The Speex Codec Manual
Version 1.2 Beta 3

Jean-Marc Valin

December 11, 2007

Contents

List of Tables

1 Introduction to Speex

The Speex codec (`http://www.speex.org/`) exists because there is a need for a speech codec that is open-source and free from software patent royalties. These are essential conditions for being usable in any open-source software. In essence, Speex is to speech what Vorbis is to audio/music. Unlike many other speech codecs, Speex is not designed for mobile phones but rather for packet networks and voice over IP (VoIP) applications. File-based compression is of course also supported.

The Speex codec is designed to be very flexible and support a wide range of speech quality and bit-rate. Support for very good quality speech also means that Speex can encode wideband speech (16 kHz sampling rate) in addition to narrowband speech (telephone quality, 8 kHz sampling rate).

Designing for VoIP instead of mobile phones means that Speex is robust to lost packets, but not to corrupted ones. This is based on the assumption that in VoIP, packets either arrive unaltered or don't arrive at all. Because Speex is targeted at a wide range of devices, it has modest (adjustable) complexity and a small memory footprint.

All the design goals led to the choice of CELP as the encoding technique. One of the main reasons is that CELP has long proved that it could work reliably and scale well to both low bit-rates (e.g. DoD CELP @ 4.8 kbps) and high bit-rates (e.g. G.728 @ 16 kbps).

1.1 Getting help

As for many open source projects, there are many ways to get help with Speex. These include:

- This manual
- Other documentation on the Speex website (http://www.speex.org/)
- Mailing list: Discuss any Speex-related topic on speex-dev@xiph.org (not just for developers)
- IRC: The main channel is #speex on irc.freenode.net. Note that due to time differences, it may take a while to get someone, so please be patient.
- Email the author privately at jean-marc.valin@usherbrooke.ca **only** for private/delicate topics you do not wish to discuss publically.

Before asking for help (mailing list or IRC), **it is important to first read this manual** (OK, so if you made it here it's already a good sign). It is generally considered rude to ask on a mailing list about topics that are clearly detailed in the documentation. On the other hand, it's perfectly OK (and encouraged) to ask for clarifications about something covered in the manual. This manual does not (yet) cover everything about Speex, so everyone is encouraged to ask questions, send comments, feature requests, or just let us know how Speex is being used.

Here are some additional guidelines related to the mailing list. Before reporting bugs in Speex to the list, it is strongly recommended (if possible) to first test whether these bugs can be reproduced using the speexenc and speexdec (see Section 4) command-line utilities. Bugs reported based on 3rd party code are both harder to find and far too often caused by errors that have nothing to do with Speex.

1.2 About this document

This document is divided in the following way. Section 2 describes the different Speex features and defines many basic terms that are used throughout this manual. Section 4 documents the standard command-line tools provided in the Speex distribution. Section 5 includes detailed instructions about programming using the libspeex API. Section 7 has some information related to Speex and standards.

The three last sections describe the algorithms used in Speex. These sections require signal processing knowledge, but are not required for merely using Speex. They are intended for people who want to understand how Speex really works and/or want to do research based on Speex. Section 8 explains the general idea behind CELP, while sections 9 and 10 are specific to Speex.

2 Codec description

This section describes Speex and its features into more details.

2.1 Concepts

Before introducing all the Speex features, here are some concepts in speech coding that help better understand the rest of the manual. Although some are general concepts in speech/audio processing, others are specific to Speex.

Sampling rate

The sampling rate expressed in Hertz (Hz) is the number of samples taken from a signal per second. For a sampling rate of F_s kHz, the highest frequency that can be represented is equal to $F_s/2$ kHz ($F_s/2$ is known as the Nyquist frequency). This is a fundamental property in signal processing and is described by the sampling theorem. Speex is mainly designed for three different sampling rates: 8 kHz, 16 kHz, and 32 kHz. These are respectively refered to as narrowband, wideband and ultra-wideband.

Bit-rate

When encoding a speech signal, the bit-rate is defined as the number of bits per unit of time required to encode the speech. It is measured in *bits per second* (bps), or generally *kilobits per second*. It is important to make the distinction between *kilo**bits** per second* (k**b**ps) and *kilo**bytes** per second* (k**B**ps).

Quality (variable)

Speex is a lossy codec, which means that it achives compression at the expense of fidelity of the input speech signal. Unlike some other speech codecs, it is possible to control the tradeoff made between quality and bit-rate. The Speex encoding process is controlled most of the time by a quality parameter that ranges from 0 to 10. In constant bit-rate (CBR) operation, the quality parameter is an integer, while for variable bit-rate (VBR), the parameter is a float.

