AVT Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2011

J-M. Valin
Octasic Semiconductor
T. Terriberry
Xiph.Org Foundation
G. Maxwell
Juniper Networks
C. Montgomery
Xiph.Org Foundation
July 8, 2010

Constrained-Energy Lapped Transform (CELT) Codec
draft-valin-celt-codec-02

Abstract

CELT [celt-website] is an open-source voice codec suitable for use in very low delay Voice over IP (VoIP) type applications. This document describes the encoding and decoding process.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2011.

Copyright Notice

Table of Contents

1. Introduction

This document describes the CELT codec, which is designed for transmitting full-bandwidth audio with very low delay. It is suitable for encoding both speech and music at rates starting at 32 kbit/s. It is primarily designed for transmission over the Internet and protocols such as RTP [rfc3550], but also includes a certain amount of robustness to bit errors, where this could be done at no significant cost.

The novel aspect of CELT compared to most other codecs is its very low delay, below 10 ms. There are two main advantages to having a very low delay audio link. The lower delay itself is important for some interactions, such as playing music remotely. Another advantage is its behavior in the presence of acoustic echo. When the round-trip audio delay is sufficiently low, acoustic echo is no longer perceived as a distinct repetition, but rather as extra reverberation. Applications of CELT include:

- Collaborative network music performance
- High-quality teleconferencing
- Wireless audio equipment
- Low-delay links for broadcast applications

The source code for the reference implementation of the CELT codec is provided in Appendix A. This source code is the normative specification of the codec. The corresponding text description in this document is provided for informative purposes.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [rfc2119].

2. Overview of the CELT Codec

CELT stands for _Constrained Energy Lapped Transform_. This is the fundamental principle of the codec: the quantization process is designed in such a way as to preserve the energy in a certain number of bands. The theoretical aspects of the codec are described in greater detail [celt-tasl] and [celt-eusipco]. Although these papers describe slightly older versions of the codec (version 0.3.2 and 0.5.1, respectively), the principles remain the same.

CELT is a transform codec, based on the Modified Discrete Cosine Transform (MDCT). The MDCT is derived from the DCT-IV by adding an overlap with time-domain aliasing cancellation [mdct]. The main characteristics of CELT are as follows:

- Ultra-low algorithmic delay (scalable, typically 4 to 9 ms)
- Sampling rates from 32 kHz to 48 kHz and above (full audio bandwidth)
- Applicability to both speech and music
- Support for mono and stereo
- Adaptive bit-rate from 32 kbit/s to 128 kbit/s per channel and above
- Scalable complexity
- Robustness to packet loss (scalable trade-off between quality and loss-robustness)
- Open source implementation (floating-point and fixed-point)
- No known intellectual property issues

2.1. Bit-stream definition

This document contains a detailed description of both the encoder and the decoder, along with a reference implementation. In most circumstances, and unless otherwise stated, the calculations do *not* need to produce results that are bit-identical with the reference implementation, so alternate algorithms can sometimes be used. However, there are a few (clearly identified) cases, such as the bit allocation, where bit-exactness with the reference implementation is required. An implementation is considered to be compatible if, for any valid bit-stream, the decoder's output is perceptually indistinguishable from the output produced by the reference decoder.

The CELT codec does not use a standard _bit-packer_, but rather uses a range coder to pack both integers and entropy-coded symbols. In mono mode, the bit-stream generated by the encoder contains the following parameters (in order):

- Feature flags I, P, S, F (2-4 bits)
- if P=1
  - Pitch period
- if S=1
  - Transient scalefactor
  - if scalefactor=(1 or 2) AND more than 2 short MDCTs
    - ID of block before transient
  - if scalefactor=3
    - Transient time
- Coarse energy encoding (for each band)
- Fine energy encoding (for each band)
- For each band
  - if P=1 and band is at the beginning of a pitch band
    - Pitch gain bit
  - PVQ indices
- More fine energy (using all remaining bits)

Note that due to the use of a range coder, all of the parameters have to be encoded and decoded in order.

The CELT bit-stream is "octet-based" in the sense that the encoder always produces an integer number of octets when encoding a frame. Also, the bit-rate used by the CELT encoder can *only* be determined by the number of octets produced. In many cases (e.g. UDP/RTP), the transport layer already encodes the data length, so no extra information is necessary to signal the bit-rate. In cases where this is not true, or when there are multiple compressed frames per packet, the size of each compressed frame MUST be signalled in some way.

3. CELT Modes

The operation of both the encoder and decoder depends on the mode data. A mode definition can be created by celt_create_mode() (modes.c (Appendix A.6)) based on three parameters:

- frame size (number of samples)
- sampling rate (samples per second)
- number of channels (1 or 2)

The frame size can be any even number of samples from 64 to 1024, inclusively. The sampling rate must be between 32000 Hz and 96000 Hz. The mode data that is created defines how the encoder and the decoder operate. More specifically, the following information is contained in the mode object:

- Frame size
- Sampling rate
- Windowing overlap
- Number of channels
- Definition of the bands
- Definition of the _pitch bands_
- Decay coefficients of the Laplace distributions for coarse energy
- Bit allocation matrix

The windowing overlap is the amount of overlap between the frames. CELT uses a low-overlap window that is typically half of the frame size. For a frame size of 256 samples, the overlap is 128 samples, so the total algorithmic delay is 256+128=384. CELT divides the audio into frequency bands, for which the energy is preserved. These bands are chosen to follow the ear's critical bands, with the exception that each band has to contain at least 3 frequency bins.

The energy bands are based on the Bark scale. The Bark band edges (in Hz) are defined as [0, 100, 200, 300, 400, 510, 630, 770, 920, 1080, 1270, 1480, 1720, 2000, 2320, 2700, 3150, 3700, 4400, 5300, 6400, 7700, 9500, 12000, 15500, 20000]. The actual bands used by the codec depend on the sampling rate and the frame size being used. The mapping from Hz to MDCT bins is done by multiplying by sampling_rate/

(2*frame_size) and rounding to the nearest value. An exception is made for the lower frequencies to ensure that all bands contain at least 3 MDCT bins. The definition of the Bark bands is computed in compute_ebands() (modes.c (Appendix A.6)).

CELT includes a pitch predictor for which the gains are defined over a set of _pitch bands_. The pitch bands are defined (in Hz) as [0, 345, 689, 1034, 1378, 2067, 3273, 5340, 6374]. The Hz values are mapped to MDCT bins in the same was as the energy bands. The pitch band boundaries are aligned to energy band boundaries. The definition of the pitch bands is computed in compute_pbands() (modes.c (Appendix A.6)).

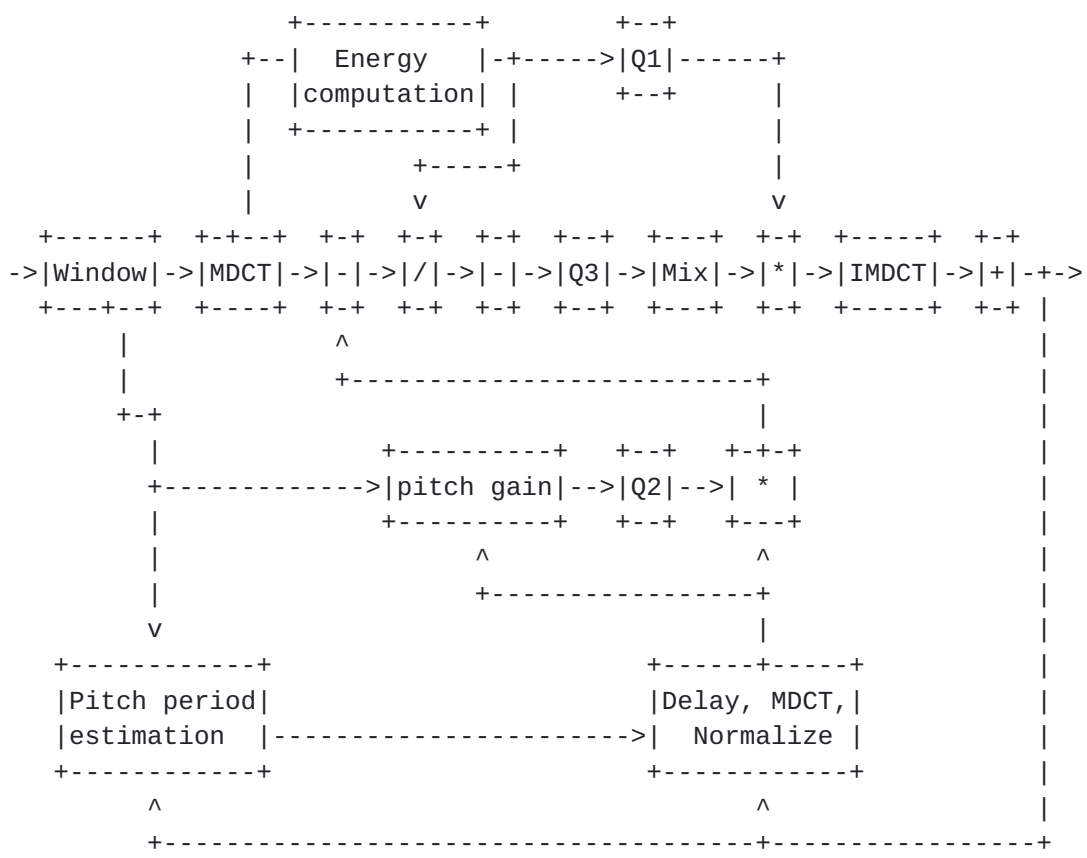
The mode contains a bit allocation table that is derived from a prototype allocation table, specified in the band_allocation matrix (modes.c (Appendix A.6)). Each row of the table is a single prototype allocation, in bits per Bark band, and assuming 256-sample frames. These rows must be projected onto the actual band layout in use at the current frame size and sample rate, using exact integer calculations. The reference implementation pre-computes these projections in compute_allocation_table() (modes.c (Appendix A.6)) and any other implementation MUST produces bit-identical allocation results.

Every entry in the allocation table is multiplied by the current number of channels and the current frame size. Each prototype allocation is projected independently using the following process: the upper band frequencies (in Hz) from the current Bark band and current CELT band are compared. (When the process begins, these will each be the first band, but will increment independently.) If the current Bark band's upper edge frequency is less than the current CELT band's upper edge frequency, the entire value of the Bark band plus any carried remainder is assigned to the current CELT band, and the process continues with the next Bark band in sequence and zero remainder. If the current Bark band's upper edge frequency is equal to or greater than that of the current CELT band, the CELT band will receive only part of this Bark band's allocation. This portion allocated to the CELT band is then calculated by multiplying the Bark band's allocation by the difference in Hz between the Bark band's upper frequency and the current CELT band's lower frequency, adding the width of the current Bark band divided by two, and then dividing this total by the width of the current Bark band in Hz. The partial value plus any carried remainder is added to the current CELT band, and the difference between the partial value and the Bark target is taken as the new carried remainder. The process begins then again starting at the next CELT band and next Bark band. Once all bands in a prototype allocation have been considered, any remainder is added to the last CELT band. All of the resulting values are rescaled by

adding 128 and dividing by 256.

4. CELT Encoder

The top-level function for encoding a CELT frame in the reference implementation is celt_encode() (celt.c (Appendix A.4)). The basic block diagram of the CELT encoder is illustrated in Figure 1. The encoder contains most of the building blocks of the decoder and can, with very little extra computation, compute the signal that would be decoded by the decoder. CELT has three main quantizers denoted Q1, Q2 and Q3. These apply to band energies (Section 4.5), pitch gains (Section 4.7) and normalized MDCT bins (Section 4.8), respectively.



Block diagram of the CELT encoder

Figure 1

The input audio first goes through a pre-emphasis filter (just before the windowing in Figure 1), which attenuates the _spectral tilt_. The filter is has the transfer function A(z)=1-alpha_p*z^-1, with alpha_p=0.8. The inverse of the pre-emphasis is applied at the decoder.

4.1. Range Coder

CELT uses an entropy coder based upon [range-coding], which is itself a rediscovery of the FIFO arithmetic code introduced by [coding-thesis]. It is very similar to arithmetic encoding, except that encoding is done with digits in any base instead of with bits, so it is faster when using larger bases (i.e.: an octet). All of the calculations in the range coder must use bit-exact integer arithmetic.

The range coder also acts as the bit-packer for CELT. It is used in three different ways, to encode:

- entropy-coded symbols with a fixed probability model using ec_encode(), (rangeenc.c (Appendix A.29))
- integers from 0 to 2^M-1 using ec_enc_uint() or ec_enc_bits(), (entenc.c (Appendix A.25))
- integers from 0 to N-1 (where N is not a power of two) using ec_enc_uint(). (entenc.c (Appendix A.25))

The range encoder maintains an internal state vector composed of the four-tuple (low,rng,rem,ext), representing the low end of the current range, the size of the current range, a single buffered output octet, and a count of additional carry-propagating output octets. Both rng and low are 32-bit unsigned integer values, rem is an octet value or the special value -1, and ext is an integer with at least 16 bits. This state vector is initialized at the start of each each frame to the value (0,2^31,-1,0).

Each symbol is drawn from a finite alphabet and coded in a separate context which describes the size of the alphabet and the relative frequency of each symbol in that alphabet. CELT only uses static contexts; they are not adapted to the statistics of the data that is coded.

4.1.1. Encoding Symbols

The main encoding function is ec_encode() (rangeenc.c (Appendix A.29)), which takes as an argument a three-tuple (fl,fh,ft) describing the range of the symbol to be encoded in the current context, with 0 <= fl < fh <= ft <= 65535. The values of this tuple are derived from the probability model for the symbol. Let f(i) be the frequency of the ith symbol in the current context. Then the three-tuple corresponding to the kth symbol is given by fl=sum(f(i),i<k), fh=fl+f(i), and ft=sum(f(i)).

ec_encode() updates the state of the encoder as follows. If fl is greater than zero, then low = low + rng - (rng/ft)*(ft-fl) and rng = (rng/ft)*(fh-fl). Otherwise, low is unchanged and rng = rng - (rng/ft)*(fh-fl). The divisions here are exact integer division. After this update, the range is normalized.

To normalize the range, the following process is repeated until rng > 2^23. First, the top 9 bits of low, (low>>23), are placed into a carry buffer. Then, low is set to (low << 8 & 0x7FFFFFFF) and rng is set to (rng<<8). This process is carried out by ec_enc_normalize() (rangeenc.c (Appendix A.29)).

The 9 bits produced in each iteration of the normalization loop consist of 8 data bits and a carry flag. The final value of the output bits is not determined until carry propagation is accounted for. Therefore the reference implementation buffers a single (non-propagating) output octet and keeps a count of additional propagating (0xFF) output octets. An implementation MAY choose to use any mathematically equivalent scheme to perform carry propagation.

The function ec_enc_carry_out() (rangeenc.c (Appendix A.29)) performs this buffering. It takes a 9-bit input value, c, from the normalization 8-bit output and a carry bit. If c is 0xFF, then ext is incremented and no octets are output. Otherwise, if rem is not the special value -1, then the octet (rem+(c>>8)) is output. Then ext octets are output with the value 0 if the carry bit is set, or 0xFF if it is not, and rem is set to the lower 8 bits of c. After this, ext is set to zero.

In the reference implementation, a special version of ec_encode() called ec_encode_bin() (rangeenc.c (Appendix A.29)) is defined to take a two-tuple (fl,ftb), where 0 <= fl < 2^ftb and ftb < 16. It is mathematically equivalent to calling ec_encode() with the three-tuple (fl,fl+1,1<<ftb), but avoids using division.

4.1.2. Encoding Uniformly Distributed Integers

Functions ec_enc_uint() or ec_enc_bits() are based on ec_encode() and encode one of N equiprobable symbols, each with a frequency of 1, where N may be as large as 2^32-1. Because ec_encode() is limited to a total frequency of 2^16-1, this is done by encoding a series of symbols in smaller contexts.

ec_enc_bits() (entenc.c (Appendix A.25)) is defined, like ec_encode_bin(), to take a two-tuple (fl,ftb), with 0 <= fl < 2^ftb and ftb < 32. While ftb is greater than 8, it encodes bits (ftb-8) to (ftb-1) of fl, e.g., (fl>>ftb-8&0xFF) using ec_encode_bin() and subtracts 8 from ftb. Then, it encodes the remaining bits of fl,

e.g., (fl&(1<<ftb)-1), again using ec_encode_bin().

ec_enc_uint() (entenc.c (Appendix A.25)) takes a two-tuple (fl,ft), where ft is not necessarily a power of two. Let ftb be the location of the highest 1 bit in the two's-complement representation of (ft-1), or -1 if no bits are set. If ftb>8, then the top 8 bits of fl are encoded using ec_encode() with the three-tuple (fl>>ftb-8,(fl>>ftb-8)+1,(ft-1>>ftb-8)+1), and the remaining bits are encoded with ec_enc_bits using the two-tuple (fl&(1<<ftb-8)-1,ftb-8). Otherwise, fl is encoded with ec_encode() directly using the three-tuple (fl,fl+1,ft).

4.1.3. Finalizing the Stream

After all symbols are encoded, the stream must be finalized by outputting a value inside the current range. Let end be the integer in the interval [low,low+rng) with the largest number of trailing zero bits. Then while end is not zero, the top 9 bits of end, e.g., (end>>23), are sent to the carry buffer, and end is replaced by (end<<8&0x7FFFFFF). Finally, if the value in carry buffer, rem, is neither zero nor the special value -1, or the carry count, ext, is greater than zero, then 9 zero bits are sent to the carry buffer. After the carry buffer is finished outputting octets, the rest of the output buffer is padded with zero octets. Finally, rem is set to the special value -1. This process is implemented by ec_enc_done() (rangeenc.c (Appendix A.29)).

4.1.4. Current Bit Usage

The bit allocation routines in CELT need to be able to determine a conservative upper bound on the number of bits that have been used to encode the current frame thus far. This drives allocation decisions and ensures that the range code will not overflow the output buffer. This is computed in the reference implementation to fractional bit precision by the function ec_enc_tell() (rangeenc.c (Appendix A.29)). Like all operations in the range encoder, it must be implemented in a bit-exact manner.

4.2. Encoder Feature Selection

The CELT codec has several optional features that can be switched on or off in each frame, some of which are mutually exclusive. The four main flags are intra-frame energy (I), pitch (P), short blocks (S), and folding (F). Those are described in more detail below. There are eight valid combinations of these four features, and they are encoded into the stream first using a variable length code (Table 1). It is left to the implementor to choose when to enable each of the flags, with the only restriction that the combination of the four

flags MUST correspond to a valid entry in Table 1.

Encoding of the feature flags

I	P	S	F	Encoding
0	0	0	1	00
0	1	0	1	01
1	0	0	1	110
1	0	1	1	111
0	0	0	0	1000
0	0	1	1	1001
0	1	0	0	1010
1	0	0	0	1011

Table 1

4.2.1. Intra-frame energy (I)

CELT uses prediction to encode the energy in each frequency band. In order to make frames independent, however, it is possible to disable the part of the prediction that depends on previous frames. This is called _intra-frame energy_ and requires around 12 more bits per frame. It is enabled with the _I_ bit (Table. flags-encoding (Table 1)). The use of intra energy is OPTIONAL and the decision method is left to the implementor. The reference code describes one way of deciding which frames would benefit most from having their energy encoded without prediction. The intra_decision() (quant_bands.c (Appendix A.34)) function looks for frames where the log-spectral distance between consecutive frames is more than 9 dB. When such a difference is found between two frames, the next frame (not the one for which the difference is detected) is marked encoded with intra energy. The one-frame delay is to ensure that when a frame containing a transient is lost, then the next frame will be decoded without accumulating error from the lost frame.

4.2.2. Pitch prediction (P)

CELT can use a pitch predictor (also known as long-term predictor) to improve the voice quality at lower bit-rates. When the _P_ bit is set, the pitch period is encoded after the flag bits. The value encoded is an integer in the range [0, 1024-N-overlap-1].

4.2.3. Short blocks (S)

To improve audio quality during transients, CELT can use a _short block_ multiple-MDCT transform. Unlike other transform codecs, the multiple MDCTs are jointly quantized as if the coefficients were obtained from a single MDCT. For that reason, it is better to consider the short block case as using a different transform of the same length rather than as multiple independent MDCTs. In the reference implementation, the decision to use short blocks is made by transient_analysis() (celt.c (Appendix A.4)) based on the pre-emphasized signal's peak values, but other methods can be used. When the _S_ bit is set, a 2-bit transient scalefactor is encoded directly after the flag bits. If the scalefactor is 0, then the multiple-MDCT output is unmodified. If the scalefactor is 1 or 2, then the output of the MDCTs that follow the transient is scaled down by 2^scalefactor. If the scalefactor is equal to 3, then a time-domain pre-emphasis window is applied *before* computing the MDCTs and no further scaling is applied to the MDCTs output. The window value is 1 from the beginning of the frame to 16 samples before the transient time. It is a Hanning window from there to the transient time, and then the value is 1/8 up to the end of the frame. The Hanning window part is defined as:

```
static const float transientWindow[16] = { 0.0085135, 0.0337639,
0.0748914, 0.1304955, 0.1986827, 0.2771308, 0.3631685, 0.4538658,
0.5461342, 0.6368315, 0.7228692, 0.8013173, 0.8695045, 0.9251086,
0.9662361, 0.9914865};
```

When the scalefactor is 3, the transient time is the exact time of the transient determined by the encoder and encoded as an integer number of samples with the range [0, N+overlap-1] directly after the scalefactor.

In the case where the scalefactor is 1 or 2 and the mode is defined to use more than 2 MDCTs, the last MDCT to which the scaling is *not* applied is encoded using an integer in the range [0, B-2], where B is the number of short MDCTs used for the mode.

4.2.4. Spectral folding (F)

The last encoding feature in CELT is spectral folding. It is designed to prevent _birdie_ artifacts caused by the sparse spectra often generated by low-bitrate transform codecs. When folding is enabled, a copy of the low-frequency spectrum is added to the higher-frequency bands (above ~6400 Hz). The folding operation is described in more detail in Section 4.8.

4.3. Forward MDCT

The MDCT implementation has no special characteristics. The input is a windowed signal (after pre-emphasis) of 2*N samples and the output is N frequency-domain samples. A _low-overlap_ window is used to reduce the algorithmic delay. It is derived from a basic (full overlap) window that is the same as the one used in the Vorbis codec: W(n)=[sin(pi/2*sin(pi/2*(n+.5)/L))]^2. The low-overlap window is created by zero-padding the basic window and inserting ones in the middle, such that the resulting window still satisfies power complementarity. The MDCT is computed in mdct_forward() (mdct.c (Appendix A.20)), which includes the windowing operation and a scaling of 2/N.

4.4. Bands and Normalization

The MDCT output is divided into bands that are designed to match the ear's critical bands, with the exception that each band has to be at least 3 bins wide. For each band, the encoder computes the energy that will later be encoded. Each band is then normalized by the square root of the *non-quantized* energy, such that each band now forms a unit vector X. The energy and the normalization are computed by compute_band_energies() and normalise_bands() (bands.c (Appendix A.8)), respectively.

4.5. Energy Envelope Quantization

It is important to quantize the energy with sufficient resolution because any energy quantization error cannot be compensated for at a later stage. Regardless of the resolution used for encoding the shape of a band, it is perceptually important to preserve the energy in each band. CELT uses a coarse-fine strategy for encoding the energy in the base-2 log domain, as implemented in quant_bands.c (Appendix A.34)

4.5.1. Coarse energy quantization

The coarse quantization of the energy uses a fixed resolution of 6 dB and is the only place where entropy coding is used. To minimize the

bitrate, prediction is applied both in time (using the previous frame) and in frequency (using the previous bands). The 2-D z-transform of the prediction filter is: A(z_l, z_b)=(1-a*z_l^-1)*(1-z_b^-1)/(1-b*z_b^-1) where b is the band index and l is the frame index. The prediction coefficients are a=0.8 and b=0.7 when not using intra energy and a=b=0 when using intra energy. The time-domain prediction is based on the final fine quantization of the previous frame, while the frequency domain (within the current frame) prediction is based on coarse quantization only (because the fine quantization has not been computed yet). We approximate the ideal probability distribution of the prediction error using a Laplace distribution. The coarse energy quantization is performed by quant_coarse_energy() and quant_coarse_energy() (quant_bands.c (Appendix A.34)).

The Laplace distribution for each band is defined by a 16-bit (Q15) decay parameter. Thus, the value 0 has a frequency count of p[0]=2*(16384*(16384-decay)/(16384+decay)). The values +/- i each have a frequency count p[i] = (p[i-1]*decay)>>14. The value of p[i] is always rounded down (to avoid exceeding 32768 as the sum of all frequency counts), so it is possible for the sum to be less than 32768. In that case additional values with a frequency count of 1 are encoded. The signed values corresponding to symbols 0, 1, 2, 3, 4, ... are [0, +1, -1, +2, -2, ...]. The encoding of the Laplace-distributed values is implemented in ec_laplace_encode() (laplace.c (Appendix A.32)).

4.5.2. Fine energy quantization

After the coarse energy quantization and encoding, the bit allocation is computed (Section 4.6) and the number of bits to use for refining the energy quantization is determined for each band. Let B_i be the number of fine energy bits for band i; the refinement is an integer f in the range [0,2^B_i-1]. The mapping between f and the correction applied to the coarse energy is equal to (f+1/2)/2^B_i - 1/2. Fine energy quantization is implemented in quant_fine_energy() (quant_bands.c (Appendix A.34)).

If any bits are unused at the end of the encoding process, these bits are used to increase the resolution of the fine energy encoding in some bands. Priority is given to the bands for which the allocation (Section 4.6) was rounded down. At the same level of priority, lower bands are encoded first. Refinement bits are added until there are no unused bits. This is implemented in quant_energy_finalise() (quant_bands.c (Appendix A.34)).

4.6. Bit Allocation

Bit allocation is performed based only on information available to both the encoder and decoder. The same calculations are performed in a bit-exact manner in both the encoder and decoder to ensure that the result is always exactly the same. Any mismatch would cause an error in the decoded output. The allocation is computed by compute_allocation() (rate.c (Appendix A.16)), which is used in both the encoder and the decoder.

For a given band, the bit allocation is nearly constant across frames that use the same number of bits for Q1, yielding a pre-defined signal-to-mask ratio (SMR) for each band. Because the bands each have a width of one Bark, this is equivalent to modeling the masking occurring within each critical band, while ignoring inter-band masking and tone-vs-noise characteristics. While this is not an optimal bit allocation, it provides good results without requiring the transmission of any allocation information.

For every encoded or decoded frame, a target allocation must be computed using the projected allocation. In the reference implementation this is performed by compute_allocation() (rate.c (Appendix A.16)). The target computation begins by calculating the available space as the number of whole bits which can be fit in the frame after Q1 is stored according to the range coder (ec_[enc/dec]_tell()) and then multiplying by 8. Then the two projected prototype allocations whose sums multiplied by 8 are nearest to that value are determined. These two projected prototype allocations are then interpolated by finding the highest integer interpolation coefficient in the range 0-8 such that the sum of the higher prototype times the coefficient, plus the sum of the lower prototype multiplied by the difference of 16 and the coefficient, is less than or equal to the available sixteenth-bits. The reference implementation performs this step using a binary search in interp_bits2pulses() (rate.c (Appendix A.16)). The target allocation is the interpolation coefficient times the higher prototype, plus the lower prototype multiplied by the difference of 16 and the coefficient, for each of the CELT bands.

Because the computed target will sometimes be somewhat smaller than the available space, the excess space is divided by the number of bands, and this amount is added equally to each band. Any remaining space is added to the target one sixteenth-bit at a time, starting from the first band. The new target now matches the available space, in sixteenth-bits, exactly.

The allocation target is separated into a portion used for fine energy and a portion used for the Spherical Vector Quantizer (PVQ).

The fine energy quantizer operates in whole-bit steps. For each band the number of bits per channel used for fine energy is calculated by 50 minus the log2_frac(), with 1/16 bit precision, of the number of MDCT bins in the band. That result is multiplied by the number of bins in the band and again by twice the number of channels, and then the value is set to zero if it is less than zero. Added to that result is 16 times the number of MDCT bins times the number of channels, and it is finally divided by 32 times the number of MDCT bins times the number of channels. If the result times the number of channels is greater than than the target divided by 16, the result is set to the target divided by the number of channels divided by 16. Then if the value is greater than 7 it is reset to 7 because a larger amount of fine energy resolution was determined not to be make an improvement in perceived quality. The resulting number of fine energy bits per channel is then multiplied by the number of channels and then by 16, and subtracted from the target allocation. This final target allocation is what is used for the PVQ.

4.7. Pitch Prediction

This section needs to be updated.

4.8. Spherical Vector Quantization

CELT uses a Pyramid Vector Quantization (PVQ) [PVQ] codebook for quantizing the details of the spectrum in each band that have not been predicted by the pitch predictor. The PVQ codebook consists of all sums of K signed pulses in a vector of N samples, where two pulses at the same position are required to have the same sign. Thus the codebook includes all integer codevectors y of N dimensions that satisfy sum(abs(y(j))) = K.

In bands where neither pitch nor folding is used, the PVQ is used to encode the unit vector that results from the normalization in Section 4.4 directly. Given a PVQ codevector y, the unit vector X is obtained as X = y/||y||, where ||.|| denotes the L2 norm.

4.8.1. Bits to Pulses

Although the allocation is performed in 1/16 bit units, the quantization requires an integer number of pulses K. To do this, the encoder searches for the value of K that produces the number of bits that is the nearest to the allocated value (rounding down if exactly half-way between two values), subject to not exceeding the total number of bits available. The computation is performed in 1/16 of bits using log2_frac() and ec_enc_tell(). The number of codebooks entries can be computed as explained in Section 4.8.3. The difference between the number of bits allocated and the number of

bits used is accumulated to a _balance_ (initialised to zero) that helps adjusting the allocation for the next bands. One third of the balance is subtracted from the bit allocation of the next band to help achieving the target allocation. The only exceptions are the band before the last and the last band, for which half the balance and the whole balance are subtracted, respectively.

4.8.2. PVQ Search

The search for the best codevector y is performed by alg_quant() (vq.c (Appendix A.12)). There are several possible approaches to the search with a tradeoff between quality and complexity. The method used in the reference implementation computes an initial codeword y1 by projecting the residual signal R = X - p' onto the codebook pyramid of K-1 pulses:

```
y0 = round_towards_zero( (K-1) * R / sum(abs(R)))
```

Depending on N, K and the input data, the initial codeword y0 may contain from 0 to K-1 non-zero values. All the remaining pulses, with the exception of the last one, are found iteratively with a greedy search that minimizes the normalized correlation between y and R:

```
J = -R^T*y / ||y||
```

The search described above is considered to be a good trade-off between quality and computational cost. However, there are other possible ways to search the PVQ codebook and the implementors MAY use any other search methods.

4.8.3. Index Encoding

The best PVQ codeword is encoded as a uniformly-distributed integer value by encode_pulses() (cwrs.c (Appendix A.10)). The codeword is converted to a unique index in the same way as specified in [PVQ]. The indexing is based on the calculation of V(N,K) (denoted N(L,K) in [PVQ]), which is the number of possible combinations of K pulses in N samples. The number of combinations can be computed recursively as V(N,K) = V(N+1,K) + V(N,K+1) + V(N+1,K+1), with V(N,0) = 1 and V(0,K) = 0, K != 0. There are many different ways to compute V(N,K), including pre-computed tables and direct use of the recursive formulation. The reference implementation applies the recursive formulation one line (or column) at a time to save on memory use, along with an alternate, univariate recurrence to initialise an arbitrary line, and direct polynomial solutions for small N. All of these methods are equivalent, and have different trade-offs in speed, memory usage, and code size. Implementations MAY use any methods

they like, as long as they are equivalent to the mathematical definition.

The indexing computations are performed using 32-bit unsigned integers. For large codebooks, 32-bit integers are not sufficient. Instead of using 64-bit integers (or more), the encoding is made slightly sub-optimal by splitting each band into two equal (or near-equal) vectors of size (N+1)/2 and N/2, respectively. The number of pulses in the first half, K1, is first encoded as an integer in the range [0,K]. Then, two codebooks are encoded with V((N+1)/2, K1) and V(N/2, K-K1). The split operation is performed recursively, in case one (or both) of the split vectors still requires more than 32 bits. For compatibility reasons, the handling of codebooks of more than 32 bits MUST be implemented with the splitting method, even if 64-bit arithmetic is available.

4.9. Stereo support

When encoding a stereo stream, some parameters are shared across the left and right channels, while others are transmitted separately for each channel, or jointly encoded. Only one copy of the flags for the features, transients and pitch (pitch period and gains) are transmitted. The coarse and fine energy parameters are transmitted separately for each channel. Both the coarse energy and fine energy (including the remaining fine bits at the end of the stream) have the left and right bands interleaved in the stream, with the left band encoded first.

The main difference between mono and stereo coding is the PVQ coding of the normalized vectors. In stereo mode, a normalized mid-side (M-S) encoding is used. Let L and R be the normalized vector of a certain band for the left and right channels, respectively. The mid and side vectors are computed as M=L+R and S=L-R and no longer have unit norm.

From M and S, an angular parameter theta=2/pi*atan2(||S||, ||M||) is computed. The theta parameter is converted to a Q14 fixed-point parameter itheta, which is quantized on a scale from 0 to 1 with an interval of 2^-qb, where qb = (b-2*(N-1)*(40-log2_frac(N,4)))/(32*(N-1)), b is the number of bits allocated to the band, and log2_frac() is defined in cwrs.c (Appendix A.10). From here on, the value of itheta MUST be treated in a bit-exact manner since both the encoder and decoder rely on it to infer the bit allocation.

Let m=M/||M|| and s=S/||S||; m and s are separately encoded with the PVQ encoder described in Section 4.8. The number of bits allocated to m and s depends on the value of itheta. The number of bits allocated to coding m is obtained by:

```
imid = bitexact_cos(itheta);

iside = bitexact_cos(16384-itheta);

delta = (N-1)*(log2_frac(iside,6)-log2_frac(imid,6))>>2;

qalloc = log2_frac((1<<qb)+1,4);

mbits = (b-qalloc/2-delta)/2;
```

where bitexact_cos() is a fixed-point cosine approximation that MUST be bit-exact with the reference implementation in mathops.h (Appendix A.36). The spectral folding operation is performed independently for the mid and side vectors.

4.10. Synthesis

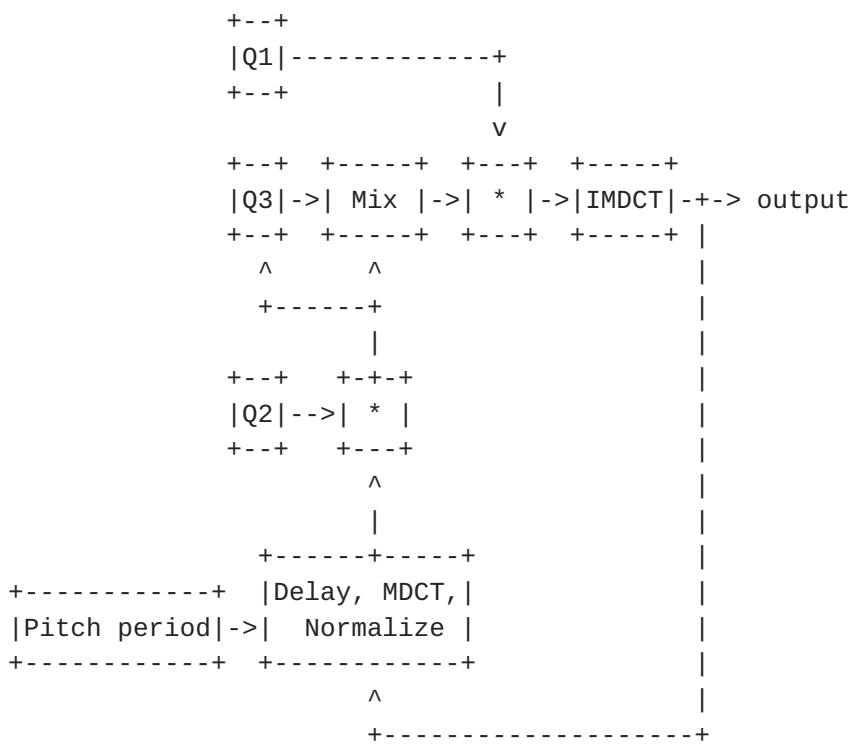
After all the quantization is completed, the quantized energy is used along with the quantized normalized band data to resynthesize the MDCT spectrum. The inverse MDCT (Section 5.6) and the weighted overlap-add are applied and the signal is stored in the _synthesis buffer_ so it can be used for pitch prediction. The encoder MAY omit this step of the processing if it knows that it will not be using the pitch predictor for the next few frames. If the de-emphasis filter (Section 5.6) is applied to this resynthesized signal, then the output will be the same (within numerical precision) as the decoder's output.

4.11. Variable Bitrate (VBR)

Each CELT frame can be encoded in a different number of octets, making it possible to vary the bitrate at will. This property can be used to implement source-controlled variable bitrate (VBR). Support for VBR is OPTIONAL for the encoder, but a decoder MUST be prepared to decode a stream that changes its bit-rate dynamically. The method used to vary the bit-rate in VBR mode is left to the implementor, as long as each frame can be decoded by the reference decoder.

5. CELT Decoder

Like most audio codecs, the CELT decoder is less complex than the encoder, as can be observed in the decoder block diagram in Figure 2. In fact, most of the operations performed by the decoder are also performed by the encoder.



Block diagram of the CELT decoder

Figure 2

The decoder extracts information from the range-coded bit-stream in the same order as it was encoded by the encoder. In some circumstances, it is possible for a decoded value to be out of range due to a very small amount of redundancy in the encoding of large integers by the range coder. In that case, the decoder should assume there has been an error in the coding, decoding, or transmission and SHOULD take measures to conceal the error and/or report to the application that a problem has occurred.

5.1. Range Decoder

The range decoder extracts the symbols and integers encoded using the range encoder in Section 4.1. The range decoder maintains an

internal state vector composed of the two-tuple (dif,rng), representing the difference between the high end of the current range and the actual coded value, and the size of the current range, respectively. Both dif and rng are 32-bit unsigned integer values. rng is initialized to 2^7. dif is initialized to rng minus the top 7 bits of the first input octet. Then the range is immediately normalized, using the procedure described in the following section.

5.1.1. Decoding Symbols

Decoding symbols is a two-step process. The first step determines a value fs that lies within the range of some symbol in the current context. The second step updates the range decoder state with the three-tuple (fl,fh,ft) corresponding to that symbol, as defined in Section 4.1.1.

The first step is implemented by ec_decode() (rangedec.c (Appendix A.30)), and computes fs = ft-min((dif-1)/(rng/ft)+1,ft), where ft is the sum of the frequency counts in the current context, as described in Section 4.1.1. The divisions here are exact integer division.

In the reference implementation, a special version of ec_decode() called ec_decode_bin() (rangeenc.c (Appendix A.29)) is defined using the parameter ftb instead of ft. It is mathematically equivalent to calling ec_decode() with ft = (1<<ftb), but avoids one of the divisions.

The decoder then identifies the symbol in the current context corresponding to fs; i.e., the one whose three-tuple (fl,fh,ft) satisfies fl <= fs < fh. This tuple is used to update the decoder state according to dif = dif - (rng/ft)*(ft-fh), and if fl is greater than zero, rng = (rng/ft)*(fh-fl), or otherwise rng = rng - (rng/ft)*(ft-fh). After this update, the range is normalized.

To normalize the range, the following process is repeated until rng > 2^23. First, rng is set to (rng<8)&0xFFFFFFFF. Then the next 8 bits of input are read into sym, using the remaining bit from the previous input octet as the high bit of sym, and the top 7 bits of the next octet for the remaining bits of sym. If no more input octets remain, zero bits are used instead. Then, dif is set to (dif<<8)-sym&0xFFFFFFFF (i.e., using wrap-around if the subtraction overflows a 32-bit register). Finally, if dif is larger than 2^31, dif is then set to dif - 2^31. This process is carried out by ec_dec_normalize() (rangedec.c (Appendix A.30)).

5.1.2. Decoding Uniformly Distributed Integers

Functions ec_dec_uint() or ec_dec_bits() are based on ec_decode() and decode one of N equiprobable symbols, each with a frequency of 1, where N may be as large as 2^32-1. Because ec_decode() is limited to a total frequency of 2^16-1, this is done by decoding a series of symbols in smaller contexts.

ec_dec_bits() (entdec.c (Appendix A.27)) is defined, like ec_decode_bin(), to take a single parameter ftb, with ftb < 32. and ftb < 32, and produces an ftb-bit decoded integer value, t, initialized to zero. While ftb is greater than 8, it decodes the next 8 most significant bits of the integer, s = ec_decode_bin(8), updates the decoder state with the 3-tuple (s,s+1,256), adds those bits to the current value of t, t = t<<8 | s, and subtracts 8 from ftb. Then it decodes the remaining bits of the integer, s = ec_decode_bin(ftb), updates the decoder state with the 3 tuple (s,s+1,1<<ftb), and adds those bits to the final values of t, t = t<<ftb | s.

ec_dec_uint() (entdec.c (Appendix A.27)) takes a single parameter, ft, which is not necessarily a power of two, and returns an integer, t, with a value between 0 and ft-1, inclusive, which is initialized to zero. Let ftb be the location of the highest 1 bit in the two's-complement representation of (ft-1), or -1 if no bits are set. If ftb>8, then the top 8 bits of t are decoded using t = ec_decode((ft-1>>ftb-8)+1), the decoder state is updated with the three-tuple (s,s+1,(ft-1>>ftb-8)+1), and the remaining bits are decoded with t = t<<ftb-8|ec_dec_bits(ftb-8). If, at this point, t >= ft, then the current frame is corrupt, and decoding should stop. If the original value of ftb was not greater than 8, then t is decoded with t = ec_decode(ft), and the decoder state is updated with the three-tuple (t,t+1,ft).

5.1.3. Current Bit Usage

The bit allocation routines in CELT need to be able to determine a conservative upper bound on the number of bits that have been used to decode from the current frame thus far. This drives allocation decisions which must match those made in the encoder. This is computed in the reference implementation to fractional bit precision by the function ec_dec_tell() (rangedec.c (Appendix A.30)). Like all operations in the range decoder, it must be implemented in a bit-exact manner, and must produce exactly the same value returned by ec_enc_tell() after encoding the same symbols.

5.2. Energy Envelope Decoding

The energy of each band is extracted from the bit-stream in two steps according to the same coarse-fine strategy used in the encoder. First, the coarse energy is decoded in unquant_coarse_energy() (quant_bands.c (Appendix A.34)) based on the probability of the Laplace model used by the encoder.

After the coarse energy is decoded, the same allocation function as used in the encoder is called (Section 4.6). This determines the number of bits to decode for the fine energy quantization. The decoding of the fine energy bits is performed by unquant_fine_energy() (quant_bands.c (Appendix A.34)). Finally, like the encoder, the remaining bits in the stream (that would otherwise go unused) are decoded using unquant_energy_finalise() (quant_bands.c (Appendix A.34)).

5.3. Pitch prediction decoding

If the pitch bit is set, then the pitch period is extracted from the bit-stream. The pitch gain bits are extracted within the PVQ decoding as encoded by the encoder. When the folding bit is set, the folding prediction is computed in exactly the same way as the encoder, with the same gain, by the function intra_fold() (vq.c (Appendix A.12)).

5.4. Spherical VQ Decoder

In order to correctly decode the PVQ codewords, the decoder must perform exactly the same bits to pulses conversion as the encoder (see Section 4.8.1).

5.4.1. Index Decoding

The decoding of the codeword from the index is performed as specified in [PVQ], as implemented in function decode_pulses() (cwrs.c (Appendix A.10)).

5.4.2. Normalised Vector Decoding

The spherical codebook is decoded by alg_unquant() (vq.c (Appendix A.12)). The index of the PVQ entry is obtained from the range coder and converted to a pulse vector by decode_pulses() (cwrs.c (Appendix A.10)).

The decoded normalized vector for each band is equal to

$$X' = y/||y||,$$

This operation is implemented in mix_pitch_and_residual() (vq.c (Appendix A.12)), which is the same function as used in the encoder.

5.5. Denormalization

Just like each band was normalized in the encoder, the last step of the decoder before the inverse MDCT is to denormalize the bands. Each decoded normalized band is multiplied by the square root of the decoded energy. This is done by denormalise_bands() (bands.c (Appendix A.8)).

5.6. Inverse MDCT

The inverse MDCT implementation has no special characteristics. The input is N frequency-domain samples and the output is 2*N time-domain samples, while scaling by 1/2. The output is windowed using the same _low-overlap_ window as the encoder. The IMDCT and windowing are performed by mdct_backward (mdct.c (Appendix A.20)). If a time-domain pre-emphasis window was applied in the encoder, the (inverse) time-domain de-emphasis window is applied on the IMDCT result. After the overlap-add process, the signal is de-emphasized using the inverse of the pre-emphasis filter used in the encoder: 1/A(z)=1/(1-alpha_p*z^-1).

5.7. Packet Loss Concealment (PLC)

Packet loss concealment (PLC) is an optional decoder-side feature which SHOULD be included when transmitting over an unreliable channel. Because PLC is not part of the bit-stream, there are several possible ways to implement PLC with different complexity/quality trade-offs. The PLC in the reference implementation finds a periodicity in the decoded signal and repeats the windowed waveform using the pitch offset. The windowed waveform is overlapped in such a way as to preserve the time-domain aliasing cancellation with the previous frame and the next frame. This is implemented in celt_decode_lost() (mdct.c (Appendix A.4)).

6. Security Considerations

A potential denial-of-service threat exists for data encodings using compression techniques that have non-uniform receiver-end computational load. The attacker can inject pathological datagrams into the stream which are complex to decode and cause the receiver to become overloaded. However, this encoding does not exhibit any significant non-uniformity.

With the exception of the first four bits, the bit-stream produced by CELT for an unknown audio stream is not easily predictable, due to the use of entropy coding. This should make CELT less vulnerable to attacks based on plaintext guessing when encryption is used. Also, since almost all possible bit combinations can be interpreted as a valid bit-stream, it is likely more difficult to determine from the decrypted bit-stream whether a guessed decryption key is valid.

When operating CELT in variable-bitrate (VBR) mode, some of the properties described above no longer hold. More specifically, the size of the packet leaks a very small, but non-zero, amount of information about both the original signal and the bit-stream plaintext.

7. IANA Considerations

This document has no actions for IANA.

8. Acknowledgments

The authors would also like to thank the CELT users who contributed patches, bug reports, feature requests, suggestions or comments.

9. References

9.1. Normative References

[rfc2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119.

[rfc3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for real-time applications", RFC 3550.

9.2. Informative References

[celt-tasl] Valin, JM., Terriberry, T., Montgomery, C., and G. Maxwell, "A High-Quality Speech and Audio Codec With Less Than 10 ms delay", To appear in IEEE Transactions on Audio, Speech and Language Processing 2009.

[celt-eusipco] Valin, JM., Terriberry, T., and G. Maxwell, "A Full-Bandwidth Audio Codec with Low Complexity and Very Low Delay", Accepted for EUSIPCO 2009.

[celt-website] "The CELT ultra-low delay audio codec", CELT website http://www.celt-codec.org/.

[mdct] "Modified Discrete Cosine Transform", MDCT http://en.wikipedia.org/wiki/Modified_discrete_cosine_transform.

[range-coding] Nigel, G. and N. Martin, "Range encoding: An algorithm for removing redundancy from a digitised message", Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording , 1979.

[coding-thesis] Pasco, R., "Source coding algorithms for fast data compression", Ph.D. thesis Dept. of Electrical Engineering, Stanford University, May 1976.

