MPEG-2 Video Transcoding

by

Delbert Dueck

Final report submitted in partial satisfaction of the requirements for the degree of

Bachelor of Science

in Computer Engineering

in the

Faculty of Engineering

of the

University of Manitoba

Faculty and Industry Supervisors:

Dr. Ken Ferens, Assistant Professor

Fern Berard, Norsat International Inc.

Spring 2001

© Copyright by Delbert Dueck, 2001.

ABSTRACT

Common sense dictates that as Internet connection speeds increase, so too does the bandwidth of commonly transmitted data. Presumably, this means that today's Internet of text and images will evolve into an Internet with high-quality video conforming to the MPEG standard. This report begins by introducing MPEG, the MPEG-2 standard, and providing a description of the structure of an MPEG-2 video elementary stream. This is followed by a detailed examination of the MPEG-2 picture encoding process. The fourth chapter extends the encoding procedure by developing and implementing an algorithm for transcoding pictures and video streams. Finally, pictures from transcoded video are evaluated and it is demonstrated that they retain much of the picture quality of the original high-bandwidth video encoding.

ACKNOWLEDGEMENTS

I would like to thank Dr. Ken Ferens for the assistance and guidance he gave over the course of this project. I would also like to acknowledge the contributions of Norsat International Inc., who provided me with the MPEG-2 standards documents on which this thesis is based. I also had the privilege of using their MPEG-2 video encoding hardware and software to produce the test bit streams used in this report.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Gordon Moore, founder of the microchip giant Intel Corporation, centered his business on a principle he once enunciated. Moore's law states that computer processing power doubles every eighteen months while costs remain constant. This law is widely applicable to information technology-related issues, including bandwidth for data transmission.

The history of the Internet illustrates this well. When Internet access became widely available in the early 1990's, websites were primarily text. As the standard modem connection speed increased beyond 14.4 kbps, pages were increasingly filled with images demanding these faster connections. High-speed cable and digital subscriber line connections of the past few years represent the continuation of this trend, and the data of choice is now digitized audio, often in the MP3 file format.

The next step up is digital video. Low-bandwidth (and thus low-quality) video web-casts are already available at many Internet media outlets, especially newsgathering organizations. Higher-quality video service will likely become more common when mass-market satellite Internet service becomes widespread. Two matters, however, need to be settled before video becomes as much a part of the Internet as text and images. First, standards must be in place for compression and distribution of digital video. Second, a balance must be reached between offering high-quality broadband video and offering services compatible with an assortment of connection types and speeds.

The Moving Pictures Experts Group (MPEG), formed in 1988 under the direction of the International Organization for Standardization (ISO), has mostly resolved the first issue. Their mandate was to devise standards for the encoding and transmission of multimedia data (including audio and video). The ensuing MPEG specifications have enjoyed a high degree of acceptance, exemplified by the popularity of the MP3 (MPEG-1|2 audio layer 3) audio file format and the MPEG-2 digital versatile disc (DVD).

The second issue, however, is not addressed by the MPEG specifications. There is an ongoing search for a method of providing web-based video service compatible with a wide range of connection speeds, short of offering segregated services for each class of connection.

The following report explains the MPEG-2 compression standard, specifically as it relates to digital video. This thesis also investigates a procedure for partially decoding and re-encoding MPEG-2 video bit streams. This technique, known as transcoding, allows fundamental stream characteristics (e.g. bit rate) to be customized in the course of transmission. This could be used to provide an efficient personalized video service capable of adapting to each client's needs.

2 BACKGROUND TO MPEG-2 VIDEO

The following sections provide background information to the MPEG standards, and a more specific overview of the MPEG-2 standard. This is followed by a detailed description of the structure of an MPEG-2 video bit stream.

2.1 OVERVIEW OF MPEG

The Moving Pictures Experts Group (MPEG) is a committee formed in 1988 to standardize the storage of digital audio and video on compact discs. It operates under the joint direction of International Organization for Standardization (ISO) and the International Electro-Technical Commission (IEC)[1].

MPEG released its first standard, *MPEG-1: Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s*, in 1993. At that time it consisted of three parts: audio, video, and systems (combining the audio and video into a single stream). A fourth part describing conformance testing of coded bit streams was subsequently released in 1995. There is also a fifth part to the standard, offering a sample software implementation of a compatible encoder [3 128].

The MPEG-1 standard is directed towards the compact disc medium that has a single-speed data rate of 1.2 Mbps. Bit streams typically contain audio sampled at 44.1 kHz and video sampled at 30 frames/second with a resolution of 360 by 240, which bears a close resemblance to North American interlaced NTSC television pictures. MPEG-1 bit streams coded with these resolution parameters and within the 1.2 Mbps bandwidth constraint have a video quality similar to consumer VHS videotape. The standard is also crafted to allow for cost-effective hardware decoder implementations that require only 512 KB of memory for the video buffer [4].

As technology progressed in the early 1990s, it became increasingly clear that the quality of MPEG-1 constrained within those parameters was unsatisfactory. Even though the resolution of MPEG-1 video could be scaled up to $4080 \times 4080 \times 60$ fps (which would largely solve the quality problem), MPEG-1 had difficulty encoding interlaced video, especially with motion vectors used

[1] MPEG is actually a nickname. The official title for the group is ISO/IEC JTC1 SC29 WG11 (ISO/IEC Joint Technical Committee 1, Sub-committee 29, working group 11).

for compression. To address these shortcomings, MPEG released a new standard in 1995, *MPEG-2: Generic Coding of Moving Pictures and Associated Audio*.

MPEG-2 bit streams usually fall within a range of bit rates between two and fifteen Mbps, which is suitable for transmission via satellite or cable, or storage on Digital Versatile Discs (DVDs). It contains several key improvements over MPEG-1, including support for both interlaced and progressive (non-interlaced) video formats, support for multi-channel surround sound and/or multilingual sound (MPEG-1 only supports stereo audio), and provision for a wider range of audio sampling frequencies [4]. A more thorough overview of MPEG-2 is given in §2.2.

Work also began on a third specification, MPEG-3, aimed at 20-40 Mbps High Definition Television (HDTV) applications. By 1993, however, it became apparent that this standard would not be completed in time for proposed launches of the technology. In addition, leading HDTV providers determined that MPEG-2 would satisfactorily scale up for this application and committed to use it rather than MPEG-3 – this lead to the quick demise of MPEG-3 [4].

MPEG remains active today, though it has broadened its focus beyond motion pictures. In October of 1998, *MPEG-4: Very Low Bit Rate Audio-Visual Coding* was released, providing a standard for multimedia applications (including specifications for low bit rate speech audio coding, video, still images, and text). MPEG-7, entitled *Multimedia Content Description Interface*, is scheduled for release in July 2001. Work on the latest standard, MPEG-21 *Multimedia Framework*, began in June 2000 [3 130].

2.2 OVERVIEW OF MPEG-2

The MPEG-2 standard contains 10 parts, the first five of which are analogous to the MPEG-1 standard (systems, video, audio, conformance testing, and software implementation). The sixth part, subtitled *Digital Storage Media Command and Control (DSM-CC)*, provides a framework for performing VCR-like functions on an MPEG bit stream (e.g. fast forward, pause). The remaining three parts describe a non-backwards compatible audio standard, higher-quality video extension (subsequently withdrawn due to lack of interest), a real-time interface for set-top box decoders, and conformance test requirements for DSM-CC (still under development) [3 128]. The following section gives a brief description of the structure of MPEG-2 systems, with technical details taken from the text of the standard [1].

The MPEG-2 systems standard specifies two stream formats for combining audio, video, and other data. First, it specifies the **Transport Stream**, which is appropriate for use in transmission environments where data errors are likely. Transport streams often multiplex many individual programs together, making it suitable for use in satellite or digital cable television applications. Second, the MPEG-2 systems standard defines the **Program Stream**, which is designed for use in data storage environments where bit errors are unlikely. Program Streams can carry only one program with a common time base (which can have many elementary audio, video, or private components), making it suitable for use in multimedia storage applications such as DVDs. The MPEG-2 systems hierarchy is illustrated in Figure 2-1 [1 §0].

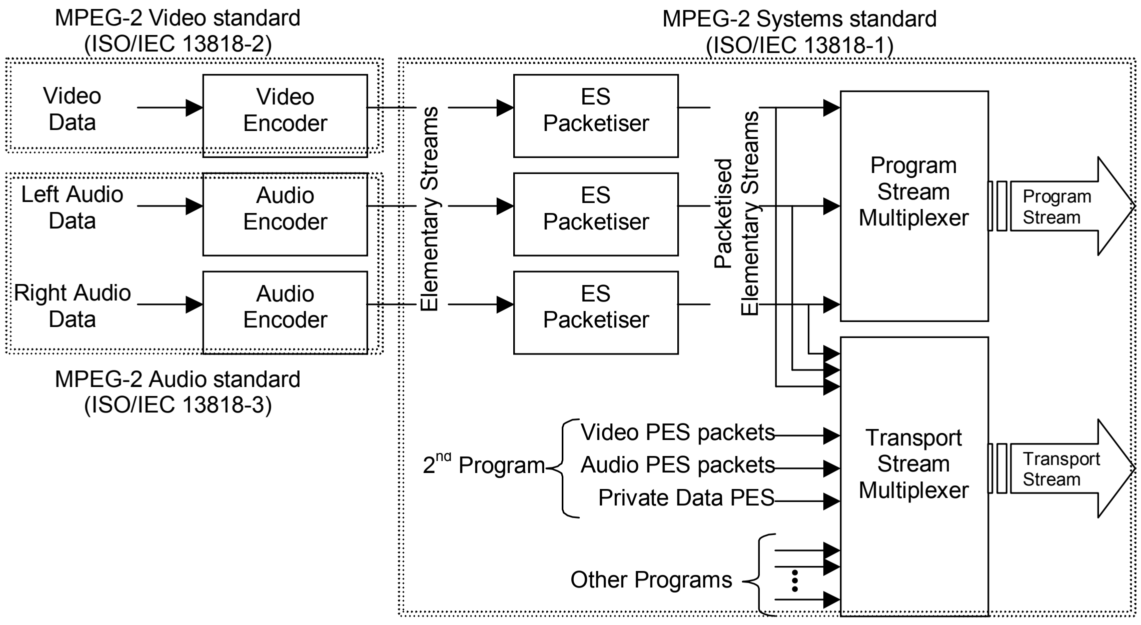


Figure 2-1: Overview of MPEG-2 Audio, Video, Systems Standards

At its lowest level, the MPEG-2 systems standard describes the process of transforming audio and video elementary streams (as output from the MPEG-2 standard's audio and video parts) into **Packetized Elementary Stream (PES) packets**. PES packets normally do not exceed 64 kilo-bytes in length, though the syntax allows for arbitrary length. All packets begin with a 4-byte packet_start_code, which includes a 3-byte prefix of 0x000001 (hexadecimal) and a single-byte stream_id ranging from 0xBC to 0xFF. This stream_id is used to distinguish between video and audio channels of a single program (e.g. multilingual audio).

PES packet headers also contain 33-bit Decoding Time Stamp (DTS) and Presentation Time Stamp (PTS) fields. These contain time values (with a resolution of 90 kHz) for when elementary stream content should enter the decoder's buffer, and be presented. The general structure of a

PES packet is given in Figure 2-2. A more thorough discussion of packet structure can be found in the MPEG specification [1 §2.4.3.7].

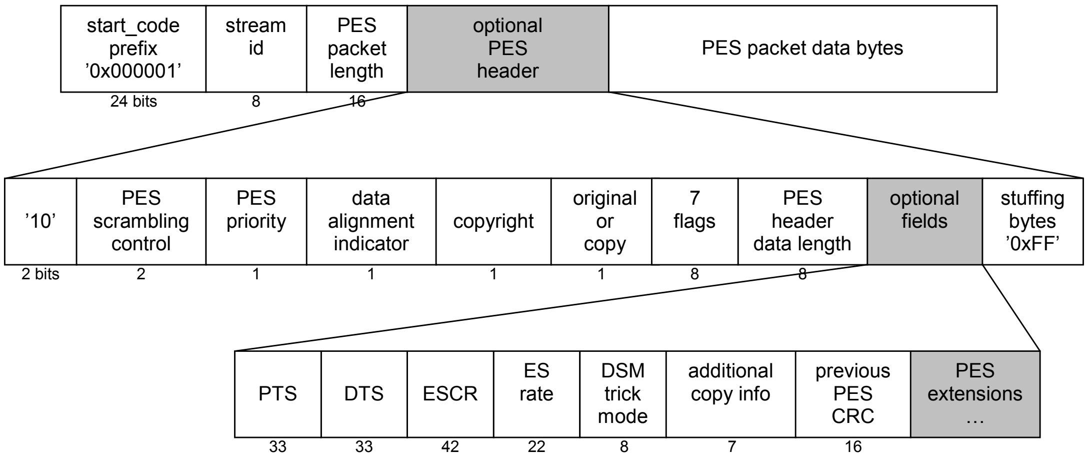


Figure 2-2: PES packet syntax diagram

MPEG-2 Transport Stream data is transmitted in 188-byte packets – the small packet size permits rapid synchronization and error recovery. A Transport Stream packet consists of a required header (4 bytes), an adaptation field (an optional extension to the header which does not exceed 26 bytes excluding private data), and a data payload (remainder of the 188 bytes). The payload may contain multiplexed elementary, program, and/or transport streams.

Transport Stream packets begin with an 8-bit sync_byte field used for decoder synchronization (the decoder must be able to locate the boundary between TS packets). The header of each Transport Stream packet also contains a 13-bit packet identifier (PID), used to differentiate between different programs (analogous to television stations) in the Transport Stream. The adaptation field contains a 42-bit program clock reference (PCR) indicating the intended arrival time, with 27 MHz resolution, of the current packet to the decoder. The general structure of the Transport Stream is shown in Figure 2-3 [1 §2.4.3].

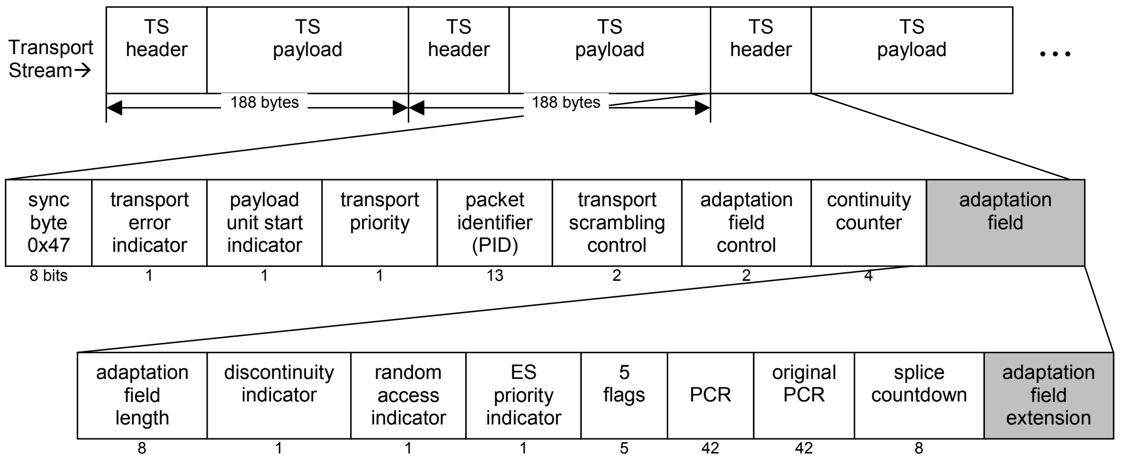


Figure 2-3: Transport stream syntax diagram

MPEG-2 Program Streams are divided into variable-length **packs**. All packs begin with a 14-byte header that includes a 4-byte pack_start_code (0x000001BA) and a 42-bit system clock reference (SCR) analogous to the PCR of Transport Streams. A system header must follow the first pack header of a Program Stream. It enumerates all elementary streams contained within the Program Stream, indexed by stream_id. These headers are followed by pack data consisting of a series of PES packets coming from any stream within the program. The general structure of a Program Stream is shown in Figure 2-4 [1 §2.5.3].

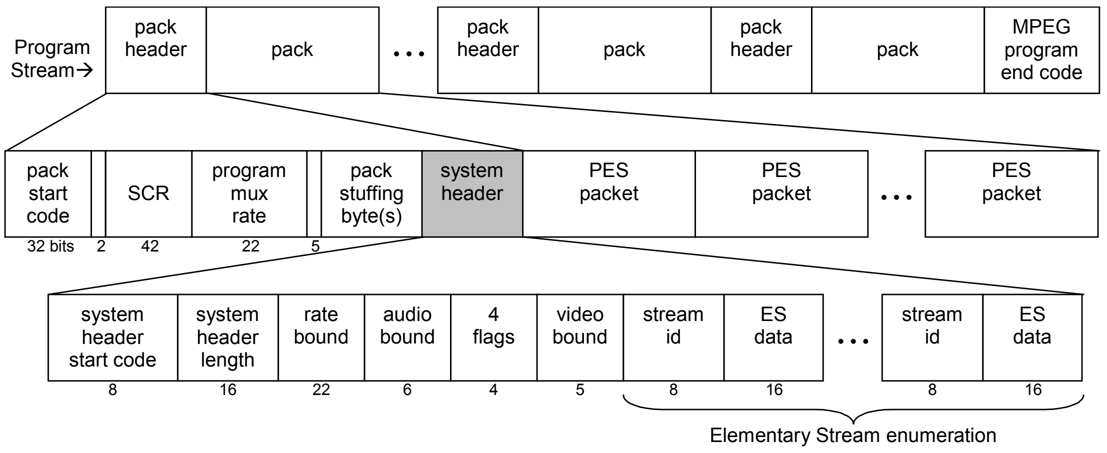


Figure 2-4: Program Stream syntax diagram

2.3 STRUCTURE OF MPEG-2 VIDEO

As stated earlier in §2.2, the MPEG-2 systems standard accepts audio and video elementary streams as inputs to its specification. These elementary streams arise from the coding procedures defined in the audio and video parts of the MPEG-2 standard. The following section contains a comprehensive structural description of a compliant video elementary stream, as defined in the MPEG-2 video standard [2 §6].

All MPEG-2 bit streams contain recurring access points known as **start codes**. The specification defines start codes as consisting of a prefix bit string of at least 23 '0'-bits followed by a single '1'-bit and an 8-byte start_code_value describing the nature of the data ahead. There should be enough zero-bits in a start code so that the data immediately following it is byte-aligned.

Start codes provide a mechanism for searching through a coded bit stream without fully decoding it (random access or "channel surfing") and for re-synchronization in the presence of bit errors. As hinted by the name, all MPEG-2 data structures begin with a start code, and they must never be emulated within the body of an MPEG-2 data structure. The specification mandates that marker '1'-bits appear in many structures to prevent this[2].

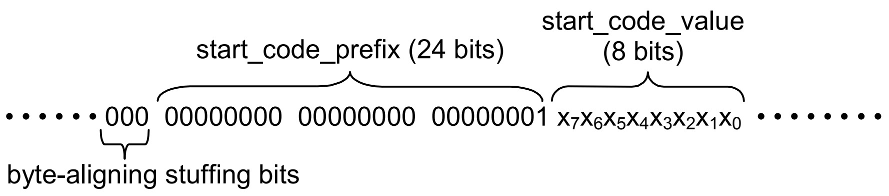

start_code_value (8 bits)
start_code_prefix (24 bits)
• • • • • • 000 00000000 00000000 00000001 x7x6x5x4x3x2x1x0 • • • • • • • •
byte-aligning stuffing bits

Figure 2-5: MPEG-2 video start code

The following four sections provide an overview of the structure of MPEG-2 video. **Video sequences**, and the headers associated with them, are explained in §2.3.1. Several optional extensions to the MPEG standard, which may apply to entire video sequences, are discussed in §2.3.2. The chapter concludes with descriptions of elements underlying video sequences – **groups of pictures** and **pictures** are discussed in §2.3.3 and §2.3.4.

2.3.1 VIDEO SEQUENCES

Sequence Header

The highest-level MPEG-2 video data structure is the autonomous video sequence. Video sequences are headed by a **sequence_header** [2 §6.3.3], identified with a start_code_value of 0xB3. A typical MPEG-2 multimedia file contains one video sequence, though for random access pur-

---

[2] An example of marker bits preventing start code emulation is found in the coding of 64-bit copyright identifiers. Lest the identifier contain a string of 23 or more '0' bits, the specification splits it into 22-bit pieces, each separated by a marker '1'-bit.

poses, the standard encourages encoders to repeatedly encode the sequence_header so they are spaced through a bit stream every few seconds. A **sequence_end_code** (with start_code_value 0xB7) signals the end of a video sequence.

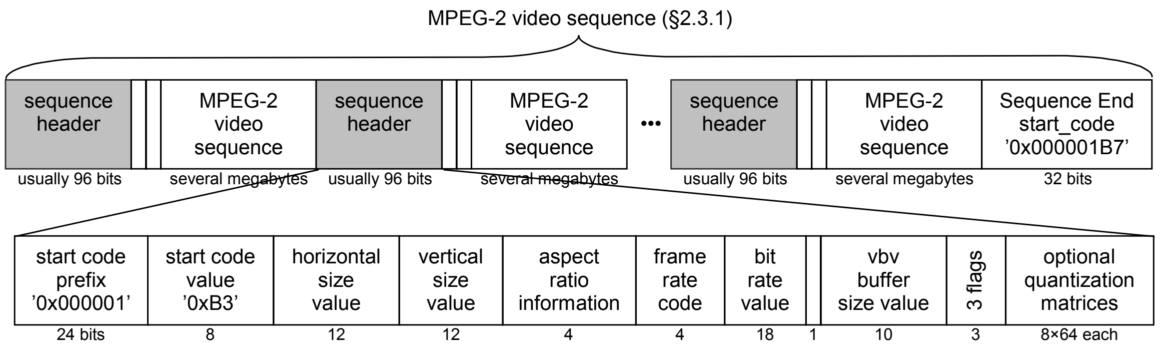


Figure 2-6: Sequence header syntax diagram

As shown in Figure 2-6, the sequence_header contains parameters applicable to an entire video sequence. The horizontal_size_value and vertical_size_value are unsigned integers indicating the dimension of all pictures in the video sequence. The frame_rate_code specifies that the intended (as opposed to the coded) video frame rate is one of the following values: 24, 25, 30, 50, or 60 fps[3]. The 4-bit aspect_ratio_information code indicates that pictures in the video sequence have a pre-defined aspect ratio of 4:3, 16:9, 2.21:1, or that the picture dimensions determine the aspect ratio (meaning pixels are perfect squares).

The sequence header also contains a bit_rate_value field, which generally specifies the average bit rate of an MPEG-2 bit stream in units of 400 bps. The vbv_buffer_size_value gives the size, in two-kilobyte units, of the video buffering verifier (vbv). The concept of the video buffering verifier is beyond the scope of this report – it relates to the memory buffer wherein just-decoded pictures are stored during the decoding process. Finally, the sequence_header contains flags for loading in user-defined 8×8 quantization matrices that override default values. This option is further discussed in §2.3.2.

Sequence Extension

The MPEG-2 specification also defines structures known as extension headers, signified by a start_code_value of 0xB5. A 4-bit extension_start_code_identifier, characterizing the nature of

[3] The frame rate values 23.976, 29.97, and 59.94 ($\frac{24000}{1001}$, $\frac{30000}{1001}$, and $\frac{60000}{1001}$) are also included for compatibility with various film digitizing processes.

the data in the extension header, immediately follows the start code. The MPEG-1 specification uses the same start code and header system as that used in MPEG-2. All fields introduced in MPEG-2 to support new functionality lie within MPEG-2 extension headers and outside the traditional MPEG-1 headers.

The MPEG-1 sequence_header, as described above, inadequately characterizes newer MPEG-2 video sequences. An MPEG-2 **sequence_extension** header [2 §6.3.5], signified by an extension_start_code_identifier of '0001', follows each MPEG-1 sequence_header in an MPEG-2 bit stream. The layout of a sequence_extension header is shown in Figure 2-7.

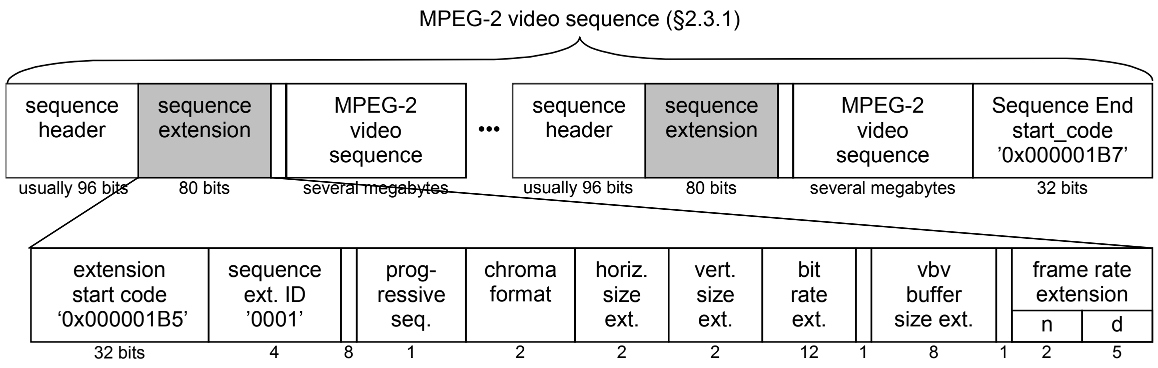


Figure 2-7: Sequence extension syntax diagram

The MPEG-2 standard extends the capabilities of MPEG-1 by supporting the encoding of pictures as a pair of interlaced fields, rather than a non-interlaced picture frame. The progressive_-sequence flag should be set to '1' if and only if the video sequence consists entirely of progressive (non-interlaced) pictures. The chroma_format field is a 2-bit code specifying the format of chrominance sampling (4:2:0, 4:2:2, and 4:4:4 are supported) – the chrominance/luminance color space is defined in §3.1.

The MPEG-2 sequence_extension header also contains fields that allow sequence_header parameters to exceed limits set out in the MPEG-1 standard. The two 2-bit horizontal/vertical size -extensions, the 12-bit bit_rate_extension, and the 8-bit vbv_buffer_size_extension fields are inserted in front of the most significant bits of their corresponding sequence_header parameters. For example, these extend the maximum possible picture dimension from 4095 pixels (MPEG-1) to 16 383 pixels (MPEG-2). Finally, the 2-bit frame_rate_extension_n and 5-bit frame_rate_-extension_d fields broaden the frame rate values possible with MPEG-1 by performing the multiplication shown in equation (2.1) on the frame_rate_value decoded from frame_rate_code in the sequence header:

$$frame\_rate = frame\_rate\_value \cdot \left(\frac{1 + frame\_rate\_extension\_n}{1 + frame\_rate\_extension\_d} \right) \quad \textbf{(2.1)}$$

Sequence Display Extension

The **sequence_display_extension** header [2 §6.3.6], as shown in Figure 2-8, indicated by an extension_start_code_identifier of '0010', is a header that can optionally follow each sequence_-extension. It specifies supplementary information about the target video display device – MPEG-2 compliant decoders can ignore and discard information in this header. Within the extension header, the 3-bit video_format code indicates the display type from which the video sequence was originally encoded (e.g. PAL, NTSC, SECAM). This is followed by three optional 8-bit codes describing color space characteristics of the video source. The header concludes with two 14-bit integers, display_horizontal_size and display_vertical_size, which specify the size of the intended display's active region in pixels. A display size smaller than that of the video picture (horizontal_size_value×vertical_size_value) indicates that only a portion of the picture should be on-screen. Conversely, a larger size indicates that the video should be displayed only on a portion of the active region.