Complexity (variable)

With Speex, it is possible to vary the complexity allowed for the encoder. This is done by controlling how the search is performed with an integer ranging from 1 to 10 in a way that's similar to the -1 to -9 options to *gzip* and *bzip2* compression utilities. For normal use, the noise level at complexity 1 is between 1 and 2 dB higher than at complexity 10, but the CPU requirements for complexity 10 is about 5 times higher than for complexity 1. In practice, the best trade-off is between complexity 2 and 4, though higher settings are often useful when encoding non-speech sounds like DTMF tones.

Variable Bit-Rate (VBR)

Variable bit-rate (VBR) allows a codec to change its bit-rate dynamically to adapt to the "difficulty" of the audio being encoded. In the example of Speex, sounds like vowels and high-energy transients require a higher bit-rate to achieve good quality, while fricatives (e.g. s,f sounds) can be coded adequately with less bits. For this reason, VBR can achive lower bit-rate for the same quality, or a better quality for a certain bit-rate. Despite its advantages, VBR has two main drawbacks: first, by only specifying quality, there's no guaranty about the final average bit-rate. Second, for some real-time applications like voice over IP (VoIP), what counts is the maximum bit-rate, which must be low enough for the communication channel.

Average Bit-Rate (ABR)

Average bit-rate solves one of the problems of VBR, as it dynamically adjusts VBR quality in order to meet a specific target bit-rate. Because the quality/bit-rate is adjusted in real-time (open-loop), the global quality will be slightly lower than that obtained by encoding in VBR with exactly the right quality setting to meet the target average bit-rate.

Voice Activity Detection (VAD)

When enabled, voice activity detection detects whether the audio being encoded is speech or silence/background noise. VAD is always implicitly activated when encoding in VBR, so the option is only useful in non-VBR operation. In this case, Speex detects non-speech periods and encode them with just enough bits to reproduce the background noise. This is called "comfort noise generation" (CNG).

Discontinuous Transmission (DTX)

Discontinuous transmission is an addition to VAD/VBR operation, that allows to stop transmitting completely when the background noise is stationary. In file-based operation, since we cannot just stop writing to the file, only 5 bits are used for such frames (corresponding to 250 bps).

Perceptual enhancement

Perceptual enhancement is a part of the decoder which, when turned on, attempts to reduce the perception of the noise/distortion produced by the encoding/decoding process. In most cases, perceptual enhancement brings the sound further from the original *objectively* (e.g. considering only SNR), but in the end it still *sounds* better (subjective improvement).

Latency and algorithmic delay

Every speech codec introduces a delay in the transmission. For Speex, this delay is equal to the frame size, plus some amount of "look-ahead" required to process each frame. In narrowband operation (8 kHz), the delay is 30 ms, while for wideband (16 kHz), the delay is 34 ms. These values don't account for the CPU time it takes to encode or decode the frames.

2.2 Codec

The main characteristics of Speex can be summarized as follows:

- Free software/open-source, patent and royalty-free
- Integration of narrowband and wideband using an embedded bit-stream
- Wide range of bit-rates available (from 2.15 kbps to 44 kbps)
- Dynamic bit-rate switching (AMR) and Variable Bit-Rate (VBR) operation
- Voice Activity Detection (VAD, integrated with VBR) and discontinuous transmission (DTX)
- Variable complexity
- Embedded wideband structure (scalable sampling rate)
- Ultra-wideband sampling rate at 32 kHz
- Intensity stereo encoding option
- Fixed-point implementation

2.3 Preprocessor

This part refers to the preprocessor module introduced in the 1.1.x branch. The preprocessor is designed to be used on the audio *before* running the encoder. The preprocessor provides three main functionalities:

- noise suppression
- automatic gain control (AGC)
- voice activity detection (VAD)

