CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Faculty of Computer Science

Chair for Real Time Systems

Seminar Paper

Implementation of a basic H.264/AVC Decoder

Author: Martin Fiedler

Supervisor: Dr. Robert Baumgartl

Date of Submission: June 1, 2004

Contents

1. Introduction 3

2. Reference Implementations 4

3. H.264 Outline 5

3.1. NAL units, slices, fields and frames . . . 5

3.2. Motion Compensation . . . 6

3.3. Macroblock Layer . . . 6

3.4. Block Transformation and Encoding . . . 7

3.5. Advanced Coding Tools . . . 8

4. Implementation Details 10

4.1. Implemented Features . . . 10

4.2. Basic structure . . . 10

4.3. The input system . . . 13

4.4. Parsing basic headers . . . 15

4.5. Mode Prediction . . . 17

4.6. Slice Reconstruction . . . 19

4.7. Intra and Inter Prediction . . . 21

4.8. Parsing and Rendering of Residuals . . . 23

4.9. The DSP port . . . 23

5. Performance 25

5.1. Identifying Hot Spots . . . 25

5.2. Memory usage . . . 26

6. Limitations and Future Work 27

A. References 28

1. Introduction

The emerging high-efficient H.264, or MPEG-4 Part 10: Advanced Video Coding (AVC) video compression standard is on its way to become a general replacement for the rather old MPEG-2 standard. The High Definition DVD standard (HD-DVD) contains H.264 as one of the three mandatory video formats, next to MPEG-2 and Windows Media Video 9 (WMV9), which is a H.264 variant. Another very close variant, Sorenson Video 3 (SVQ3) is already the defacto standard for the distribution of movie trailers over the Internet. The Digital Video Broadcasting (DVB) family of digital video infrastructure standards is also adopting H.264 for usage in the new DVB-Handheld (DVB-H) sub-standard which provides low-resolution, low-bandwidth video broadcasting for mobile devices.

The implementation presented here is a written-from-scratch plain C version of a minimal H.264 decoder that is able to decode but the simplest data streams. Despite these limitations, the (theoretical) compression efficiency of the small implemented subset of H.264's features is already very close to MPEG-4 Advanced Simple Profile (AVC), whose most popular implementations are DivX and XviD.

The decoder core is designed to be platform independent, but two specific platforms will be discussed in detail: First, a normal x86 personal computer as the primary development platform, and second, the Texas Instruments TMS320C64xx series of Digital Signal Processors (DSP).

This document describes the inner workings of this specific implementation, but it can also be read as a generic description of H.264's basic concepts. It tries to describe the algorithm at an intermediate level, slightly more detailed than a pure H.264 feature description, but not as over-structured and pedantic as the official standard paper. It is *not* a replacement for a full-fledged specification as detailed information such as precise syntax specifications and code tables are out of the scope of this document.

2. Reference Implementations

The main source of information used in the implementation is the Joint Video Team (JVT) Working Document JVT-G050r1 [1], which seems to be the last freely available official H.264 draft. The final ITU-T recommendation is not free of charge, but there are presumably only negligible differences, if any. The document and the enclosed example source code [5, version JM7.2 was used] are very detailed and complete, but also hard to read and understand.

The example encoder was compiled and used to encode test streams during development. Most of the debugging was done by analyzing the trace files produced by this encoder and decoder, or by adding additional trace file output to the decoder.

Other source code can be found in the open-source library libavcodec [6]. The H.264 and SVQ3 codec module `h264.c` is written and maintained by Michael Niedermayer. Most functions of the codec are concentrated in this single file. By the time the implementation discussed in this paper was started, libavcodec's H.264 decoder also had a similarly limited set of features. Because of this and its integration into the media player application MPlayer [7], Niedermayer's decoder was chosen as the main reference.

No source code was copied, directly or indirectly, from these reference implementations.

3. H.264 Outline

H.264 is a block-based, motion-compensated video compression method. It is designed to be scalable, that is, its efficiency is roughly equally high for all purposes from low-bandwidth streaming up to high definition broadcast and storage.

The following chapter explains the basic outline of H.264.

3.1. NAL units, slices, fields and frames

A H.264 video stream is organized in discrete packets, called "**NAL units**" (Network Abstraction Layer units). Each of these packets can contain a part of a **slice**, that is, there may be one or more NAL units per slice. But not all NAL units contain slice data, there are also NAL unit types for other purposes, such as signalling, headers and additional data.

The slices, in turn, contain a part of a video **frame**. In normal bitstreams, each frame consists of a single slice whose data is stored in a single NAL unit. Nevertheless, the possibility to spread frames over an almost arbitrary number of NAL units can be useful if the stream is transmitted over an error-prone medium: The decoder may resynchronize after each NAL unit instead of skipping a whole frame if a single error occurs.

H.264 also supports optional *interlaced* encoding. In this encoding mode, a frame is split into two **fields**. Fields may be encoded using spacial or temporal interleaving.

To encode color images, H.264 uses the YCbCr color space like its predecessors, separating the image into luminance (or "luma", brightness) and chrominance (or "chroma", color) planes. It is, however, fixed at 4:2:0 subsampling, i.e. the chroma channels each have half the resolution of the luma channel.

Slice types

H.264 defines five different slice types: I, P, B, SI and SP.

I slices or "Intra" slices describe a full still image, containing only references to itself. A video stream may consist only of I slices, but this is typically not used. However, the first frame of a sequence always needs to be built out of I slices.

P slices or "Predicted" slices use one or more recently decoded slices as a reference (or "prediction") for picture construction. The prediction is usually not exactly the same as the actual picture content, so a "residual" may be added.

B slices or "Bi-Directional Predicted" slices work like P slices with the exception that former *and future* I or P slices (in playback order) may be used as reference pictures. For this to work, B slices must be decoded *after* the following I or P slice.

SI and SP slices or “Switching” slices may be used for transitions between two different H.264 video streams. This is a very uncommon feature.

3.2. Motion Compensation

Since MPEG-1, motion compensation is a standard coding tool for video compression. Using motion compensation, motion between frames can be encoded in a very efficient manner.

A typical P-type block copies an area of the last decoded frame into the current frame buffer to serve as a prediction. If this block is assigned a nonzero **motion vector**, the source area for this copy process will not be the same as the destination area. It will be moved by some pixels, allowing to accomodate for the motion of the object that occupies that block.