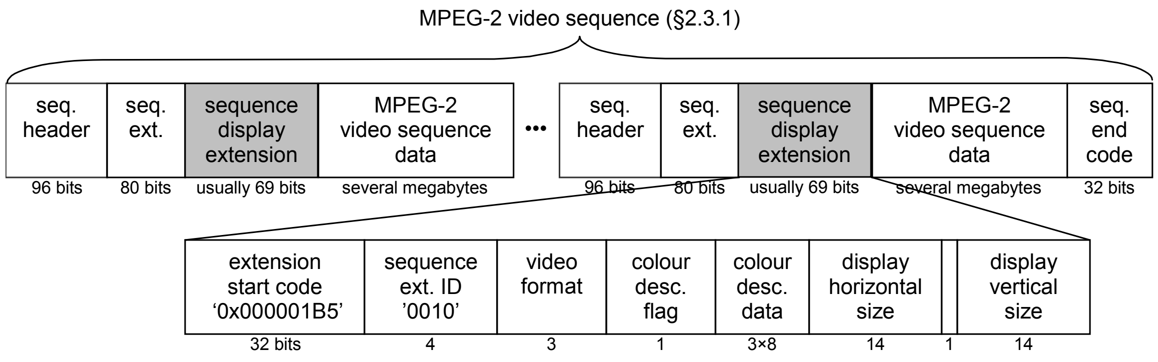


Figure 2-8: Sequence display extension syntax diagram

2.3.2 Other Extensions

Copyright Extension

The MPEG-2 specification mentions other optional extension headers applicable to an entire video sequence. The **copyright extension** header, indicated by an extension_start_code_-identifier of '0100', is shown in Figure 2-9. Setting the copyright_flag to '1' indicates that all data following the present copyright_extension until the next copyright_extension is copyrighted. The 8-bit copyright_identifier code identifies the copyright registration authority under which the work is registered, e.g. U.S. National Copyright Office. Finally, the 64-bit copyright_number is the unique identifier assigned to the copyrighted work by the registration authority.

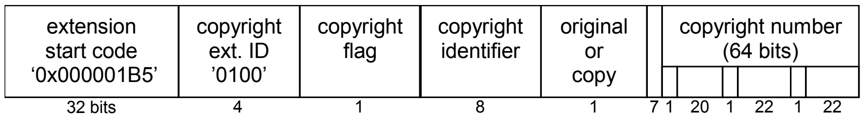


Figure 2-9: Copyright extension syntax diagram

Quantization Matrix Extension

Each time a decoder processes a sequence_header, an MPEG-2 video decoder resets the quantization matrices resident in its memory back to default values specified in the standard. The sequence_header may contain custom matrices replacing these default values, however, a **quant_-matrix_extension** header may contain matrices overriding both the MPEG-2 defaults and those in the sequence_header. The quant_matrix_extension extends the capability of the sequence_-header (and that of MPEG-1) by allowing the bit stream to contain separate matrices for chrominance and luminance color components.

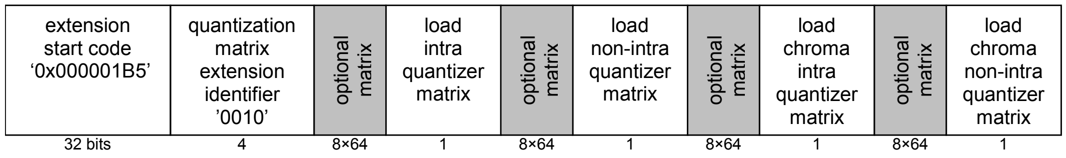

extension start code '0x000001B5'
quantization matrix extension identifier '0010'
optional matrix
load intra quantizer matrix
optional matrix
load non-intra quantizer matrix
optional matrix
load chroma intra quantizer matrix
optional matrix
load chroma non-intra quantizer matrix
32 bits
4
8×64
1
8×64
1
8×64
1
8×64
1

Figure 2-10: Quantization matrix extension syntax diagram

User Data Header

The **user data header**, identified by a start_code_value of 0xB2, is another regular MPEG-1/MPEG-2 construct that can be used for extending the capabilities of the specification. This creates opportunities for text, such as closed-captioning or commentary data. In order for this private information to be useful, both encoder and decoder must know the protocol behind it, requiring conformance with an additional MPEG-compliant standard.

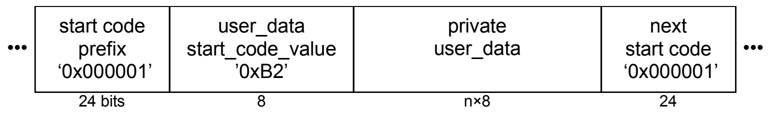

•••
start code prefix '0x000001'
user_data start_code_value '0xB2'
private user_data
next start code '0x000001'
•••
24 bits
8
n×8
24

Figure 2-11: User data syntax diagram

Scalable Extensions

A final type of extension to MPEG-1 is the set of MPEG-2 scalable extensions. Scalability allows video to be encoded in two separate layers: a base layer that can be decoded alone into meaningful video, and an enhancement layer containing picture data that improves the quality of the base layer. One possible application of scalability would be to transmit the base layer along a low-speed error-free channel and the enhancement over a less-reliable high-speed channel.

The **sequence_scalable_extension** header is used for partitioning a bit stream into these layers. It contains a code describing the form(s) of the scalability employed, including spatial, signal-to-noise ratio (SNR), and temporal scalability. Spatial scalability uses an enhancement layer to enlarge the base layer picture size, SNR scalability uses an enhancement layer to improve the picture quality of the base layer, and temporal scalability adds extra pictures to the base layer in order to increase the frame rate [2 §I.4.2].

2.3.3 Groups of Pictures

Video bit streams are generally divided into groups of pictures below the video sequence level, as shown in Figure 2-12. Each group is preceded by a **group_of_pictures_header** [2 §6.3.8], indicated by a start_code_value of 0xB8. The header contains a 25-bit time_code with hour, minute, and second fields, which is used for tracking video sequence progress similar to time counters on videocassette recorders. The header also contains flags indicating either that the group is autonomous or that pictures within the group rely on pictures from a neighboring group.

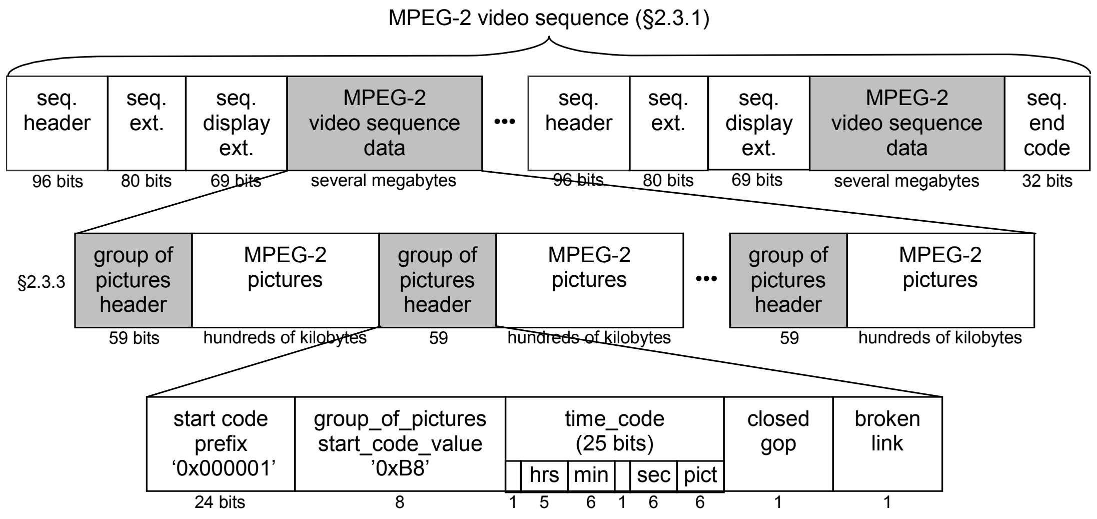


Figure 2-12: Group of pictures syntax diagram

The group_of_pictures_header is required by the MPEG-1 standard, however, it became optional with MPEG-2 because its contents have no effect on the decoding process. Most MPEG-2 encoders, however, include it to improve bit stream organization.

2.3.4 PICTURES

Picture Header

As the name suggests, groups of pictures consist of pictures. Each MPEG-2 picture is headed by a **picture_header** structure [2 §6.3.9], shown in Figure 2-13, with start_code_value 0x00. The 10-bit temporal_reference integer is used as a picture counter, to be incremented for each encoded picture (two interlaced fields count as a single picture), and reset to zero after each group_of_-pictures_header. The vbv_delay field contains instantaneous bit rate information used by the video buffering verifier.

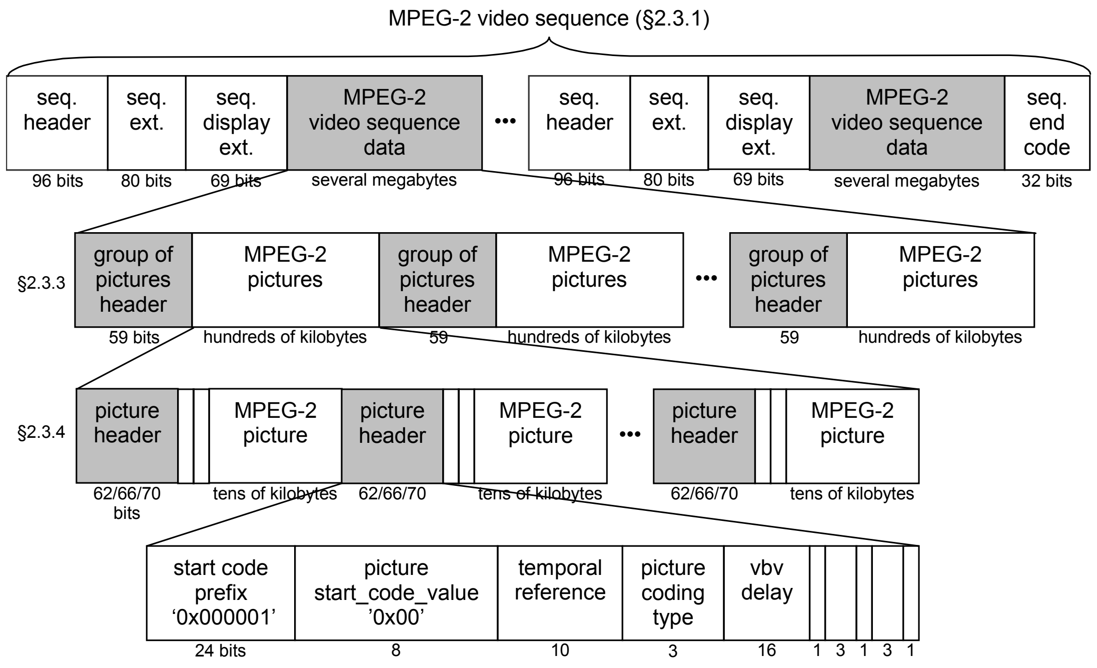


Figure 2-13: Picture header syntax diagram

The most important field in a picture_header is the 3-bit picture_coding_type code, used for classifying pictures as **intra-coded (I)**, **predictive-coded (P)**, or **bi-directionally predictive-coded (B)**[4]. I-pictures contain full picture information and are the least compressed type of picture – they are similar to a JPEG image. P-pictures reduce the coding of redundant picture regions by referencing sections of the previous I-picture or P-picture. B-pictures take this concept a step further by referencing the previously displayed I- or P-picture and the next I- or P-picture to be displayed.

[4] The MPEG-1 specification also allowed for rarely-used DC-coded pictures (D-pictures), which consisted of single-color blocks. They were not carried forward to the MPEG-2 standard. [4]

A video bit stream should contain I-pictures often enough that predictive errors arising from P- and B- pictures do not accumulate. By convention, most encoders make every twelfth picture an I-picture; this means they occur every 0.4 seconds (30 Hz frame rate), adequate for random access requirements. This leads to the common picture organization, shown Figure 2-14.

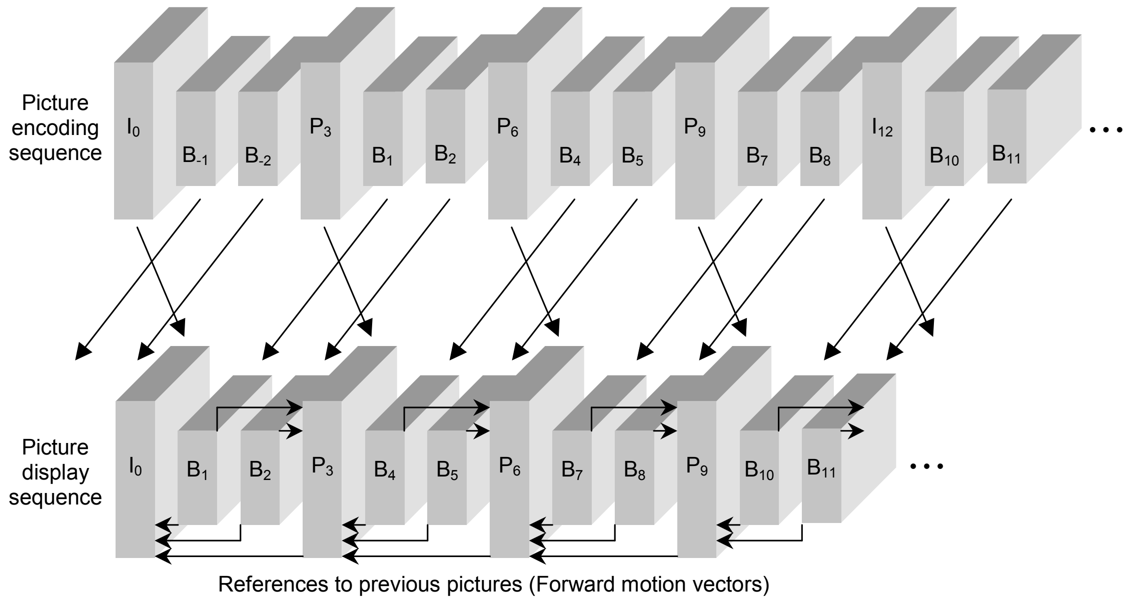


Figure 2-14: Typical picture sequencing

Figure 2-14 also illustrates that the bit stream picture ordering is not the same as the display order. B-pictures, which rely on past and future pictures, must be encoded after both reference pictures.

Picture Coding Extension

The MPEG-2 standard appends a **picture_coding_extension** [2 §6.3.10], containing picture information not found in MPEG-1, after each MPEG-1 picture_header. The picture_coding_-extension header is identified by an extension_start_code_identifier of '1000' – its general structure is shown in Figure 2-15.

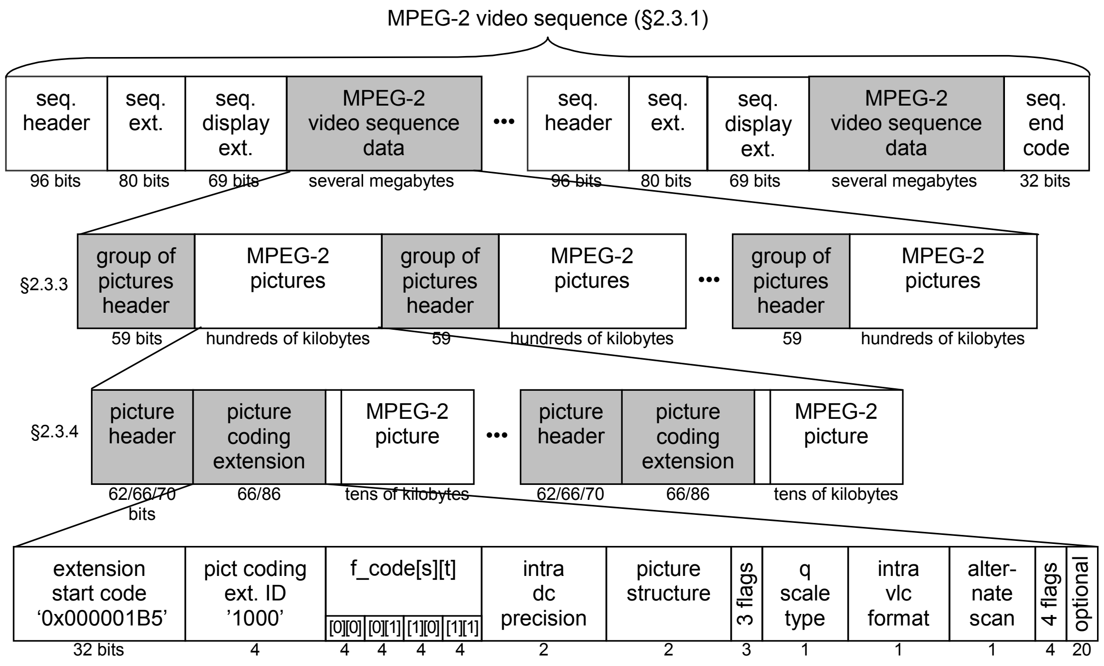


Figure 2-15: Picture coding extension syntax diagram

The f_code fields are 4-bit integers used for determining motion vector precision (their role in P- and B-picture references to previous/next pictures is summarized later in §3.2.2). The 2-bit picture_structure field code indicates that the current picture is encoded either as a full non-interlaced picture or as the top or bottom field of an interlaced picture. The intra_dc_precision, q_scale_type, intra_vlc_format, and alternate_scan flags control the image decoding method (these fields are discussed in §3.3).

Picture Display Extension

An optional **picture_display_extension** [2 §6.3.12] may follow the picture_coding_extension just as a sequence_display_extension optionally follows the sequence_extension. The picture_-display_extension header contains a pair of 16-bit signed integers, frame_centre_horizontal_-offset and frame_centre_vertical_offset, for each encoded picture or interlaced field following the picture header. These two integers specify the intended offset between the center of the display's active region and the center of the displayed picture. The main application for this is clipping irrelevant sidebar regions from 16:9 wide-screen pictures into the standard 4:3 format.

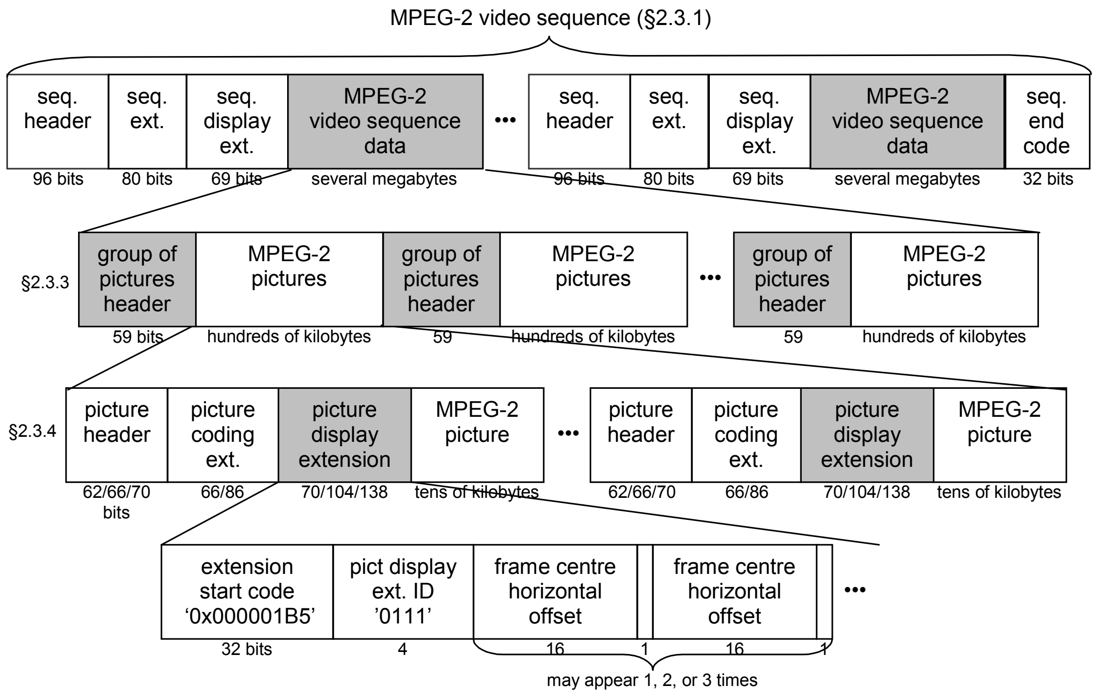


Figure 2-16: Picture display extension syntax diagram

Summary

The preceding chapter provided a brief description of MPEG and MPEG-2, and a more thorough overview of the structure of an MPEG-2 video bit stream. This included discussion of the headers associated with video sequences, the highest syntactic structure in MPEG-2 video. Groups of pictures and pictures (structures that lie beneath video sequences) were also explained in detail.

3 MPEG-2 VIDEO ENCODING

The MPEG-2 video specification sets forth a standard for a bit stream, and defines a compatible decoder as being a device capable of correctly processing this bit stream. There is no elaboration on encoding techniques – they are left for private developers to invent. The following section, adapted from §6.3.15 - §6.3.16 of the MPEG-2 video specification [2], describes the general procedure for compressing and encoding an image into an MPEG-2 picture.

3.1 YCbCr Color Space

The first stage in encoding an MPEG-2 picture is to convert the source image into the same color space as the target MPEG-2 video sequence. As mentioned in §2.3.1, MPEG-2 uses television's luminance/chrominance (YCbCr) color space as opposed to the CRT display's red/green/blue (RGB) color space. Unless the sequence_display_extension specifies otherwise, the following system of equations[5] should be used for converting between the two color spaces:

$$\begin{cases} Y &= 0.1818R + 0.612G + 0.06168B + 16 \\ Cb &= -0.1006R - 0.3299G + 0.4375B + 128 \\ Cr &= 0.4375R - 0.3973G - 0.04025B + 128, \end{cases} \qquad \textbf{(3.1)}$$
$$where \;\; 0 \le R, G, B, Y, Cb, Cr \le 255 .$$

Applying equations (3.1) to an RGB image yields a result in the YCbCr color space with chrominance format 4:4:4. Because chrominance (Cr, Cb) data is less relevant to image integrity than luminance (Y) data, most images undergo a further down-sampling of their chrominance components into the 4:2:2 format for studio editing or the 4:2:0 format for mainstream consumer appliances. As shown in Figure 3-1, an image containing $n \times n$ luminance samples contains $n \times n$, $\frac{n}{2} \times n$, or $\frac{n}{2} \times \frac{n}{2}$ chrominance samples in the 4:4:4, 4:2:2, and 4:2:0 formats, respectively.

[5] taken from ITU-R Recommendation ITU-R BT.709 (1990). [2 §6.3.6]

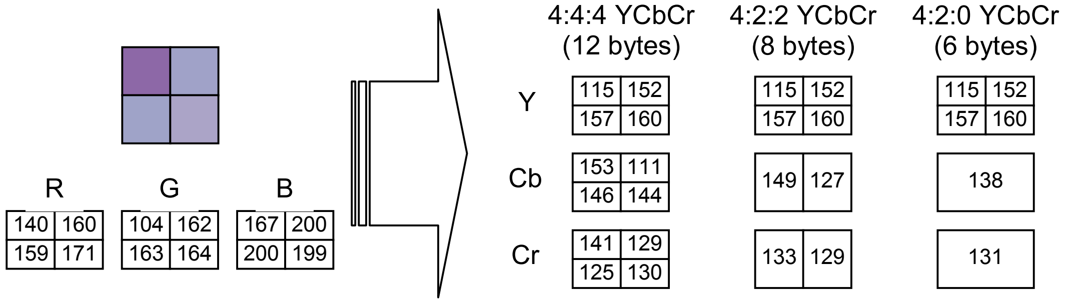


Figure 3-1: Conversion between the RGB and YCbCr color spaces

3.2 Picture Slices and Macroblocks

Once a picture is in the proper color space, it can be divided into **slices** and **macroblocks**. The MPEG-2 specification defines a macroblock as an image section consisting of a 16×16 matrix of luminance samples and the corresponding chrominance data. A slice is defined as a series of consecutive macroblocks, where macroblocks are ordered left-to-right and then top-to-bottom. Figure 3-2 depicts an image divided into macroblocks and several slices.

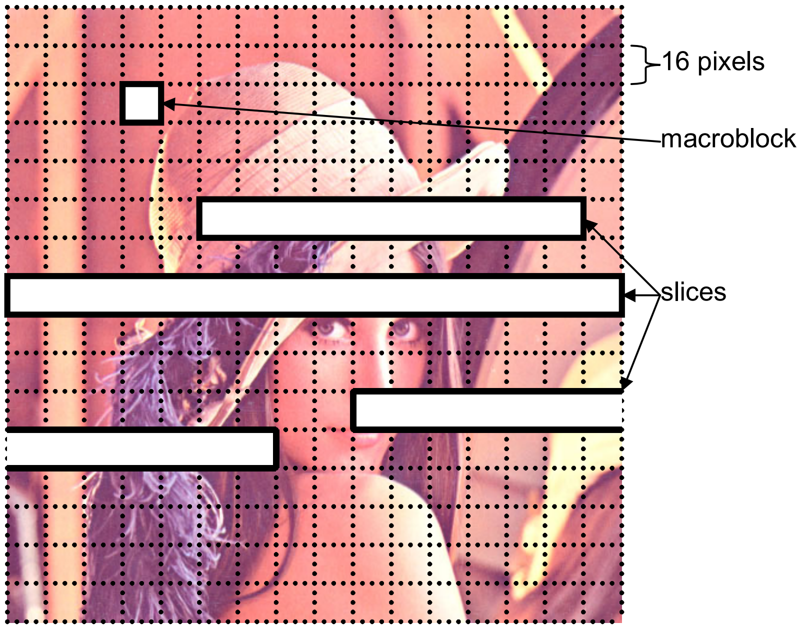


Figure 3-2: Slices and Macroblocks

Picture slices are inserted into an MPEG-2 video elementary stream after their picture's picture_-header and picture_coding_extension structures. Slices are organized in a manner similar to other headers in that they begin with a start code and contain some header information. The start_-

code_value for a slice ranges between 0x01 and 0xAF, and indicates in which macroblock row of the picture the slice begins. A 3-bit field, slice_vertical_position_extension, should appear after the start_code_value for pictures with a height greater than 2800[6]. These bits should be inserted as the most significant bits of an expanded macroblock row counter. Figure 3-3 shows the structure of a slice in the context of an MPEG-2 picture.