[PVQ] Fischer, T., "A Pyramid Vector Quantizer", IEEE Trans. on Information Theory, Vol. 32 pp. 568-583, July 1986.

Appendix A. Reference Implementation

This appendix contains the complete source code for a floating-point reference implementation of the CELT codec written in C. This implementation is derived from version 0.8.1 of the implementation available on the [celt-website], which can be compiled for either floating-point or fixed-point architectures.

The implementation can be compiled with either a C89 or a C99 compiler. It is reasonably optimized for most platforms such that only architecture-specific optimizations are likely to be useful. The FFT used is a slightly modified version of the KISS-FFT package, but it is easy to substitute any other FFT library.

The testcelt executable can be used to test the encoding and decoding process:

```
testcelt <rate> <channels> <frame size> <octets per packet>
[<complexity> [packet loss rate]] <input> <output>
```

where "rate" is the sampling rate in Hz, "channels" is the number of channels (1 or 2), "frame size" is the number of samples in a frame (64 to 1024) and "octets per packet" is the number of octets desired for each compressed frame. The input and output files are assumed to be a 16-bit PCM file in the machine native endianness. The optional "complexity" argument can select the quality vs complexity tradeoff (0-10) and the "packet loss rate" argument simulates random packet loss (argument is in tenths or a percent).

A.1. Makefile

```
CC = gcc
CFLAGS = -c -O2 -g
LIBS = -lm

OBJS = bands.o celt.o cwrs.o entcode.o entdec.o entenc.o kiss_fft.o \
        kiss_fftr.o laplace.o mdct.o modes.o pitch.o \
        quant_bands.o rangedec.o rangeenc.o rate.o testcelt.o vq.o plc.o

.c.o:
        $(CC) $(CFLAGS) $<

testcelt: $(OBJS)
        $(CC) -o $@ $(OBJS) $(LIBS)

clean:
        rm -f testcelt *.o
```

A.2. testcelt.c

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "celt.h"
#include "arch.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define MAX_PACKET 1024
```

```
int
main(int argc, char *argv[])
{
  int             err;
  char           *inFile,
                 *outFile;
  FILE           *fin,
                 *fout;
  CELTMode       *mode = NULL;
  CELTEncoder    *enc;
  CELTDecoder    *dec;
  int             len;
  celt_int32      frame_size,
                  channels;
  int             bytes_per_packet;
  unsigned char   data[MAX_PACKET];
  int             rate;
  int             complexity;

  int             i;
  double          rmsd = 0;

  int             count = 0;
  celt_int32      skip;
  celt_int16     *in,
                 *out;
  if (argc != 9 && argc != 8 && argc != 7)
  {
    fprintf(stderr, "Usage: testcelt <rate> <channels> <frame size> "
            " <bytes per packet> [<complexity> [packet loss rate]] "
            "<input> <output>\n");
    return 1;
  }

  rate = atoi(argv[1]);
  channels = atoi(argv[2]);
  frame_size = atoi(argv[3]);
  mode = celt_mode_create(rate, frame_size, NULL);
  celt_mode_info(mode, CELT_GET_LOOKAHEAD, &skip);

  if (mode == NULL)
  {
    fprintf(stderr, "failed to create a mode\n");
    return 1;
  }

  bytes_per_packet = atoi(argv[4]);
  if (bytes_per_packet < 0 || bytes_per_packet > MAX_PACKET)
```

```
    {
      fprintf(stderr, "bytes per packet must be between 0 and %d\n",
              MAX_PACKET);
      return 1;
    }

    inFile = argv[argc - 2];
    fin = fopen(inFile, "rb");
    if (!fin)
    {
      fprintf(stderr, "Could not open input file %s\n",
              argv[argc - 2]);
      return 1;
    }
    outFile = argv[argc - 1];
    fout = fopen(outFile, "wb+");
    if (!fout)
    {
      fprintf(stderr, "Could not open output file %s\n",
              argv[argc - 1]);
      return 1;
    }

    enc = celt_encoder_create(mode, channels, &err);
    if (err != 0)
    {
      fprintf(stderr, "Failed to create the encoder: %s\n",
              celt_strerror(err));
      return 1;
    }
    dec = celt_decoder_create(mode, channels, &err);
    if (err != 0)
    {
      fprintf(stderr, "Failed to create the decoder: %s\n",
              celt_strerror(err));
      return 1;
    }

    if (argc > 7)
    {
      complexity = atoi(argv[5]);
      celt_encoder_ctl(enc, CELT_SET_COMPLEXITY(complexity));
    }

    in = (celt_int16 *) malloc(frame_size * channels *
                               sizeof(celt_int16));
    out =
        (celt_int16 *) malloc(frame_size * channels *
```

```
                              sizeof(celt_int16));
     while (!feof(fin))
     {
       err = fread(in, sizeof(short), frame_size * channels, fin);
       if (feof(fin))
         break;
       len =
           celt_encode_resynthesis(enc, in, in, frame_size, data,
                                   bytes_per_packet);
       if (len <= 0)
       {
         fprintf(stderr, "celt_encode() returned %d\n", len);
         return 1;
       }
       /* This is for simulating bit errors */
       /* This is to simulate packet loss */
       if (argc == 9 && rand() % 1000 < atoi(argv[argc - 3]))
         /* if (errors && (errors%2==0)) */
         celt_decode(dec, NULL, len, out, frame_size);
       else
         celt_decode(dec, data, len, out, frame_size);

       for (i = 0; i < frame_size * channels; i++)
       {
         rmsd += (in[i] - out[i]) * 1.0 * (in[i] - out[i]);
         /* out[i] -= in[i]; */
       }

       count++;
       fwrite(out + skip, sizeof(short), (frame_size - skip) * channels,
              fout);
       skip = 0;
     }
     ;

     celt_encoder_destroy(enc);
     celt_decoder_destroy(dec);
     fclose(fin);
     fclose(fout);
     celt_mode_destroy(mode);
     free(in);
     free(out);

     if (rmsd > 0)
     {
       rmsd = sqrt(rmsd / (1.0 * frame_size * channels * count));
       fprintf(stderr, "Error: encoder doesn't match decoder\n");
       fprintf(stderr, "RMS mismatch is %f\n", rmsd);
```

```
      return 1;
    } else
    {
      fprintf(stderr, "Encoder matches decoder!!\n");
    }

    return 0;
  }
```

A.3. celt.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Copyright (c) 2008 Gregory Maxwell Written by
   Jean-Marc Valin and Gregory Maxwell */
/**
  @file celt.h
  @brief Contains all the functions for encoding and decoding audio
 */

/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
```

```
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef CELT_H
#define CELT_H

#include "celt_types.h"

#ifdef __cplusplus
extern          "C" {
#endif

#if defined(__GNUC__) && defined(CELT_BUILD)
#define EXPORT __attribute__ ((visibility ("default")))
#elif defined(WIN32)
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

#define _celt_check_int(x) (((void)((x) == (celt_int32)0)), (celt_int32\
)(x))
#define _celt_check_mode_ptr_ptr(ptr) ((ptr) + ((ptr) - (CELTMode**)(pt\
r)))

  /* Error codes */
/** No error */
#define CELT_OK                0
/** An (or more) invalid argument (e.g. out of range) */
#define CELT_BAD_ARG          -1
/** The mode struct passed is invalid */
#define CELT_INVALID_MODE     -2
/** An internal error was detected */
#define CELT_INTERNAL_ERROR   -3
/** The data passed (e.g. compressed data to decoder) is corrupted */
#define CELT_CORRUPTED_DATA   -4
/** Invalid/unsupported request number */
#define CELT_UNIMPLEMENTED    -5
/** An encoder or decoder structure is invalid or already freed */
#define CELT_INVALID_STATE    -6
/** Memory allocation has failed */
#define CELT_ALLOC_FAIL       -7

  /* Requests */
#define CELT_GET_MODE_REQUEST    1
/** Get the CELTMode used by an encoder or decoder */
#define CELT_GET_MODE(x) CELT_GET_MODE_REQUEST, _celt_check_mode_ptr_pt\
```

```
r(x)
#define CELT_SET_COMPLEXITY_REQUEST    2
/** Controls the complexity from 0-10 (int) */
#define CELT_SET_COMPLEXITY(x) CELT_SET_COMPLEXITY_REQUEST, _celt_check\
_int(x)
#define CELT_SET_PREDICTION_REQUEST    4
/** Controls the use of interframe prediction.
    0=Independent frames
    1=Short term interframe prediction allowed
    2=Long term prediction allowed
 */
#define CELT_SET_PREDICTION(x) CELT_SET_PREDICTION_REQUEST, _celt_check\
_int(x)
#define CELT_SET_VBR_RATE_REQUEST    6
/** Set the target VBR rate in bits per second(int); 0=CBR (default) */
#define CELT_SET_VBR_RATE(x) CELT_SET_VBR_RATE_REQUEST, _celt_check_int\
(x)
/** Reset the encoder/decoder memories to zero*/
#define CELT_RESET_STATE_REQUEST        8
#define CELT_RESET_STATE       CELT_RESET_STATE_REQUEST

#define CELT_SET_START_BAND_REQUEST    10000
/** Controls the complexity from 0-10 (int) */
#define CELT_SET_START_BAND(x) CELT_SET_START_BAND_REQUEST, _celt_check\
_int(x)

/** GET the lookahead used in the current mode */
#define CELT_GET_LOOKAHEAD    1001
/** GET the sample rate used in the current mode */
#define CELT_GET_SAMPLE_RATE  1003

/** GET the bit-stream version for compatibility check */
#define CELT_GET_BITSTREAM_VERSION 2000

/** Contains the state of an encoder. One encoder state is needed
    for each stream. It is initialised once at the beginning of the
    stream. Do *not* re-initialise the state for every frame.
   @brief Encoder state
 */
  typedef struct CELTEncoder CELTEncoder;

/** State of the decoder. One decoder state is needed for each stream.
    It is initialised once at the beginning of the stream. Do *not*
    re-initialise the state for every frame */
  typedef struct CELTDecoder CELTDecoder;

/** The mode contains all the information necessary to create an
    encoder. Both the encoder and decoder need to be initialised
```

```
    with exactly the same mode, otherwise the quality will be very
    bad */
  typedef struct CELTMode CELTMode;

/** \defgroup codec Encoding and decoding */
  /* @{ */

  /* Mode calls */

/** Creates a new mode struct. This will be passed to an encoder or
    decoder. The mode MUST NOT BE DESTROYED until the encoders and
    decoders that use it are destroyed as well.
 @param Fs Sampling rate (32000 to 96000 Hz)
 @param frame_size Number of samples (per channel) to encode in each
                   packet (even values; 64 - 512)
 @param error Returned error code (if NULL, no error will be returned)
 @return A newly created mode
*/
  EXPORT CELTMode *celt_mode_create(celt_int32 Fs, int frame_size,
                                    int *error);

/** Destroys a mode struct. Only call this after all encoders and
    decoders using this mode are destroyed as well.
 @param mode Mode to be destroyed
*/
  EXPORT void     celt_mode_destroy(CELTMode * mode);

/** Query information from a mode */
  EXPORT int      celt_mode_info(const CELTMode * mode, int request,
                                 celt_int32 * value);

  /* Encoder stuff */

/** Creates a new encoder state. Each stream needs its own encoder
    state (can't be shared across simultaneous streams).
 @param mode Contains all the information about the characteristics of
 *  the stream (must be the same characteristics as used for the
 *  decoder)
 @param channels Number of channels
 @param error Returns an error code
 @return Newly created encoder state.
*/
  EXPORT CELTEncoder *celt_encoder_create(const CELTMode * mode,
                                          int channels, int *error);

/** Destroys a an encoder state.
 @param st Encoder state to be destroyed
 */
```

```
  EXPORT void     celt_encoder_destroy(CELTEncoder * st);

/** Encodes a frame of audio.
 @param st Encoder state
 @param pcm PCM audio in float format, with a normal range of +/-1.0.
 *          Samples with a range beyond +/-1.0 are supported but will
 *          be clipped by decoders using the integer API and should
 *          only be used if it is known that the far end supports
 *          extended dynmaic range. There must be exactly
 *          frame_size samples per channel.
 @param optional_resynthesis If not NULL, the encoder copies the audio \
signal that
 *          the decoder would decode. It is the same as calling the
 *          decoder on the compressed data, just faster.
 *          This may alias pcm.
 @param compressed The compressed data is written here. This may not al\
ias pcm or
 *                 optional_synthesis.
 @param nbCompressedBytes Maximum number of bytes to use for compressin\
g the frame
 *          (can change from one frame to another)
 @return Number of bytes written to "compressed". Will be the same as
 *       "nbCompressedBytes" unless the stream is VBR and will never be\
 larger.
 *       If negative, an error has occurred (see error codes). It is IM\
PORTANT that
 *       the length returned be somehow transmitted to the decoder. Oth\
erwise, no
 *       decoding is possible.
*/
  EXPORT int      celt_encode_resynthesis_float(CELTEncoder * st,
                                                const float *pcm,
                                                float
                                                *optional_resynthesis,
                                                int frame_size,
                                                unsigned char
                                                *compressed,
                                                int
                                                nbCompressedBytes);

/** Encodes a frame of audio.
 @param st Encoder state
 @param pcm PCM audio in float format, with a normal range of +/-1.0.
 *          Samples with a range beyond +/-1.0 are supported but will
 *          be clipped by decoders using the integer API and should
 *          only be used if it is known that the far end supports
 *          extended dynmaic range. There must be exactly
 *          frame_size samples per channel.
```

```
 @param compressed The compressed data is written here. This may not al\
ias pcm or
 *                 optional_synthesis.
 @param nbCompressedBytes Maximum number of bytes to use for compressin\
g the frame
 *          (can change from one frame to another)
 @return Number of bytes written to "compressed". Will be the same as
 *       "nbCompressedBytes" unless the stream is VBR and will never be\
 larger.
 *       If negative, an error has occurred (see error codes). It is IM\
PORTANT that
 *       the length returned be somehow transmitted to the decoder. Oth\
erwise, no
 *       decoding is possible.
*/
  EXPORT int      celt_encode_float(CELTEncoder * st,
                                    const float *pcm, int frame_size,
                                    unsigned char *compressed,
                                    int nbCompressedBytes);

/** Encodes a frame of audio.
 @param st Encoder state
 @param pcm PCM audio in signed 16-bit format (native endian). There mu\
st be
 *          exactly frame_size samples per channel.
 @param optional_resynthesis If not NULL, the encoder copies the audio \
signal that
 *                          the decoder would decode. It is the same as \
calling the
 *                          decoder on the compressed data, just faster.
 *                          This may alias pcm.
 @param compressed The compressed data is written here. This may not al\
ias pcm or
 *                          optional_synthesis.
 @param nbCompressedBytes Maximum number of bytes to use for compressin\
g the frame
 *                         (can change from one frame to another)
 @return Number of bytes written to "compressed". Will be the same as
 *       "nbCompressedBytes" unless the stream is VBR and will never be\
 larger.
 *       If negative, an error has occurred (see error codes). It is IM\
PORTANT that
 *       the length returned be somehow transmitted to the decoder. Oth\
erwise, no
 *       decoding is possible.
 */
  EXPORT int      celt_encode_resynthesis(CELTEncoder * st,
                                          const celt_int16 * pcm,
```

```
                                        celt_int16 *
                                        optional_resynthesis,
                                        int frame_size,
                                        unsigned char *compressed,
                                        int nbCompressedBytes);

/** Encodes a frame of audio.
 @param st Encoder state
 @param pcm PCM audio in signed 16-bit format (native endian). There mu\
st be
 *          exactly frame_size samples per channel.
 @param compressed The compressed data is written here. This may not al\
ias pcm or
 *                          optional_synthesis.
 @param nbCompressedBytes Maximum number of bytes to use for compressin\
g the frame
 *                         (can change from one frame to another)
 @return Number of bytes written to "compressed". Will be the same as
 *       "nbCompressedBytes" unless the stream is VBR and will never be\
 larger.
 *       If negative, an error has occurred (see error codes). It is IM\
PORTANT that
 *       the length returned be somehow transmitted to the decoder. Oth\
erwise, no
 *       decoding is possible.
 */
  EXPORT int      celt_encode(CELTEncoder * st,
                              const celt_int16 * pcm, int frame_size,
                              unsigned char *compressed,
                              int nbCompressedBytes);

/** Query and set encoder parameters
 @param st Encoder state
 @param request Parameter to change or query
 @param value Pointer to a 32-bit int value
 @return Error code
*/
  EXPORT int      celt_encoder_ctl(CELTEncoder * st, int request,
                                   ...);

  /* Decoder stuff */

/** Creates a new decoder state. Each stream needs its own decoder stat\
e (can't
    be shared across simultaneous streams).
 @param mode Contains all the information about the characteristics of \
the
             stream (must be the same characteristics as used for the e\
```

```
ncoder)
 @param channels Number of channels
 @param error Returns an error code
 @return Newly created decoder state.
 */
  EXPORT CELTDecoder *celt_decoder_create(const CELTMode * mode,
                                          int channels, int *error);

/** Destroys a a decoder state.
 @param st Decoder state to be destroyed
 */
  EXPORT void     celt_decoder_destroy(CELTDecoder * st);

/** Decodes a frame of audio.
 @param st Decoder state
 @param data Compressed data produced by an encoder
 @param len Number of bytes to read from "data". This MUST be exactly t\
he number
            of bytes returned by the encoder. Using a larger value WILL\
 NOT WORK.
 @param pcm One frame (frame_size samples per channel) of decoded PCM w\
ill be
            returned here in float format.
 @return Error code.
   */
  EXPORT int      celt_decode_float(CELTDecoder * st,
                                    const unsigned char *data,
                                    int len, float *pcm,
                                    int frame_size);

/** Decodes a frame of audio.
 @param st Decoder state
 @param data Compressed data produced by an encoder
 @param len Number of bytes to read from "data". This MUST be exactly t\
he number
            of bytes returned by the encoder. Using a larger value WILL\
 NOT WORK.
 @param pcm One frame (frame_size samples per channel) of decoded PCM w\
ill be
            returned here in 16-bit PCM format (native endian).
 @return Error code.
 */
  EXPORT int      celt_decode(CELTDecoder * st,
                              const unsigned char *data, int len,
                              celt_int16 * pcm, int frame_size);

/** Query and set decoder parameters
   @param st Decoder state
```

```
   @param request Parameter to change or query
   @param value Pointer to a 32-bit int value
   @return Error code
 */
  EXPORT int      celt_decoder_ctl(CELTDecoder * st, int request,
                                   ...);

/** Returns the English string that corresponds to an error code
 * @param error Error code (negative for an error, 0 for success
 * @return Constant string (must NOT be freed)
 */
  EXPORT const char *celt_strerror(int error);

  /* @} */

#ifdef __cplusplus
}
#endif
#endif                          /* CELT_H */
```

A.4. celt.c

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Copyright (c) 2008 Gregory Maxwell Written by
   Jean-Marc Valin and Gregory Maxwell */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
```

```
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#define CELT_C

#include "os_support.h"
#include "mdct.h"
#include <math.h>
#include "celt.h"
#include "pitch.h"
#include "bands.h"
#include "modes.h"
#include "entcode.h"
#include "quant_bands.h"
#include "rate.h"
#include "stack_alloc.h"
#include "mathops.h"
#include <stdarg.h>
#include "plc.h"

static const float preemph = (0.8f);

static const float transientWindow[16] = {
  0.0085135, 0.0337639, 0.0748914, 0.1304955,
  0.1986827, 0.2771308, 0.3631685, 0.4538658,
  0.5461342, 0.6368315, 0.7228692, 0.8013173,
  0.8695045, 0.9251086, 0.9662361, 0.9914865
};

#define ENCODERVALID   0x4c434554
#define ENCODERPARTIAL 0x5445434c
#define ENCODERFREED   0x4c004500

/** Encoder state
 @brief Encoder state
 */
struct CELTEncoder {
  celt_uint32     marker;
  const CELTMode *mode;  /**< Mode used by the encoder */
```

```
  int             overlap;
  int             channels;

  int             pitch_enabled;        /* Complexity level is
                                           allowed to use pitch */
  int             pitch_permitted;      /* Use of the LTP is
                                           permitted by the user */
  int             pitch_available;      /* Amount of pitch buffer
                                           available */
  int             force_intra;
  int             delayedIntra;
  float           tonal_average;
  int             fold_decision;
  float           gain_prod;
  float           frame_max;
  int             start,
                  end;

  /* VBR-related parameters */
  celt_int32      vbr_reservoir;
  celt_int32      vbr_drift;
  celt_int32      vbr_offset;
  celt_int32      vbr_count;

  celt_int32      vbr_rate_norm;        /* Target number of 16th
                                           bits per frame */
  float          *restrict preemph_memE;
  float          *restrict preemph_memD;

  float          *in_mem;
  float          *out_mem;
  float          *pitch_buf;
  float           xmem;

  float          *oldBandE;
};

static int
check_encoder(const CELTEncoder * st)
{
  if (st == NULL)
  {
    celt_warning("NULL passed as an encoder structure");
    return CELT_INVALID_STATE;
  }
  if (st->marker == ENCODERVALID)
    return CELT_OK;
  if (st->marker == ENCODERFREED)
```

```
    celt_warning
        ("Referencing an encoder that has already been freed");
  else
    celt_warning("This is not a valid CELT encoder structure");
  return CELT_INVALID_STATE;
}

CELTEncoder    *
celt_encoder_create(const CELTMode * mode, int channels, int *error)
{
  int             C;
  CELTEncoder    *st;

  if (check_mode(mode) != CELT_OK)
  {
    if (error)
      *error = CELT_INVALID_MODE;
    return NULL;
  }

  if (channels < 0 || channels > 2)
  {
    celt_warning("Only mono and stereo supported");
    if (error)
      *error = CELT_BAD_ARG;
    return NULL;
  }

  C = channels;
  st = celt_alloc(sizeof(CELTEncoder));

  if (st == NULL)
  {
    if (error)
      *error = CELT_ALLOC_FAIL;
    return NULL;
  }
  st->marker = ENCODERPARTIAL;
  st->mode = mode;
  st->overlap = mode->overlap;
  st->channels = channels;

  st->start = 0;
  st->end = st->mode->nbEBands;

  st->vbr_rate_norm = 0;
  st->pitch_enabled = 1;
  st->pitch_permitted = 1;
```

```
  st->pitch_available = 1;
  st->force_intra = 0;
  st->delayedIntra = 1;
  st->tonal_average = (1.f);
  st->fold_decision = 1;

  st->in_mem = celt_alloc(st->overlap * C * sizeof(float));
  st->out_mem =
      celt_alloc((MAX_PERIOD + st->overlap) * C * sizeof(float));
  st->pitch_buf =
      celt_alloc(((MAX_PERIOD >> 1) + 2) * sizeof(float));

  st->oldBandE =
      (float *) celt_alloc(C * mode->nbEBands * sizeof(float));

  st->preemph_memE = (float *) celt_alloc(C * sizeof(float));
  st->preemph_memD = (float *) celt_alloc(C * sizeof(float));

  if ((st->in_mem != NULL) && (st->out_mem != NULL)
      && (st->oldBandE != NULL) && (st->preemph_memE != NULL)
      && (st->preemph_memD != NULL))
  {
    if (error)
      *error = CELT_OK;
    st->marker = ENCODERVALID;
    return st;
  }
  /* If the setup fails for some reason deallocate it. */
  celt_encoder_destroy(st);
  if (error)
    *error = CELT_ALLOC_FAIL;
  return NULL;
}

void
celt_encoder_destroy(CELTEncoder * st)
{
  if (st == NULL)
  {
    celt_warning("NULL passed to celt_encoder_destroy");
    return;
  }

  if (st->marker == ENCODERFREED)
  {
    celt_warning("Freeing an encoder which has already been freed");
    return;
  }
```

```
  if (st->marker != ENCODERVALID && st->marker != ENCODERPARTIAL)
  {
    celt_warning("This is not a valid CELT encoder structure");
    return;
  }
  /* Check_mode is non-fatal here because we can still free the
     encoder memory even if the mode is bad, although calling the
     free functions in this order is a violation of the API. */
  check_mode(st->mode);

  celt_free(st->in_mem);
  celt_free(st->out_mem);
  celt_free(st->pitch_buf);
  celt_free(st->oldBandE);

  celt_free(st->preemph_memE);
  celt_free(st->preemph_memD);

  st->marker = ENCODERFREED;

  celt_free(st);
}

static inline celt_int16
FLOAT2INT16(float x)
{
  x = x * CELT_SIG_SCALE;
  x = MAX32(x, -32768);
  x = MIN32(x, 32767);
  return (celt_int16) float2int(x);
}

static inline float
SIG2WORD16(float x)
{

  return (float) x;

}

static int
transient_analysis(const float *restrict in, int len, int C,
                   int *transient_time, int *transient_shift,
                   float *frame_max, int overlap)
{
  int             i,
                  n;
  float           ratio;
```

```
  float           threshold;
  VARDECL(float, begin);
  SAVE_STACK;
  ALLOC(begin, len + 1, float);
  begin[0] = 0;
  if (C == 1)
  {
    for (i = 0; i < len; i++)
      begin[i + 1] = MAX32(begin[i], ABS32(in[i]));
  } else
  {
    for (i = 0; i < len; i++)
      begin[i + 1] = MAX32(begin[i], MAX32(ABS32(in[C * i]),
                                           ABS32(in[C * i + 1])));
  }
  n = -1;

  threshold = (((.2f)) * (begin[len]));
  /* If the following condition isn't met, there's just no way we'll
     have a transient */
  if (*frame_max < threshold)
  {
    /* It's likely we have a transient, now find it */
    for (i = 8; i < len - 8; i++)
    {
      if (begin[i + 1] < threshold)
        n = i;
    }
  }
  if (n < 32)
  {
    n = -1;
    ratio = 0;
  } else
  {
    ratio = ((begin[len]) / (1 + MAX32(*frame_max, begin[n - 16])));
  }
  if (ratio < 0)
    ratio = 0;
  if (ratio > 1000)
    ratio = 1000;

  if (ratio > 45)
    *transient_shift = 3;
  else
    *transient_shift = 0;

  *transient_time = n;
```

```
  *frame_max = begin[len - overlap];

  RESTORE_STACK;
  return ratio > 4;
}

/** Apply window and compute the MDCT for all sub-frames and
    all channels in a frame */
static void
compute_mdcts(const CELTMode * mode, int shortBlocks,
              float *restrict in, float *restrict out, int _C,
              int LM)
{
  const int       C = CHANNELS(_C);
  if (C == 1 && !shortBlocks)
  {
    const int       overlap = OVERLAP(mode);
    clt_mdct_forward(&mode->mdct, in, out, mode->window, overlap,
                     mode->maxLM - LM);
  } else
  {
    const int       overlap = OVERLAP(mode);
    int             N = mode->shortMdctSize << LM;
    int             B = 1;
    int             b,
                    c;
    VARDECL(float, x);
    VARDECL(float, tmp);
    SAVE_STACK;
    if (shortBlocks)
    {
      /* lookup = &mode->mdct[0]; */
      N = mode->shortMdctSize;
      B = shortBlocks;
    }
    ALLOC(x, N + overlap, float);
    ALLOC(tmp, N, float);
    for (c = 0; c < C; c++)
    {
      for (b = 0; b < B; b++)
      {
        int             j;
        for (j = 0; j < N + overlap; j++)
          x[j] = in[C * (b * N + j) + c];
        clt_mdct_forward(&mode->mdct, x, tmp, mode->window, overlap,
                         shortBlocks ? mode->maxLM : mode->maxLM -
                         LM);
        /* Interleaving the sub-frames */
```

```
        for (j = 0; j < N; j++)
          out[(j * B + b) + c * N * B] = tmp[j];
      }
    }
    RESTORE_STACK;
  }
}

/** Compute the IMDCT and apply window for all sub-frames and
    all channels in a frame */
static void
compute_inv_mdcts(const CELTMode * mode, int shortBlocks, float *X,
                  int transient_time, int transient_shift,
                  float *restrict out_mem, int _C, int LM)
{
  int             c,
                  N4;
  const int       C = CHANNELS(_C);
  const int       N = mode->shortMdctSize << LM;
  const int       overlap = OVERLAP(mode);
  N4 = (N - overlap) >> 1;
  for (c = 0; c < C; c++)
  {
    int             j;
    if (transient_shift == 0 && C == 1 && !shortBlocks)
    {
      clt_mdct_backward(&mode->mdct, X,
                        out_mem + C * (MAX_PERIOD - N - N4),
                        mode->window, overlap, mode->maxLM - LM);
    } else
    {
      VARDECL(float, x);
      VARDECL(float, tmp);
      int             b;
      int             N2 = N;
      int             B = 1;
      int             n4offset = 0;
      SAVE_STACK;

      ALLOC(x, 2 * N, float);
      ALLOC(tmp, N, float);

      if (shortBlocks)
      {
        /* lookup = &mode->mdct[0]; */
        N2 = mode->shortMdctSize;
        B = shortBlocks;
        n4offset = N4;
```

```
      }
      /* Prevents problems from the imdct doing the overlap-add */
      CELT_MEMSET(x + N4, 0, N2);

      for (b = 0; b < B; b++)
      {
        /* De-interleaving the sub-frames */
        for (j = 0; j < N2; j++)
          tmp[j] = X[(j * B + b) + c * N2 * B];
        clt_mdct_backward(&mode->mdct, tmp, x + n4offset + N2 * b,
                          mode->window, overlap,
                          shortBlocks ? mode->maxLM : mode->maxLM -
                          LM);
      }

      if (transient_shift > 0)
      {

        for (j = 0; j < 16; j++)
          x[N4 + transient_time + j - 16] *=
              1 + transientWindow[j] * ((1 << transient_shift) - 1);
        for (j = transient_time; j < N + overlap; j++)
          x[N4 + j] *= 1 << transient_shift;

      }
      /* The first and last part would need to be set to zero if we
         actually wanted to use them. */
      for (j = 0; j < overlap; j++)
        out_mem[C * (MAX_PERIOD - N) + C * j + c] += x[j + N4];
      for (j = 0; j < overlap; j++)
        out_mem[C * (MAX_PERIOD) + C * (overlap - j - 1) + c] =
            x[2 * N - j - N4 - 1];
      for (j = 0; j < 2 * N4; j++)
        out_mem[C * (MAX_PERIOD - N) + C * (j + overlap) + c] =
            x[j + N4 + overlap];
      RESTORE_STACK;
    }
  }
}

#define FLAG_NONE        0
#define FLAG_INTRA       (1U<<13)
#define FLAG_PITCH       (1U<<12)
#define FLAG_SHORT       (1U<<11)
#define FLAG_FOLD        (1U<<10)
#define FLAG_MASK        (FLAG_INTRA|FLAG_PITCH|FLAG_SHORT|FLAG_FOLD)

static const int flaglist[8] = {
```

```
  0 /* 00 */  | FLAG_FOLD,
  1 /* 01 */  | FLAG_PITCH | FLAG_FOLD,
  8 /* 1000 */  | FLAG_NONE,
  9 /* 1001 */  | FLAG_SHORT | FLAG_FOLD,
  10 /* 1010 */  | FLAG_PITCH,
  11 /* 1011 */  | FLAG_INTRA,
  6 /* 110 */  | FLAG_INTRA | FLAG_FOLD,
  7 /* 111 */  | FLAG_INTRA | FLAG_SHORT | FLAG_FOLD
};

static void
encode_flags(ec_enc * enc, int intra_ener, int has_pitch,
             int shortBlocks, int has_fold)
{
  int             i;
  int             flags = FLAG_NONE;
  int             flag_bits;
  flags |= intra_ener ? FLAG_INTRA : 0;
  flags |= has_pitch ? FLAG_PITCH : 0;
  flags |= shortBlocks ? FLAG_SHORT : 0;
  flags |= has_fold ? FLAG_FOLD : 0;
  for (i = 0; i < 8; i++)
  {
    if (flags == (flaglist[i] & FLAG_MASK))
    {
      flag_bits = flaglist[i] & 0xf;
      break;
    }
  }
  celt_assert(i < 8);
  /* printf ("enc %d: %d %d %d %d\n", flag_bits, intra_ener,
     has_pitch, shortBlocks, has_fold); */
  if (i < 2)
    ec_enc_uint(enc, flag_bits, 4);
  else if (i < 6)
  {
    ec_enc_uint(enc, flag_bits >> 2, 4);
    ec_enc_uint(enc, flag_bits & 0x3, 4);
  } else
  {
    ec_enc_uint(enc, flag_bits >> 1, 4);
    ec_enc_uint(enc, flag_bits & 0x1, 2);
  }
}

static void
decode_flags(ec_dec * dec, int *intra_ener, int *has_pitch,
             int *shortBlocks, int *has_fold)
```

```
{
  int             i;
  int             flag_bits;
  flag_bits = ec_dec_uint(dec, 4);
  /* printf ("(%d) ", flag_bits); */
  if (flag_bits == 2)
    flag_bits = (flag_bits << 2) | ec_dec_uint(dec, 4);
  else if (flag_bits == 3)
    flag_bits = (flag_bits << 1) | ec_dec_uint(dec, 2);
  for (i = 0; i < 8; i++)
    if (flag_bits == (flaglist[i] & 0xf))
      break;
  celt_assert(i < 8);
  *intra_ener = (flaglist[i] & FLAG_INTRA) != 0;
  *has_pitch = (flaglist[i] & FLAG_PITCH) != 0;
  *shortBlocks = (flaglist[i] & FLAG_SHORT) != 0;
  *has_fold = (flaglist[i] & FLAG_FOLD) != 0;
  /* printf ("dec %d: %d %d %d %d\n", flag_bits, *intra_ener,
     *has_pitch, *shortBlocks, *has_fold); */
}

void
deemphasis(float *in, float *pcm, int N, int _C, float coef,
           float *mem)
{
  const int       C = CHANNELS(_C);
  int             c;
  for (c = 0; c < C; c++)
  {
    int             j;
    float          *restrict x;
    float          *restrict y;
    float           m = mem[c];
    x = &in[C * (MAX_PERIOD - N) + c];
    y = pcm + c;
    for (j = 0; j < N; j++)
    {
      float           tmp = ((*x) + (coef) * (m));
      m = tmp;
      *y = SCALEOUT(SIG2WORD16(tmp));
      x += C;
      y += C;
    }
    mem[c] = m;
  }
}

static void
```

```
mdct_shape(const CELTMode * mode, float *X, int start,
           int end, int N,
           int mdct_weight_shift, int _C, int renorm, int M)
{
  int             m,
                  i,
                  c;
  const int       C = CHANNELS(_C);
  for (c = 0; c < C; c++)
    for (m = start; m < end; m++)
      for (i = m + c * N; i < (c + 1) * N; i += M)

        X[i] = (1.f / (1 << mdct_weight_shift)) * X[i];

  if (renorm)
    renormalise_bands(mode, X, C, M);
}

static signed char tf_select_table[4][8] = {
  {0, -1, 0, -1, 0, -1, 0, -1},
  {0, -1, 0, -2, 1, 0, 1 - 1},
  {0, -2, 0, -3, 2, 0, 1 - 1},
  {0, -2, 0, -3, 2, 0, 1 - 1},
};

static int
tf_analysis(float *bandLogE, float *oldBandE, int len, int C,
            int isTransient, int *tf_res, int nbCompressedBytes)
{
  int             i;
  float           threshold;
  VARDECL(float, metric);
  float           average = 0;
  float           cost0;
  float           cost1;
  VARDECL(int, path0);
  VARDECL(int, path1);
  float           lambda;
  int             tf_select = 0;
  SAVE_STACK;

  if (nbCompressedBytes < 40)
    lambda = (5.f);
  else if (nbCompressedBytes < 60)
    lambda = (2.f);
  else if (nbCompressedBytes < 100)
    lambda = (1.f);
  else
```

```
    lambda = (.5f);

  ALLOC(metric, len, float);
  ALLOC(path0, len, int);
  ALLOC(path1, len, int);
  for (i = 0; i < len; i++)
  {
    metric[i] = ((bandLogE[i]) - (oldBandE[i]));
    average += metric[i];
  }
  if (C == 2)
  {
    average = 0;
    for (i = 0; i < len; i++)
    {
      metric[i] =
          (.5f * (metric[i])) +
          (.5f * (((bandLogE[i + len]) - (oldBandE[i + len]))));
      average += metric[i];
    }
  }
  average = ((average) / (len));
  /* if (!isTransient) printf ("%f\n", average); */
  if (isTransient)
  {
    threshold = (1.f);
    tf_select = average > (3.f);
  } else
  {
    threshold = (.5f);
    tf_select = average > (1.f);
  }
  cost0 = 0;
  cost1 = lambda;
  /* Viterbi forward pass */
  for (i = 1; i < len; i++)
  {
    float           curr0,
                    curr1;
    float           from0,
                    from1;

    from0 = cost0;
    from1 = cost1 + lambda;
    if (from0 < from1)
    {
      curr0 = from0;
      path0[i] = 0;
```

```
    } else
    {
      curr0 = from1;
      path0[i] = 1;
    }

    from0 = cost0 + lambda;
    from1 = cost1;
    if (from0 < from1)
    {
      curr1 = from0;
      path1[i] = 0;
    } else
    {
      curr1 = from1;
      path1[i] = 1;
    }
    cost0 = curr0 + (metric[i] - threshold);
    cost1 = curr1;
  }
  tf_res[len - 1] = cost0 < cost1 ? 0 : 1;
  /* Viterbi backward pass to check the decisions */
  for (i = len - 2; i >= 0; i--)
  {
    if (tf_res[i + 1] == 1)
      tf_res[i] = path1[i + 1];
    else
      tf_res[i] = path0[i + 1];
  }
  RESTORE_STACK;
  return tf_select;
}

static void
tf_encode(int len, int isTransient, int *tf_res,
          int nbCompressedBytes, int LM, int tf_select, ec_enc * enc)
{
  int             curr,
                  i;
  if (8 * nbCompressedBytes - ec_enc_tell(enc, 0) < 100)
  {
    for (i = 0; i < len; i++)
      tf_res[i] = isTransient;
  } else
  {
    ec_enc_bit_prob(enc, tf_res[0], isTransient ? 16384 : 4096);
    curr = tf_res[0];
    for (i = 1; i < len; i++)
```

```
    {
      ec_enc_bit_prob(enc, tf_res[i] ^ curr,
                      isTransient ? 4096 : 2048);
      curr = tf_res[i];
    }
  }
  ec_enc_bits(enc, tf_select, 1);
  for (i = 0; i < len; i++)
    tf_res[i] =
        tf_select_table[LM][4 * isTransient + 2 * tf_select +
                            tf_res[i]];
}

static void
tf_decode(int len, int C, int isTransient, int *tf_res,
          int nbCompressedBytes, int LM, ec_dec * dec)
{
  int             i,
                  curr,
                  tf_select;
  if (8 * nbCompressedBytes - ec_dec_tell(dec, 0) < 100)
  {
    for (i = 0; i < len; i++)
      tf_res[i] = isTransient;
  } else
  {
    tf_res[0] = ec_dec_bit_prob(dec, isTransient ? 16384 : 4096);
    curr = tf_res[0];
    for (i = 1; i < len; i++)
    {
      tf_res[i] =
          ec_dec_bit_prob(dec, isTransient ? 4096 : 2048) ^ curr;
      curr = tf_res[i];
    }
  }
  tf_select = ec_dec_bits(dec, 1);
  for (i = 0; i < len; i++)
    tf_res[i] =
        tf_select_table[LM][4 * isTransient + 2 * tf_select +
                            tf_res[i]];
}

int
celt_encode_with_ec_float(CELTEncoder * restrict st,
                          const float *pcm,
                          float *optional_resynthesis,
                          int frame_size, unsigned char *compressed,
                          int nbCompressedBytes, ec_enc * enc)
```

```
{

  int             i,
                  c,
                  N,
                  NN,
                  N4;
  int             has_pitch;
  int             pitch_index;
  int             bits;
  int             has_fold = 1;
  int             coarse_needed;
  ec_byte_buffer  buf;
  ec_enc          _enc;
  VARDECL(float, in);
  VARDECL(float, freq);
  VARDECL(float, pitch_freq);
  VARDECL(float, X);
  VARDECL(float, bandE);
  VARDECL(float, bandLogE);
  VARDECL(int, fine_quant);
  VARDECL(float, error);
  VARDECL(int, pulses);
  VARDECL(int, offsets);
  VARDECL(int, fine_priority);
  VARDECL(int, tf_res);
  int             intra_ener = 0;
  int             shortBlocks = 0;
  int             isTransient = 0;
  int             transient_time;
  int             transient_shift;
  int             resynth;
  const int       C = CHANNELS(st->channels);
  int             mdct_weight_shift = 0;
  int             mdct_weight_pos = 0;
  int             gain_id = 0;
  int             norm_rate;
  int             LM,
                  M;
  int             tf_select;
  celt_int32      vbr_rate = 0;
  float           max_decay;
  int             nbFilledBytes,
                  nbAvailableBytes;
  SAVE_STACK;

  if (check_encoder(st) != CELT_OK)
    return CELT_INVALID_STATE;
```

```
  if (check_mode(st->mode) != CELT_OK)
    return CELT_INVALID_MODE;

  if (nbCompressedBytes < 0 || pcm == NULL)
    return CELT_BAD_ARG;

  for (LM = 0; LM < 4; LM++)
    if (st->mode->shortMdctSize << LM == frame_size)
      break;
  if (LM >= MAX_CONFIG_SIZES)
    return CELT_BAD_ARG;
  M = 1 << LM;

  if (enc == NULL)
  {
    ec_byte_writeinit_buffer(&buf, compressed, nbCompressedBytes);
    ec_enc_init(&_enc, &buf);
    enc = &_enc;
    nbFilledBytes = 0;
  } else
  {
    nbFilledBytes = (ec_enc_tell(enc, 0) + 4) >> 3;
  }
  nbAvailableBytes = nbCompressedBytes - nbFilledBytes;

  N = M * st->mode->shortMdctSize;
  N4 = (N - st->overlap) >> 1;
  ALLOC(in, 2 * C * N - 2 * C * N4, float);

  CELT_COPY(in, st->in_mem, C * st->overlap);
  for (c = 0; c < C; c++)
  {
    const float    *restrict pcmp = pcm + c;
    float          *restrict inp = in + C * st->overlap + c;
    for (i = 0; i < N; i++)
    {
      /* Apply pre-emphasis */
      float           tmp = SCALEIN(((*pcmp)));
      *inp = ((tmp) - ((((preemph) * (st->preemph_memE[c])))));
      st->preemph_memE[c] = SCALEIN(*pcmp);
      inp += C;
      pcmp += C;
    }
  }
  CELT_COPY(st->in_mem, in + C * (2 * N - 2 * N4 - st->overlap),
            C * st->overlap);

  /* Transient handling */
```

```
  transient_time = -1;
  transient_shift = 0;
  isTransient = 0;

  resynth = st->pitch_available > 0 || optional_resynthesis != NULL;

  if (M > 1
      && transient_analysis(in, N + st->overlap, C, &transient_time,
                            &transient_shift, &st->frame_max,
                            st->overlap))
  {

    float           gain_1;

    /* Apply the inverse shaping window */
    if (transient_shift)
    {
      for (c = 0; c < C; c++)
        for (i = 0; i < 16; i++)
          in[C * (transient_time + i - 16) + c] /= 1 +
              transientWindow[i] * ((1 << transient_shift) - 1);
      gain_1 = 1. / (1 << transient_shift);
      for (c = 0; c < C; c++)
        for (i = transient_time; i < N + st->overlap; i++)
          in[C * i + c] *= gain_1;

    }
    isTransient = 1;
    has_fold = 1;
  }

  if (isTransient)
    shortBlocks = M;
  else
    shortBlocks = 0;

  ALLOC(freq, C * N, float);/**< Interleaved signal MDCTs */
  ALLOC(bandE, st->mode->nbEBands * C, float);
  ALLOC(bandLogE, st->mode->nbEBands * C, float);
  /* Compute MDCTs */
  compute_mdcts(st->mode, shortBlocks, in, freq, C, LM);

  norm_rate =
      (nbAvailableBytes -
       5) * 8 * (celt_uint32) st->mode->Fs / (C * N) >> 10;
  /* Pitch analysis: we do it early to save on the peak stack space */
  /* Don't use pitch if there isn't enough data available yet, or
     if we're using shortBlocks */
```

```
  has_pitch = st->pitch_enabled && st->pitch_permitted && (N <= 512)
      && (st->pitch_available >= MAX_PERIOD) && (!shortBlocks)
      && norm_rate < 50;
  if (has_pitch)
  {
    VARDECL(float, x_lp);
    SAVE_STACK;
    ALLOC(x_lp, (2 * N - 2 * N4) >> 1, float);
    pitch_downsample(in, x_lp, 2 * N - 2 * N4, N, C, &st->xmem,
                     &st->pitch_buf[MAX_PERIOD >> 1]);
    pitch_search(st->mode, x_lp, st->pitch_buf, 2 * N - 2 * N4,
                 MAX_PERIOD - (2 * N - 2 * N4), &pitch_index,
                 &st->xmem, M);
    RESTORE_STACK;
  }

  /* Deferred allocation after find_spectral_pitch() to reduce the
     peak memory usage */
  ALLOC(X, C * N, float);/**< Interleaved normalised MDCTs */

  ALLOC(pitch_freq, C * N, float);/**< Interleaved signal MDCTs */
  if (has_pitch)
  {
    compute_mdcts(st->mode, 0, st->out_mem + pitch_index * C,
                  pitch_freq, C, LM);
    has_pitch =
        compute_pitch_gain(st->mode, freq, pitch_freq, norm_rate,
                           &gain_id, C, &st->gain_prod, M);
  }

  if (has_pitch)
    apply_pitch(st->mode, freq, pitch_freq, gain_id, 1, C, M);

  compute_band_energies(st->mode, freq, bandE, C, M);
  for (i = 0; i < st->mode->nbEBands * C; i++)
    bandLogE[i] = amp2Log(bandE[i]);

  /* Band normalisation */
  normalise_bands(st->mode, freq, X, bandE, C, M);
  if (!shortBlocks
      && !folding_decision(st->mode, X, &st->tonal_average,
                           &st->fold_decision, C, M))
    has_fold = 0;

  /* Don't use intra energy when we're operating at low bit-rate */
  intra_ener = st->force_intra || (!has_pitch && st->delayedIntra
                                   && nbAvailableBytes >
                                   st->mode->nbEBands);
```

```
  if (shortBlocks
      || intra_decision(bandLogE, st->oldBandE, st->mode->nbEBands))
    st->delayedIntra = 1;
  else
    st->delayedIntra = 0;

  NN = M * st->mode->eBands[st->mode->nbEBands];
  if (shortBlocks && !transient_shift)
  {
    float           sum[8] = { 1, 1, 1, 1, 1, 1, 1, 1 };
    int             m;
    for (c = 0; c < C; c++)
    {
      m = 0;
      do
      {
        float           tmp = 0;
        for (i = m + c * N; i < c * N + NN; i += M)
          tmp += ABS32(X[i]);
        sum[m++] += tmp;
      }
      while (m < M);
    }
    m = 0;
    do
    {
      if (sum[m + 1] > 8 * sum[m])
      {
        mdct_weight_shift = 2;
        mdct_weight_pos = m;
      } else if (sum[m + 1] > 2 * sum[m] && mdct_weight_shift < 2)
      {
        mdct_weight_shift = 1;
        mdct_weight_pos = m;
      }
      m++;
    }
    while (m < M - 1);

    if (mdct_weight_shift)
      mdct_shape(st->mode, X, mdct_weight_pos + 1, M, N,
                 mdct_weight_shift, C, 0, M);
  }

  encode_flags(enc, intra_ener, has_pitch, shortBlocks, has_fold);
  if (has_pitch)
  {
    ec_enc_uint(enc, pitch_index, MAX_PERIOD - (2 * N - 2 * N4));
```