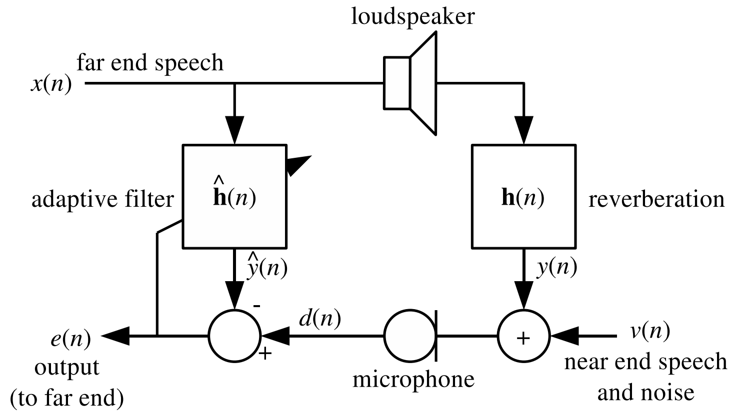

Figure 2.1: Acoustic echo model

The denoiser can be used to reduce the amount of background noise present in the input signal. This provides higher quality speech whether or not the denoised signal is encoded with Speex (or at all). However, when using the denoised signal with the codec, there is an additional benefit. Speech codecs in general (Speex included) tend to perform poorly on noisy input, which tends to amplify the noise. The denoiser greatly reduces this effect.

Automatic gain control (AGC) is a feature that deals with the fact that the recording volume may vary by a large amount between different setups. The AGC provides a way to adjust a signal to a reference volume. This is useful for voice over IP because it removes the need for manual adjustment of the microphone gain. A secondary advantage is that by setting the microphone gain to a conservative (low) level, it is easier to avoid clipping.

The voice activity detector (VAD) provided by the preprocessor is more advanced than the one directly provided in the codec.

2.4 Adaptive Jitter Buffer

When transmitting voice (or any content for that matter) over UDP or RTP, packet may be lost, arrive with different delay, or even out of order. The purpose of a jitter buffer is to reorder packets and buffer them long enough (but no longer than necessary) so they can be sent to be decoded.

2.5 Acoustic Echo Canceller

In any hands-free communication system (Fig. 2.1), speech from the remote end is played in the local loudspeaker, propagates in the room and is captured by the microphone. If the audio captured from the microphone is sent directly to the remote end, then the remove user hears an echo of his voice. An acoustic echo canceller is designed to remove the acoustic echo before it is sent to the remote end. It is important to understand that the echo canceller is meant to improve the quality on the **remote** end.

2.6 Resampler

In some cases, it may be useful to convert audio from one sampling rate to another. There are many reasons for that. It can be for mixing streams that have different sampling rates, for supporting sampling rates that the soundcard doesn't support, for transcoding, etc. That's why there is now a resampler that is part of the Speex project. This resampler can be used to convert between any two arbitrary rates (the ratio must only be a rational number) and there is control over the quality/complexity tradeoff.

3 Compiling and Porting

Compiling Speex under UNIX/Linux or any other platform supported by autoconf (e.g. Win32/cygwin) is as easy as typing:

```
% ./configure [options]
% make
% make install
```

The options supported by the Speex configure script are:

–prefix=<path> Specifies the base path for installing Speex (e.g. /usr)

–enable-shared/–disable-shared Whether to compile shared libraries

–enable-static/–disable-static Whether to compile static libraries

–disable-wideband Disable the wideband part of Speex (typically to save space)

–enable-valgrind Enable extra hits for valgrind for debugging purposes (do not use by default)

–enable-sse Enable use of SSE instructions (x86/float only)

–enable-fixed-point Compile Speex for a processor that does not have a floating point unit (FPU)

–enable-arm4-asm Enable assembly specific to the ARMv4 architecture (gcc only)

–enable-arm5e-asm Enable assembly specific to the ARMv5E architecture (gcc only)

–enable-fixed-point-debug Use only for debugging the fixed-point code (very slow)

–enable-epic-48k Enable a special (and non-compatible) 4.8 kbps narrowband mode (broken in 1.1.x and 1.2beta)

–enable-ti-c55x Enable support for the TI C5x family

–enable-blackfin-asm Enable assembly specific to the Blackfin DSP architecture (gcc only)

–enable-vorbis-psycho Make the encoder use the Vorbis psycho-acoustic model. This is very experimental and may be removed in the future.

3.1 Platforms

Speex is known to compile and work on a large number of architectures, both floating-point and fixed-point. In general, any architecture that can natively compute the multiplication of two signed 16-bit numbers (32-bit result) and runs at a sufficient clock rate (architecture-dependent) is capable of running Speex. Architectures on which Speex is **known** to work (it probably works on many others) are:

- x86 & x86-64
- Power
- SPARC
- ARM
- Blackfin
- Coldfire (68k family)
- TI C54xx & C55xx

- TI C6xxx
- TriMedia (experimental)

Operating systems on top of which Speex is known to work include (it probably works on many others):

- Linux
- μClinux
- MacOS X
- BSD
- Other UNIX/POSIX variants
- Symbian

The source code directory include additional information for compiling on certain architectures or operating systems in README.xxx files.