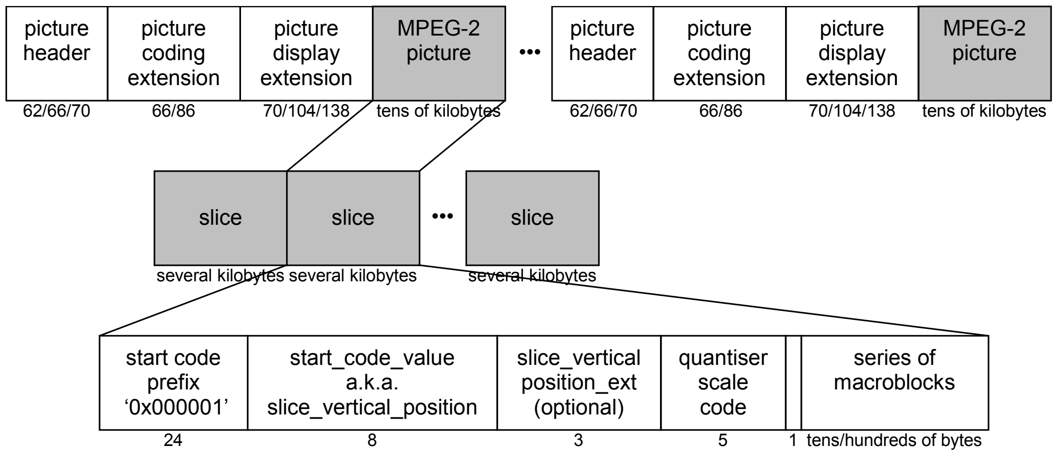


Figure 3-3: Slice syntax diagram

A picture slice also contains a 5-bit **quantiser_scale_code** integer. This important parameter is used in a later stage of the encoding/decoding process and is thoroughly discussed in §3.3.2.

3.2.1 Macroblocks

After the brief header information, an MPEG-2 picture slice contains a sequence of ordered macroblocks. Macroblocks are not randomly accessible – picture slices are the lowest level MPEG-2 video structure using start code access points. In addition, macroblocks (and structures below them) make frequent use of variable-length codes as a means of removing redundant bits from the video bit stream. Higher-level header structures can afford the luxury of fixed-length fields, as those headers represent a negligible percentage of bit stream bandwidth.

[6] Note that 2800 = 16×0xAF, where 16 is the height of a macroblock

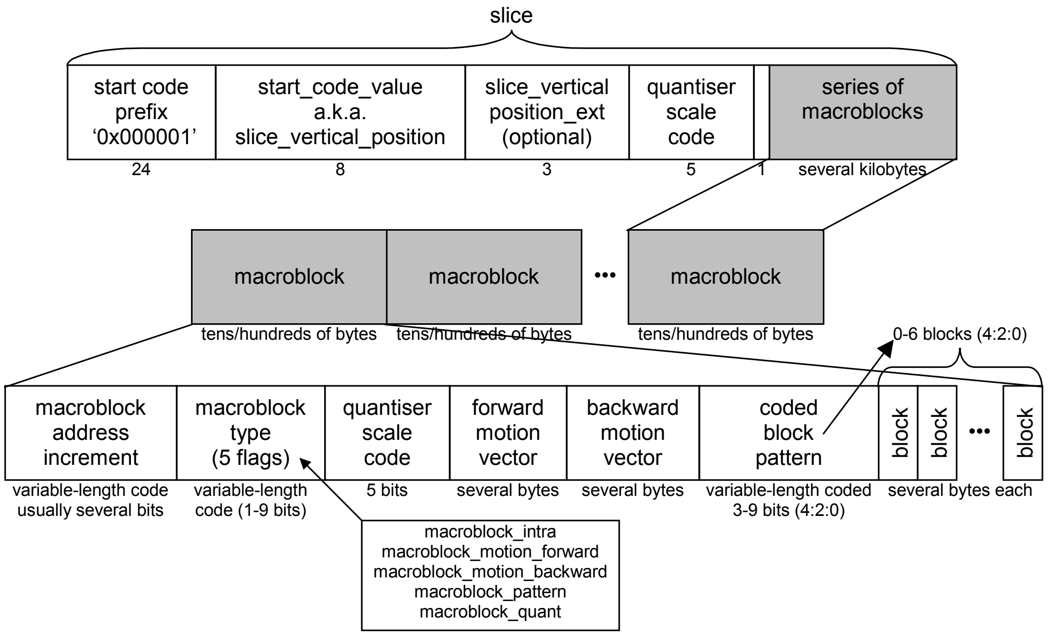


Figure 3-4: Macroblock syntax diagram

Macroblocks begin with the macroblock_address_increment variable-length code, which establishes the horizontal position of the macroblock (the slice start code establishes its vertical position). The most common scenario, a macroblock_address_increment code of '1', means that the forthcoming macroblock lies immediately beside the previously coded macroblock. Longer variable-length codes can be used to represent an arbitrary number of skipped macroblocks. When an encoder chooses to skip over a macroblock in a picture, the bit stream's eventual decoder displays the macroblock from the previous picture with identical coordinates.

Macroblock Types

The next variable-length encoded field occurring in a macroblock is the crucial **macroblock_type** code. It consists of five flags that specify which MPEG-2 video compression features are used for the current macroblock. Having the **macroblock_intra** flag set signifies that the macroblock is intra-coded (*i.e.* it makes no reference to other macroblocks from other pictures). If the **macroblock_motion_forward** and/or **macroblock_motion_backward** flags are set, the macroblock references a macroblock from another picture using motion vectors. The **macroblock_pattern** flag is often used along with the macroblock_motion_forward/backward flags to code a correction to the macroblock resulting from the motion prediction.

Eligibility to use the various compression features incorporated in the macroblock_type code depends on what type of picture (I, P, or B) the macroblock resides in. Intra-coded (I-) pictures are required to contain a full image that can be decoded without reliance on neighboring pictures;

macroblocks within an I-picture are prevented from using the macroblock_motion_-forward/backward and macroblock_pattern flags. Similarly, macroblocks from P-pictures are limited to unidirectional references to the previous P-picture's macroblocks — they cannot use the macroblock_motion_backward flag. All picture types are permitted to contain macroblocks with the macroblock_intra flag, though intra macroblocks typically only appear as a last resort in P- and B-pictures. Table 3-1 lists the possible combinations of macroblock_type flags with picture types – '×' denotes a "don't-care" condition.

Table 3-1: Permitted Macroblock Types

	✓ macroblock_motion_forward	✓ macroblock_motion_backward	✓ macroblock_pattern	✓ macroblock_intra	✓ macroblock_quant
I-, P-, B-pictures				✓	×
P-pictures			✓		×
P-, B-pictures	✓				
P-, B-pictures	✓		✓		×
B-pictures	×	✓			
B-pictures	×	✓	✓		×

The fifth flag from the macroblock_type code is the **macroblock_quant** flag. It can be set whenever the macroblock_intra or macroblock_pattern flags are set. If the macroblock_quant flag is set, a 5-bit integer, **quantiser_scale_code**, follows the macroblock_type variable-length code as shown in Figure 3-4. This field has the identical function to the quantiser_scale_code found in a picture slice header. In fact, when present, this quantiser_scale_code replaces the slice header code's value. The quantiser_scale_code integer is explained later in §3.3.2.

3.2.2 Macroblock Motion Compensation

As many as two motion vectors may appear after the quantiser_scale_code field, depending on the values of the macroblock_motion_forward/backward flags. The current macroblock can reference a similar macroblock in the previous and/or next I-/P-picture to conserve bandwidth by avoiding fully intra-coding each macroblock. Motion vectors separately code their vertical and horizontal components with differing precision, taking advantage of the fact that horizontal motion tends to be more pronounced than vertical motion in typical video sequences. Each component consists of a variable-length motion_code (biased in favor of spatially close macroblocks) for roughly locating the referenced macroblock, and a motion_residual integer of f_code[7] bits that has a finer resolution of a half-pixel. In addition, these vectors are coded as an offset of the previous macroblock's motion vector, reflecting the reality that clusters of macroblocks tend to move in concert.

Macroblocks from B-pictures man contain forward-pointing motion vectors, backward-pointing motion vectors, or both. In the latter case, where both the macroblock_motion_forward and macroblock_motion_backward flags are set, the resulting macroblock should be an interpolation of both predictions – this effectively smoothes video motion. Conversely, the ability to select which direction (forwards or backwards) motion vectors should reference overcomes the problem of momentary object occlusion in the picture. For a more thorough discussion of motion compensation techniques, consult §7.6 of the MPEG-2 video specification [2].

If the macroblock_pattern flag is set, the coded_block_pattern code appears next in a macroblock as shown in Figure 3-4. Motion compensation techniques contain inaccuracies, and therefore macroblocks cannot simply be lifted from one picture and directly pasted into another. The motion prediction error for a macroblock is encoded similar to an intra macroblock, with the exception that the coefficients are relative offsets from the prediction instead of absolute pixel values. The coded_block_pattern variable-length code appears in the macroblock header if the macroblock_pattern flag is set; it directs which macroblock sections (blocks) have those additional error coefficients (offsets) coded. It should also be noted that this field may appear in P-pictures without any motion compensation. In this case, it is assumed that no motion has taken place and the predicted macroblock is taken from the previous picture at the same spatial coordinates.

3.3 BLOCKS

MPEG-2 video pictures are divided into slices comprised of 16×16 macroblocks. Beyond this, the lowest-level syntactic structure used in MPEG-2 video is the **block**. Blocks result from macroblocks being further divided along spatial and luminance/chrominance component lines into 8×8 matrices with 8-bit coefficients. They are encoded sequentially into the bit stream after macroblock header information, motion vectors, and coded_block_pattern codes. Depending on the

[7] f_code is an integer between one and nine coded into each picture_header. There are four separate f_code fields for vertical/horizontal and forward/backward motion vectors.

chroma_format (4:2:0, 4:2:2, or 4:4:4) of the video sequence, a macroblock may contain 6, 8, or 12 ordered blocks, respectively. The decomposition of macroblocks into blocks and the order these blocks appear in the bit stream are shown in Figure 3-5.

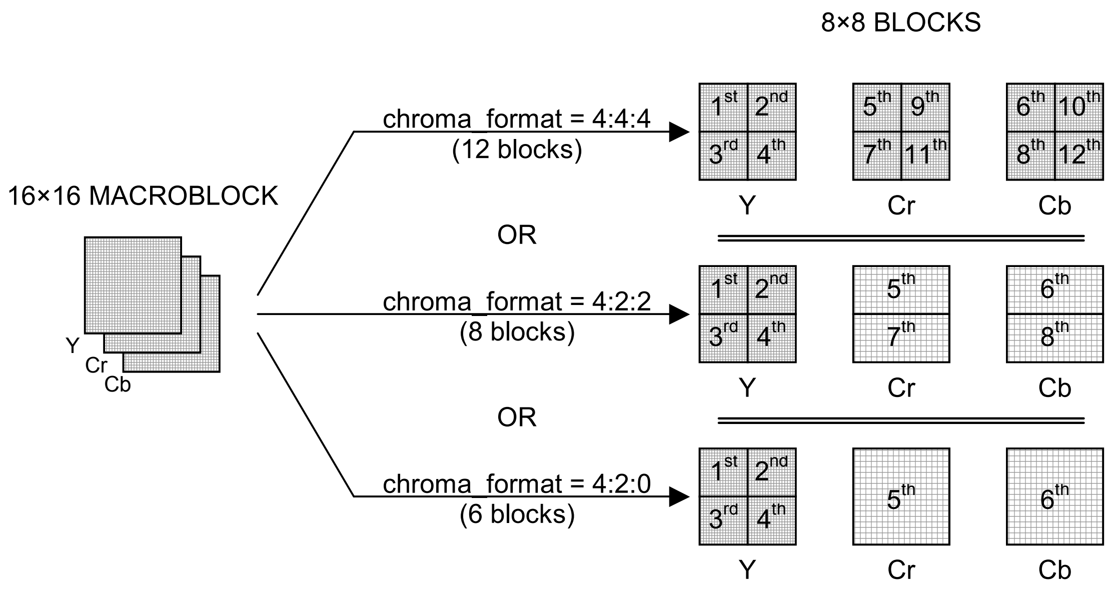


Figure 3-5: Macroblocks and Blocks

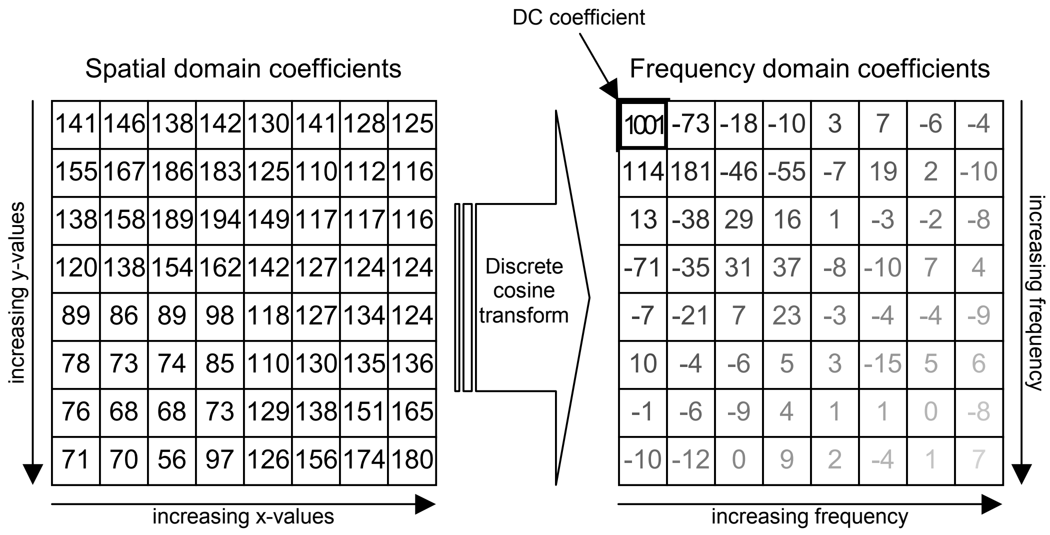
Intra-coded macroblocks should have all 6, 8, or 12 component blocks encoded in the bit stream at the end of the macroblock structure. Non-intra macroblocks with coded prediction errors (macroblock_pattern flag set) should encode these offsets into the block structure and then place these at the end of the macroblock structure, as intra macroblocks do with their absolute pixel values. As the coded_block_pattern field stipulates, blocks are usually missing from the encoded sequence – these represent macroblock sections where the prediction error is insignificant.

Rather than inefficiently occupying 64 bytes each, blocks undergo a multi-stage compression process while being inserted into the video bit stream. Removing redundant and irrelevant data from blocks by this process can often reduce their size to just a few bytes each. The next sections briefly describe the compression procedure (the reader is encouraged to consult §7.1 - §7.5 of the MPEG-2 video specification [2] for a more thorough treatment).

3.3.1 DISCRETE COSINE TRANSFORM

The first step in the compression of an 8×8 block containing raw pixel data is to perform a discrete cosine transform on it. This turns an 8×8 block of 8-bit coefficients in the spatial domain into an 8×8 block of 12-bit coefficients in the frequency domain. Equation (3.2), taken from Annex A of the specification [2], performs a two-dimensional discrete cosine transform on a matrix with spatial coordinates (x, y) and frequency domain coordinates (u, v).

$$\begin{aligned} F(u,v) &= \tfrac{1}{4}C(u)C(v)\cdot\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cdot\cos\left(\tfrac{(2x+1)u\pi}{16}\right)\cdot\cos\left(\tfrac{(2y+1)v\pi}{16}\right) \\ & u,\, v,\, x,\, y = 0, 1, 2, \ldots 7\ ; \quad C(n) = \begin{cases} \frac{1}{\sqrt{2}},\ n = 0 \\ 1,\ n \neq 0 \end{cases} \end{aligned} \tag{3.2}$$



141	146	138	142	130	141	128	125
155	167	186	183	125	110	112	116
138	158	189	194	149	117	117	116
120	138	154	162	142	127	124	124
89	86	89	98	118	127	134	124
78	73	74	85	110	130	135	136
76	68	68	73	129	138	151	165
71	70	56	97	126	156	174	180

1001	-73	-18	-10	3	7	-6	-4
114	181	-46	-55	-7	19	2	-10
13	-38	29	16	1	-3	-2	-8
-71	-35	31	37	-8	-10	7	4
-7	-21	7	23	-3	-4	-4	-9
10	-4	-6	5	3	-15	5	6
-1	-6	-9	4	1	1	0	-8
-10	-12	0	9	2	-4	1	7

Figure 3-6: Spatial domain and frequency domain blocks

Figure 3-6 shows the discrete cosine transform performed on a sample 8×8 spatial domain block. The entire compression process will be performed on this sample block in the remaining sections of this chapter.

3.3.2 QUANTIZATION

All 64 coefficients of a spatial domain picture block are of equal importance – this is not true for the corresponding frequency domain block. The human eye has difficulty perceiving high frequency image components in a similar way that the ear has difficulty detecting noise near and above 20 kHz. The MPEG-2 video specification compresses pictures by removing irrelevant and largely unnoticeable image data, so the higher-frequency block coefficients situated in the lower right corner are obvious candidates for cutbacks. The frequency domain block coefficients in Figure 3-6 are shown with text boldness proportional to their importance.

The standard achieves this data reduction by quantizing coefficients, *i.e.* reducing their precision by dividing them by large integers. The specification provides for a set of 8×8 quantization matrices containing the integers by which each corresponding block coefficient is divided. The default matrices for intra and non-intra blocks are shown in Figure 3-7.

Intra-coded blocks
macroblock_intra = '1'
macroblock_pattern = '0'

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

Non-intra coded blocks
macroblock_intra = '0'
macroblock_pattern = '1'

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

Figure 3-7: Default Quantization matrices for intra and non-intra blocks

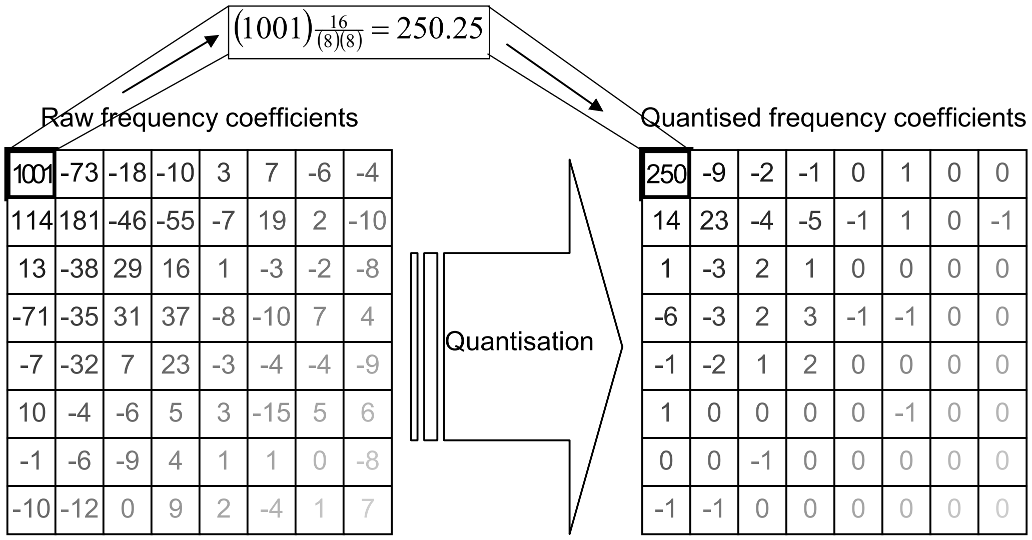
Intra blocks, which contain transformed absolute pixel values, are quantized to a much greater degree (especially at high frequencies) than non-intra blocks, which contain small offsets correcting motion vector block predictions. The quant_matrix_extension header, referred to in §2.3.2, can be used to override the default values for these matrices. In addition, the extension header allows the encoder to specify a separate pair of intra/non-intra matrices for chrominance and luminance blocks if the 4:2:2 or 4:4:4 chrominance format is in use.

The MPEG-2 video standard also specifies a second stage of quantization, where all 64 coefficients are further divided by a scalar. The integer divisor, quantiser_scale, is determined by decoding the 5-bit quantiser_scale_code field found in the slice and macroblock headers according to Table 3-2. The q_scale_type flag is located in the picture_coding_extension. The non-linear step size associated with a set q_scale_type flag is supposedly an improvement over the linear step size (q_scale_type = '0') from MPEG-1.

Table 3-2: Interpretation of the quantiser_scale_code field

quantiser_scale_code	quantiser_scale	
	q_scale_type=0	q_scale_type=1
1	2	1
2	4	2
3	6	3
4	8	4
⋮	⋮	⋮
15	30	22
16	32	24
17	34	28
18	36	32
19	38	36
20	40	40
⋮	⋮	⋮
29	58	96
30	60	104
31	62	112

Figure 3-8 applies the quantization stage to the sample block from the previous section, assuming quantiser_scale_code = 4. All coefficients are multiplied by 16 before the quantization step – this preliminary correction is further discussed in §4.1. For example, in Figure 3-8 the DC coefficient 1001 is multiplied by 16 and divided by its quantization matrix coefficient (8) and quantiser_scale (8), with a result of 250.


$(1001)\frac{16}{(8)(8)} = 250.25$
Raw frequency coefficients
Quantised frequency coefficients
Quantisation

Raw frequency coefficients

1001	-73	-18	-10	3	7	-6	-4
114	181	-46	-55	-7	19	2	-10
13	-38	29	16	1	-3	-2	-8
-71	-35	31	37	-8	-10	7	4
-7	-32	7	23	-3	-4	-4	-9
10	-4	-6	5	3	-15	5	6
-1	-6	-9	4	1	1	0	-8
-10	-12	0	9	2	-4	1	7

Quantised frequency coefficients

250	-9	-2	-1	0	1	0	0
14	23	-4	-5	-1	1	0	-1
1	-3	2	1	0	0	0	0
-6	-3	2	3	-1	-1	0	0
-1	-2	1	2	0	0	0	0
1	0	0	0	0	-1	0	0
0	0	-1	0	0	0	0	0
-1	-1	0	0	0	0	0	0

Figure 3-8: Sample block quantization

3.3.3 Scanning and Run-Length Encoding

The quantization step removes data irrelevant to picture integrity from blocks. The scanning/run-length encoding step re-arranges the quantized data in a format that reduces data redundancy in the bit stream. The reader should notice that many higher-frequency coefficients in the block are zero, and that most non-zero coefficients are concentrated near the DC coefficient in the upper-left corner of the block. In preparation for run-length encoding, the 8×8 block should be scanned into a 64-entry array in a zigzag fashion, as shown on the sample block in Figure 3-9. An alternative scanning order, activated when the alternate_scan flag in the picture_coding_extension is set, was introduced to MPEG-2 to improve performance with interlaced video [4].

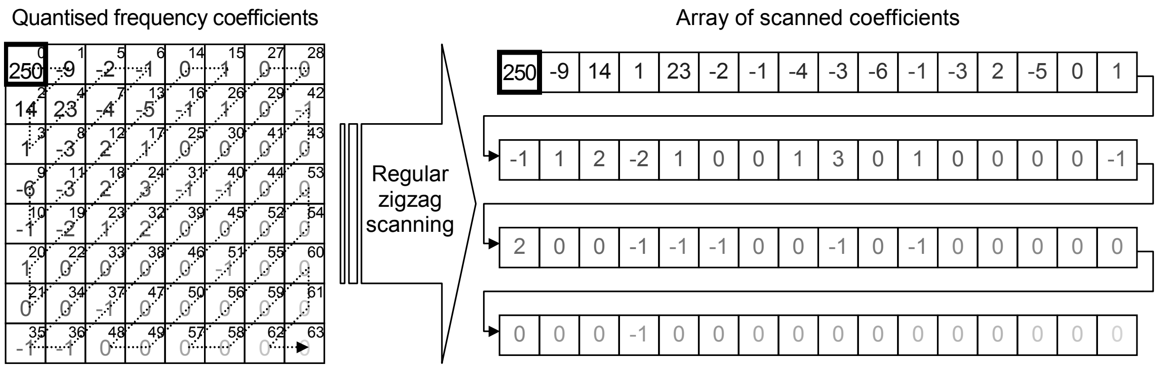


Figure 3-9: Sample block zigzag scanning

The block, which has now become a 64-entry array, should next be manipulated using a *modified* run-length encoding algorithm transforming it into a series of integer pairs consisting of a *run* component and a signed *level* component. The *run* integer represents the number of zero entries preceding a *level* value in the array. The final run of zero entries in the array ends at the 64th block entry and not with a signed *level* – this reality is encoded as *End of Block*. This syntax is illustrated in Figure 3-10, which applies the algorithm to the scanned sample block.

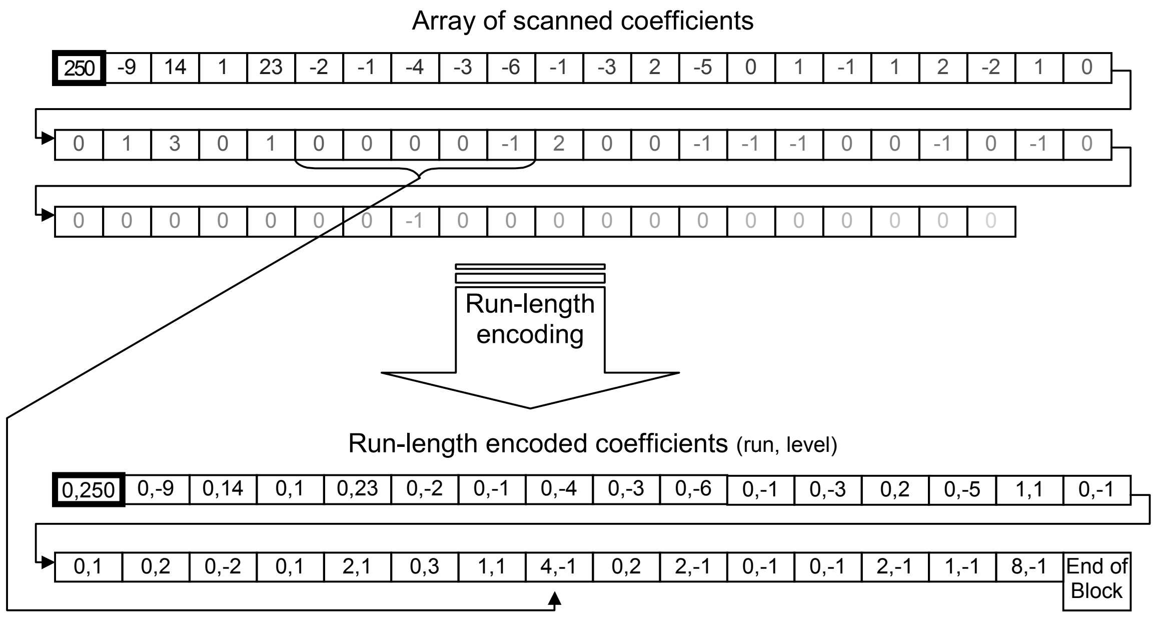


Figure 3-10: Sample block run-length encoding

3.3.4 VARIABLE-LENGTH ENCODING

The final stage of processing block data is to format the array run-length encoding into a bit sequence for insertion into the video bit stream. Certain combinations of *run* and *level* are statistically common, and table B-14[8] in the MPEG-2 video specification [2] assigns space-saving variable-length codes for 113 of these combinations. Provision is made for encoding the other 261 000 statistically rare combinations – the bit string '000001' is an escape code signaling that a 6-bit fixed-length *run* field and a 12-bit fixed length *level* field follow. The *End of Block* marker is encoded with the shortest variable-length code of all: '10'.