```
    ec_enc_uint(enc, gain_id, 16);
  }
  if (shortBlocks)
  {
    if (transient_shift)
    {
      ec_enc_uint(enc, transient_shift, 4);
      ec_enc_uint(enc, transient_time, N + st->overlap);
    } else
    {
      ec_enc_uint(enc, mdct_weight_shift, 4);
      if (mdct_weight_shift && M != 2)
        ec_enc_uint(enc, mdct_weight_pos, M - 1);
    }
  }

  ALLOC(fine_quant, st->mode->nbEBands, int);
  ALLOC(pulses, st->mode->nbEBands, int);

  vbr_rate = M * st->vbr_rate_norm;
  /* Computes the max bit-rate allowed in VBR more to avoid busting
     the budget */
  if (st->vbr_rate_norm > 0)
  {
    celt_int32      vbr_bound,
                    max_allowed;

    vbr_bound = vbr_rate;
    max_allowed =
        (vbr_rate + vbr_bound - st->vbr_reservoir) >> (BITRES + 3);
    if (max_allowed < 4)
      max_allowed = 4;
    if (max_allowed < nbAvailableBytes)
      nbAvailableBytes = max_allowed;
  }

  ALLOC(tf_res, st->mode->nbEBands, int);
  tf_select =
      tf_analysis(bandLogE, st->oldBandE, st->mode->nbEBands, C,
                  isTransient, tf_res, nbAvailableBytes);

  /* Bit allocation */
  ALLOC(error, C * st->mode->nbEBands, float);

  max_decay = .125 * nbAvailableBytes;

  coarse_needed =
      quant_coarse_energy(st->mode, st->start, bandLogE,
```

```
                          st->oldBandE,
                          nbFilledBytes * 8 + nbAvailableBytes * 4 -
                          8, intra_ener, st->mode->prob, error, enc,
                          C, max_decay);
  coarse_needed -= nbFilledBytes * 8;
  coarse_needed = ((coarse_needed * 3 - 1) >> 3) + 1;
  if (coarse_needed > nbAvailableBytes)
    coarse_needed = nbAvailableBytes;
  /* Variable bitrate */
  if (vbr_rate > 0)
  {
    float           alpha;
    celt_int32      delta;
    /* The target rate in 16th bits per frame */
    celt_int32      target = vbr_rate;

    /* Shortblocks get a large boost in bitrate, but since they are
       uncommon long blocks are not greatly effected */
    if (shortBlocks)
      target *= 2;
    else if (M > 1)
      target -= (target + 14) / 28;

    /* The average energy is removed from the target and the actual
       energy added */
    target =
        target + st->vbr_offset - 588 + ec_enc_tell(enc, BITRES);

    /* In VBR mode the frame size must not be reduced so much that
       it would result in the coarse energy busting its budget */
    target = IMAX(coarse_needed, (target + 64) / 128);
    target = IMIN(nbAvailableBytes, target);
    /* Make the adaptation coef (alpha) higher at the beginning */
    if (st->vbr_count < 990)
    {
      st->vbr_count++;
      alpha = celt_rcp(((st->vbr_count + 10)));
      /* printf ("%d %d\n", st->vbr_count+10, alpha); */
    } else
      alpha = (.001f);

    /* By how much did we "miss" the target on that frame */
    delta = (8 << BITRES) * (celt_int32) target - vbr_rate;
    /* How many bits have we used in excess of what we're allowed */
    st->vbr_reservoir += delta;
    /* printf ("%d\n", st->vbr_reservoir); */

    /* Compute the offset we need to apply in order to reach the
```

```
       target */
    st->vbr_drift +=
        ((alpha) * (delta - st->vbr_offset - st->vbr_drift));
    st->vbr_offset = -st->vbr_drift;
    /* printf ("%d\n", st->vbr_drift); */

    /* We could use any multiple of vbr_rate as bound (depending on
       the delay) */
    if (st->vbr_reservoir < 0)
    {
      /* We're under the min value -- increase rate */
      int             adjust =
          1 - (st->vbr_reservoir - 1) / (8 << BITRES);
      st->vbr_reservoir += adjust * (8 << BITRES);
      target += adjust;
      /* printf ("+%d\n", adjust); */
    }
    if (target < nbAvailableBytes)
      nbAvailableBytes = target;
    nbCompressedBytes = nbAvailableBytes + nbFilledBytes;

    /* This moves the raw bits to take into account the new
       compressed size */
    ec_byte_shrink(&buf, nbCompressedBytes);
  }

  tf_encode(st->mode->nbEBands, isTransient, tf_res,
            nbAvailableBytes, LM, tf_select, enc);

  ALLOC(offsets, st->mode->nbEBands, int);
  ALLOC(fine_priority, st->mode->nbEBands, int);

  for (i = 0; i < st->mode->nbEBands; i++)
    offsets[i] = 0;
  bits = nbCompressedBytes * 8 - ec_enc_tell(enc, 0) - 1;
  compute_allocation(st->mode, st->start, offsets, bits, pulses,
                     fine_quant, fine_priority, C, M);

  quant_fine_energy(st->mode, st->start, bandE, st->oldBandE, error,
                    fine_quant, enc, C);

  /* Residual quantisation */
  quant_all_bands(1, st->mode, st->start, X, C == 2 ? X + N : NULL,
                  bandE, pulses, shortBlocks, has_fold, tf_res,
                  resynth, nbCompressedBytes * 8, enc, LM);

  quant_energy_finalise(st->mode, st->start, bandE, st->oldBandE,
                        error, fine_quant, fine_priority,
```

```
                      nbCompressedBytes * 8 - ec_enc_tell(enc, 0),
                      enc, C);

  /* Re-synthesis of the coded audio if required */
  if (resynth)
  {
    if (st->pitch_available > 0 && st->pitch_available < MAX_PERIOD)
      st->pitch_available += N;

    if (mdct_weight_shift)
    {
      mdct_shape(st->mode, X, 0, mdct_weight_pos + 1, N,
                 mdct_weight_shift, C, 1, M);
    }

    /* Synthesis */
    denormalise_bands(st->mode, X, freq, bandE, C, M);

    CELT_MOVE(st->out_mem, st->out_mem + C * N,
              C * (MAX_PERIOD + st->overlap - N));

    if (has_pitch)
      apply_pitch(st->mode, freq, pitch_freq, gain_id, 0, C, M);

    compute_inv_mdcts(st->mode, shortBlocks, freq, transient_time,
                      transient_shift, st->out_mem, C, LM);

    /* De-emphasis and put everything back at the right place in
       the synthesis history */
    if (optional_resynthesis != NULL)
    {
      deemphasis(st->out_mem, optional_resynthesis, N, C, preemph,
                 st->preemph_memD);

    }
  }

  /* If there's any room left (can only happen for very high rates),
     fill it with zeros */
  while (nbCompressedBytes * 8 - ec_enc_tell(enc, 0) >= 8)
    ec_enc_bits(enc, 0, 8);
  ec_enc_done(enc);

  RESTORE_STACK;
  return nbCompressedBytes;
}
int
celt_encode_with_ec(CELTEncoder * restrict st,
```

```
                    const celt_int16 * pcm,
                    celt_int16 * optional_resynthesis,
                    int frame_size, unsigned char *compressed,
                    int nbCompressedBytes, ec_enc * enc)
{
  int             j,
                  ret,
                  C,
                  N,
                  LM,
                  M;
  VARDECL(float, in);
  SAVE_STACK;

  if (check_encoder(st) != CELT_OK)
    return CELT_INVALID_STATE;

  if (check_mode(st->mode) != CELT_OK)
    return CELT_INVALID_MODE;

  if (pcm == NULL)
    return CELT_BAD_ARG;

  for (LM = 0; LM < 4; LM++)
    if (st->mode->shortMdctSize << LM == frame_size)
      break;
  if (LM >= MAX_CONFIG_SIZES)
    return CELT_BAD_ARG;
  M = 1 << LM;

  C = CHANNELS(st->channels);
  N = M * st->mode->shortMdctSize;
  ALLOC(in, C * N, float);
  for (j = 0; j < C * N; j++)
  {
    in[j] = SCALEOUT(pcm[j]);
  }

  if (optional_resynthesis != NULL)
  {
    ret =
        celt_encode_with_ec_float(st, in, in, frame_size, compressed,
                                  nbCompressedBytes, enc);
    for (j = 0; j < C * N; j++)
      optional_resynthesis[j] = FLOAT2INT16(in[j]);
  } else
  {
    ret =
```

```
        celt_encode_with_ec_float(st, in, NULL, frame_size,
                                  compressed, nbCompressedBytes,
                                  enc);
  }
  RESTORE_STACK;
  return ret;
}

int
celt_encode(CELTEncoder * restrict st, const celt_int16 * pcm,
            int frame_size, unsigned char *compressed,
            int nbCompressedBytes)
{
  return celt_encode_with_ec(st, pcm, NULL, frame_size, compressed,
                             nbCompressedBytes, NULL);
}

int
celt_encode_float(CELTEncoder * restrict st, const float *pcm,
                  int frame_size, unsigned char *compressed,
                  int nbCompressedBytes)
{
  return celt_encode_with_ec_float(st, pcm, NULL, frame_size,
                                   compressed, nbCompressedBytes,
                                   NULL);
}

int
celt_encode_resynthesis(CELTEncoder * restrict st,
                        const celt_int16 * pcm,
                        celt_int16 * optional_resynthesis,
                        int frame_size, unsigned char *compressed,
                        int nbCompressedBytes)
{
  return celt_encode_with_ec(st, pcm, optional_resynthesis,
                             frame_size, compressed,
                             nbCompressedBytes, NULL);
}

int
celt_encode_resynthesis_float(CELTEncoder * restrict st,
                              const float *pcm,
                              float *optional_resynthesis,
                              int frame_size,
                              unsigned char *compressed,
                              int nbCompressedBytes)
{
  return celt_encode_with_ec_float(st, pcm, optional_resynthesis,
```

```
                                    frame_size, compressed,
                                    nbCompressedBytes, NULL);
}

int
celt_encoder_ctl(CELTEncoder * restrict st, int request, ...)
{
  va_list         ap;

  if (check_encoder(st) != CELT_OK)
    return CELT_INVALID_STATE;

  va_start(ap, request);
  if ((request != CELT_GET_MODE_REQUEST)
      && (check_mode(st->mode) != CELT_OK))
    goto bad_mode;
  switch (request)
  {
  case CELT_GET_MODE_REQUEST:
    {
      const CELTMode **value = va_arg(ap, const CELTMode **);
      if (value == 0)
        goto bad_arg;
      *value = st->mode;
    }
    break;
  case CELT_SET_COMPLEXITY_REQUEST:
    {
      int             value = va_arg(ap, celt_int32);
      if (value < 0 || value > 10)
        goto bad_arg;
      if (value <= 2)
      {
        st->pitch_enabled = 0;
        st->pitch_available = 0;
      } else
      {
        st->pitch_enabled = 1;
        if (st->pitch_available < 1)
          st->pitch_available = 1;
      }
    }
    break;
  case CELT_SET_START_BAND_REQUEST:
    {
      celt_int32      value = va_arg(ap, celt_int32);
      if (value < 0 || value >= st->mode->nbEBands)
        goto bad_arg;
```

```
      st->start = value;
    }
    break;
  case CELT_SET_PREDICTION_REQUEST:
    {
      int             value = va_arg(ap, celt_int32);
      if (value < 0 || value > 2)
        goto bad_arg;
      if (value == 0)
      {
        st->force_intra = 1;
        st->pitch_permitted = 0;
      } else if (value == 1)
      {
        st->force_intra = 0;
        st->pitch_permitted = 0;
      } else
      {
        st->force_intra = 0;
        st->pitch_permitted = 1;
      }
    }
    break;
  case CELT_SET_VBR_RATE_REQUEST:
    {
      celt_int32      value = va_arg(ap, celt_int32);
      int             frame_rate;
      int             N = st->mode->shortMdctSize;
      if (value < 0)
        goto bad_arg;
      if (value > 3072000)
        value = 3072000;
      frame_rate = ((st->mode->Fs << 3) + (N >> 1)) / N;
      st->vbr_rate_norm =
          ((value << (BITRES + 3)) + (frame_rate >> 1)) / frame_rate;
    }
    break;
  case CELT_RESET_STATE:
    {
      const CELTMode *mode = st->mode;
      int             C = st->channels;

      if (st->pitch_available > 0)
        st->pitch_available = 1;

      CELT_MEMSET(st->in_mem, 0, st->overlap * C);
      CELT_MEMSET(st->out_mem, 0, (MAX_PERIOD + st->overlap) * C);
```

```
      CELT_MEMSET(st->oldBandE, 0, C * mode->nbEBands);

      CELT_MEMSET(st->preemph_memE, 0, C);
      CELT_MEMSET(st->preemph_memD, 0, C);
      st->delayedIntra = 1;

      st->fold_decision = 1;
      st->tonal_average = (1.f);
      st->gain_prod = 0;
      st->vbr_reservoir = 0;
      st->vbr_drift = 0;
      st->vbr_offset = 0;
      st->vbr_count = 0;
      st->xmem = 0;
      st->frame_max = 0;
      CELT_MEMSET(st->pitch_buf, 0, (MAX_PERIOD >> 1) + 2);
    }
    break;
  default:
    goto bad_request;
  }
  va_end(ap);
  return CELT_OK;
bad_mode:
  va_end(ap);
  return CELT_INVALID_MODE;
bad_arg:
  va_end(ap);
  return CELT_BAD_ARG;
bad_request:
  va_end(ap);
  return CELT_UNIMPLEMENTED;
}

/**********************************************************************/
/* */
/* DECODER */
/* */
/**********************************************************************/
#define DECODE_BUFFER_SIZE 2048

#define DECODERVALID   0x4c434454
#define DECODERPARTIAL 0x5444434c
#define DECODERFREED   0x4c004400

/** Decoder state
 @brief Decoder state
 */
```

```
struct CELTDecoder {
  celt_uint32     marker;
  const CELTMode *mode;
  int             overlap;
  int             channels;

  int             start,
                  end;
  ec_byte_buffer  buf;
  ec_enc          enc;

  float          *restrict preemph_memD;

  float          *out_mem;
  float          *decode_mem;

  float          *oldBandE;

  float          *lpc;

  int             last_pitch_index;
  int             loss_count;
};

int
check_decoder(const CELTDecoder * st)
{
  if (st == NULL)
  {
    celt_warning("NULL passed a decoder structure");
    return CELT_INVALID_STATE;
  }
  if (st->marker == DECODERVALID)
    return CELT_OK;
  if (st->marker == DECODERFREED)
    celt_warning
        ("Referencing a decoder that has already been freed");
  else
    celt_warning("This is not a valid CELT decoder structure");
  return CELT_INVALID_STATE;
}

CELTDecoder    *
celt_decoder_create(const CELTMode * mode, int channels, int *error)
{
  int             C;
  CELTDecoder    *st;
```

```
  if (check_mode(mode) != CELT_OK)
  {
    if (error)
      *error = CELT_INVALID_MODE;
    return NULL;
  }

  if (channels < 0 || channels > 2)
  {
    celt_warning("Only mono and stereo supported");
    if (error)
      *error = CELT_BAD_ARG;
    return NULL;
  }

  C = CHANNELS(channels);
  st = celt_alloc(sizeof(CELTDecoder));

  if (st == NULL)
  {
    if (error)
      *error = CELT_ALLOC_FAIL;
    return NULL;
  }

  st->marker = DECODERPARTIAL;
  st->mode = mode;
  st->overlap = mode->overlap;
  st->channels = channels;

  st->start = 0;
  st->end = st->mode->nbEBands;

  st->decode_mem =
      celt_alloc((DECODE_BUFFER_SIZE +
                  st->overlap) * C * sizeof(float));
  st->out_mem = st->decode_mem + DECODE_BUFFER_SIZE - MAX_PERIOD;

  st->oldBandE =
      (float *) celt_alloc(C * mode->nbEBands * sizeof(float));

  st->preemph_memD = (float *) celt_alloc(C * sizeof(float));

  st->lpc = (float *) celt_alloc(C * LPC_ORDER * sizeof(float));

  st->loss_count = 0;

  if ((st->decode_mem != NULL) && (st->out_mem != NULL)
```

```
      && (st->oldBandE != NULL) && (st->lpc != NULL)
      && (st->preemph_memD != NULL))
  {
    if (error)
      *error = CELT_OK;
    st->marker = DECODERVALID;
    return st;
  }
  /* If the setup fails for some reason deallocate it. */
  celt_decoder_destroy(st);
  if (error)
    *error = CELT_ALLOC_FAIL;
  return NULL;
}

void
celt_decoder_destroy(CELTDecoder * st)
{
  if (st == NULL)
  {
    celt_warning("NULL passed to celt_decoder_destroy");
    return;
  }

  if (st->marker == DECODERFREED)
  {
    celt_warning("Freeing a decoder which has already been freed");
    return;
  }

  if (st->marker != DECODERVALID && st->marker != DECODERPARTIAL)
  {
    celt_warning("This is not a valid CELT decoder structure");
    return;
  }

  /* Check_mode is non-fatal here because we can still free the
     encoder memory even if the mode is bad, although calling the
     free functions in this order is a violation of the API. */
  check_mode(st->mode);

  celt_free(st->decode_mem);
  celt_free(st->oldBandE);
  celt_free(st->preemph_memD);
  celt_free(st->lpc);

  st->marker = DECODERFREED;
```

```
  celt_free(st);
}

static void
celt_decode_lost(CELTDecoder * restrict st, float *restrict pcm,
                 int N, int LM)
{
  int             c;
  int             pitch_index;
  int             overlap = st->mode->overlap;
  float           fade = 1.0f;
  int             i,
                  len;
  const int       C = CHANNELS(st->channels);
  int             offset;
  SAVE_STACK;

  len = N + st->mode->overlap;

  if (st->loss_count == 0)
  {
    float           pitch_buf[MAX_PERIOD >> 1];
    float           tmp = 0;
    float           mem0[2] = { 0, 0 };
    float           mem1[2] = { 0, 0 };
    int             len2 = len;
    /* FIXME: This is a kludge */
    if (len2 > MAX_PERIOD >> 1)
      len2 = MAX_PERIOD >> 1;
    pitch_downsample(st->out_mem, pitch_buf, MAX_PERIOD, MAX_PERIOD,
                     C, mem0, mem1);
    pitch_search(st->mode, pitch_buf + ((MAX_PERIOD - len2) >> 1),
                 pitch_buf, len2, MAX_PERIOD - len2 - 100,
                 &pitch_index, &tmp, 1 << LM);
    pitch_index = MAX_PERIOD - len2 - pitch_index;
    st->last_pitch_index = pitch_index;
  } else
  {
    pitch_index = st->last_pitch_index;
    if (st->loss_count < 5)
      fade = (.8f);
    else
      fade = 0;
  }

  for (c = 0; c < C; c++)
  {
    /* FIXME: This is more memory than necessary */
```

```
    float           e[2 * MAX_PERIOD];
    float           exc[2 * MAX_PERIOD];
    float           ac[LPC_ORDER + 1];
    float           decay = 1;
    float           S1 = 0;
    float           mem[LPC_ORDER] = { 0 };

    offset = MAX_PERIOD - pitch_index;
    for (i = 0; i < MAX_PERIOD; i++)
      exc[i] = (st->out_mem[i * C + c]);

    if (st->loss_count == 0)
    {
      _celt_autocorr(exc, ac, st->mode->window, st->mode->overlap,
                     LPC_ORDER, MAX_PERIOD);

      /* Noise floor -40 dB */

      ac[0] *= 1.0001;

      /* Lag windowing */
      for (i = 1; i <= LPC_ORDER; i++)
      {
        /* ac[i] *= exp(-.5*(2*M_PI*.002*i)*(2*M_PI*.002*i)); */

        ac[i] -= ac[i] * (.008 * i) * (.008 * i);

      }

      _celt_lpc(st->lpc + c * LPC_ORDER, ac, LPC_ORDER);
    }
    fir(exc, st->lpc + c * LPC_ORDER, exc, MAX_PERIOD, LPC_ORDER,
        mem);
    /* for (i=0;i<MAX_PERIOD;i++)printf("%d ", exc[i]);
       printf("\n"); */
    /* Check if the waveform is decaying (and if so how fast) */
    {
      float           E1 = 1,
          E2 = 1;
      int             period;
      if (pitch_index <= MAX_PERIOD / 2)
        period = pitch_index;
      else
        period = MAX_PERIOD / 2;
      for (i = 0; i < period; i++)
      {
        E1 +=
            (((exc[MAX_PERIOD - period + i]) *
```

```
            (exc[MAX_PERIOD - period + i])));
      E2 +=
          (((exc[MAX_PERIOD - 2 * period + i]) *
            (exc[MAX_PERIOD - 2 * period + i])));
    }
    if (E1 > E2)
      E1 = E2;
    decay = celt_sqrt(frac_div32((E1), E2));
  }

  /* Copy excitation, taking decay into account */
  for (i = 0; i < len + st->mode->overlap; i++)
  {
    if (offset + i >= MAX_PERIOD)
    {
      offset -= pitch_index;
      decay = ((decay) * (decay));
    }
    e[i] = ((((decay) * (exc[offset + i]))));
    S1 +=
        (((st->out_mem[offset + i]) * (st->out_mem[offset + i])));
  }

  iir(e, st->lpc + c * LPC_ORDER, e, len + st->mode->overlap,
      LPC_ORDER, mem);

  {
    float           S2 = 0;
    for (i = 0; i < len + overlap; i++)
      S2 += (((e[i]) * (e[i])));
    /* This checks for an "explosion" in the synthesis */

    /* Float test is written this way to catch NaNs at the same
       time */
    if (!(S1 > 0.2f * S2))

    {
      for (i = 0; i < len + overlap; i++)
        e[i] = 0;
    } else if (S1 < S2)
    {
      float           ratio =
          celt_sqrt(frac_div32((S1) + 1, S2 + 1.));
      for (i = 0; i < len + overlap; i++)
        e[i] = ((ratio) * (e[i]));
    }
  }
```

```
    for (i = 0; i < MAX_PERIOD + st->mode->overlap - N; i++)
      st->out_mem[C * i + c] = st->out_mem[C * (N + i) + c];

    /* Apply TDAC to the concealed audio so that it blends with the
       previous and next frames */
    for (i = 0; i < overlap / 2; i++)
    {
      float           tmp1,
                      tmp2;
      tmp1 = ((st->mode->window[i]) * (e[i])) -
          ((st->mode->window[overlap - i - 1]) *
           (e[overlap - i - 1]));
      tmp2 =
          ((st->mode->window[i]) * (e[N + overlap - 1 - i])) +
          ((st->mode->window[overlap - i - 1]) * (e[N + i]));
      tmp1 = ((fade) * (tmp1));
      tmp2 = ((fade) * (tmp2));
      st->out_mem[C * (MAX_PERIOD + i) + c] =
          ((st->mode->window[overlap - i - 1]) * (tmp2));
      st->out_mem[C * (MAX_PERIOD + overlap - i - 1) + c] =
          ((st->mode->window[i]) * (tmp2));
      st->out_mem[C * (MAX_PERIOD - N + i) + c] +=
          ((st->mode->window[i]) * (tmp1));
      st->out_mem[C * (MAX_PERIOD - N + overlap - i - 1) + c] -=
          ((st->mode->window[overlap - i - 1]) * (tmp1));
    }
    for (i = 0; i < N - overlap; i++)
      st->out_mem[C * (MAX_PERIOD - N + overlap + i) + c] =
          ((fade) * (e[overlap + i]));
  }

  deemphasis(st->out_mem, pcm, N, C, preemph, st->preemph_memD);

  st->loss_count++;

  RESTORE_STACK;
}

int
celt_decode_with_ec_float(CELTDecoder * restrict st,
                          const unsigned char *data, int len,
                          float *restrict pcm, int frame_size,
                          ec_dec * dec)
{

  int             c,
                  i,
                  N,
```

```
                N4;
  int           has_pitch,
                has_fold;
  int           pitch_index;
  int           bits;
  ec_dec        _dec;
  ec_byte_buffer  buf;
  VARDECL(float, freq);
  VARDECL(float, pitch_freq);
  VARDECL(float, X);
  VARDECL(float, bandE);
  VARDECL(int, fine_quant);
  VARDECL(int, pulses);
  VARDECL(int, offsets);
  VARDECL(int, fine_priority);
  VARDECL(int, tf_res);

  int           shortBlocks;
  int           isTransient;
  int           intra_ener;
  int           transient_time;
  int           transient_shift;
  int           mdct_weight_shift = 0;
  const int     C = CHANNELS(st->channels);
  int           mdct_weight_pos = 0;
  int           gain_id = 0;
  int           LM,
                M;
  int           nbFilledBytes,
                nbAvailableBytes;
  SAVE_STACK;

  if (check_decoder(st) != CELT_OK)
    return CELT_INVALID_STATE;

  if (check_mode(st->mode) != CELT_OK)
    return CELT_INVALID_MODE;

  if (pcm == NULL)
    return CELT_BAD_ARG;

  for (LM = 0; LM < 4; LM++)
    if (st->mode->shortMdctSize << LM == frame_size)
      break;
  if (LM >= MAX_CONFIG_SIZES)
    return CELT_BAD_ARG;
  M = 1 << LM;
```

```
  N = M * st->mode->shortMdctSize;
  N4 = (N - st->overlap) >> 1;

  ALLOC(freq, C * N, float);/**< Interleaved signal MDCTs */
  ALLOC(X, C * N, float);/**< Interleaved normalised MDCTs */
  ALLOC(bandE, st->mode->nbEBands * C, float);
  for (c = 0; c < C; c++)
    for (i = 0; i < M * st->mode->eBands[st->start]; i++)
      X[c * N + i] = 0;

  if (data == NULL)
  {
    celt_decode_lost(st, pcm, N, LM);
    RESTORE_STACK;
    return 0;
  }
  if (len < 0)
  {
    RESTORE_STACK;
    return CELT_BAD_ARG;
  }

  if (dec == NULL)
  {
    ec_byte_readinit(&buf, (unsigned char *) data, len);
    ec_dec_init(&_dec, &buf);
    dec = &_dec;
    nbFilledBytes = 0;
  } else
  {
    nbFilledBytes = (ec_dec_tell(dec, 0) + 4) >> 3;
  }
  nbAvailableBytes = len - nbFilledBytes;

  decode_flags(dec, &intra_ener, &has_pitch, &isTransient,
               &has_fold);
  if (isTransient)
    shortBlocks = M;
  else
    shortBlocks = 0;

  if (isTransient)
  {
    transient_shift = ec_dec_uint(dec, 4);
    if (transient_shift == 3)
    {
      transient_time = ec_dec_uint(dec, N + st->mode->overlap);
    } else
```

```
    {
      mdct_weight_shift = transient_shift;
      if (mdct_weight_shift && M > 2)
        mdct_weight_pos = ec_dec_uint(dec, M - 1);
      transient_shift = 0;
      transient_time = 0;
    }
  } else
  {
    transient_time = -1;
    transient_shift = 0;
  }

  if (has_pitch)
  {
    int             maxpitch = MAX_PERIOD - (2 * N - 2 * N4);
    if (maxpitch < 0)
    {
      celt_notify
          ("detected pitch when not allowed, bit corruption suspected");
      pitch_index = 0;
      has_pitch = 0;
    } else
    {
      pitch_index = ec_dec_uint(dec, maxpitch);
      gain_id = ec_dec_uint(dec, 16);
    }
  } else
  {
    pitch_index = 0;
  }

  ALLOC(fine_quant, st->mode->nbEBands, int);
  /* Get band energies */
  unquant_coarse_energy(st->mode, st->start, bandE, st->oldBandE,
                        nbFilledBytes * 8 + nbAvailableBytes * 4 - 8,
                        intra_ener, st->mode->prob, dec, C);

  ALLOC(tf_res, st->mode->nbEBands, int);
  tf_decode(st->mode->nbEBands, C, isTransient, tf_res,
            nbAvailableBytes, LM, dec);

  ALLOC(pulses, st->mode->nbEBands, int);
  ALLOC(offsets, st->mode->nbEBands, int);
  ALLOC(fine_priority, st->mode->nbEBands, int);

  for (i = 0; i < st->mode->nbEBands; i++)
    offsets[i] = 0;
```

```
  bits = len * 8 - ec_dec_tell(dec, 0) - 1;
  compute_allocation(st->mode, st->start, offsets, bits, pulses,
                     fine_quant, fine_priority, C, M);
  /* bits = ec_dec_tell(dec, 0); compute_fine_allocation(st->mode,
     fine_quant, (20*C+len*8/5-(ec_dec_tell(dec, 0)-bits))/C); */

  unquant_fine_energy(st->mode, st->start, bandE, st->oldBandE,
                      fine_quant, dec, C);

  ALLOC(pitch_freq, C * N, float);/**< Interleaved signal MDCTs */
  if (has_pitch)
  {
    /* Pitch MDCT */
    compute_mdcts(st->mode, 0, st->out_mem + pitch_index * C,
                  pitch_freq, C, LM);
  }

  /* Decode fixed codebook and merge with pitch */
  quant_all_bands(0, st->mode, st->start, X, C == 2 ? X + N : NULL,
                  NULL, pulses, shortBlocks, has_fold, tf_res, 1,
                  len * 8, dec, LM);

  unquant_energy_finalise(st->mode, st->start, bandE, st->oldBandE,
                          fine_quant, fine_priority,
                          len * 8 - ec_dec_tell(dec, 0), dec, C);

  if (mdct_weight_shift)
  {
    mdct_shape(st->mode, X, 0, mdct_weight_pos + 1, N,
               mdct_weight_shift, C, 1, M);
  }

  /* Synthesis */
  denormalise_bands(st->mode, X, freq, bandE, C, M);

  CELT_MOVE(st->decode_mem, st->decode_mem + C * N,
            C * (DECODE_BUFFER_SIZE + st->overlap - N));

  if (has_pitch)
    apply_pitch(st->mode, freq, pitch_freq, gain_id, 0, C, M);

  for (c = 0; c < C; c++)
    for (i = 0; i < M * st->mode->eBands[st->start]; i++)
      freq[c * N + i] = 0;

  /* Compute inverse MDCTs */
  compute_inv_mdcts(st->mode, shortBlocks, freq, transient_time,
                    transient_shift, st->out_mem, C, LM);
```

```
  deemphasis(st->out_mem, pcm, N, C, preemph, st->preemph_memD);
  st->loss_count = 0;
  RESTORE_STACK;
  return 0;
}
int
celt_decode_with_ec(CELTDecoder * restrict st,
                    const unsigned char *data, int len,
                    celt_int16 * restrict pcm, int frame_size,
                    ec_dec * dec)
{
  int             j,
                  ret,
                  C,
                  N,
                  LM,
                  M;
  VARDECL(float, out);
  SAVE_STACK;

  if (check_decoder(st) != CELT_OK)
    return CELT_INVALID_STATE;

  if (check_mode(st->mode) != CELT_OK)
    return CELT_INVALID_MODE;

  if (pcm == NULL)
    return CELT_BAD_ARG;

  for (LM = 0; LM < 4; LM++)
    if (st->mode->shortMdctSize << LM == frame_size)
      break;
  if (LM >= MAX_CONFIG_SIZES)
    return CELT_BAD_ARG;
  M = 1 << LM;

  C = CHANNELS(st->channels);
  N = M * st->mode->shortMdctSize;
  ALLOC(out, C * N, float);

  ret =
      celt_decode_with_ec_float(st, data, len, out, frame_size, dec);

  for (j = 0; j < C * N; j++)
    pcm[j] = FLOAT2INT16(out[j]);

  RESTORE_STACK;
  return ret;
```

```
}

int
celt_decode(CELTDecoder * restrict st, const unsigned char *data,
            int len, celt_int16 * restrict pcm, int frame_size)
{
  return celt_decode_with_ec(st, data, len, pcm, frame_size, NULL);
}

int
celt_decode_float(CELTDecoder * restrict st,
                  const unsigned char *data, int len,
                  float *restrict pcm, int frame_size)
{
  return celt_decode_with_ec_float(st, data, len, pcm, frame_size,
                                   NULL);
}

int
celt_decoder_ctl(CELTDecoder * restrict st, int request, ...)
{
  va_list         ap;

  if (check_decoder(st) != CELT_OK)
    return CELT_INVALID_STATE;

  va_start(ap, request);
  if ((request != CELT_GET_MODE_REQUEST)
      && (check_mode(st->mode) != CELT_OK))
    goto bad_mode;
  switch (request)
  {
  case CELT_GET_MODE_REQUEST:
    {
      const CELTMode **value = va_arg(ap, const CELTMode **);
      if (value == 0)
        goto bad_arg;
      *value = st->mode;
    }
    break;
  case CELT_SET_START_BAND_REQUEST:
    {
      celt_int32      value = va_arg(ap, celt_int32);
      if (value < 0 || value >= st->mode->nbEBands)
        goto bad_arg;
      st->start = value;
    }
    break;
```

```
  case CELT_RESET_STATE:
    {
      const CELTMode *mode = st->mode;
      int             C = st->channels;

      CELT_MEMSET(st->decode_mem, 0,
                  (DECODE_BUFFER_SIZE + st->overlap) * C);
      CELT_MEMSET(st->oldBandE, 0, C * mode->nbEBands);

      CELT_MEMSET(st->preemph_memD, 0, C);

      st->loss_count = 0;

      CELT_MEMSET(st->lpc, 0, C * LPC_ORDER);
    }
    break;
  default:
    goto bad_request;
  }
  va_end(ap);
  return CELT_OK;
bad_mode:
  va_end(ap);
  return CELT_INVALID_MODE;
bad_arg:
  va_end(ap);
  return CELT_BAD_ARG;
bad_request:
  va_end(ap);
  return CELT_UNIMPLEMENTED;
}

const char     *
celt_strerror(int error)
{
  static const char *error_strings[8] = {
    "success",
    "invalid argument",
    "invalid mode",
    "internal error",
    "corrupted stream",
    "request not implemented",
    "invalid state",
    "memory allocation failed"
  };
  if (error > 0 || error < -7)
    return "unknown error";
  else
```

```
    return error_strings[-error];
}
```

A.5. modes.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Copyright (c) 2008 Gregory Maxwell Written by
   Jean-Marc Valin and Gregory Maxwell */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef MODES_H
#define MODES_H

#include "celt_types.h"
#include "celt.h"
#include "arch.h"
#include "mdct.h"
```

```
#include "pitch.h"
#include "entenc.h"
#include "entdec.h"

#define MAX_CONFIG_SIZES 5

#define CELT_BITSTREAM_VERSION 0x8000000c

#ifdef STATIC_MODES
#include "static_modes.h"
#endif

#define MAX_PERIOD 1024

#ifndef MCHANNELS
# ifdef DISABLE_STEREO
#  define MCHANNELS(mode) (1)
# else
#  define MCHANNELS(mode) ((mode)->nbChannels)
# endif
#endif

#ifndef CHANNELS
# ifdef DISABLE_STEREO
#  define CHANNELS(_C) (1)
# else
#  define CHANNELS(_C) (_C)
# endif
#endif

#ifndef OVERLAP
#define OVERLAP(mode) ((mode)->overlap)
#endif

#ifndef FRAMESIZE
#define FRAMESIZE(mode) ((mode)->mdctSize)
#endif

/** Mode definition (opaque)
 @brief Mode definition
 */
struct CELTMode {
  celt_uint32     marker_start;
  celt_int32      Fs;
  int             overlap;

  int             nbEBands;
  int             pitchEnd;
```

```
  const celt_int16 *eBands;  /**< Definition for each "pseudo-critical \
band" */

  float           ePredCoef;
                   /**< Prediction coefficient for the energy encoding \
*/

  int             nbAllocVectors;
                       /**< Number of lines in the matrix below */
  const unsigned char *allocVectors;  /**< Number of bits in each band \
for several rates */

  const celt_int16 *const **bits;
  const celt_int16 *const *(_bits[MAX_CONFIG_SIZES]);   /**< Cache for \
pulses->bits mapping in each band */

  /* Stuff that could go in the {en,de}coder, but we save space this
     way */
  mdct_lookup     mdct;

  const float    *window;

  int             maxLM;
  int             nbShortMdcts;
  int             shortMdctSize;

  int            *prob;
  const celt_int16 *logN;
  celt_uint32     marker_end;
};

int             check_mode(const CELTMode * mode);

/* Prototypes for _ec versions of the encoder/decoder calls (not
   public) */
int             celt_encode_with_ec(CELTEncoder * restrict st,
                                    const celt_int16 * pcm,
                                    celt_int16 *
                                    optional_resynthesis,
                                    int frame_size,
                                    unsigned char *compressed,
                                    int nbCompressedBytes,
                                    ec_enc * enc);
int             celt_encode_with_ec_float(CELTEncoder * restrict st,
                                          const float *pcm,
                                          float
                                          *optional_resynthesis,
                                          int frame_size,
```

```
                                        unsigned char *compressed,
                                        int nbCompressedBytes,
                                        ec_enc * enc);
int             celt_decode_with_ec(CELTDecoder * restrict st,
                                    const unsigned char *data,
                                    int len,
                                    celt_int16 * restrict pcm,
                                    int frame_size, ec_dec * dec);
int             celt_decode_with_ec_float(CELTDecoder * restrict st,
                                        const unsigned char *data,
                                        int len,
                                        float *restrict pcm,
                                        int frame_size,
                                        ec_dec * dec);
#endif
```

A.6. modes.c

```
   /* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
      Foundation Copyright (c) 2008 Gregory Maxwell Written by
      Jean-Marc Valin and Gregory Maxwell */
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.

      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
      CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
      INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
      OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
      PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
```

```
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "celt.h"
#include "modes.h"
#include "rate.h"
#include "os_support.h"
#include "stack_alloc.h"
#include "quant_bands.h"

#define MODEVALID   0xa110ca7e
#define MODEPARTIAL 0x7eca10a1
#define MODEFREED   0xb10cf8ee

#define M_PI 3.141592653

int
celt_mode_info(const CELTMode * mode, int request,
               celt_int32 * value)
{
  if (check_mode(mode) != CELT_OK)
    return CELT_INVALID_MODE;
  switch (request)
  {
  case CELT_GET_LOOKAHEAD:
    *value = mode->overlap;
    break;
  case CELT_GET_BITSTREAM_VERSION:
    *value = CELT_BITSTREAM_VERSION;
    break;
  case CELT_GET_SAMPLE_RATE:
    *value = mode->Fs;
    break;
  default:
    return CELT_UNIMPLEMENTED;
  }
  return CELT_OK;
}

/* Defining 25 critical bands for the full 0-20 kHz audio bandwidth
   Taken from
   http://ccrma.stanford.edu/~jos/bbt/Bark_Frequency_Scale.html */
#define BARK_BANDS 25
```

```
   static const celt_int16 bark_freq[BARK_BANDS + 1] = {
     0, 100, 200, 300, 400,
     510, 630, 770, 920, 1080,
     1270, 1480, 1720, 2000, 2320,
     2700, 3150, 3700, 4400, 5300,
     6400, 7700, 9500, 12000, 15500,
     20000
   };

   /* This allocation table is per critical band. When creating a mode,
      the bits get added together into the codec bands, which are
      sometimes larger than one critical band at low frequency */

   #define BITALLOC_SIZE 12

   static const celt_int16 eband5ms[] = {
     0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, 28, 34, 40, 48,
         60, 78, 100
   };

   static const unsigned char band_allocation[] = {
     /* 0 200 400 600 800 1k 1.2 1.4 1.6 2k 2.4 2.8 3.2 4k 4.8 5.6 6.8
        8k 9.6 12k 15.6 */
     10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     10, 3, 8, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     10, 6, 8, 6, 5, 4, 3, 2, 7, 10, 11, 9, 7, 3, 1, 0, 0, 0, 0, 0, 0,
     10, 10, 14, 11, 10, 8, 6, 5, 10, 12, 13, 11, 8, 4, 2, 1, 0, 0, 0,
         0, 0,
     13, 10, 17, 16, 14, 12, 10, 8, 12, 14, 14, 12, 9, 5, 3, 2, 2, 1, 0,
         0, 0,
     17, 21, 23, 26, 24, 20, 17, 16, 17, 18, 16, 14, 11, 6, 3, 2, 2, 1,
         1, 0, 0,
     21, 21, 36, 32, 28, 24, 23, 23, 22, 18, 18, 14, 11, 7, 5, 5, 5, 3,
         3, 0, 0,
     31, 35, 40, 32, 30, 28, 26, 26, 25, 24, 19, 15, 15, 13, 9, 9, 8, 7,
         5, 2, 0,
     42, 46, 46, 37, 35, 34, 33, 32, 34, 35, 32, 31, 27, 24, 23, 23, 18,
         14, 11, 7, 0,
     46, 49, 46, 46, 42, 43, 44, 47, 50, 52, 51, 48, 39, 32, 27, 24, 22,
         19, 17, 11, 5,
     53, 53, 49, 48, 55, 66, 71, 71, 71, 65, 64, 64, 56, 47, 41, 37, 31,
         24, 20, 16, 10,
     60, 64, 74, 74, 87, 103, 106, 102, 101, 100, 101, 95, 80, 69, 63,
         55, 47, 36, 26, 21, 15,
   };

   static celt_int16 *
   compute_ebands(celt_int32 Fs, int frame_size, int res, int *nbEBands)
```

```
   {
     celt_int16     *eBands;
     int             i,
                     lin,
                     low,
                     high,
                     nBark,
                     offset = 0;

     if (Fs == 400 * (celt_int32) frame_size && Fs >= 40000)
     {
       *nbEBands = sizeof(eband5ms) / sizeof(eband5ms[0]) - 1;
       eBands = celt_alloc(sizeof(celt_int16) * (*nbEBands + 2));
       for (i = 0; i < *nbEBands + 2; i++)
         eBands[i] = eband5ms[i];
       eBands[*nbEBands + 1] = frame_size;
       return eBands;
     }
     /* Find the number of critical bands supported by our sampling
        rate */
     for (nBark = 1; nBark < BARK_BANDS; nBark++)
       if (bark_freq[nBark + 1] * 2 >= Fs)
         break;

     /* Find where the linear part ends (i.e. where the spacing is more
        than min_width */
     for (lin = 0; lin < nBark; lin++)
       if (bark_freq[lin + 1] - bark_freq[lin] >= res)
         break;

     low = (bark_freq[lin] + res / 2) / res;
     high = nBark - lin;
     *nbEBands = low + high;
     eBands = celt_alloc(sizeof(celt_int16) * (*nbEBands + 2));

     if (eBands == NULL)
       return NULL;

     /* Linear spacing (min_width) */
     for (i = 0; i < low; i++)
       eBands[i] = i;
     if (low > 0)
       offset = eBands[low - 1] * res - bark_freq[lin - 1];
     /* Spacing follows critical bands */
     for (i = 0; i < high; i++)
     {
       int             target = bark_freq[lin + i];
       eBands[i + low] = (target + (offset + res) / 2) / res;
```

```
      offset = eBands[i + low] * res - target;
    }
    /* Enforce the minimum spacing at the boundary */
    for (i = 0; i < *nbEBands; i++)
      if (eBands[i] < i)
        eBands[i] = i;
    eBands[*nbEBands] = (bark_freq[nBark] + res / 2) / res;
    eBands[*nbEBands + 1] = frame_size;
    if (eBands[*nbEBands] > eBands[*nbEBands + 1])
      eBands[*nbEBands] = eBands[*nbEBands + 1];
    for (i = 1; i < *nbEBands - 1; i++)
    {
      if (eBands[i + 1] - eBands[i] < eBands[i] - eBands[i - 1])
      {
        eBands[i] -=
            (2 * eBands[i] - eBands[i - 1] - eBands[i + 1]) / 2;
      }
    }
    /* for (i=0;i<= *nbEBands+1;i++) printf ("%d ", eBands[i]); printf
       ("\n"); exit(1); */
    /* FIXME: Remove last band if too small */
    return eBands;
  }

  static void
  compute_allocation_table(CELTMode * mode, int res)
  {
    int             i,
                    j;
    unsigned char  *allocVectors;
    int             maxBands =
        sizeof(eband5ms) / sizeof(eband5ms[0]) - 1;

    mode->nbAllocVectors = BITALLOC_SIZE;
    allocVectors =
        celt_alloc(sizeof(unsigned char) *
                   (BITALLOC_SIZE * mode->nbEBands));
    if (allocVectors == NULL)
      return;

    /* Check for standard mode */
    if (mode->Fs == 400 * (celt_int32) mode->shortMdctSize
        && mode->Fs >= 40000)
    {
      for (i = 0; i < BITALLOC_SIZE * mode->nbEBands; i++)
        allocVectors[i] = band_allocation[i];
      mode->allocVectors = allocVectors;
      return;
```

```
    }
    /* If not the standard mode, interpolate */

    /* Compute per-codec-band allocation from per-critical-band matrix
     */
    for (i = 0; i < BITALLOC_SIZE; i++)
    {
      celt_int32      current = 0;
      int             eband = 0;
      /* We may be looping over too many bands, but eband will stop
         being incremented once we reach the last band */
      for (j = 0; j < maxBands; j++)
      {
        int             edge,
                        low,
                        high;
        celt_int32      alloc;
        alloc =
            band_allocation[i * maxBands +
                            j] * (mode->eBands[eband + 1] -
                                  mode->eBands[eband]) << 4;
        low = eband5ms[j] * 200;
        high = eband5ms[j + 1] * 200;
        edge = mode->eBands[eband + 1] * res;
        while (edge <= high && eband < mode->nbEBands)
        {
          celt_int32      num;
          int             den,
                          bits;
          int             N =
              (mode->eBands[eband + 1] - mode->eBands[eband]);
          num = alloc * (edge - low);
          den = high - low;
          /* Divide with rounding */
          bits = (2 * num + den) / (2 * den);
          allocVectors[i * mode->nbEBands + eband] =
              (2 * (current + bits) + (N << 4)) / (2 * N << 4);
          /* Remove the part of the band we just allocated */
          low = edge;
          alloc -= bits;

          /* Move to next eband */
          current = 0;
          eband++;
          edge = mode->eBands[eband + 1] * res;
        }
        current += alloc;
      }
```

```
      if (eband < mode->nbEBands)
      {
        int             N =
            (mode->eBands[eband + 1] - mode->eBands[eband]);
        allocVectors[i * mode->nbEBands + eband] =
            (2 * current + (N << 4)) / (2 * N << 4);
      }
    }
    /* printf ("\n"); for (i=0;i<BITALLOC_SIZE;i++) { for
       (j=0;j<mode->nbEBands;j++) printf ("%d ",
       allocVectors[i*mode->nbEBands+j]); printf ("\n"); } exit(0); */

    mode->allocVectors = allocVectors;
  }

  CELTMode       *
  celt_mode_create(celt_int32 Fs, int frame_size, int *error)
  {
    int             i;
    int             LM;
    int             res;
    CELTMode       *mode = NULL;
    float          *window;
    celt_int16     *logN;
    ALLOC_STACK;

    if (global_stack == NULL)
      goto failure;

    /* The good thing here is that permutation of the arguments will
       automatically be invalid */

    if (Fs < 32000 || Fs > 96000)
    {
      celt_warning("Sampling rate must be between 32 kHz and 96 kHz");
      if (error)
        *error = CELT_BAD_ARG;
      return NULL;
    }
    if (frame_size < 64 || frame_size > 1024 || frame_size % 2 != 0)
    {
      celt_warning
          ("Only even frame sizes from 64 to 1024 are supported");
      if (error)
        *error = CELT_BAD_ARG;
      return NULL;
    }
```

```
    mode = celt_alloc(sizeof(CELTMode));
    if (mode == NULL)
      goto failure;
    mode->marker_start = MODEPARTIAL;
    mode->Fs = Fs;
    mode->ePredCoef = (.8f);

    if (frame_size >= 640 && (frame_size % 16) == 0)
    {
      LM = 3;
    } else if (frame_size >= 320 && (frame_size % 8) == 0)
    {
      LM = 2;
    } else if (frame_size >= 160 && (frame_size % 4) == 0)
    {
      LM = 1;
    } else
    {
      LM = 0;
    }

    mode->maxLM = LM;
    mode->nbShortMdcts = 1 << LM;
    mode->shortMdctSize = frame_size / mode->nbShortMdcts;
    res = (mode->Fs + mode->shortMdctSize) / (2 * mode->shortMdctSize);

    mode->eBands =
        compute_ebands(Fs, mode->shortMdctSize, res, &mode->nbEBands);
    if (mode->eBands == NULL)
      goto failure;

    mode->pitchEnd = 4000 * (celt_int32) mode->shortMdctSize / Fs;

    /* Overlap must be divisible by 4 */
    if (mode->nbShortMdcts > 1)
      mode->overlap = (mode->shortMdctSize >> 2) << 2;
    else
      mode->overlap = (frame_size >> 3) << 2;

    compute_allocation_table(mode, res);
    if (mode->allocVectors == NULL)
      goto failure;

    window = (float *) celt_alloc(mode->overlap * sizeof(float));
    if (window == NULL)
      goto failure;

    for (i = 0; i < mode->overlap; i++)
```

```
      window[i] =
          1.0f * sin(.5 * M_PI *
                     sin(.5 * M_PI * (i + .5) / mode->overlap) *
                     sin(.5 * M_PI * (i + .5) / mode->overlap));

    mode->window = window;

    mode->bits = mode->_bits + 1;
    for (i = 0; (1 << i) <= mode->nbShortMdcts; i++)
    {
      mode->bits[i] =
          (const celt_int16 **) compute_alloc_cache(mode, 1 << i);
      if (mode->bits[i] == NULL)
        goto failure;
    }
    mode->bits[-1] =
        (const celt_int16 **) compute_alloc_cache(mode, 0);
    if (mode->bits[-1] == NULL)
      goto failure;

    logN =
        (celt_int16 *) celt_alloc(mode->nbEBands * sizeof(celt_int16));
    if (logN == NULL)
      goto failure;

    for (i = 0; i < mode->nbEBands; i++)
      logN[i] =
          log2_frac(mode->eBands[i + 1] - mode->eBands[i], BITRES);
    mode->logN = logN;

    clt_mdct_init(&mode->mdct,
                  2 * mode->shortMdctSize * mode->nbShortMdcts, LM);
    if ((mode->mdct.trig == NULL) || (mode->mdct.kfft == NULL))
      goto failure;

    mode->prob = quant_prob_alloc(mode);
    if (mode->prob == NULL)
      goto failure;

    mode->marker_start = MODEVALID;
    mode->marker_end = MODEVALID;
    if (error)
      *error = CELT_OK;
    return mode;
  failure:
    if (error)
      *error = CELT_INVALID_MODE;
    if (mode != NULL)
```

```
      celt_mode_destroy(mode);
    return NULL;
  }

  void
  celt_mode_destroy(CELTMode * mode)
  {
    int             i,
                    m;
    const celt_int16 *prevPtr = NULL;
    if (mode == NULL)
    {
      celt_warning("NULL passed to celt_mode_destroy");
      return;
    }

    if (mode->marker_start == MODEFREED
        || mode->marker_end == MODEFREED)
    {
      celt_warning("Freeing a mode which has already been freed");
      return;
    }

    if (mode->marker_start != MODEVALID
        && mode->marker_start != MODEPARTIAL)
    {
      celt_warning("This is not a valid CELT mode structure");
      return;
    }
    mode->marker_start = MODEFREED;

    for (m = 0; (1 << m) <= mode->nbShortMdcts; m++)
    {
      if (mode->bits[m] != NULL)
      {
        for (i = 0; i < mode->nbEBands; i++)
        {
          if (mode->bits[m][i] != prevPtr)
          {
            prevPtr = mode->bits[m][i];
            celt_free((int *) mode->bits[m][i]);
          }
        }
      }
      celt_free((celt_int16 **) mode->bits[m]);
    }
    if (mode->bits[-1] != NULL)
    {
```

```
      for (i = 0; i < mode->nbEBands; i++)
      {
        if (mode->bits[-1][i] != prevPtr)
        {
          prevPtr = mode->bits[-1][i];
          celt_free((int *) mode->bits[-1][i]);
        }
      }
    }
    celt_free((celt_int16 **) mode->bits[-1]);

    celt_free((celt_int16 *) mode->eBands);
    celt_free((celt_int16 *) mode->allocVectors);

    celt_free((float *) mode->window);
    celt_free((celt_int16 *) mode->logN);

    clt_mdct_clear(&mode->mdct);

    quant_prob_free(mode->prob);
    mode->marker_end = MODEFREED;
    celt_free((CELTMode *) mode);
  }

  int
  check_mode(const CELTMode * mode)
  {
    if (mode == NULL)
      return CELT_INVALID_MODE;
    if (mode->marker_start == MODEVALID
        && mode->marker_end == MODEVALID)
      return CELT_OK;
    if (mode->marker_start == MODEFREED
        || mode->marker_end == MODEFREED)
      celt_warning("Using a mode that has already been freed");
    else
      celt_warning("This is not a valid CELT mode");
    return CELT_INVALID_MODE;
  }
```