3.2 Porting and Optimising

Here are a few things to consider when porting or optimising Speex for a new platform or an existing one.

3.2.1 CPU optimisation

The single that will affect the CPU usage of Speex the most is whether it is compiled for floating point or fixed-point. If your CPU/DSP does not have a floating-point unit FPU, then compiling as fixed-point will be orders of magnitudes faster. If there is an FPU present, then it is important to test which version is faster. On the x86 architecture, floating-point is **generally** faster, but not always. To compile Speex as fixed-point, you need to pass –fixed-point to the configure script or define the FIXED_POINT macro for the compiler. As of 1.2beta3, it is now possible to disable the floating-point compatibility API, which means that your code can link without a float emulation library. To do that configure with –disable-float-api or define the DISABLE_FLOAT_API macro. Until the VBR feature is ported to fixed-point, you will also need to configure with –disable-vbr or define DISABLE_VBR.

Other important things to check on some DSP architectures are:

- Make sure the cache is set to write-back mode
- If the chip has SRAM instead of cache, make sure as much code and data are in SRAM, rather than in RAM

If you are going to be writing assembly, then the following functions are **usually** the first ones you should consider optimising:

- `filter_mem16()`
- `iir_mem16()`
- `vq_nbest()`
- `pitch_xcorr()`
- `interp_pitch()`

The filtering functions `filter_mem16()` and `iir_mem16()` are implemented in the direct form II transposed (DF2T). However, for architectures based on multiply-accumulate (MAC), DF2T requires frequent reload of the accumulator, which can make the code very slow. For these architectures (e.g. Blackfin and Coldfire), a better approach is to implement those functions as direct form I (DF1), which is easier to express in terms of MAC. When doing that however, **it is important to make sure that the DF1 implementation still behaves like the original DF2T behaviour when it comes to filter values**. This is necessary because the filter is time-varrying and must compute exactly the same value (not counting machine rounding) on any encoder or decoder.

3.2.2 Memory optimisation

Memory optimisation is mainly something that should be considered for small embedded platforms. For PCs, Speex is already so tiny that it's just not worth doing any of the things suggested here. There are several ways to reduce the memory usage of Speex, both in terms of code size and data size. For optimising code size, the trick is to first remove features you do not need. Some examples of things that can easily be disabled **if you don't need them** are:

- Wideband support (–disable-wideband)
- Support for stereo (removing stereo.c)
- VBR support (–disable-vbr or DISABLE_VBR)
- Static codebooks that are not needed for the bit-rates you are using (*_table.c files)

Speex also has several methods for allocating temporary arrays. When using a compiler that supports C99 properly (as of 2007, Microsoft compilers don't, but gcc does), it is best to define VAR_ARRAYS. That makes use of the variable-size array feature of C99. The next best is to define USE_ALLOCA so that Speex can use alloca() to allocate the temporary arrays. Note that on many systems, alloca() is buggy so it may not work. If none of VAR_ARRAYS and USE_ALLOCA are defined, then Speex falls back to allocating a large "scratch space" and doing its own internal allocation. The main disadvantage of this solution is that it is wasteful. It needs to allocate enough stack for the worst case scenario (worst bit-rate, highest complexity setting, ...) and by default, the memory isn't shared between multiple encoder/decoder states. Still, if the "manual" allocation is the only option left, there are a few things that can be improved. By overriding the speex_alloc_scratch() call in os_support.h, it is possible to always return the same memory area for all states[1]. In addition to that, by redefining the NB_ENC_STACK and NB_DEC_STACK (or similar for wideband), it is possible to only allocate memory for a scenario that is known in advange. In this case, it is important to measure the amount of memory required for the specific sampling rate, bit-rate and complexity level being used.

[1] In this case, one must be careful with threads

4 Command-line encoder/decoder

The base Speex distribution includes a command-line encoder (*speexenc*) and decoder (*speexdec*). Those tools produce and read Speex files encapsulated in the Ogg container. Although it is possible to encapsulate Speex in any container, Ogg is the recommended container for files. This section describes how to use the command line tools for Speex files in Ogg.