The DC (top-left) coefficient of intra-coded blocks strongly echoes the general (or mean) color level of the block. The MPEG-2 specification takes advantage of the fact that many pictures contain relatively uniform color levels by not encoding the absolute DC value of a block. Instead, the signed offset from the DC coefficient in the previously processed block (of the same luminance/chrominance channel) is encoded. Reflecting the importance of a precise DC coefficient

[8] Table B-15 contains an alternate listing of variable-length codes, improved since MPEG-1, suitable for intra-coded macroblocks. This table should be used if the picture_coding_extension flag intra_vlc_format is set.

offset, it is encoded outside the previously mentioned variable-length coding scheme, and treated as an integer field with a length determined by a preceding variable-length code[9].

Figure 3-11 shows the MPEG-2 variable-length coding scheme applied to the sample block from the previous sections. The sample block would now be ready for insertion into the MPEG-2 video elementary stream.

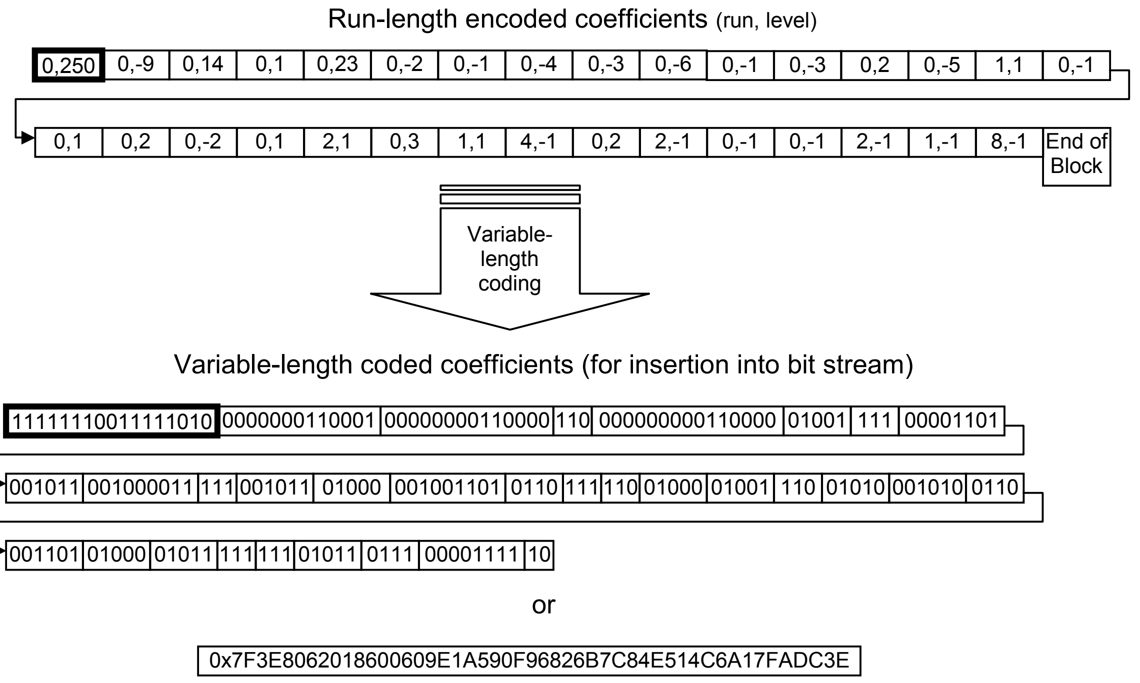


Figure 3-11: Variable-length encoding of sample block coefficients

Summary

The proceeding chapter described the procedure for encoding a picture into an MPEG-2 video bit stream. The first step, conversion into the chrominance/luminance (YCbCr) color space, was briefly considered. The decomposition of a picture into slices and macroblocks, along with macroblock motion compensation techniques, was reviewed next. The chapter concluded with a description of the compression procedure for blocks – this algorithm was able to compress a high-entropy 64-byte sample block into a bit stream occupying less than 25 bytes.

[9] The 2-bit intra_dc_precision code found in the picture_coding_extension header refers to the precision of the absolute DC value and not the offset DC value.

4 MPEG-2 VIDEO TRANSCODING

The process of converting between different compression formats and/or further reducing the bit rate of a previously compressed signal is known as transcoding. Transcoding involves partial decoding of a compressed bit stream, but stops short of full decoding because, in the case of video, there is no need to display the video to a user. It also involves partial re-encoding, though processing power is conserved because transcoders re-use many first-generation encoder decisions to produce second-generation bit streams [5].

Video transcoding has many real-world transmission applications. Servers of video bit streams, when equipped with transcoding capabilities, could provide an arbitrary number of bit streams with varying quality and bandwidth characteristics, all derived from a single premium video encoding. This would support clients with a range of connection qualities, or even be responsive to a single client with a connection quality that varies over time. Alternatively, in a storage medium environment, transcoding could be used to reduce video file size after encoding, possibly allowing data to be ported from a high-capacity storage medium to another with a lower capacity.

The following sections describe the operation and implementation of an MPEG-2 video transcoder, and provide analysis and quantitative results of a bit stream subjected to transcoding.

4.1 Background to MPEG-2 Video Transcoding

Speed is a primary concern for video transcoding, as a video server making use of it must be capable of transcoding one or more bit streams into an assortment of second-generation bit streams, *all in real-time*. There are three common levels of transcoding (see Figure 4-1), each offering a different balance between output quality and computational complexity.

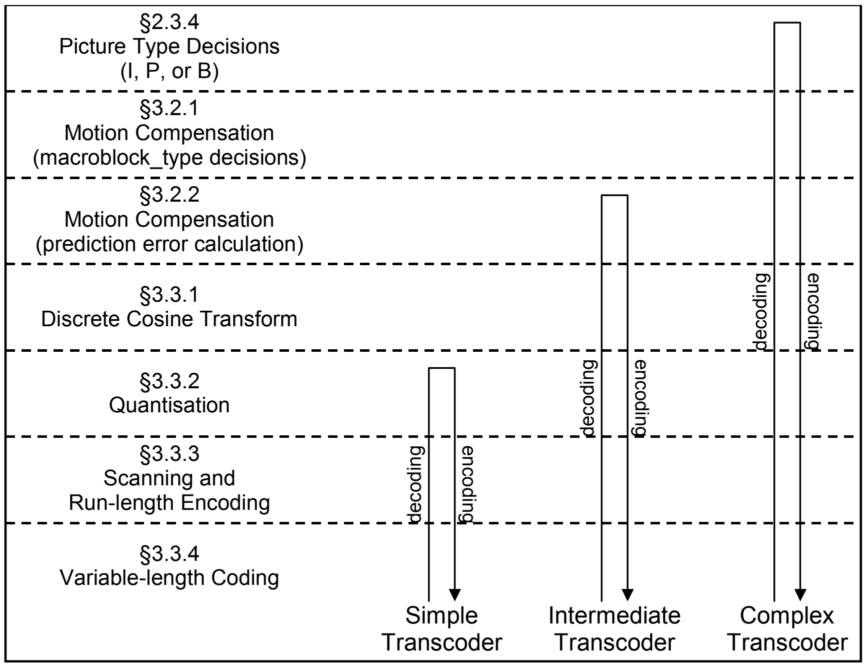


Figure 4-1: Levels of Transcoding

The **simplest** transcoder partially decodes a video bit stream, parsing picture slices and macroblocks up to the variable-length encoded blocks as described in §3.3.4. The variable-length encoding, run-length encoding/scanning (§3.3.3), and quantization (§3.3.2) are rolled back, stopping short of performing an inverse discrete cosine transform on the 8×8 block. The block is then re-quantized – this time with larger quantization divisors – and re-scanned, run-length encoded, and variable-length encoded into a block bit string shorter than the original.

An **intermediate** transcoder goes further than the previous example for P- and B-pictures by performing the inverse discrete cosine transform and re-calculating the motion vector's prediction error. This new prediction error would be with respect to the *second-generation* reference macroblock as opposed to the *first-generation* macroblock [5]. The forward discrete cosine transform is redone, and the bit stream is further re-constructed according to the same procedure as the simple transcoder.

The two previous transcoders retained the original video encoder's decisions regarding picture types, motion prediction, and coding decisions [5]. The most **complex** form of transcoding performs almost a full decoding (stopping short of display-specific decoding, such as YCbCr-to-RGB color space conversion), and re-encodes the bit stream with fewer I-pictures and more B-pictures, increased reliance on motion vectors, and coding decisions with lower precision and thresholds than before.

Each step up in transcoding intensity entails a substantial increase in computational complexity. The extra inverse/forward cosine transforms of the second option, and the additional motion prediction decisions in the third option, tend to rule out the possibility of both from being implemented as a software add-on to conventional real-time video servers. The software transcoder implementation accompanying this report uses the superficial transcoding techniques of the first option.

The treatment of MPEG-2 video block quantization in §3.3.2 neglected certain factors in the quantization equations, particularly for non-intra blocks. Equations (4.1) and (4.2) contain the forward and inverse quantization equations for intra blocks, respectively. F_q represents the quantized 8×8 block of integers, F the unquantized block, and Q the current quantization matrix – u and v are integer indices ranging from 0 to 7.

$$F_q[v][u] = \frac{16 \cdot F[v][u]}{Q[v][u] \cdot quantiser\_scale} \tag{4.1}$$

$$F[v][u] = \frac{F_q[v][u] \cdot Q[v][u] \cdot quantiser\_scale}{16} \tag{4.2}$$

The forward and inverse quantization equations for non-intra blocks (4.3 and 4.4) introduce additional complexity by mixing the signum function with integer division. The inverse quantization function is taken from §7.4.2.3 of the MPEG-2 video specification [2].

$$F[v][u] = \frac{\left(2 \cdot F_q[v][u] + \operatorname{sgn}(F_q[v][u])\right) \cdot Q[v][u] \cdot quantiser\_scale}{32} \tag{4.3}$$

$$F_q[v][u] = \frac{\dfrac{32 \cdot F[v][u]}{Q[v][u] \cdot quantiser\_scale} - \operatorname{sgn}\left(\dfrac{32 \cdot F[v][u]}{Q[v][u] \cdot quantiser\_scale}\right)}{2} \tag{4.4}$$

4.2 MPEG-2 Video Transcoder Implementation

MPEG-2 transport streams were obtained by encoding analog video signals with iCompression's iVAC hardware MPEG-2 encoder. Video elementary streams (.M2V files) were extracted from the resulting .MPG transport stream files using tools from Microsoft's DirectX Media 6.0 SDK. Video elementary stream files could then be viewed with Windows Media Player software.

The sample MPEG-2 video transcoder was written with Microsoft Visual C++ 6.0, running on a 600 MHz Pentium III computer. The source code was based entirely on the text of the MPEG-2 video standard; it contains few optimizations[10] and is thus quite readable. For a complete listing of the source code, see the Appendix.

The sample transcoder implements the most superficial transcoder (see §4.1) for the Windows NT operating system. The graphical user interface allows the user to select an input MPEG-2 video elementary stream (.M2V) file and an output .M2V file with standard Windows NT dialog boxes. The user can also input a small *quantiser_scale_code_increment* integer representing a uniform increment of the quantiser_scale_code parameter in each slice and macroblock header. Upon execution, the transcoder parses the input file and decodes it up to (and including) the inverse quantization step. The file is then re-encoded with a new quantiser_scale_code, incremented to strengthen the quantization, and saved as the output file.

4.3 Experimental Results and Analysis

An uncompressed still image was captured and encoded into a 60-second MPEG-2 video elementary stream at a high bit rate of 8.00 Mbps. The resulting .MPG transport stream file was next filtered into an .M2V video elementary stream file. This original elementary stream file was then transcoded into several second-generation streams with a variety of quantiser_scale_code_-increment parameters.

The first I-picture from each stream was extracted and saved as a Windows .BMP file. Luminance peak signal-to-noise ratios (PSNR) were calculated for each second-generation transcoded bitmap. This measurement treated the I-picture from the first-generation video stream as the original signal rather than the original still image, because the digital-to-analog-to-digital conversion of the still image into the first-generation video changed image dimensions and caused color brightness distortion. The quantitative results of this experiment are shown in Table 4-1. Figure 4-2 shows: (a) the original still image, (b) the I-picture from the first-generation encoding, (c-f) and the I-pictures from the four transcoded videos.

[10] The transcoder typically runs at one-fifth of real-time speed, though optimizations (particularly for frequently-used variable-length decoding routines) could resolve this issue.

Table 4-1: Quantitative transcoding results (8 Mbps)

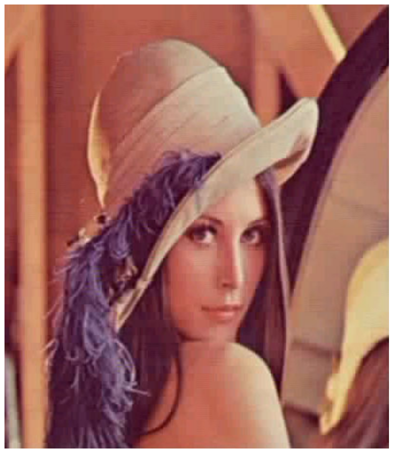
	quantiser_scale_-code_increment	Average quan-tiser_scale_code	Average quan-tiser_scale value	Average Bit Rate	Luminance PSNR
	0 (original)	3	6	8.00 Mbps	—
transcoded bit streams	1	4	8	6.78 Mbps	35.91 dB
	2	5	10	3.88 Mbps	35.48 dB
	4	7	14	3.12 Mbps	33.57 dB
	6	9	18	2.61 Mbps	32.93 dB

(a) Original Still Image

(b) First-generation encoded video (8 Mbps)
(quantiser_scale_code_increment = 0)

(c) Transcoded video (6.78 Mbps)
(quantiser_scale_code_increment = 1)

(d) Transcoded video (3.88 Mbps)
(quantiser_scale_code_increment = 2)


(e) Transcoded video (3.12 Mbps)
(quantiser_scale_code_increment = 4)

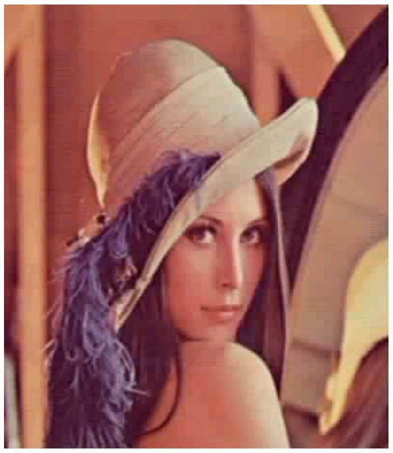

(f) Transcoded video (2.61 Mbps)
(quantiser_scale_code_increment = 6)

Figure 4-2: Qualitative transcoding results (8 Mbps)

As Table 4-1 indicates, transcoding degrades picture quality only slightly, according to the peak signal-to-noise ratio measure. This observation is confirmed by inspection of the images in Figure 4-2 – only the heavily compressed images (d-f) show signs of "blockiness" in facial areas. There is also a barely-noticeable loss of detail in the feathers on the hat. Note that because each video contained a still image displayed for 60 seconds, the I-pictures are representative of the entire video.

The original 8 Mbps video could not be transcoded down any lower than the 2.61 Mbps reached with quantiser_scale_code_increment = 6. Transcoding of any kind modifies the bit rate, which in turn affects the operation of the video buffering verifier. During video playback, the bit stream is routed through the video buffering verifier. This buffer must be kept in a delicate state where it cannot overflow its capacity (as specified in the sequence header) and it can never be emptied. Each picture_header contains a 16-bit vbv_delay field specifying a time (with 11 μs resolution) to delay decoding the next picture. The sample transcoder implementation did not properly recalculate this field, though efforts were made for it to do so. Therefore, whenever transcoded videos were viewed, the media player would crash as soon as this buffer emptied.

For 8 Mbps video transcoded with quantiser_scale_code_increment greater than 6, the media player would crash before any I-pictures could be decoded and viewed. To determine whether the transcoding results apply to video streams with lower bit rates, the original still image was encoded into a lower-bandwidth 2 Mbps video elementary stream, and the transcoding experiment was repeated. The quantitative and qualitative results of this repeated experiment are shown in Table 4-2 and Figure 4-3, which mirror the organization of Table 4-1 and Figure 4-2.

Table 4-2: Quantitative transcoding results (2 Mbps)

quantiser_scale_-code_increment	Average quantiser_-scale_code	Average quantiser_scale value	Average Bit Rate	Luminance PSNR
0 (original)	14	28	2.00 Mbps	—
1	15	30	1.87 Mbps	36.15 dB
4	18	36	1.80 Mbps	36.00 dB
6	20	40	1.70 Mbps	35.16 dB
15	27	54	1.64 Mbps	31.98 dB

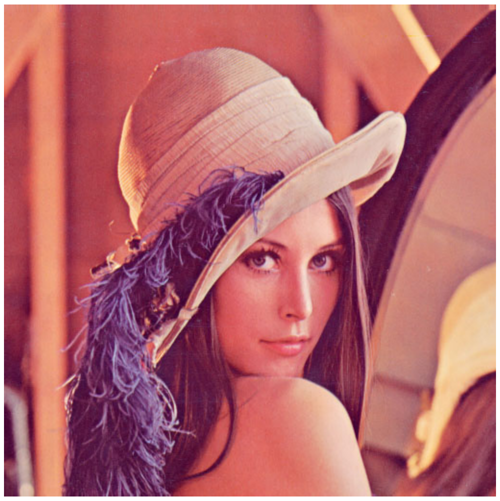
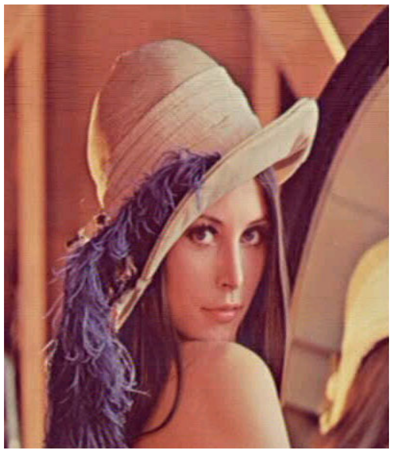
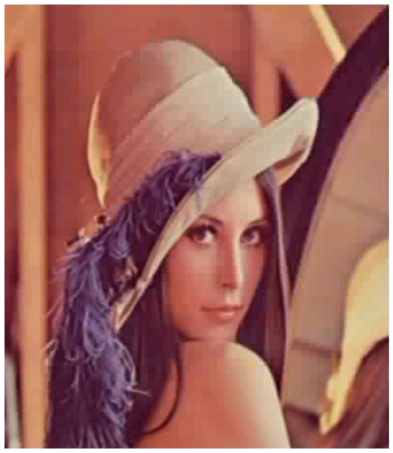
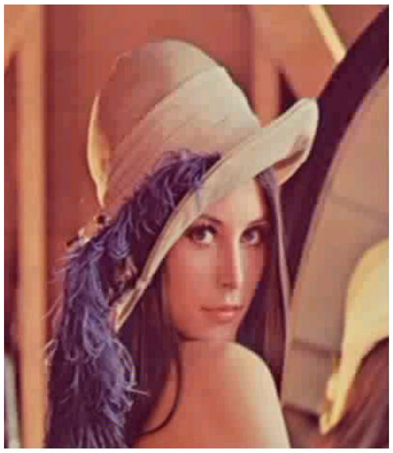
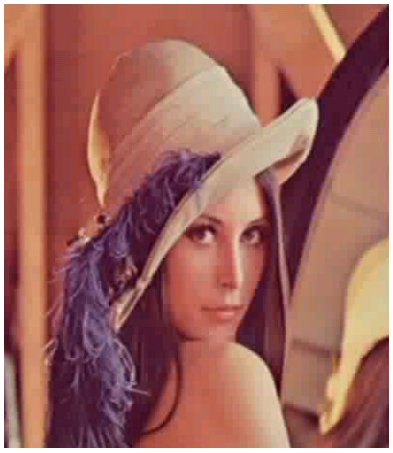
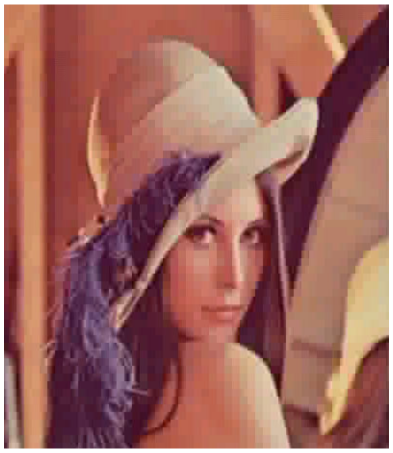
 (a) Original Still Image	 (b) First-generation encoded video (2 Mbps) (quantiser_scale_code_increment = 0)
 (c) Transcoded video (1.87 Mbps) (quantiser_scale_code_increment = 1)	 (d) Transcoded video (1.80 Mbps) (quantiser_scale_code_increment = 4)
 (e) Transcoded video (1.70 Mbps) (quantiser_scale_code_increment = 6)	 (f) Transcoded video (1.64 Mbps) (quantiser_scale_code_increment = 15)

Figure 4-3: Qualitative transcoding results (2 Mbps)

The results in Table 4-2 and Figure 2-1 show that transcoding 2 Mbps video degrades quality more so than with 8 Mbps video – picture (f) appears especially blocky. The bandwidth reductions are also less spectacular, possibly because encoded blocks (which are compressed with this form of transcoding) account for a smaller proportion of the bandwidth. Overhead from bit stream elements left untouched by the transcoding algorithm, such as headers and motion vectors, accounts for a greater proportion of the bandwidth.

5 CONCLUSIONS AND RECOMMENDATIONS

The preceding report presented descriptions of MPEG, the MPEG-2 standard, and the structure of an MPEG-2 video elementary stream. This was followed by a detailed examination of the MPEG-2 picture encoding process. An algorithm for transcoding MPEG-2 bit streams, for the purpose of decreasing stream bandwidth, was then developed, implemented, and analyzed.

The transcoding results in §4.3 demonstrate that transcoding video bit streams can significantly decrease the bandwidth of a bit stream at the expense of picture quality. The qualitative results suggest that this is a price worth paying. The results are promising, but preliminary – the video buffering verifier under-run problem needs to be resolved so transcoded videos can be examined in their entirety. In this case, only the first I-picture of each video could be retrieved and analyzed before the buffer emptied and the media player crashed.

The results showed that superficial transcoding techniques, where only the block data is transcoded, face natural compression limits. It appears as though video stream bandwidth cannot be reduced by more than an order of magnitude with these techniques, because of the overhead of standard headers and excess motion vectors. Using the alternative quantization scheme[11] may extend this range to some extent, but perhaps a more intrusive method of transcoding should be considered, where relatively unimportant motion vectors are removed from the bit stream altogether.

Transcoding could also be examined for video streams created by a variety of MPEG-2 encoders at a wider range of bit rates. This would include software encoders, where an original still image could be digitally inserted into a video stream without the distortion of digital-to-analog-to-digital conversion. This would make a quantitative comparison between transcoded video and an original image possible.

Finally, it should be acknowledged that the peak signal-to-noise ratio measure of image degradation is not the ultimate measure of image quality. Two images having few 'close' pixel values may be poles apart according to a mathematical algorithm but virtually indistinguishable to the human eye. The best judge of video quality will always be a subjective human observer, viewing at full size and full frame rate.

[11] For more information, see the description of the q_scale_flag in §3.3.2.