A.7. bands.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Copyright (c) 2008-2009 Gregory Maxwell Written by
   Jean-Marc Valin and Gregory Maxwell */
/*
   Redistribution and use in source and binary forms, with or
```

```
#ifndef BANDS_H
#define BANDS_H

#include "arch.h"
#include "modes.h"
#include "entenc.h"
#include "entdec.h"
#include "rate.h"

/** Compute the amplitude (sqrt energy) in each of the bands
 * @param m Mode data
 * @param X Spectrum
 * @param bands Square root of the energy for each band (returned)
 */
void            compute_band_energies(const CELTMode * m,
                                      const float *X, float *bands,
                                      int _C, int M);
```

```
/* void compute_noise_energies(const CELTMode *m, const celt_sig *X,
   const celt_word16 *tonality, celt_ener *bank); */

/** Normalise each band of X such that the energy in each band is
    equal to 1
 * @param m Mode data
 * @param X Spectrum (returned normalised)
 * @param bands Square root of the energy for each band
 */
void            normalise_bands(const CELTMode * m,
                                const float *restrict freq,
                                float *restrict X,
                                const float *bands, int _C, int M);

void            renormalise_bands(const CELTMode * m,
                                  float *restrict X, int _C, int M);

/** Denormalise each band of X to restore full amplitude
 * @param m Mode data
 * @param X Spectrum (returned de-normalised)
 * @param bands Square root of the energy for each band
 */
void            denormalise_bands(const CELTMode * m,
                                  const float *restrict X,
                                  float *restrict freq,
                                  const float *bands, int _C, int M);

/** Compute the pitch predictor gain for each pitch band
 * @param m Mode data
 * @param X Spectrum to predict
 * @param P Pitch vector (normalised)
 * @param gains Gain computed for each pitch band (returned)
 * @param bank Square root of the energy for each band
 */
int             compute_pitch_gain(const CELTMode * m,
                                   const float *X, const float *P,
                                   int norm_rate, int *gain_id,
                                   int _C, float *gain_prod, int M);

void            apply_pitch(const CELTMode * m, float *X,
                            const float *P, int gain_id, int pred,
                            int _C, int M);

int             folding_decision(const CELTMode * m, float *X,
                                 float *average, int *last_decision,
                                 int _C, int M);

/** Quantisation/encoding of the residual spectrum
```

```
 * @param m Mode data
 * @param X Residual (normalised)
 * @param total_bits Total number of bits that can be used for the fram\
e (including the ones already spent)
 * @param enc Entropy encoder
 */
void            quant_all_bands(int encode, const CELTMode * m,
                                int start, float *X, float *Y,
                                const float *bandE, int *pulses,
                                int time_domain, int fold,
                                int *tf_res, int resynth,
                                int total_bits, void *enc, int M);

void            stereo_decision(const CELTMode * m,
                                float *restrict X, int *stereo_mode,
                                int len, int M);

#endif                          /* BANDS_H */
```

A.8. bands.c

```
  /* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
     Foundation Copyright (c) 2008-2009 Gregory Maxwell Written by
     Jean-Marc Valin and Gregory Maxwell */
  /*
     Redistribution and use in source and binary forms, with or
     without modification, are permitted provided that the following
     conditions are met:

     - Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

     - Redistributions in binary form must reproduce the above
     copyright notice, this list of conditions and the following
     disclaimer in the documentation and/or other materials provided
     with the distribution.

     - Neither the name of the Xiph.org Foundation nor the names of
     its contributors may be used to endorse or promote products
     derived from this software without specific prior written
     permission.

     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
     CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
     INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
     MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
```

```
    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
    OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
    PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
    OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
    THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
    TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
    OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
    OF SUCH DAMAGE. */

#include "config.h"

#include <math.h>
#include "bands.h"
#include "modes.h"
#include "vq.h"
#include "cwrs.h"
#include "stack_alloc.h"
#include "os_support.h"
#include "mathops.h"
#include "rate.h"
/* Compute the amplitude (sqrt energy) in each of the bands */
void
compute_band_energies(const CELTMode * m, const float *X,
                      float *bank, int _C, int M)
{
  int             i,
                  c,
                  N;
  const celt_int16 *eBands = m->eBands;
  const int       C = CHANNELS(_C);
  N = M * m->eBands[m->nbEBands + 1];
  for (c = 0; c < C; c++)
  {
    for (i = 0; i < m->nbEBands; i++)
    {
      int             j;
      float           sum = 1e-10;
      for (j = M * eBands[i]; j < M * eBands[i + 1]; j++)
        sum += X[j + c * N] * X[j + c * N];
      bank[i + c * m->nbEBands] = sqrt(sum);
      /* printf ("%f ", bank[i+c*m->nbEBands]); */
    }
  }
  /* printf ("\n"); */
}

/* Normalise each band such that the energy is one. */
void
```

```
  normalise_bands(const CELTMode * m, const float *restrict freq,
                  float *restrict X, const float *bank, int _C, int M)
  {
    int             i,
                    c,
                    N;
    const celt_int16 *eBands = m->eBands;
    const int       C = CHANNELS(_C);
    N = M * m->eBands[m->nbEBands + 1];
    for (c = 0; c < C; c++)
    {
      for (i = 0; i < m->nbEBands; i++)
      {
        int             j;
        float           g = 1.f / (1e-10f + bank[i + c * m->nbEBands]);
        for (j = M * eBands[i]; j < M * eBands[i + 1]; j++)
          X[j + c * N] = freq[j + c * N] * g;
      }
    }
  }

  void
  renormalise_bands(const CELTMode * m, float *restrict X, int _C,
                    int M)
  {
    int             i,
                    c;
    const celt_int16 *eBands = m->eBands;
    const int       C = CHANNELS(_C);
    for (c = 0; c < C; c++)
    {
      i = 0;
      do
      {
        renormalise_vector(X + M * eBands[i] +
                           c * M * eBands[m->nbEBands + 1], 1.0f,
                           M * eBands[i + 1] - M * eBands[i], 1);
      }
      while (++i < m->nbEBands);
    }
  }

  /* De-normalise the energy to produce the synthesis from the
     unit-energy bands */
  void
  denormalise_bands(const CELTMode * m, const float *restrict X,
                    float *restrict freq, const float *bank, int _C,
                    int M)
```

```
  {
    int             i,
                    c,
                    N;
    const celt_int16 *eBands = m->eBands;
    const int       C = CHANNELS(_C);
    N = M * m->eBands[m->nbEBands + 1];
    if (C > 2)
      celt_fatal
          ("denormalise_bands() not implemented for >2 channels");
    for (c = 0; c < C; c++)
    {
      float          *restrict f;
      const float    *restrict x;
      f = freq + c * N;
      x = X + c * N;
      for (i = 0; i < m->nbEBands; i++)
      {
        int             j,
                        end;
        float           g = (bank[i + c * m->nbEBands]);
        j = M * eBands[i];
        end = M * eBands[i + 1];
        do
        {
          *f++ = (((*x) * (g)));
          x++;
        }
        while (++j < end);
      }
      for (i = M * eBands[m->nbEBands];
           i < M * eBands[m->nbEBands + 1]; i++)
        *f++ = 0;
    }
  }

  int
  compute_pitch_gain(const CELTMode * m, const float *X,
                     const float *P, int norm_rate, int *gain_id,
                     int _C, float *gain_prod, int M)
  {
    int             j,
                    c;
    float           g;
    float           delta;
    const int       C = CHANNELS(_C);
    float           Sxy = 0,
        Sxx = 0,
```

```
      Syy = 0;
    int             len = M * m->pitchEnd;
    int             N = M * m->eBands[m->nbEBands + 1];
    delta = ((1.0f) / (len));
    for (c = 0; c < C; c++)
    {
      float           gg = 1.0f;
      for (j = 0; j < len; j++)
      {
        float           Xj,
                        Pj;
        Xj = ((X[j + c * N]));
        Pj = ((gg) * (((P[j + c * N]))));
        Sxy = ((Sxy) + (Xj) * (Pj));
        Sxx = ((Sxx) + (Pj) * (Pj));
        Syy = ((Syy) + (Xj) * (Xj));
        gg = ((gg) - (delta));
      }
    }
    {
      float           fact = .04f * norm_rate;
      if (fact < 1)
        fact = 1;
      g = Sxy / (.1f + Sxx + .03f * Syy);
      if (Sxy < .5f * fact * celt_sqrt(1 + Sxx * Syy))
        g = 0;
      /* This MUST round down so that we don't over-estimate the gain */
      *gain_id = floor(20 * (g - .5f));
    }

    /* This prevents the pitch gain from being above 1.0 for too long
       by bounding the maximum error amplification factor to 2.0 */
    g = (((.5f)) + ((((.05f)) * (*gain_id))));
    *gain_prod = MAX16((1.f), ((*gain_prod) * (g)));
    if (*gain_prod > (2.f))
    {
      *gain_id = 9;
      *gain_prod = (2.f);
    }

    if (*gain_id < 0)
    {
      *gain_id = 0;
      return 0;
    } else
    {
      if (*gain_id > 15)
        *gain_id = 15;
```

```
    return 1;
  }
}

void
apply_pitch(const CELTMode * m, float *X, const float *P,
            int gain_id, int pred, int _C, int M)
{
  int             j,
                  c,
                  N;
  float           gain;
  float           delta;
  const int       C = CHANNELS(_C);
  int             len = M * m->pitchEnd;

  N = M * m->eBands[m->nbEBands + 1];
  gain = (((.5f)) + ((((.05f)) * (gain_id))));
  delta = ((gain) / (len));
  if (pred)
    gain = -gain;
  else
    delta = -delta;
  for (c = 0; c < C; c++)
  {
    float           gg = gain;
    for (j = 0; j < len; j++)
    {
      X[j + c * N] += (((gg) * (P[j + c * N])));
      gg = ((gg) + (delta));
    }
  }
}

static void
stereo_band_mix(const CELTMode * m, float *X, float *Y,
                const float *bank, int stereo_mode, int bandID,
                int dir, int N)
{
  int             i = bandID;
  int             j;
  float           a1,
                  a2;
  if (stereo_mode == 0)
  {
    /* Do mid-side when not doing intensity stereo */
    a1 = (.70711f);
    a2 = dir * (.70711f);
```

```
    } else
    {
      float           left,
                      right;
      float           norm;

      left = (bank[i]);
      right = (bank[i + m->nbEBands]);
      norm =
          1e-15f + celt_sqrt(1e-15f + ((left) * (left)) +
                             ((right) * (right)));
      a1 = ((((left))) / (norm));
      a2 = dir * ((((right))) / (norm));
    }
    for (j = 0; j < N; j++)
    {
      float           r,
                      l;
      l = X[j];
      r = Y[j];
      X[j] = ((a1) * (l)) + ((a2) * (r));
      Y[j] = ((a1) * (r)) - ((a2) * (l));
    }
  }

  int
  folding_decision(const CELTMode * m, float *X, float *average,
                   int *last_decision, int _C, int M)
  {
    int             i,
                    c,
                    N0;
    int             NR = 0;
    float           ratio = 1e-15f;
    const int       C = CHANNELS(_C);
    const celt_int16 *restrict eBands = m->eBands;

    N0 = M * m->eBands[m->nbEBands + 1];

    for (c = 0; c < C; c++)
    {
      for (i = 0; i < m->nbEBands; i++)
      {
        int             j,
                        N;
        int             max_i = 0;
        float           max_val = 1e-15f;
        float           floor_ener = 1e-15f;
```

```
       float          *restrict x = X + M * eBands[i] + c * N0;
       N = M * eBands[i + 1] - M * eBands[i];
       for (j = 0; j < N; j++)
       {
         if (ABS16(x[j]) > max_val)
         {
           max_val = ABS16(x[j]);
           max_i = j;
         }
       }

       floor_ener = (1.) - ((max_val) * (max_val));
       if (max_i < N - 1)
         floor_ener -= ((x[(max_i + 1)]) * (x[(max_i + 1)]));
       if (max_i < N - 2)
         floor_ener -= ((x[(max_i + 2)]) * (x[(max_i + 2)]));
       if (max_i > 0)
         floor_ener -= ((x[(max_i - 1)]) * (x[(max_i - 1)]));
       if (max_i > 1)
         floor_ener -= ((x[(max_i - 2)]) * (x[(max_i - 2)]));
       floor_ener = MAX32(floor_ener, 1e-15f);

       if (N > 7)
       {
         float           r;
         float           den = celt_sqrt(floor_ener);
         den = MAX32((.02f), den);
         r = ((((max_val))) / (den));
         ratio = ((ratio) + ((r)));
         NR++;
       }
     }
   }
   if (NR > 0)
     ratio = ((ratio) / (NR));
   ratio = (((.5f * (ratio))) + ((.5f * (*average))));
   if (!*last_decision)
   {
     *last_decision = (ratio < (1.8f));
   } else
   {
     *last_decision = (ratio < (3.f));
   }
   *average = (ratio);
   return *last_decision;
 }

 static void
```

```
  interleave_vector(float *X, int N0, int stride)
  {
    int             i,
                    j;
    VARDECL(float, tmp);
    int             N;
    SAVE_STACK;
    N = N0 * stride;
    ALLOC(tmp, N, float);
    for (i = 0; i < stride; i++)
      for (j = 0; j < N0; j++)
        tmp[j * stride + i] = X[i * N0 + j];
    for (j = 0; j < N; j++)
      X[j] = tmp[j];
    RESTORE_STACK;
  }

  static void
  deinterleave_vector(float *X, int N0, int stride)
  {
    int             i,
                    j;
    VARDECL(float, tmp);
    int             N;
    SAVE_STACK;
    N = N0 * stride;
    ALLOC(tmp, N, float);
    for (i = 0; i < stride; i++)
      for (j = 0; j < N0; j++)
        tmp[i * N0 + j] = X[j * stride + i];
    for (j = 0; j < N; j++)
      X[j] = tmp[j];
    RESTORE_STACK;
  }

  static void
  haar1(float *X, int N0, int stride)
  {
    int             i,
                    j;
    N0 >>= 1;
    for (i = 0; i < stride; i++)
      for (j = 0; j < N0; j++)
      {
        float           tmp = X[stride * 2 * j + i];
        X[stride * 2 * j + i] =
            (((.7070678f)) *
             (X[stride * 2 * j + i] + X[stride * (2 * j + 1) + i]));
```

```
      X[stride * (2 * j + 1) + i] =
          (((.7070678f)) * (tmp - X[stride * (2 * j + 1) + i]));
    }
  }

  /* This function is responsible for encoding and decoding a band for
     both the mono and stereo case. Even in the mono case, it can
     split the band in two and transmit the energy difference with the
     two half-bands. It can be called recursively so bands can end up
     being split in 8 parts. */
  static void
  quant_band(int encode, const CELTMode * m, int i, float *X, float *Y,
             int N, int b, int spread, int tf_change, float *lowband,
             int resynth, void *ec, celt_int32 * remaining_bits,
             int LM, float *lowband_out, const float *bandE, int level)
  {
    int             q;
    int             curr_bits;
    int             stereo,
                    split;
    int             imid = 0,
        iside = 0;
    int             N0 = N;
    int             N_B = N;
    int             N_B0;
    int             spread0 = spread;
    int             time_divide = 0;
    int             recombine = 0;

    if (spread)
      N_B /= spread;
    N_B0 = N_B;

    split = stereo = Y != NULL;

    /* Special case for one sample */
    if (N == 1)
    {
      int             c;
      float          *x = X;
      for (c = 0; c < 1 + stereo; c++)
      {
        int             sign = 0;
        if (b >= 1 << BITRES && *remaining_bits >= 1 << BITRES)
        {
          if (encode)
          {
            sign = x[0] < 0;
```

```
            ec_enc_bits((ec_enc *) ec, sign, 1);
          } else
          {
            sign = ec_dec_bits((ec_dec *) ec, 1);
          }
          *remaining_bits -= 1 << BITRES;
          b -= 1 << BITRES;
        }
        if (resynth)
          x[0] = sign ? -1.f : 1.f;
        x = Y;
      }
      if (lowband_out)
        lowband_out[0] = X[0];
      return;
    }

    /* Band recombining to increase frequency resolution */
    if (!stereo && spread > 1 && level == 0 && tf_change > 0)
    {
      while (spread > 1 && tf_change > 0)
      {
        spread >>= 1;
        N_B <<= 1;
        if (encode)
          haar1(X, N_B, spread);
        if (lowband)
          haar1(lowband, N_B, spread);
        recombine++;
        tf_change--;
      }
      spread0 = spread;
      N_B0 = N_B;
    }

    /* Increasing the time resolution */
    if (!stereo && level == 0)
    {
      while ((N_B & 1) == 0 && tf_change < 0 && spread <= (1 << LM))
      {
        if (encode)
          haar1(X, N_B, spread);
        if (lowband)
          haar1(lowband, N_B, spread);
        spread <<= 1;
        N_B >>= 1;
        time_divide++;
        tf_change++;
```

```
      }
      spread0 = spread;
      N_B0 = N_B;
    }

    /* Reorganize the samples in time order instead of frequency order
     */
    if (!stereo && spread0 > 1 && level == 0)
    {
      if (encode)
        deinterleave_vector(X, N_B, spread0);
      if (lowband)
        deinterleave_vector(lowband, N_B, spread0);
    }

    /* If we need more than 32 bits, try splitting the band in two. */
    if (!stereo && LM != -1 && b > 32 << BITRES && N > 2)
    {
      if (LM > 0 || (N & 1) == 0)
      {
        N >>= 1;
        Y = X + N;
        split = 1;
        LM -= 1;
        spread = (spread + 1) >> 1;
      }
    }

    if (split)
    {
      int             qb;
      int             itheta = 0;
      int             mbits,
                      sbits,
                      delta;
      int             qalloc;
      float           mid,
                      side;
      int             offset,
                      N2;
      offset = m->logN[i] + (LM << BITRES) - QTHETA_OFFSET;

      /* Decide on the resolution to give to the split parameter theta
       */
      N2 = 2 * N - 1;
      if (stereo && N > 2)
        N2--;
      qb = (b + N2 * offset) / (N2 << BITRES);
```

```
      if (qb > (b >> (BITRES + 1)) - 1)
        qb = (b >> (BITRES + 1)) - 1;

      if (qb < 0)
        qb = 0;
      if (qb > 14)
        qb = 14;

      qalloc = 0;
      if (qb != 0)
      {
        int             shift;
        shift = 14 - qb;

        if (encode)
        {
          if (stereo)
            stereo_band_mix(m, X, Y, bandE, qb == 0, i, 1, N);

          mid = renormalise_vector(X, 1.0f, N, 1);
          side = renormalise_vector(Y, 1.0f, N, 1);

          /* theta is the atan() of the ration between the
             (normalized) side and mid. With just that parameter, we
             can re-scale both mid and side because we know that 1)
             they have unit norm and 2) they are orthogonal. */

          itheta = floor(.5f + 16384 * 0.63662f * atan2(side, mid));

          itheta = (itheta + (1 << shift >> 1)) >> shift;
        }

        /* Entropy coding of the angle. We use a uniform pdf for the
           first stereo split but a triangular one for the rest. */
        if (stereo || qb > 9 || spread > 1)
        {
          if (encode)
            ec_enc_uint((ec_enc *) ec, itheta, (1 << qb) + 1);
          else
            itheta = ec_dec_uint((ec_dec *) ec, (1 << qb) + 1);
          qalloc = log2_frac((1 << qb) + 1, BITRES);
        } else
        {
          int             fs = 1,
              ft;
          ft = ((1 << qb >> 1) + 1) * ((1 << qb >> 1) + 1);
          if (encode)
          {
```

```
            int             j;
            int             fl = 0;
            j = 0;
            while (1)
            {
              if (j == itheta)
                break;
              fl += fs;
              if (j < (1 << qb >> 1))
                fs++;
              else
                fs--;
              j++;
            }
            ec_encode((ec_enc *) ec, fl, fl + fs, ft);
          } else
          {
            int             fl = 0;
            int             j,
                            fm;
            fm = ec_decode((ec_dec *) ec, ft);
            j = 0;
            while (1)
            {
              if (fm < fl + fs)
                break;
              fl += fs;
              if (j < (1 << qb >> 1))
                fs++;
              else
                fs--;
              j++;
            }
            itheta = j;
            ec_dec_update((ec_dec *) ec, fl, fl + fs, ft);
          }
          qalloc = log2_frac(ft, BITRES) - log2_frac(fs, BITRES) + 1;
        }
        itheta <<= shift;
      }

      if (itheta == 0)
      {
        imid = 32767;
        iside = 0;
        delta = -10000;
      } else if (itheta == 16384)
      {
```

```
        imid = 0;
        iside = 32767;
        delta = 10000;
      } else
      {
        imid = bitexact_cos(itheta);
        iside = bitexact_cos(16384 - itheta);
        /* This is the mid vs side allocation that minimizes squared
           error in that band. */
        delta =
            (N - 1) * (log2_frac(iside, BITRES + 2) -
                       log2_frac(imid, BITRES + 2)) >> 2;
      }

      /* This is a special case for N=2 that only works for stereo and
         takes advantage of the fact that mid and side are orthogonal
         to encode the side with just one bit. */
      if (N == 2 && stereo)
      {
        int             c,
                        c2;
        int             sign = 1;
        float           v[2],
                        w[2];
        float          *x2,
                       *y2;
        mbits = b - qalloc;
        sbits = 0;
        if (itheta != 0 && itheta != 16384)
          sbits = 1 << BITRES;
        mbits -= sbits;
        c = itheta > 8192 ? 1 : 0;
        *remaining_bits -= qalloc + sbits;

        x2 = X;
        y2 = Y;
        if (encode)
        {
          c2 = 1 - c;

          if (c == 0)
          {
            v[0] = x2[0];
            v[1] = x2[1];
            w[0] = y2[0];
            w[1] = y2[1];
          } else
          {
```

```
            v[0] = y2[0];
            v[1] = y2[1];
            w[0] = x2[0];
            w[1] = x2[1];
          }
          /* Here we only need to encode a sign for the side */
          if (v[0] * w[1] - v[1] * w[0] > 0)
            sign = 1;
          else
            sign = -1;
        }
        quant_band(encode, m, i, v, NULL, N, mbits, spread, tf_change,
                   lowband, resynth, ec, remaining_bits, LM,
                   lowband_out, NULL, level + 1);
        if (sbits)
        {
          if (encode)
          {
            ec_enc_bits((ec_enc *) ec, sign == 1, 1);
          } else
          {
            sign = 2 * ec_dec_bits((ec_dec *) ec, 1) - 1;
          }
        } else
        {
          sign = 1;
        }
        w[0] = -sign * v[1];
        w[1] = sign * v[0];
        if (c == 0)
        {
          x2[0] = v[0];
          x2[1] = v[1];
          y2[0] = w[0];
          y2[1] = w[1];
        } else
        {
          x2[0] = w[0];
          x2[1] = w[1];
          y2[0] = v[0];
          y2[1] = v[1];
        }
      } else
      {
        /* "Normal" split code */
        float          *next_lowband2 = NULL;
        float          *next_lowband_out1 = NULL;
        int             next_level = 0;
```

```
        /* Give more bits to low-energy MDCTs than they would
           otherwise deserve */
        if (spread > 1 && !stereo)
          delta >>= 1;

        mbits = (b - qalloc / 2 - delta) / 2;
        if (mbits > b - qalloc)
          mbits = b - qalloc;
        if (mbits < 0)
          mbits = 0;
        sbits = b - qalloc - mbits;
        *remaining_bits -= qalloc;

        if (lowband && !stereo)
          next_lowband2 = lowband + N;
        if (stereo)
          next_lowband_out1 = lowband_out;
        else
          next_level = level + 1;

        quant_band(encode, m, i, X, NULL, N, mbits, spread, tf_change,
                   lowband, resynth, ec, remaining_bits, LM,
                   next_lowband_out1, NULL, next_level);
        quant_band(encode, m, i, Y, NULL, N, sbits, spread, tf_change,
                   next_lowband2, resynth, ec, remaining_bits, LM,
                   NULL, NULL, level);
      }

    } else
    {
      /* This is the basic no-split case */
      q = bits2pulses(m, m->bits[LM][i], N, b);
      curr_bits = pulses2bits(m->bits[LM][i], N, q);
      *remaining_bits -= curr_bits;

      /* Ensures we can never bust the budget */
      while (*remaining_bits < 0 && q > 0)
      {
        *remaining_bits += curr_bits;
        q--;
        curr_bits = pulses2bits(m->bits[LM][i], N, q);
        *remaining_bits -= curr_bits;
      }

      if (encode)
        alg_quant(X, N, q, spread, lowband, resynth, (ec_enc *) ec);
      else
        alg_unquant(X, N, q, spread, lowband, (ec_dec *) ec);
```

```
    }

    /* This code is used by the decoder and by the resynthesis-enabled
       encoder */
    if (resynth)
    {
      int             k;

      if (split)
      {
        int             j;
        float           mid,
                        side;

        mid = (1.f / 32768) * imid;
        side = (1.f / 32768) * iside;

        for (j = 0; j < N; j++)
          X[j] = ((X[j]) * (mid));
        for (j = 0; j < N; j++)
          Y[j] = ((Y[j]) * (side));
      }

      if (!stereo && spread0 > 1 && level == 0)
      {
        interleave_vector(X, N_B, spread0);
        if (lowband)
          interleave_vector(lowband, N_B, spread0);
      }

      /* Undo time-freq changes that we did earlier */
      N_B = N_B0;
      spread = spread0;
      for (k = 0; k < time_divide; k++)
      {
        spread >>= 1;
        N_B <<= 1;
        haar1(X, N_B, spread);
        if (lowband)
          haar1(lowband, N_B, spread);
      }

      for (k = 0; k < recombine; k++)
      {
        haar1(X, N_B, spread);
        if (lowband)
          haar1(lowband, N_B, spread);
        N_B >>= 1;
```

```
        spread <<= 1;
      }

      if (lowband_out && !stereo)
      {
        int             j;
        float           n;
        n = celt_sqrt(((N0)));
        for (j = 0; j < N0; j++)
          lowband_out[j] = ((n) * (X[j]));
      }

      if (stereo)
      {
        stereo_band_mix(m, X, Y, bandE, 0, i, -1, N);
        renormalise_vector(X, 1.0f, N, 1);
        renormalise_vector(Y, 1.0f, N, 1);
      }
    }
  }

  void
  quant_all_bands(int encode, const CELTMode * m, int start, float *_X,
                  float *_Y, const float *bandE, int *pulses,
                  int shortBlocks, int fold, int *tf_res, int resynth,
                  int total_bits, void *ec, int LM)
  {
    int             i,
                    balance;
    celt_int32      remaining_bits;
    const celt_int16 *restrict eBands = m->eBands;
    float          *restrict norm;
    VARDECL(float, _norm);
    int             B;
    int             M;
    int             spread;
    float          *lowband;
    int             update_lowband = 1;
    int             C = _Y != NULL ? 2 : 1;
    SAVE_STACK;

    M = 1 << LM;
    B = shortBlocks ? M : 1;
    spread = fold ? B : 0;
    ALLOC(_norm, M * eBands[m->nbEBands + 1], float);
    norm = _norm;

    balance = 0;
```

```
    lowband = NULL;
    for (i = start; i < m->nbEBands; i++)
    {
      int             tell;
      int             b;
      int             N;
      int             curr_balance;
      float          *restrict X,
                     *restrict Y;
      int             tf_change = 0;

      X = _X + M * eBands[i];
      if (_Y != NULL)
        Y = _Y + M * eBands[i];
      else
        Y = NULL;
      N = M * eBands[i + 1] - M * eBands[i];
      if (encode)
        tell = ec_enc_tell((ec_enc *) ec, BITRES);
      else
        tell = ec_dec_tell((ec_dec *) ec, BITRES);

      if (i != start)
        balance -= tell;
      remaining_bits = (total_bits << BITRES) - tell - 1;
      curr_balance = (m->nbEBands - i);
      if (curr_balance > 3)
        curr_balance = 3;
      curr_balance = balance / curr_balance;
      b = IMIN(remaining_bits + 1, pulses[i] + curr_balance);
      if (b < 0)
        b = 0;
      /* Prevents ridiculous bit depths */
      if (b > C * 16 * N << BITRES)
        b = C * 16 * N << BITRES;

      if (M * eBands[i] - N >= M * eBands[start])
      {
        if (update_lowband)
          lowband = norm + M * eBands[i] - N;
      } else
        lowband = NULL;

      tf_change = tf_res[i];
      quant_band(encode, m, i, X, Y, N, b, spread, tf_change, lowband,
                 resynth, ec, &remaining_bits, LM,
                 norm + M * eBands[i], bandE, 0);
```

```
      balance += pulses[i] + tell;

      /* Update the folding position only as long as we have 2
         bit/sample depth */
      update_lowband = (b >> BITRES) > 2 * N;
    }
    RESTORE_STACK;
  }
```

A.9. cwrs.h

```
   /* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
      Foundation Copyright (c) 2007-2009 Timothy B. Terriberry Written
      by Timothy B. Terriberry and Jean-Marc Valin */
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.

      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
      CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
      INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
      OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
      PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
      OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
      THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
      TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
      OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
      OF SUCH DAMAGE. */

   #ifndef CWRS_H
```

```
   #define CWRS_H

   #include "arch.h"
   #include "stack_alloc.h"
   #include "entenc.h"
   #include "entdec.h"

   int             log2_frac(ec_uint32 val, int frac);

   void            get_required_bits(celt_int16 * bits, int N, int K,
                                     int frac);

   void            encode_pulses(const int *_y, int N, int K,
                                 ec_enc * enc);

   void            decode_pulses(int *_y, int N, int K, ec_dec * dec);

   #endif                          /* CWRS_H */
```

A.10. cwrs.c

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Copyright (c) 2007-2009 Timothy B. Terriberry Written
   by Timothy B. Terriberry and Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
```

```
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "os_support.h"
#include <stdlib.h>
#include <string.h>
#include "cwrs.h"
#include "mathops.h"
#include "arch.h"

/* Guaranteed to return a conservatively large estimate of the
   binary logarithm with frac bits of fractional precision. Tested
   for all possible 32-bit inputs with frac=4, where the maximum
   overestimation is 0.06254243 bits. */
int
log2_frac(ec_uint32 val, int frac)
{
  int             l;
  l = EC_ILOG(val);
  if (val & val - 1)
  {
    /* This is (val>>l-16), but guaranteed to round up, even if
       adding a bias before the shift would cause overflow (e.g.,
       for 0xFFFFxxxx). */
    if (l > 16)
      val =
          (val >> l - 16) + ((val & (1 << l - 16) - 1) +
                             (1 << l - 16) - 1 >> l - 16);
    else
      val <<= 16 - l;
    l = l - 1 << frac;
    /* Note that we always need one iteration, since the rounding up
       above means that we might need to adjust the integer part of
       the logarithm. */
    do
    {
      int             b;
      b = (int) (val >> 16);
      l += b << frac;
      val = val + b >> b;
```

```
      val = val * val + 0x7FFF >> 15;
    }
    while (frac-- > 0);
    /* If val is not exactly 0x8000, then we have to round up the
       remainder. */
    return l + (val > 0x8000);
  }
  /* Exact powers of two require no rounding. */
  else
    return l - 1 << frac;
}

#define MASK32 (0xFFFFFFFF)

/* INV_TABLE[i] holds the multiplicative inverse of (2*i+1) mod
   2**32. */
static const celt_uint32 INV_TABLE[64] = {
  0x00000001, 0xAAAAAAAB, 0xCCCCCCCD, 0xB6DB6DB7,
  0x38E38E39, 0xBA2E8BA3, 0xC4EC4EC5, 0xEEEEEEEF,
  0xF0F0F0F1, 0x286BCA1B, 0x3CF3CF3D, 0xE9BD37A7,
  0xC28F5C29, 0x684BDA13, 0x4F72C235, 0xBDEF7BDF,
  0x3E0F83E1, 0x8AF8AF8B, 0x914C1BAD, 0x96F96F97,
  0xC18F9C19, 0x2FA0BE83, 0xA4FA4FA5, 0x677D46CF,
  0x1A1F58D1, 0xFAFAFAFB, 0x8C13521D, 0x586FB587,
  0xB823EE09, 0xA08AD8F3, 0xC10C9715, 0xBEFBEFBF,
  0xC0FC0FC1, 0x07A44C6B, 0xA33F128D, 0xE327A977,
  0xC7E3F1F9, 0x962FC963, 0x3F2B3885, 0x613716AF,
  0x781948B1, 0x2B2E43DB, 0xFCFCFCFD, 0x6FD0EB67,
  0xFA3F47E9, 0xD2FD2FD3, 0x3F4FD3F5, 0xD4E25B9F,
  0x5F02A3A1, 0xBF5A814B, 0x7C32B16D, 0xD3431B57,
  0xD8FD8FD9, 0x8D28AC43, 0xDA6C0965, 0xDB195E8F,
  0x0FDBC091, 0x61F2A4BB, 0xDCFDCFDD, 0x46FDD947,
  0x56BE69C9, 0xEB2FDEB3, 0x26E978D5, 0xEFDFBF7F,
  /*
     0x0FE03F81,0xC9484E2B,0xE133F84D,0xE1A8C537,
     0x077975B9,0x70586723,0xCD29C245,0xFAA11E6F,
     0x0FE3C071,0x08B51D9B,0x8CE2CABD,0xBF937F27,
     0xA8FE53A9,0x592FE593,0x2C0685B5,0x2EB11B5F,
     0xFCD1E361,0x451AB30B,0x72CFE72D,0xDB35A717,
     0xFB74A399,0xE80BFA03,0x0D516325,0x1BCB564F,
     0xE02E4851,0xD962AE7B,0x10F8ED9D,0x95AEDD07,
     0xE9DC0589,0xA18A4473,0xEA53FA95,0xEE936F3F,
     0x90948F41,0xEAFEAFEB,0x3D137E0D,0xEF46C0F7,
     0x028C1979,0x791064E3,0xC04FEC05,0xE115062F,
     0x32385831,0x6E68575B,0xA10D387D,0x6FECF2E7,
     0x3FB47F69,0xED4BFB53,0x74FED775,0xDB43BB1F,
     0x87654321,0x9BA144CB,0x478BBCED,0xBFB912D7,
     0x1FDCD759,0x14B2A7C3,0xCB125CE5,0x437B2E0F,
```

```
     0x10FEF011,0xD2B3183B,0x386CAB5D,0xEF6AC0C7,
     0x0E64C149,0x9A020A33,0xE6B41C55,0xFEFEFEFF */
};

/* Computes (_a*_b-_c)/(2*_d+1) when the quotient is known to be
   exact. _a, _b, _c, and _d may be arbitrary so long as the
   arbitrary precision result fits in 32 bits, but currently the
   table for multiplicative inverses is only valid for _d<128. */
static inline celt_uint32
imusdiv32odd(celt_uint32 _a, celt_uint32 _b, celt_uint32 _c, int _d)
{
  return (_a * _b - _c) * INV_TABLE[_d] & MASK32;
}

/* Computes (_a*_b-_c)/_d when the quotient is known to be exact. _d
   does not actually have to be even, but imusdiv32odd will be
   faster when it's odd, so you should use that instead. _a and _d
   are assumed to be small (e.g., _a*_d fits in 32 bits; currently
   the table for multiplicative inverses is only valid for _d<=256).
   _b and _c may be arbitrary so long as the arbitrary precision
   reuslt fits in 32 bits. */
static inline celt_uint32
imusdiv32even(celt_uint32 _a, celt_uint32 _b, celt_uint32 _c, int _d)
{
  celt_uint32     inv;
  int             mask;
  int             shift;
  int             one;
  celt_assert(_d > 0);
  shift = EC_ILOG(_d ^ _d - 1);
  celt_assert(_d <= 256);
  inv = INV_TABLE[_d - 1 >> shift];
  shift--;
  one = 1 << shift;
  mask = one - 1;
  return (_a * (_b >> shift) - (_c >> shift) +
          (_a * (_b & mask) + one - (_c & mask) >> shift) -
          1) * inv & MASK32;
}

/* Compute floor(sqrt(_val)) with exact arithmetic. This has been
   tested on all possible 32-bit inputs. */
static unsigned
isqrt32(celt_uint32 _val)
{
  unsigned        b;
  unsigned        g;
  int             bshift;
```

```
  /* Uses the second method from
     http://www.azillionmonkeys.com/qed/sqroot.html The main idea is
     to search for the largest binary digit b such that (g+b)*(g+b)
     <= _val, and add it to the solution g. */
  g = 0;
  bshift = EC_ILOG(_val) - 1 >> 1;
  b = 1U << bshift;
  do
  {
    celt_uint32     t;
    t = ((celt_uint32) g << 1) + b << bshift;
    if (t <= _val)
    {
      g += b;
      _val -= t;
    }
    b >>= 1;
    bshift--;
  }
  while (bshift >= 0);
  return g;
}

/* Although derived separately, the pulse vector coding scheme is
   equivalent to a Pyramid Vector Quantizer \cite{Fis86}. Some
   additional notes about an early version appear at
   http://people.xiph.org/~tterribe/notes/cwrs.html, but the
   codebook ordering and the definitions of some terms have evolved
   since that was written.

   The conversion from a pulse vector to an integer index (encoding)
   and back (decoding) is governed by two related functions, V(N,K)
   and U(N,K).

   V(N,K) = the number of combinations, with replacement, of N
   items, taken K at a time, when a sign bit is added to each item
   taken at least once (i.e., the number of N-dimensional unit pulse
   vectors with K pulses). One way to compute this is via V(N,K) =
   K>0 ? sum(k=1...K,2**k*choose(N,k)*choose(K-1,k-1)) : 1, where
   choose() is the binomial function. A table of values for N<10 and
   K<10 looks like: V[10][10] = { {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
   {1, 2, 2, 2, 2, 2, 2, 2, 2, 2}, {1, 4, 8, 12, 16, 20, 24, 28, 32,
   36}, {1, 6, 18, 38, 66, 102, 146, 198, 258, 326}, {1, 8, 32, 88,
   192, 360, 608, 952, 1408, 1992}, {1, 10, 50, 170, 450, 1002,
   1970, 3530, 5890, 9290}, {1, 12, 72, 292, 912, 2364, 5336, 10836,
   20256, 35436}, {1, 14, 98, 462, 1666, 4942, 12642, 28814, 59906,
   115598}, {1, 16, 128, 688, 2816, 9424, 27008, 68464, 157184,
   332688}, {1, 18, 162, 978, 4482, 16722, 53154, 148626, 374274,
```

864146} };

U(N,K) = the number of such combinations wherein N-1 objects are taken at most K-1 at a time. This is given by U(N,K) = sum(k=0...K-1,V(N-1,k)) = K>0 ? (V(N-1,K-1) + V(N,K-1))/2 : 0. The latter expression also makes clear that U(N,K) is half the number of such combinations wherein the first object is taken at least once. Although it may not be clear from either of these definitions, U(N,K) is the natural function to work with when enumerating the pulse vector codebooks, not V(N,K). U(N,K) is not well-defined for N=0, but with the extension U(0,K) = K>0 ? 0 : 1, the function becomes symmetric: U(N,K) = U(K,N), with a similar table: U[10][10] = { {1, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 1, 1, 1, 1, 1, 1, 1, 1, 1}, {0, 1, 3, 5, 7, 9, 11, 13, 15, 17}, {0, 1, 5, 13, 25, 41, 61, 85, 113, 145}, {0, 1, 7, 25, 63, 129, 231, 377, 575, 833}, {0, 1, 9, 41, 129, 321, 681, 1289, 2241, 3649}, {0, 1, 11, 61, 231, 681, 1683, 3653, 7183, 13073}, {0, 1, 13, 85, 377, 1289, 3653, 8989, 19825, 40081}, {0, 1, 15, 113, 575, 2241, 7183, 19825, 48639, 108545}, {0, 1, 17, 145, 833, 3649, 13073, 40081, 108545, 265729} };

With this extension, V(N,K) may be written in terms of U(N,K): V(N,K) = U(N,K) + U(N,K+1) for all N>=0, K>=0. Thus U(N,K+1) represents the number of combinations where the first element is positive or zero, and U(N,K) represents the number of combinations where it is negative. With a large enough table of U(N,K) values, we could write O(N) encoding and O(min(N*log(K),N+K)) decoding routines, but such a table would be prohibitively large for small embedded devices (K may be as large as 32767 for small N, and N may be as large as 200).

Both functions obey the same recurrence relation: V(N,K) = V(N-1,K) + V(N,K-1) + V(N-1,K-1), U(N,K) = U(N-1,K) + U(N,K-1) + U(N-1,K-1), for all N>0, K>0, with different initial conditions at N=0 or K=0. This allows us to construct a row of one of the tables above given the previous row or the next row. Thus we can derive O(NK) encoding and decoding routines with O(K) memory using only addition and subtraction.