4.1 *speexenc*

The *speexenc* utility is used to create Speex files from raw PCM or wave files. It can be used by calling:

```
speexenc [options] input_file output_file
```

The value '-' for input_file or output_file corresponds respectively to stdin and stdout. The valid options are:

–narrowband (-n) Tell Speex to treat the input as narrowband (8 kHz). This is the default

–wideband (-w) Tell Speex to treat the input as wideband (16 kHz)

–ultra-wideband (-u) Tell Speex to treat the input as "ultra-wideband" (32 kHz)

–quality n Set the encoding quality (0-10), default is 8

–bitrate n Encoding bit-rate (use bit-rate n or lower)

–vbr Enable VBR (Variable Bit-Rate), disabled by default

–abr n Enable ABR (Average Bit-Rate) at n kbps, disabled by default

–vad Enable VAD (Voice Activity Detection), disabled by default

–dtx Enable DTX (Discontinuous Transmission), disabled by default

–nframes n Pack n frames in each Ogg packet (this saves space at low bit-rates)

–comp n Set encoding speed/quality tradeoff. The higher the value of n, the slower the encoding (default is 3)

-V Verbose operation, print bit-rate currently in use

–help (-h) Print the help

–version (-v) Print version information

Speex comments

–comment Add the given string as an extra comment. This may be used multiple times.

–author Author of this track.

–title Title for this track.

Raw input options

–rate n Sampling rate for raw input

–stereo Consider raw input as stereo

–le Raw input is little-endian

–be Raw input is big-endian

–8bit Raw input is 8-bit unsigned

–16bit Raw input is 16-bit signed

4.2 *speexdec*

The *speexdec* utility is used to decode Speex files and can be used by calling:

```
speexdec [options] speex_file [output_file]
```

The value '-' for input_file or output_file corresponds respectively to stdin and stdout. Also, when no output_file is specified, the file is played to the soundcard. The valid options are:

–enh enable post-filter (default)

–no-enh disable post-filter

–force-nb Force decoding in narrowband

–force-wb Force decoding in wideband

–force-uwb Force decoding in ultra-wideband

–mono Force decoding in mono

–stereo Force decoding in stereo

–rate n Force decoding at n Hz sampling rate

–packet-loss n Simulate n % random packet loss

-V Verbose operation, print bit-rate currently in use

–help (-h) Print the help

–version (-v) Print version information

5 Using the Speex Codec API (*libspeex*)

The *libspeex* library contains all the functions for encoding and decoding speech with the Speex codec. When linking on a UNIX system, one must add *-lspeex -lm* to the compiler command line. One important thing to know is that **libspeex calls are reentrant, but not thread-safe**. That means that it is fine to use calls from many threads, but **calls using the same state from multiple threads must be protected by mutexes**. Examples of code can also be found in Appendix A and the complete API documentation is included in the Documentation section of the Speex website (http://www.speex.org/).

5.1 Encoding

In order to encode speech using Speex, one first needs to:

```
#include <speex/speex.h>
```

Then in the code, a Speex bit-packing struct must be declared, along with a Speex encoder state:

```
SpeexBits bits;
void *enc_state;
```

The two are initialized by:

```
speex_bits_init(&bits);
enc_state = speex_encoder_init(&speex_nb_mode);
```

For wideband coding, *speex_nb_mode* will be replaced by *speex_wb_mode*. In most cases, you will need to know the frame size used at the sampling rate you are using. You can get that value in the *frame_size* variable (expressed in **samples**, not bytes) with:

```
speex_encoder_ctl(enc_state,SPEEX_GET_FRAME_SIZE,&frame_size);
```

In practice, *frame_size* will correspond to 20 ms when using 8, 16, or 32 kHz sampling rate. There are many parameters that can be set for the Speex encoder, but the most useful one is the quality parameter that controls the quality vs bit-rate tradeoff. This is set by:

```
speex_encoder_ctl(enc_state,SPEEX_SET_QUALITY,&quality);
```

where *quality* is an integer value ranging from 0 to 10 (inclusively). The mapping between quality and bit-rate is described in Fig. 9.2 for narrowband.

Once the initialization is done, for every input frame:

```
speex_bits_reset(&bits);
speex_encode_int(enc_state, input_frame, &bits);
nbBytes = speex_bits_write(&bits, byte_ptr, MAX_NB_BYTES);
```

where *input_frame* is a *(*short **)* pointing to the beginning of a speech frame, *byte_ptr* is a *(char *)* where the encoded frame will be written, *MAX_NB_BYTES* is the maximum number of bytes that can be written to *byte_ptr* without causing an overflow and *nbBytes* is the number of bytes actually written to *byte_ptr* (the encoded size in bytes). Before calling speex_bits_write, it is possible to find the number of bytes that need to be written by calling `speex_bits_nbytes(&bits)`, which returns a number of bytes.

