≠ 0xBF? /* if not private data stream 2 */
0 1 nextbits(8)=0xFF? /* while byte padding */
8 stuffing_byte /* r/w 0xFF */
0 1 nextbits(2)='01'? /* if buffer scale and size next */
2 '01' /* r/w 2-bit fixed pattern */
1 STD_buffer_scale /* r/w buffer scale */
13 STD_buffer_size /* r/w buffer size */
time_stamps() /* r/w time stamps if needed */
i=0 /* initialize "for" loop counter */
0 1 i<N? /* if counter < number of data bytes */
8 packet_data_byte /* r/w byte of packet data */
i=i+1 /* increment counter */
Done

Figure 7.11: Flowchart for **packet()** function.

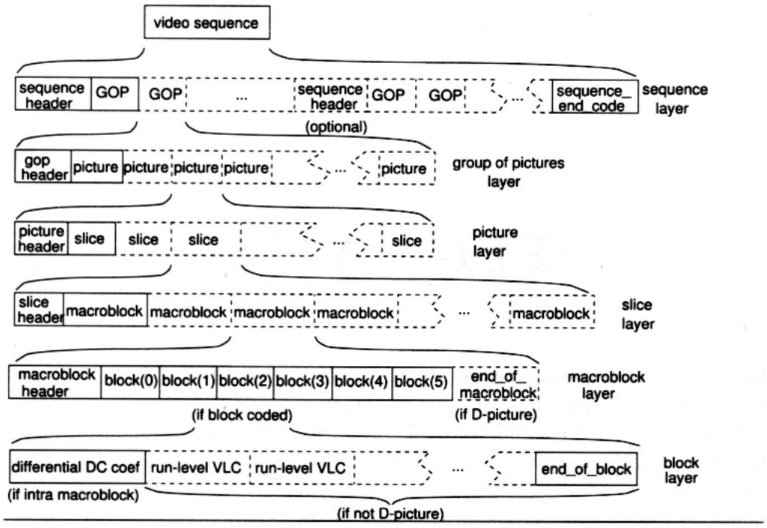
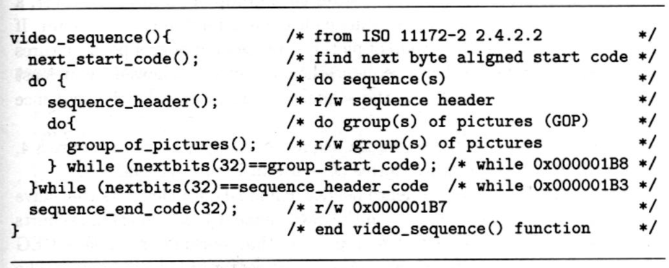


Figure 8.1: The layers of a video stream.



```
video_sequence(){              /* from ISO 11172-2 2.4.2.2          */
  next_start_code();           /* find next byte aligned start code */
  do {                         /* do sequence(s)                    */
    sequence_header();         /* r/w sequence header               */
    do{                        /* do group(s) of pictures (GOP)     */
      group_of_pictures();     /* r/w group(s) of pictures          */
    } while (nextbits(32)==group_start_code); /* while 0x000001B8 */
  }while (nextbits(32)==sequence_header_code  /* while 0x000001B3 */
  sequence_end_code(32);       /* r/w 0x000001B7                    */
}                              /* end video_sequence() function     */
```

Figure 8.2: Video_sequence() function.

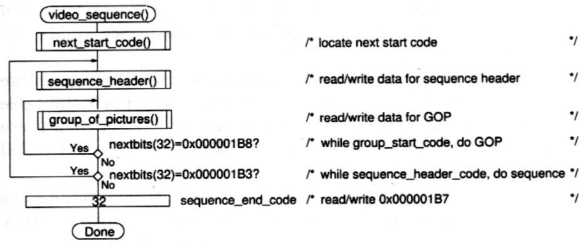
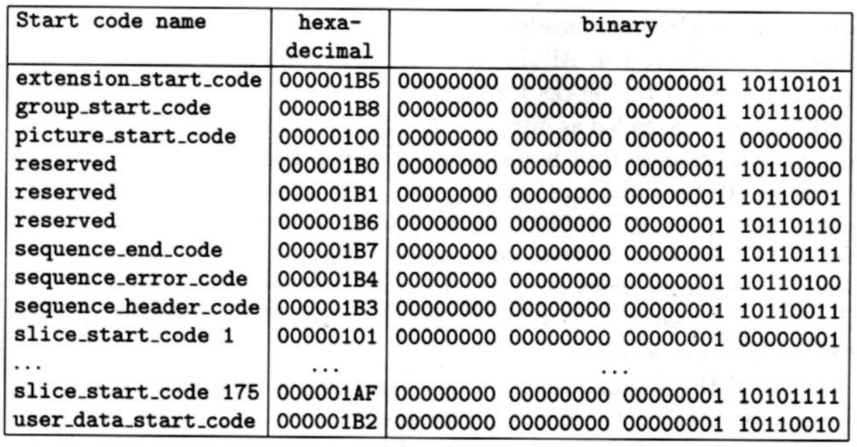
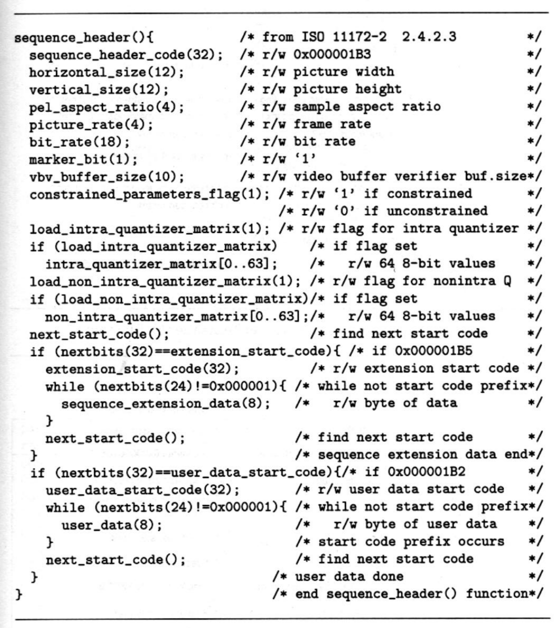