APPENDIX: TRANSCODER SOURCE CODE

FILE: WinXCode.cpp

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "resource.h"
#include "stream.h"
#include "mpeg2hdr.h"

/* functions */
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT OutputM2VAttributes(HANDLE, HWND);
LRESULT TranscodeFile(HANDLE, HANDLE, INT, INT, INT);

/* global variables */
TCHAR szAppName[] = TEXT("WinXCode");

/*********************************************************************************
 *
 * Function: WinMain(HINSTANCE, HINSTANCE, LPSTR, INT)
 *
 * Description: Entry point for all Windows programs.  Registers window class
 *   and displays dialog box window.
 *
 *********************************************************************************/
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE /*hPrevInstance*/, LPSTR /*lpCmdLine*/, INT nShowCmd) {
    /* Initialize the common controls library (commctrl.dll) */
    INITCOMMONCONTROLSEX initcommoncontrolsex = {sizeof(INITCOMMONCONTROLSEX), ICC_UPDOWN_CLASS | ICC_PROGRESS_CLASS};
    InitCommonControlsEx(&initcommoncontrolsex);

    /* Register window class for dialog box */
    WNDCLASS wndclass = {0};
    wndclass.style         = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc   = WndProc;
    wndclass.cbClsExtra    = 0;
    wndclass.cbWndExtra    = DLGWINDOWEXTRA;
    wndclass.hInstance     = hInstance;
    wndclass.hIcon         = LoadIcon(hInstance, szAppName);
    wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
    wndclass.lpszMenuName  = NULL;
    wndclass.lpszClassName = szAppName;
    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 1;
    }

    /* Create and show dialog box window */
    HWND hwnd = CreateDialogParam(hInstance, szAppName, HWND_DESKTOP, NULL, 0);
    SendMessage(hwnd, WM_INITDIALOG, 0, 0);
    ShowWindow(hwnd, nShowCmd);

    /* Message-processing loop; exit when WM_QUIT comes */
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return 0;
}

/*********************************************************************************
 *
 * Function: WndProc(HWND, UINT, WPARAM, LPARAM)
 *
 * Description: Window procedure for dialog box.  Processes windows messages
 *   sent to the dialog box.
 *
 *********************************************************************************/
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static TCHAR szFilter[] = TEXT("MPEG-2 Video Elementary Streams (*.M2V)\0*.m2v\0")
                              TEXT("All Files (*.*)\0*.*\0\0"),
                 szInputCaption[] = TEXT("Select Transcoder Input File..."),
                 szOutputCaption[] = TEXT("Select Transcoder Output File...");
    static OPENFILENAME ofn = {sizeof(OPENFILENAME), 0}, sfn = {sizeof(OPENFILENAME), 0};
    HANDLE hInputFile = INVALID_HANDLE_VALUE, hOutputFile = INVALID_HANDLE_VALUE;

    switch(message) {
    case WM_INITDIALOG:
        /* Initialize OPENFILENAME structure for file selection common dialog box */
        ofn.hwndOwner         = hwnd;
        ofn.lpstrFilter       = szFilter;
        ofn.lpstrCustomFilter = NULL;
        ofn.lpstrFileTitle    = NULL;
        ofn.lpstrInitialDir   = NULL;
        ofn.lpstrTitle        = NULL;
        ofn.nFileOffset       = 0;
        ofn.nFileExtension    = 0;
        ofn.lpstrDefExt       = TEXT("*.M2V");
        ofn.lCustData         = 0;
        ofn.lpfnHook          = NULL;
        ofn.lpTemplateName    = NULL;
        CopyMemory(&sfn, &ofn, sizeof(OPENFILENAME));
        ofn.lpstrFile = (LPTSTR)malloc(MAX_PATH*sizeof(TCHAR)); ZeroMemory(ofn.lpstrFile, MAX_PATH*sizeof(TCHAR)); ofn.nMaxFile =
                    MAX_PATH;
        sfn.lpstrFile = (LPTSTR)malloc(MAX_PATH*sizeof(TCHAR)); ZeroMemory(sfn.lpstrFile, MAX_PATH*sizeof(TCHAR)); sfn.nMaxFile =
                    MAX_PATH;
        ofn.lpstrTitle = szInputCaption;
        sfn.lpstrTitle = szOutputCaption;
        ofn.Flags = 0;
        sfn.Flags = OFN_OVERWRITEPROMPT;
        /* Initialize spin control for quantiser_scale_code_increment */
        SendDlgItemMessage(hwnd, IDC_SPINQSCI, UDM_SETBASE, 10, 0);
        SendDlgItemMessage(hwnd, IDC_SPINQSCI, UDM_SETBUDDY, (WPARAM)GetDlgItem(hwnd, IDC_EDITQSCI), 0);
        SendDlgItemMessage(hwnd, IDC_SPINQSCI, UDM_SETRANGE, 0, MAKELONG(31, 0));
        return 0;
    case WM_COMMAND:
        switch(wParam) {
        case MAKELONG(IDC_INPUTBROWSE, BN_CLICKED): /* Browse for input file button clicked */
            if (GetOpenFileName(&ofn)) {
                Edit_SetText(GetDlgItem(hwnd, IDC_INPUTFILE), ofn.lpstrFile);
                hInputFile = CreateFile(ofn.lpstrFile, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
                        FILE_FLAG_SEQUENTIAL_SCAN, NULL);
                if (hInputFile == INVALID_HANDLE_VALUE) {
                    ;
                } else {
                    SetCursor(LoadCursor(NULL, IDC_WAIT)); ShowCursor(TRUE);
                    OutputM2VAttributes(hInputFile, GetDlgItem(hwnd, IDC_INPUTINFO));
                    ShowCursor(FALSE); SetCursor(LoadCursor(NULL, IDC_ARROW));
                    CloseHandle(hInputFile);
                }
            }
            break;
        case MAKELONG(IDC_OUTPUTBROWSE, BN_CLICKED): /* Browse for output file button clicked */
            if (GetSaveFileName(&sfn)) Edit_SetText(GetDlgItem(hwnd, IDC_OUTPUTFILE), sfn.lpstrFile);
            break;
        case MAKELONG(IDC_CHECKQSCI, BN_CLICKED): /* Toggle check box for quantiser_scale_code_increment */
            Edit_Enable(GetDlgItem(hwnd, IDC_EDITQSCI), (Button_GetCheck((HWND)lParam) == BST_CHECKED));
            EnableWindow(GetDlgItem(hwnd, IDC_SPINQSCI), (Button_GetCheck((HWND)lParam) == BST_CHECKED));
            break;
        case MAKELONG(IDC_CHECKBITRATE, BN_CLICKED): /* Toggle check box for bit rate edit box */
            Edit_Enable(GetDlgItem(hwnd, IDC_EDITBITRATE), (Button_GetCheck((HWND)lParam) == BST_CHECKED));
            break;
        case MAKELONG(IDC_CHECKVBVSIZE, BN_CLICKED): /* Toggle check box for vbv_buffer edit box */
            Edit_Enable(GetDlgItem(hwnd, IDC_EDITVBVSIZE), (Button_GetCheck((HWND)lParam) == BST_CHECKED));
            break;
        case MAKELONG(IDC_APPLY, BN_CLICKED): /* Apply or OK button clicked; start transcoding */
            /* Create handles for input and output files */
            hInputFile = CreateFile(ofn.lpstrFile, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN,
                    NULL);
            if (hInputFile == INVALID_HANDLE_VALUE) {
                MessageBox(hwnd, TEXT("Invalid Input File"), szAppName, MB_ICONERROR);
                SetFocus(GetDlgItem(hwnd, IDC_INPUTFILE));
                break;
            }
            hOutputFile = CreateFile(sfn.lpstrFile, GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
                    NULL);
            if (hOutputFile == INVALID_HANDLE_VALUE) {
                MessageBox(hwnd, TEXT("Invalid Ouput File"), szAppName, MB_ICONERROR);
                SetFocus(GetDlgItem(hwnd, IDC_OUTPUTFILE));
                break;
            }
            /* Set up transcoding variables, begin transcoding */
            INT quantiser_scale_code_increment, new_bit_rate_value, new_vbv_buffer_size_value;
            /* Set parameters to -1 if they should be ignored (check box disabled) */
            quantiser_scale_code_increment = (Button_GetCheck(GetDlgItem(hwnd,IDC_CHECKQSCI)) == BST_CHECKED) ?
                    GetDlgItemInt(hwnd, IDC_EDITQSCI, NULL, FALSE) : -1;
            new_bit_rate_value = (Button_GetCheck(GetDlgItem(hwnd,IDC_CHECKBITRATE)) == BST_CHECKED) ? GetDlgItemInt(hwnd,
                    IDC_EDITBITRATE, NULL, FALSE) : -1;
            new_vbv_buffer_size_value = (Button_GetCheck(GetDlgItem(hwnd,IDC_CHECKVBVSIZE)) == BST_CHECKED) ? GetDlgItemInt(hwnd,
                    IDC_EDITVBVSIZE, NULL, FALSE) : -1;
            SetCursor(LoadCursor(NULL, IDC_WAIT)); ShowCursor(TRUE);
            TranscodeFile(hInputFile, hOutputFile, quantiser_scale_code_increment, new_bit_rate_value, new_vbv_buffer_size_value);
            ShowCursor(FALSE); SetCursor(LoadCursor(NULL, IDC_ARROW));
            CloseHandle(hInputFile);
            CloseHandle(hOutputFile);
            break;
        case MAKELONG(IDOK, BN_CLICKED):
            /* Emulate clicking the Apply button */
            SendMessage(hwnd, WM_COMMAND, MAKELONG(IDC_APPLY, BN_CLICKED), (LPARAM)GetDlgItem(hwnd, IDC_APPLY));
            /* fall through to exit dialog box */
        case MAKELONG(IDCANCEL, BN_CLICKED):
            SendMessage(hwnd, WM_CLOSE, 0, 0);
            break;
        }
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}

#define STREAMSIZE 128*1024
/*******************************************************************************
 *
 * Function: OutputM2VAttributes(HANDLE, HWND)
 *
 * Description: Calculates quantiser_scale_code, bitrate, and vbv_buffer stats
 *   for the M2V file in hFile.  Outputs results as a string to the static
 *   control from hwndStatic.
 *
 *******************************************************************************/
LRESULT OutputM2VAttributes(HANDLE hFile, HWND hwndStatic) {
    MPEG2HEADERS hdrs;
    TCHAR szBuffer[200];
    INT start_code;
    INT bit_rate_value = 0, vbv_buffer_size_value = 0;
    INT qsc_min=32, qsc_max = -1, qsc_avg = 0, qsc_num = 0;
    STREAM stream(STREAMSIZE);

    /* Calculates statistics based on first video sequence packet of file */
    while(1) {
        stream.ReadM2VFile(hFile);
        stream.ResetPtr();
        start_code = stream.GetBits(32);

        if (start_code == SEQUENCE_END_CODE) break;
        if (start_code == SEQUENCE_HEADER_CODE) {
            if (hdrs.sequence_header.bit_rate_value) break;
            stream.ResetPtr();
            hdrs.sequence_header.Read(stream);
            bit_rate_value |= (hdrs.sequence_header.bit_rate_value*400);
            vbv_buffer_size_value |= (hdrs.sequence_header.vbv_buffer_size_value);
        }
        if (start_code == EXTENSION_START_CODE && stream.PeekBits(4) == SEQUENCE_EXTENSION_ID) {
            stream.ResetPtr();
            hdrs.sequence_extension.Read(stream);
            bit_rate_value |= ((hdrs.sequence_extension.bit_rate_extension*400)<<18);
            vbv_buffer_size_value |= (hdrs.sequence_extension.vbv_buffer_size_extension<<10);
        }
        if (start_code >= 0x00000101 && start_code <= 0x000001af) { /* slice */
            start_code = stream.PeekBits(5);
            if (start_code < qsc_min) qsc_min = start_code;
            if (start_code > qsc_max) qsc_max = start_code;
            qsc_avg += start_code; qsc_num++;
        }
    }
    qsc_avg = (qsc_avg+(qsc_num>>1))/qsc_num;
    wsprintf(szBuffer,
        TEXT("MPEG Elementary Stream file properties:\n")
        TEXT("\tquantiser_scale_code: %d avg,   %d min,   %d max\n")
        TEXT("\tbit_rate_value in bps: %d\n")
        TEXT("\tvbv_buffer_size_value: %d\n")
        ,qsc_avg, qsc_min, qsc_max, bit_rate_value, vbv_buffer_size_value);
    Static_SetText(hwndStatic, szBuffer);
    return 0;
}

/******************************************************************************
 *
 * Function: TranscodeFile(HANDLE, HANDLE, INT, INT, INT);
 *
 * Description: Transcodes the M2V file of hInputFile into hOutputFile.  The
 *   parameters quantiser_scale_code_increment, new_vbv_buffer_size, and
 *   new_bit_rate_value are ignored if they're -1.
 *
 ******************************************************************************/
LRESULT TranscodeFile(HANDLE hInputFile, HANDLE hOutputFile, INT quantiser_scale_code_increment, INT new_bit_rate_value, INT
                     new_vbv_buffer_size_value) {
    INT start_code;
    STREAM rstream(STREAMSIZE), wstream(STREAMSIZE); /* read stream and write stream */
    MPEG2HEADERS hdrs;
    SLICE slice;

    if (new_bit_rate_value != -1) new_bit_rate_value /= 400; /* convert to bit stream resolution */
    while(1) {
        rstream.ReadM2VFile(hInputFile); rstream.ResetPtr();
        start_code = rstream.PeekBits(32);
        switch(start_code) {
        case PICTURE_START_CODE: /* set vbv_delay to VBR (0xffff) */
            hdrs.picture_header.Read(rstream);
            hdrs.picture_header.vbv_delay = 0xffff;
            wstream.ResetStream();  hdrs.picture_header.Write(wstream);
            wstream.ResetPtr();     wstream.WriteM2VFile(hOutputFile);
            break;
        case SEQUENCE_ERROR_CODE:
        case USER_DATA_START_CODE:
            rstream.WriteM2VFile(hInputFile);
            break;
        case SEQUENCE_HEADER_CODE: /* set bit_rate_value and vbv_buffer_size_value, if necessary */
            hdrs.sequence_header.Read(rstream);
            if (new_bit_rate_value != -1)
                hdrs.sequence_header.bit_rate_value = (new_bit_rate_value & ((1<<18)-1));
            if (new_vbv_buffer_size_value != -1)
                hdrs.sequence_header.vbv_buffer_size_value = (USHORT)(new_vbv_buffer_size_value & ((1<<10)-1));
            wstream.ResetStream();  hdrs.sequence_header.Write(wstream);
            wstream.ResetPtr();     wstream.WriteM2VFile(hOutputFile);
            break;
        case SEQUENCE_END_CODE:
            hdrs.sequence_end.Read(rstream);
            rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
            return 0; /* Exit function at end of bit stream */
        case GROUP_START_CODE:
            hdrs.group_of_pictures_header.Read(rstream);
            rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
            break;
        case EXTENSION_START_CODE:
            rstream.GetBits(32); start_code = rstream.PeekBits(4); rstream.ResetPtr();
            switch(start_code) { /* switch (extension_id) */
            case SEQUENCE_EXTENSION_ID: /* set bit_rate_extension and vbv_buffer_size_extension, if necessary */
                hdrs.sequence_extension.Read(rstream);
                if (new_bit_rate_value != -1)
                    hdrs.sequence_extension.bit_rate_extension = (USHORT)((new_bit_rate_value >> 18) & ((1<<12)-1));
                if (new_vbv_buffer_size_value != -1)
                    hdrs.sequence_extension.vbv_buffer_size_extension = (UCHAR)((new_vbv_buffer_size_value >> 10) & ((1<<8)-1));
                wstream.ResetStream();
                hdrs.sequence_extension.Write(wstream);
                wstream.ResetPtr();  wstream.WriteM2VFile(hOutputFile);
                break;
            case SEQUENCE_DISPLAY_EXTENSION_ID:
                hdrs.sequence_display_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case QUANT_MATRIX_EXTENSION_ID:
                hdrs.quant_matrix_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case COPYRIGHT_EXTENSION_ID:
                hdrs.copyright_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case SEQUENCE_SCALABLE_EXTENSION_ID:
                hdrs.sequence_scalable_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case PICTURE_DISPLAY_EXTENSION_ID:
                hdrs.picture_display_extension.Read(rstream,
                    hdrs.sequence_extension.progressive_sequence,
                    hdrs.picture_coding_extension.picture_structure,
                    hdrs.picture_coding_extension.repeat_first_field);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case PICTURE_CODING_EXTENSION_ID:
                hdrs.picture_coding_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case PICTURE_SPATIAL_SCALABLE_EXTENSION_ID:
                hdrs.picture_spatial_scalable_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            case PICTURE_TEMPORAL_SCALABLE_EXTENSION_ID:
                hdrs.picture_temporal_scalable_extension.Read(rstream);
                rstream.ResetPtr();  rstream.WriteM2VFile(hOutputFile);
                break;
            }
            break;
        default:
            if (start_code < 0x00000101 || start_code > 0x000001af) break;
            /* Transcode slice by re-quantisation */
            rstream.ResetPtr();
            if (quantiser_scale_code_increment == -1) {rstream.WriteM2VFile(hOutputFile); break;}
            wstream.ResetStream();
            slice.c.hdrs = &hdrs; slice.c.rstm = &rstream; slice.c.wstm = &wstream;
            slice.RW_xcode(quantiser_scale_code_increment);
            wstream.ResetPtr();   wstream.WriteM2VFile(hOutputFile);
        }
    }
    return 0;
}
```

FILE: WinXCode.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

WINXCODE DIALOG DISCARDABLE  100, 100, 220, 164
STYLE DS_SETFOREGROUND | DS_3DLOOK | DS_NOFAILCREATE | WS_MINIMIZEBOX |
    WS_CAPTION | WS_SYSMENU
CAPTION "MPEG-2 Video Transcoder for Windows"
CLASS "WinXCode"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_INPUTFILE,7,7,150,14,ES_AUTOHSCROLL | ES_READONLY
    PUSHBUTTON      "Browse...",IDC_INPUTBROWSE,163,7,50,14
    EDITTEXT        IDC_OUTPUTFILE,7,26,150,14,ES_AUTOHSCROLL | ES_READONLY
    PUSHBUTTON      "Browse...",IDC_OUTPUTBROWSE,163,26,50,14
    LTEXT           "<No input file properties to display>",IDC_INPUTINFO,7,
                    44,202,38
    CONTROL         "Increment quantiser_scale_code by",IDC_CHECKQSCI,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,7,89,126,10
    EDITTEXT        IDC_EDITQSCI,134,87,38,15,ES_AUTOHSCROLL | WS_DISABLED |
                    NOT WS_TABSTOP
    CONTROL         "Spin1",IDC_SPINQSCI,"msctls_updown32",UDS_SETBUDDYINT |
                    UDS_ALIGNRIGHT | UDS_ARROWKEYS | UDS_HOTTRACK |
                    WS_DISABLED,172,87,11,14
    CONTROL         "Set video sequence bitrate to",IDC_CHECKBITRATE,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,7,106,107,10
    EDITTEXT        IDC_EDITBITRATE,115,104,68,14,ES_AUTOHSCROLL | ES_NUMBER |
                    WS_DISABLED
    CONTROL         "Set vbv_buffer_size to",IDC_CHECKVBVSIZE,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,7,123,85,10
    EDITTEXT        IDC_EDITVBVSIZE,94,121,89,14,ES_AUTOHSCROLL | ES_NUMBER |
                    WS_DISABLED
    DEFPUSHBUTTON   "OK",IDOK,26,143,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,84,143,50,14
    PUSHBUTTON      "Apply",IDC_APPLY,142,143,50,14
END

/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
WINXCODE                ICON    DISCARDABLE     "WinXCode.ico"

/////////////////////////////////////////////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,0
 PRODUCTVERSION 1,0,0,0
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x40004L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904b0"
        BEGIN
            VALUE "CompanyName", "University of Manitoba\0"
            VALUE "FileDescription", "Sample Application for 24.433 Thesis\0"
            VALUE "FileVersion", "1.00\0"
            VALUE "InternalName", "programmer: Delbert Dueck\0"
            VALUE "LegalCopyright", "Copyright © 2001\0"
            VALUE "OriginalFilename", "WinXCode.exe\0"
            VALUE "ProductName", "MPEG-2 Video Transcoder for Windows\0"
            VALUE "ProductVersion", "1.00\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x0, 1200
    END
END
```

FILE: Stream.h

```
#ifndef __STREAM_H__
#define __STREAM_H__

#include <windows.h>

typedef struct _HUFFMANENTRY {
     ULONG value;
     USHORT mask, code;
} HUFFMANENTRY, *LPHUFFMANENTRY;

/*******************************************************************************
 *
 * Class: STREAM
 *
 * Description: Used for processing bits sequentially, as in a bit stream.
 *   Entire class is declared inline for performance purposes.
 *
 *******************************************************************************/
class STREAM {
    private: PUCHAR m_lpbitstream; /* pointer to the actual stream data */
    private: ULONG m_streamlength, m_streamptr; /* length of the stream, pointer position (in bits) */

    public: STREAM(int buffersize) { /* constructor */
                m_lpbitstream = (PUCHAR)malloc(buffersize);
                m_streamlength = m_streamptr = 0;
                return;
            }
    public: ~STREAM() { /* destructor */
                free(m_lpbitstream);
                m_lpbitstream = NULL;
                m_streamlength = m_streamptr = 0;
                return;
            }
    /* Returns the length of the stream data (not the buffer length) */
    public: ULONG Length_in_bytes() {return m_streamlength>>3;}
    public: ULONG Length_in_bits() {return m_streamlength;}

    /* Resets stream pointer back to the beginning, and returns length of stream */
    public: LRESULT inline ResetPtr() {m_streamptr = 0; return m_streamlength;}

    /* Resets stream pointer to beginning and sets stream length to zero */
    public: LRESULT inline ResetStream() {m_streamptr = m_streamlength = 0; return 0;}

    /* Returns the number of unread bits in the bit stream */
    public: LRESULT inline UnreadBits() {return (m_streamlength-m_streamptr);}

    /* Returns the value of the next "nbits" bits in the stream; does not move the pointer */
    public: ULONG PeekBits(ULONG nbits) {
                ULONG ret=0;
                int shiftptr = 24 + (m_streamptr&((1<<3)-1)); /* shiftptr = [24..31] */
                UINT byteptr = m_streamptr>>3;

                /* fully load the ULONG with next bits */
                while (byteptr <= m_streamlength>>3) {
                    if (shiftptr < 8) {
                        ret |= (m_lpbitstream[byteptr]>>(shiftptr));
                        break;
                    }
                    ret |= (m_lpbitstream[byteptr++]<<shiftptr);
                    shiftptr -= 8;
                }
                ret >>= (32-nbits); /* shift bits from MSB to LSB */
                return ret;
            }

    /* Returns the value of the next "nbits" bits in the stream, and increments the pointer by "nbits" */
    public: ULONG inline GetBits(ULONG nbits) {
                ULONG ret = PeekBits(nbits);
                m_streamptr += nbits;
                return ret;
            }

    /* Appends the least significant "nbits" of "bits" to the end of the bit stream */
    public: LRESULT PutBits(ULONG bits, ULONG nbits) {
                int shiftptr = nbits + (m_streamptr&0x7) - 8;
                ULONG byteptr = m_streamptr>>3;
                ULONG mask = (1<<nbits) - 1;

                if (nbits == 32) mask = 0xffffffff;
                bits &= mask;
                while (1) {
                    if (shiftptr < 0) {
                        m_lpbitstream[byteptr] = (UCHAR)((m_lpbitstream[byteptr]&~(mask<<-shiftptr)) | (bits<<-shiftptr));
                        break;
                    }
                    m_lpbitstream[byteptr++] = (UCHAR)((m_lpbitstream[byteptr]&~(mask>>shiftptr)) | (bits>>shiftptr));
                    shiftptr -= 8;
                }
                m_streamptr += nbits;
                if (m_streamlength < m_streamptr) m_streamlength = m_streamptr;
                return 0;
            }

    /* Returns the next Huffman-decoded value in the bit stram, according to the dictionary "lphman"
       The stream pointer is not incremented */
    public: ULONG PeekHuffman(LPHUFFMANENTRY lphman) {
                INT i;
                ULONG vlc;

                i = m_streamlength-m_streamptr;
                if (i >= 32) vlc = PeekBits(32);
                else vlc = PeekBits(i)<<(32-i);
                for (i = 0; lphman[i].mask != 0; i++) {
                    if ((vlc & lphman[i].mask) == lphman[i].code)
                        return lphman[i].value;
                }
                _ASSERT(0);
                return 0xffffffff;
            }

    /* Returns the next Huffman-decoded value in the bit stream, according to the dictionary "lphman"
       The stream pointer is incremented by the appropriate amount */
    public: ULONG GetHuffman(LPHUFFMANENTRY lphman) {
                INT i;
                USHORT vlc;

                i = m_streamlength-m_streamptr;
                if (i >= 16) vlc = (USHORT)PeekBits(16);
                else vlc = (USHORT)PeekBits(i)<<(16-i);

                for (i = 0; lphman[i].mask != 0; i++) {
                    if ((vlc&lphman[i].mask) == lphman[i].code) {
                        vlc = lphman[i].mask;
                        while(vlc) {m_streamptr++; vlc <<= 1;}
                        return lphman[i].value;
                    }
                }
                _ASSERT(0);
                return 0xffffffff;
            }

    /* Huffman-encodes "value" with the dictionary "lphman", and appends it to the bit stream */
    public: LRESULT PutHuffman(LPHUFFMANENTRY lphman, ULONG value) {
                INT i, j;

                for (i=0,j=0; lphman[i].mask; i++) {
                    if (lphman[i].value == value) {
                        value = lphman[i].mask;
                        while((USHORT)value) {j++; value <<= 1;}
                        value = lphman[i].code;
                        return PutBits(lphman[i].code>>(16-j), j);
                    }
                }
                return -1;
            }

    /* Places the next start code (and its payload) from the M2V file "hFile" in the bit stream. */
    public: INT ReadM2VFile(HANDLE hFile) {
                DWORD nBytes;
                UCHAR ucTemp;
                int startcode=-2;
                m_streamlength = m_streamptr = 0;
                while (1) {
                    if (ReadFile(hFile, &ucTemp, 1, &nBytes, NULL) == 0 || nBytes == 0) return -1;
                    PutBits(ucTemp, 8);
                    if (ucTemp == 0) startcode++;
                    else if (startcode >= 2 && ucTemp == 1) {
                        SetFilePointer(hFile, -3, NULL, FILE_CURRENT);
                        m_streamlength -= 24;
                        startcode = 0;
                        return 0;
                    }
                    else startcode = 0;
                }
                return 0;
            }

    /* Writes the bit stream to the M2V file "hFile" and padds with zeroes for byte-alignment */
    public: INT WriteM2VFile(HANDLE hFile) {
                DWORD nBytes;
                UCHAR ucTemp;
                INT nbits = UnreadBits();
                while (nbits > 0) {
                    if (nbits >= 8)
                        ucTemp = (UCHAR)GetBits(8);
                    else
                        ucTemp = (UCHAR)(GetBits(nbits)<<(8-nbits));
                    if (!WriteFile(hFile, &ucTemp, 1, &nBytes, NULL)) return -1;
                    nbits = UnreadBits();
                }
                return 0;
            }
};

#endif
```