Motion vectors need not be integer values: In H.264, motion vector precision is one-quarter pixel (one-eighth pixel in chroma). Interpolation is used to determine the intensity values at non-integer pixel positions. Additionally, motion vectors may point to regions outside of the image. In this case, edge pixels are repeated.

Motion vector prediction

Because adjacent blocks tend to move in the same directions, the motion vectors are also encoded using prediction. When a block’s motion vector is encoded, the surrounding blocks’ motion vectors are used to estimate the current motion vector. Then, only the difference between this prediction and the actual vector is stored.

3.3. Macroblock Layer

Each slice consists of **macroblocks** (or, when using interlaced encoding, macroblock pairs) of 16x16 pixels. The encoder may choose between a multitude of encoding modes for each macroblock.

3.3.1. Macroblock modes for I slices

In H.264, I slices also use a prediction/residual scheme: Already decoded macroblocks of the same frame may be used as references for this so-called “intra prediction” process. The macroblock mode indicates which of two possible prediction types is used:

Intra_16x16 uses one intra prediction scheme for the whole macroblock. Pixels may be filled from surrounding macroblocks at the left and the upper edge using one of four possible prediction modes.
Intra prediction is also performed for the chroma planes using the same range of prediction modes. However, different modes may be selected for luma and chroma.

Intra_4x4 subdivides the macroblock into 16 subblocks and assigns one of nine prediction modes to each of these 4x4 blocks. The prediction modes offered in Intra_4x4 blocks support gradients or other smooth structures that run in one of eight distinct directions. One additional mode fills a whole subblock with a single value and is used if no other mode fits the actual pixel data inside a block.

Intra chroma prediction is performed in an identical way to the one used in Intra_16x16 prediction. Thus, the chroma predictions are *not* split into subblocks as it is done for luma.

Additionally, a third intra macroblock type exists: **I_PCM** macroblocks contain raw uncompressed pixel data for the luma and chroma planes. This macroblock type is normally not used.

3.3.2. Macroblock modes for P and B slices

P and B slices use another range of macroblock modes, but the encoder may use intra coded macroblocks in P or B slices as well.

In regions of uniform motion without texture changes, macroblocks may be **skipped**. In this case, no further data is stored for the macroblock, but it is motion-compensated using the predicted motion vector.

Otherwise, the macroblock of 16x16 pixels is divided into **macroblock partitions**:

- It may be stored as a single partition of 16x16 pixels.
- It may be split horizontally into two partitions of 16x8 pixels each.
- It may be split vertically into two partitions of 8x16 pixels each.
- It may be split in both directions, resulting in four **sub-macroblock partitions** of 8x8 pixels. Each of these may be split similarly into partitions of 8x8, 8x4, 4x8 or 4x4 pixels. Not all sub-macroblock partitions need to be split in the same manner, thus allowing for any number of partitions from 1 to 16.

Each macroblock or sub-macroblock partition stores its own motion vector, thus allowing for motion isolation of objects as small as 4x4 pixels. Additionally, a reference frame number that is used for prediction is stored for each partition.

3.4. Block Transformation and Encoding

The basic image encoding algorithm of H.264 uses a separable transformation. The mode of operation is similar to that of JPEG and MPEG, but the transformation used is not an 8x8 DCT, but an 4x4 integer transformation derived from the DCT. This transformation is very simple and fast; it can be computed using only additions/subtractions and binary shifts. It decomposes the image into its spacial frequency components like the DCT, but due to its smaller size, it is not as prone to high frequency “mosquito” artifacts as its predecessors.

An image block B is transformed to B' using the following formula. The necessary post-scaling step is integrated into quantization (see below) and therefore omitted:

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix}$$

$$B' = MBM^T$$

The basic functionality of the H.264 image transformation process is as follows: For each block, the actual image data is subtracted from the prediction. The resulting **residual** is transformed. The coefficients of this transform are divided by a constant integer number. This procedure is called **quantization**; it is the only step in the whole encoding process that is actually lossy. The divisor used is called the **quantization parameter**; different quantization parameters are used for luma and chroma channels.

The quantized coefficients are then read out from the 4x4 coefficient matrix into a single 16-element **scan**. This scan is then encoded using sophisticated (lossless) entropy coding. In the decoder, these steps are performed in reversed order.

3.4.1. Entropy encoding modes

H.264 supports two different methods for the final entropy encoding step: **CAVLC**, or Context-Adaptive Variable Length Coding, is the standard method using simple variable length huffmann-like codes and codebooks. **CABAC**, or Context-Adaptive Binary Arithmetic Coding, on the other side, is an optional, highly efficient binary encoding scheme.

3.4.2. DC transformation

The upper left transform coefficient (i.e. the first coefficient in scan order) is treated separately for Intra_16x16 macroblocks and chroma residuals. Because this coefficient indicates the average intensity value of a block, correlations between the DC values of adjacent equally predicted blocks can be exploited this way.

For Intra_16x16 macroblocks, the DC coefficients are transformed using a separable 4x4 Hadamard transform with the following matrix:

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix}$$

Chroma residuals are always transformed in one group per macroblock. Thus, there are 4 chroma blocks per macroblock and channel. A separable 2x2 transform is used:

$$M = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

3.5. Advanced Coding Tools

Beyond the basic structure described above, H.264 offers additional features to improve coding efficiency.

3.5.1. In-Loop Deblocking Filter

After each individial frame has been decoded, a deblocking filter that reduces the most visible blocking artifacts is applied. This has been available since MPEG-4 Simple Profile as an optional post-processing operation, but in H.264, it is closely integrated into the encoding and decoding process: The already deblocked frames are used as reference frames by the following P or B slices. This technique circumvents noticeable blocking artifacts as far as possible.

3.5.2. Advanced prediction

H.264 may not only use interframe references to the last decoded frame, but to a (theoretically) arbitrary number of frames. This greatly improves the encoding efficiency of periodic movements (**long-term prediction**). Moreover, multiple predictions may be mixed by arbitrary ratios (**weighted prediction**).

3.5.3. Arbitrary Slice Ordering

Since the slices of a picture can be decoded independently (except for the slice edges which can only be approximated), slices need not be decoded in the correct order to render an image in acceptable quality. This is useful e.g. for UDP streaming where packets may be delivered out-of-order.