video_sequence()
next_start_code()
/* locate next start code */
sequence_header()
/* read/write data for sequence header */
group_of_pictures()
/* read/write data for GOP */
Yes
nextbits(32)=0x000001B8?
/* while group_start_code, do GOP */
No
Yes
nextbits(32)=0x000001B3?
/* while sequence_header_code, do sequence */
No
32
sequence_end_code
/* read/write 0x000001B7 */
Done

Figure 8.3: Flowchart for the video_sequence() function.



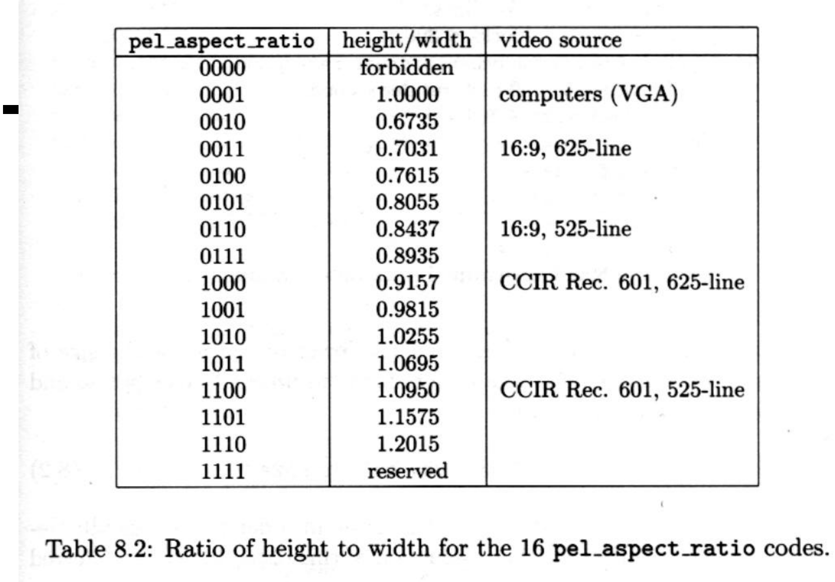
Start code name	hexa- decimal	binary
extension_start_code	000001B5	00000000 00000000 00000001 10110101
group_start_code	000001B8	00000000 00000000 00000001 10111000
picture_start_code	00000100	00000000 00000000 00000001 00000000
reserved	000001B0	00000000 00000000 00000001 10110000
reserved	000001B1	00000000 00000000 00000001 10110001
reserved	000001B6	00000000 00000000 00000001 10110110
sequence_end_code	000001B7	00000000 00000000 00000001 10110111
sequence_error_code	000001B4	00000000 00000000 00000001 10110100
sequence_header_code	000001B3	00000000 00000000 00000001 10110011
slice_start_code 1	00000101	00000000 00000000 00000001 00000001
...	...	...
slice_start_code 175	000001AF	00000000 00000000 00000001 10101111
user_data_start_code	000001B2	00000000 00000000 00000001 10110010

Table 8.1: MPEG video start codes.