FILE: MPEG2hdr.h

```
#ifndef __MPEG2HDR_H__
#define __MPEG2HDR_H__

#include <windows.h>
#include "stream.h"

/* MPEG-2 header constants */
#define PICTURE_START_CODE   0x00000100
```

```
#define USER_DATA_START_CODE 0x000001b2
#define SEQUENCE_HEADER_CODE 0x000001b3
#define SEQUENCE_ERROR_CODE  0x000001b4
#define EXTENSION_START_CODE 0x000001b5
#define SEQUENCE_END_CODE    0x000001b7
#define GROUP_START_CODE     0x000001b8
#define SEQUENCE_EXTENSION_ID                 0x1
#define SEQUENCE_DISPLAY_EXTENSION_ID         0x2
#define QUANT_MATRIX_EXTENSION_ID             0x3
#define COPYRIGHT_EXTENSION_ID                0x4
#define SEQUENCE_SCALABLE_EXTENSION_ID        0x5
#define PICTURE_DISPLAY_EXTENSION_ID          0x7
#define PICTURE_CODING_EXTENSION_ID           0x8
#define PICTURE_SPATIAL_SCALABLE_EXTENSION_ID  0x9
#define PICTURE_TEMPORAL_SCALABLE_EXTENSION_ID 0xa

/*******************************************************************************
 *
 * Class: SEQUENCE_HEADER
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a sequence header.  Contains member functions to
 *   parse (Read) a bit stream and to output (Write) a bit stream.
 *
 *******************************************************************************/
class SEQUENCE_HEADER {
public: /* public data */
    ULONG sequence_header_code; /* 32 bits */
    USHORT horizontal_size_value; /* 12 bits */
    USHORT vertical_size_value; /* 12 bits */
    BYTE aspect_ratio_information; /* 4 bits */
    BYTE frame_rate_code; /* 4 bits */
    ULONG bit_rate_value; /* 18 bits */
    USHORT vbv_buffer_size_value; /* 10 bits */
    BYTE constrained_parameters_flag; /* 1 bit */
    BYTE load_intra_quantiser_matrix; /* 1 bit */
    BYTE intra_quantiser_matrix[64]; /* 64 8-bit coefficients */
    BYTE load_non_intra_quantiser_matrix; /* 1 bit */
    BYTE non_intra_quantiser_matrix[64]; /* 64 8-bit coefficients */
public: /* public methods */
    SEQUENCE_HEADER() {memset(this, 0, sizeof(SEQUENCE_HEADER)); return;}
    LRESULT Read(STREAM& stream) {
        INT i;
        sequence_header_code = stream.GetBits(32);
        horizontal_size_value = (USHORT)stream.GetBits(12);
        vertical_size_value = (USHORT)stream.GetBits(12);
        aspect_ratio_information = (BYTE)stream.GetBits(4);
        frame_rate_code = (BYTE)stream.GetBits(4);
        bit_rate_value = stream.GetBits(18);
        if (stream.GetBits(1) == 0) return -1;
        vbv_buffer_size_value = (USHORT)stream.GetBits(10);
        constrained_parameters_flag = (BYTE)stream.GetBits(1);
        if (load_intra_quantiser_matrix = (BYTE)stream.GetBits(1))
            for (i = 0; i < 64; i++)
                intra_quantiser_matrix[i] = (BYTE)stream.GetBits(8);
        if (load_non_intra_quantiser_matrix = (BYTE)stream.GetBits(1))
            for (INT i = 0; i < 64; i++)
                non_intra_quantiser_matrix[i] = (BYTE)stream.GetBits(8);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        INT i;
        stream.PutBits(sequence_header_code, 32);
        stream.PutBits(horizontal_size_value, 12);
        stream.PutBits(vertical_size_value, 12);
        stream.PutBits(aspect_ratio_information, 4);
        stream.PutBits(frame_rate_code, 4);
        stream.PutBits(bit_rate_value, 18);
        stream.PutBits(1, 1);
        stream.PutBits(vbv_buffer_size_value, 10);
        stream.PutBits(constrained_parameters_flag, 1);
        stream.PutBits(load_intra_quantiser_matrix, 1);
        if (load_intra_quantiser_matrix)
            for (i = 0; i < 64; i++)
                stream.PutBits(intra_quantiser_matrix[i], 8);
        stream.PutBits(load_non_intra_quantiser_matrix, 1);
        if (load_non_intra_quantiser_matrix)
            for (i = 0; i < 64; i++)
                stream.PutBits(non_intra_quantiser_matrix[i], 8);
        return 0;
    }
};

/*******************************************************************************
 *
 * Class: SEQUENCE_END
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a sequence end code.  Contains member functions to
 *   parse (Read) a bit stream and to output (Write) a bit stream.
 *
 *******************************************************************************/
class SEQUENCE_END {
public: /* public data */
    ULONG sequence_end_code; /* 32 bits */
public: /* public methods */
    SEQUENCE_END() {memset(this, 0, sizeof(SEQUENCE_END)); return;}
    LRESULT Read(STREAM& stream) {
        sequence_end_code = stream.GetBits(32);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(sequence_end_code, 32);
        return 0;
    }
};
```

```
/*******************************************************************************
 *
 * Class: SEQUENCE_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a sequence extension header.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 *******************************************************************************/
class SEQUENCE_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE sequence_extension_id; /* 4 bits */
    BYTE profile_and_level_indication; /* 8 bits */
    BYTE progressive_sequence; /* 1 bit */
    enum CHROMA_FORMAT_ENUM {_4_2_0=1, _4_2_2=2, _4_4_4=3} chroma_format; /* 2 bits */
    BYTE horizontal_size_extension; /* 2 bits */
    BYTE vertical_size_extension; /* 2 bits */
    USHORT bit_rate_extension; /* 12 bits */
    BYTE vbv_buffer_size_extension; /* 8 bits */
    BYTE low_delay; /* 1 bit */
    BYTE frame_rate_extension_n; /* 2 bits */
    BYTE frame_rate_extension_d; /* 5 bits */
public: /* public methods */
    SEQUENCE_EXTENSION() {memset(this, 0, sizeof(SEQUENCE_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        sequence_extension_id = (BYTE)stream.GetBits(4);
        profile_and_level_indication = (BYTE)stream.GetBits(8);
        progressive_sequence = (BYTE)stream.GetBits(1);
        chroma_format = (CHROMA_FORMAT_ENUM)stream.GetBits(2);
        horizontal_size_extension = (BYTE)stream.GetBits(2);
        vertical_size_extension = (BYTE)stream.GetBits(2);
        bit_rate_extension = (USHORT)stream.GetBits(12);
        if (stream.GetBits(1) == 0) return -1;
        vbv_buffer_size_extension = (BYTE)stream.GetBits(8);
        low_delay = (BYTE)stream.GetBits(1);
        frame_rate_extension_n = (BYTE)stream.GetBits(2);
        frame_rate_extension_d = (BYTE)stream.GetBits(5);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(sequence_extension_id, 4);
        stream.PutBits(profile_and_level_indication, 8);
        stream.PutBits(progressive_sequence, 1);
        stream.PutBits(chroma_format, 2);
        stream.PutBits(horizontal_size_extension, 2);
        stream.PutBits(vertical_size_extension, 2);
        stream.PutBits(bit_rate_extension, 12);
        stream.PutBits(1, 1);
        stream.PutBits(vbv_buffer_size_extension, 8);
        stream.PutBits(low_delay, 1);
        stream.PutBits(frame_rate_extension_n, 2);
        stream.PutBits(frame_rate_extension_d, 5);
        return 0;
    }
};

/*******************************************************************************
 *
 * Class: SEQUENCE_DISPLAY_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a sequence display extension header.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 *******************************************************************************/
class SEQUENCE_DISPLAY_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE sequence_display_extension_id; /* 4 bits */
    enum VIDEO_FORMAT_ENUM {component=0, PAL=1, NTSC=2, SECAM=3, MAC=4, unspec=5} video_format; /* 3 bits */
    BYTE colour_description; /* 1 bit */
    BYTE colour_primaries; /* 8 bits */
    BYTE transfer_characteristics; /* 8 bits */
    BYTE matrix_coefficients; /* 8 bits */
    USHORT display_horizontal_size; /* 14 bits */
    USHORT display_vertical_size; /* 14 bits */
public: /* public methods */
    SEQUENCE_DISPLAY_EXTENSION() {memset(this, 0, sizeof(SEQUENCE_DISPLAY_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        sequence_display_extension_id = (BYTE)stream.GetBits(4);
        video_format = (VIDEO_FORMAT_ENUM)stream.GetBits(3);
        colour_description = (BYTE)stream.GetBits(1);
        if (colour_description) {
            colour_primaries = (BYTE)stream.GetBits(8);
            transfer_characteristics = (BYTE)stream.GetBits(8);
            matrix_coefficients = (BYTE)stream.GetBits(8);
        }
        display_horizontal_size = (USHORT)stream.GetBits(14);
        if (stream.GetBits(1) == 0) return -1;
        display_vertical_size = (USHORT)stream.GetBits(14);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(sequence_display_extension_id, 4);
        stream.PutBits(video_format, 3);
        stream.PutBits(colour_description, 1);
        if (colour_description) {
            stream.PutBits(colour_primaries, 8);
            stream.PutBits(transfer_characteristics, 8);
            stream.PutBits(matrix_coefficients, 8);
        }
        stream.PutBits(display_horizontal_size, 14);
        stream.PutBits(1, 1);
        stream.PutBits(display_vertical_size, 14);
        return 0;
    }
};

/*******************************************************************************
 *
 * Class: SEQUENCE_SCALABLE_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a sequence scalable extension.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 *******************************************************************************/
class SEQUENCE_SCALABLE_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE sequence_scalable_extension_id; /* 4 bits */
    enum SCALABLE_MODE_ENUM {DATA_PARTITIONING=0, SPATIAL_SCALABILITY=1, SNR_SCALABILITY=2, TEMPORAL_SCALABILITY=3}
        scalable_mode; /* 2 bits */
    BYTE layer_id; /* 4 bits */
    USHORT lower_layer_prediction_horizontal_size; /* 14 bits */
    USHORT lower_layer_prediction_vertical_size; /* 14 bits */
    BYTE horizontal_subsampling_factor_m; /* 5 bits */
    BYTE horizontal_subsampling_factor_n; /* 5 bits */
    BYTE vertical_subsampling_factor_m; /* 5 bits */
    BYTE vertical_subsampling_factor_n; /* 5 bits */
    BYTE picture_mux_enable; /* 1 bit */
    BYTE mux_to_progressive_sequence; /* 1 bit */
    BYTE picture_mux_order; /* 3 bits */
    BYTE picture_mux_factor; /* 3 bits */
public: /* public methods */
    SEQUENCE_SCALABLE_EXTENSION() {memset(this, 0, sizeof(SEQUENCE_SCALABLE_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        sequence_scalable_extension_id = (BYTE)stream.GetBits(4);
        scalable_mode = (SCALABLE_MODE_ENUM)stream.GetBits(2);
        layer_id = (BYTE)stream.GetBits(4);
        if (scalable_mode == SPATIAL_SCALABILITY) {
            lower_layer_prediction_horizontal_size = (USHORT)stream.GetBits(14);
            if (stream.GetBits(0) == 0) return -1;
            lower_layer_prediction_vertical_size = (USHORT)stream.GetBits(14);
            horizontal_subsampling_factor_m = (BYTE)stream.GetBits(5);
            horizontal_subsampling_factor_n = (BYTE)stream.GetBits(5);
            vertical_subsampling_factor_m = (BYTE)stream.GetBits(5);
            vertical_subsampling_factor_n = (BYTE)stream.GetBits(5);
        }
        if (scalable_mode == TEMPORAL_SCALABILITY) {
            picture_mux_enable = (BYTE)stream.GetBits(1);
            if (picture_mux_enable) mux_to_progressive_sequence = (BYTE)stream.GetBits(1);
            picture_mux_order = (BYTE)stream.GetBits(3);
            picture_mux_factor = (BYTE)stream.GetBits(3);
        }
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(sequence_scalable_extension_id, 4);
        stream.PutBits(scalable_mode, 2);
        stream.PutBits(layer_id, 4);
        if (scalable_mode == SPATIAL_SCALABILITY) {
            stream.PutBits(lower_layer_prediction_horizontal_size, 14);
            stream.PutBits(1, 1);
            stream.PutBits(lower_layer_prediction_vertical_size, 14);
            stream.PutBits(horizontal_subsampling_factor_m, 5);
            stream.PutBits(horizontal_subsampling_factor_n, 5);
            stream.PutBits(vertical_subsampling_factor_m, 5);
            stream.PutBits(vertical_subsampling_factor_n, 5);
        }
        if (scalable_mode == TEMPORAL_SCALABILITY) {
            stream.PutBits(picture_mux_enable, 1);
            if (picture_mux_enable) stream.PutBits(mux_to_progressive_sequence, 1);
            stream.PutBits(picture_mux_order, 3);
            stream.PutBits(picture_mux_factor, 3);
        }
        return 0;
    }
};

/*******************************************************************************
 *
 * Class: GROUP_OF_PICTURES_HEADER
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a group fo pictures header.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 *******************************************************************************/
class GROUP_OF_PICTURES_HEADER {
public: /* public data */
    ULONG group_start_code; /* 32 bits */
    ULONG time_code; /* 25 bits */
    BYTE closed_gop; /* 1 bit */
    BYTE broken_link; /* 1 bit */
public: /* public methods */
    GROUP_OF_PICTURES_HEADER() {memset(this, 0, sizeof(GROUP_OF_PICTURES_HEADER)); return;}
    LRESULT Read(STREAM& stream) {
        group_start_code = stream.GetBits(32);
        time_code = stream.GetBits(25);
        closed_gop = (BYTE)stream.GetBits(1);
        broken_link = (BYTE)stream.GetBits(1);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(group_start_code, 32);
        stream.PutBits(time_code, 25);
        stream.PutBits(closed_gop, 1);
        stream.PutBits(broken_link, 1);
        return 0;
    }
};
```

```
/********************************************************************************
 *
 * Class: PICTURE_HEADER
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a picture header.  Contains member functions to
 *   parse (Read) a bit stream and to output (Write) a bit stream.
 *
 ********************************************************************************/
class PICTURE_HEADER {
public: /* public data */
    ULONG picture_start_code; /* 32 bits */
    USHORT temporal_reference; /* 10 bits */
    enum PICTURE_CODING_TYPE_ENUM {I_PICTURE=1, P_PICTURE=2, B_PICTURE=3} picture_coding_type; /* 3 bits */
    USHORT vbv_delay; /* 16 bits */
    BYTE full_pel_forward_vector; /* 1 bit */
    BYTE forward_f_code; /* 3 bits */
    BYTE full_pel_backward_vector; /* 1 bit */
    BYTE backward_f_code; /* 3 bits */
    BYTE extra_bit_picture; /* 1 bit */
public: /* public methods */
    PICTURE_HEADER() {memset(this, 0, sizeof(PICTURE_HEADER)); return;}
    LRESULT Read(STREAM& stream) {
        picture_start_code = stream.GetBits(32);
        temporal_reference = (USHORT)stream.GetBits(10);
        picture_coding_type = (PICTURE_CODING_TYPE_ENUM)stream.GetBits(3);
        vbv_delay = (USHORT)stream.GetBits(16);
        if (picture_coding_type == P_PICTURE || picture_coding_type == B_PICTURE) {
            full_pel_forward_vector = (BYTE)stream.GetBits(1);
            forward_f_code = (BYTE)stream.GetBits(3);
        }
        if (picture_coding_type == B_PICTURE) {
            full_pel_backward_vector = (BYTE)stream.GetBits(1);
            backward_f_code = (BYTE)stream.GetBits(3);
        }
        if (extra_bit_picture = (BYTE)stream.GetBits(1)) return -1;
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(picture_start_code, 32);
        stream.PutBits(temporal_reference, 10);
        stream.PutBits(picture_coding_type, 3);
        stream.PutBits(vbv_delay, 16);
        if (picture_coding_type == P_PICTURE || picture_coding_type == B_PICTURE) {
            stream.PutBits(full_pel_forward_vector, 1);
            stream.PutBits(forward_f_code, 3);
        }
        if (picture_coding_type == B_PICTURE) {
            stream.PutBits(full_pel_backward_vector, 1);
            stream.PutBits(backward_f_code, 3);
        }
        stream.PutBits(extra_bit_picture, 1);
        return 0;
    }
};

/********************************************************************************
 *
 * Class: PICTURE_CODING_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a picture coding extension.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 ********************************************************************************/
class PICTURE_CODING_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE picture_coding_extension_id; /* 4 bits */
    BYTE f_code[2][2]; /* 4 bits each */
    BYTE intra_dc_precision; /* 2 bits */
    enum PICTURE_STRUCTURE_ENUM {TOP_FIELD=1, BOTTOM_FIELD=2, FRAME_PICTURE=3} picture_structure; /* 2 bits */
    BYTE top_field_first; /* 1 bit */
    BYTE frame_pred_frame_dct; /* 1 bit */
    BYTE concealment_motion_vectors; /* 1 bit */
    BYTE q_scale_type; /* 1 bit */
    BYTE intra_vlc_format; /* 1 bit */
    BYTE alternate_scan; /* 1 bit */
    BYTE repeat_first_field; /* 1 bit */
    BYTE chroma_420_type; /* 1 bit */
    BYTE progressive_frame; /* 1 bit */
    BYTE composite_display_flag; /* 1 bit */
    BYTE v_axis; /* 1 bit */
    BYTE field_sequence; /* 3 bits */
    BYTE sub_carrier; /* 1 bit */
    BYTE burst_amplitude; /* 7 bits */
    BYTE sub_carrier_phase; /* 8 bits */
public: /* public methods */
    PICTURE_CODING_EXTENSION() {memset(this, 0, sizeof(PICTURE_CODING_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        picture_coding_extension_id = (BYTE)stream.GetBits(4);
        f_code[0][0] = (BYTE)stream.GetBits(4); /* forward horizontal */
        f_code[0][1] = (BYTE)stream.GetBits(4); /* forward vertical */
        f_code[1][0] = (BYTE)stream.GetBits(4); /* backward horizontal */
        f_code[1][1] = (BYTE)stream.GetBits(4); /* backward vertical */
        intra_dc_precision = (BYTE)stream.GetBits(2);
        picture_structure = (PICTURE_STRUCTURE_ENUM)stream.GetBits(2);
        top_field_first = (BYTE)stream.GetBits(1);
        frame_pred_frame_dct = (BYTE)stream.GetBits(1);
        concealment_motion_vectors = (BYTE)stream.GetBits(1);
        q_scale_type = (BYTE)stream.GetBits(1);
        intra_vlc_format = (BYTE)stream.GetBits(1);
        alternate_scan = (BYTE)stream.GetBits(1);
        repeat_first_field = (BYTE)stream.GetBits(1);
        chroma_420_type = (BYTE)stream.GetBits(1);
        progressive_frame = (BYTE)stream.GetBits(1);
        composite_display_flag = (BYTE)stream.GetBits(1);
        if (composite_display_flag == 1) {
            v_axis = (BYTE)stream.GetBits(1);
            field_sequence = (BYTE)stream.GetBits(3);
            sub_carrier = (BYTE)stream.GetBits(1);
            burst_amplitude = (BYTE)stream.GetBits(7);
            sub_carrier_phase = (BYTE)stream.GetBits(8);
        }
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(picture_coding_extension_id, 4);
        stream.PutBits(f_code[0][0], 4);
        stream.PutBits(f_code[0][1], 4);
        stream.PutBits(f_code[1][0], 4);
        stream.PutBits(f_code[1][1], 4);
        stream.PutBits(intra_dc_precision, 2);
        stream.PutBits(picture_structure, 2);
        stream.PutBits(top_field_first, 1);
        stream.PutBits(frame_pred_frame_dct, 1);
        stream.PutBits(concealment_motion_vectors, 1);
        stream.PutBits(q_scale_type, 1);
        stream.PutBits(intra_vlc_format, 1);
        stream.PutBits(alternate_scan, 1);
        stream.PutBits(repeat_first_field, 1);
        stream.PutBits(chroma_420_type, 1);
        stream.PutBits(progressive_frame, 1);
        stream.PutBits(composite_display_flag, 1);
        if (composite_display_flag == 1) {
            stream.PutBits(v_axis, 1);
            stream.PutBits(field_sequence, 3);
            stream.PutBits(sub_carrier, 1);
            stream.PutBits(burst_amplitude, 7);
            stream.PutBits(sub_carrier_phase, 8);
        }
        return 0;
    }
};

/*******************************************************************************
 *
 * Class: QUANT_MATRIX_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a quantization matrix extension.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 ******************************************************************************/
class QUANT_MATRIX_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE quant_matrix_extension_id; /* 4 bits */
    BYTE load_intra_quantiser_matrix; /* 1 bit */
    BYTE intra_quantiser_matrix[64];
    BYTE load_non_intra_quantiser_matrix; /* 1 bit */
    BYTE non_intra_quantiser_matrix[64];
    BYTE load_chroma_intra_quantiser_matrix; /* 1 bit */
    BYTE chroma_intra_quantiser_matrix[64];
    BYTE load_chroma_non_intra_quantiser_matrix; /* 1 bit */
    BYTE chroma_non_intra_quantiser_matrix[64];
public: /* public methods */
    QUANT_MATRIX_EXTENSION() {memset(this, 0, sizeof(QUANT_MATRIX_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        INT i;
        extension_start_code = stream.GetBits(32);
        quant_matrix_extension_id = (BYTE)stream.GetBits(4);
        if (load_intra_quantiser_matrix = (BYTE)stream.GetBits(1))
            for (i = 0; i < 64; i++)
                intra_quantiser_matrix[i] = (BYTE)stream.GetBits(8);
        if (load_non_intra_quantiser_matrix = (BYTE)stream.GetBits(1))
            for (i = 0; i < 64; i++)
                non_intra_quantiser_matrix[i] = (BYTE)stream.GetBits(8);
        if (load_chroma_intra_quantiser_matrix = (BYTE)stream.GetBits(1))
            for (i = 0; i < 64; i++)
                chroma_intra_quantiser_matrix[i] = (BYTE)stream.GetBits(8);
        if (load_chroma_non_intra_quantiser_matrix = (BYTE)stream.GetBits(1))
            for (i = 0; i < 64; i++)
                chroma_non_intra_quantiser_matrix[i] = (BYTE)stream.GetBits(8);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        INT i;
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(quant_matrix_extension_id, 4);
        stream.PutBits(load_intra_quantiser_matrix, 1);
        if (load_intra_quantiser_matrix)
            for (i = 0; i < 64; i++)
                stream.PutBits(intra_quantiser_matrix[i], 8);
        stream.PutBits(load_non_intra_quantiser_matrix, 1);
            for (i = 0; i < 64; i++)
                stream.PutBits(non_intra_quantiser_matrix[i], 8);
        stream.PutBits(load_chroma_intra_quantiser_matrix, 1);
            for (i = 0; i < 64; i++)
                stream.PutBits(chroma_intra_quantiser_matrix[i], 8);
        stream.PutBits(load_chroma_non_intra_quantiser_matrix, 1);
            for (i = 0; i < 64; i++)
                stream.PutBits(chroma_non_intra_quantiser_matrix[i], 8);
        return 0;
    }
};


/*******************************************************************************
 *
 * Class: PICTURE_DISPLAY_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a picture display extension.  Contains member
 *   functions to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 * Note that the Write function requires parameters from other structures --
 *   this is because the number of offsets is not constant.
 *
 ******************************************************************************/
```

```
class PICTURE_DISPLAY_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE picture_display_extension_id; /* 4 bits */
    USHORT frame_centre_horizontal_offset[3]; /* 16 bits */
    USHORT frame_centre_vertical_offset[3]; /* 16 bits */
public: /* public methods */
    PICTURE_DISPLAY_EXTENSION() {memset(this, 0, sizeof(PICTURE_DISPLAY_EXTENSION)); return;}
    LRESULT Read(STREAM& stream, BYTE progressive_sequence, PICTURE_CODING_EXTENSION::PICTURE_STRUCTURE_ENUM picture_structure,
                    BYTE repeat_first_field) {
        INT i, number_of_frame_centre_offsets;
        if (progressive_sequence == 1 || picture_structure != PICTURE_CODING_EXTENSION::FRAME_PICTURE)
                number_of_frame_centre_offsets = 1;
        else if (repeat_first_field == 1)
            number_of_frame_centre_offsets = 3;
        else
            number_of_frame_centre_offsets = 2;
        for (i = 0; i < number_of_frame_centre_offsets; i++) {
            frame_centre_horizontal_offset[i] = (USHORT)stream.GetBits(16);
            if (stream.GetBits(1) == 0) return -1;
            frame_centre_vertical_offset[i] = (USHORT)stream.GetBits(16);
            if (stream.GetBits(1) == 0) return -1;
        }
        return 0;
    }
    LRESULT Write(STREAM& stream, BYTE progressive_sequence, PICTURE_CODING_EXTENSION::PICTURE_STRUCTURE_ENUM picture_structure,
                    BYTE repeat_first_field) {
        INT i, number_of_frame_centre_offsets;
        if (progressive_sequence == 1 || picture_structure != PICTURE_CODING_EXTENSION::FRAME_PICTURE)
                number_of_frame_centre_offsets = 1;
        else if (repeat_first_field == 1)
            number_of_frame_centre_offsets = 3;
        else
            number_of_frame_centre_offsets = 2;
        for (i = 0; i < number_of_frame_centre_offsets; i++) {
            stream.PutBits(frame_centre_horizontal_offset[i], 16);
            stream.PutBits(1, 1);
            stream.PutBits(frame_centre_vertical_offset[i], 16);
            stream.PutBits(1, 1);
        }
        return 0;
    }
};

/********************************************************************************
 *
 * Class: PICTURE_TEMPORAL_SCALABLE_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a picture temporal scalable extension.  Contains
 *   member functions to parse (Read) a stream and to output (Write) a stream.
 *
 ********************************************************************************/
class PICTURE_TEMPORAL_SCALABLE_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE picture_temporal_scalable_extension_id; /* 4 bits */
    BYTE reference_select_code; /* 2 bits */
    USHORT forward_temporal_reference; /* 10 bits */
    USHORT backward_temporal_reference; /* 10 bits */
public: /* public methods */
    PICTURE_TEMPORAL_SCALABLE_EXTENSION() {memset(this, 0, sizeof(PICTURE_TEMPORAL_SCALABLE_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        picture_temporal_scalable_extension_id = (BYTE)stream.GetBits(4);
        reference_select_code = (BYTE)stream.GetBits(2);
        forward_temporal_reference = (USHORT)stream.GetBits(10);
        backward_temporal_reference = (USHORT)stream.GetBits(10);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(picture_temporal_scalable_extension_id, 4);
        stream.PutBits(reference_select_code, 2);
        stream.PutBits(forward_temporal_reference, 10);
        stream.PutBits(backward_temporal_reference, 10);
        return 0;
    }
};

/********************************************************************************
 *
 * Class: PICTURE_SPATIAL_SCALABLE_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a picture spatial scalable extension.  Contains
 *   member functions to parse (Read) a stream and to output (Write) a stream.
 *
 ********************************************************************************/
class PICTURE_SPATIAL_SCALABLE_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE picture_spatial_scalable_extension_id; /* 4 bits */
    USHORT lower_layer_temporal_reference; /* 10 bits */
    USHORT lower_layer_horizontal_offset; /* 15 bits */
    USHORT lower_layer_vertical_offset; /* 15 bits */
    BYTE spatial_temporal_weight_code_table_index; /* 2 bits */
    BYTE lower_layer_progressive_frame; /* 1 bit */
    BYTE lower_layer_deinterlaced_field_select; /* 1 bit */
public: /* public methods */
    PICTURE_SPATIAL_SCALABLE_EXTENSION() {memset(this, 0, sizeof(PICTURE_SPATIAL_SCALABLE_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        picture_spatial_scalable_extension_id = (BYTE)stream.GetBits(4);
        lower_layer_temporal_reference = (USHORT)stream.GetBits(10);
        if (stream.GetBits(1) == 0) return -1;
        lower_layer_horizontal_offset = (USHORT)stream.GetBits(15);
        if (stream.GetBits(1) == 0) return -1;
        lower_layer_vertical_offset = (USHORT)stream.GetBits(15);
        spatial_temporal_weight_code_table_index = (BYTE)stream.GetBits(2);
        lower_layer_progressive_frame = (BYTE)stream.GetBits(1);
        lower_layer_deinterlaced_field_select = (BYTE)stream.GetBits(1);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(picture_spatial_scalable_extension_id, 4);
        stream.PutBits(lower_layer_temporal_reference, 10);
        stream.PutBits(1, 1);
        stream.PutBits(lower_layer_horizontal_offset, 15);
        stream.PutBits(1, 1);
        stream.PutBits(lower_layer_vertical_offset, 15);
        stream.PutBits(spatial_temporal_weight_code_table_index, 2);
        stream.PutBits(lower_layer_progressive_frame, 1);
        stream.PutBits(lower_layer_deinterlaced_field_select, 1);
        return 0;
    }
};

/*******************************************************************************
 *
 * Class: COPYRIGHT_EXTENSION
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a copyright extension.  Contains member functions
 *   to parse (Read) a bit stream and to output (Write) a bit stream.
 *
 ******************************************************************************/
class COPYRIGHT_EXTENSION {
public: /* public data */
    ULONG extension_start_code; /* 32 bits */
    BYTE copyright_extension_id; /* 4 bits */
    BYTE copyright_flag; /* 1 bit */
    BYTE copyright_identifier; /* 8 bits */
    BYTE original_or_copy; /* 1 bit */
    BYTE reserved; /* 7 bits */
    ULONG copyright_number_1; /* 20 bits */
    ULONG copyright_number_2; /* 22 bits */
    ULONG copyright_number_3; /* 22 bits */
public: /* public methods */
    COPYRIGHT_EXTENSION() {memset(this, 0, sizeof(COPYRIGHT_EXTENSION)); return;}
    LRESULT Read(STREAM& stream) {
        extension_start_code = stream.GetBits(32);
        copyright_extension_id = (BYTE)stream.GetBits(4);
        copyright_flag = (BYTE)stream.GetBits(1);
        copyright_identifier = (BYTE)stream.GetBits(8);
        original_or_copy = (BYTE)stream.GetBits(1);
        reserved = (BYTE)stream.GetBits(7);
        if (stream.GetBits(1) == 0) return -1;
        copyright_number_1 = stream.GetBits(20);
        if (stream.GetBits(1) == 0) return -1;
        copyright_number_2 = stream.GetBits(22);
        if (stream.GetBits(1) == 0) return -1;
        copyright_number_3 = stream.GetBits(22);
        return 0;
    }
    LRESULT Write(STREAM& stream) {
        stream.PutBits(extension_start_code, 32);
        stream.PutBits(copyright_extension_id, 4);
        stream.PutBits(copyright_flag, 1);
        stream.PutBits(copyright_identifier, 8);
        stream.PutBits(original_or_copy, 1);
        stream.PutBits(reserved, 7);
        stream.PutBits(1, 1);
        stream.PutBits(copyright_number_1, 20);
        stream.PutBits(1, 1);
        stream.PutBits(copyright_number_2, 22);
        stream.PutBits(1, 1);
        stream.PutBits(copyright_number_3, 22);
        return 0;
    }
};

/* Structure collects all MPEG-2 video headers;
   This is used in processing slices, macroblocks, and blocks */
typedef struct _MPEG2HEADERS {
    SEQUENCE_HEADER sequence_header;
    SEQUENCE_END sequence_end;
    SEQUENCE_EXTENSION sequence_extension;
    SEQUENCE_DISPLAY_EXTENSION sequence_display_extension;
    SEQUENCE_SCALABLE_EXTENSION sequence_scalable_extension;
    GROUP_OF_PICTURES_HEADER group_of_pictures_header;
    PICTURE_HEADER picture_header;
    PICTURE_CODING_EXTENSION picture_coding_extension;
    QUANT_MATRIX_EXTENSION quant_matrix_extension;
    PICTURE_DISPLAY_EXTENSION picture_display_extension;
    PICTURE_TEMPORAL_SCALABLE_EXTENSION picture_temporal_scalable_extension;
    PICTURE_SPATIAL_SCALABLE_EXTENSION picture_spatial_scalable_extension;
    COPYRIGHT_EXTENSION copyright_extension;
} MPEG2HEADERS, *LPMPEG2HEADERS;

class SLICE; typedef SLICE *LPSLICE;
class MACROBLOCK; typedef MACROBLOCK *LPMACROBLOCK;
class BLOCK; typedef BLOCK *LPBLOCK;
/* Context header is used for processing slices, macroblocks, and blocks
   Contains a complete set of MPEG-2 video headers and read/write streams
   -May also contain a slice (if working with a macroblock or block)
   -May also contain a macroblock (if working with a block)
*/
typedef struct _MPEG2CONTEXT {
    STREAM *rstm, *wstm;
    LPMPEG2HEADERS hdrs;
    LPSLICE slc;
    LPMACROBLOCK mbl;
} MPEG2CONTEXT, *LPMPEG2CONTEXT;
```

```
/*******************************************************************************
 *
 * Class: SLICE
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a slice.  Contains one member function to read
 *   a bit stream from the context's read-stream, transcode it, and write
 *   it to the context's write-stream.
 *