3.5.4. Flexible Macroblock Ordering

The macroblocks inside a slice may be encoded in any order. This can be used to increase robustness against transmission errors, for example. It is also reminiscent of MPEG-4's “video object planes” system which could be used to encode each object of a scene individually. In normal video streams, however, this feature is not used, and macroblocks are sent in the normal scanline order.

4. Implementation Details

4.1. Implemented Features

The implementation presented here only handles a minimal feature set. This means that the following features are **not** implemented:

- CABAC.
- Interlacing.
- Any kind of data partitioning.
- Any kind of arbitrary slice ordering (ASO) or flexible macroblock ordering (FMO).
- B and switching slices.
- Any kind of long-term prediction (i.e. only the last frame may be used as reference frame).
- In-Loop deblocking filtering.
- Non-constrained intra prediction.

Although this feature set is very limited, the coding efficiency is already close to that of MPEG-4 Advanced Simple Profile (without postprocessing filters). Moreover, there is no limit to the size of a video frame.

4.2. Basic structure

The decoder implementation is written in plain C and compiles at least with GCC versions 2.95 and 3.3 as well as the Texas Instruments C Compiler for TMS320C6xxx CPUs. Other compilers are very likely to work, too.

The core decoder makes no assumptions about the platform used; any 16 or 32-bit target platform and any operating system should work. File operations and memory management is accomplished with standard C library functions, such as `fopen`/`fread`, `malloc`/`free` or `memset`. However, there are exceptions: The stand-alone player application `playh264` requires the external library libSDL (Simple DirectMedia Library) for its graphical output. The performance counter library is written for Pentium-compatible processors, but it can easily be disabled or ported.

4.2.1. The modules

The decoder consists of the following modules:

Module	**Purpose**
`block`	Handles coefficient transformation and block entering at a high level.
`cavlc`	Contains functions for parsing Exp-Golomb and arbitrary codes.
`cavlc_tables`	Contains pre-defined code tables for various syntax elements.
`common`	Contains common data types, macros and functions, e.g. for frame handling.
`coretrans`	Performs coefficient transformation and block entering at a low level.
`in_file`	The file input handler.
`input`	Contains functions for bitwise input.
`intra_pred`	Implements the various Intra Prediction Modes.
`main`	The libh264 frontend functions (see below).
`mbmodes`	Contains definitions, macros and functions for parsing (sub-)macroblock modes.
`mocomp`	Performs motion compensation.
`mode_pred`	Contains definitions, macros and functions for prediction e.g. of motion vectors.
`nal`	Contains a simple NAL unit parser for H.264 Annex B elementary streams.
`params`	Defines structures for picture and sequence parameter sets and parses them.
`perf`	Contains a simple performance counter library for x86 targets.
`playh264`	A standalone player application.
`residual`	Parses CAVLC-coded coefficient transform levels.
`slice`	Decodes a slice. This is where most of the work is done.
`slicehdr`	Defines structures for slice headers and parses them.

4.2.2. The libh264 library

By default, a `make all` compiles the core decoder modules into a static library file `libh264.a`. This library's interface consists of just five functions. Decoding a H.264 stream is therefore a simple as

```
#include <h264.h>

void main() {
  frame *f;
  if( ! h264_open( /* file name */ )) {
    /* something went wrong */
  }
  while( (f=h264_decode_frame(0)) ) {
    /* do something with the decoded frame */
  }
  h264_close();
}
```

4.2.2.1. The API functions

```
int h264_open(char *filename);
```

The function `h264_open` is used to open a H.264 elementary stream from a file. `filename` is the name of the file being opened. On success, the return value contains the size of a frame in pixels in an encoded format. The macros `H264_WIDTH` and `H264_HEIGHT` are used to derive the actual frame width and height from this value. If the file cannot be opened, an error message is printed out to the standard error stream and 0 is returned.

Note that there is no kind of handle or identifier. The library only handles a single global decoder context.

Internally, this function opens the file and reads NAL units until both a sequence parameter set and a picture parameter set are decoded to build up the neccessary information for the decoder. Then, memory for all further operations is allocated.

```
frame *h264_decode_frame(int verbose);
```

The function `h264_decode_frame` decodes the next frame from the opened file. If `verbose` is non-zero, a frame-counter and slice type indicator is written to standard output before decoding.
The return value is a pointer to a `frame` structure (see below) that contains the decoded frame image. `NULL` is returned if there was an error during the decoding process or EOF is reached on the input file.

Internally, this function reads NAL units until a decodable slice is reached, which is in turn decoded.

```
void h264_close();
```

The function `h264_close` closes the stream file and frees all associated memory.

```
int h264_frame_no();
```

The function `h264_frame_no` returns the number of the last frame that was decoded successfully.

```
void h264_rewind();
```

The function `h264_rewind` rewinds the current stream to its start and resets the frame counter. All associated memory is kept.
Due to the fact that `h264_decode_frame` silently skips any sequence or picture parameter sets, this can be used to loop over a file when EOF is reached.

4.2.2.2. The frame structure

Decoded frames are stored in a structure that contains all information about its dimensions as well as pointers to the actual image data:

```
typedef struct __frame {
  int Lwidth,Lheight,Lpitch;
  int Cwidth,Cheight,Cpitch;
  unsigned char *L, *C[2];
} frame;
```

The Lwidth, Lheight and Lpitch members contain the image width and height in pixels and the luminance channel's line distance (pitch) in bytes, respectively. For the time being, pitch is always equal to width. The Cwidth, Cheight and Cpitch members contain the same information for the chrominance channels. Because H.264 is limited to 4:2:0 subsampling, the chroma channels always have half the size of the luma channel.

Pixel data is stored in the memory block pointed to by L for luminance and C for chrominance: C[0] refers to the Cb channel, C[1] refers to the Cr channel. Pixels are stored bytewise and without padding as an array of unsigned bytes ranging from 0 (0.0) to 255 (1.0).

4.3. The input system

The input system reads the input file blockwise and offers an interface for bitwise stream reading as well as parsing of common syntax elements, such as Exp-Golomb codes and static code tables. It is also directly integrated with the NAL unit parser.