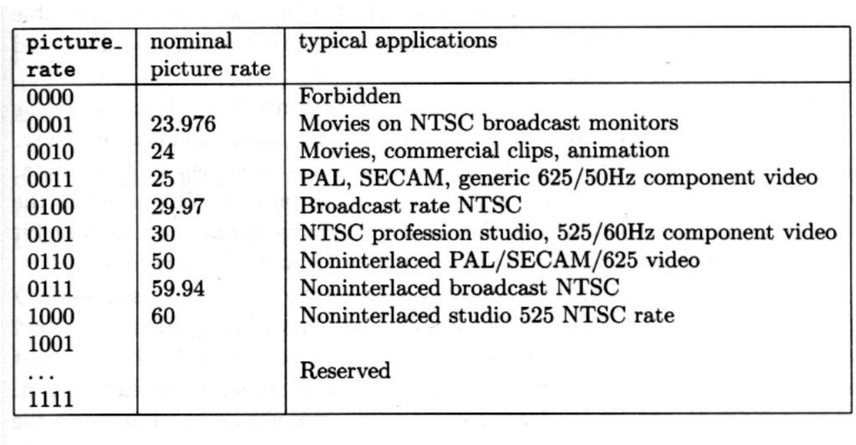
```
sequence_header(){            /* from ISO 11172-2  2.4.2.3          */
  sequence_header_code(32);   /* r/w 0x000001B3                     */
  horizontal_size(12);        /* r/w picture width                  */
  vertical_size(12);          /* r/w picture height                 */
  pel_aspect_ratio(4);        /* r/w sample aspect ratio            */
  picture_rate(4);            /* r/w frame rate                     */
  bit_rate(18);               /* r/w bit rate                       */
  marker_bit(1);              /* r/w ‘1’                            */
  vbv_buffer_size(10);        /* r/w video buffer verifier buf.size*/
  constrained_parameters_flag(1); /* r/w ‘1’ if constrained         */
                                  /* r/w ‘0’ if unconstrained       */
  load_intra_quantizer_matrix(1); /* r/w flag for intra quantizer */
  if (load_intra_quantizer_matrix)    /* if flag set                */
    intra_quantizer_matrix[0..63];    /*   r/w 64 8-bit values      */
  load_non_intra_quantizer_matrix(1); /* r/w flag for nonintra Q    */
  if (load_non_intra_quantizer_matrix)/* if flag set                */
    non_intra_quantizer_matrix[0..63];/*   r/w 64 8-bit values      */
  next_start_code();                  /* find next start code       */
  if (nextbits(32)==extension_start_code){ /* if 0x000001B5         */
    extension_start_code(32);         /* r/w extension start code */
    while (nextbits(24)!=0x000001){ /* while not start code prefix*/
      sequence_extension_data(8);   /*   r/w byte of data           */
    }
    next_start_code();              /* find next start code         */
  }                                 /* sequence extension data end*/
  if (nextbits(32)==user_data_start_code){/* if 0x000001B2          */
    user_data_start_code(32);       /* r/w user data start code     */
    while (nextbits(24)!=0x000001){ /* while not start code prefix*/
      user_data(8);                 /*   r/w byte of user data      */
    }                               /* start code prefix occurs     */
    next_start_code();              /* find next start code         */
  }                               /* user data done                 */
}                                 /* end sequence_header() function*/
```

Figure 8.4: The sequence_header() function.

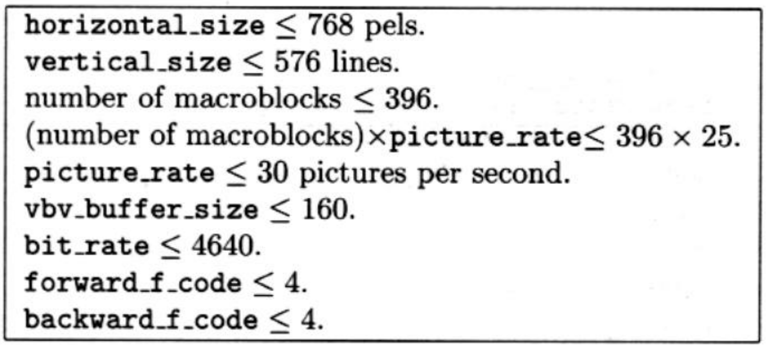


pel_aspect_ratio	height/width	video source
0000	forbidden	
0001	1.0000	computers (VGA)
0010	0.6735	
0011	0.7031	16:9, 625-line
0100	0.7615	
0101	0.8055	
0110	0.8437	16:9, 525-line
0111	0.8935	
1000	0.9157	CCIR Rec. 601, 625-line
1001	0.9815	
1010	1.0255	
1011	1.0695	
1100	1.0950	CCIR Rec. 601, 525-line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