 *******************************************************************************/
class SLICE {
public: /* public data */
    MPEG2CONTEXT c;
    ULONG slice_start_code; /* 32 bits */
    BYTE slice_vertical_position_extension; /* 3 bits */
    USHORT slice_vertical_position;
    BYTE priority_breakpoint; /* 7 bits */
    BYTE quantiser_scale_code; /* 5 bits */
    BYTE intra_slice_flag; /* 1 bit */
    BYTE intra_slice; /* 1 bit */
    BYTE reserved_bits; /* 7 bits */
public: /* slice-processing procedures */
    LRESULT RW_xcode(INT);
};

/*******************************************************************************
 *
 * Class: MACROBLOCK
 *
 * Description: Class contains member variables mirroring the MPGE-2 video
 *   standard fields within a macroblock, and adds fields for some structures
 *   below macroblocks e.g. macroblock_modes, motion_vectors, motion_vector, and
 *   coded_block_pattern.  Contains a public member function to transcode the
 *   macroblock by parsing the macroblock in the context's read stream, and
 *   outputting it to the context's write stream.  Also contains private member
 *   functions for parsing sub-macroblock structures.
 *
 *******************************************************************************/
class MACROBLOCK {
public:
    MPEG2CONTEXT c;
    USHORT macroblock_address_increment; /* 11 bits */
    BYTE macroblock_quant, macroblock_motion_forward, macroblock_motion_backward, macroblock_pattern, macroblock_intra;
    BYTE spatial_temporal_weight_code_flag, spatial_temporal_weight_code, frame_motion_type, field_motion_type;
    enum MV_FORMAT_ENUM {FIELD = 0, FRAME = 1} mv_format;
    BYTE decode_dct_type, dct_type;
    BYTE motion_vector_count, dmv;
    BYTE quantiser_scale_code;
    BYTE motion_vertical_field_select[2][2];
    INT motion_code[2][2][2], motion_residual[2][2][2], dmvector[2];
    BYTE cbp, coded_block_pattern_1, coded_block_pattern_2, pattern_code[12], block_count;
public:
    LRESULT RW_xcode(INT);
private:
    LRESULT RW_MACROBLOCK_MODES();
    LRESULT RW_MOTION_VECTORS(INT);
    LRESULT RW_MOTION_VECTOR(INT, INT);
    LRESULT RW_CODED_BLOCK_PATTERN();
};

/*******************************************************************************
 *
 * Class: BLOCK
 *
 * Description: Class contains member variables mirroring the MPEG-2 video
 *   standard fields within a block, and adds fields for processing the block
 *   at different stages (scanning, rle, quantization).  Contains member
 *   functions for reading/writing blocks from/to context streams and for
 *   forward/inverse quantizing a block.
 *
 *******************************************************************************/
class BLOCK {
public:
    MPEG2CONTEXT c;
    ULONG quant_table, i, cc;
    ULONG dc_dct_size, dc_dct_differential;
    INT QFS[64], QF[8][8], F__[8][8];
    INT intra_dc_mult, quantiser_scale;
public: /* block-processing procedures */
    inline INT Signum(INT x) {return (x>0 ? 1 : (x?-1:0) );}
    LRESULT Read();
    LRESULT InvQuant();
    LRESULT Quant(INT);
    LRESULT Write();
};

/* This is used for scanning blocks; the scanning orders are from ISO/IEC 13818-2 */
const ULONG scan[2][8][8]={{{ 0, 1, 5, 6,14,15,27,28}, { 2, 4, 7,13,16,26,29,42},
                            { 3, 8,12,17,25,30,41,43}, { 9,11,18,24,31,40,44,53},
                            {10,19,23,32,39,45,52,54}, {20,22,33,38,46,51,55,60},
                            {21,34,37,47,50,56,59,61}, {35,36,48,49,57,58,62,63}},
                           {{ 0, 4, 6,20,22,36,38,52}, { 1, 5, 7,21,23,37,39,53},
                            { 2, 8,19,24,34,40,50,54}, { 3, 9,18,25,35,41,51,55},
                            {10,17,26,30,42,46,56,60}, {11,16,27,31,43,47,57,61},
                            {12,15,28,32,44,48,58,62}, {13,14,29,33,45,49,59,63}}};

/* This is used to translate quantiser_scale_code into quantiser_scale (ISO/IEC 13818-2) */
const ULONG quantiser_scale[2][32] =  {{ 0, 2, 4, 6, 8,10,12,14,16,18,20,22,24,26,28,30,
                                         32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62},
                                       { 1, 2, 3, 4, 5, 6, 7, 8,10,12,14,16,18,20,22,24,
                                         28,32,36,40,44,48,52,56,64,72,80,88,96,104,112}};

/* These are the default quantisation matrices, taken from ISO/IEC 13818-2 */
const int W[4][8][8] = {
    {{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}},
    {{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}},
    {{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}},
    {{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}}
};


/*
    Huffman coding tables (dictionary)
    adapted from ISO/IEC 13818-2 Annex B
    -note that this algorithm is extremely inefficient, but it was the quickest (and most accurate) to code in C++.
    -A better solution would be to use small linked arrays for each bit, leading to the end value.

    0xaaaabbbbcccccccc
      \__/\__/\______/
       |   |     '------value
       |   '------------mask
       '----------------code
*/
const unsigned __int64 huffman1[] = {
    0x8000800000000001, 0x6000E00000000002, 0x4000E00000000003, 0x3000F00000000004, 0x2000F00000000005, 0x1800F80000000006,
    0x1000F80000000007, 0x0E00FE0000000008, 0x0C00FE0000000009, 0x0B00FF000000000A, 0x0A00FF000000000B, 0x0900FF000000000C,
    0x0800FF000000000D, 0x0700FF000000000E, 0x0600FF000000000F, 0x05C0FFC000000010, 0x0580FFC000000011, 0x0540FFC000000012,
    0x0500FFC000000013, 0x04C0FFC000000014, 0x0480FFC000000015, 0x0460FFE000000016, 0x0440FFE000000017, 0x0420FFE000000018,
    0x0400FFE000000019, 0x03E0FFE00000001A, 0x03C0FFE00000001B, 0x03A0FFE00000001C, 0x0380FFE00000001D, 0x0360FFE00000001E,
    0x0340FFE00000001F, 0x0320FFE000000020, 0x0300FFE000000021, 0x0100FFE000000063, 0};
const unsigned __int64 huffman2[] = {0x8000800000000002, 0x4000C00000000022, 0};
const unsigned __int64 huffman3[] = {
    0x8000800000000014, 0x4000C00000000004, 0x2000E00000000010, 0x1800F80000000002, 0x1000F80000000034, 0x0800F80000000024,
    0x0400FC0000000022, 0};
const unsigned __int64 huffman4[] = {
    0x8000C00000000018, 0xC000C0000000001C, 0x4000E00000000008, 0x6000E0000000000C, 0x2000F00000000010, 0x3000F00000000014,
    0x1800F80000000002, 0x1000F8000000003C, 0x0C00FC0000000034, 0x0800FC000000002C, 0x0400FC0000000022, 0};
const unsigned __int64 huffman9[] = {
    0xE000E0000000003C, 0xD000F00000000004, 0xC000F00000000008, 0xB000F00000000010, 0xA000F00000000020, 0x9800F8000000000C,
    0x9000F80000000030, 0x8800F80000000014, 0x8000F80000000028, 0x7800F8000000001C, 0x7000F8000000002C, 0x6800F80000000034,
    0x6000F80000000038, 0x5800F80000000001, 0x5000F8000000003D, 0x4800F80000000002, 0x4000F8000000003E, 0x3C00FC0000000018,
    0x3800FC0000000024, 0x3400FC0000000003, 0x3000FC000000003F, 0x2E00FE0000000005, 0x2C00FE0000000009, 0x2A00FE0000000011,
    0x2800FE0000000021, 0x2600FE0000000006, 0x2400FE000000000A, 0x2200FE0000000012, 0x2000FE0000000022, 0x1F00FF0000000007,
    0x1E00FF000000000B, 0x1D00FF0000000013, 0x1C00FF0000000023, 0x1B00FF000000000D, 0x1A00FF0000000031, 0x1900FF0000000015,
    0x1800FF0000000029, 0x1700FF000000000E, 0x1600FF0000000032, 0x1500FF0000000016, 0x1400FF000000002A, 0x1300FF000000000F,
    0x1200FF0000000033, 0x1100FF0000000017, 0x1000FF000000002B, 0x0F00FF0000000019, 0x0E00FF0000000025, 0x0D00FF000000001A,
    0x0C00FF0000000026, 0x0B00FF000000001D, 0x0A00FF000000002D, 0x0900FF0000000035, 0x0800FF0000000039, 0x0700FF000000001E,
    0x0600FF000000002E, 0x0500FF0000000036, 0x0400FF000000003A, 0x0380FF800000001F, 0x0300FF800000002F, 0x0280FF8000000037,
    0x0200FF800000003B, 0x0180FF800000001B, 0x0100FF8000000027, 0x0080FF8000000000, 0};
const unsigned __int64 huffman10[] = {
    0x0320FFE0FFFFFFF0, 0x0360FFE0FFFFFFF1, 0x03A0FFE0FFFFFFF2, 0x03E0FFE0FFFFFFF3, 0x0420FFE0FFFFFFF4, 0x0460FFE0FFFFFFF5,
    0x04C0FFC0FFFFFFF6, 0x0540FFC0FFFFFFF7, 0x05C0FFC0FFFFFFF8, 0x0700FF00FFFFFFF9, 0x0900FF00FFFFFFFA, 0x0B00FF00FFFFFFFB,
    0x0E00FE00FFFFFFFC, 0x1800F800FFFFFFFD, 0x3000F000FFFFFFFE, 0x6000E000FFFFFFFF, 0x8000800000000000, 0x4000E00000000001,
    0x2000F00000000002, 0x1000F80000000003, 0x0C00FE0000000004, 0x0A00FF0000000005, 0x0800FF0000000006, 0x0600FF0000000007,
    0x0580FFC000000008, 0x0500FFC000000009, 0x0480FFC00000000A, 0x0440FFE00000000B, 0x0400FFE00000000C, 0x03C0FFE00000000D,
    0x0380FFE00000000E, 0x0340FFE00000000F, 0x0300FFE000000010, 0};
const unsigned __int64 huffman11[] = {
    0xC000C000FFFFFFFF, 0x0000800000000000, 0x8000C00000000001, 0};
const unsigned __int64 huffman12[] = {
    0x8000E00000000000, 0x0000C00000000001, 0x4000C00000000002, 0xA000E00000000003, 0xC000E00000000004, 0xE000F00000000005,
    0xF000F80000000006, 0xF800FC0000000007, 0xFC00FE0000000008, 0xFE00FF0000000009, 0xFF00FF800000000A, 0xFF80FF800000000B, 0};
const unsigned __int64 huffman13[] = {
    0x0000C00000000000, 0x4000C00000000001, 0x8000C00000000002, 0xC000E00000000003, 0xE000F00000000004, 0xF000F80000000005,
    0xF800FC0000000006, 0xFC00FE0000000007, 0xFE00FF0000000008, 0xFF00FF8000000009, 0xFF80FFC00000000A, 0xFFC0FFC00000000B, 0};
const unsigned __int64 huffman14[] = {
    0x8000C000FFFFFFFF, 0xC000C00000000001, 0x6000E00000000101, 0x4000F00000000002, 0x5000F00000000201, 0x2800F80000000003,
    0x3800F80000000301, 0x3000F80000000401, 0x1800FC0000000102, 0x1C00FC0000000501, 0x1400FC0000000601, 0x1000FC0000000701,
    0x0C00FE0000000004, 0x0800FE0000000202, 0x0E00FE0000000801, 0x0A00FE0000000901, 0x0400FC0000000000, 0x2600FF0000000005,
    0x2100FF0000000006, 0x2500FF0000000103, 0x2400FF0000000302, 0x2700FF0000000A01, 0x2300FF0000000B01, 0x2200FF0000000C01,
    0x2000FF0000000D01, 0x0280FFC000000007, 0x0300FFC000000104, 0x02C0FFC000000203, 0x03C0FFC000000402, 0x0240FFC000000502,
    0x0380FFC000000E01, 0x0340FFC000000F01, 0x0200FFC000001001, 0x01D0FFF000000008, 0x0180FFF000000009, 0x0130FFF00000000A,
    0x0100FFF00000000B, 0x01B0FFF000000105, 0x0140FFF000000204, 0x01C0FFF000000303, 0x0120FFF000000403, 0x01E0FFF000000602,
    0x0150FFF000000702, 0x0110FFF000000802, 0x01F0FFF000001101, 0x01A0FFF000001201, 0x0190FFF000001301, 0x0170FFF000001401,
    0x0160FFF000001501, 0x00D0FFF80000000C, 0x00C8FFF80000000D, 0x00C0FFF80000000E, 0x00B8FFF80000000F, 0x00B0FFF800000106,
    0x00A8FFF800000107, 0x00A0FFF800000205, 0x0098FFF800000304, 0x0090FFF800000503, 0x0088FFF800000902, 0x0080FFF800000A02,
    0x00F8FFF800001601, 0x00F0FFF800001701, 0x00E8FFF800001801, 0x00E0FFF800001901, 0x00D8FFF800001A01, 0x007CFFFC00000010,
    0x0078FFFC00000011, 0x0074FFFC00000012, 0x0070FFFC00000013, 0x006CFFFC00000014, 0x0068FFFC00000015, 0x0064FFFC00000016,
    0x0060FFFC00000017, 0x005CFFFC00000018, 0x0058FFFC00000019, 0x0054FFFC0000001A, 0x0050FFFC0000001B, 0x004CFFFC0000001C,
    0x0048FFFC0000001D, 0x0044FFFC0000001E, 0x0040FFFC0000001F, 0x0030FFFE00000020, 0x002EFFFE00000021, 0x002CFFFE00000022,
    0x002AFFFE00000023, 0x0028FFFE00000024, 0x0026FFFE00000025, 0x0024FFFE00000026, 0x0022FFFE00000027, 0x0020FFFE00000028,
    0x003EFFFE00000108, 0x003CFFFE00000109, 0x003AFFFE0000010A, 0x0038FFFE0000010B, 0x0036FFFE0000010C, 0x0034FFFE0000010D,
    0x0032FFFE0000010E, 0x0013FFFF0000010F, 0x0012FFFF00000110, 0x0011FFFF00000111, 0x0010FFFF00000112, 0x0014FFFF00000603,
    0x001AFFFF00000B02, 0x0019FFFF00000C02, 0x0018FFFF00000D02, 0x0017FFFF00000E02, 0x0016FFFF00000F02, 0x0015FFFF00001002,
    0x001FFFFF00001B01, 0x001EFFFF00001C01, 0x001DFFFF00001D01, 0x001CFFFF00001E01, 0x001BFFFF00001F01, 0};
const unsigned __int64 huffman15[] = {
    0x6000F000FFFFFFFF, 0x8000C00000000001, 0x4000E00000000101, 0xC000E00000000002, 0x2800F80000000201, 0x7000F00000000003,
    0x3800F80000000301, 0x1800FC0000000401, 0x3000F80000000102, 0x1C00FC0000000501, 0x0C00FE0000000601, 0x0800FE0000000701,
    0xE000F80000000004, 0x0E00FE0000000202, 0x0A00FE0000000801, 0xF000FE0000000901, 0x0400FC0000000000, 0xE800F80000000005,
    0x1400FC0000000006, 0xF200FE0000000103, 0x2600FF0000000302, 0xF400FE0000000A01, 0x2100FF0000000B01, 0x2500FF0000000C01,
    0x2400FF0000000D01, 0x1000FC0000000007, 0x2700FF0000000104, 0xFC00FF0000000203, 0xFD00FF0000000402, 0x0200FF8000000502,
    0x0280FF8000000E01, 0x0380FF8000000F01, 0x0340FFC000001001, 0xF600FE0000000008, 0xF800FE0000000009, 0x2300FF000000000A,
    0x2200FF000000000B, 0x2000FF0000000105, 0x0300FFC000000204, 0x01C0FFF000000303, 0x0120FFF000000403, 0x01E0FFF000000602,
    0x0150FFF000000702, 0x0110FFF000000802, 0x01F0FFF000001101, 0x01A0FFF000001201, 0x0190FFF000001301, 0x0170FFF000001401,
    0x0160FFF000001501, 0xFA00FF000000000C, 0xFB00FF000000000D, 0xFE00FF000000000E, 0xFF00FF000000000F, 0x00B0FFF800000106,
    0x00A8FFF800000107, 0x00A0FFF800000205, 0x0098FFF800000304, 0x0090FFF800000503, 0x0088FFF800000902, 0x0080FFF800000A02,
    0x00F8FFF800001601, 0x00F0FFF800001701, 0x00E8FFF800001801, 0x00E0FFF800001901, 0x00D8FFF800001A01, 0x007CFFFC00000010,
    0x0078FFFC00000011, 0x0074FFFC00000012, 0x0070FFFC00000013, 0x006CFFFC00000014, 0x0068FFFC00000015, 0x0064FFFC00000016,
    0x0060FFFC00000017, 0x005CFFFC00000018, 0x0058FFFC00000019, 0x0054FFFC0000001A, 0x0050FFFC0000001B, 0x004CFFFC0000001C,
    0x0048FFFC0000001D, 0x0044FFFC0000001E, 0x0040FFFC0000001F, 0x0030FFFE00000020, 0x002EFFFE00000021, 0x002CFFFE00000022,
    0x002AFFFE00000023, 0x0028FFFE00000024, 0x0026FFFE00000025, 0x0024FFFE00000026, 0x0022FFFE00000027, 0x0020FFFE00000028,
    0x003EFFFE00000108, 0x003CFFFE00000109, 0x003AFFFE0000010A, 0x0038FFFE0000010B, 0x0036FFFE0000010C, 0x0034FFFE0000010D,
    0x0032FFFE0000010E, 0x0013FFFF0000010F, 0x0012FFFF00000110, 0x0011FFFF00000111, 0x0010FFFF00000112, 0x0014FFFF00000603,
    0x001AFFFF00000B02, 0x0019FFFF00000C02, 0x0018FFFF00000D02, 0x0017FFFF00000E02, 0x0016FFFF00000F02, 0x0015FFFF00001002,
    0x001FFFFF00001B01, 0x001EFFFF00001C01, 0x001DFFFF00001D01, 0x001CFFFF00001E01, 0x001BFFFF00001F01, 0};
const LPHUFFMANENTRY huffman[] = {NULL, (LPHUFFMANENTRY)huffman1, (LPHUFFMANENTRY)huffman2, (LPHUFFMANENTRY)huffman3,
    (LPHUFFMANENTRY)huffman4, NULL,NULL,NULL,NULL, (LPHUFFMANENTRY)huffman9, (LPHUFFMANENTRY)huffman10, (LPHUFFMANENTRY)huffman11,
    (LPHUFFMANENTRY)huffman12, (LPHUFFMANENTRY)huffman13, (LPHUFFMANENTRY)huffman14, (LPHUFFMANENTRY)huffman15};

#endif
```

```
#include "stream.h"
#include "mpeg2hdr.h"

/*******************************************************************************
 *
 * Function: SLICE::RW_xcode(INT)
 *
 * Description: Transcodes a slice from a read-stream into a new slice in a
 *   write-stream.  Contains context information as follows: current set of MPEG
 *   headers, read stream, write stream.
 *   Offset is an alias for the quantiser_scale_code_increment parameter.
 *
 ******************************************************************************/
LRESULT SLICE::RW_xcode(INT offset) {
    MACROBLOCK mbl;
    memcpy(&mbl.c, &c, sizeof(c));
    mbl.c.slc = this;

    slice_start_code = c.rstm->GetBits(32);
    c.wstm->PutBits(slice_start_code, 32);
    slice_vertical_position = (USHORT)slice_start_code & 0xff;
    if ((c.hdrs->sequence_extension.vertical_size_extension<<12) ||
        c.hdrs->sequence_header.vertical_size_value > 2800) {
        slice_vertical_position_extension = (BYTE)c.rstm->GetBits(3);
        c.wstm->PutBits(slice_vertical_position_extension, 3);
    }

    if (c.hdrs->sequence_scalable_extension.extension_start_code)
        if (c.hdrs->sequence_scalable_extension.scalable_mode == SEQUENCE_SCALABLE_EXTENSION::DATA_PARTITIONING) {
            priority_breakpoint = (BYTE)c.rstm->GetBits(7);
            c.wstm->PutBits(priority_breakpoint, 7);
        }

    quantiser_scale_code = (BYTE)c.rstm->GetBits(5);
    c.wstm->PutBits(min(quantiser_scale_code+offset, 31), 5);

    if (c.rstm->GetBits(1) == 1) {
        c.wstm->PutBits(1, 1);
        intra_slice_flag = (BYTE)c.rstm->GetBits(1);
        c.wstm->PutBits(intra_slice_flag, 1);
        intra_slice = (BYTE)c.rstm->GetBits(1);
        c.wstm->PutBits(intra_slice, 1);
        reserved_bits = (BYTE)c.rstm->GetBits(7);
        c.wstm->PutBits(reserved_bits, 7);
        _ASSERTE(c.rstm->GetBits(1) == 0);
    }
    c.wstm->PutBits(0, 1);

    /* Keep processing macroblocks until the bits run out */
    while (c.rstm->UnreadBits() >= 8)
        mbl.RW_xcode(offset);
    return 0;
}

/*******************************************************************************
 *
 * Function: MACROBLOCK::RW_xcode(INT)
 *
 * Description: Transcodes a macroblock from a read-stream into a new slice in a
 *   write-stream.  Contains context information as follows: current set of MPEG
 *   headers, read stream, write stream, current slice.
 *   Offset is an alias for the quantiser_scale_code_increment parameter.
 *
 ******************************************************************************/
LRESULT MACROBLOCK::RW_xcode(INT offset) {
    ULONG i;
    BLOCK bl;
    memcpy(&bl.c, &c, sizeof(c));
    bl.c.mbl = this;
    macroblock_address_increment = 0;
    while (c.rstm->PeekBits(11) == 0x8) {
        c.wstm->PutBits(0x8, 11);
        macroblock_address_increment += 33;
    }
    i = c.rstm->GetHuffman(huffman[1]);
    c.wstm->PutHuffman(huffman[1], i);
    macroblock_address_increment += (USHORT)i;

    /* process the macroblock_modes() section, especially macroblock_type */
    this->RW_MACROBLOCK_MODES();
    if (macroblock_quant) {
        quantiser_scale_code = (BYTE)c.rstm->GetBits(5); /* //? */
        c.slc->quantiser_scale_code = (BYTE)quantiser_scale_code;
        c.wstm->PutBits(min(quantiser_scale_code+offset, 31), 5);
    }
    else quantiser_scale_code = c.slc->quantiser_scale_code;

    if (macroblock_intra && c.hdrs->picture_coding_extension.concealment_motion_vectors) {
        _ASSERTE(c.rstm->GetBits(1) == 1);
        c.wstm->PutBits(1, 1);
        _ASSERTE(0);
    }

    /* Do a straight copy of all motion vectors from read-stream to write-stream.
       The problem is that you need to completely decode them (they're Huffman coded) */
    if (macroblock_motion_forward ||
        (macroblock_intra && c.hdrs->picture_coding_extension.concealment_motion_vectors))
        RW_MOTION_VECTORS(0);
    if (macroblock_motion_backward)
        RW_MOTION_VECTORS(1);

    /* Decode the coded_block_pattern() section */
    if (macroblock_pattern)
```

```
        RW_CODED_BLOCK_PATTERN();

    for (i = 0; i < 12; i++) pattern_code[i] = macroblock_intra;
    if (macroblock_pattern && !macroblock_intra) { // ?
        for (i = 0; i < 6; i++) if (cbp & (1<<(5-i))) pattern_code[i] = 1;
        if (c.hdrs->sequence_extension.chroma_format == SEQUENCE_EXTENSION::_4_2_2)
            for (i = 6; i < 8; i++) if (coded_block_pattern_1 & (1<<(7-i))) pattern_code[i] = 1;
        if (c.hdrs->sequence_extension.chroma_format == SEQUENCE_EXTENSION::_4_4_4)
            for (i = 6; i < 12; i++) if (coded_block_pattern_2 & (1<<(11-i))) pattern_code[i] = 1;
    }
    switch(c.hdrs->sequence_extension.chroma_format) {
    case SEQUENCE_EXTENSION::_4_2_0: block_count = 6; break;
    case SEQUENCE_EXTENSION::_4_2_2: block_count = 8; break;
    case SEQUENCE_EXTENSION::_4_4_4: block_count = 12; break;
    }

    bl.quant_table = 14;
    if (c.hdrs->picture_coding_extension.intra_vlc_format && !macroblock_intra)
        bl.quant_table=15;
    for (bl.i = 0; bl.i < block_count; bl.i++) {
        bl.cc = (bl.i>=4 ? 1 : 0) * (1 + (bl.i&1)); /* cc is defined in the MPEG spec */
        bl.Read(); /* Read the block from read-stream */
        bl.InvQuant(); /* inverse quantize the block */
        bl.Quant(offset); /* quantize the block with new divisors */
        bl.Write(); /* Write the block to the write-stream */
    }
    return 0;
}

/*******************************************************************************
 *
 * Function: MACROBLOCK::MACROBLOCK_MODES()
 *
 * Description: Passes over the macroblock_modes() section of the macroblock,
 *   copying each bit identically from the read-stream to the write-stream.
 *   The macroblock_type flags are particularly important here.
 *
 *******************************************************************************/
LRESULT MACROBLOCK::RW_MACROBLOCK_MODES() {
    ULONG i;

    switch(c.hdrs->picture_header.picture_coding_type) {
    case PICTURE_HEADER::I_PICTURE: i=c.rstm->GetHuffman(huffman[2]); c.wstm->PutHuffman(huffman[2], i); break;
    case PICTURE_HEADER::P_PICTURE: i=c.rstm->GetHuffman(huffman[3]); c.wstm->PutHuffman(huffman[3], i); break;
    case PICTURE_HEADER::B_PICTURE: i=c.rstm->GetHuffman(huffman[4]); c.wstm->PutHuffman(huffman[4], i); break;
    }
    macroblock_quant = (BYTE)((i&0x20)>>5);
    macroblock_motion_forward = (BYTE)((i&0x10)>>4);
    macroblock_motion_backward = (BYTE)((i&0x08)>>3);
    macroblock_pattern = (BYTE)((i&0x04)>>2);
    macroblock_intra = (BYTE)((i&0x02)>>1);
    spatial_temporal_weight_code_flag = (BYTE)(i&0x01);
    if (spatial_temporal_weight_code_flag == 1 &&
        c.hdrs->picture_spatial_scalable_extension.spatial_temporal_weight_code_table_index != 0) {
        spatial_temporal_weight_code = (BYTE)(c.rstm->GetBits(2));
        c.wstm->PutBits(spatial_temporal_weight_code, 2);
        _ASSERTE(0);
    }

    motion_vector_count = 1; dmv = 0;
    if (c.hdrs->picture_coding_extension.picture_structure == PICTURE_CODING_EXTENSION::FRAME_PICTURE)
        mv_format = FRAME; else mv_format = FIELD;
    if (macroblock_motion_forward || macroblock_motion_backward)
        if (c.hdrs->picture_coding_extension.picture_structure == PICTURE_CODING_EXTENSION::FRAME_PICTURE) {
            if (c.hdrs->picture_coding_extension.frame_pred_frame_dct == 0) {
                frame_motion_type = (BYTE)(c.rstm->GetBits(2)); c.wstm->PutBits(frame_motion_type, 2);
                motion_vector_count = frame_motion_type==1 ? 2 : 1;
                mv_format = frame_motion_type==2 ? FRAME : FIELD;
                dmv = frame_motion_type==3 ? 1 : 0;
            }
        } else {
            field_motion_type = (BYTE)(c.rstm->GetBits(2)); c.wstm->PutBits(frame_motion_type, 2);
            motion_vector_count = field_motion_type==2 ? 2 : 1;
            mv_format = FIELD;
            dmv = field_motion_type==3 ? 1 : 0;
        }

    if (c.hdrs->picture_coding_extension.picture_structure == PICTURE_CODING_EXTENSION::FRAME_PICTURE &&
        c.hdrs->picture_coding_extension.frame_pred_frame_dct == 0 &&
        (macroblock_intra || macroblock_pattern)) {
        dct_type = (BYTE)(c.rstm->GetBits(1));
        c.wstm->PutBits(dct_type, 1);
    }
    return 0;
}

/*******************************************************************************
 *
 * Function: MACROBLOCK::RW_MOTION_VECTORS()
 *
 * Description: Parses through the variable-length coded motion_vectors() section
 *   of a macroblock, copying each bit identically from the read-stream to the
 *   write-stream.
 *
 *******************************************************************************/
LRESULT MACROBLOCK::RW_MOTION_VECTORS(INT s) {
    if (motion_vector_count == 1) {
        if (mv_format == FIELD && dmv != 1) {
            motion_vertical_field_select[0][s] = (BYTE)(c.rstm->GetBits(1));
            c.wstm->PutBits(motion_vertical_field_select[0][s], 1);
        }
        RW_MOTION_VECTOR(0, s);
    } else {
        motion_vertical_field_select[0][s] = (BYTE)(c.rstm->GetBits(1));
        c.wstm->PutBits(motion_vertical_field_select[0][s], 1);
        RW_MOTION_VECTOR(0, s);
        motion_vertical_field_select[1][s] = (BYTE)(c.rstm->GetBits(1));
        c.wstm->PutBits(motion_vertical_field_select[1][s], 1);
        RW_MOTION_VECTOR(1, s);
    }
    return 0;
```

```
}

/********************************************************************************
 *
 * Function: MACROBLOCK::RW_MOTION_VECTOR()
 *
 * Description: Passes over the individual motion_vector() codes within a
 *   macroblock, copying each bit identically from the read-stream to the
 *   write_stream.
 *
 ********************************************************************************/
LRESULT MACROBLOCK::RW_MOTION_VECTOR(INT r, INT s) {
    INT r_size;
    motion_code[r][s][0] = c.rstm->GetHuffman(huffman[10]);
    c.wstm->PutHuffman(huffman[10], motion_code[r][s][0]);
    if (c.hdrs->picture_coding_extension.f_code[s][0] != 1 && motion_code[r][s][0] != 0) {
        r_size = c.hdrs->picture_coding_extension.f_code[s][0] - 1;
        motion_residual[r][s][0] = c.rstm->GetBits(r_size);
        c.wstm->PutBits(motion_residual[r][s][0], r_size);
    }

    if (dmv == 1) {
        dmvector[0] = c.rstm->GetHuffman(huffman[11]);
        c.wstm->PutHuffman(huffman[11], dmvector[0]);
    }

    motion_code[r][s][1] = c.rstm->GetHuffman(huffman[10]);
    c.wstm->PutHuffman(huffman[10], motion_code[r][s][1]);
    if (c.hdrs->picture_coding_extension.f_code[s][1] != 1 && motion_code[r][s][1] != 0) {
        r_size = c.hdrs->picture_coding_extension.f_code[s][1] - 1;
        motion_residual[r][s][1] = c.rstm->GetBits(r_size);
        c.wstm->PutBits(motion_residual[r][s][1], r_size);
    }
    if (dmv == 1) {
        dmvector[1] = c.rstm->GetHuffman(huffman[11]);
        c.wstm->PutHuffman(huffman[11], dmvector[1]);
    }
    return 0;
}

/********************************************************************************
 *
 * Function: MACROBLOCK::RW_CODED_BLOCK_PATTERN()
 *
 * Description: Parses the coded_block_pattern() section of a macroblock, copying
 *   each bit identically from the read-stream to the write-stream.  This is
 *   important for determining how many blocks the macroblock contains, all of
 *   which must be transcoded.
 *
 ********************************************************************************/
LRESULT MACROBLOCK::RW_CODED_BLOCK_PATTERN() {
    if (macroblock_pattern) { /* coded_block_pattern present flag */
        cbp = (BYTE)(c.rstm->GetHuffman(huffman[9]));
        c.wstm->PutHuffman(huffman[9], cbp);
        if (c.hdrs->sequence_extension.chroma_format == 0x2) { /* 4:2:2 */
            coded_block_pattern_1 = (BYTE)(c.rstm->GetBits(2));
            c.wstm->PutBits(coded_block_pattern_1, 2);
        }
        if (c.hdrs->sequence_extension.chroma_format == 0x3) { /* 4:4:4 */
            coded_block_pattern_2 = (BYTE)(c.rstm->GetBits(6));
            c.wstm->PutBits(coded_block_pattern_2, 6);
        }
    }
    return 0;
}

/********************************************************************************
 *
 * Function: BLOCK::Read()
 *
 * Description: Reads in block coefficients from the read-stream, using context
 *   information for the current MPEG headers, slice, and macroblock.
 *
 ********************************************************************************/
LRESULT BLOCK::Read() {
    int m, n=0, run, level;

    if (c.mbl->pattern_code[i]) {
        if (c.mbl->macroblock_intra) {
            dc_dct_size = c.rstm->GetHuffman(huffman[cc?13:12]);
            if (dc_dct_size == 0) { /* predictor is correct */
                QFS[0] = 0;
            } else { /* intra dc coefficient */
                dc_dct_differential = c.rstm->GetBits(dc_dct_size);
                if (dc_dct_differential >= (ULONG)(1<<(dc_dct_size-1)))
                    QFS[0] = dc_dct_differential;
                else
                    QFS[0] = dc_dct_differential + 1 - (1<<dc_dct_size);
            }
            n = 1;
        }
        while (1) {
            m = (int)c.rstm->GetHuffman(huffman[quant_table]);

            if (n == 0) {
                if (m == -1) {run = 0; level = +1;}
                if (m == +1) {run = 0; level = -1;}
            } else { /* n > 0 */
                if (m == -1) {while(n < 64) QFS[n++] = 0; break;}
            }
            if (m == 0) { /* escape code */
                run = (int)c.rstm->GetBits(6);
                level = (int)c.rstm->GetBits(12);
                if (level&0x800) level |= 0xFFFFF000;
            }
            if ((n==0 && m>1) || (n>0 && m>0)) {
                run = (m>>8)&0xFF; level = m&0xFF;
                if (c.rstm->GetBits(1) == 1) level = -level;
            }

            for (m = 0; m < run; m++) QFS[n++] = 0; QFS[n++] = level;
        }
    }
    _ASSERTE(n == 64 || n == 0);
    return 0;
}

/*******************************************************************************
 *
 * Function: BLOCK::InvQuant()
 *
 * Description: Inverse quantises the current block coefficients into raw
 *   frequency-domain coefficients.
 *
 ******************************************************************************/
LRESULT BLOCK::InvQuant() {
    int u, v, w;
    for (v = 0; v < 8; v++)
        for (u = 0; u < 8; u++)
            QF[v][u] = QFS[scan[c.hdrs->picture_coding_extension.alternate_scan][v][u]];
    intra_dc_mult = 1<<(3-c.hdrs->picture_coding_extension.intra_dc_precision);
    quantiser_scale = ::quantiser_scale[c.hdrs->picture_coding_extension.q_scale_type][c.mbl->quantiser_scale_code];
    w = (~c.mbl->macroblock_intra)&1;
    if (cc && c.hdrs->sequence_extension.chroma_format != SEQUENCE_EXTENSION::_4_2_0) w |= 0x2;
    for (v = 0; v < 8; v++)
        for (u = 0; u < 8; u++)
            if (u==0 && v==0 && c.mbl->macroblock_intra)
                F__[v][u] = intra_dc_mult * QF[v][u];
            else if (c.mbl->macroblock_intra)
                F__[v][u] = (2*QF[v][u] * W[w][v][u] * (int)quantiser_scale)/32;
                else F__[v][u] = ((2*QF[v][u] + Signum(QF[v][u])) * W[w][v][u] * quantiser_scale)/32;
    return 0;
}

/*******************************************************************************
 *
 * Function: BLOCK::Quant()
 *
 * Description: Forward-quantizes a block, using the following quantiser_scale_code:
 *       quantiser_scale_code = quantiser_scale_code + offset,
 *   where the offset parameter is just an alias for quantiser_scale_code_increment.
 *
 ******************************************************************************/
LRESULT BLOCK::Quant(INT offset) {
    int u, v, w;
    intra_dc_mult = 1<<(3-c.hdrs->picture_coding_extension.intra_dc_precision);
    quantiser_scale = ::quantiser_scale[c.hdrs->picture_coding_extension.q_scale_type][min(c.mbl->quantiser_scale_code+offset,
                      31)];
    w = (~c.mbl->macroblock_intra)&1;
    if (cc && c.hdrs->sequence_extension.chroma_format != SEQUENCE_EXTENSION::_4_2_0) w |= 0x2;
    for (v = 0; v < 8; v++)
        for (u = 0; u < 8; u++) {
            if (u==0 && v==0 && c.mbl->macroblock_intra)
                QF[v][u] = F__[v][u] / intra_dc_mult;
            else if (c.mbl->macroblock_intra)
                QF[v][u] = (16*F__[v][u]) / (W[w][v][u]*quantiser_scale);
            else {
                QF[v][u] = (32*F__[v][u]) / (W[w][v][u]*quantiser_scale);
                QF[v][u] = (QF[v][u] - Signum(QF[v][u])) / 2; /* ? */
            }
        }
    for (v = 0; v < 8; v++)
        for (u = 0; u < 8; u++)
            QFS[scan[c.hdrs->picture_coding_extension.alternate_scan][v][u]] = QF[v][u];
    return 0;
}

/*******************************************************************************
 *
 * Function: BLOCK::Read()
 *
 * Description: Writes block coefficients to the write-stream, using context
 *   information for the current MPEG headers, slice, and macroblock.
 *
 ******************************************************************************/
LRESULT BLOCK::Write() {
    int m, n=0, run, level;

    if (c.mbl->pattern_code[i]) {
        if (c.mbl->macroblock_intra) {
            c.wstm->PutHuffman(huffman[cc?13:12], dc_dct_size);
            if (dc_dct_size) c.wstm->PutBits(dc_dct_differential, dc_dct_size);
            n = 1;
        }
        while (1) { /* ac coefficients */
            run = 0;
            while (QFS[n]==0) {run++; if((++n)==64) break;}
            if (n >= 64) {c.wstm->PutHuffman(huffman[quant_table], (ULONG)(-1)); break;}
            level = QFS[n];
            if (n == 0 && run == 0 && level == +1) m = -1;
            else if (n == 0 && run == 0 && level == -1) m = +1;
            else m = (run<<8) | (level >= 0 ? level : -level);
            if (c.wstm->PutHuffman(huffman[quant_table], m) == -1) {
                c.wstm->PutHuffman(huffman[quant_table], 0);
                c.wstm->PutBits(run, 6);
                c.wstm->PutBits(level&0xfff, 12);
            } else if ((n==0 && m>1) || (n>0 && m>0)) {
                c.wstm->PutBits((level >= 0 ? 0 : 1), 1);
            }
            n++;
            if (n >= 64) {c.wstm->PutHuffman(huffman[quant_table], (ULONG)(-1)); break;}
        }
    }
    return 0;
}
```