4.3.1. Ring-buffered file input

Data is read from the input file blockwise and stored in a global ring buffer of fixed size (ring_buf in module input). The buffer is split into two parts; after one half of the buffer has been passed, a new block is read from the file. To detect EOF, the number of bytes still remaining in the buffer is stored in a variable called input_remain, which is incremented each time a new block has been read from disk and decremented on each read operation inside the buffer. Management of the buffer is almost completely done by the NAL unit parser described below.

Access to the file is provided by the module in_file, which contains only few functions. input_read reads a number of bytes from disk into memory and updates the input_remain counter. (In fact, the number of bytes to read is always half of the buffer size.) input_rewind seeks to the begin of the file and completely fills the ring buffer. Finally, there are input_open to open and rewind and input_close to close the file.

The module in_file can be easily replaced by specialized modules for other types of input. Actually, the in_probe module of the DSP port is a replacement for in_file.

4.3.2. NAL unit separation

The H.264 bitstream is organized in discrete packets, called “NAL units”, of variable length. Thus, a method of separating packets must exist. In the H.264 specification, Annex B, a simple method for doing this is described: Basically, NAL units are separated by a simple (at least) 4-byte sequence containing a big-endian 1, i.e. 00 00 00 01. Finding a NAL unit boundary is therefore as simple as searching for this byte sequence. The NAL unit separators themselves are discarded, only the following NAL unit content is processed further.

This process is implemented in the first part of the function get_next_nal_unit in module nal: The ring buffer content is scanned for a 00 00 00 01 sequence, wrapping around at the boundary and fetching new blocks from disk, if necessary. When a NAL unit start is found, its end is searched in a similar way by scanning for 00 00 00, but this time, all bytes are copied into a second buffer, called nal_buf.

4.3.3. Bitwise input

For bitwise access to the contents of `nal_buf`, the module `input` defines a number of variables: `nal_pos` is the current position inside `nal_buf` in bytes and `nal_bit` is the bit pointer inside the specified byte.

To read a number of bits from the buffer, the function `input_get_bits` reads the next four bytes from the stream into a 32-bit integer register and cuts away unwanted bits by shifting appropriately. If only one bit is needed, `input_get_one_bit` directly retrieves the bit from the current byte. Both functions advance the input pointer after extracting their values.

4.3.4. Exp-Golomb codes

The main syntax element in H.264 are Exp-Golomb-coded integers. Exp-Golomb codes are special huffmann codes with a regular construction scheme that favors small numbers by assigning them shorter codes.

Parsing of Exp-Golomb codes is implemented in the module `cavlc` by the functions `get_unsigned_exp_golomb` and `get_signed_exp_golomb`.

4.3.4.1. Unsigned Exp-Golomb codes

Exp-Golomb codes consist of three parts: First, there is a number of n 0 bits, followed by one 1 bit and again n bits with an “offset” value v. The final code value x is derived by

$$x = 2^n - 1 + v$$

Some examples for Exp-Golomb codes are:

Value	Code
0	**1**
1	0**1**0
2	0**1**1
3	00**1**00
4	00**1**01
⋮	⋮
42	00000**1**01011
43	00000**1**01100

4.3.4.2. Signed Exp-Golomb codes

Signed Exp-Golomb codes are parsed by retrieving an unsigned Exp-Golomb code c from the bitstream and mapping it from unsigned to signed using the following rule: If c is even, $-c/2$ is returned; if c is odd, $(c+1)/2$ is returned; zero remains unchanged.

Here again some examples for this unsigned-to-signed-mapping:

unsigned Exp-Golomb code	0	1	2	3	4	…	42	43
signed Exp-Golomb code	0	1	-1	2	-2	…	-21	22

4.3.5. Code tables

Some syntax elements in H.264 are stored as fixed, predefined huffmann codes. In order to parse these, a generic code parser is implemented in `cavlc`. The code tables are represented by a sorted list of code values. These, in turn, are shifted in a way that they occupy the most significant bits in a 32-bit-value. Parsing a code is done by reading 24 input bits, shifting them, and searching the appropriate table entry using simple binary search. This is accomplished in the function `get_code`. The code tables themselves are located in the file `cavlc_tables.h`.

4.4. Parsing basic headers

Each NAL unit has a header which, among other values, indicates the NAL unit type. A typical H.264 stream contains only four different NAL unit types (out of 16): A “sequence parameter set” and a “picture parameter set” at the very beginning of the stream define the basic structure, dimensions and used coding tools of the stream. Then, a number of “coded slice” NAL units follow (there are two different types; one is used for I and SI slices, the other one for the other slice types). Because the decoder implementation does not support any kind of data partitioning or multiple slices per frame (which are uncommon features anyway) nor interlaced video, a slice in H.264 terminology always means a full frame in this implementation. The slices itself consist of a (to some extent) common header and the actual slice data that may differ depending on the slice type.

4.4.1. NAL unit parsing

Each NAL unit consists of a one-byte header. The MSB of this byte is unused and always zero. The following 2 bits contain a so-called “reference indicator”, which is not needed for this implementation and is therefore discarded. The least significant 5 bits of the header contain the NAL unit type. The following NAL unit types are of interest for this implementation. All other NAL unit types generate a warning and are skipped:

Type	Name	Description
1	Coded slice of a non-IDR picture	Slice data for a P/SP/B slice.
5	Coded slice of an IDR picture	Slice data for an I/SI slice.
7	Sequence parameter set	One of the stream headers.
8	Picture parameter set	One of the stream headers.

After the header, the NAL unit data follows. Normally, it is encoded using an escape rule that avoids emulation of the Annex B NAL unit separator in the raw data bytes. But the probability for such a situation is almost zero. Actually, it can only occur in I_PCM macroblocks which are generally not used in normal video streams. Therefore, the escape rule is ignored in this implementation.

Parsing of the NAL unit header is done in the second part of the function `get_next_nal_unit` of module `nal`.

4.4.2. The sequence and picture parameter sets

The sequence parameter set (abbreviated SPS) and picture parameter set (PPS) contain the basic stream headers. Each of these parameter sets is stored in its own NAL unit, usually occupying only few bytes. Both parameter sets have their own ID values so that multiple video streams can be transferred in only one H.264 elementary stream. This feature is not supported by this implementation; all IDs are ignored.

Most of the data fields of the parameter sets are stored as Exp-Golomb codes or single-bit flags.