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

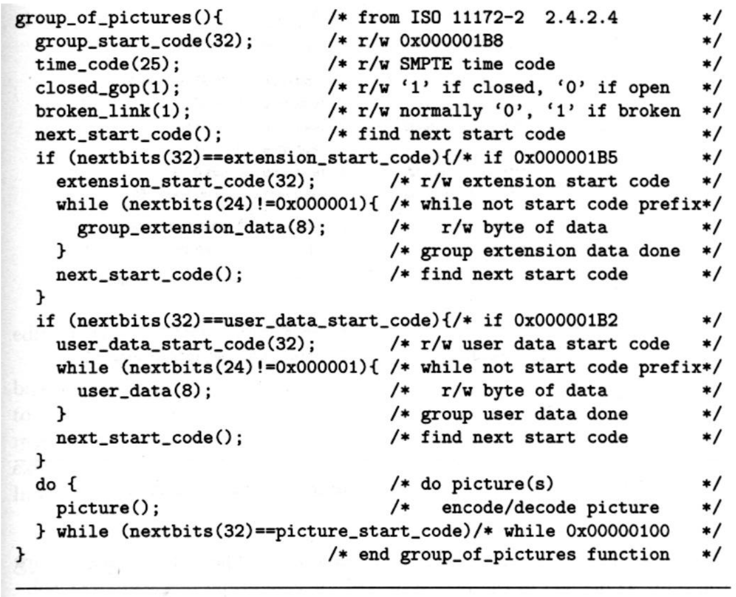


picture_ rate	nominal picture rate	typical applications
0000		Forbidden
0001	23.976	Movies on NTSC broadcast monitors
0010	24	Movies, commercial clips, animation
0011	25	PAL, SECAM, generic 625/50Hz component video
0100	29.97	Broadcast rate NTSC
0101	30	NTSC profession studio, 525/60Hz component video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



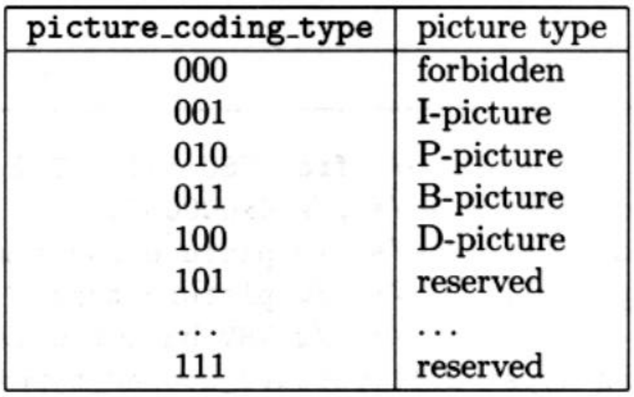
horizontal_size $\leq$ 768 pels.
vertical_size $\leq$ 576 lines.
number of macroblocks $\leq$ 396.
(number of macroblocks)$\times$**picture_rate**$\leq 396 \times 25$.
picture_rate $\leq$ 30 pictures per second.
vbv_buffer_size $\leq$ 160.
bit_rate $\leq$ 4640.
forward_f_code $\leq$ 4.
backward_f_code $\leq$ 4.

Table 8.4: Constrained parameters bounds.



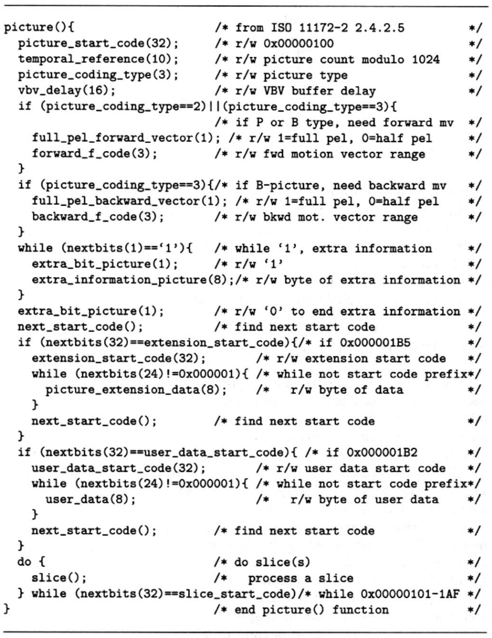
```
group_of_pictures(){          /* from ISO 11172-2  2.4.2.4        */
  group_start_code(32);       /* r/w 0x000001B8                   */
  time_code(25);              /* r/w SMPTE time code              */
  closed_gop(1);              /* r/w '1' if closed, '0' if open   */
  broken_link(1);             /* r/w normally '0', '1' if broken  */
  next_start_code();          /* find next start code             */
  if (nextbits(32)==extension_start_code){/* if 0x000001B5        */
    extension_start_code(32);       /* r/w extension start code   */
    while (nextbits(24)!=0x000001){ /* while not start code prefix*/
      group_extension_data(8);      /*   r/w byte of data         */
    }                               /* group extension data done  */
    next_start_code();              /* find next start code       */
  }
  if (nextbits(32)==user_data_start_code){/* if 0x000001B2        */
    user_data_start_code(32);       /* r/w user data start code   */
    while (nextbits(24)!=0x000001){ /* while not start code prefix*/
      user_data(8);                 /*   r/w byte of data         */
    }                               /* group user data done       */
    next_start_code();              /* find next start code       */
  }
  do {                              /* do picture(s)              */
    picture();                      /*   encode/decode picture    */
  } while (nextbits(32)==picture_start_code)/* while 0x00000100   */
}                             /* end group_of_pictures function   */
```

Figure 8.7: The **group_of_pictures()** function.