When encoding, we build up from the U(2,K) row and work our way forwards. When decoding, we need to start at the U(N,K) row and work our way backwards, which requires a means of computing U(N,K). U(N,K) may be computed from two previous values with the same N: U(N,K) = ((2*N-1)*U(N,K-1) - U(N,K-2))/(K-1) + U(N,K-2) for all N>1, and since U(N,K) is symmetric, a similar relation holds for two previous values with the same K: U(N,K>1) = ((2*K-1)*U(N-1,K) - U(N-2,K))/(N-1) + U(N-2,K) for all K>1. This allows us to construct an arbitrary row of the U(N,K) table by

```
   starting with the first two values, which are constants. This
   saves roughly 2/3 the work in our O(NK) decoding routine, but
   costs O(K) multiplications. Similar relations can be derived for
   V(N,K), but are not used here.

   For N>0 and K>0, U(N,K) and V(N,K) take on the form of an
   (N-1)-degree polynomial for fixed N. The first few are U(1,K) = 1,
   U(2,K) = 2*K-1, U(3,K) = (2*K-2)*K+1, U(4,K) =
   (((4*K-6)*K+8)*K-3)/3, U(5,K) = ((((2*K-4)*K+10)*K-8)*K+3)/3, and
   V(1,K) = 2, V(2,K) = 4*K, V(3,K) = 4*K*K+2, V(4,K) = 8*(K*K+2)*K/3,
   V(5,K) = ((4*K*K+20)*K*K+6)/3, for all K>0. This allows us to derive
   O(N) encoding and O(N*log(K)) decoding routines for small N (and
   indeed decoding is also O(N) for N<3).

   @ARTICLE{Fis86, author="Thomas R. Fischer", title="A Pyramid Vector
   Quantizer", journal="IEEE Transactions on Information Theory",
   volume="IT-32", number=4, pages="568--583", month=Jul, year=1986 } */

/* Determines if V(N,K) fits in a 32-bit unsigned integer. N and K
   are themselves limited to 15 bits. */
static int
fits_in32(int _n, int _k)
{
  static const celt_int16 maxN[15] = {
    32767, 32767, 32767, 1476, 283, 109, 60, 40,
    29, 24, 20, 18, 16, 14, 13
  };
  static const celt_int16 maxK[15] = {
    32767, 32767, 32767, 32767, 1172, 238, 95, 53,
    36, 27, 22, 18, 16, 15, 13
  };
  if (_n >= 14)
  {
    if (_k >= 14)
      return 0;
    else
      return _n <= maxN[_k];
  } else
  {
    return _k <= maxK[_n];
  }
}

/* Compute U(1,_k). */
static inline unsigned
ucwrs1(int _k)
{
  return _k ? 1 : 0;
```

```
}

/* Compute V(1,_k). */
static inline unsigned
ncwrs1(int _k)
{
  return _k ? 2 : 1;
}

/* Compute U(2,_k). Note that this may be called with _k=32768
   (maxK[2]+1). */
static inline unsigned
ucwrs2(unsigned _k)
{
  return _k ? _k + (_k - 1) : 0;
}

/* Compute V(2,_k). */
static inline celt_uint32
ncwrs2(int _k)
{
  return _k ? 4 * (celt_uint32) _k : 1;
}

/* Compute U(3,_k). Note that this may be called with _k=32768
   (maxK[3]+1). */
static inline celt_uint32
ucwrs3(unsigned _k)
{
  return _k ? (2 * (celt_uint32) _k - 2) * _k + 1 : 0;
}

/* Compute V(3,_k). */
static inline celt_uint32
ncwrs3(int _k)
{
  return _k ? 2 * (2 * (unsigned) _k * (celt_uint32) _k + 1) : 1;
}

/* Compute U(4,_k). */
static inline celt_uint32
ucwrs4(int _k)
{
  return _k ? imusdiv32odd(2 * _k,
                           (2 * _k - 3) * (celt_uint32) _k + 4, 3,
                           1) : 0;
}
```

```
/* Compute V(4,_k). */
static inline celt_uint32
ncwrs4(int _k)
{
  return _k ? ((_k * (celt_uint32) _k + 2) * _k) / 3 << 3 : 1;
}

/* Compute U(5,_k). */
static inline celt_uint32
ucwrs5(int _k)
{
  return _k
      ? (((((_k - 2) * (unsigned) _k + 5) * (celt_uint32) _k -
           4) * _k) / 3 << 1) + 1 : 0;
}

/* Compute V(5,_k). */
static inline celt_uint32
ncwrs5(int _k)
{
  return _k ? (((_k * (unsigned) _k + 5) * (celt_uint32) _k * _k) /
              3 << 2) + 2 : 1;
}

/* Computes the next row/column of any recurrence that obeys the
   relation u[i][j]=u[i-1][j]+u[i][j-1]+u[i-1][j-1]. _ui0 is the
   base case for the new row/column. */
static inline void
unext(celt_uint32 * _ui, unsigned _len, celt_uint32 _ui0)
{
  celt_uint32     ui1;
  unsigned        j;
  /* This do-while will overrun the array if we don't have storage
     for at least 2 values. */
  j = 1;
  do
  {
    ui1 = ((((_ui[j]) + (_ui[j - 1]))) + (_ui0));
    _ui[j - 1] = _ui0;
    _ui0 = ui1;
  }
  while (++j < _len);
  _ui[j - 1] = _ui0;
}

/* Computes the previous row/column of any recurrence that obeys the
   relation u[i-1][j]=u[i][j]-u[i][j-1]-u[i-1][j-1]. _ui0 is the
   base case for the new row/column. */
```

```
static inline void
uprev(celt_uint32 * _ui, unsigned _n, celt_uint32 _ui0)
{
  celt_uint32     ui1;
  unsigned        j;
  /* This do-while will overrun the array if we don't have storage
     for at least 2 values. */
  j = 1;
  do
  {
    ui1 = ((((_ui[j]) - (_ui[j - 1]))) - (_ui0));
    _ui[j - 1] = _ui0;
    _ui0 = ui1;
  }
  while (++j < _n);
  _ui[j - 1] = _ui0;
}

/* Compute V(_n,_k), as well as U(_n,0..._k+1). _u: On exit, _u[i]
   contains U(_n,i) for i in [0..._k+1]. */
static celt_uint32
ncwrs_urow(unsigned _n, unsigned _k, celt_uint32 * _u)
{
  celt_uint32     um2;
  unsigned        len;
  unsigned        k;
  len = _k + 2;
  /* We require storage at least 3 values (e.g., _k>0). */
  celt_assert(len >= 3);
  _u[0] = 0;
  _u[1] = um2 = 1;

  if (_n <= 6 || _k > 255)

  {
    /* If _n==0, _u[0] should be 1 and the rest should be 0. */
    /* If _n==1, _u[i] should be 1 for i>1. */
    celt_assert(_n >= 2);
    /* If _k==0, the following do-while loop will overflow the
       buffer. */
    celt_assert(_k > 0);
    k = 2;
    do
      _u[k] = (k << 1) - 1;
    while (++k < len);
    for (k = 2; k < _n; k++)
      unext(_u + 1, _k + 1, 1);
  }
```

```
  else
  {
    celt_uint32     um1;
    celt_uint32     n2m1;
    _u[2] = n2m1 = um1 = (_n << 1) - 1;
    for (k = 3; k < len; k++)
    {
      /* U(N,K) = ((2*N-1)*U(N,K-1)-U(N,K-2))/(K-1) + U(N,K-2) */
      _u[k] = um2 = imusdiv32even(n2m1, um1, um2, k - 1) + um2;
      if (++k >= len)
        break;
      _u[k] = um1 = imusdiv32odd(n2m1, um2, um1, k - 1 >> 1) + um1;
    }
  }

  return _u[_k] + _u[_k + 1];
}

/* Returns the _i'th combination of _k elements (at most 32767)
   chosen from a set of size 1 with associated sign bits. _y:
   Returns the vector of pulses. */
static inline void
cwrsi1(int _k, celt_uint32 _i, int *_y)
{
  int             s;
  s = -(int) _i;
  _y[0] = _k + s ^ s;
}

/* Returns the _i'th combination of _k elements (at most 32767)
   chosen from a set of size 2 with associated sign bits. _y:
   Returns the vector of pulses. */
static inline void
cwrsi2(int _k, celt_uint32 _i, int *_y)
{
  celt_uint32     p;
  int             s;
  int             yj;
  p = ucwrs2(_k + 1U);
  s = -(_i >= p);
  _i -= p & s;
  yj = _k;
  _k = _i + 1 >> 1;
  p = ucwrs2(_k);
  _i -= p;
  yj -= _k;
  _y[0] = yj + s ^ s;
  cwrsi1(_k, _i, _y + 1);
```

```
}

/* Returns the _i'th combination of _k elements (at most 32767)
   chosen from a set of size 3 with associated sign bits. _y:
   Returns the vector of pulses. */
static void
cwrsi3(int _k, celt_uint32 _i, int *_y)
{
  celt_uint32     p;
  int             s;
  int             yj;
  p = ucwrs3(_k + 1U);
  s = -(_i >= p);
  _i -= p & s;
  yj = _k;
  /* Finds the maximum _k such that ucwrs3(_k)<=_i (tested for all
     _i<2147418113=U(3,32768)). */
  _k = _i > 0 ? isqrt32(2 * _i - 1) + 1 >> 1 : 0;
  p = ucwrs3(_k);
  _i -= p;
  yj -= _k;
  _y[0] = yj + s ^ s;
  cwrsi2(_k, _i, _y + 1);
}

/* Returns the _i'th combination of _k elements (at most 1172)
   chosen from a set of size 4 with associated sign bits. _y:
   Returns the vector of pulses. */
static void
cwrsi4(int _k, celt_uint32 _i, int *_y)
{
  celt_uint32     p;
  int             s;
  int             yj;
  int             kl;
  int             kr;
  p = ucwrs4(_k + 1);
  s = -(_i >= p);
  _i -= p & s;
  yj = _k;
  /* We could solve a cubic for k here, but the form of the direct
     solution does not lend itself well to exact integer arithmetic.
     Instead we do a binary search on U(4,K). */
  kl = 0;
  kr = _k;
  for (;;)
  {
    _k = kl + kr >> 1;
```

```
    p = ucwrs4(_k);
    if (p < _i)
    {
      if (_k >= kr)
        break;
      kl = _k + 1;
    } else if (p > _i)
      kr = _k - 1;
    else
      break;
  }
  _i -= p;
  yj -= _k;
  _y[0] = yj + s ^ s;
  cwrsi3(_k, _i, _y + 1);
}

/* Returns the _i'th combination of _k elements (at most 238) chosen
   from a set of size 5 with associated sign bits. _y: Returns the
   vector of pulses. */
static void
cwrsi5(int _k, celt_uint32 _i, int *_y)
{
  celt_uint32     p;
  int             s;
  int             yj;
  p = ucwrs5(_k + 1);
  s = -(_i >= p);
  _i -= p & s;
  yj = _k;
  /* A binary search on U(5,K) avoids the need for 64-bit arithmetic
   */
  {
    int             kl = 0;
    int             kr = _k;
    for (;;)
    {
      _k = kl + kr >> 1;
      p = ucwrs5(_k);
      if (p < _i)
      {
        if (_k >= kr)
          break;
        kl = _k + 1;
      } else if (p > _i)
        kr = _k - 1;
      else
        break;
```

```
    }
  }
  _i -= p;
  yj -= _k;
  _y[0] = yj + s ^ s;
  cwrsi4(_k, _i, _y + 1);
}

/* Returns the _i'th combination of _k elements chosen from a set of
   size _n with associated sign bits. _y: Returns the vector of
   pulses. _u: Must contain entries [0..._k+1] of row _n of U() on
   input. Its contents will be destructively modified. */
static void
cwrsi(int _n, int _k, celt_uint32 _i, int *_y, celt_uint32 * _u)
{
  int             j;
  celt_assert(_n > 0);
  j = 0;
  do
  {
    celt_uint32     p;
    int             s;
    int             yj;
    p = _u[_k + 1];
    s = -(_i >= p);
    _i -= p & s;
    yj = _k;
    p = _u[_k];
    while (p > _i)
      p = _u[--_k];
    _i -= p;
    yj -= _k;
    _y[j] = yj + s ^ s;
    uprev(_u, _k + 2, 0);
  }
  while (++j < _n);
}

/* Returns the index of the given combination of K elements chosen
   from a set of size 1 with associated sign bits. _y: The vector of
   pulses, whose sum of absolute values is K. _k: Returns K. */
static inline celt_uint32
icwrs1(const int *_y, int *_k)
{
  *_k = abs(_y[0]);
  return _y[0] < 0;
}
```

```
/* Returns the index of the given combination of K elements chosen
   from a set of size 2 with associated sign bits. _y: The vector of
   pulses, whose sum of absolute values is K. _k: Returns K. */
static inline celt_uint32
icwrs2(const int *_y, int *_k)
{
  celt_uint32     i;
  int             k;
  i = icwrs1(_y + 1, &k);
  i += ucwrs2(k);
  k += abs(_y[0]);
  if (_y[0] < 0)
    i += ucwrs2(k + 1U);
  *_k = k;
  return i;
}

/* Returns the index of the given combination of K elements chosen
   from a set of size 3 with associated sign bits. _y: The vector of
   pulses, whose sum of absolute values is K. _k: Returns K. */
static inline celt_uint32
icwrs3(const int *_y, int *_k)
{
  celt_uint32     i;
  int             k;
  i = icwrs2(_y + 1, &k);
  i += ucwrs3(k);
  k += abs(_y[0]);
  if (_y[0] < 0)
    i += ucwrs3(k + 1U);
  *_k = k;
  return i;
}

/* Returns the index of the given combination of K elements chosen
   from a set of size 4 with associated sign bits. _y: The vector of
   pulses, whose sum of absolute values is K. _k: Returns K. */
static inline celt_uint32
icwrs4(const int *_y, int *_k)
{
  celt_uint32     i;
  int             k;
  i = icwrs3(_y + 1, &k);
  i += ucwrs4(k);
  k += abs(_y[0]);
  if (_y[0] < 0)
    i += ucwrs4(k + 1);
  *_k = k;
```

```
  return i;
}

/* Returns the index of the given combination of K elements chosen
   from a set of size 5 with associated sign bits. _y: The vector of
   pulses, whose sum of absolute values is K. _k: Returns K. */
static inline celt_uint32
icwrs5(const int *_y, int *_k)
{
  celt_uint32     i;
  int             k;
  i = icwrs4(_y + 1, &k);
  i += ucwrs5(k);
  k += abs(_y[0]);
  if (_y[0] < 0)
    i += ucwrs5(k + 1);
  *_k = k;
  return i;
}

/* Returns the index of the given combination of K elements chosen
   from a set of size _n with associated sign bits. _y: The vector
   of pulses, whose sum of absolute values must be _k. _nc: Returns
   V(_n,_k). */
celt_uint32
icwrs(int _n, int _k, celt_uint32 * _nc, const int *_y,
      celt_uint32 * _u)
{
  celt_uint32     i;
  int             j;
  int             k;
  /* We can't unroll the first two iterations of the loop unless
     _n>=2. */
  celt_assert(_n >= 2);
  _u[0] = 0;
  for (k = 1; k <= _k + 1; k++)
    _u[k] = (k << 1) - 1;
  i = icwrs1(_y + _n - 1, &k);
  j = _n - 2;
  i += _u[k];
  k += abs(_y[j]);
  if (_y[j] < 0)
    i += _u[k + 1];
  while (j-- > 0)
  {
    unext(_u, _k + 2, 0);
    i += _u[k];
    k += abs(_y[j]);
```

```
    if (_y[j] < 0)
      i += _u[k + 1];
  }
  *_nc = _u[k] + _u[k + 1];
  return i;
}

void
get_required_bits(celt_int16 * _bits, int _n, int _maxk, int _frac)
{
  int             k;
  /*_maxk==0 => there's nothing to do.*/
  celt_assert(_maxk > 0);
  if (_n == 1)
  {
    _bits[0] = 0;
    for (k = 1; k < _maxk; k++)
      _bits[k] = 1 << _frac;
  } else
  {
    _bits[0] = 0;
    if (_maxk > 1)
    {
      VARDECL(celt_uint32, u);
      SAVE_STACK;
      ALLOC(u, _maxk + 1U, celt_uint32);
      ncwrs_urow(_n, _maxk - 1, u);
      for (k = 1; k < _maxk && fits_in32(_n, k); k++)
        _bits[k] = log2_frac(u[k] + u[k + 1], _frac);
      for (; k < _maxk; k++)
        _bits[k] = 10000;
      RESTORE_STACK;
    }
  }
}

void
encode_pulses(const int *_y, int _n, int _k, ec_enc * _enc)
{
  celt_uint32     i;
  if (_k == 0)
    return;
  celt_assert(fits_in32(_n, _k));
  switch (_n)
  {
  case 1:
    {
      i = icwrs1(_y, &_k);
```

```
      celt_assert(ncwrs1(_k) == 2);
      ec_enc_bits(_enc, i, 1);
    }
    break;

  case 2:
    {
      i = icwrs2(_y, &_k);
      ec_enc_uint(_enc, i, ncwrs2(_k));
    }
    break;
  case 3:
    {
      i = icwrs3(_y, &_k);
      ec_enc_uint(_enc, i, ncwrs3(_k));
    }
    break;
  case 4:
    {
      i = icwrs4(_y, &_k);
      ec_enc_uint(_enc, i, ncwrs4(_k));
    }
    break;
  case 5:
    {
      i = icwrs5(_y, &_k);
      ec_enc_uint(_enc, i, ncwrs5(_k));
    }
    break;

  default:
    {
      VARDECL(celt_uint32, u);
      celt_uint32     nc;
      SAVE_STACK;
      ALLOC(u, _k + 2U, celt_uint32);
      i = icwrs(_n, _k, &nc, _y, u);
      ec_enc_uint(_enc, i, nc);
      RESTORE_STACK;
    };
  }
}

void
decode_pulses(int *_y, int _n, int _k, ec_dec * _dec)
{
  if (_k == 0)
  {
```

```
    int             i;
    for (i = 0; i < _n; i++)
      _y[i] = 0;
    return;
  }
  celt_assert(fits_in32(_n, _k));
  switch (_n)
  {
  case 1:
    {
      celt_assert(ncwrs1(_k) == 2);
      cwrsi1(_k, ec_dec_bits(_dec, 1), _y);
    }
    break;

  case 2:
    cwrsi2(_k, ec_dec_uint(_dec, ncwrs2(_k)), _y);
    break;
  case 3:
    cwrsi3(_k, ec_dec_uint(_dec, ncwrs3(_k)), _y);
    break;
  case 4:
    cwrsi4(_k, ec_dec_uint(_dec, ncwrs4(_k)), _y);
    break;
  case 5:
    cwrsi5(_k, ec_dec_uint(_dec, ncwrs5(_k)), _y);
    break;

  default:
    {
      VARDECL(celt_uint32, u);
      SAVE_STACK;
      ALLOC(u, _k + 2U, celt_uint32);
      cwrsi(_n, _k, ec_dec_uint(_dec, ncwrs_urow(_n, _k, u)), _y, u);
      RESTORE_STACK;
    }
  }
}
```

A.11. vq.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/**
   @file vq.h
   @brief Vector quantisation of the residual
 */
```

```
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef VQ_H
#define VQ_H

#include "entenc.h"
#include "entdec.h"
#include "modes.h"

/** Algebraic pulse-vector quantiser. The signal x is replaced by the s\
um of
  * the pitch and a combination of pulses such that its norm is still e\
qual
  * to 1. This is the function that will typically require the most CPU\
.
 * @param x Residual signal to quantise/encode (returns quantised versi\
on)
 * @param W Perceptual weight to use when optimising (currently unused)
```

```
 * @param N Number of samples to encode
 * @param K Number of pulses to use
 * @param p Pitch vector (it is assumed that p+x is a unit vector)
 * @param enc Entropy encoder state
*/
void            alg_quant(float *X, int N, int K, int spread,
                          float *lowband, int resynth, ec_enc * enc);

/** Algebraic pulse decoder
 * @param x Decoded normalised spectrum (returned)
 * @param N Number of samples to decode
 * @param K Number of pulses to use
 * @param p Pitch vector (automatically added to x)
 * @param dec Entropy decoder state
 */
void            alg_unquant(float *X, int N, int K, int spread,
                            float *lowband, ec_dec * dec);

float           renormalise_vector(float *X, float value, int N,
                                   int stride);

/** Intra-frame predictor that matches a section of the current frame (\
at lower
 * frequencies) to encode the current band.
 * @param x Residual signal to quantise/encode (returns quantised versi\
on)
 * @param W Perceptual weight
 * @param N Number of samples to encode
 * @param K Number of pulses to use
 * @param Y Lower frequency spectrum to use, normalised to the same sta\
ndard deviation
 * @param P Pitch vector (it is assumed that p+x is a unit vector)
 * @param B Stride (number of channels multiplied by the number of MDCT\
s per frame)
 * @param N0 Number of valid offsets
 */
void            intra_fold(const CELTMode * m, int start, int N,
                           const float *restrict Y,
                           float *restrict P, int N0, int B, int M);

#endif                          /* VQ_H */
```

A.12. vq.c

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
```

```
#include "config.h"

#include "mathops.h"
#include "cwrs.h"
#include "vq.h"
#include "arch.h"
#include "os_support.h"
#include "rate.h"

#define M_PI 3.141592653

static void
exp_rotation1(float *X, int len, int dir, int stride, float c,
              float s)
{
  int             i;
  float          *Xptr;
```

```
  if (dir > 0)
    s = -s;
  Xptr = X;
  for (i = 0; i < len - stride; i++)
  {
    float           x1,
                    x2;
    x1 = Xptr[0];
    x2 = Xptr[stride];
    Xptr[stride] = ((((c) * (x2)) + ((s) * (x1))));
    *Xptr++ = ((((c) * (x1)) - ((s) * (x2))));
  }
  Xptr = &X[len - 2 * stride - 1];
  for (i = len - 2 * stride - 1; i >= 0; i--)
  {
    float           x1,
                    x2;
    x1 = Xptr[0];
    x2 = Xptr[stride];
    Xptr[stride] = ((((c) * (x2)) + ((s) * (x1))));
    *Xptr-- = ((((c) * (x1)) - ((s) * (x2))));
  }
}

static void
exp_rotation(float *X, int len, int dir, int stride, int K)
{
  int             i;
  float           c,
                  s;
  float           gain,
                  theta;
  int             stride2 = 0;
  /* int i; if (len>=30) { for (i=0;i<len;i++) X[i] = 0; X[14] = 1;
     K=5; } */
  if (2 * K >= len)
    return;
  gain = celt_div((float) ((1.f) * (len)), (float) (len + 10 * K));
  /* FIXME: Make that HALF16 instead of HALF32 */
  theta = (.5f * (((gain) * (gain))));

  c = celt_cos_norm((theta));
  s = celt_cos_norm((((1.0f) - (theta))));      /* sin(theta) */

  if (len >= 8 * stride)
  {
    stride2 = 1;
    /* This is just a simple way of computing sqrt(len/stride) with
```

```
       rounding. It's basically incrementing long as (stride2+0.5)^2
       < len/stride. I _think_ it is bit-exact */
    while ((stride2 * stride2 + stride2) * stride + (stride >> 2) <
           len)
      stride2++;
  }
  len /= stride;
  for (i = 0; i < stride; i++)
  {
    if (dir < 0)
    {
      if (stride2)
        exp_rotation1(X + i * len, len, dir, stride2, s, c);
      exp_rotation1(X + i * len, len, dir, 1, c, s);
    } else
    {
      exp_rotation1(X + i * len, len, dir, 1, c, s);
      if (stride2)
        exp_rotation1(X + i * len, len, dir, stride2, s, c);
    }
  }
  /* if (len>=30) { for (i=0;i<len;i++) printf ("%f ", X[i]); printf
     ("\n"); exit(0); } */
}

/** Takes the pitch vector and the decoded residual vector, computes th\
e gain
    that will give ||p+g*y||=1 and mixes the residual with the pitch. */
static void
normalise_residual(int *restrict iy, float *restrict X, int N, int K,
                   float Ryy)
{
  int             i;

  float           t;
  float           g;

  t = (Ryy);
  g = celt_rsqrt_norm(t);

  i = 0;
  do
    X[i] = ((((g) * (iy[i]))));
  while (++i < N);
}

void
alg_quant(float *X, int N, int K, int spread, float *lowband,
```

```
          int resynth, ec_enc * enc)
{
  VARDECL(float, y);
  VARDECL(int, iy);
  VARDECL(float, signx);
  int             j,
                  is;
  float           s;
  int             pulsesLeft;
  float           sum;
  float           xy,
                  yy;
  int             N_1;          /* Inverse of N, in Q14 format (even
                                   for float) */

  SAVE_STACK;

  if (K == 0)
  {
    if (lowband != NULL && resynth)
    {
      for (j = 0; j < N; j++)
        X[j] = lowband[j];
      renormalise_vector(X, 1.0f, N, 1);
    } else
    {
      /* This is important for encoding the side in stereo mode */
      for (j = 0; j < N; j++)
        X[j] = 0;
    }
    return;
  }
  K = get_pulses(K);

  ALLOC(y, N, float);
  ALLOC(iy, N, int);
  ALLOC(signx, N, float);
  N_1 = 512 / N;

  if (spread)
    exp_rotation(X, N, 1, spread, K);

  sum = 0;
  j = 0;
  do
  {
    if (X[j] > 0)
      signx[j] = 1;
```

```
    else
    {
      signx[j] = -1;
      X[j] = -X[j];
    }
    iy[j] = 0;
    y[j] = 0;
  }
  while (++j < N);

  xy = yy = 0;

  pulsesLeft = K;

  /* Do a pre-search by projecting on the pyramid */
  if (K > (N >> 1))
  {
    float           rcp;
    j = 0;
    do
    {
      sum += X[j];
    }
    while (++j < N);

    if (sum <= 1e-15f)

    {
      X[0] = (1.f);
      j = 1;
      do
        X[j] = 0;
      while (++j < N);
      sum = (1.f);
    }
    /* Do we have sufficient accuracy here? */
    rcp = (((K - 1) * (celt_rcp(sum))));
    j = 0;
    do
    {

      iy[j] = floor(rcp * X[j]);

      y[j] = (iy[j]);
      yy = ((yy) + (y[j]) * (y[j]));
      xy = ((xy) + (X[j]) * (y[j]));
      y[j] *= 2;
      pulsesLeft -= iy[j];
```

```
    }
    while (++j < N);
  }
  celt_assert2(pulsesLeft >= 1,
               "Allocated too many pulses in the quick pass");

  while (pulsesLeft > 0)
  {
    int             pulsesAtOnce = 1;
    int             best_id;
    float           magnitude;
    float           best_num = -1e15f;
    float           best_den = 0;

    /* Decide on how many pulses to find at once */
    pulsesAtOnce = (pulsesLeft * N_1) >> 9;     /* pulsesLeft/N */
    if (pulsesAtOnce < 1)
      pulsesAtOnce = 1;

    magnitude = (pulsesAtOnce);

    best_id = 0;
    /* The squared magnitude term gets added anyway, so we might as
       well add it outside the loop */
    yy = ((yy) + (magnitude) * (magnitude));
    /* Choose between fast and accurate strategy depending on where
       we are in the search */
    /* This should ensure that anything we can process will have a
       better score */
    j = 0;
    do
    {
      float           Rxy,
                      Ryy;
      /* Select sign based on X[j] alone */
      s = magnitude;
      /* Temporary sums of the new pulse(s) */
      Rxy = ((((xy) + (s) * (X[j]))));
      /* We're multiplying y[j] by two so we don't have to do it
         here */
      Ryy = ((((yy) + (s) * (y[j]))));

      /* Approximate score: we maximise Rxy/sqrt(Ryy) (we're
         guaranteed that Rxy is positive because the sign is
         pre-computed) */
      Rxy = ((Rxy) * (Rxy));
      /* The idea is to check for num/den >= best_num/best_den, but
         that way we can do it without any division */
```

```
      /* OPT: Make sure to use conditional moves here */
      if (((best_den) * (Rxy)) > ((Ryy) * (best_num)))
      {
        best_den = Ryy;
        best_num = Rxy;
        best_id = j;
      }
    }
    while (++j < N);

    j = best_id;
    is = pulsesAtOnce;
    s = (is);

    /* Updating the sums of the new pulse(s) */
    xy = xy + ((s) * (X[j]));
    /* We're multiplying y[j] by two so we don't have to do it here */
    yy = yy + ((s) * (y[j]));

    /* Only now that we've made the final choice, update y/iy */
    /* Multiplying y[j] by 2 so we don't have to do it everywhere
       else */
    y[j] += 2 * s;
    iy[j] += is;
    pulsesLeft -= pulsesAtOnce;
  }
  j = 0;
  do
  {
    X[j] = ((signx[j]) * (X[j]));
    if (signx[j] < 0)
      iy[j] = -iy[j];
  }
  while (++j < N);
  encode_pulses(iy, N, K, enc);

  /* Recompute the gain in one pass to reduce the encoder-decoder
     mismatch due to the recursive computation used in quantisation.
   */
  if (resynth)
  {
    normalise_residual(iy, X, N, K, ((yy)));
    if (spread)
      exp_rotation(X, N, -1, spread, K);
  }
  RESTORE_STACK;
}
```

```
/** Decode pulse vector and combine the result with the pitch vector to\
 produce
    the final normalised signal in the current band. */
void
alg_unquant(float *X, int N, int K, int spread, float *lowband,
            ec_dec * dec)
{
  int             i;
  float           Ryy;
  VARDECL(int, iy);
  SAVE_STACK;
  if (K == 0)
  {
    if (lowband != NULL)
    {
      for (i = 0; i < N; i++)
        X[i] = lowband[i];
      renormalise_vector(X, 1.0f, N, 1);
    } else
    {
      /* This is important for encoding the side in stereo mode */
      for (i = 0; i < N; i++)
        X[i] = 0;
    }
    return;
  }
  K = get_pulses(K);
  ALLOC(iy, N, int);
  decode_pulses(iy, N, K, dec);
  Ryy = 0;
  i = 0;
  do
  {
    Ryy = ((Ryy) + (iy[i]) * (iy[i]));
  }
  while (++i < N);
  normalise_residual(iy, X, N, K, Ryy);
  if (spread)
    exp_rotation(X, N, -1, spread, K);
  RESTORE_STACK;
}

float
renormalise_vector(float *X, float value, int N, int stride)
{
  int             i;

  float           E = 1e-15f;
```

```
  float           g;
  float           t;
  float          *xptr = X;
  for (i = 0; i < N; i++)
  {
    E = ((E) + (*xptr) * (*xptr));
    xptr += stride;
  }

  t = (E);
  g = ((value) * (celt_rsqrt_norm(t)));

  xptr = X;
  for (i = 0; i < N; i++)
  {
    *xptr = ((((g) * (*xptr))));
    xptr += stride;
  }
  return celt_sqrt(E);
}
```

A.13. pitch.h

```
   /* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
      Foundation Written by Jean-Marc Valin */
   /**
      @file pitch.h
      @brief Pitch analysis
    */

   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.
```

```
   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef _PITCH_H
#define _PITCH_H

#include "modes.h"

void            pitch_downsample(const float *restrict x,
                                 float *restrict x_lp, int len,
                                 int end, int _C,
                                 float *restrict xmem,
                                 float *restrict filt_mem);

void            pitch_search(const CELTMode * m,
                             const float *restrict x_lp,
                             float *restrict y, int len,
                             int max_pitch, int *pitch, float *xmem,
                             int M);

#endif
```

A.14. pitch.c

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/**
   @file pitch.c
   @brief Pitch analysis
 */

/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:
```

```
#include "config.h"

#include "pitch.h"
#include "os_support.h"
#include "modes.h"
#include "stack_alloc.h"
#include "mathops.h"

void
find_best_pitch(float *xcorr, float maxcorr, float *y, int yshift,
                int len, int max_pitch, int best_pitch[2])
{
  int             i,
                  j;
  float           Syy = 1;
  float           best_num[2];
  float           best_den[2];

  best_num[0] = -1;
  best_num[1] = -1;
  best_den[0] = 0;
```

```
    best_den[1] = 0;
    best_pitch[0] = 0;
    best_pitch[1] = 1;
    for (j = 0; j < len; j++)
      Syy = ((Syy) + (y[j]) * (y[j]));
    for (i = 0; i < max_pitch; i++)
    {
      float           score;
      if (xcorr[i] > 0)
      {
        float           num;
        float           xcorr16;
        xcorr16 = ((xcorr[i]));
        num = ((xcorr16) * (xcorr16));
        score = num * 1. / Syy;
        if (((num) * (best_den[1])) > ((best_num[1]) * (Syy)))
        {
          if (((num) * (best_den[0])) > ((best_num[0]) * (Syy)))
          {
            best_num[1] = best_num[0];
            best_den[1] = best_den[0];
            best_pitch[1] = best_pitch[0];
            best_num[0] = num;
            best_den[0] = Syy;
            best_pitch[0] = i;
          } else
          {
            best_num[1] = num;
            best_den[1] = Syy;
            best_pitch[1] = i;
          }
        }
      }
      Syy += (((y[i + len]) * (y[i + len]))) - (((y[i]) * (y[i])));
      Syy = MAX32(1, Syy);
    }
  }

  void
  pitch_downsample(const float *restrict x, float *restrict x_lp,
                   int len, int end, int _C, float *restrict xmem,
                   float *restrict filt_mem)
  {
    int             i;
    const int       C = CHANNELS(_C);
    for (i = 1; i < len >> 1; i++)
      x_lp[i] =
          ((.5f *
```

```
            ((.5f * (x[(2 * i - 1) * C] + x[(2 * i + 1) * C])) +
             x[2 * i * C])));
    x_lp[0] = ((.5f * ((.5f * (*xmem + x[C])) + x[0])));
    *xmem = x[end - C];
    if (C == 2)
    {
      for (i = 1; i < len >> 1; i++)
        x_lp[i] =
            ((.5f *
              ((.5f *
                (x[(2 * i - 1) * C + 1] + x[(2 * i + 1) * C + 1])) +
               x[2 * i * C + 1])));
      x_lp[0] += ((.5f * ((.5f * (x[C + 1])) + x[1])));
      *xmem += x[end - C + 1];
    }
  }

  void
  pitch_search(const CELTMode * m, const float *restrict x_lp,
               float *restrict y, int len, int max_pitch, int *pitch,
               float *xmem, int M)
  {
    int             i,
                    j;
    const int       lag = MAX_PERIOD;
    const int       N = M * m->eBands[m->nbEBands + 1];
    int             best_pitch[2] = { 0 };
    VARDECL(float, x_lp4);
    VARDECL(float, y_lp4);
    VARDECL(float, xcorr);
    float           maxcorr = 1;
    int             offset;
    int             shift = 0;

    SAVE_STACK;

    ALLOC(x_lp4, len >> 2, float);
    ALLOC(y_lp4, lag >> 2, float);
    ALLOC(xcorr, max_pitch >> 1, float);

    /* Downsample by 2 again */
    for (j = 0; j < len >> 2; j++)
      x_lp4[j] = x_lp[2 * j];
    for (j = 0; j < lag >> 2; j++)
      y_lp4[j] = y[2 * j];
    /* Coarse search with 4x decimation */

    for (i = 0; i < max_pitch >> 2; i++)
```

```
    {
      float           sum = 0;
      for (j = 0; j < len >> 2; j++)
        sum = ((sum) + (x_lp4[j]) * (y_lp4[i + j]));
      xcorr[i] = MAX32(-1, sum);
      maxcorr = MAX32(maxcorr, sum);
    }
    find_best_pitch(xcorr, maxcorr, y_lp4, 0, len >> 2, max_pitch >> 2,
                    best_pitch);

    /* Finer search with 2x decimation */
    maxcorr = 1;
    for (i = 0; i < max_pitch >> 1; i++)
    {
      float           sum = 0;
      xcorr[i] = 0;
      if (abs(i - 2 * best_pitch[0]) > 2
          && abs(i - 2 * best_pitch[1]) > 2)
        continue;
      for (j = 0; j < len >> 1; j++)
        sum += (((x_lp[j]) * (y[i + j])));
      xcorr[i] = MAX32(-1, sum);
      maxcorr = MAX32(maxcorr, sum);
    }
    find_best_pitch(xcorr, maxcorr, y, shift, len >> 1, max_pitch >> 1,
                    best_pitch);

    /* Refine by pseudo-interpolation */
    if (best_pitch[0] > 0 && best_pitch[0] < (max_pitch >> 1) - 1)
    {
      float           a,
                      b,
                      c;
      a = xcorr[best_pitch[0] - 1];
      b = xcorr[best_pitch[0]];
      c = xcorr[best_pitch[0] + 1];
      if ((c - a) > (((.7f)) * (b - a)))
        offset = 1;
      else if ((a - c) > (((.7f)) * (b - c)))
        offset = -1;
      else
        offset = 0;
    } else
    {
      offset = 0;
    }
    *pitch = 2 * best_pitch[0] - offset;
```

```
     CELT_MOVE(y, y + (N >> 1), (lag - N) >> 1);
     CELT_MOVE(y + ((lag - N) >> 1), x_lp, N >> 1);

     RESTORE_STACK;

     /* printf ("%d\n", *pitch); */
   }
```

A.15. rate.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef RATE_H
#define RATE_H
```

```
#define MAX_PSEUDO 40
#define LOG_MAX_PSEUDO 6

#define MAX_PULSES 128
#define LOG_MAX_PULSES 7

#define BITRES 3
#define FINE_OFFSET 25
#define QTHETA_OFFSET 18

#define BITOVERFLOW 30000

#include "cwrs.h"

static inline int
get_pulses(int i)
{
  return i < 8 ? i : (8 + (i & 7)) << ((i >> 3) - 1);
}

static inline int
bits2pulses(const CELTMode * m, const celt_int16 * cache, int N,
            int bits)
{
  int             i;
  int             lo,
                  hi;

  lo = 0;
  hi = MAX_PSEUDO - 1;
  for (i = 0; i < LOG_MAX_PSEUDO; i++)
  {
    int             mid = (lo + hi) >> 1;
    /* OPT: Make sure this is implemented with a conditional move */
    if (cache[mid] >= bits)
      hi = mid;
    else
      lo = mid;
  }
  if (bits - cache[lo] <= cache[hi] - bits)
    return lo;
  else
    return hi;

  lo = 0;
  hi = MAX_PULSES - 1;

#if 0                           /* Disabled until we can make that
```

```
                                  useful */
  /* Use of more than MAX_PULSES is disabled until we are able to
     cwrs that decently */
  if (bits > cache[MAX_PULSES - 1] && N <= 4)
  {
    /* int pulses; pulses = 127; while (16 +
       log2_frac(2*(pulses+1)*(pulses+1) + 1, BITRES) <= bits &&
       pulses < 32767) pulses++; */
    lo = 127;
    switch (N)
    {
    case 3:
      hi = 1024;
      for (i = 0; i < 10; i++)
      {
        int             pulses = (lo + hi) >> 1;
        if (log2_frac
            (((((pulses) * (pulses)) >> 1) + 1) >> 1, BITRES) > bits)
          hi = pulses;
        else
          lo = pulses;
      }
      break;
    case 4:
      hi = 1024;
      for (i = 0; i < 10; i++)
      {
        int             pulses = (lo + hi) >> 1;
        if (log2_frac
            ((((((pulses) * (pulses)) + 2) * (pulses))) / 3 << 3,
             BITRES) > bits)
          hi = pulses;
        else
          lo = pulses;
      }
      break;
    }
    return lo;
  }
#endif
  /* Instead of using the "bisection condition" we use a fixed
     number of iterations because it should be faster */
  /* while (hi-lo != 1) */
  for (i = 0; i < LOG_MAX_PULSES; i++)
  {
    int             mid = (lo + hi) >> 1;
    /* OPT: Make sure this is implemented with a conditional move */
    if (cache[mid] >= bits)
```

```
      hi = mid;
    else
      lo = mid;
  }
  if (bits - cache[lo] <= cache[hi] - bits)
    return lo;
  else
    return hi;
}

static inline int
pulses2bits(const celt_int16 * cache, int N, int pulses)
{
#if 0                           /* Use of more than MAX_PULSES is
                                   disabled until we are able to
                                   cwrs that decently */
  if (pulses > 127)
  {
    int             bits;
    switch (N)
    {
    case 3:
      bits =
          log2_frac(((((pulses) * (pulses)) >> 1) + 1) >> 1, BITRES);
      break;
    case 4:
      bits =
          log2_frac((((((pulses) * (pulses)) +
                       2) * (pulses))) / 3 << 3, BITRES);
      break;
    }
    /* printf ("%d <- %d\n", bits, pulses); */
    return bits;
  }
#endif
  return cache[pulses];
}

/** Computes a cache of the pulses->bits mapping in each band */
celt_int16    **compute_alloc_cache(CELTMode * m, int M);

/** Compute the pulse allocation, i.e. how many pulses will go in each
  * band.
 @param m mode
 @param offsets Requested increase or decrease in the number of bits for
                each band
 @param total Number of bands
 @param pulses Number of pulses per band (returned)
```

```
 @return Total number of bits allocated
*/
void            compute_allocation(const CELTMode * m, int start,
                                   int *offsets, int total,
                                   int *pulses, int *ebits,
                                   int *fine_priority, int _C,
                                   int M);

#endif
```

A.16. rate.c

```
   /* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
      Foundation Written by Jean-Marc Valin */
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.

      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
      CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
      INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
      OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
      PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
      OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
      THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
      TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
      OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
      OF SUCH DAMAGE. */

   #include "config.h"
```

```
#include <math.h>
#include "modes.h"
#include "cwrs.h"
#include "arch.h"
#include "os_support.h"

#include "entcode.h"
#include "rate.h"

celt_int16    **
compute_alloc_cache(CELTMode * m, int M)
{
  int             i,
                  prevN;
  int             error = 0;
  celt_int16    **bits;
  const celt_int16 *eBands = m->eBands;

  bits = celt_alloc(m->nbEBands * sizeof(celt_int16 *));
  if (bits == NULL)
    return NULL;

  prevN = -1;
  for (i = 0; i < m->nbEBands; i++)
  {
    int             N;
    if (M > 0)
      N = M * (eBands[i + 1] - eBands[i]);
    else
      N = (eBands[i + 1] - eBands[i]) >> 1;
    if (N == 0)
    {
      bits[i] = NULL;
      continue;
    }
    if (N == prevN)
    {
      bits[i] = bits[i - 1];
    } else
    {
      bits[i] = celt_alloc(MAX_PSEUDO * sizeof(celt_int16));
      if (bits[i] != NULL)
      {
        int             j;
        celt_int16      tmp[MAX_PULSES];
        get_required_bits(tmp, N, MAX_PULSES, BITRES);
        for (j = 0; j < MAX_PSEUDO; j++)
          bits[i][j] = tmp[get_pulses(j)];
```

```
        } else
        {
          error = 1;
        }
        prevN = N;
      }
    }
    if (error)
    {
      const celt_int16 *prevPtr = NULL;
      if (bits != NULL)
      {
        for (i = 0; i < m->nbEBands; i++)
        {
          if (bits[i] != prevPtr && bits[i] != NULL)
          {
            prevPtr = bits[i];
            celt_free((int *) bits[i]);
          }
        }
        free(bits);
        bits = NULL;
      }
    }
    return bits;
  }

  static inline void
  interp_bits2pulses(const CELTMode * m, int start, int *bits1,
                     int *bits2, int total, int *bits, int *ebits,
                     int *fine_priority, int len, int _C, int M)
  {
    int             psum;
    int             lo,
                    hi;
    int             j;
    int             logM;
    const int       C = CHANNELS(_C);
    SAVE_STACK;

    logM = log2_frac(M, BITRES);
    lo = 0;
    hi = 1 << BITRES;
    while (hi - lo != 1)
    {
      int             mid = (lo + hi) >> 1;
      psum = 0;
      for (j = start; j < len; j++)
```

```
        psum +=
            (((1 << BITRES) - mid) * bits1[j] +
             mid * bits2[j]) >> BITRES;
      if (psum > (total << BITRES))
        hi = mid;
      else
        lo = mid;
    }
    psum = 0;
    /* printf ("interp bisection gave %d\n", lo); */
    for (j = start; j < len; j++)
    {
      bits[j] =
          (((1 << BITRES) - lo) * bits1[j] + lo * bits2[j]) >> BITRES;
      psum += bits[j];
    }
    /* Allocate the remaining bits */
    {
      int             left,
                      perband;
      left = (total << BITRES) - psum;
      perband = left / (len - start);
      for (j = start; j < len; j++)
        bits[j] += perband;
      left = left - len * perband;
      for (j = start; j < start + left; j++)
        bits[j]++;
    }
    for (j = start; j < len; j++)
    {
      int             N,
                      d;
      int             offset;

      N = M * (m->eBands[j + 1] - m->eBands[j]);
      /* Compensate for the extra DoF in stereo */
      d = (C * N + ((C == 2 && N > 2) ? 1 : 0)) << BITRES;
      offset = FINE_OFFSET - m->logN[j] - logM;
      /* Offset for the number of fine bits compared to their "fair
         share" of total/N */
      offset = bits[j] - offset * N * C;
      /* Compensate for the prediction gain in stereo */
      if (C == 2)
        offset -= 1 << BITRES;
      if (offset < 0)
        offset = 0;
      ebits[j] = (2 * offset + d) / (2 * d);
      fine_priority[j] = ebits[j] * d >= offset;
```

```
      if (N == 1)
        ebits[j] = (bits[j] / C >> BITRES) - 1;
      /* Make sure not to bust */
      if (C * ebits[j] > (bits[j] >> BITRES))
        ebits[j] = bits[j] / C >> BITRES;

      if (ebits[j] > 7)
        ebits[j] = 7;
      /* The bits used for fine allocation can't be used for pulses */
      bits[j] -= C * ebits[j] << BITRES;
      if (bits[j] < 0)
        bits[j] = 0;
    }
    RESTORE_STACK;
  }

  void
  compute_allocation(const CELTMode * m, int start, int *offsets,
                     int total, int *pulses, int *ebits,
                     int *fine_priority, int _C, int M)
  {
    int             lo,
                    hi,
                    len,
                    j;
    const int       C = CHANNELS(_C);
    VARDECL(int, bits1);
    VARDECL(int, bits2);
    SAVE_STACK;

    len = m->nbEBands;
    ALLOC(bits1, len, int);
    ALLOC(bits2, len, int);

    lo = 0;
    hi = m->nbAllocVectors - 1;
    while (hi - lo != 1)
    {
      int             psum = 0;
      int             mid = (lo + hi) >> 1;
      for (j = start; j < len; j++)
      {
        int             N = m->eBands[j + 1] - m->eBands[j];
        bits1[j] =
            (C * M * N * m->allocVectors[mid * len + j] + offsets[j]);
        if (bits1[j] < 0)
          bits1[j] = 0;
        psum += bits1[j];
```

```
        /* printf ("%d ", bits[j]); */
      }
      /* printf ("\n"); */
      if (psum > (total << BITRES))
        hi = mid;
      else
        lo = mid;
      /* printf ("lo = %d, hi = %d\n", lo, hi); */
    }
    /* printf ("interp between %d and %d\n", lo, hi); */
    for (j = start; j < len; j++)
    {
      int             N = m->eBands[j + 1] - m->eBands[j];
      bits1[j] =
          C * M * N * m->allocVectors[lo * len + j] + offsets[j];
      bits2[j] =
          C * M * N * m->allocVectors[hi * len + j] + offsets[j];
      if (bits1[j] < 0)
        bits1[j] = 0;
      if (bits2[j] < 0)
        bits2[j] = 0;
    }
    interp_bits2pulses(m, start, bits1, bits2, total, pulses, ebits,
                       fine_priority, len, C, M);
    RESTORE_STACK;
  }
```

A.17. plc.h

```
  /* Copyright (c) 2009-2010 Xiph.Org Foundation Written by Jean-Marc
     Valin */
  /*
     Redistribution and use in source and binary forms, with or
     without modification, are permitted provided that the following
     conditions are met:

     - Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

     - Redistributions in binary form must reproduce the above
     copyright notice, this list of conditions and the following
     disclaimer in the documentation and/or other materials provided
     with the distribution.

     - Neither the name of the Xiph.org Foundation nor the names of
     its contributors may be used to endorse or promote products
     derived from this software without specific prior written
```

```
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef PLC_H
#define PLC_H

#include "arch.h"

#define LPC_ORDER 24

void            _celt_lpc(float *_lpc, const float *ac, int p);

void            fir(const float *x,
                    const float *num,
                    float *y, int N, int ord, float *mem);

void            iir(const float *x,
                    const float *den,
                    float *y, int N, int ord, float *mem);

void            _celt_autocorr(const float *x, float *ac,
                               const float *window, int overlap,
                               int lag, int n);

#endif                          /* PLC_H */
```

A.18. plc.c

```
/* Copyright (c) 2009-2010 Xiph.Org Foundation Written by Jean-Marc
   Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:
```

```
#include "config.h"

#include "plc.h"
#include "stack_alloc.h"
#include "mathops.h"

void
_celt_lpc(float *_lpc,          /* out: [0...p-1] LPC coefficients */
          const float *ac,      /* in: [0...p] autocorrelation
                                   values */
          int p)
{
  int             i,
                  j;
  float           r;
  float           error = ac[0];

  float          *lpc = _lpc;

  for (i = 0; i < p; i++)
    lpc[i] = 0;
```

```
    if (ac[0] != 0)
    {
      for (i = 0; i < p; i++)
      {
        /* Sum up this iteration's reflection coefficient */
        float           rr = 0;
        for (j = 0; j < i; j++)
          rr += ((lpc[j]) * (ac[i - j]));
        rr += (ac[i + 1]);
        r = -frac_div32((rr), error);
        /* Update LPC coefficients and total error */
        lpc[i] = (r);
        for (j = 0; j < (i + 1) >> 1; j++)
        {
          float           tmp1,
                          tmp2;
          tmp1 = lpc[j];
          tmp2 = lpc[i - 1 - j];
          lpc[j] = tmp1 + ((r) * (tmp2));
          lpc[i - 1 - j] = tmp2 + ((r) * (tmp1));
        }

        error = error - ((((r) * (r))) * (error));
        /* Bail out once we get 30 dB gain */

        if (error < .001 * ac[0])
          break;

      }
    }

  }

  void
  fir(const float *x,
      const float *num, float *y, int N, int ord, float *mem)
  {
    int             i,
                    j;

    for (i = 0; i < N; i++)
    {
      float           sum = ((x[i]));
      for (j = 0; j < ord; j++)
      {
        sum += ((num[j]) * (mem[j]));
      }
      for (j = ord - 1; j >= 1; j--)
```

```
      {
        mem[j] = mem[j - 1];
      }
      mem[0] = x[i];
      y[i] = (sum);
    }
  }

  void
  iir(const float *x,
      const float *den, float *y, int N, int ord, float *mem)
  {
    int             i,
                    j;
    for (i = 0; i < N; i++)
    {
      float           sum = x[i];
      for (j = 0; j < ord; j++)
      {
        sum -= ((den[j]) * (mem[j]));
      }
      for (j = ord - 1; j >= 1; j--)
      {
        mem[j] = mem[j - 1];
      }
      mem[0] = (sum);
      y[i] = sum;
    }
  }

  void
  _celt_autocorr(const float *x,  /* in: [0...n-1] samples x */
                 float *ac,       /* out: [0...lag-1] ac values */
                 const float *window, int overlap, int lag, int n)
  {
    float           d;
    int             i;
    VARDECL(float, xx);
    SAVE_STACK;
    ALLOC(xx, n, float);
    for (i = 0; i < n; i++)
      xx[i] = x[i];
    for (i = 0; i < overlap; i++)
    {
      xx[i] = ((x[i]) * (window[i]));
      xx[n - i - 1] = ((x[n - i - 1]) * (window[i]));
    }
    while (lag >= 0)
```

```
    {
      for (i = lag, d = 0; i < n; i++)
        d += xx[i] * xx[i - lag];
      ac[lag] = d;
      /* printf ("%f ", ac[lag]); */
      lag--;
    }
    /* printf ("\n"); */
    ac[0] += 10;

    RESTORE_STACK;
  }
```

A.19. mdct.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2008 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
```

```
   OF SUCH DAMAGE. */

/* This is a simple MDCT implementation that uses a N/4 complex FFT
   to do most of the work. It should be relatively straightforward
   to plug in pretty much and FFT here.