The most important fields of a **sequence parameter set** are:

- A profile and level indicator signalling conformance to a profile/level combination specified in H.264 Annex A.
- Information about the decoding method of the picture order. Only the trivial picture order (i.e. decoding order is identical to playback order) is supported.
- The number of reference frames. Because long-term prediction is not implemented at all, only a value of 1 is supported.
- The frame size in macroblocks as well as the interlaced encoding flag. Interlaced video is not supported.
- Frame cropping information for enabling non-multiple-of-16 frame sizes. These values are ignored, the decoder always decodes full macroblocks.
- Video Usability Information (VUI) parameters, such as aspect ratio or color space details. These are ignored.

The most important fields of a **picture parameter set** are:

- A flag indicating which entropy coding mode is used. Only CAVLC is supported.
- Information about slice data partitioning and macroblock reordering. Neither are supported.
- The maximum reference picture list index. As this is directly related to long-term prediction, it is not supported.
- Flags indicating the usage of weighted (bi)prediction. This is not supported.
- The initial quantization parameters as well as the luma/chroma quantization parameter offset.
- A flag indicating whether inter-predicted macroblocks may be used for intra prediction or not (“constrained intra prediction”).

Structures to hold the information from the parameter sets, functions to parse them and a function to check for unsupported coding tools are located in the module `params`.

4.4.3. The slice header

The slice header precedes every slice and contains (among others) the following information:

- The slice type (I/P/B/SI/SP), dependent on the NAL unit type.
- The corresponding parameter set ID. (Ignored.)
- The frame/field picture indicator. Only used in interlaced encoding mode and therefore not supported.
- Some fields concerning long-term or bidirectional prediction, and macroblock reordering.
- The quantization parameter difference to the last slice.
- Deblocking filter control hints.

The structures and functions to parse slice headers are implemented in their own module, `slicehdr`.

4.5. Mode Prediction

Most per-block decoding parameters in H.264 are not encoded directly, but as a difference to a predicted value derived from surrounding blocks. Precisely, the residual data context, Intra_4x4 prediction modes and motion vectors use a scheme like this. All structures, macros and functions are concentrated in the module `mode_pred`. All the information needed to perform mode prediction are kept in a single structure:

```
typedef struct _mode_pred_info {
  // per-macroblock information     (16x16)
  int MbWidth, MbHeight, MbPitch;
  int *MbMode;
  // per-chroma block information    (8x8)
  int CbWidth, CbHeight, CbPitch;
  int *TotalCoeffC[2];
  // per-transform block information (4x4)
  int TbWidth, TbHeight, TbPitch;
  int *TotalCoeffL;
  int *Intra4x4PredMode;
  int *MVx,*MVy;
} mode_pred_info;
```

Depending on the type of information, it is stored as one item each 16x16 pixels (i.e. one macroblock), 8x8 pixels (i.e. one chroma transform block) or 4x4 pixels (i.e. one luma transform block). Consequently, three copies of the `Width`, `Height` and `Pitch` fields (with semantics identical to those in the `frame` structure) exist. Actual data is stored in arrays of integer values of typically 32 bits (but 16 bits may be used as well):

`MbMode` stores the macroblock mode for each macroblock. Macroblocks that are not yet decoded are assigned the value -1.

`TotalCoeffL` stores the total number of nonzero transform coefficients for each luma transform block.

`TotalCoeffC` stores the total number of nonzero transform coefficients for each chroma transform block.

`Intra4x4PredMode` stores the Intra_4x4 prediction mode of each luma block if it is encoded using Intra_4x4 prediction. Other blocks are assigned the value -1.

`MVx` and `MVy` store the motion vectors for blocks that are encoded using inter prediction. Any other block is assigned the value `MV_NA` = `0x80808080`.

A `mode_pred_info` structure is typically only allocated once when image dimensions are known (i.e. after parsing the parameter sets) by the function `alloc_mode_pred_info` and freed upon decoder deinitialization by `free_mode_pred_info`. In addition, it must be cleared before decoding a frame with the function `clear_mode_pred_info`.

For each member of the `mode_pred_info` structure, a macro exists that facilitates access to a specific item in the two-dimensional raster scan array.

4.5.1. Macroblock mode retrieval

To read back a stored macroblock mode, a function `get_mb_mode` exists that simply returns the macroblock mode at the specified position in macroblock coordinates, or -1 if the position points out of the image.

4.5.2. TotalCoeff prediction

Residual data syntax elements (i.e. transform coefficient levels) are parsed using a value named `nC` as a parameter. This value is an approximation of the `TotalCoeff` that can be expected for this block, and it depends on the `TotalCoeff` values of surrounding blocks. The idea behind such a prediction mechanism is that areas with similar `TotalCoeff` values (i.e. similar spectral complexity) are likely to occur. So, an optimized encoding for the expected `TotalCoeff` range can be selected.

The prediction process for `nC` works as follows: First, the `TotalCoeff` values of the upper and left neighbor of the current transform block are extracted. If none of these so-called “candidate predictors” is available, e.g. because the corresponding blocks have not been decoded yet or are located outside the bounds of the image, zero is returned. If only one of the candidate predictors is available, it is returned. Otherwise (both are available), the average of both values is rounded up and is returned.

4.5.3. Intra_4x4 prediction mode prediction

The Intra_4x4 prediction modes are encoded using a prediction scheme themselves because it is likely that neighboring blocks contain gradients in the same direction.

To predict the Intra_4x4 mode for a specific transform block, the Intra_4x4 prediction mode numbers for the blocks directly above and to the left of this block are read first. If one of these candidate predictors is **either** located inside an Intra_16x16 predicted macroblock **or** constrained intra prediction is off and it is located inside an Inter macroblock, prediction mode 2 (Intra_4x4_DC) is pretended to be the value of this candidate predictor.

If any of these candidate predictors is not available because it is located outside of the image, is has not been decoded yet or it is located inside an Inter predicted macroblock while constrained intra prediction is used, prediction mode 2 (Intra_4x4_DC) is returned. Otherwise, the minimum of both candidate values is returned.

In the implementation (functions `get_Intra4x4PredMode` and `get_predIntra4x4PredMode`), most of these special cases are eliminated using -1 as the “empty” value for not-yet-decoded blocks. If one of the candidate predictors contains this value, the minimum is forced down to -1, which is replaced by output value 2 afterwards. On the other side, upon rendering an Intra_16x16 macroblock, all Intra_4x4 prediction mode values inside this macroblock are filled with the ‘artificial’ value 2.