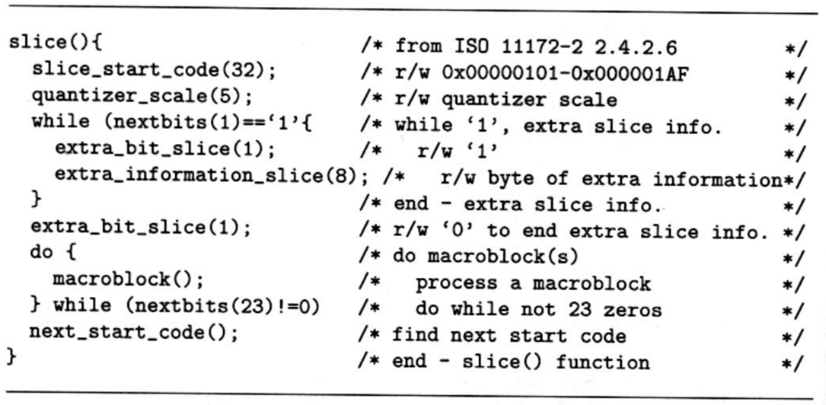
picture_coding_type	picture type
000	forbidden
001	I-picture
010	P-picture
011	B-picture
100	D-picture
101	reserved
...	...
111	reserved

Table 8.5: Picture type codes.



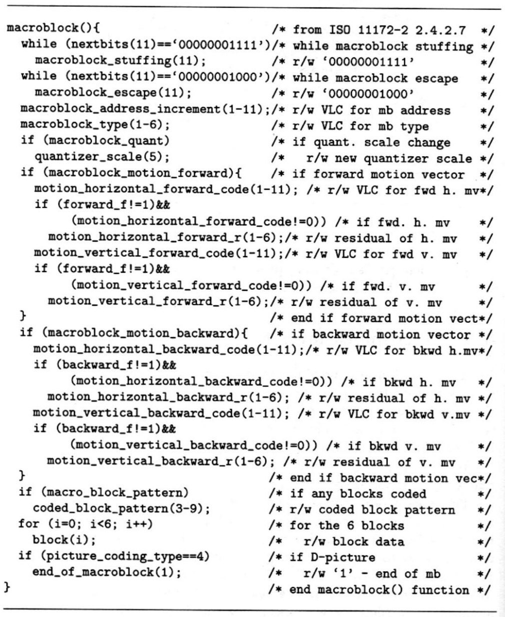
```
picture(){                      /* from ISO 11172-2 2.4.2.5         */
  picture_start_code(32);       /* r/w 0x00000100                   */
  temporal_reference(10);       /* r/w picture count modulo 1024    */
  picture_coding_type(3);       /* r/w picture type                 */
  vbv_delay(16);                /* r/w VBV buffer delay             */
  if (picture_coding_type==2)||(picture_coding_type==3){
                                /* if P or B type, need forward mv  */
    full_pel_forward_vector(1); /* r/w 1=full pel, 0=half pel       */
    forward_f_code(3);          /* r/w fwd motion vector range      */
  }
  if (picture_coding_type==3){/* if B-picture, need backward mv     */
    full_pel_backward_vector(1); /* r/w 1=full pel, 0=half pel      */
    backward_f_code(3);         /* r/w bkwd mot. vector range       */
  }
  while (nextbits(1)=='1'){     /* while '1', extra information     */
    extra_bit_picture(1);       /* r/w '1'                          */
    extra_information_picture(8);/* r/w byte of extra information */
  }
  extra_bit_picture(1);         /* r/w '0' to end extra information */
  next_start_code();            /* find next start code             */
  if (nextbits(32)==extension_start_code){/* if 0x000001B5          */
    extension_start_code(32);         /* r/w extension start code   */
    while (nextbits(24)!=0x000001){ /* while not start code prefix*/
      picture_extension_data(8);    /*   r/w byte of data           */
    }
    next_start_code();          /* find next start code             */
  }
  if (nextbits(32)==user_data_start_code){ /* if 0x000001B2         */
    user_data_start_code(32);         /* r/w user data start code   */
    while (nextbits(24)!=0x000001){ /* while not start code prefix*/
      user_data(8);                 /*   r/w byte of user data      */
    }
    next_start_code();          /* find next start code             */
  }
  do {                          /* do slice(s)                      */
    slice();                    /*   process a slice                */
  } while (nextbits(32)==slice_start_code)/* while 0x00000101-1AF */
}                               /* end picture() function           */
```

Figure 8.9: The picture() function.