   This replaces the Vorbis FFT (and uses the exact same API), which
   was a bit too messy and that was ending up duplicating code
   (might as well use the same FFT everywhere).

   The algorithm is similar to (and inspired from) Fabrice Bellard's
   MDCT implementation in FFMPEG, but has differences in signs,
   ordering and scaling in many places. */

#ifndef MDCT_H
#define MDCT_H

#include "kiss_fft.h"
#include "arch.h"

typedef struct {
  int             n;
  int             maxshift;
  kiss_fft_cfg   *kfft;
  kiss_twiddle_scalar *restrict trig;
} mdct_lookup;

void            clt_mdct_init(mdct_lookup * l, int N, int maxshift);
void            clt_mdct_clear(mdct_lookup * l);

/** Compute a forward MDCT and scale by 4/N */
void            clt_mdct_forward(const mdct_lookup * l,
                                 kiss_fft_scalar * in,
                                 kiss_fft_scalar * out,
                                 const float *window, int overlap,
                                 int shift);

/** Compute a backward MDCT (no scaling) and performs weighted overlap-\
add
    (scales implicitly by 1/2) */
void            clt_mdct_backward(const mdct_lookup * l,
                                  kiss_fft_scalar * in,
                                  kiss_fft_scalar * out,
                                  const float *restrict window,
                                  int overlap, int shift);

#endif
```

A.20. mdct.c

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2008 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

/* This is a simple MDCT implementation that uses a N/4 complex FFT
   to do most of the work. It should be relatively straightforward
   to plug in pretty much and FFT here.

   This replaces the Vorbis FFT (and uses the exact same API), which
   was a bit too messy and that was ending up duplicating code
   (might as well use the same FFT everywhere).

   The algorithm is similar to (and inspired from) Fabrice Bellard's
   MDCT implementation in FFMPEG, but has differences in signs,
   ordering and scaling in many places. */
```

```
  #include "config.h"

  #include "mdct.h"
  #include "kfft_double.h"
  #include <math.h>
  #include "os_support.h"
  #include "mathops.h"
  #include "stack_alloc.h"

  #define M_PI 3.141592653

  void
  clt_mdct_init(mdct_lookup * l, int N, int maxshift)
  {
    int             i;
    int             N2,
                    N4;
    l->n = N;
    N2 = N >> 1;
    N4 = N >> 2;
    l->kfft = celt_alloc(sizeof(kiss_fft_cfg) * (maxshift + 1));
    l->maxshift = maxshift;
    for (i = 0; i <= maxshift; i++)
    {
      l->kfft[i] = cpx32_fft_alloc(N >> 2 >> i);

      if (l->kfft[i] == NULL)
        return;

    }
    l->trig =
        (kiss_twiddle_scalar *) celt_alloc((N4 + 1) *
                                           sizeof
                                           (kiss_twiddle_scalar));
    if (l->trig == NULL)
      return;
    /* We have enough points that sine isn't necessary */
    for (i = 0; i <= N4; i++)
      l->trig[i] = cos(2 * M_PI * i / N);

  }

  void
  clt_mdct_clear(mdct_lookup * l)
  {
    int             i;
    for (i = 0; i <= l->maxshift; i++)
      cpx32_fft_free(l->kfft[i]);
```

```
    celt_free(l->kfft);
    celt_free(l->trig);
  }

  void
  clt_mdct_forward(const mdct_lookup * l, kiss_fft_scalar * in,
                   kiss_fft_scalar * restrict out, const float *window,
                   int overlap, int shift)
  {
    int             i;
    int             N,
                    N2,
                    N4;
    kiss_twiddle_scalar sine;
    VARDECL(kiss_fft_scalar, f);
    SAVE_STACK;
    N = l->n;
    N >>= shift;
    N2 = N >> 1;
    N4 = N >> 2;
    ALLOC(f, N2, kiss_fft_scalar);
    /* sin(x) ~= x here */

    sine = 2 * M_PI * (.125f) / N;

    /* Consider the input to be composed of four blocks: [a, b, c, d] */
    /* Window, shuffle, fold */
    {
      /* Temp pointers to make it really clear to the compiler what
         we're doing */
      const kiss_fft_scalar *restrict xp1 = in + (overlap >> 1);
      const kiss_fft_scalar *restrict xp2 =
          in + N2 - 1 + (overlap >> 1);
      kiss_fft_scalar *restrict yp = out;
      const float    *restrict wp1 = window + (overlap >> 1);
      const float    *restrict wp2 = window + (overlap >> 1) - 1;
      for (i = 0; i < (overlap >> 2); i++)
      {
        /* Real part arranged as -d-cR, Imag part arranged as -b+aR */
        *yp++ = ((*wp2) * (xp1[N2])) + ((*wp1) * (*xp2));
        *yp++ = ((*wp1) * (*xp1)) - ((*wp2) * (xp2[-N2]));
        xp1 += 2;
        xp2 -= 2;
        wp1 += 2;
        wp2 -= 2;
      }
      wp1 = window;
      wp2 = window + overlap - 1;
```

```
      for (; i < N4 - (overlap >> 2); i++)
      {
        /* Real part arranged as a-bR, Imag part arranged as -c-dR */
        *yp++ = *xp2;
        *yp++ = *xp1;
        xp1 += 2;
        xp2 -= 2;
      }
      for (; i < N4; i++)
      {
        /* Real part arranged as a-bR, Imag part arranged as -c-dR */
        *yp++ = -((*wp1) * (xp1[-N2])) + ((*wp2) * (*xp2));
        *yp++ = ((*wp2) * (*xp1)) + ((*wp1) * (xp2[N2]));
        xp1 += 2;
        xp2 -= 2;
        wp1 += 2;
        wp2 -= 2;
      }
    }
    /* Pre-rotation */
    {
      kiss_fft_scalar *restrict yp = out;
      kiss_fft_scalar *t = &l->trig[0];
      for (i = 0; i < N4; i++)
      {
        kiss_fft_scalar re,
                        im,
                        yr,
                        yi;
        re = yp[0];
        im = yp[1];
        yr = -S_MUL(re, t[i << shift]) - S_MUL(im,
                                               t[(N4 - i) << shift]);
        yi = -S_MUL(im, t[i << shift]) + S_MUL(re,
                                               t[(N4 - i) << shift]);
        /* works because the cos is nearly one */
        *yp++ = yr + S_MUL(yi, sine);
        *yp++ = yi - S_MUL(yr, sine);
      }
    }

    /* N/4 complex FFT, down-scales by 4/N */
    cpx32_fft(l->kfft[shift], out, f, N4);

    /* Post-rotate */
    {
      /* Temp pointers to make it really clear to the compiler what
         we're doing */
```

```
      const kiss_fft_scalar *restrict fp = f;
      kiss_fft_scalar *restrict yp1 = out;
      kiss_fft_scalar *restrict yp2 = out + N2 - 1;
      kiss_fft_scalar *t = &l->trig[0];
      /* Temp pointers to make it really clear to the compiler what
         we're doing */
      for (i = 0; i < N4; i++)
      {
        kiss_fft_scalar yr,
                        yi;
        yr = S_MUL(fp[1], t[(N4 - i) << shift]) + S_MUL(fp[0],
                                                        t[i << shift]);
        yi = S_MUL(fp[0], t[(N4 - i) << shift]) - S_MUL(fp[1],
                                                        t[i << shift]);
        /* works because the cos is nearly one */
        *yp1 = yr - S_MUL(yi, sine);
        *yp2 = yi + S_MUL(yr, sine);;
        fp += 2;
        yp1 += 2;
        yp2 -= 2;
      }
    }
    RESTORE_STACK;
  }

  void
  clt_mdct_backward(const mdct_lookup * l, kiss_fft_scalar * in,
                    kiss_fft_scalar * restrict out,
                    const float *restrict window, int overlap,
                    int shift)
  {
    int             i;
    int             N,
                    N2,
                    N4;
    kiss_twiddle_scalar sine;
    VARDECL(kiss_fft_scalar, f);
    VARDECL(kiss_fft_scalar, f2);
    SAVE_STACK;
    N = l->n;
    N >>= shift;
    N2 = N >> 1;
    N4 = N >> 2;
    ALLOC(f, N2, kiss_fft_scalar);
    ALLOC(f2, N2, kiss_fft_scalar);
    /* sin(x) ~= x here */

    sine = 2 * M_PI * (.125f) / N;
```

```
    /* Pre-rotate */
    {
      /* Temp pointers to make it really clear to the compiler what
         we're doing */
      const kiss_fft_scalar *restrict xp1 = in;
      const kiss_fft_scalar *restrict xp2 = in + N2 - 1;
      kiss_fft_scalar *restrict yp = f2;
      kiss_fft_scalar *t = &l->trig[0];
      for (i = 0; i < N4; i++)
      {
        kiss_fft_scalar yr,
                        yi;
        yr = -S_MUL(*xp2, t[i << shift]) + S_MUL(*xp1,
                                                 t[(N4 - i) << shift]);
        yi = -S_MUL(*xp2, t[(N4 - i) << shift]) - S_MUL(*xp1,
                                                        t[i << shift]);
        /* works because the cos is nearly one */
        *yp++ = yr - S_MUL(yi, sine);
        *yp++ = yi + S_MUL(yr, sine);
        xp1 += 2;
        xp2 -= 2;
      }
    }

    /* Inverse N/4 complex FFT. This one should *not* downscale even
       in fixed-point */
    cpx32_ifft(l->kfft[shift], f2, f, N4);

    /* Post-rotate */
    {
      kiss_fft_scalar *restrict fp = f;
      kiss_fft_scalar *t = &l->trig[0];

      for (i = 0; i < N4; i++)
      {
        kiss_fft_scalar re,
                        im,
                        yr,
                        yi;
        re = fp[0];
        im = fp[1];
        /* We'd scale up by 2 here, but instead it's done when mixing
           the windows */
        yr = S_MUL(re, t[i << shift]) - S_MUL(im,
                                              t[(N4 - i) << shift]);
        yi = S_MUL(im, t[i << shift]) + S_MUL(re,
                                              t[(N4 - i) << shift]);
        /* works because the cos is nearly one */
```

```
        *fp++ = yr - S_MUL(yi, sine);
        *fp++ = yi + S_MUL(yr, sine);
      }
    }
    /* De-shuffle the components for the middle of the window only */
    {
      const kiss_fft_scalar *restrict fp1 = f;
      const kiss_fft_scalar *restrict fp2 = f + N2 - 1;
      kiss_fft_scalar *restrict yp = f2;
      for (i = 0; i < N4; i++)
      {
        *yp++ = -*fp1;
        *yp++ = *fp2;
        fp1 += 2;
        fp2 -= 2;
      }
    }

    /* Mirror on both sides for TDAC */
    {
      kiss_fft_scalar *restrict fp1 = f2 + N4 - 1;
      kiss_fft_scalar *restrict xp1 = out + N2 - 1;
      kiss_fft_scalar *restrict yp1 = out + N4 - overlap / 2;
      const float    *restrict wp1 = window;
      const float    *restrict wp2 = window + overlap - 1;
      for (i = 0; i < N4 - overlap / 2; i++)
      {
        *xp1 = *fp1;
        xp1--;
        fp1--;
      }
      for (; i < N4; i++)
      {
        kiss_fft_scalar x1;
        x1 = *fp1--;
        *yp1++ += -((*wp1) * (x1));
        *xp1-- += ((*wp2) * (x1));
        wp1++;
        wp2--;
      }
    }
    {
      kiss_fft_scalar *restrict fp2 = f2 + N4;
      kiss_fft_scalar *restrict xp2 = out + N2;
      kiss_fft_scalar *restrict yp2 = out + N - 1 - (N4 - overlap / 2);
      const float    *restrict wp1 = window;
      const float    *restrict wp2 = window + overlap - 1;
      for (i = 0; i < N4 - overlap / 2; i++)
```

```
      {
        *xp2 = *fp2;
        xp2++;
        fp2++;
      }
      for (; i < N4; i++)
      {
        kiss_fft_scalar x2;
        x2 = *fp2++;
        *yp2-- = ((*wp1) * (x2));
        *xp2++ = ((*wp2) * (x2));
        wp1++;
        wp2--;
      }
    }
    RESTORE_STACK;
  }
```

A.21. ecintrin.h

```
/* Copyright (c) 2003-2008 Timothy B. Terriberry Copyright (c) 2008
   Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
```

```
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

/* Some common macros for potential platform-specific optimization. */
#include <math.h>
#include <limits.h>
#if !defined(_ecintrin_H)
# define _ecintrin_H (1)

/* Some specific platforms may have optimized intrinsic or inline
   assembly versions of these functions which can substantially
   improve performance. We define macros for them to allow easy
   incorporation of these non-ANSI features. */

/* Note that we do not provide a macro for abs(), because it is
   provided as a library function, which we assume is translated
   into an intrinsic to avoid the function call overhead and then
   implemented in the smartest way for the target platform. With
   modern gcc (4.x), this is true: it uses cmov instructions if the
   architecture supports it and branchless bit-twiddling if it does
   not (the speed difference between the two approaches is not
   measurable). Interestingly, the bit-twiddling method was patented
   in 2000 (US 6,073,150) by Sun Microsystems, despite prior art
   dating back to at least 1996:
   http://web.archive.org/web/19961201174141/www.x86.org/ftp/articles/p\
entopt/PENTOPT.TXT
   On gcc 3.x, however, our assumption is not true, as abs() is transla\
ted to a
   conditional jump, which is horrible on deeply piplined architectures\
 (e.g., all
   consumer architectures for the past decade or more) when the sign ca\
nnot be reliably
   predicted. */

/* Modern gcc (4.x) can compile the naive versions of min and max
   with cmov if given an appropriate architecture, but the
   branchless bit-twiddling versions are just as fast, and do not
   require any special target architecture. Earlier gcc versions
   (3.x) compiled both code to the same assembly instructions,
   because of the way they represented ((_b)>(_a)) internally. */
#define EC_MAXI(_a,_b)      ((_a)-((_a)-(_b)&-((_b)>(_a))))
#define EC_MINI(_a,_b)      ((_a)+((_b)-(_a)&-((_b)<(_a))))
/* This has a chance of compiling branchless, and is just as fast as
   the bit-twiddling method, which is slightly less portable, since
```

```
   it relies on a sign-extended rightshift, which is not guaranteed
   by ANSI (but present on every relevant platform). */
#define EC_SIGNI(_a)        (((_a)>0)-((_a)<0))
/* Slightly more portable than relying on a sign-extended
   right-shift (which is not guaranteed by ANSI), and just as fast,
   since gcc (3.x and 4.x both) compile it into the right-shift
   anyway. */
#define EC_SIGNMASK(_a)     (-((_a)<0))
/* Clamps an integer into the given range. If _a>_c, then the lower
   bound _a is respected over the upper bound _c (this behavior is
   required to meet our documented API behavior). _a: The lower
   bound. _b: The value to clamp. _c: The upper boud. */
#define EC_CLAMPI(_a,_b,_c) (EC_MAXI(_a,EC_MINI(_b,_c)))

/* Count leading zeros. This macro should only be used for
   implementing ec_ilog(), if it is defined. All other code should
   use EC_ILOG() instead. */
#ifdef __GNUC_PREREQ
#if __GNUC_PREREQ(3,4)
# if INT_MAX>=2147483647
#  define EC_CLZ0 sizeof(unsigned)*CHAR_BIT
#  define EC_CLZ(_x) (__builtin_clz(_x))
# elif LONG_MAX>=2147483647L
#  define EC_CLZ0 sizeof(unsigned long)*CHAR_BIT
#  define EC_CLZ(_x) (__builtin_clzl(_x))
# endif
#endif
#endif

#if defined(EC_CLZ)
/* Note that __builtin_clz is not defined when _x==0, according to
   the gcc documentation (and that of the BSR instruction that
   implements it on x86). The majority of the time we can never pass
   it zero. When we need to, it can be special cased. */
# define EC_ILOG(_x) (EC_CLZ0-EC_CLZ(_x))
#elif defined(ENABLE_TI_DSPLIB)
#include "dsplib.h"
#define EC_ILOG(x) (31 - _lnorm(x))
#else
# define EC_ILOG(_x) (ec_ilog(_x))
#endif

#endif
```

A.22. entcode.h

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
   2008-2009 Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "celt_types.h"

#if !defined(_entcode_H)
# define _entcode_H (1)
# include <limits.h>
# include "ecintrin.h"

typedef celt_int32 ec_int32;
typedef celt_uint32 ec_uint32;
typedef struct ec_byte_buffer ec_byte_buffer;
```

```
   /* The number of bits to code at a time when coding bits directly. */
   # define EC_UNIT_BITS  (8)
   /* The mask for the given bits. */
   # define EC_UNIT_MASK  ((1U<<EC_UNIT_BITS)-1)

   /* Simple libogg1-style buffer. */
   struct ec_byte_buffer {
     unsigned char  *buf;
     unsigned char  *ptr;
     unsigned char  *end_ptr;
     long            storage;
   };

   /* Encoding functions. */
   void            ec_byte_writeinit_buffer(ec_byte_buffer * _b,
                                            unsigned char *_buf,
                                            long _size);
   void            ec_byte_shrink(ec_byte_buffer * _b, long _size);
   void            ec_byte_writeinit(ec_byte_buffer * _b);
   void            ec_byte_writetrunc(ec_byte_buffer * _b, long _bytes);
   void            ec_byte_write1(ec_byte_buffer * _b, unsigned _value);
   void            ec_byte_write_at_end(ec_byte_buffer * _b,
                                        unsigned _value);
   void            ec_byte_write4(ec_byte_buffer * _b,
                                  ec_uint32 _value);
   void            ec_byte_writecopy(ec_byte_buffer * _b, void *_source,
                                     long _bytes);
   void            ec_byte_writeclear(ec_byte_buffer * _b);
   /* Decoding functions. */
   void            ec_byte_readinit(ec_byte_buffer * _b,
                                    unsigned char *_buf, long _bytes);
   int             ec_byte_look1(ec_byte_buffer * _b);
   unsigned char   ec_byte_look_at_end(ec_byte_buffer * _b);
   int             ec_byte_look4(ec_byte_buffer * _b, ec_uint32 * _val);
   void            ec_byte_adv1(ec_byte_buffer * _b);
   void            ec_byte_adv4(ec_byte_buffer * _b);
   int             ec_byte_read1(ec_byte_buffer * _b);
   int             ec_byte_read4(ec_byte_buffer * _b, ec_uint32 * _val);
   /* Shared functions. */
   static inline void
   ec_byte_reset(ec_byte_buffer * _b)
   {
     _b->ptr = _b->buf;
   }

   static inline long
   ec_byte_bytes(ec_byte_buffer * _b)
   {
```

```
  return _b->ptr - _b->buf;
}

static inline unsigned char *
ec_byte_get_buffer(ec_byte_buffer * _b)
{
  return _b->buf;
}

int             ec_ilog(ec_uint32 _v);

#endif
```

A.23. entcode.c

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */
```

```
   #include "config.h"

   #include "entcode.h"

   int
   ec_ilog(ec_uint32 _v)
   {

     /* On a Pentium M, this branchless version tested as the fastest
        on 1,000,000,000 random 32-bit integers, edging out a similar
        version with branches, and a 256-entry LUT version. */
     int             ret;
     int             m;
     ret = !!_v;
     m = !!(_v & 0xFFFF0000) << 4;
     _v >>= m;
     ret |= m;
     m = !!(_v & 0xFF00) << 3;
     _v >>= m;
     ret |= m;
     m = !!(_v & 0xF0) << 2;
     _v >>= m;
     ret |= m;
     m = !!(_v & 0xC) << 1;
     _v >>= m;
     ret |= m;
     ret += !!(_v & 0x2);
     return ret;

   }
```

A.24. entenc.h

```
   /* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
      2008-2009 Xiph.Org Foundation */
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.
```

```
   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#if !defined(_entenc_H)
# define _entenc_H (1)
# include <stddef.h>
# include "entcode.h"

typedef struct ec_enc ec_enc;

/* The entropy encoder. */
struct ec_enc {
  /* Buffered output. */
  ec_byte_buffer *buf;
  /* A buffered output symbol, awaiting carry propagation. */
  int             rem;
  /* Number of extra carry propagating symbols. */
  size_t          ext;
  /* The number of values in the current range. */
  ec_uint32       rng;
  /* The low end of the current range (inclusive). */
  ec_uint32       low;
  /* Byte that will be written at the end */
  unsigned char   end_byte;
  /* Number of valid bits in end_byte */
  int             end_bits_left;
  int             nb_end_bits;
};

/* Initializes the encoder. _buf: The buffer to store output bytes
   in. This must have already been initialized for writing and
   reset. */
```

```
   void            ec_enc_init(ec_enc * _this, ec_byte_buffer * _buf);
   /* Encodes a symbol given its frequency information. The frequency
      information must be discernable by the decoder, assuming it has
      read only the previous symbols from the stream. It is allowable
      to change the frequency information, or even the entire source
      alphabet, so long as the decoder can tell from the context of the
      previously encoded information that it is supposed to do so as
      well. _fl: The cumulative frequency of all symbols that come
      before the one to be encoded. _fh: The cumulative frequency of all
      symbols up to and including the one to be encoded. Together with
      _fl, this defines the range [_fl,_fh) in which the decoded value
      will fall. _ft: The sum of the frequencies of all the symbols */
   void            ec_encode(ec_enc * _this, unsigned _fl, unsigned _fh,
                             unsigned _ft);
   void            ec_encode_bin(ec_enc * _this, unsigned _fl,
                                 unsigned _fh, unsigned _bits);
   void            ec_encode_raw(ec_enc * _this, unsigned _fl,
                                 unsigned _fh, unsigned bits);
   /* Encodes a sequence of raw bits in the stream. _fl: The bits to
      encode. _ftb: The number of bits to encode. This must be at least
      one, and no more than 32. */
   void            ec_enc_bits(ec_enc * _this, ec_uint32 _fl, int _ftb);
   /* Encodes a raw unsigned integer in the stream. _fl: The integer to
      encode. _ft: The number of integers that can be encoded (one more
      than the max). This must be at least one, and no more than
      2**32-1. */
   void            ec_enc_uint(ec_enc * _this, ec_uint32 _fl,
                               ec_uint32 _ft);

   /* Encode a bit that has a _prob/65536 probability of being a one */
   void            ec_enc_bit_prob(ec_enc * _this, int val,
                                   unsigned _prob);

   /* Returns the number of bits "used" by the encoded symbols so far.
      This same number can be computed by the decoder, and is suitable
      for making coding decisions. _b: The number of extra bits of
      precision to include. At most 16 will be accurate. Return: The
      number of bits scaled by 2**_b. This will always be slightly
      larger than the exact value (e.g., all rounding error is in the
      positive direction). */
   long            ec_enc_tell(ec_enc * _this, int _b);

   /* Indicates that there are no more symbols to encode. All reamining
      output bytes are flushed to the output buffer. ec_enc_init() must
      be called before the encoder can be used again. */
   void            ec_enc_done(ec_enc * _this);

   #endif
```

A.25. entenc.c

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
   2008-2009 Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "os_support.h"
#include "entenc.h"
#include "arch.h"

void
ec_byte_writeinit_buffer(ec_byte_buffer * _b, unsigned char *_buf,
                         long _size)
{
  _b->ptr = _b->buf = _buf;
```

```
    _b->end_ptr = _b->buf + _size - 1;
    _b->storage = _size;
  }

  void
  ec_byte_shrink(ec_byte_buffer * _b, long _size)
  {
    _b->end_ptr = _b->buf + _size - 1;
    _b->storage = _size;
  }

  void
  ec_byte_write1(ec_byte_buffer * _b, unsigned _value)
  {
    ptrdiff_t       endbyte;
    endbyte = _b->ptr - _b->buf;
    if (endbyte >= _b->storage)
    {
      celt_fatal("range encoder overflow\n");
    }
    *(_b->ptr++) = (unsigned char) _value;
  }

  void
  ec_byte_write_at_end(ec_byte_buffer * _b, unsigned _value)
  {
    if (_b->end_ptr < _b->ptr)
    {
      celt_fatal("byte buffer collision");
    }
    *(_b->end_ptr--) = (unsigned char) _value;
  }

  void
  ec_enc_bits(ec_enc * _this, ec_uint32 _fl, int _ftb)
  {
    unsigned        fl;
    unsigned        ft;
    while (_ftb > EC_UNIT_BITS)
    {
      _ftb -= EC_UNIT_BITS;
      fl = (unsigned) (_fl >> _ftb) & EC_UNIT_MASK;
      ec_encode_raw(_this, fl, fl + 1, EC_UNIT_BITS);
    }
    ft = 1 << _ftb;
    fl = (unsigned) _fl & ft - 1;
    ec_encode_raw(_this, fl, fl + 1, _ftb);
  }
```

```
void
ec_enc_uint(ec_enc * _this, ec_uint32 _fl, ec_uint32 _ft)
{
  unsigned        ft;
  unsigned        fl;
  int             ftb;
  /* In order to optimize EC_ILOG(), it is undefined for the value
     0. */
  celt_assert(_ft > 1);
  _ft--;
  ftb = EC_ILOG(_ft);
  if (ftb > EC_UNIT_BITS)
  {
    ftb -= EC_UNIT_BITS;
    ft = (_ft >> ftb) + 1;
    fl = (unsigned) (_fl >> ftb);
    ec_encode(_this, fl, fl + 1, ft);
    ec_enc_bits(_this, _fl, ftb);
  } else
  {
    ec_encode(_this, _fl, _fl + 1, _ft + 1);
  }
}
```

A.26. entdec.h

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
   2008-2009 Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
```

```
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#if !defined(_entdec_H)
# define _entdec_H (1)
# include "entcode.h"

typedef struct ec_dec ec_dec;

/* The entropy decoder. */
struct ec_dec {
  /* The buffer to decode. */
  ec_byte_buffer *buf;
  /* The remainder of a buffered input symbol. */
  int             rem;
  /* The number of values in the current range. */
  ec_uint32       rng;
  /* The difference between the top of the current range and the
     input value. */
  ec_uint32       dif;
  /* Normalization factor. */
  ec_uint32       nrm;
  /* Byte that will be written at the end */
  unsigned char   end_byte;
  /* Number of valid bits in end_byte */
  int             end_bits_left;
  int             nb_end_bits;
};

/* Initializes the decoder. _buf: The input buffer to use. Return: 0
   on success, or a negative value on error. */
void            ec_dec_init(ec_dec * _this, ec_byte_buffer * _buf);
/* Calculates the cumulative frequency for the next symbol. This can
   then be fed into the probability model to determine what that
   symbol is, and the additional frequency information required to
   advance to the next symbol. This function cannot be called more
   than once without a corresponding call to ec_dec_update(), or
   decoding will not proceed correctly. _ft: The total frequency of
```

```
      the symbols in the alphabet the next symbol was encoded with.
      Return: A cumulative frequency representing the encoded symbol.
      If the cumulative frequency of all the symbols before the one
      that was encoded was fl, and the cumulative frequency of all the
      symbols up to and including the one encoded is fh, then the
      returned value will fall in the range [fl,fh). */
   unsigned        ec_decode(ec_dec * _this, unsigned _ft);
   unsigned        ec_decode_bin(ec_dec * _this, unsigned _bits);
   unsigned        ec_decode_raw(ec_dec * _this, unsigned bits);

   /* Advance the decoder past the next symbol using the frequency
      information the symbol was encoded with. Exactly one call to
      ec_decode() must have been made so that all necessary
      intermediate calculations are performed. _fl: The cumulative
      frequency of all symbols that come before the symbol decoded.
      _fh: The cumulative frequency of all symbols up to and including
      the symbol decoded. Together with _fl, this defines the range
      [_fl,_fh) in which the value returned above must fall. _ft: The
      total frequency of the symbols in the alphabet the symbol decoded
      was encoded in. This must be the same as passed to the preceding
      call to ec_decode(). */
   void            ec_dec_update(ec_dec * _this, unsigned _fl,
                                 unsigned _fh, unsigned _ft);
   /* Extracts a sequence of raw bits from the stream. The bits must
      have been encoded with ec_enc_bits(). No call to ec_dec_update()
      is necessary after this call. _ftb: The number of bits to
      extract. This must be at least one, and no more than 32. Return:
      The decoded bits. */
   ec_uint32       ec_dec_bits(ec_dec * _this, int _ftb);
   /* Extracts a raw unsigned integer with a non-power-of-2 range from
      the stream. The bits must have been encoded with ec_enc_uint().
      No call to ec_dec_update() is necessary after this call. _ft: The
      number of integers that can be decoded (one more than the max).
      This must be at least one, and no more than 2**32-1. Return: The
      decoded bits. */
   ec_uint32       ec_dec_uint(ec_dec * _this, ec_uint32 _ft);

   /* Decode a bit that has a _prob/65536 probability of being a one */
   int             ec_dec_bit_prob(ec_dec * _this, unsigned _prob);

   /* Returns the number of bits "used" by the encoded symbols so far.
      This same number can be computed by the encoder, and is suitable
      for making coding decisions. _b: The number of extra bits of
      precision to include. At most 16 will be accurate. Return: The
      number of bits scaled by 2**_b. This will always be slightly
      larger than the exact value (e.g., all rounding error is in the
      positive direction). */
   long            ec_dec_tell(ec_dec * _this, int _b);
```

```
  #endif
```

A.27. entdec.c

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
   2008-2009 Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include <stddef.h>
#include "entdec.h"
#include "os_support.h"
#include "arch.h"

void
ec_byte_readinit(ec_byte_buffer * _b, unsigned char *_buf,
```

```
                  long _bytes)
 {
   _b->buf = _b->ptr = _buf;
   _b->storage = _bytes;
   _b->end_ptr = _b->buf + _bytes - 1;
 }

 unsigned char
 ec_byte_look_at_end(ec_byte_buffer * _b)
 {
   if (_b->end_ptr < _b->buf)
   {
     celt_fatal
         ("Trying to read raw bits before the beginning of the stream");
   }
   return *(_b->end_ptr--);
 }

 void
 ec_byte_adv1(ec_byte_buffer * _b)
 {
   _b->ptr++;
 }

 int
 ec_byte_read1(ec_byte_buffer * _b)
 {
   ptrdiff_t       endbyte;
   endbyte = _b->ptr - _b->buf;
   if (endbyte >= _b->storage)
     return -1;
   else
     return *(_b->ptr++);
 }

 ec_uint32
 ec_dec_bits(ec_dec * _this, int _ftb)
 {
   ec_uint32       t;
   unsigned        s;
   t = 0;
   while (_ftb > EC_UNIT_BITS)
   {
     s = ec_decode_raw(_this, EC_UNIT_BITS);
     /* ec_dec_update(_this,s,s+1,EC_UNIT_MASK+1); */
     t = t << EC_UNIT_BITS | s;
     _ftb -= EC_UNIT_BITS;
   }
```

```
   s = ec_decode_raw(_this, _ftb);
   /* ec_dec_update(_this,s,s+1,ft); */
   t = t << _ftb | s;
   return t;
 }

 ec_uint32
 ec_dec_uint(ec_dec * _this, ec_uint32 _ft)
 {
   ec_uint32       t;
   unsigned        ft;
   unsigned        s;
   int             ftb;
   t = 0;
   /* In order to optimize EC_ILOG(), it is undefined for the value
      0. */
   celt_assert(_ft > 1);
   _ft--;
   ftb = EC_ILOG(_ft);
   if (ftb > EC_UNIT_BITS)
   {
     ftb -= EC_UNIT_BITS;
     ft = (unsigned) (_ft >> ftb) + 1;
     s = ec_decode(_this, ft);
     ec_dec_update(_this, s, s + 1, ft);
     t = t << EC_UNIT_BITS | s;
     t = t << ftb | ec_dec_bits(_this, ftb);
     if (t > _ft)
     {
       celt_notify("uint decode error");
       t = _ft;
     }
     return t;
   } else
   {
     _ft++;
     s = ec_decode(_this, (unsigned) _ft);
     ec_dec_update(_this, s, s + 1, (unsigned) _ft);
     t = t << ftb | s;
     return t;
   }
 }
```

A.28. mfrngcod.h

```
   /* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
      2008-2009 Xiph.Org Foundation */
```

```
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.

      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
      CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
      INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
      OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
      PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
      OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
      THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
      TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
      OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
      OF SUCH DAMAGE. */

   #if !defined(_mfrngcode_H)
   # define _mfrngcode_H (1)
   # include "entcode.h"

   /* Constants used by the entropy encoder/decoder. */

   /* The number of bits to output at a time. */
   # define EC_SYM_BITS   (8)
   /* The total number of bits in each of the state registers. */
   # define EC_CODE_BITS  (32)
   /* The maximum symbol value. */
   # define EC_SYM_MAX    ((1U<<EC_SYM_BITS)-1)
   /* Bits to shift by to move a symbol into the high-order position. */
   # define EC_CODE_SHIFT (EC_CODE_BITS-EC_SYM_BITS-1)
   /* Carry bit of the high-order range symbol. */
   # define EC_CODE_TOP   (((ec_uint32)1U)<<EC_CODE_BITS-1)
```

```
   /* Low-order bit of the high-order range symbol. */
   # define EC_CODE_BOT   (EC_CODE_TOP>>EC_SYM_BITS)
   /* Code for which propagating carries are possible. */
   # define EC_CODE_CARRY (((ec_uint32)EC_SYM_MAX)<<EC_CODE_SHIFT)
   /* The number of bits available for the last, partial symbol in the
      code field. */
   # define EC_CODE_EXTRA ((EC_CODE_BITS-2)%EC_SYM_BITS+1)
   /* A mask for the bits available in the coding buffer. This allows
      different platforms to use a variable with more bits, if it is
      convenient. We will only use EC_CODE_BITS of it. */
   # define EC_CODE_MASK  ((((ec_uint32)1U)<<EC_CODE_BITS-1)-1<<1|1)

   #endif
```

A.29. rangeenc.c

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
   2008-2009 Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
```

```
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "arch.h"
#include "entenc.h"
#include "mfrngcod.h"

/* A range encoder. See rangedec.c and the references for
   implementation details \cite{Mar79,MNW98}.

   @INPROCEEDINGS{Mar79, author="Martin, G.N.N.", title="Range
   encoding: an algorithm for removing redundancy from a digitised
   message", booktitle="Video \& Data Recording Conference",
   year=1979, address="Southampton", month=Jul } @ARTICLE{MNW98,
   author="Alistair Moffat and Radford Neal and Ian H. Witten",
   title="Arithmetic Coding Revisited", journal="{ACM} Transactions
   on Information Systems", year=1998, volume=16, number=3,
   pages="256--294", month=Jul,
   URL="http://www.stanford.edu/class/ee398/handouts/papers/Moffat98Ari\
thmCoding.pdf"
   } */

/* Outputs a symbol, with a carry bit. If there is a potential to
   propagate a carry over several symbols, they are buffered until
   it can be determined whether or not an actual carry will occur.
   If the counter for the buffered symbols overflows, then the
   stream becomes undecodable. This gives a theoretical limit of a
   few billion symbols in a single packet on 32-bit systems. The
   alternative is to truncate the range in order to force a carry,
   but requires similar carry tracking in the decoder, needlessly
   slowing it down. */
static void
ec_enc_carry_out(ec_enc * _this, int _c)
{
  if (_c != EC_SYM_MAX)
  {
    /* No further carry propagation possible, flush buffer. */
    int             carry;
    carry = _c >> EC_SYM_BITS;
    /* Don't output a byte on the first write. This compare should
       be taken care of by branch-prediction thereafter. */
    if (_this->rem >= 0)
      ec_byte_write1(_this->buf, _this->rem + carry);
    if (_this->ext > 0)
    {
      unsigned       sym;
```

```
      sym = EC_SYM_MAX + carry & EC_SYM_MAX;
      do
        ec_byte_write1(_this->buf, sym);
      while (--(_this->ext) > 0);
    }
    _this->rem = _c & EC_SYM_MAX;
  } else
    _this->ext++;
}

static inline void
ec_enc_normalize(ec_enc * _this)
{
  /* If the range is too small, output some bits and rescale it. */
  while (_this->rng <= EC_CODE_BOT)
  {
    ec_enc_carry_out(_this, (int) (_this->low >> EC_CODE_SHIFT));
    /* Move the next-to-high-order symbol into the high-order
       position. */
    _this->low = _this->low << EC_SYM_BITS & EC_CODE_TOP - 1;
    _this->rng <<= EC_SYM_BITS;
  }
}

void
ec_enc_init(ec_enc * _this, ec_byte_buffer * _buf)
{
  _this->buf = _buf;
  _this->rem = -1;
  _this->ext = 0;
  _this->low = 0;
  _this->rng = EC_CODE_TOP;
  _this->end_byte = 0;
  _this->end_bits_left = 8;
  _this->nb_end_bits = 0;
}

void
ec_encode(ec_enc * _this, unsigned _fl, unsigned _fh, unsigned _ft)
{
  ec_uint32       r;
  r = _this->rng / _ft;
  if (_fl > 0)
  {
    _this->low += _this->rng - ((r) * ((_ft - _fl)));
    _this->rng = ((r) * ((_fh - _fl)));
  } else
    _this->rng -= ((r) * ((_ft - _fh)));
```

```
  ec_enc_normalize(_this);
}

void
ec_encode_bin(ec_enc * _this, unsigned _fl, unsigned _fh,
              unsigned _bits)
{
  ec_uint32       r;
  r = _this->rng >> _bits;
  if (_fl > 0)
  {
    _this->low += _this->rng - ((r) * (((1 << _bits) - _fl)));
    _this->rng = ((r) * ((_fh - _fl)));
  } else
    _this->rng -= ((r) * (((1 << _bits) - _fh)));
  ec_enc_normalize(_this);
}

/* The probability of having a "one" is given in 1/65536. */
void
ec_enc_bit_prob(ec_enc * _this, int _val, unsigned _prob)
{
  ec_uint32       r;
  ec_uint32       s;
  ec_uint32       l;
  r = _this->rng;
  l = _this->low;
  s = (r >> 16) * _prob;
  r -= s;
  if (_val)
    _this->low = l + r;
  _this->rng = _val ? s : r;
  ec_enc_normalize(_this);
}

void
ec_encode_raw(ec_enc * _this, unsigned _fl, unsigned _fh,
              unsigned bits)
{
  _this->nb_end_bits += bits;
  while (bits >= _this->end_bits_left)
  {
    _this->end_byte |= (_fl << (8 - _this->end_bits_left)) & 0xff;
    _fl >>= _this->end_bits_left;
    ec_byte_write_at_end(_this->buf, _this->end_byte);
    _this->end_byte = 0;
    bits -= _this->end_bits_left;
    _this->end_bits_left = 8;
```

```
  }
  _this->end_byte |= (_fl << (8 - _this->end_bits_left)) & 0xff;
  _this->end_bits_left -= bits;
}

long
ec_enc_tell(ec_enc * _this, int _b)
{
  ec_uint32       r;
  int             l;
  long            nbits;
  nbits =
      (ec_byte_bytes(_this->buf) + (_this->rem >= 0) +
       _this->ext) * EC_SYM_BITS;
  /* To handle the non-integral number of bits still left in the
     encoder state, we compute the number of bits of low that must
     be encoded to ensure that the value is inside the range for any
     possible subsequent bits. */
  nbits += EC_CODE_BITS + 1 + _this->nb_end_bits;
  nbits <<= _b;
  l = EC_ILOG(_this->rng);
  r = _this->rng >> l - 16;
  while (_b-- > 0)
  {
    int             b;
    r = r * r >> 15;
    b = (int) (r >> 16);
    l = l << 1 | b;
    r >>= b;
  }
  return nbits - l;
}

void
ec_enc_done(ec_enc * _this)
{
  ec_uint32       msk;
  ec_uint32       end;
  int             l;
  /* We output the minimum number of bits that ensures that the
     symbols encoded thus far will be decoded correctly regardless
     of the bits that follow. */
  l = EC_CODE_BITS - EC_ILOG(_this->rng);
  msk = EC_CODE_TOP - 1 >> l;
  end = _this->low + msk & ~msk;
  if ((end | msk) >= _this->low + _this->rng)
  {
    l++;
```

```
    msk >>= 1;
    end = _this->low + msk & ~msk;
  }
  while (l > 0)
  {
    ec_enc_carry_out(_this, (int) (end >> EC_CODE_SHIFT));
    end = end << EC_SYM_BITS & EC_CODE_TOP - 1;
    l -= EC_SYM_BITS;
  }
  /* If we have a buffered byte flush it into the output buffer. */
  if (_this->rem >= 0 || _this->ext > 0)
  {
    ec_enc_carry_out(_this, 0);
    _this->rem = -1;
  }
  {
    unsigned char  *ptr = _this->buf->ptr;
    while (ptr <= _this->buf->end_ptr)
      *ptr++ = 0;
    if (_this->end_bits_left != 8)
      *_this->buf->end_ptr |= _this->end_byte;
  }
}
```

A.30. rangedec.c

```
/* Copyright (c) 2001-2008 Timothy B. Terriberry Copyright (c)
   2008-2009 Xiph.Org Foundation */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
```

```
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "arch.h"
#include "entdec.h"
#include "mfrngcod.h"

/* A range decoder. This is an entropy decoder based upon
   \cite{Mar79}, which is itself a rediscovery of the FIFO
   arithmetic code introduced by \cite{Pas76}. It is very similar to
   arithmetic encoding, except that encoding is done with digits in
   any base, instead of with bits, and so it is faster when using
   larger bases (i.e.: a byte). The author claims an average waste
   of $\frac{1}{2}\log_b(2b)$ bits, where $b$ is the base, longer
   than the theoretical optimum, but to my knowledge there is no
   published justification for this claim. This only seems true when
   using near-infinite precision arithmetic so that the process is
   carried out with no rounding errors.

   IBM (the author's employer) never sought to patent the idea, and
   to my knowledge the algorithm is unencumbered by any patents,
   though its performance is very competitive with proprietary
   arithmetic coding. The two are based on very similar ideas,
   however. An excellent description of implementation details is
   available at http://www.arturocampos.com/ac_range.html A recent
   work \cite{MNW98} which proposes several changes to arithmetic
   encoding for efficiency actually re-discovers many of the
   principles behind range encoding, and presents a good theoretical
   analysis of them.

   End of stream is handled by writing out the smallest number of
   bits that ensures that the stream will be correctly decoded
   regardless of the value of any subsequent bits. ec_dec_tell() can
   be used to determine how many bits were needed to decode all the
   symbols thus far; other data can be packed in the remaining bits
   of the input buffer. @PHDTHESIS{Pas76, author="Richard Clark
```

```
   Pasco", title="Source coding algorithms for fast data
   compression", school="Dept. of Electrical Engineering, Stanford
   University", address="Stanford, CA", month=May, year=1976 }
   @INPROCEEDINGS{Mar79, author="Martin, G.N.N.", title="Range
   encoding: an algorithm for removing redundancy from a digitised
   message", booktitle="Video & Data Recording Conference",
   year=1979, address="Southampton", month=Jul } @ARTICLE{MNW98,
   author="Alistair Moffat and Radford Neal and Ian H. Witten",
   title="Arithmetic Coding Revisited", journal="{ACM} Transactions
   on Information Systems", year=1998, volume=16, number=3,
   pages="256--294", month=Jul,
   URL="http://www.stanford.edu/class/ee398/handouts/papers/Moffat98Ari\
thmCoding.pdf"
   } */

/* Gets the next byte of input. After all the bytes in the current
   packet have been consumed, and the extra end code returned if
   needed, this function will continue to return zero each time it
   is called. Return: The next byte of input. */
static int
ec_dec_in(ec_dec * _this)
{
  int             ret;
  ret = ec_byte_read1(_this->buf);
  if (ret < 0)
  {
    ret = 0;
    /* Needed to keep oc_dec_tell() operating correctly. */
    ec_byte_adv1(_this->buf);
  }
  return ret;
}

/* Normalizes the contents of dif and rng so that rng lies entirely
   in the high-order symbol. */
static inline void
ec_dec_normalize(ec_dec * _this)
{
  /* If the range is too small, rescale it and input some bits. */
  while (_this->rng <= EC_CODE_BOT)
  {
    int             sym;
    _this->rng <<= EC_SYM_BITS;
    /* Use up the remaining bits from our last symbol. */
    sym = _this->rem << EC_CODE_EXTRA & EC_SYM_MAX;
    /* Read the next value from the input. */
    _this->rem = ec_dec_in(_this);
    /* Take the rest of the bits we need from this new symbol. */
```

```
    sym |= _this->rem >> EC_SYM_BITS - EC_CODE_EXTRA;
    _this->dif = (_this->dif << EC_SYM_BITS) - sym & EC_CODE_MASK;
    /* dif can never be larger than EC_CODE_TOP. This is equivalent
       to the slightly more readable:
       if(_this->dif>EC_CODE_TOP)_this->dif-=EC_CODE_TOP; */
    _this->dif ^= _this->dif & _this->dif - 1 & EC_CODE_TOP;
  }
}

void
ec_dec_init(ec_dec * _this, ec_byte_buffer * _buf)
{
  _this->buf = _buf;
  _this->rem = ec_dec_in(_this);
  _this->rng = 1U << EC_CODE_EXTRA;
  _this->dif =
      _this->rng - (_this->rem >> EC_SYM_BITS - EC_CODE_EXTRA);
  /* Normalize the interval. */
  ec_dec_normalize(_this);
  /*_this->end_byte=ec_byte_look_at_end(_this->buf);*/
  _this->end_bits_left = 0;
  _this->nb_end_bits = 0;
}

unsigned
ec_decode(ec_dec * _this, unsigned _ft)
{
  unsigned        s;
  _this->nrm = _this->rng / _ft;
  s = (unsigned) ((_this->dif - 1) / _this->nrm);
  return _ft - EC_MINI(s + 1, _ft);
}

unsigned
ec_decode_bin(ec_dec * _this, unsigned _bits)
{
  unsigned        s;
  _this->nrm = _this->rng >> _bits;
  s = (unsigned) ((_this->dif - 1) / _this->nrm);
  return (1 << _bits) - EC_MINI(s + 1, 1 << _bits);
}

unsigned
ec_decode_raw(ec_dec * _this, unsigned bits)
{
  unsigned        value = 0;
  int             count = 0;
  _this->nb_end_bits += bits;
```

```
  while (bits >= _this->end_bits_left)
  {
    value |= _this->end_byte >> (8 - _this->end_bits_left) << count;
    count += _this->end_bits_left;
    bits -= _this->end_bits_left;
    _this->end_byte = ec_byte_look_at_end(_this->buf);
    _this->end_bits_left = 8;
  }
  value |=
      ((_this->
        end_byte >> (8 - _this->end_bits_left)) & ((1 << bits) -
                                                   1)) << count;
  _this->end_bits_left -= bits;
  return value;
}

void
ec_dec_update(ec_dec * _this, unsigned _fl, unsigned _fh,
              unsigned _ft)
{
  ec_uint32       s;
  s = ((_this->nrm) * ((_ft - _fh)));
  _this->dif -= s;
  _this->rng =
      _fl > 0 ? ((_this->nrm) * ((_fh - _fl))) : _this->rng - s;
  ec_dec_normalize(_this);
}

/* The probability of having a "one" is given in 1/65536. */
int
ec_dec_bit_prob(ec_dec * _this, unsigned _prob)
{
  ec_uint32       r;
  ec_uint32       s;
  ec_uint32       d;
  int             val;
  r = _this->rng;
  d = _this->dif;
  s = (r >> 16) * _prob;
  val = d <= s;
  if (!val)
    _this->dif = d - s;
  _this->rng = val ? s : r - s;
  ec_dec_normalize(_this);
  return val;
}

long
```

```
ec_dec_tell(ec_dec * _this, int _b)
{
  ec_uint32       r;
  int             l;
  long            nbits;
  nbits =
      (ec_byte_bytes(_this->buf) -
       (EC_CODE_BITS + EC_SYM_BITS - 1) / EC_SYM_BITS) * EC_SYM_BITS;
  /* To handle the non-integral number of bits still left in the
     decoder state, we compute the number of bits of low that must
     be encoded to ensure that the value is inside the range for any
     possible subsequent bits. */
  nbits += EC_CODE_BITS + 1 + _this->nb_end_bits;
  nbits <<= _b;
  l = EC_ILOG(_this->rng);
  r = _this->rng >> l - 16;
  while (_b-- > 0)
  {
    int             b;
    r = r * r >> 15;
    b = (int) (r >> 16);
    l = l << 1 | b;
    r >>= b;
  }
  return nbits - l;
}
```