4.5.4. Motion vector prediction

Motion vector prediction is a standard coding tool since MPEG-4 Simple Profile. It enables efficient compression of large uniformly moving areas. For the derivation of motion vector predictors for macroblock or sub-macroblock partitions, H.264 uses three candidate predictors A, B and C, represented by the motion vector of the left, upper and upper-right neighbor of the current partition, in that order.

A candidate predictor is marked *available* if it is located inside the already decoded area of the image. Additionally, it is marked *valid* if it is located inside an Inter macroblock. Both unavailable and invalid predictors are implicitly assigned the motion vector coordinates $(0, 0)$. All this is done in function `get_MV` which returns the motion vector and its availability status in a temporary structure.

The actual prediction process first loads the three candidate predictors described above. If C itself is not available, the upper-left neighbor is used instead. Then, for 16x8 and 8x16 partitions, special rules may directly return one of the candidate predictors as the final motion vector predictor. If this is not the case, but one and only one of the candidate predictors is valid, it is returned directly. Otherwise, the median of all three candidate predictors is computed and returned as the motion vector predictor. This algorithm is implemented in function `PredictMV`.

For P_Skip macroblocks, the prediction process (implemented in `Predict_P_Skip_MV`) differs slightly: First, only the left and upper neighbor are examined. If any of them is available, valid, and equal to $(0, 0)$, the resulting motion vector is set to $(0, 0)$. Otherwise, normal motion vector prediction for a 16x16 partition is performed.

Because motion vectors are potentially determined for areas greater than one transform block, multiple items in the `MVx` and `MVy` members of the `mode_pred_info` structure (which therefore can be considered as a “motion map”) may be updated simultaneously. For this purpose, a function `FillMVs` exists.

Complete motion vector derivation is done by the functions `DeriveMVs` and `Derive_P_Skip_MVs`, which perform motion vector prediction, read and apply the motion vector difference Exp-Golomb codes if applicable, and fill the final motion vector into the motion map in a single step.

4.6. Slice Reconstruction

The main decoding work is done by the single function `decode_slice_data` in module `slice`. This function performs the complete slice parsing and rendering process. It receives the current slice header, the sequence and picture parameter set, the current NAL unit, a reference and a working frame, and a mode prediction structure as inputs and operates as follows:

1. **Initialization**

    (1) The mode prediction information is cleared.

   (2) The current quantization parameter is initialized with the slice quantization parameter.
   (3) The current macroblock address is set to 0.

   All following steps are performed for each macroblock.

2. **Handling of P_Skip macroblocks**
   In P slices, the value `mb_skip_run` is read and the indicated number of P_Skip macroblocks is rendered using appropriate motion compensation.

3. The macroblock mode and associated information, such as the number of partitions, their shape and possibly the Intra_16x16 prediction mode, are decoded.

4. **Prediction information parsing**

   - **I_PCM** macroblocks are parsed and rendered directly.
   - For **Intra_4x4** macroblocks, the 16 Intra_4x4 prediction modes are decoded as defined in the H.264 standard: For each subblock the predicted Intra_4x4 prediction mode is determined and a flag is read from the bitstream, indicating whether the prediction shall be used or not. If not, an additional 3-bit value is read. To encode the 9 possible prediction modes in 3 bits, the predicted value is used in a special way: If the value read is equal to or greater than the prediction, it is increased by 1 to obtain the final Intra_4x4 prediction mode number.
   - For Inter macroblocks, the motion vectors for all the macroblock partitions are decoded.
   - For Inter macroblocks with sub-macroblock partitioning, the number and shapes of the sub-macroblock partitions and their motion vectors are determined.

5. The **coded block pattern** value is decoded. This is a number indicating which blocks are actually containing non-zero transform coefficients and therefore must be encoded explicitly.

6. The transform coefficient level cache is cleared.

7. **Transform coefficient level parsing**
   Depending on the macroblock type and coded block pattern value, Transform coefficient levels are parsed and the resulting `TotalCoeff` values are filled into the mode prediction information arrays.

8. **Rendering**

   - For **Intra_4x4** macroblocks, each of the 16 luma subblocks is first predicted, transformed and the difference is merged into the image. Intra Chroma prediction is performed afterwards.
   - For **Intra_16x16** macroblocks, the whole macroblock is predicted, then the 16 luma sub-blocks are transformed and merged into the image, and finally Intra Chroma prediction is performed.
   - For **Inter** macroblocks, the 16 luma subblocks are motion-compensated, transformed and merged into the image. Chroma motion compensation is performed afterwards.
   - For all macroblock modes, the chroma residual is transformed and merged into the image.

The (sub-)macroblock mode parsing steps mentioned above are not directly performed inside `decode_slice_data`, but in an extra module `mbmodes` by the functions `decode_mb_mode` and `decode_sub_mb_mode`. This module contains symbolic constants for the various modes as well as tables containing the values associated to the mode numbers.

4.7. Intra and Inter Prediction

Intra and inter prediction are implemented in two different modules: `intra_pred` for intra prediction and `mocomp` for motion compensation, which is synonymous with inter prediction.

4.7.1. Intra_4x4 prediction

H.264 defines nine Intra_4x4 prediction modes that use the 13 surrounding pixels around the current block as a reference:

No.	Name	Description
0	Vertical	The 4 upper neighbor pixels are repeated downwards.
1	Horizontal	The 4 left neighbor pixels are repeated to the right.
2	DC	The average of the 8 left and upper neighbor pixels is computed and the block is filled with it.
3	Diagonal Down Left	Pixels are repeated in a 45-degree angle from the upper right to the lower left corner.
4	Diagonal Down Right	Pixels are repeated in a 45-degree angle from the upper left to the lower right corner.
5	Vertical Right	Pixels are repeated in an approx. 30-degree angle from the upper edge to the lower edge, slightly drifting to the right.
6	Horizontal Down	Pixels are repeated in an approx. 30-degree angle from the left edge to the right edge, slightly drifting downwards.
7	Vertical Left	Pixels are repeated in an approx. 30-degree angle from the upper edge to the lower edge, slightly drifting to the left.
8	Horizontal Up	Pixels are repeated in an approx. 30-degree angle from the left edge to the right edge, slightly drifting upwards.