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <math.h>
#include "resource.h"

/* Required for floating-point string formatting */
#include <tchar.h>
#include <stdio.h>

/* functions */
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
DOUBLE BMP_PSNR(HANDLE, HANDLE, HWND hwnd = HWND_DESKTOP);

/* global variables */
TCHAR szAppName[] = TEXT("WinPSNR");

/*******************************************************************************
 *
 * Function: WinMain(HINSTANCE, HINSTANCE, LPSTR, INT)
 *
 * Description: Entry point for all Windows programs.  Registers window class
 *   and displays dialog box window.
 *
 *******************************************************************************/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE /*hPrevInstance*/, LPSTR /*lpCmdLine*/, int nShowCmd) {
    /* Register window class for dialog box */
    WNDCLASS wndclass = {0};
    wndclass.style         = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc   = WndProc;
    wndclass.cbClsExtra    = 0;
    wndclass.cbWndExtra    = DLGWINDOWEXTRA;
    wndclass.hInstance     = hInstance;
    wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
    wndclass.lpszMenuName  = NULL;
    wndclass.lpszClassName = szAppName;
    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 1;
    }

    /* Create and show dialog box */
    HWND hwnd = CreateDialogParam(hInstance, szAppName, HWND_DESKTOP, NULL, 0);
    SendMessage(hwnd, WM_INITDIALOG, 0, 0);
    ShowWindow(hwnd, nShowCmd);

    /* Message-processing loop; exit when WM_QUIT comes */
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return 0;
}

/*******************************************************************************
 *
 * Function: WndProc(HWND, UINT, WPARAM, LPARAM)
 *
 * Description: Window procedure for dialog box.  Processes windows messages
 *   sent to the dialog box.
 *
 *******************************************************************************/
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static TCHAR szFilter[] = TEXT("Bitmap Images (*.BMP)\0*.bmp\0")
                              TEXT("All Files (*.*)\0*.*\0\0"),
                 szOutput[100] = TEXT("PSNR (luminance) : _?_ dB");
    static OPENFILENAME ofn1 = {sizeof(OPENFILENAME), 0}, ofn2 = {sizeof(OPENFILENAME), 0};
    HANDLE hFile1 = INVALID_HANDLE_VALUE, hFile2 = INVALID_HANDLE_VALUE;
    DOUBLE PSNR;

    switch(message) {
    case WM_INITDIALOG:
        /* Initialize OPENFILENAME structure for file selection common dialog box */
        ofn1.lStructSize       = sizeof(OPENFILENAME);
        ofn1.hwndOwner         = hwnd;
        ofn1.lpstrFilter       = szFilter;
        ofn1.lpstrCustomFilter = NULL;
        ofn1.lpstrFileTitle    = NULL;
        ofn1.lpstrInitialDir   = NULL;
        ofn1.Flags             = 0;
        ofn1.lpstrTitle        = NULL;
        ofn1.nFileOffset       = 0;
        ofn1.nFileExtension    = 0;
        ofn1.lpstrDefExt       = TEXT("*.BMP");
        ofn1.lCustData         = 0;
        ofn1.lpfnHook          = NULL;
        ofn1.lpTemplateName    = NULL;
        CopyMemory(&ofn2, &ofn1, sizeof(OPENFILENAME));
        ofn1.lpstrFile = (LPTSTR)malloc(MAX_PATH*sizeof(TCHAR)); ZeroMemory(ofn1.lpstrFile, MAX_PATH*sizeof(TCHAR));
        ofn2.lpstrFile = (LPTSTR)malloc(MAX_PATH*sizeof(TCHAR)); ZeroMemory(ofn2.lpstrFile, MAX_PATH*sizeof(TCHAR));
        ofn1.nMaxFile = ofn2.nMaxFile = MAX_PATH;

        /* Initialize static output control */
        Static_SetText(GetDlgItem(hwnd, IDC_OUTPUT), szOutput);
        return 0;
    case WM_COMMAND:
        switch(wParam) {
        case MAKELONG(IDC_BROWSE1, BN_CLICKED): /* Browse for file button clicked */
            if (GetOpenFileName(&ofn1)) Edit_SetText(GetDlgItem(hwnd, IDC_EDITFILE1), ofn1.lpstrFile);
            break;
        case MAKELONG(IDC_BROWSE2, BN_CLICKED): /* Browse for file button clicked */
            if (GetOpenFileName(&ofn2)) Edit_SetText(GetDlgItem(hwnd, IDC_EDITFILE2), ofn2.lpstrFile);
            break;
        case MAKELONG(IDC_COMPARE, BN_CLICKED): /* "Compare" button clicked */
            hFile1 = CreateFile(ofn1.lpstrFile, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN,
                      NULL);
            if (hFile1 == INVALID_HANDLE_VALUE) {
                MessageBox(hwnd, TEXT("Invalid Filename"), szAppName, MB_ICONERROR);
                SetFocus(GetDlgItem(hwnd, IDC_EDITFILE1));
                break;
            }
            hFile2 = CreateFile(ofn2.lpstrFile, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN,
                      NULL);
            if (hFile2 == INVALID_HANDLE_VALUE) {
                MessageBox(hwnd, TEXT("Invalid Filename"), szAppName, MB_ICONERROR);
                SetFocus(GetDlgItem(hwnd, IDC_EDITFILE2));
                break;
            }
            PSNR = BMP_PSNR(hFile1, hFile2, hwnd); /* calculate PSNR for two BMP files */
            if (PSNR != 0) {
                _stprintf(szOutput, TEXT("PSNR (luminance) : %.4f dB"), PSNR);
                Static_SetText(GetDlgItem(hwnd, IDC_OUTPUT), szOutput);
            }
            CloseHandle(hFile1); CloseHandle(hFile2);
            break;
        case MAKELONG(IDOK, BN_CLICKED):
        case MAKELONG(IDCANCEL, BN_CLICKED):
            SendMessage(hwnd, WM_CLOSE, 0, 0);
            break;
        }
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}

/******************************************************************************
 *
 * Function: BMP_PSNR(HANDLE, HANDLE, HWND)
 *
 * Description: Calculates the peak signal-to-noise ratio between the two
 *   bitmaps, and returns it as a double value.  Returns 0 on failure.
 *
 ******************************************************************************/
DOUBLE BMP_PSNR(HANDLE hFile1, HANDLE hFile2, HWND hwnd) {
    LRESULT ret = 0;
    BITMAPFILEHEADER *pbmfh1, *pbmfh2;
    BITMAPV5HEADER *pbmih1, *pbmih2;
    DWORD dwFile1Size, dwFile2Size, dwBytesRead;
    PBYTE pFile1, pFile2;

    if ((dwFile1Size = GetFileSize(hFile1, NULL)) == 0xffffffff) return 0;
    if ((dwFile2Size = GetFileSize(hFile2, NULL)) == 0xffffffff) return 0;
    pFile1 = (PBYTE)malloc(dwFile1Size);
    pFile2 = (PBYTE)malloc(dwFile2Size);

    SetCursor(LoadCursor(NULL, IDC_WAIT)); ShowCursor(TRUE);
    if (!ReadFile(hFile1, pFile1, dwFile1Size, &dwBytesRead, NULL) || (dwBytesRead != dwFile1Size)) ret = -1;
    if (!ReadFile(hFile2, pFile2, dwFile2Size, &dwBytesRead, NULL) || (dwBytesRead != dwFile1Size)) ret = -2;
    if (ret) {
        MessageBox(hwnd, TEXT("Error reading bitmap file(s)"), szAppName, MB_ICONERROR);
        goto EndCompareFiles;
    }

    /* Read bitmap file attributes and determine if they're identical */
    pbmfh1 = (LPBITMAPFILEHEADER)pFile1;
    pbmfh2 = (LPBITMAPFILEHEADER)pFile2;
    pbmih1 = (LPBITMAPV5HEADER)(pFile1 + sizeof(BITMAPFILEHEADER));
    pbmih2 = (LPBITMAPV5HEADER)(pFile2 + sizeof(BITMAPFILEHEADER));
    if (pbmfh1->bfType != (('M'<<8)|'B') || pbmfh1->bfType != (('M'<<8)|'B')) {
        MessageBox(hwnd, TEXT("Invalid Image File Type"), szAppName, MB_ICONERROR);
        goto EndCompareFiles;
    }
    if (pbmih1->bV5Width != pbmih2->bV5Width || pbmih1->bV5Height != pbmih2->bV5Height) {
        MessageBox(hwnd, TEXT("Image dimensions do not match"), szAppName, MB_ICONERROR);
        goto EndCompareFiles;
    }
    if (pbmih1->bV5BitCount != pbmih2->bV5BitCount || pbmih1->bV5BitCount != 24) {
        MessageBox(hwnd, TEXT("Invalid image color depth"), szAppName, MB_ICONERROR);
        goto EndCompareFiles;
    }
    if (pbmih1->bV5Size >= sizeof(BITMAPINFOHEADER))
        if (pbmih1->bV5Compression != BI_RGB || pbmih2->bV5Compression != BI_RGB) {
            MessageBox(hwnd, TEXT("Images must be uncompressed"), szAppName, MB_ICONERROR);
            goto EndCompareFiles;
        }

    DOUBLE PSNR, MSE_n, MSE_d;
    INT x, y, RowLength;
    RGBTRIPLE *pRGB1, *pRGB2;

    /* Calculate PSNR by looping through each bitmap's pixel */
    PSNR = MSE_n = MSE_d = 0;
    pRGB1 = (RGBTRIPLE*)(pFile1+pbmfh1->bfOffBits);
    pRGB2 = (RGBTRIPLE*)(pFile2+pbmfh1->bfOffBits);
    RowLength = ((pbmih1->bV5Width * pbmih1->bV5BitCount + 31) & ~31) >> 3;
    for (y = 0; y < pbmih1->bV5Height; y++) {
        for (x = 0; x <= pbmih1->bV5Width-1; x++) {
            /* Use luminance values */
            PSNR = (0.1818*pRGB1->rgbtRed + 0.612*pRGB1->rgbtGreen + 0.06168*pRGB1->rgbtBlue)
                 - (0.1818*pRGB2->rgbtRed + 0.612*pRGB2->rgbtGreen + 0.06168*pRGB2->rgbtBlue);
            MSE_n += (PSNR*PSNR);
            MSE_d += 1;
            pRGB1++;
            pRGB2++;
        }
        pRGB1 = (RGBTRIPLE*)((PBYTE)(pRGB1) + (RowLength - 3*pbmih1->bV5Width));
        pRGB2 = (RGBTRIPLE*)((PBYTE)(pRGB2) + (RowLength - 3*pbmih2->bV5Width));
    }
    PSNR = 10*log10((255*255)/(MSE_n/MSE_d));

EndCompareFiles:
    ShowCursor(FALSE);
    SetCursor(LoadCursor(NULL, IDC_ARROW));
    free(pFile1); free(pFile2);
    if (!ret) return PSNR; else return 0;
}
```

FILE: WinPSNR.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

WINPSNR DIALOG DISCARDABLE  0, 0, 220, 117
STYLE DS_SETFOREGROUND | DS_3DLOOK | DS_NOFAILCREATE | WS_MINIMIZEBOX |
    WS_CAPTION | WS_SYSMENU
CAPTION "Windows Bitmap PSNR Calculator"
CLASS "WinPSNR"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_EDITFILE1,7,7,150,14,ES_AUTOHSCROLL | ES_READONLY
    PUSHBUTTON      "Browse...",IDC_BROWSE1,163,7,50,14
    EDITTEXT        IDC_EDITFILE2,7,27,150,14,ES_AUTOHSCROLL | ES_READONLY
    PUSHBUTTON      "Browse...",IDC_BROWSE2,163,27,50,14
    LTEXT           "Static",IDC_OUTPUT,7,48,206,39
    DEFPUSHBUTTON   "Close",IDOK,54,96,50,14
    PUSHBUTTON      "Compare",IDC_COMPARE,116,96,50,14
END

/////////////////////////////////////////////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x40004L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904b0"
        BEGIN
            VALUE "CompanyName", "University of Manitoba\0"
            VALUE "FileDescription", "Calculates PSNR for two 24-bit BMP files\0"
            VALUE "FileVersion", "1.00\0"
            VALUE "InternalName", "programmer: Delbert Dueck\0"
            VALUE "LegalCopyright", "Copyright © 2001\0"
            VALUE "OriginalFilename", "WinPSNR.exe\0"
            VALUE "ProductName", "Windows Bitmap PSNR Calculator\0"
            VALUE "ProductVersion", "1.00\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END
```

REFERENCES

[1] ISO/IEC 13818-1, 1994. *Generic Coding of Moving Pictures and Associated Audio Information: Systems*. Draft Recommendation. 13 Nov 1994.

[2] ISO/IEC 13818-2, 1994. *Generic Coding of Moving Pictures and Associated Audio Information: Video*. Draft Recommendation. 9 Nov 1994.

[3] Chiariglione, Leonardo, 2001. *Open Source in MPEG.* Linux Journal, March 2001, No. 83, pp. 126-132.

[4] Gadegast, Frank, 1995. *The MPEG FAQ*. Version 4.1, 2 Jun 1996. Available at http://www.landfield.com/faqs/mpeg-faq/.

[5] Tudor, P.N. and Werner, O.H., 1997. *Real-time Transcoding of MPEG-2 Video Bit Streams*. International Broadcasting Convention, September, pp. 286-301.