A.31. laplace.h

```
   /* Copyright (c) 2007 CSIRO Copyright (c) 2007-2009 Xiph.Org
      Foundation Written by Jean-Marc Valin */
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
```

```
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "entenc.h"
#include "entdec.h"

int             ec_laplace_get_start_freq(int decay);

/** Encode a value that is assumed to be the realisation of a
    Laplace-distributed random process
 @param enc Entropy encoder state
 @param value Value to encode
 @param decay Probability of the value +/- 1, multiplied by 16384
*/
void            ec_laplace_encode(ec_enc * enc, int *value,
                                  int decay);

void            ec_laplace_encode_start(ec_enc * enc, int *value,
                                        int decay, int fs);

/** Decode a value that is assumed to be the realisation of a
    Laplace-distributed random process
 @param dec Entropy decoder state
 @param decay Probability of the value +/- 1, multiplied by 16384
 @return Value decoded
 */
int             ec_laplace_decode(ec_dec * dec, int decay);

int             ec_laplace_decode_start(ec_dec * dec, int decay,
                                        int fs);
```

A.32. laplace.c

```
/* Copyright (c) 2007 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#include "config.h"

#include "laplace.h"

int
ec_laplace_get_start_freq(int decay)
{
  int            fs =
      (((ec_uint32) 32768) * (16384 - decay)) / (16384 + decay);
  /* Making fs even so we're sure that all the range is used for +/-
     values */
```

```
  fs -= (fs & 1);
  return fs;
}

void
ec_laplace_encode_start(ec_enc * enc, int *value, int decay, int fs)
{
  int             i;
  int             fl;
  unsigned int    ft;
  int             s = 0;
  int             val = *value;
  if (val < 0)
  {
    s = 1;
    val = -val;
  }
  ft = 32768;
  fl = -fs;
  for (i = 0; i < val; i++)
  {
    int             tmp_l,
                    tmp_s;
    tmp_l = fl;
    tmp_s = fs;
    fl += fs * 2;
    fs = (fs * (ec_int32) decay) >> 14;
    if (fs == 0)
    {
      if (fl + 2 <= ft)
      {
        fs = 1;
      } else
      {
        fs = tmp_s;
        fl = tmp_l;
        if (s)
          *value = -i;
        else
          *value = i;
        break;
      }
    }
  }
  if (fl < 0)
    fl = 0;
  if (s)
    fl += fs;
```

```
  ec_encode_bin(enc, fl, fl + fs, 15);
}

void
ec_laplace_encode(ec_enc * enc, int *value, int decay)
{
  int             fs = ec_laplace_get_start_freq(decay);
  ec_laplace_encode_start(enc, value, decay, fs);
}

int
ec_laplace_decode_start(ec_dec * dec, int decay, int fs)
{
  int             val = 0;
  int             fl,
                  fh,
                  fm;
  unsigned int    ft;
  fl = 0;
  ft = 32768;
  fh = fs;
  fm = ec_decode_bin(dec, 15);
  while (fm >= fh && fs != 0)
  {
    fl = fh;
    fs = (fs * (ec_int32) decay) >> 14;
    if (fs == 0 && fh + 2 <= ft)
    {
      fs = 1;
    }
    fh += fs * 2;
    val++;
  }
  if (fl > 0)
  {
    if (fm >= fl + fs)
    {
      val = -val;
      fl += fs;
    } else
    {
      fh -= fs;
    }
  }
  /* Preventing an infinite loop in case something screws up in the
     decoding */
  if (fl == fh)
    fl--;
```

```
  ec_dec_update(dec, fl, fh, ft);
  return val;
}

int
ec_laplace_decode(ec_dec * dec, int decay)
{
  int             fs = ec_laplace_get_start_freq(decay);
  return ec_laplace_decode_start(dec, decay, fs);
}
```

A.33. quant_bands.h

```
/* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
   Foundation Written by Jean-Marc Valin */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */
```

```
#ifndef QUANT_BANDS
#define QUANT_BANDS

#include "arch.h"
#include "modes.h"
#include "entenc.h"
#include "entdec.h"
#include "mathops.h"

static inline float
amp2Log(float amp)
{
  return celt_log2(MAX32((.001f), (amp)));
}

static inline float
log2Amp(float lg)
{
  return (celt_exp2((lg)));
}

int            *quant_prob_alloc(const CELTMode * m);
void            quant_prob_free(int *freq);

void            compute_fine_allocation(const CELTMode * m,
                                        int *bits, int budget);

int             intra_decision(float *eBands, float *oldEBands,
                               int len);

unsigned        quant_coarse_energy(const CELTMode * m, int start,
                                    const float *eBands,
                                    float *oldEBands, int budget,
                                    int intra, int *prob,
                                    float *error, ec_enc * enc,
                                    int _C, float max_decay);

void            quant_fine_energy(const CELTMode * m, int start,
                                  float *eBands, float *oldEBands,
                                  float *error, int *fine_quant,
                                  ec_enc * enc, int _C);

void            quant_energy_finalise(const CELTMode * m, int start,
                                      float *eBands,
                                      float *oldEBands, float *error,
                                      int *fine_quant,
                                      int *fine_priority,
                                      int bits_left, ec_enc * enc,
```

```
                                     int _C);

   void            unquant_coarse_energy(const CELTMode * m, int start,
                                     float *eBands,
                                     float *oldEBands, int budget,
                                     int intra, int *prob,
                                     ec_dec * dec, int _C);

   void            unquant_fine_energy(const CELTMode * m, int start,
                                    float *eBands, float *oldEBands,
                                    int *fine_quant, ec_dec * dec,
                                    int _C);

   void            unquant_energy_finalise(const CELTMode * m,
                                      int start, float *eBands,
                                      float *oldEBands,
                                      int *fine_quant,
                                      int *fine_priority,
                                      int bits_left, ec_dec * dec,
                                      int _C);

   #endif                          /* QUANT_BANDS */
```

A.34. quant_bands.c

```
   /* Copyright (c) 2007-2008 CSIRO Copyright (c) 2007-2009 Xiph.Org
      Foundation Written by Jean-Marc Valin */
   /*
      Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the following
      conditions are met:

      - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

      - Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

      - Neither the name of the Xiph.org Foundation nor the names of
      its contributors may be used to endorse or promote products
      derived from this software without specific prior written
      permission.

      THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
      CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
```

```
      INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
      DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
      LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
      OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
      PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
      OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
      THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
      TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
      OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
      OF SUCH DAMAGE. */

   #include "config.h"

   #include "quant_bands.h"
   #include "laplace.h"
   #include <math.h>
   #include "os_support.h"
   #include "arch.h"
   #include "mathops.h"
   #include "stack_alloc.h"

   #define E_MEANS_SIZE (5)

   const float     eMeans[E_MEANS_SIZE] =
       { (7.5f), -(1.33f), -(2.f), -(0.42f), (0.17f) };

   /* FIXME: Implement for stereo */
   int
   intra_decision(float *eBands, float *oldEBands, int len)
   {
     int             i;
     float           dist = 0;
     for (i = 0; i < len; i++)
     {
       float           d = ((eBands[i]) - (oldEBands[i]));
       dist = ((dist) + (d) * (d));
     }
     return (dist) > 2 * len;
   }

   int            *
   quant_prob_alloc(const CELTMode * m)
   {
     int             i;
     int            *prob;
     prob = celt_alloc(4 * m->nbEBands * sizeof(int));
     if (prob == NULL)
```

```
      return NULL;
    for (i = 0; i < m->nbEBands; i++)
    {
      prob[2 * i] = 6000 - i * 200;
      prob[2 * i + 1] = ec_laplace_get_start_freq(prob[2 * i]);
    }
    for (i = 0; i < m->nbEBands; i++)
    {
      prob[2 * m->nbEBands + 2 * i] = 9000 - i * 240;
      prob[2 * m->nbEBands + 2 * i + 1] =
          ec_laplace_get_start_freq(prob[2 * m->nbEBands + 2 * i]);
    }
    return prob;
  }

  void
  quant_prob_free(int *freq)
  {
    celt_free(freq);
  }

  unsigned
  quant_coarse_energy(const CELTMode * m, int start,
                      const float *eBands, float *oldEBands,
                      int budget, int intra, int *prob, float *error,
                      ec_enc * enc, int _C, float max_decay)
  {
    int             i,
                    c;
    unsigned        bits_used = 0;
    float           prev[2] = { 0, 0 };
    float           coef = m->ePredCoef;
    float           beta;
    const int       C = CHANNELS(_C);

    if (intra)
    {
      coef = 0;
      prob += 2 * m->nbEBands;
    }
    /* The .8 is a heuristic */
    beta = (((.8f)) * (coef));

    /* Encode at a fixed coarse resolution */
    for (i = start; i < m->nbEBands; i++)
    {
      c = 0;
      do
```

```
      {
        int             qi;
        float           q;
        float           x;
        float           f;
        float           mean =
            (i - start <
             E_MEANS_SIZE) ? ((((eMeans[i - start]))) -
                              (((coef) * (eMeans[i - start])))) : 0;
        x = eBands[i + c * m->nbEBands];

        f = x - mean - coef * oldEBands[i + c * m->nbEBands] - prev[c];
        /* Rounding to nearest integer here is really important! */
        qi = (int) floor(.5f + f);

        if (qi < 0 && x < oldEBands[i + c * m->nbEBands] - max_decay)
        {
          qi += (oldEBands[i + c * m->nbEBands] - max_decay - x);
          if (qi > 0)
            qi = 0;
        }
        /* If we don't have enough bits to encode all the energy, just
           assume something safe. We allow slightly busting the budget
           here */
        bits_used = ec_enc_tell(enc, 0);
        if (bits_used > budget)
        {
          qi = -1;
          error[i + c * m->nbEBands] = (.5f);
        } else
        {
          ec_laplace_encode_start(enc, &qi, prob[2 * i],
                                  prob[2 * i + 1]);
          error[i + c * m->nbEBands] = (f) - (qi);
        }
        q = (qi);

        oldEBands[i + c * m->nbEBands] =
            (((coef) * (oldEBands[i + c * m->nbEBands])) + mean +
             prev[c] + ((q)));
        prev[c] = mean + prev[c] + ((q)) - ((beta) * (q));
      }
      while (++c < C);
    }
    return bits_used;
  }

  void
```

```
   quant_fine_energy(const CELTMode * m, int start, float *eBands,
                     float *oldEBands, float *error, int *fine_quant,
                     ec_enc * enc, int _C)
   {
     int             i,
                     c;
     const int       C = CHANNELS(_C);

     /* Encode finer resolution */
     for (i = start; i < m->nbEBands; i++)
     {
       celt_int16      frac = 1 << fine_quant[i];
       if (fine_quant[i] <= 0)
         continue;
       c = 0;
       do
       {
         int             q2;
         float           offset;

         q2 = (int) floor((error[i + c * m->nbEBands] + .5f) * frac);

         if (q2 > frac - 1)
           q2 = frac - 1;
         if (q2 < 0)
           q2 = 0;
         ec_enc_bits(enc, q2, fine_quant[i]);

         offset =
             (q2 + .5f) * (1 << (14 - fine_quant[i])) * (1.f / 16384) -
             .5f;

         oldEBands[i + c * m->nbEBands] += offset;
         error[i + c * m->nbEBands] -= offset;
         /* printf ("%f ", error[i] - offset); */
       }
       while (++c < C);
     }
   }

   void
   quant_energy_finalise(const CELTMode * m, int start, float *eBands,
                         float *oldEBands, float *error,
                         int *fine_quant, int *fine_priority,
                         int bits_left, ec_enc * enc, int _C)
   {
     int             i,
                     prio,
```

```
                  c;
    const int       C = CHANNELS(_C);

    /* Use up the remaining bits */
    for (prio = 0; prio < 2; prio++)
    {
      for (i = start; i < m->nbEBands && bits_left >= C; i++)
      {
        if (fine_quant[i] >= 7 || fine_priority[i] != prio)
          continue;
        c = 0;
        do
        {
          int             q2;
          float           offset;
          q2 = error[i + c * m->nbEBands] < 0 ? 0 : 1;
          ec_enc_bits(enc, q2, 1);

          offset =
              (q2 -
               .5f) * (1 << (14 - fine_quant[i] - 1)) * (1.f / 16384);

          oldEBands[i + c * m->nbEBands] += offset;
          bits_left--;
        }
        while (++c < C);
      }
    }
    c = 0;
    do
    {
      for (i = start; i < m->nbEBands; i++)
      {
        eBands[i + c * m->nbEBands] =
            log2Amp(oldEBands[i + c * m->nbEBands]);
        if (oldEBands[i + c * m->nbEBands] < -(7.f))
          oldEBands[i + c * m->nbEBands] = -(7.f);
      }
    }
    while (++c < C);
  }

  void
  unquant_coarse_energy(const CELTMode * m, int start, float *eBands,
                        float *oldEBands, int budget, int intra,
                        int *prob, ec_dec * dec, int _C)
  {
    int             i,
```

```
                  c;
    float         prev[2] = { 0, 0 };
    float         coef = m->ePredCoef;
    float         beta;
    const int     C = CHANNELS(_C);

    if (intra)
    {
      coef = 0;
      prob += 2 * m->nbEBands;
    }
    /* The .8 is a heuristic */
    beta = (((.8f)) * (coef));

    /* Decode at a fixed coarse resolution */
    for (i = start; i < m->nbEBands; i++)
    {
      c = 0;
      do
      {
        int             qi;
        float           q;
        float           mean =
            (i - start <
             E_MEANS_SIZE) ? ((((eMeans[i - start]))) -
                              (((coef) * (eMeans[i - start])))) : 0;
        /* If we didn't have enough bits to encode all the energy,
           just assume something safe. We allow slightly busting the
           budget here */
        if (ec_dec_tell(dec, 0) > budget)
          qi = -1;
        else
          qi = ec_laplace_decode_start(dec, prob[2 * i],
                                       prob[2 * i + 1]);
        q = (qi);

        oldEBands[i + c * m->nbEBands] =
            (((coef) * (oldEBands[i + c * m->nbEBands])) + mean +
             prev[c] + ((q)));
        prev[c] = mean + prev[c] + ((q)) - ((beta) * (q));
      }
      while (++c < C);
    }
  }

  void
  unquant_fine_energy(const CELTMode * m, int start, float *eBands,
                      float *oldEBands, int *fine_quant, ec_dec * dec,
```

```
                      int _C)
  {
    int             i,
                    c;
    const int       C = CHANNELS(_C);
    /* Decode finer resolution */
    for (i = start; i < m->nbEBands; i++)
    {
      if (fine_quant[i] <= 0)
        continue;
      c = 0;
      do
      {
        int             q2;
        float           offset;
        q2 = ec_dec_bits(dec, fine_quant[i]);

        offset =
            (q2 + .5f) * (1 << (14 - fine_quant[i])) * (1.f / 16384) -
            .5f;

        oldEBands[i + c * m->nbEBands] += offset;
      }
      while (++c < C);
    }
  }

  void
  unquant_energy_finalise(const CELTMode * m, int start, float *eBands,
                          float *oldEBands, int *fine_quant,
                          int *fine_priority, int bits_left,
                          ec_dec * dec, int _C)
  {
    int             i,
                    prio,
                    c;
    const int       C = CHANNELS(_C);

    /* Use up the remaining bits */
    for (prio = 0; prio < 2; prio++)
    {
      for (i = start; i < m->nbEBands && bits_left >= C; i++)
      {
        if (fine_quant[i] >= 7 || fine_priority[i] != prio)
          continue;
        c = 0;
        do
        {
```

```
          int             q2;
          float           offset;
          q2 = ec_dec_bits(dec, 1);

          offset =
              (q2 -
               .5f) * (1 << (14 - fine_quant[i] - 1)) * (1.f / 16384);

          oldEBands[i + c * m->nbEBands] += offset;
          bits_left--;
        }
        while (++c < C);
      }
    }
    c = 0;
    do
    {
      for (i = start; i < m->nbEBands; i++)
      {
        eBands[i + c * m->nbEBands] =
            log2Amp(oldEBands[i + c * m->nbEBands]);
        if (oldEBands[i + c * m->nbEBands] < -(7.f))
          oldEBands[i + c * m->nbEBands] = -(7.f);
      }
    }
    while (++c < C);
  }
```

A.35. arch.h

```
/* Copyright (C) 2003-2008 Jean-Marc Valin */
/**
   @file arch.h
   @brief Various architecture definitions for CELT
*/
/*
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

- Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
```

```
- Neither the name of the Xiph.org Foundation nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

#ifndef ARCH_H
#define ARCH_H

#include "celt_types.h"

#define CELT_SIG_SCALE 32768.

#define celt_fatal(str) _celt_fatal(str, __FILE__, __LINE__);
#ifdef ENABLE_ASSERTIONS
#define celt_assert(cond) {if (!(cond)) \
                {celt_fatal("assertion failed: " #cond);}}
#define celt_assert2(cond, message) {if (!(cond)) \
                {celt_fatal("assertion failed: " #cond "\n" message);}}
#else
#define celt_assert(cond)
#define celt_assert2(cond, message)
#endif


#define ABS(x) ((x) < 0 ? (-(x)) : (x))
#define ABS16(x) ((x) < 0 ? (-(x)) : (x))
#define MIN16(a,b) ((a) < (b) ? (a) : (b))
#define MAX16(a,b) ((a) > (b) ? (a) : (b))
#define ABS32(x) ((x) < 0 ? (-(x)) : (x))
#define MIN32(a,b) ((a) < (b) ? (a) : (b))
#define MAX32(a,b) ((a) > (b) ? (a) : (b))
#define IMIN(a,b) ((a) < (b) ? (a) : (b))
#define IMAX(a,b) ((a) > (b) ? (a) : (b))

#define float2int(flt) ((int)(floor(.5+flt)))
```

```
#define SCALEIN(a)      ((a)*CELT_SIG_SCALE)
#define SCALEOUT(a)     ((a)*(1/CELT_SIG_SCALE))

#ifndef GLOBAL_STACK_SIZE
#ifdef FIXED_POINT
#define GLOBAL_STACK_SIZE 25000
#else
#define GLOBAL_STACK_SIZE 40000
#endif
#endif

#endif /* ARCH_H */
```

A.36. mathops.h

```
/* Copyright (c) 2002-2008 Jean-Marc Valin Copyright (c) 2007-2008
   CSIRO Copyright (c) 2007-2009 Xiph.Org Foundation Written by
   Jean-Marc Valin */
/**
   @file mathops.h
   @brief Various math functions
*/
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
```

```
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef MATHOPS_H
#define MATHOPS_H

#include "arch.h"
#include "entcode.h"
#include "os_support.h"

#ifndef OVERRIDE_FIND_MAX16
static inline int
find_max16(float *x, int len)
{
  float           max_corr = -1e15f;
  int             i,
                  id = 0;
  for (i = 0; i < len; i++)
  {
    if (x[i] > max_corr)
    {
      id = i;
      max_corr = x[i];
    }
  }
  return id;
}
#endif

#ifndef OVERRIDE_FIND_MAX32
static inline int
find_max32(float *x, int len)
{
  float           max_corr = -1e15f;
  int             i,
                  id = 0;
  for (i = 0; i < len; i++)
  {
    if (x[i] > max_corr)
    {
      id = i;
      max_corr = x[i];
    }
  }
```

```
  return id;
}
#endif

/* Multiplies two 16-bit fractional values. Bit-exactness of this
   macro is important */
#define FRAC_MUL16(a,b) ((16384+((celt_int32)(celt_int16)(a)*(celt_int1\
6)(b)))>>15)

/* This is a cos() approximation designed to be bit-exact on any
   platform. Bit exactness with this approximation is important
   because it has an impact on the bit allocation */
static inline celt_int16
bitexact_cos(celt_int16 x)
{
  celt_int32      tmp;
  celt_int16      x2;
  tmp = (4096 + ((celt_int32) (x) * (x))) >> 13;
  if (tmp > 32767)
    tmp = 32767;
  x2 = tmp;
  x2 = (32767 - x2) + FRAC_MUL16(x2,
                                 (-7651 +
                                  FRAC_MUL16(x2,
                                             (8277 +
                                              FRAC_MUL16(-626,
                                                         x2)))));
  if (x2 > 32766)
    x2 = 32766;
  return 1 + x2;
}

#ifndef FIXED_POINT

#define celt_sqrt(x) ((float)sqrt(x))
#define celt_psqrt(x) ((float)sqrt(x))
#define celt_rsqrt(x) (1.f/celt_sqrt(x))
#define celt_rsqrt_norm(x) (celt_rsqrt(x))
#define celt_acos acos
#define celt_exp exp
#define celt_cos_norm(x) (cos((.5f*M_PI)*(x)))
#define celt_atan atan
#define celt_rcp(x) (1.f/(x))
#define celt_div(a,b) ((a)/(b))
#define frac_div32(a,b) ((float)(a)/(b))

#ifdef FLOAT_APPROX
```

```
/* Note: This assumes radix-2 floating point with the exponent at
   bits 23..30 and an offset of 127 denorm, +/- inf and NaN are
   *not* handled */

/** Base-2 log approximation (log2(x)). */
static inline float
celt_log2(float x)
{
  int             integer;
  float           frac;
  union {
    float           f;
    celt_uint32     i;
  } in;
  in.f = x;
  integer = (in.i >> 23) - 127;
  in.i -= integer << 23;
  frac = in.f - 1.5f;
  frac = -0.41445418f + frac * (0.95909232f
                                + frac * (-0.33951290f +
                                          frac * 0.16541097f));
  return 1 + integer + frac;
}

/** Base-2 exponential approximation (2^x). */
static inline float
celt_exp2(float x)
{
  int             integer;
  float           frac;
  union {
    float           f;
    celt_uint32     i;
  } res;
  integer = floor(x);
  if (integer < -50)
    return 0;
  frac = x - integer;
  /* K0 = 1, K1 = log(2), K2 = 3-4*log(2), K3 = 3*log(2) - 2 */
  res.f = 0.99992522f + frac * (0.69583354f
                                + frac * (0.22606716f +
                                          0.078024523f * frac));
  res.i = (res.i + (integer << 23)) & 0x7fffffff;
  return res.f;
}

#else
#define celt_log2(x) (1.44269504088963387*log(x))
```

```
#define celt_exp2(x) (exp(0.6931471805599453094*(x)))
#endif

#endif

#ifdef FIXED_POINT

#include "os_support.h"

#ifndef OVERRIDE_CELT_ILOG2
/** Integer log in base2. Undefined for zero and negative numbers */
static inline celt_int16
celt_ilog2(celt_int32 x)
{
  celt_assert2(x > 0,
               "celt_ilog2() only defined for strictly positive numbers\
");
  return EC_ILOG(x) - 1;
}
#endif

#ifndef OVERRIDE_CELT_MAXABS16
static inline float
celt_maxabs16(float *x, int len)
{
  int             i;
  float           maxval = 0;
  for (i = 0; i < len; i++)
    maxval = MAX16(maxval, ABS16(x[i]));
  return maxval;
}
#endif

/** Integer log in base2. Defined for zero, but not for negative number\
s */
static inline celt_int16
celt_zlog2(float x)
{
  return x <= 0 ? 0 : celt_ilog2(x);
}

/** Reciprocal sqrt approximation in the range [0.25,1) (Q16 in, Q14 ou\
t) */
static inline float
celt_rsqrt_norm(float x)
{
  float           n;
  float           r;
```

```
  float           r2;
  float           y;
  /* Range of n is [-16384,32767] ([-0.5,1) in Q15). */
  n = x - 32768;
  /* Get a rough initial guess for the root. The optimal minimax
     quadratic approximation (using relative error) is r =
     1.437799046117536+n*(-0.823394375837328+n*0.4096419668459485).
     Coefficients here, and the final result r, are Q14. */
  r = ((23557) + (((n) * (((-13490) + (((n) * (6713)))))))));
  /* We want y = x*r*r-1 in Q15, but x is 32-bit Q16 and r is Q14.
     We can compute the result from n and r using Q15 multiplies
     with some adjustment, carefully done to avoid overflow. Range
     of y is [-1564,1594]. */
  r2 = ((r) * (r));
  y = (((((((r2) * (n))) + (r2))) - (16384)));
  /* Apply a 2nd-order Householder iteration: r +=
     r*y*(y*0.375-0.5). This yields the Q14 reciprocal square root
     of the Q16 x, with a maximum relative error of 1.04956E-4, a
     (relative) RMSE of 2.80979E-5, and a peak absolute error of
     2.26591/16384. */
  return ((r) + (((r) * (((y) * (((((y) * (12288))) - (16384)))))))));

}

/** Reciprocal sqrt approximation (Q30 input, Q0 output or equivalent) \
*/
static inline float
celt_rsqrt(float x)
{
  int             k;
  k = celt_ilog2(x) >> 1;
  x = (x);
  return (celt_rsqrt_norm(x));
}

/** Sqrt approximation (QX input, QX/2 output) */
static inline float
celt_sqrt(float x)
{
  int             k;
  float           n;
  float           rt;
  static const float C[5] = { 23175, 11561, -3011, 1699, -664 };
  if (x == 0)
    return 0;
  k = (celt_ilog2(x) >> 1) - 7;
  x = (x);
  n = x - 32768;
```

```
  rt = ((C[0]) +
        (((n) *
          (((C[1]) +
            (((n) *
              (((C[2]) +
                (((n) * (((C[3]) + (((n) * ((C[4]))))))))))))))));

  rt = (rt);
  return rt;
}

/** Sqrt approximation (QX input, QX/2 output) that assumes that the in\
put is
    strictly positive */
static inline float
celt_psqrt(float x)
{
  int             k;
  float           n;
  float           rt;
  static const float C[5] = { 23175, 11561, -3011, 1699, -664 };
  k = (celt_ilog2(x) >> 1) - 7;
  x = (x);
  n = x - 32768;
  rt = ((C[0]) +
        (((n) *
          (((C[1]) +
            (((n) *
              (((C[2]) +
                (((n) * (((C[3]) + (((n) * ((C[4]))))))))))))))));

  rt = (rt);
  return rt;
}

#define L1 32767
#define L2 -7651
#define L3 8277
#define L4 -626

static inline float
_celt_cos_pi_2(float x)
{
  float           x2;

  x2 = ((x) * (x));
  return ((1) +
          (MIN16
```

```
           (32766,
            ((((L1) - (x2))) +
             (((x2) *
               (((L2) +
                 (((x2) * (((L3) + (((L4) * (x2))))))))))))));

}

#undef L1
#undef L2
#undef L3
#undef L4

static inline float
celt_cos_norm(float x)
{
  x = x & 0x0001ffff;
  if (x > ((1)))
    x = ((((1))) - (x));
  if (x & 0x00007fff)
  {
    if (x < ((1)))
    {
      return _celt_cos_pi_2((x));
    } else
    {
      return (-(_celt_cos_pi_2((65536 - x))));
    }
  } else
  {
    if (x & 0x0000ffff)
      return 0;
    else if (x & 0x0001ffff)
      return -32767;
    else
      return 32767;
  }
}

static inline float
celt_log2(float x)
{
  int             i;
  float           n,
                  frac;
  /* -0.41509302963303146, 0.9609890551383969, -0.31836011537636605,
     0.15530808010959576, -0.08556153059057618 */
  static const float C[5] =
```

```
      { -6801 + (1 << 13 - DB_SHIFT), 15746, -5217, 2545, -1401 };
  if (x == 0)
    return -32767;
  i = celt_ilog2(x);
  n = (x) - 32768 - 16384;
  frac =
      ((C[0]) +
       (((n) *
         (((C[1]) +
           (((n) *
             (((C[2]) +
               (((n) * (((C[3]) + (((n) * (C[4]))))))))))))))));
  return (i - 13) + (frac);
}

/*
   K0 = 1 K1 = log(2) K2 = 3-4*log(2) K3 = 3*log(2) - 2 */
#define D0 16383
#define D1 22804
#define D2 14819
#define D3 10204
/** Base-2 exponential approximation (2^x). (Q11 input, Q16 output) */
static inline float
celt_exp2(float x)
{
  int             integer;
  float           frac;
  integer = (x);
  if (integer > 14)
    return 0x7f000000;
  else if (integer < -15)
    return 0;
  frac = (x - (integer));
  frac =
      ((D0) +
       (((frac) *
         (((D1) + (((frac) * (((D2) + (((D3) * (frac))))))))))));
  return ((frac));
}

/** Reciprocal approximation (Q15 input, Q16 output) */
static inline float
celt_rcp(float x)
{
  int             i;
  float           n;
  float           r;
  celt_assert2(x > 0, "celt_rcp() only defined for positive values");
```

```
  i = celt_ilog2(x);
  /* n is Q15 with range [0,1). */
  n = (x) - 32768;
  /* Start with a linear approximation: r =
     1.8823529411764706-0.9411764705882353*n. The coefficients and
     the result are Q14 in the range [15420,30840]. */
  r = ((30840) + (((-15420) * (n))));
  /* Perform two Newton iterations: r -= r*((r*n)-1.Q15) =
     r*((r*n)+(r-1.Q15)). */
  r = ((r) - (((r) * (((((r) * (n))) + (((r) + (-32768)))))))));

  /* We subtract an extra 1 in the second iteration to avoid
     overflow; it also neatly compensates for truncation error in
     the rest of the process. */
  r = ((r) -
       (((1) + (((r) * (((((r) * (n))) + (((r) + (-32768)))))))))));

  /* r is now the Q15 solution to 2/(n+1), with a maximum relative
     error of 7.05346E-5, a (relative) RMSE of 2.14418E-5, and a
     peak absolute error of 1.24665/32768. */
  return ((r));
}

#define celt_div(a,b) MULT32_32_Q31((celt_word32)(a),celt_rcp(b))

static inline float
frac_div32(float a, float b)
{
  float           rcp;
  float           result,
                  rem;
  int             shift = 30 - celt_ilog2(b);
  a = (a);
  b = (b);

  /* 16-bit reciprocal */
  rcp = (celt_rcp((b)));
  result = (((rcp) * (a)));
  rem = a - ((result) * (b));
  result += (((rcp) * (rem)));
  return result;
}

#define M1 32767
#define M2 -21
#define M3 -11943
#define M4 4936
```

```
/* Atan approximation using a 4th order polynomial. Input is in Q15
   format and normalized by pi/4. Output is in Q15 format */
static inline float
celt_atan01(float x)
{
  return ((x) *
          (((M1) +
            (((x) *
              (((M2) + (((x) * (((M3) + (((M4) * (x))))))))))))));
}

#undef M1
#undef M2
#undef M3
#undef M4

/* atan2() approximation valid for positive input values */
static inline float
celt_atan2p(float y, float x)
{
  if (y < x)
  {
    float           arg;
    arg = celt_div(((y)), x);
    if (arg >= 32767)
      arg = 32767;
    return (celt_atan01((arg)));
  } else
  {
    float           arg;
    arg = celt_div(((x)), y);
    if (arg >= 32767)
      arg = 32767;
    return 25736 - (celt_atan01((arg)));
  }
}

#endif                          /* FIXED_POINT */

#endif                          /* MATHOPS_H */
```

A.37. os_support.h

```
/* Copyright (C) 2007 Jean-Marc Valin

   File: os_support.h This is the (tiny) OS abstraction layer. Aside
   from math.h, this is the only place where system headers are
```

```
#ifndef OS_SUPPORT_H
#define OS_SUPPORT_H

#ifdef CUSTOM_SUPPORT
#  include "custom_support.h"
#endif

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

/** CELT wrapper for calloc(). To do your own dynamic allocation, all y\
ou need to do is replace this function, celt_realloc and celt_free
    NOTE: celt_alloc needs to CLEAR THE MEMORY */
#ifndef OVERRIDE_CELT_ALLOC
static inline void *
celt_alloc(int size)
```

```
{
  /* WARNING: this is not equivalent to malloc(). If you want to use
     malloc() or your own allocator, YOU NEED TO CLEAR THE MEMORY
     ALLOCATED. Otherwise you will experience strange bugs */
  return calloc(size, 1);
}
#endif

/** Same as celt_alloc(), except that the area is only needed inside a \
CELT call (might cause problem with wideband though) */
#ifndef OVERRIDE_CELT_ALLOC_SCRATCH
static inline void *
celt_alloc_scratch(int size)
{
  /* Scratch space doesn't need to be cleared */
  return calloc(size, 1);
}
#endif

/** CELT wrapper for realloc(). To do your own dynamic allocation, all \
you need to do is replace this function, celt_alloc and celt_free */
#ifndef OVERRIDE_CELT_REALLOC
static inline void *
celt_realloc(void *ptr, int size)
{
  return realloc(ptr, size);
}
#endif

/** CELT wrapper for free(). To do your own dynamic allocation, all you\
 need to do is replace this function, celt_realloc and celt_alloc */
#ifndef OVERRIDE_CELT_FREE
static inline void
celt_free(void *ptr)
{
  free(ptr);
}
#endif

/** Same as celt_free(), except that the area is only needed inside a C\
ELT call (might cause problem with wideband though) */
#ifndef OVERRIDE_CELT_FREE_SCRATCH
static inline void
celt_free_scratch(void *ptr)
{
  free(ptr);
}
#endif
```

```
/** Copy n bytes of memory from src to dst. The 0* term provides compil\
e-time type checking  */
#ifndef OVERRIDE_CELT_COPY
#define CELT_COPY(dst, src, n) (memcpy((dst), (src), (n)*sizeof(*(dst))\
 + 0*((dst)-(src)) ))
#endif

/** Copy n bytes of memory from src to dst, allowing overlapping region\
s. The 0* term
    provides compile-time type checking */
#ifndef OVERRIDE_CELT_MOVE
#define CELT_MOVE(dst, src, n) (memmove((dst), (src), (n)*sizeof(*(dst)\
) + 0*((dst)-(src)) ))
#endif

/** Set n bytes of memory to value of c, starting at address s */
#ifndef OVERRIDE_CELT_MEMSET
#define CELT_MEMSET(dst, c, n) (memset((dst), (c), (n)*sizeof(*(dst))))
#endif

#ifndef OVERRIDE_CELT_FATAL
static inline void
_celt_fatal(const char *str, const char *file, int line)
{
  fprintf(stderr, "Fatal (internal) error in %s, line %d: %s\n",
          file, line, str);
  abort();
}
#endif

#ifndef OVERRIDE_CELT_WARNING
static inline void
celt_warning(const char *str)
{
#ifndef DISABLE_WARNINGS
  fprintf(stderr, "warning: %s\n", str);
#endif
}
#endif

#ifndef OVERRIDE_CELT_WARNING_INT
static inline void
celt_warning_int(const char *str, int val)
{
#ifndef DISABLE_WARNINGS
  fprintf(stderr, "warning: %s %d\n", str, val);
#endif
}
```

```
#endif

#ifndef OVERRIDE_CELT_NOTIFY
static inline void
celt_notify(const char *str)
{
#ifndef DISABLE_NOTIFICATIONS
  fprintf(stderr, "notification: %s\n", str);
#endif
}
#endif

/* #ifdef __GNUC__ #pragma GCC poison printf sprintf #pragma GCC
   poison malloc free realloc calloc #endif */

#endif                          /* OS_SUPPORT_H */
```

A.38. stack_alloc.h

```
/* Copyright (C) 2002-2003 Jean-Marc Valin Copyright (C) 2007-2009
   Xiph.Org Foundation */
/**
   @file stack_alloc.h
   @brief Temporary memory allocation on stack
*/
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
```

```
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef STACK_ALLOC_H
#define STACK_ALLOC_H

#ifdef USE_ALLOCA
# ifdef WIN32
#  include <malloc.h>
# else
#  ifdef HAVE_ALLOCA_H
#   include <alloca.h>
#  else
#   include <stdlib.h>
#  endif
# endif
#endif

/**
 * @def ALIGN(stack, size)
 *
 * Aligns the stack to a 'size' boundary
 *
 * @param stack Stack
 * @param size  New size boundary
 */

/**
 * @def PUSH(stack, size, type)
 *
 * Allocates 'size' elements of type 'type' on the stack
 *
 * @param stack Stack
 * @param size  Number of elements
 * @param type  Type of element
 */

/**
 * @def VARDECL(var)
 *
 * Declare variable on stack
```

```
 *
 * @param var Variable to declare
 */

/**
 * @def ALLOC(var, size, type)
 *
 * Allocate 'size' elements of 'type' on stack
 *
 * @param var  Name of variable to allocate
 * @param size Number of elements
 * @param type Type of element
 */

#if defined(VAR_ARRAYS)

#define VARDECL(type, var)
#define ALLOC(var, size, type) type var[size]
#define SAVE_STACK
#define RESTORE_STACK
#define ALLOC_STACK

#elif defined(USE_ALLOCA)

#define VARDECL(type, var) type *var
#define ALLOC(var, size, type) var = ((type*)alloca(sizeof(type)*(size)\
))
#define SAVE_STACK
#define RESTORE_STACK
#define ALLOC_STACK

#else

#ifdef CELT_C
char           *global_stack = 0;
#else
extern char    *global_stack;
#endif                          /* CELT_C */

#ifdef ENABLE_VALGRIND

#include <valgrind/memcheck.h>

#ifdef CELT_C
char           *global_stack_top = 0;
#else
extern char    *global_stack_top;
#endif                          /* CELT_C */
```

```
#define ALIGN(stack, size) ((stack) += ((size) - (long)(stack)) & ((siz\
e) - 1))
#define PUSH(stack, size, type) (VALGRIND_MAKE_MEM_NOACCESS(stack, glob\
al_stack_top-stack),ALIGN((stack),sizeof(type)/sizeof(char)),VALGRIND_M\
AKE_MEM_UNDEFINED(stack, ((size)*sizeof(type)/sizeof(char))),(stack)+=(\
2*(size)*sizeof(type)/sizeof(char)),(type*)((stack)-(2*(size)*sizeof(ty\
pe)/sizeof(char))))
#define RESTORE_STACK ((global_stack = _saved_stack),VALGRIND_MAKE_MEM_\
NOACCESS(global_stack, global_stack_top-global_stack))
#define ALLOC_STACK ((global_stack = (global_stack==0) ? ((global_stack\
_top=celt_alloc_scratch(GLOBAL_STACK_SIZE*2)+(GLOBAL_STACK_SIZE*2))-(GL\
OBAL_STACK_SIZE*2)) : global_stack),VALGRIND_MAKE_MEM_NOACCESS(global_s\
tack, global_stack_top-global_stack))

#else

#define ALIGN(stack, size) ((stack) += ((size) - (long)(stack)) & ((siz\
e) - 1))
#define PUSH(stack, size, type) (ALIGN((stack),sizeof(type)/sizeof(char\
)),(stack)+=(size)*(sizeof(type)/sizeof(char)),(type*)((stack)-(size)*(\
sizeof(type)/sizeof(char))))
#define RESTORE_STACK (global_stack = _saved_stack)
#define ALLOC_STACK (global_stack = (global_stack==0) ? celt_alloc_scra\
tch(GLOBAL_STACK_SIZE) : global_stack)

#endif                          /* ENABLE_VALGRIND */

#include "os_support.h"
#define VARDECL(type, var) type *var
#define ALLOC(var, size, type) var = PUSH(global_stack, size, type)
#define SAVE_STACK char *_saved_stack = global_stack;

#endif                          /* VAR_ARRAYS */

#endif                          /* STACK_ALLOC_H */
```

A.39. celt_types.h

```
#ifndef _CELT_TYPES_H
#define _CELT_TYPES_H

typedef short celt_int16;
typedef unsigned short celt_uint16;
typedef int celt_int32;
typedef unsigned int celt_uint32;
typedef long long celt_int64;
typedef unsigned long long celt_uint64;

#endif /* _CELT_TYPES_H */
```