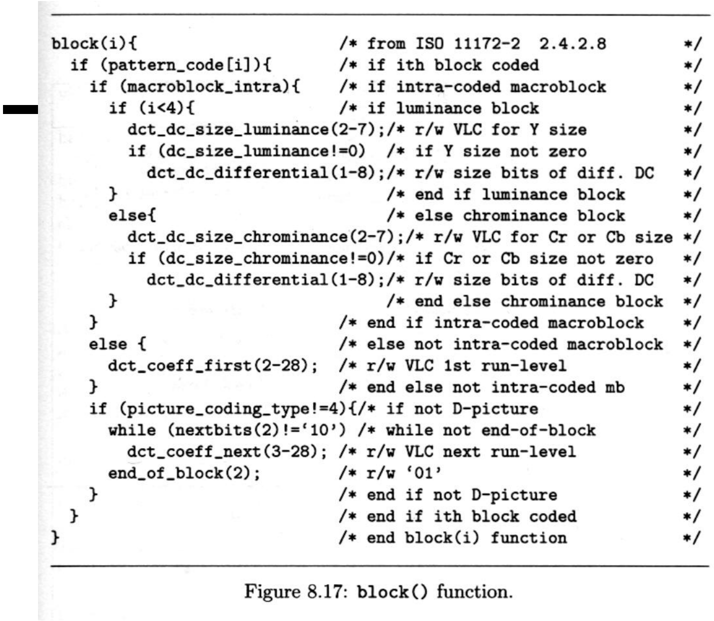
```
slice(){                        /* from ISO 11172-2 2.4.2.6         */
  slice_start_code(32);         /* r/w 0x00000101-0x000001AF        */
  quantizer_scale(5);           /* r/w quantizer scale              */
  while (nextbits(1)=='1'{      /* while '1', extra slice info.     */
    extra_bit_slice(1);         /*   r/w '1'                        */
    extra_information_slice(8); /*   r/w byte of extra information*/
  }                             /* end - extra slice info.          */
  extra_bit_slice(1);           /* r/w '0' to end extra slice info. */
  do {                          /* do macroblock(s)                 */
    macroblock();               /*   process a macroblock           */
  } while (nextbits(23)!=0)     /*   do while not 23 zeros          */
  next_start_code();            /* find next start code             */
}                               /* end - slice() function           */
```

Figure 8.11: The slice() function.



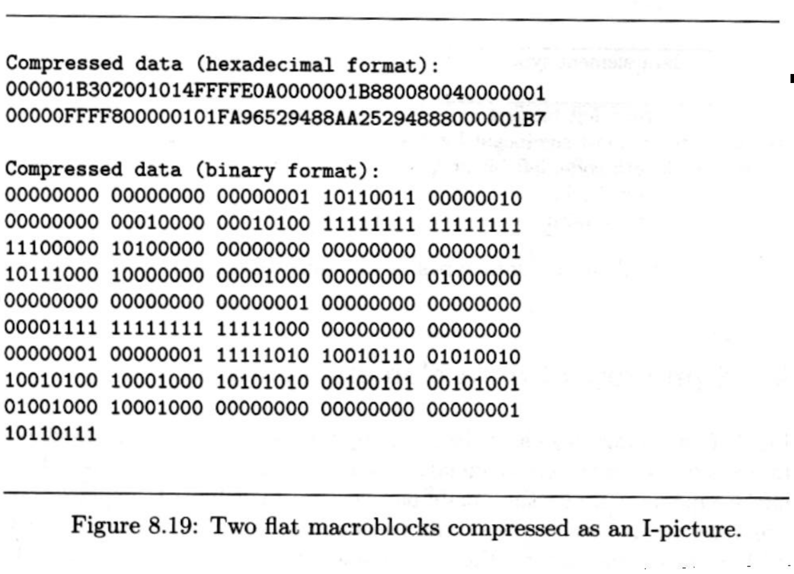
```
macroblock(){                               /* from ISO 11172-2 2.4.2.7  */
  while (nextbits(11)=='00000001111')/* while macroblock stuffing */
    macroblock_stuffing(11);                /* r/w '00000001111'         */
  while (nextbits(11)=='00000001000')/* while macroblock escape   */
    macroblock_escape(11);                  /* r/w '00000001000'         */
  macroblock_address_increment(1-11);/* r/w VLC for mb address    */
  macroblock_type(1-6);                     /* r/w VLC for mb type       */
  if (macroblock_quant)                     /* if quant. scale change    */
    quantizer_scale(5);                     /*   r/w new quantizer scale */
  if (macroblock_motion_forward){           /* if forward motion vector  */
    motion_horizontal_forward_code(1-11); /* r/w VLC for fwd h. mv*/
    if (forward_f!=1)&&
         (motion_horizontal_forward_code!=0)) /* if fwd. h. mv      */
      motion_horizontal_forward_r(1-6);/* r/w residual of h. mv   */
    motion_vertical_forward_code(1-11);/* r/w VLC for fwd v. mv   */
    if (forward_f!=1)&&
         (motion_vertical_forward_code!=0)) /* if fwd. v. mv        */
      motion_vertical_forward_r(1-6);/* r/w residual of v. mv     */
  }                                         /* end if forward motion vect*/
  if (macroblock_motion_backward){          /* if backward motion vector */
    motion_horizontal_backward_code(1-11);/* r/w VLC for bkwd h.mv*/
    if (backward_f!=1)&&
         (motion_horizontal_backward_code!=0)) /* if bkwd h. mv     */
      motion_horizontal_backward_r(1-6); /* r/w residual of h. mv */
    motion_vertical_backward_code(1-11); /* r/w VLC for bkwd v.mv */
    if (backward_f!=1)&&
         (motion_vertical_backward_code!=0)) /* if bkwd v. mv       */
      motion_vertical_backward_r(1-6); /* r/w residual of v. mv   */
  }                                         /* end if backward motion vec*/
  if (macro_block_pattern)                  /* if any blocks coded       */
    coded_block_pattern(3-9);               /* r/w coded block pattern   */
  for (i=0; i<6; i++)                       /* for the 6 blocks          */
    block(i);                               /*   r/w block data          */
  if (picture_coding_type==4)               /* if D-picture              */
    end_of_macroblock(1);                   /*   r/w '1' - end of mb     */
}                                           /* end macroblock() function */
```