The 9 modes are implemented in static functions in `intra_pred`, one for each mode. These can not be called directly; only the function `Intra_4x4_Dispatch` is public. It loads the surrounding pixels used for the prediction process into a temporary array. If some of the reference pixels are not available because they are not yet decoded, the available edge pixels are repeated accordingly. The actual prediction functions use these temporary values to perform their task.

4.7.2. Intra_16x16 and Intra_Chroma prediction

These two prediction types work in an almost identical way, with the only exception that Intra_16x16 operates on blocks of 16x16 pixels, while Intra_Chroma uses 8x8-sized blocks. Accordingly, the implementations of both are almost equal. Both use a static functions / public dispatch function scheme, too. The exported functions have the names `Intra_16x16_Dispatch` and `Intra_Chroma_Dispatch`.

The following prediction modes are defined for these two types:

No.	Name	Description
0	DC	The average of all neigboring pixels directly above and to the left of the current block is computed and the block is filled with it.
1	Horizontal	The 8 or 16 left neighbor pixels are repeated to the right.
2	Vertical	The 8 or 16 upper neighbor pixels are repeated downwards.
3	Plane	A two-dimensional gradient is fitted to the block edges.

4.7.3. Motion compensation

Motion compensation is always performed on 4x4-pixel blocks for luma and 2x2-pixel blocks for chroma. Although motion vectors may be specified for larger areas, they are entered into the motion map inside the mode prediction information structure for single 4x4-pixel blocks, and thus, they are processed in those quantities.

The main motion compensation function is `MotionCompensateTB`, which motion-compensates a transform block in all luma and chroma channels from the reference frame into the working frame using the block coordinates and the motion vector supplied as parameters. It is normally not called directly; instead, the function `MotionCompensateMB` motion-compensates all 16 sub-blocks inside a macroblock using the motion vectors in the mode prediction information structure, calling `MotionCompensateTB` for each block with the appropriate parameters.

4.7.3.1. Luma motion compensation

Luma motion compensation is accomplished in two steps: First, a fixed-size area of 9x9 pixels (block size 4 + filter tap size 4 + 1 spare pixel) is copied from the *integer* motion vector position of the reference frame buffer into a temporary buffer. During this process, all edge pixels are repeated outwards so that regions outside the reference image contain sensible values. In the second step, the fractional pixel positions are calculated using fixed integer pixel positions inside the temporary buffer. This two-step method is not very efficient, but it facilitates handling of out-of-image motion vectors and fractional pixel positions by simply treating them separately.

The intensity values of fractional pixel positions are determined using a 6-tap FIR filter for half-pixel positions and bilinear interpolation for quarter-sample positions. The rather complex 6-tap filter is used in H.264 because it retains much more sharpness than a pure bilinear filter like in MPEG-4.
For example, if A to F are pixel values at integer positions, the half-pixel value cd between C and D is computed as follows:

$$cd = \frac{1}{32}(A - 5B + 20C + 20D - 5E + F)$$

4.7.3.2. Chroma motion compensation

Chroma motion compensation is performed in a similar manner, with the exception that the temporary buffer has a size of 3x3 pixels (block size 2 + 1 spare pixel) and the fractional pixel position determination algorithm is a simple bilinear interpolation of one-eighth pixels.

4.8. Parsing and Rendering of Residuals

A total of three modules is involved in the residual handling process: First, the residual module with its single function residual_block parses transform coefficient levels from the data stream into arrays. Then, the modules block (high-level operations) and coretrans (low-level operations) dequantize the coefficients, perform the inverse transforms and finally render the blocks into the image.

4.8.1. Coefficient Level Parsing

The parsing function uses the nC value already discussed in the mode prediction section to determine the code table that has to be used for parsing the CoeffToken syntax element. After this has been finished, the variables TotalCoeff (the number of non-zero coefficients) and variables TrailingOnes (the number of coefficients at the end of the scan with an absolute value of 1) are derived from CoeffToken.

Thereafter, the levels of the non-zero coefficients are read into a temporary array in reversed order, only reading a single sign bit for the first TrailingOnes coefficients. The other coefficients are read using a prefix/suffix scheme that accommodates monotonic increasing sequences which can usually be found in that place.

Having read the coefficient levels, the TotalZeros syntax element is read using one of 15 different code tables selected by the TotalCoeff value. It indicates the number of zeros left to fill in between two adjacent non-zero coefficients. The actual distribution of these zero-leveled coefficients is determined afterwards by parsing the run_before syntax elements stored for each coefficient until no zeros are left.

With both the coefficient level and zero run information, the final coefficient scan data is compiled.

4.8.2. DC level transformation

The functions transform_luma_dc and transform_chroma_dc in module block handle the 4x4 Hadamard luma DC transformation used in Intra_16x16 macroblocks and the 2x2 chroma DC transformation, respectively. Both functions include the necessary pre-dequantization step.

4.8.3. Quantization, Transformation and Rendering

These three sub-steps are closely integrated and are in fact all handled by the wrapper functions enter_luma_block and enter_chroma_block in module block. These call inverse_quantize to perform the position-dependent dequantization. Transformation and rendering is performed in a single step in function direct_ict in module coretrans. This function uses the separated two-step algorithm described in the H.264 standard paper and directly adds the calculated pixel values to the frame buffer at the end of the second transformation pass.

4.9. The DSP port

The C implementation was ported to the Texas Instruments TMS320C64xx platform using the Code Composer Studio v2 compiler and IDE. Most of the code runs out-of-the-box with this compiler, with the following exceptions:

- **File input is not available.**
  The module `in_file` that handles file input on normal disk operating systems is not available in the DSP platform. For this purpose, another module named `in_probe` with the same interface syntax and semantics exists. Using this module, the decoder obtains its input using the probe and File I/O functions of the CCS IDE. This is accomplished via a probe point in the function `input_read`.

- **x86 assembler is not available.**
  The directive `#define USE_X86_ASM` must be removed from `common.h`.

- **Direct performance counting is not available.**
  In `perf.h`, `#define PERF_METHOD_none` must be set (and, additionally, `INT64` must be redefined to be a 32-bit integer type). This disables the built-in performance counter library; to measure code efficiency on the DSP, the CCS IDE's integrated profiling functions have to be used.

- **The player application is not available.**
  The `playh264` application does, of course, not work on the DSP platform. Instead, an alternate main module named `dspmain.c` is supplied. It contains a simple decoding loop for use with another probe point, this time connected to a video window inside the CCS IDE.