A.40. _kiss_fft_guts.h

```
/*
   Copyright (c) 2003-2004, Mark Borgerding

   All rights reserved.

   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   * Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer. *
   Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution. * Neither the author nor the names of any
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
   TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
   ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE. */

#ifndef KISS_FFT_GUTS_H
```

```
#define KISS_FFT_GUTS_H

#define MIN(a,b) ((a)<(b) ? (a):(b))
#define MAX(a,b) ((a)>(b) ? (a):(b))

/* kiss_fft.h defines kiss_fft_scalar as either short or a float
   type and defines typedef struct { kiss_fft_scalar r;
   kiss_fft_scalar i; }kiss_fft_cpx; */
#include "kiss_fft.h"

#define MAXFACTORS 32
/* e.g. an fft of length 128 has 4 factors as far as kissfft is
   concerned 4*4*4*2 */

struct kiss_fft_state {
  int             nfft;
#ifndef FIXED_POINT
  kiss_fft_scalar scale;
#endif
  int             factors[2 * MAXFACTORS];
  int            *bitrev;
  kiss_twiddle_cpx twiddles[1];
};

/*
   Explanation of macros dealing with complex math:

   C_MUL(m,a,b) : m = a*b C_FIXDIV( c , div ) : if a fixed point
   impl., c /= div. noop otherwise C_SUB( res, a,b) : res = a - b
   C_SUBFROM( res , a) : res -= a C_ADDTO( res , a) : res += a */
#ifdef FIXED_POINT
#include "arch.h"

#ifdef DOUBLE_PRECISION

# define FRACBITS 31
# define SAMPPROD long long
#define SAMP_MAX 2147483647
#ifdef MIXED_PRECISION
#define TWID_MAX 32767
#define TRIG_UPSCALE 1
#else
#define TRIG_UPSCALE 65536
#define TWID_MAX 2147483647
#endif
#define EXT32(a) (a)

#else                           /* DOUBLE_PRECISION */
```

```
# define FRACBITS 15
# define SAMPPROD celt_int32
#define SAMP_MAX 32767
#define TRIG_UPSCALE 1
#define EXT32(a) EXTEND32(a)

#endif                          /* !DOUBLE_PRECISION */

#define SAMP_MIN -SAMP_MAX

#if defined(CHECK_OVERFLOW)
#  define CHECK_OVERFLOW_OP(a,op,b)      if ( (SAMPPROD)(a) op (SAMPPRO\
D)(b) > SAMP_MAX || (SAMPPROD)(a) op (SAMPPROD)(b) < SAMP_MIN ) {      \
   fprintf(stderr,"WARNING:overflow @ " __FILE__ "(%d): (%d " #op" %d) \
= %ld\n",__LINE__,(a),(b),(SAMPPROD)(a) op (SAMPPROD)(b) );  }

#endif

#   define smul(a,b) ( (SAMPPROD)(a)*(b) )
#   define sround( x )  (kiss_fft_scalar)( ( (x) + ((SAMPPROD)1<<(FRACB\
ITS-1)) ) >> FRACBITS )

#ifdef MIXED_PRECISION

#   define S_MUL(a,b) MULT16_32_Q15(b, a)

#   define C_MUL(m,a,b)       do{ (m).r = SUB32(S_MUL((a).r,(b).r) , S_\
MUL((a).i,(b).i));           (m).i = ADD32(S_MUL((a).r,(b).i) , S_MUL((\
a).i,(b).r)); }while(0)

#   define C_MULC(m,a,b)       do{ (m).r = ADD32(S_MUL((a).r,(b).r) , S\
_MUL((a).i,(b).i));           (m).i = SUB32(S_MUL((a).i,(b).r) , S_MUL(\
(a).r,(b).i)); }while(0)

#   define C_MUL4(m,a,b)       do{ (m).r = SHR(SUB32(S_MUL((a).r,(b).r)\
 , S_MUL((a).i,(b).i)),2);           (m).i = SHR(ADD32(S_MUL((a).r,(b).\
i) , S_MUL((a).i,(b).r)),2); }while(0)

#   define C_MULBYSCALAR( c, s )       do{ (c).r =  S_MUL( (c).r , s ) \
;          (c).i =  S_MUL( (c).i , s ) ; }while(0)

#   define DIVSCALAR(x,k)         (x) = S_MUL(  x, (TWID_MAX-((k)>>1))/\
(k)+1 )

#   define C_FIXDIV(c,div)         do {    DIVSCALAR( (c).r , div);    \
             DIVSCALAR( (c).i  , div); }while (0)

#define  C_ADD( res, a,b)    do {(res).r=ADD32((a).r,(b).r);  (res).i=A\
```

```
DD32((a).i,(b).i);     }while(0)

#define  C_SUB( res, a,b)    do {(res).r=SUB32((a).r,(b).r);  (res).i=S\
UB32((a).i,(b).i);     }while(0)

#define C_ADDTO( res , a)    do {(res).r = ADD32((res).r, (a).r);  (res\
).i = ADD32((res).i,(a).i);    }while(0)

#define C_SUBFROM( res , a)    do {(res).r = ADD32((res).r,(a).r);  (re\
s).i = SUB32((res).i,(a).i);     }while(0)

#else                           /* MIXED_PRECISION */
#   define sround4( x )  (kiss_fft_scalar)( ( (x) + ((SAMPPROD)1<<(FRAC\
BITS-1)) ) >> (FRACBITS+2) )

#   define S_MUL(a,b) sround( smul(a,b) )

#   define C_MUL(m,a,b)       do{ (m).r = sround( smul((a).r,(b).r) - s\
mul((a).i,(b).i) );           (m).i = sround( smul((a).r,(b).i) + smul(\
(a).i,(b).r) ); }while(0)

#   define C_MULC(m,a,b)       do{ (m).r = sround( smul((a).r,(b).r) + \
smul((a).i,(b).i) );           (m).i = sround( smul((a).i,(b).r) - smul\
((a).r,(b).i) ); }while(0)

#   define C_MUL4(m,a,b)                do{ (m).r = sround4( smul((a).r\
,(b).r) - smul((a).i,(b).i) );                (m).i = sround4( smul((a)\
.r,(b).i) + smul((a).i,(b).r) ); }while(0)

#   define C_MULBYSCALAR( c, s )               do{ (c).r =  sround( sm\
ul( (c).r , s ) ) ;              (c).i =  sround( smul( (c).i , s ) ) \
; }while(0)

#   define DIVSCALAR(x,k)     (x) = sround( smul(  x, SAMP_MAX/k ) )

#   define C_FIXDIV(c,div)     do {    DIVSCALAR( (c).r , div);        \
  DIVSCALAR( (c).i  , div); }while (0)

#endif                          /* !MIXED_PRECISION */

#else                           /* not FIXED_POINT */

#define EXT32(a) (a)

#   define S_MUL(a,b) ( (a)*(b) )
#define C_MUL(m,a,b)     do{ (m).r = (a).r*(b).r - (a).i*(b).i;        \
(m).i = (a).r*(b).i + (a).i*(b).r; }while(0)
```

```
#define C_MULC(m,a,b)     do{ (m).r = (a).r*(b).r + (a).i*(b).i;       \
 (m).i = (a).i*(b).r - (a).r*(b).i; }while(0)

#define C_MUL4(m,a,b) C_MUL(m,a,b)

#   define C_FIXDIV(c,div)      /* NOOP */
#   define C_MULBYSCALAR( c, s )     do{ (c).r *= (s);        (c).i *= \
(s); }while(0)

#endif

#ifndef CHECK_OVERFLOW_OP
#  define CHECK_OVERFLOW_OP(a,op,b)     /* noop */
#endif

#ifndef C_ADD
#define  C_ADD( res, a,b)    do {         CHECK_OVERFLOW_OP((a).r,+,(b)\
.r)        CHECK_OVERFLOW_OP((a).i,+,(b).i)        (res).r=(a).r+(b).r;\
  (res).i=(a).i+(b).i;     }while(0)

#define  C_SUB( res, a,b)    do {         CHECK_OVERFLOW_OP((a).r,-,(b)\
.r)        CHECK_OVERFLOW_OP((a).i,-,(b).i)        (res).r=(a).r-(b).r;\
  (res).i=(a).i-(b).i;     }while(0)

#define C_ADDTO( res , a)    do {         CHECK_OVERFLOW_OP((res).r,+,(\
a).r)        CHECK_OVERFLOW_OP((res).i,+,(a).i)        (res).r += (a).r\
;  (res).i += (a).i;    }while(0)

#define C_SUBFROM( res , a)    do {        CHECK_OVERFLOW_OP((res).r,-,\
(a).r)        CHECK_OVERFLOW_OP((res).i,-,(a).i)        (res).r -= (a).\
r;  (res).i -= (a).i;     }while(0)

#endif                           /* C_ADD defined */

#ifdef FIXED_POINT
/* # define KISS_FFT_COS(phase)
   TRIG_UPSCALE*floor(MIN(32767,MAX(-32767,.5+32768 * cos (phase))))
   # define KISS_FFT_SIN(phase)
   TRIG_UPSCALE*floor(MIN(32767,MAX(-32767,.5+32768 * sin (phase)))) */
#  define KISS_FFT_COS(phase)  floor(.5+TWID_MAX*cos (phase))
#  define KISS_FFT_SIN(phase)  floor(.5+TWID_MAX*sin (phase))
#  define HALF_OF(x) ((x)>>1)
#elif defined(USE_SIMD)
#  define KISS_FFT_COS(phase) _mm_set1_ps( cos(phase) )
#  define KISS_FFT_SIN(phase) _mm_set1_ps( sin(phase) )
#  define HALF_OF(x) ((x)*_mm_set1_ps(.5f))
#else
#  define KISS_FFT_COS(phase) (kiss_fft_scalar) cos(phase)
```

```
#  define KISS_FFT_SIN(phase) (kiss_fft_scalar) sin(phase)
#  define HALF_OF(x) ((x)*.5f)
#endif

#define  kf_cexp(x,phase)     do{         (x)->r = KISS_FFT_COS(phase);\
        (x)->i = KISS_FFT_SIN(phase);    }while(0)

#define  kf_cexp2(x,phase)    do{       (x)->r = TRIG_UPSCALE*celt_cos_\
norm((phase));      (x)->i = TRIG_UPSCALE*celt_cos_norm((phase)-32768);\
}while(0)

#endif                          /* KISS_FFT_GUTS_H */
```

A.41. kiss_fft.h

```
/*
   Copyright (c) 2003-2004, Mark Borgerding Lots of modifications by
   Jean-Marc Valin Copyright (c) 2005-2007, Xiph.Org Foundation
   Copyright (c) 2008, Xiph.Org Foundation, CSIRO

   All rights reserved.

   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   * Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer. *
   Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution. * Neither the author nor the names of any
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
   TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
   ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
```

```
   POSSIBILITY OF SUCH DAMAGE. */

#ifndef KISS_FFT_H
#define KISS_FFT_H

#include <stdlib.h>
#include <math.h>
#include "arch.h"

#ifdef __cplusplus
extern          "C" {
#endif

  /*
     ATTENTION! If you would like a : -- a utility that will handle
     the caching of fft objects -- real-only (no imaginary time
     component ) FFT -- a multi-dimensional FFT -- a command-line
     utility to perform ffts -- a command-line utility to perform
     fast-convolution filtering

     Then see kfc.h kiss_fftr.h kiss_fftnd.h fftutil.c
     kiss_fastfir.c in the tools/ directory. */

#ifdef USE_SIMD
# include <xmmintrin.h>
# define kiss_fft_scalar __m128
#define KISS_FFT_MALLOC(nbytes) memalign(16,nbytes)
#else
#define KISS_FFT_MALLOC celt_alloc
#endif

#ifdef FIXED_POINT
#include "arch.h"
#ifdef DOUBLE_PRECISION
#  define kiss_fft_scalar celt_int32
#  define kiss_twiddle_scalar celt_int32
#  define KF_SUFFIX _celt_double
#else
#  define kiss_fft_scalar celt_int16
#  define kiss_twiddle_scalar celt_int16
#  define KF_SUFFIX _celt_single
#endif
#else
# ifndef kiss_fft_scalar
  /* default is float */
#   define kiss_fft_scalar float
#   define kiss_twiddle_scalar float
#   define KF_SUFFIX _celt_single
```

```
# endif
#endif

  /* This adds a suffix to all the kiss_fft functions so we can
     easily link with more than one copy of the fft */
#define CAT_SUFFIX(a,b) a ## b
#define SUF(a,b) CAT_SUFFIX(a, b)

#define kiss_fft_alloc SUF(kiss_fft_alloc,KF_SUFFIX)
#define kf_work SUF(kf_work,KF_SUFFIX)
#define ki_work SUF(ki_work,KF_SUFFIX)
#define kiss_fft SUF(kiss_fft,KF_SUFFIX)
#define kiss_ifft SUF(kiss_ifft,KF_SUFFIX)
#define kiss_fft_stride SUF(kiss_fft_stride,KF_SUFFIX)
#define kiss_ifft_stride SUF(kiss_ifft_stride,KF_SUFFIX)

  typedef struct {
    kiss_fft_scalar r;
    kiss_fft_scalar i;
  } kiss_fft_cpx;

  typedef struct {
    kiss_twiddle_scalar r;
    kiss_twiddle_scalar i;
  } kiss_twiddle_cpx;

  typedef struct kiss_fft_state *kiss_fft_cfg;

/**
 *  kiss_fft_alloc
 *
 *  Initialize a FFT (or IFFT) algorithm's cfg/state buffer.
 *
 *  typical usage:      kiss_fft_cfg mycfg=kiss_fft_alloc(1024,0,NULL,N\
ULL);
 *
 *  The return value from fft_alloc is a cfg buffer used internally
 *  by the fft routine or NULL.
 *
 *  If lenmem is NULL, then kiss_fft_alloc will allocate a cfg buffer u\
sing malloc.
 *  The returned value should be free()d when done to avoid memory leak\
s.
 *
 *  The state can be placed in a user supplied buffer 'mem':
 *  If lenmem is not NULL and mem is not NULL and *lenmem is large enou\
gh,
 *      then the function places the cfg in mem and the size used in *l\
```

```
enmem
 *      and returns mem.
 *
 *  If lenmem is not NULL and ( mem is NULL or *lenmem is not large eno\
ugh),
 *      then the function returns NULL and places the minimum cfg
 *      buffer size in *lenmem.
 * */

  kiss_fft_cfg    kiss_fft_alloc(int nfft, void *mem,
                                 size_t * lenmem);

  void            kf_work(kiss_fft_cpx * Fout,
                          const kiss_fft_cpx * f,
                          const size_t fstride, int in_stride,
                          int *factors, const kiss_fft_cfg st, int N,
                          int s2, int m2);

/** Internal function. Can be useful when you want to do the bit-revers\
ing yourself */
  void            ki_work(kiss_fft_cpx * Fout,
                          const kiss_fft_cpx * f,
                          const size_t fstride, int in_stride,
                          int *factors, const kiss_fft_cfg st, int N,
                          int s2, int m2);

/**
 * kiss_fft(cfg,in_out_buf)
 *
 * Perform an FFT on a complex input buffer.
 * for a forward FFT,
 * fin should be  f[0] , f[1] , ... ,f[nfft-1]
 * fout will be   F[0] , F[1] , ... ,F[nfft-1]
 * Note that each element is complex and can be accessed like
    f[k].r and f[k].i
 * */
  void            kiss_fft(kiss_fft_cfg cfg,
                           const kiss_fft_cpx * fin,
                           kiss_fft_cpx * fout);
  void            kiss_ifft(kiss_fft_cfg cfg,
                            const kiss_fft_cpx * fin,
                            kiss_fft_cpx * fout);

/**
 A more generic version of the above function. It reads its input from \
every Nth sample.
 * */
  void            kiss_fft_stride(kiss_fft_cfg cfg,
```

```
                                  const kiss_fft_cpx * fin,
                                  kiss_fft_cpx * fout,
                                  int fin_stride);
  void            kiss_ifft_stride(kiss_fft_cfg cfg,
                                   const kiss_fft_cpx * fin,
                                   kiss_fft_cpx * fout,
                                   int fin_stride);

/** If kiss_fft_alloc allocated a buffer, it is one contiguous
   buffer and can be simply free()d when no longer needed*/
#define kiss_fft_free celt_free

#ifdef __cplusplus
}
#endif
#endif
```

A.42. kiss_fft.c

```
/*
   Copyright (c) 2003-2004, Mark Borgerding Lots of modifications by
   Jean-Marc Valin Copyright (c) 2005-2007, Xiph.Org Foundation
   Copyright (c) 2008, Xiph.Org Foundation, CSIRO

   All rights reserved.

   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   * Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer. *
   Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution. * Neither the author nor the names of any
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
   TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

```
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
   ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE. */

/* config.h.  Generated from config.h.in by configure.  */
/* config.h.in.  Generated from configure.ac by autoheader.  */

/* This is a build of CELT */

/* Version extra */

/* Version major */

/* Version micro */

/* Version minor */

/* Complete version string */

#include "_kiss_fft_guts.h"
#include "arch.h"
#include "os_support.h"
#include "mathops.h"
#include "stack_alloc.h"

/* The guts header contains all the multiplication and addition
   macros that are defined for complex numbers.  It also delares the
   kf_ internal functions. */

static void
kf_bfly2(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, int m, int N, int mm)
{
  kiss_fft_cpx   *Fout2;
  kiss_twiddle_cpx *tw1;
  int             i,
                  j;
  kiss_fft_cpx   *Fout_beg = Fout;
  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    Fout2 = Fout + m;
    tw1 = st->twiddles;
    for (j = 0; j < m; j++)
    {
```

```
      kiss_fft_cpx    t;
      Fout->r = (Fout->r);
      Fout->i = (Fout->i);
      Fout2->r = (Fout2->r);
      Fout2->i = (Fout2->i);
      C_MUL(t, *Fout2, *tw1);
      tw1 += fstride;
      C_SUB(*Fout2, *Fout, t);
      C_ADDTO(*Fout, t);
      ++Fout2;
      ++Fout;
    }
  }
}

static void
ki_bfly2(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, int m, int N, int mm)
{
  kiss_fft_cpx   *Fout2;
  kiss_twiddle_cpx *tw1;
  kiss_fft_cpx    t;
  int             i,
                  j;
  kiss_fft_cpx   *Fout_beg = Fout;
  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    Fout2 = Fout + m;
    tw1 = st->twiddles;
    for (j = 0; j < m; j++)
    {
      C_MULC(t, *Fout2, *tw1);
      tw1 += fstride;
      C_SUB(*Fout2, *Fout, t);
      C_ADDTO(*Fout, t);
      ++Fout2;
      ++Fout;
    }
  }
}

static void
kf_bfly4(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, int m, int N, int mm)
{
```

```
  kiss_twiddle_cpx *tw1,
                  *tw2,
                  *tw3;
  kiss_fft_cpx    scratch[6];
  const size_t    m2 = 2 * m;
  const size_t    m3 = 3 * m;
  int             i,
                  j;

  kiss_fft_cpx   *Fout_beg = Fout;
  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    tw3 = tw2 = tw1 = st->twiddles;
    for (j = 0; j < m; j++)
    {
      C_MUL4(scratch[0], Fout[m], *tw1);
      C_MUL4(scratch[1], Fout[m2], *tw2);
      C_MUL4(scratch[2], Fout[m3], *tw3);

      Fout->r = (Fout->r);
      Fout->i = (Fout->i);
      C_SUB(scratch[5], *Fout, scratch[1]);
      C_ADDTO(*Fout, scratch[1]);
      C_ADD(scratch[3], scratch[0], scratch[2]);
      C_SUB(scratch[4], scratch[0], scratch[2]);
      Fout[m2].r = (Fout[m2].r);
      Fout[m2].i = (Fout[m2].i);
      C_SUB(Fout[m2], *Fout, scratch[3]);
      tw1 += fstride;
      tw2 += fstride * 2;
      tw3 += fstride * 3;
      C_ADDTO(*Fout, scratch[3]);

      Fout[m].r = scratch[5].r + scratch[4].i;
      Fout[m].i = scratch[5].i - scratch[4].r;
      Fout[m3].r = scratch[5].r - scratch[4].i;
      Fout[m3].i = scratch[5].i + scratch[4].r;
      ++Fout;
    }
  }
}

static void
ki_bfly4(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, int m, int N, int mm)
{
```

```
  kiss_twiddle_cpx *tw1,
                  *tw2,
                  *tw3;
  kiss_fft_cpx    scratch[6];
  const size_t    m2 = 2 * m;
  const size_t    m3 = 3 * m;
  int             i,
                  j;

  kiss_fft_cpx   *Fout_beg = Fout;
  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    tw3 = tw2 = tw1 = st->twiddles;
    for (j = 0; j < m; j++)
    {
      C_MULC(scratch[0], Fout[m], *tw1);
      C_MULC(scratch[1], Fout[m2], *tw2);
      C_MULC(scratch[2], Fout[m3], *tw3);

      C_SUB(scratch[5], *Fout, scratch[1]);
      C_ADDTO(*Fout, scratch[1]);
      C_ADD(scratch[3], scratch[0], scratch[2]);
      C_SUB(scratch[4], scratch[0], scratch[2]);
      C_SUB(Fout[m2], *Fout, scratch[3]);
      tw1 += fstride;
      tw2 += fstride * 2;
      tw3 += fstride * 3;
      C_ADDTO(*Fout, scratch[3]);

      Fout[m].r = scratch[5].r - scratch[4].i;
      Fout[m].i = scratch[5].i + scratch[4].r;
      Fout[m3].r = scratch[5].r + scratch[4].i;
      Fout[m3].i = scratch[5].i - scratch[4].r;
      ++Fout;
    }
  }
}

static void
kf_bfly3(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, int m, int N, int mm)
{
  int             i;
  size_t          k;
  const size_t    m2 = 2 * m;
  kiss_twiddle_cpx *tw1,
```

```
                *tw2;
  kiss_fft_cpx    scratch[5];
  kiss_twiddle_cpx epi3;

  kiss_fft_cpx   *Fout_beg = Fout;
  epi3 = st->twiddles[fstride * m];
  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    tw1 = tw2 = st->twiddles;
    k = m;
    do
    {
      C_FIXDIV(*Fout, 3);
      C_FIXDIV(Fout[m], 3);
      C_FIXDIV(Fout[m2], 3);

      C_MUL(scratch[1], Fout[m], *tw1);
      C_MUL(scratch[2], Fout[m2], *tw2);

      C_ADD(scratch[3], scratch[1], scratch[2]);
      C_SUB(scratch[0], scratch[1], scratch[2]);
      tw1 += fstride;
      tw2 += fstride * 2;

      Fout[m].r = Fout->r - HALF_OF(scratch[3].r);
      Fout[m].i = Fout->i - HALF_OF(scratch[3].i);

      C_MULBYSCALAR(scratch[0], epi3.i);

      C_ADDTO(*Fout, scratch[3]);

      Fout[m2].r = Fout[m].r + scratch[0].i;
      Fout[m2].i = Fout[m].i - scratch[0].r;

      Fout[m].r -= scratch[0].i;
      Fout[m].i += scratch[0].r;

      ++Fout;
    }
    while (--k);
  }
}

static void
ki_bfly3(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, size_t m, int N, int mm)
```

```
{
  size_t          i,
                  k;
  const size_t    m2 = 2 * m;
  kiss_twiddle_cpx *tw1,
                  *tw2;
  kiss_fft_cpx    scratch[5];
  kiss_twiddle_cpx epi3;

  kiss_fft_cpx   *Fout_beg = Fout;
  epi3 = st->twiddles[fstride * m];
  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    tw1 = tw2 = st->twiddles;
    k = m;
    do
    {

      C_MULC(scratch[1], Fout[m], *tw1);
      C_MULC(scratch[2], Fout[m2], *tw2);

      C_ADD(scratch[3], scratch[1], scratch[2]);
      C_SUB(scratch[0], scratch[1], scratch[2]);
      tw1 += fstride;
      tw2 += fstride * 2;

      Fout[m].r = Fout->r - HALF_OF(scratch[3].r);
      Fout[m].i = Fout->i - HALF_OF(scratch[3].i);

      C_MULBYSCALAR(scratch[0], -epi3.i);

      C_ADDTO(*Fout, scratch[3]);

      Fout[m2].r = Fout[m].r + scratch[0].i;
      Fout[m2].i = Fout[m].i - scratch[0].r;

      Fout[m].r -= scratch[0].i;
      Fout[m].i += scratch[0].r;

      ++Fout;
    }
    while (--k);
  }
}

static void
kf_bfly5(kiss_fft_cpx * Fout,
```

```
        const size_t fstride,
        const kiss_fft_cfg st, int m, int N, int mm)
{
  kiss_fft_cpx   *Fout0,
                 *Fout1,
                 *Fout2,
                 *Fout3,
                 *Fout4;
  int             i,
                  u;
  kiss_fft_cpx    scratch[13];
  kiss_twiddle_cpx *twiddles = st->twiddles;
  kiss_twiddle_cpx *tw;
  kiss_twiddle_cpx ya,
                  yb;
  kiss_fft_cpx   *Fout_beg = Fout;

  ya = twiddles[fstride * m];
  yb = twiddles[fstride * 2 * m];
  tw = st->twiddles;

  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    Fout0 = Fout;
    Fout1 = Fout0 + m;
    Fout2 = Fout0 + 2 * m;
    Fout3 = Fout0 + 3 * m;
    Fout4 = Fout0 + 4 * m;

    for (u = 0; u < m; ++u)
    {
      C_FIXDIV(*Fout0, 5);
      C_FIXDIV(*Fout1, 5);
      C_FIXDIV(*Fout2, 5);
      C_FIXDIV(*Fout3, 5);
      C_FIXDIV(*Fout4, 5);
      scratch[0] = *Fout0;

      C_MUL(scratch[1], *Fout1, tw[u * fstride]);
      C_MUL(scratch[2], *Fout2, tw[2 * u * fstride]);
      C_MUL(scratch[3], *Fout3, tw[3 * u * fstride]);
      C_MUL(scratch[4], *Fout4, tw[4 * u * fstride]);

      C_ADD(scratch[7], scratch[1], scratch[4]);
      C_SUB(scratch[10], scratch[1], scratch[4]);
      C_ADD(scratch[8], scratch[2], scratch[3]);
      C_SUB(scratch[9], scratch[2], scratch[3]);
```

```
      Fout0->r += scratch[7].r + scratch[8].r;
      Fout0->i += scratch[7].i + scratch[8].i;

      scratch[5].r =
          scratch[0].r + S_MUL(scratch[7].r,
                               ya.r) + S_MUL(scratch[8].r, yb.r);
      scratch[5].i =
          scratch[0].i + S_MUL(scratch[7].i,
                               ya.r) + S_MUL(scratch[8].i, yb.r);

      scratch[6].r =
          S_MUL(scratch[10].i, ya.i) + S_MUL(scratch[9].i, yb.i);
      scratch[6].i =
          -S_MUL(scratch[10].r, ya.i) - S_MUL(scratch[9].r, yb.i);

      C_SUB(*Fout1, scratch[5], scratch[6]);
      C_ADD(*Fout4, scratch[5], scratch[6]);

      scratch[11].r =
          scratch[0].r + S_MUL(scratch[7].r,
                               yb.r) + S_MUL(scratch[8].r, ya.r);
      scratch[11].i =
          scratch[0].i + S_MUL(scratch[7].i,
                               yb.r) + S_MUL(scratch[8].i, ya.r);
      scratch[12].r =
          -S_MUL(scratch[10].i, yb.i) + S_MUL(scratch[9].i, ya.i);
      scratch[12].i =
          S_MUL(scratch[10].r, yb.i) - S_MUL(scratch[9].r, ya.i);

      C_ADD(*Fout2, scratch[11], scratch[12]);
      C_SUB(*Fout3, scratch[11], scratch[12]);

      ++Fout0;
      ++Fout1;
      ++Fout2;
      ++Fout3;
      ++Fout4;
    }
  }
}

static void
ki_bfly5(kiss_fft_cpx * Fout,
         const size_t fstride,
         const kiss_fft_cfg st, int m, int N, int mm)
{
  kiss_fft_cpx   *Fout0,
                 *Fout1,
```

```
                *Fout2,
                *Fout3,
                *Fout4;
  int            i,
                 u;
  kiss_fft_cpx    scratch[13];
  kiss_twiddle_cpx *twiddles = st->twiddles;
  kiss_twiddle_cpx *tw;
  kiss_twiddle_cpx ya,
                 yb;
  kiss_fft_cpx   *Fout_beg = Fout;

  ya = twiddles[fstride * m];
  yb = twiddles[fstride * 2 * m];
  tw = st->twiddles;

  for (i = 0; i < N; i++)
  {
    Fout = Fout_beg + i * mm;
    Fout0 = Fout;
    Fout1 = Fout0 + m;
    Fout2 = Fout0 + 2 * m;
    Fout3 = Fout0 + 3 * m;
    Fout4 = Fout0 + 4 * m;

    for (u = 0; u < m; ++u)
    {
      scratch[0] = *Fout0;

      C_MULC(scratch[1], *Fout1, tw[u * fstride]);
      C_MULC(scratch[2], *Fout2, tw[2 * u * fstride]);
      C_MULC(scratch[3], *Fout3, tw[3 * u * fstride]);
      C_MULC(scratch[4], *Fout4, tw[4 * u * fstride]);

      C_ADD(scratch[7], scratch[1], scratch[4]);
      C_SUB(scratch[10], scratch[1], scratch[4]);
      C_ADD(scratch[8], scratch[2], scratch[3]);
      C_SUB(scratch[9], scratch[2], scratch[3]);

      Fout0->r += scratch[7].r + scratch[8].r;
      Fout0->i += scratch[7].i + scratch[8].i;

      scratch[5].r =
          scratch[0].r + S_MUL(scratch[7].r,
                               ya.r) + S_MUL(scratch[8].r, yb.r);
      scratch[5].i =
          scratch[0].i + S_MUL(scratch[7].i,
                               ya.r) + S_MUL(scratch[8].i, yb.r);
```

```
      scratch[6].r =
          -S_MUL(scratch[10].i, ya.i) - S_MUL(scratch[9].i, yb.i);
      scratch[6].i =
          S_MUL(scratch[10].r, ya.i) + S_MUL(scratch[9].r, yb.i);

      C_SUB(*Fout1, scratch[5], scratch[6]);
      C_ADD(*Fout4, scratch[5], scratch[6]);

      scratch[11].r =
          scratch[0].r + S_MUL(scratch[7].r,
                               yb.r) + S_MUL(scratch[8].r, ya.r);
      scratch[11].i =
          scratch[0].i + S_MUL(scratch[7].i,
                               yb.r) + S_MUL(scratch[8].i, ya.r);
      scratch[12].r =
          S_MUL(scratch[10].i, yb.i) - S_MUL(scratch[9].i, ya.i);
      scratch[12].i =
          -S_MUL(scratch[10].r, yb.i) + S_MUL(scratch[9].r, ya.i);

      C_ADD(*Fout2, scratch[11], scratch[12]);
      C_SUB(*Fout3, scratch[11], scratch[12]);

      ++Fout0;
      ++Fout1;
      ++Fout2;
      ++Fout3;
      ++Fout4;
    }
  }
}

static
    void
compute_bitrev_table(int Fout,
                     int *f,
                     const size_t fstride,
                     int in_stride,
                     int *factors, const kiss_fft_cfg st)
{
  const int       p = *factors++;       /* the radix */
  const int       m = *factors++;       /* stage's fft length/p */

  /* printf ("fft %d %d %d %d %d %d\n", p*m, m, p, s2,
     fstride*in_stride, N); */
  if (m == 1)
  {
    int             j;
    for (j = 0; j < p; j++)
```

```
    {
      *f = Fout + j;
      f += fstride * in_stride;
    }
  } else
  {
    int             j;
    for (j = 0; j < p; j++)
    {
      compute_bitrev_table(Fout, f, fstride * p, in_stride, factors,
                           st);
      f += fstride * in_stride;
      Fout += m;
    }
  }
}

void
kf_work(kiss_fft_cpx * Fout,
        const kiss_fft_cpx * f,
        const size_t fstride,
        int in_stride,
        int *factors, const kiss_fft_cfg st, int N, int s2, int m2)
{
  const int       p = *factors++;       /* the radix */
  const int       m = *factors++;       /* stage's fft length/p */
  /* printf ("fft %d %d %d %d %d %d %d\n", p*m, m, p, s2,
     fstride*in_stride, N, m2); */
  if (m != 1)
    kf_work(Fout, f, fstride * p, in_stride, factors, st, N * p,
            fstride * in_stride, m);

  switch (p)
  {
  case 2:
    kf_bfly2(Fout, fstride, st, m, N, m2);
    break;
  case 4:
    kf_bfly4(Fout, fstride, st, m, N, m2);
    break;

  case 3:
    kf_bfly3(Fout, fstride, st, m, N, m2);
    break;
  case 5:
    kf_bfly5(Fout, fstride, st, m, N, m2);
    break;
```

```
  }
}

void
ki_work(kiss_fft_cpx * Fout,
        const kiss_fft_cpx * f,
        const size_t fstride,
        int in_stride,
        int *factors, const kiss_fft_cfg st, int N, int s2, int m2)
{
  const int       p = *factors++;       /* the radix */
  const int       m = *factors++;       /* stage's fft length/p */
  /* printf ("fft %d %d %d %d %d %d %d\n", p*m, m, p, s2,
     fstride*in_stride, N, m2); */
  if (m != 1)
    ki_work(Fout, f, fstride * p, in_stride, factors, st, N * p,
            fstride * in_stride, m);

  switch (p)
  {
  case 2:
    ki_bfly2(Fout, fstride, st, m, N, m2);
    break;
  case 4:
    ki_bfly4(Fout, fstride, st, m, N, m2);
    break;

  case 3:
    ki_bfly3(Fout, fstride, st, m, N, m2);
    break;
  case 5:
    ki_bfly5(Fout, fstride, st, m, N, m2);
    break;

  }
}

/* facbuf is populated by p1,m1,p2,m2, ... where p[i] * m[i] =
   m[i-1] m0 = n */
static
    int
kf_factor(int n, int *facbuf)
{
  int             p = 4;

  /* factor out powers of 4, powers of 2, then any remaining primes */
  do
  {
```

```
    while (n % p)
    {
      switch (p)
      {
      case 4:
        p = 2;
        break;
      case 2:
        p = 3;
        break;
      default:
        p += 2;
        break;
      }
      if (p > 32000 || (celt_int32) p * (celt_int32) p > n)
        p = n;                  /* no more factors, skip to end */
    }
    n /= p;
    if (p > 5)
    {
      celt_warning("Only powers of 2, 3 and 5 are supported");
      return 0;
    }
    *facbuf++ = p;
    *facbuf++ = n;
  }
  while (n > 1);
  return 1;
}
/*
 *
 * User-callable function to allocate all necessary storage space for t\
he fft.
 *
 * The return value is a contiguous block of memory, allocated with mal\
loc.  As such,
 * It can be freed with free(), rather than a kiss_fft-specific functio\
n.
 * */
kiss_fft_cfg
kiss_fft_alloc(int nfft, void *mem, size_t * lenmem)
{
  kiss_fft_cfg    st = NULL;
  size_t          memneeded = sizeof(struct kiss_fft_state) + sizeof(ki\
ss_twiddle_cpx) * (nfft - 1) + sizeof(int) * nfft;       /* twiddle
                                                                      \
                                                          factors */
```

```
  if (lenmem == NULL)
  {
    st = (kiss_fft_cfg) KISS_FFT_MALLOC(memneeded);
  } else
  {
    if (mem != NULL && *lenmem >= memneeded)
      st = (kiss_fft_cfg) mem;
    *lenmem = memneeded;
  }
  if (st)
  {
    int             i;
    st->nfft = nfft;

    st->scale = 1. / nfft;

    for (i = 0; i < nfft; ++i)
    {
      const double    pi = 3.14159265358979323846264338327;
      double          phase = (-2 * pi / nfft) * i;
      kf_cexp(st->twiddles + i, phase);
    }

    if (!kf_factor(nfft, st->factors))
    {
      kiss_fft_free(st);
      return NULL;
    }

    /* bitrev */
    st->bitrev =
        (int *) ((char *) st + memneeded - sizeof(int) * nfft);
    compute_bitrev_table(0, st->bitrev, 1, 1, st->factors, st);
  }
  return st;
}

void
kiss_fft_stride(kiss_fft_cfg st, const kiss_fft_cpx * fin,
                kiss_fft_cpx * fout, int in_stride)
{
  if (fin == fout)
  {
    celt_fatal("In-place FFT not supported");
  } else
  {
    /* Bit-reverse the input */
    int             i;
```

```
    for (i = 0; i < st->nfft; i++)
    {
      fout[st->bitrev[i]] = fin[i];

      fout[st->bitrev[i]].r *= st->scale;
      fout[st->bitrev[i]].i *= st->scale;

    }
    kf_work(fout, fin, 1, in_stride, st->factors, st, 1, in_stride,
            1);
  }
}

void
kiss_fft(kiss_fft_cfg cfg, const kiss_fft_cpx * fin,
         kiss_fft_cpx * fout)
{
  kiss_fft_stride(cfg, fin, fout, 1);
}

void
kiss_ifft_stride(kiss_fft_cfg st, const kiss_fft_cpx * fin,
                 kiss_fft_cpx * fout, int in_stride)
{
  if (fin == fout)
  {
    celt_fatal("In-place FFT not supported");
  } else
  {
    /* Bit-reverse the input */
    int             i;
    for (i = 0; i < st->nfft; i++)
      fout[st->bitrev[i]] = fin[i];
    ki_work(fout, fin, 1, in_stride, st->factors, st, 1, in_stride,
            1);
  }
}

void
kiss_ifft(kiss_fft_cfg cfg, const kiss_fft_cpx * fin,
          kiss_fft_cpx * fout)
{
  kiss_ifft_stride(cfg, fin, fout, 1);
}
```

A.43. kiss_fftr.h

```
/*
   Original version: Copyright (c) 2003-2004, Mark Borgerding
   Followed by heavy modifications: Copyright (c) 2007-2008,
   Jean-Marc Valin

   All rights reserved.

   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   * Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer. *
   Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution. * Neither the author nor the names of any
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
   TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
   ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE. */

#ifndef KISS_FTR_H
#define KISS_FTR_H

#include "kiss_fft.h"
#ifdef __cplusplus
extern          "C" {
#endif

#define kiss_fftr_alloc SUF(kiss_fftr_alloc,KF_SUFFIX)
#define kiss_fftr_inplace SUF(kiss_fftr_inplace,KF_SUFFIX)
#define kiss_fftr_alloc SUF(kiss_fftr_alloc,KF_SUFFIX)
```

```
   #define kiss_fftr_twiddles SUF(kiss_fftr_twiddles,KF_SUFFIX)
   #define kiss_fftr SUF(kiss_fftr,KF_SUFFIX)
   #define kiss_fftri SUF(kiss_fftri,KF_SUFFIX)

     /*

        Real optimized version can save about 45% cpu time vs. complex
        fft of a real seq.

      */

     struct kiss_fftr_state {
       kiss_fft_cfg    substate;
       kiss_twiddle_cpx *super_twiddles;
   #ifdef USE_SIMD
       long            pad;
   #endif
     };

     typedef struct kiss_fftr_state *kiss_fftr_cfg;

     kiss_fftr_cfg   kiss_fftr_alloc(int nfft, void *mem,
                                     size_t * lenmem);
     /*
        nfft must be even

        If you don't care to allocate space, use mem = lenmem = NULL */

     /*
        input timedata has nfft scalar points output freqdata has
        nfft/2+1 complex points, packed into nfft scalar points */
     void            kiss_fftr_twiddles(kiss_fftr_cfg st,
                                        kiss_fft_scalar * freqdata);

     void            kiss_fftr(kiss_fftr_cfg st,
                               const kiss_fft_scalar * timedata,
                               kiss_fft_scalar * freqdata);
     void            kiss_fftr_inplace(kiss_fftr_cfg st,
                                       kiss_fft_scalar * X);

     void            kiss_fftri(kiss_fftr_cfg st,
                                const kiss_fft_scalar * freqdata,
                                kiss_fft_scalar * timedata);

     /*
        input freqdata has nfft/2+1 complex points, packed into nfft
        scalar points output timedata has nfft scalar points */
```

```
#define kiss_fftr_free speex_free

#ifdef __cplusplus
}
#endif
#endif
```

A.44. kiss_fftr.c

```
/*
   Original version: Copyright (c) 2003-2004, Mark Borgerding
   Followed by heavy modifications: Copyright (c) 2007-2008,
   Jean-Marc Valin

   All rights reserved.

   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   * Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer. *
   Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution. * Neither the author nor the names of any
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
   TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
   ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE. */

/* config.h.  Generated from config.h.in by configure.  */
/* config.h.in.  Generated from configure.ac by autoheader.  */

/* This is a build of CELT */
```

```
   /* Version extra */

   /* Version major */

   /* Version micro */

   /* Version minor */

   /* Complete version string */

   #include "os_support.h"
   #include "mathops.h"
   #include "kiss_fftr.h"
   #include "_kiss_fft_guts.h"

   kiss_fftr_cfg
   kiss_fftr_alloc(int nfft, void *mem, size_t * lenmem)
   {
     int             i;
     int             twiddle_size;
     kiss_fftr_cfg   st = NULL;
     size_t          subsize,
                     memneeded;

     if (nfft & 1)
     {
       celt_warning("Real FFT optimization must be even.\n");
       return NULL;
     }
     nfft >>= 1;
     twiddle_size = nfft / 2 + 1;
     kiss_fft_alloc(nfft, NULL, &subsize);
     memneeded =
         sizeof(struct kiss_fftr_state) + subsize +
         sizeof(kiss_twiddle_cpx) * twiddle_size;

     if (lenmem == NULL)
     {
       st = (kiss_fftr_cfg) KISS_FFT_MALLOC(memneeded);
     } else
     {
       if (*lenmem >= memneeded)
         st = (kiss_fftr_cfg) mem;
       *lenmem = memneeded;
     }
     if (!st)
       return NULL;
```

```
    st->substate = (kiss_fft_cfg) (st + 1);       /* just beyond
                                                     kiss_fftr_state
                                                     struct */
    st->super_twiddles =
        (kiss_twiddle_cpx *) (((char *) st->substate) + subsize);
    kiss_fft_alloc(nfft, st->substate, &subsize);

    st->substate->scale *= .5;
    for (i = 0; i < twiddle_size; ++i)
    {
      const double    pi = 3.14159265358979323846264338327;
      double          phase = pi * (((double) i) / nfft + .5);
      kf_cexp(st->super_twiddles + i, phase);
    }

    return st;
  }

  void
  kiss_fftr_twiddles(kiss_fftr_cfg st, kiss_fft_scalar * freqdata)
  {
    /* input buffer timedata is stored row-wise */
    int             k,
                    ncfft;
    kiss_fft_cpx    f2k,
                    f1k,
                    tdc,
                    tw;

    ncfft = st->substate->nfft;

    /* The real part of the DC element of the frequency spectrum in
       st->tmpbuf * contains the sum of the even-numbered elements of
       the input time sequence * The imag part is the sum of the
       odd-numbered elements * * The sum of tdc.r and tdc.i is the sum
       of the input time sequence. * yielding DC of input time
       sequence * The difference of tdc.r - tdc.i is the sum of the
       input (dot product) [1,-1,1,-1... * yielding Nyquist bin of
       input time sequence */

    tdc.r = freqdata[0];
    tdc.i = freqdata[1];
    C_FIXDIV(tdc, 2);
    CHECK_OVERFLOW_OP(tdc.r, +, tdc.i);
    CHECK_OVERFLOW_OP(tdc.r, -, tdc.i);
    freqdata[0] = tdc.r + tdc.i;
    freqdata[1] = tdc.r - tdc.i;
```

```
    for (k = 1; k <= ncfft / 2; ++k)
    {
      f2k.r =
          (((EXT32(freqdata[2 * k])) -
            (EXT32(freqdata[2 * (ncfft - k)])))));
      f2k.i =
          (((EXT32(freqdata[2 * k + 1])) +
            (EXT32(freqdata[2 * (ncfft - k) + 1]))));

      f1k.r =
          (((EXT32(freqdata[2 * k])) +
            (EXT32(freqdata[2 * (ncfft - k)])))));
      f1k.i =
          (((EXT32(freqdata[2 * k + 1])) -
            (EXT32(freqdata[2 * (ncfft - k) + 1]))));

      C_MULC(tw, f2k, st->super_twiddles[k]);

      freqdata[2 * k] = HALF_OF(f1k.r + tw.r);
      freqdata[2 * k + 1] = HALF_OF(f1k.i + tw.i);
      freqdata[2 * (ncfft - k)] = HALF_OF(f1k.r - tw.r);
      freqdata[2 * (ncfft - k) + 1] = HALF_OF(tw.i - f1k.i);

    }
  }

  void
  kiss_fftr(kiss_fftr_cfg st, const kiss_fft_scalar * timedata,
            kiss_fft_scalar * freqdata)
  {
    /* perform the parallel fft of two real signals packed in
       real,imag */
    kiss_fft(st->substate, (const kiss_fft_cpx *) timedata,
             (kiss_fft_cpx *) freqdata);

    kiss_fftr_twiddles(st, freqdata);
  }

  void
  kiss_fftr_inplace(kiss_fftr_cfg st, kiss_fft_scalar * X)
  {
    kf_work((kiss_fft_cpx *) X, NULL, 1, 1, st->substate->factors,
            st->substate, 1, 1, 1);
    kiss_fftr_twiddles(st, X);
  }

  void
  kiss_fftri(kiss_fftr_cfg st, const kiss_fft_scalar * freqdata,
```

```
              kiss_fft_scalar * timedata)
   {
     /* input buffer timedata is stored row-wise */
     int             k,
                     ncfft;

     ncfft = st->substate->nfft;

     timedata[2 * st->substate->bitrev[0]] = freqdata[0] + freqdata[1];
     timedata[2 * st->substate->bitrev[0] + 1] =
         freqdata[0] - freqdata[1];
     for (k = 1; k <= ncfft / 2; ++k)
     {
       kiss_fft_cpx    fk,
                       fnkc,
                       fek,
                       fok,
                       tmp;
       int             k1,
                       k2;
       k1 = st->substate->bitrev[k];
       k2 = st->substate->bitrev[ncfft - k];
       fk.r = freqdata[2 * k];
       fk.i = freqdata[2 * k + 1];
       fnkc.r = freqdata[2 * (ncfft - k)];
       fnkc.i = -freqdata[2 * (ncfft - k) + 1];

       C_ADD(fek, fk, fnkc);
       C_SUB(tmp, fk, fnkc);
       C_MUL(fok, tmp, st->super_twiddles[k]);
       timedata[2 * k1] = fek.r + fok.r;
       timedata[2 * k1 + 1] = fek.i + fok.i;
       timedata[2 * k2] = fek.r - fok.r;
       timedata[2 * k2 + 1] = fok.i - fek.i;
     }
     ki_work((kiss_fft_cpx *) timedata, NULL, 1, 1,
             st->substate->factors, st->substate, 1, 1, 1);
   }
```

A.45. kfft_single.h

```
/* Copyright (c) 2008 Xiph.Org Foundation, CSIRO */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:
```

```
#ifndef KFFT_SINGLE_H
#define KFFT_SINGLE_H

#ifdef ENABLE_TI_DSPLIB55

#include "dsplib.h"

#define real16_fft_alloc(length) NULL
#define real16_fft_free(state)
#define BITREV(state, i) (i)

#define real16_fft_inplace(state, X, nx)    (      cfft_SCALE(X,nx/2), \
     cbrev(X,X,nx/2),      unpack(X,nx)    )

#define real16_ifft(state, X, Y, nx)     (      unpacki(X, nx),      ci\
fft_NOSCALE(X,nx/2),      cbrev(X,Y,nx/2)    )

#else                           /* ENABLE_TI_DSPLIB */

#ifdef FIXED_POINT
```

```
#ifdef DOUBLE_PRECISION
#undef DOUBLE_PRECISION
#endif

#ifdef MIXED_PRECISION
#undef MIXED_PRECISION
#endif

#endif                          /* FIXED_POINT */

#include "kiss_fft.h"
#include "kiss_fftr.h"
#include "_kiss_fft_guts.h"

#define real16_fft_alloc(length) kiss_fftr_alloc_celt_single(length, 0,\
 0);
#define real16_fft_free(state) kiss_fft_free(state)
#define real16_fft_inplace(state, X, nx) kiss_fftr_inplace(state,X)
#define BITREV(state, i) ((state)->substate->bitrev[i])
#define real16_ifft(state, X, Y, nx) kiss_fftri(state,X, Y)

#endif                          /* !ENABLE_TI_DSPLIB */

#endif                          /* KFFT_SINGLE_H */
```

A.46. kfft_double.h

```
/* Copyright (c) 2008 Xiph.Org Foundation, CSIRO */
/*
   Redistribution and use in source and binary forms, with or
   without modification, are permitted provided that the following
   conditions are met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

   - Redistributions in binary form must reproduce the above
   copyright notice, this list of conditions and the following
   disclaimer in the documentation and/or other materials provided
   with the distribution.

   - Neither the name of the Xiph.org Foundation nor the names of
   its contributors may be used to endorse or promote products
   derived from this software without specific prior written
   permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
```

```
   CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
   OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
   TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
   OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
   OF SUCH DAMAGE. */

#ifndef KFFT_DOUBLE_H
#define KFFT_DOUBLE_H

#ifdef ENABLE_TI_DSPLIB55

#include "dsplib.h"
#include "_kiss_fft_guts.h"

#define cpx32_fft_alloc(length) NULL
#define cpx32_fft_free(state)

#define cpx32_fft(state, X, Y, nx)    (      cfft32_SCALE(X,nx),      c\
brev32(X,Y,nx)    )

#define cpx32_ifft(state, X, Y, nx)     (      cifft32_NOSCALE(X,nx),  \
    cbrev32(X,Y,nx)    )

#elif defined(ENABLE_TI_DSPLIB64)

#include "kiss_fft.h"
#include "_kiss_fft_guts.h"
#include "c64_fft.h"

#define cpx32_fft_alloc(length)     (kiss_fft_cfg)(c64_fft32_alloc(leng\
th, 0, 0))
#define cpx32_fft_free(state)         c64_fft32_free((c64_fft_t *)state)
#define cpx32_fft(state, X, Y, nx)     c64_fft32 ((c64_fft_t *)state, (\
const celt_int32 *)(X), (celt_int32 *)(Y))
#define cpx32_ifft(state, X, Y, nx)     c64_ifft32((c64_fft_t *)state, \
(const celt_int32 *)(X), (celt_int32 *)(Y))

#else                           /* ENABLE_TI_DSPLIB55/64 */

#include "kiss_fft.h"
#include "_kiss_fft_guts.h"
```

```
#define cpx32_fft_alloc(length) kiss_fft_alloc(length, 0, 0);
#define cpx32_fft_free(state) kiss_fft_free(state)
#define cpx32_fft(state, X, Y, nx) kiss_fft(state,(const kiss_fft_cpx *\
)(X), (kiss_fft_cpx *)(Y))
#define cpx32_ifft(state, X, Y, nx) kiss_ifft(state,(const kiss_fft_cpx\
 *)(X), (kiss_fft_cpx *)(Y))

#endif                          /* !ENABLE_TI_DSPLIB */

#endif                          /* KFFT_DOUBLE_H */
```

A.47. config.h

```
/* config.h.  Generated from config.h.in by configure.  */
/* config.h.in.  Generated from configure.ac by autoheader.  */

/* This is a build of CELT */
#define CELT_BUILD /**/

/* Version extra */
#define CELT_EXTRA_VERSION ""

/* Version major */
#define CELT_MAJOR_VERSION 0

/* Version micro */
#define CELT_MICRO_VERSION 2

/* Version minor */
#define CELT_MINOR_VERSION 5

/* Complete version string */
#define CELT_VERSION "0.6.0"


#define restrict
```

Authors' Addresses

Jean-Marc Valin
Octasic Semiconductor
4101, Molson Street, suite 300
Montreal, Quebec H1Y 3L1
Canada

Email: jean-marc.valin@octasic.com

Timothy B. Terriberry
Xiph.Org Foundation

Email: tterribe@xiph.org

Gregory Maxwell
Juniper Networks
2251 Corporate Park Drive, Suite 100
Herndon, VA 20171-1817
USA

Email: gmaxwell@juniper.net

Christopher Montgomery
Xiph.Org Foundation

Email: xiphmont@xiph.org