Figure 8.13: The macroblock() function.



```
block(i){                       /* from ISO 11172-2  2.4.2.8        */
  if (pattern_code[i]){         /* if ith block coded               */
    if (macroblock_intra){      /* if intra-coded macroblock        */
      if (i<4){                 /* if luminance block               */
        dct_dc_size_luminance(2-7);/* r/w VLC for Y size            */
        if (dc_size_luminance!=0)  /* if Y size not zero            */
          dct_dc_differential(1-8);/* r/w size bits of diff. DC     */
      }                            /* end if luminance block        */
      else{                        /* else chrominance block        */
        dct_dc_size_chrominance(2-7);/* r/w VLC for Cr or Cb size */
        if (dc_size_chrominance!=0)/* if Cr or Cb size not zero     */
          dct_dc_differential(1-8);/* r/w size bits of diff. DC     */
      }                            /* end else chrominance block    */
    }                           /* end if intra-coded macroblock    */
    else {                      /* else not intra-coded macroblock  */
      dct_coeff_first(2-28);    /* r/w VLC 1st run-level            */
    }                           /* end else not intra-coded mb      */
    if (picture_coding_type!=4){/* if not D-picture                 */
      while (nextbits(2)!='10') /* while not end-of-block           */
        dct_coeff_next(3-28);   /* r/w VLC next run-level           */
      end_of_block(2);          /* r/w '01'                         */
    }                           /* end if not D-picture             */
  }                             /* end if ith block coded           */
}                               /* end block(i) function            */
```

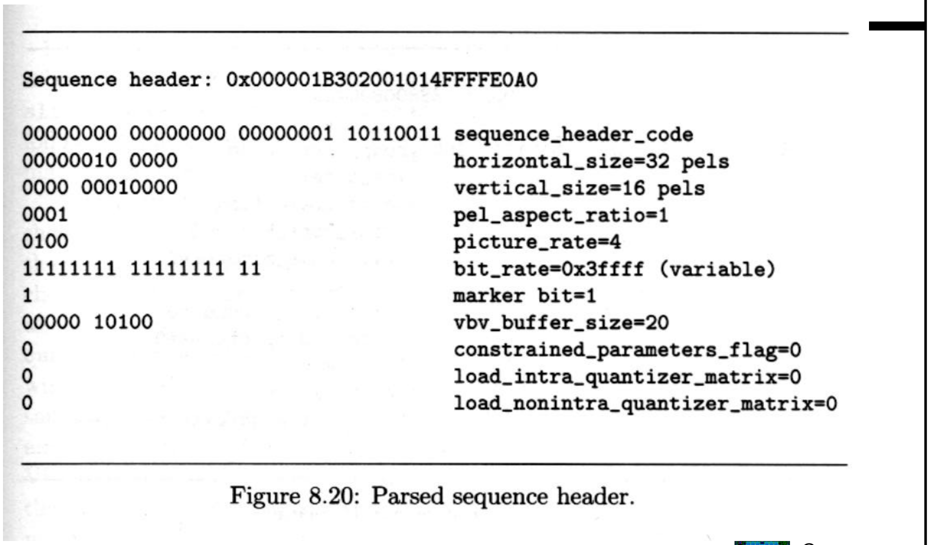
Figure 8.17: `block()` function.



```
Compressed data (hexadecimal format):
000001B302001014FFFFE0A0000001B880080040000001
00000FFFF800000101FA96529488AA25294888000001B7

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

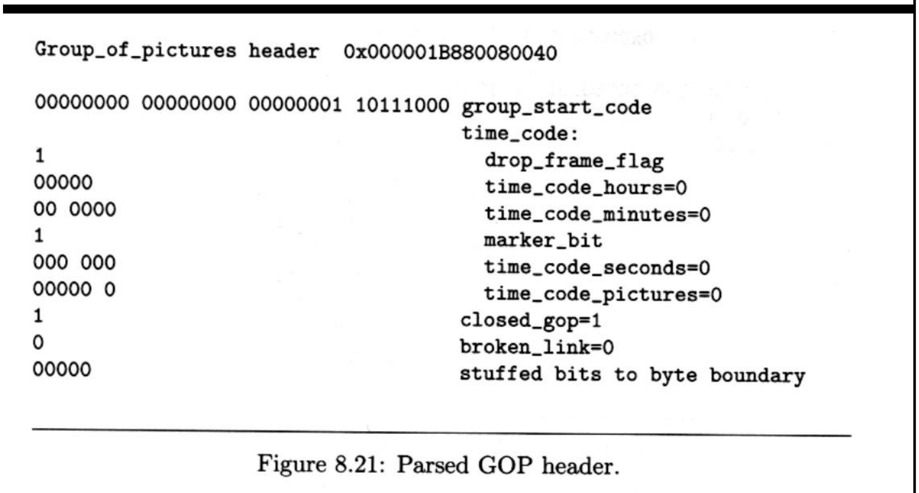
Figure 8.19: Two flat macroblocks compressed as an I-picture.