- **Target-specific settings:** If used with the C64xx Simulator, the heap size must be set explicitly in the project options. 2 MB heap size are sufficient to decode full PAL images.
  In every case, a large memory model must be selected; `-ml0` (Far Aggregate Data) suffices.

Using the decoder with ATEME NVDK64

The DSP port was also developed and tested on a C64xx development board by ATEME. The NVDK64 is a PCI board with video and audio inputs (not used) and outputs. In order to use it, the preparation steps in the board's manual need to be followed precisely.

The implementation itself contains a NVDK64-specific part as an option in `dspmain.c`. It initializes the board's VGA output at 352x576 pixels and decodes frames with a maximum size of 352x288, doubling the lines. No timing is implemented whatsoever; video frames are shown as they are decoded.

5. Performance

The C code is almost completely unoptimized. Only a small number of unrolled loops, assembly optimizations and optimized pointer arithmetics are implemented. Despite this fact, the code is comparatively fast:

- An Intel Celeron CPU at 450 MHz under Linux running `playh264` compiled with GNU GCC 3.3 (optimization level 1) achieves about 30 frames per second on an 320x144 video clip. Extrapolating this, a 1 GHz x86 would suffice to decode CIF or SIF video in realtime.
- The TMS320C6416 on the NVDK64 board running at 600 MHz decodes the same video clip at 2 to 5 frames per second, fluctuating heavily. The differences from no optimization to optimization level 1 are almost zero. Unfortunately, level 2 optimizations could not be tested because the compiler hangs on compilation of module `intra_pred`.

5.1. Identifying Hot Spots

To determine the sections in the source code that take most of the execution time, a simple performance counter library resides in the module `perf`. It simply adds CPU clocks for each part of the decoding process. The following values were determined for a 320x144 video clip with a length of 360 frames:

Sequence type and target	I frames only		full sequence	
	x86	**C64xx**	**x86**	**C64xx**
Parsing	1.9%	30.5%	1.7%	18.1%
Intra prediction	14.4%	11.7%	1.2%	1.3%
Motion compensation	—	—	43.9%	64.8%
Transformation and Rendering	83.6%	57.8%	53.2%	15.9%

The differences between the two target platforms are huge: While parsing is done almost for free on x86, it is a serious issue on the C64xx. Conversely, the transformation step takes most of the time on x86, but is clearly superseded by motion compensation on the DSP. Thus, optimization needs to be done at different points for both platforms.

The transformation and rendering process is the most critical point on the desktop PC target. It may be vastly sped up using MMX SIMD instructions. But also on the DSP platform with its bulk of general-purpose registers (all 16 pixels of the block can be held in registers) and massively parallel execution engines, significant speedup is possible.

Optimization of the motion compensation process is significantly more complicated. A complete rewrite, eliminating the first copy step if the motion vector points into the image, might be feasible. Furthermore, the 6-tap filter routine can be efficiently programmed in DSP or MMX assembler, and the whole fractional-pixel computation may be sped up by implementing an optimized control flow.

Intra prediction, on the other hand, is already astoundingly efficient. However, since the Intra_4x4 prediction modes fill their blocks using only 4 to 8 different values, a significant speedup can be expected if these values are kept in registers.

Parsing the data stream is already very fast on the PC and may be kept as C code there. But on the DSP platform, this part of the decoding process takes a tremendous amount of time. Speedups may be achieved by optimizing the bitwise input functions, but unfortunately, in general, the rather complex C code used for parsing is hard to translate into assembly language.

5.2. Memory usage

The memory footprint of the decoder is clearly dominated by frame buffer storage, but the mode prediction information structure also contributes to it. In total, 4.1 bytes of memory are needed per display pixel:

bytes	**Item**
1	L component of current frame
1/4	Cb component of current frame
1/4	Cr component of current frame
1	L component of reference frame
1/4	Cb component of reference frame
1/4	Cr component of reference frame
4/256	macroblock mode cache
4/64	TotalCoeffC cache
4/16	TotalCoeffL cache
4/16	Intra4x4PredMode cache
8/16	motion vector cache
4.078	total

In addition to these resolution-dependent requirements, about 128 KiB of additional heap need to be available for static data. Using this figure, 640 KiB of RAM are enough to decode CIF video (up to 352x288). Full PAL (720x576) fits into 2 MiB, and Full HDTV (1920x1088) needs 9 MiB of heap.

The memory usage may be slightly decreased by using smaller data types for the mode prediction information structure.

6. Limitations and Future Work

The H.264 decoder presented here is very limited in terms of both features and speed. It is a minimal, straight-forward implementation, and needs to be improved in the following areas:

First, the in-loop deblocking filter and long-term prediction need to be implemented. Until these features are available, the decoder is **not** compliant to H.264 Baseline profile.

Second, the algorithm needs heavy optimization. The DSP version currently runs at about 3 frames per second on a 600 MHz chip, which is about 10 times too slow for realtime playback. Thus, a rewrite of the inner decoding loops in linear or even machine assembly is inevitable.

A. References

[1] Thomas Wiegand, Gary Sullivan, Ajay Luthra. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 / ISO/IEC 14496-10 AVC), Document JVT-G050r1, May 2003.
http://www.stewe.org/itu-recs/h264.pdf

[2] Iain E. G. Richardson. H.264 / MPEG-4 Part 10 Tutorials.
http://www.vcodex.com/h264.html

[3] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, Ajay Luthra. Overview of the H.264/AVC Video Coding Standard. IEEE Transactions on Circuits and Systems for Video Technology, July 2003.
http://bs.hhi.de/~wiegand/csvt_overview_0305.pdf

[4] Karsten Sühring, Heiko Schwarz, Thomas Wiegand. Effizienter kodieren: Details zum kommenden Videostandard H.264/AVC. c't – Magazin für Computertechnik, Verlag Heinz Heise, Heft 6, March 10, 2003 (German)
http://bs.hhi.de/~wiegand/ct2003.pdf

[5] H.264/AVC Reference Software.
http://bs.hhi.de/~suehring/tml/download/

[6] ffmpeg and libavcodec.
http://ffmpeg.sourceforge.net/

[7] MPlayer, the movie player for Linux.
http://www.mplayerhq.hu/