```
Sequence header: 0x000001B302001014FFFFE0A0

00000000 00000000 00000001 10110011 sequence_header_code
00000010 0000                       horizontal_size=32 pels
0000 00010000                       vertical_size=16 pels
0001                                pel_aspect_ratio=1
0100                                picture_rate=4
11111111 11111111 11                bit_rate=0x3ffff (variable)
1                                   marker bit=1
00000 10100                         vbv_buffer_size=20
0                                   constrained_parameters_flag=0
0                                   load_intra_quantizer_matrix=0
0                                   load_nonintra_quantizer_matrix=0
```

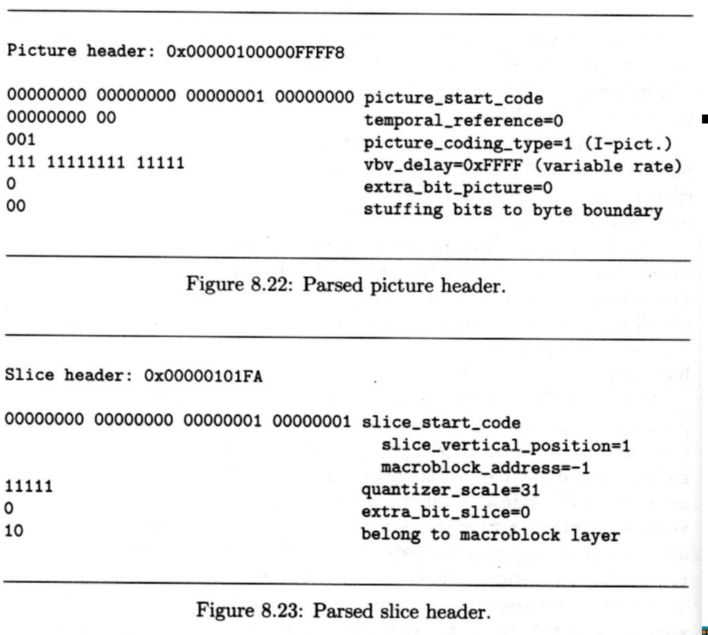
Figure 8.20: Parsed sequence header.



```
Group_of_pictures header  0x000001B880080040

00000000 00000000 00000001 10111000 group_start_code
                                    time_code:
1                                     drop_frame_flag
00000                                 time_code_hours=0
00 0000                               time_code_minutes=0
1                                     marker_bit
000 000                               time_code_seconds=0
00000 0                               time_code_pictures=0
1                                   closed_gop=1
0                                   broken_link=0
00000                               stuffed bits to byte boundary
```

Figure 8.21: Parsed GOP header.



```
Picture header: 0x00000100000FFFF8

00000000 00000000 00000001 00000000 picture_start_code
00000000 00                         temporal_reference=0
001                                 picture_coding_type=1 (I-pict.)
111 11111111 11111                  vbv_delay=0xFFFF (variable rate)
0                                   extra_bit_picture=0
00                                  stuffing bits to byte boundary
```

Figure 8.22: Parsed picture header.

```
Slice header: 0x00000101FA

00000000 00000000 00000001 00000001 slice_start_code
                                      slice_vertical_position=1
                                      macroblock_address=-1
11111                               quantizer_scale=31
0                                   extra_bit_slice=0
10                                  belong to macroblock layer
```

Figure 8.23: Parsed slice header.