It is still possible to use the *speex_encode()* function, which takes a *(float *)* for the audio. However, this would make an eventual port to an FPU-less platform (like ARM) more complicated. Internally, *speex_encode()* and *speex_encode_int()* are processed in the same way. Whether the encoder uses the fixed-point version is only decided by the compile-time flags, not at the API level.

After you're done with the encoding, free all resources with:

```
speex_bits_destroy(&bits);
speex_encoder_destroy(enc_state);
```

That's about it for the encoder.

5.2 Decoding

In order to decode speech using Speex, you first need to:

```
#include <speex/speex.h>
```

You also need to declare a Speex bit-packing struct

```
SpeexBits bits;
```

and a Speex decoder state

```
void *dec_state;
```

The two are initialized by:

```
speex_bits_init(&bits);
dec_state = speex_decoder_init(&speex_nb_mode);
```

For wideband decoding, *speex_nb_mode* will be replaced by *speex_wb_mode*. If you need to obtain the size of the frames that will be used by the decoder, you can get that value in the *frame_size* variable (expressed in **samples**, not bytes) with:

```
speex_decoder_ctl(dec_state, SPEEX_GET_FRAME_SIZE, &frame_size);
```

There is also a parameter that can be set for the decoder: whether or not to use a perceptual enhancer. This can be set by:

```
speex_decoder_ctl(dec_state, SPEEX_SET_ENH, &enh);
```

where *enh* is an int with value 0 to have the enhancer disabled and 1 to have it enabled. As of 1.2-beta1, the default is now to enable the enhancer.

Again, once the decoder initialization is done, for every input frame:

```
speex_bits_read_from(&bits, input_bytes, nbBytes);
speex_decode_int(dec_state, &bits, output_frame);
```

where input_bytes is a *(char *)* containing the bit-stream data received for a frame, *nbBytes* is the size (in bytes) of that bit-stream, and *output_frame* is a *(short *)* and points to the area where the decoded speech frame will be written. A NULL value as the second argument indicates that we don't have the bits for the current frame. When a frame is lost, the Speex decoder will do its best to "guess" the correct signal.

As for the encoder, the *speex_decode()* function can still be used, with a *(float *)* as the output for the audio. After you're done with the decoding, free all resources with:

```
speex_bits_destroy(&bits);
speex_decoder_destroy(dec_state);
```

5.3 Codec Options (speex_*_ctl)

Entities should not be multiplied beyond necessity – William of Ockham.

Just because there's an option for it doesn't mean you have to turn it on – me.

The Speex encoder and decoder support many options and requests that can be accessed through the *speex_encoder_ctl* and *speex_decoder_ctl* functions. These functions are similar to the *ioctl* system call and their prototypes are:

```
void speex_encoder_ctl(void *encoder, int request, void *ptr);
void speex_decoder_ctl(void *encoder, int request, void *ptr);
```

Despite those functions, the defaults are usually good for many applications and **optional settings should only be used when one understands them and knows that they are needed**. A common error is to attempt to set many unnecessary settings.

Here is a list of the values allowed for the requests. Some only apply to the encoder or the decoder. Because the last argument is of type `void *`, the `_ctl()` functions are **not type safe**, and shoud thus be used with care. The type `spx_int32_t` is the same as the C99 `int32_t` type.

SPEEX_SET_ENH‡ Set perceptual enhancer to on (1) or off (0) (`spx_int32_t`, default is on)

SPEEX_GET_ENH‡ Get perceptual enhancer status (`spx_int32_t`)

SPEEX_GET_FRAME_SIZE Get the number of samples per frame for the current mode (`spx_int32_t`)

SPEEX_SET_QUALITY† Set the encoder speech quality (`spx_int32_t` from 0 to 10, default is 8)

SPEEX_GET_QUALITY† Get the current encoder speech quality (`spx_int32_t` from 0 to 10)

SPEEX_SET_MODE† Set the mode number, as specified in the RTP spec (`spx_int32_t`)

SPEEX_GET_MODE† Get the current mode number, as specified in the RTP spec (`spx_int32_t`)

SPEEX_SET_VBR† Set variable bit-rate (VBR) to on (1) or off (0) (`spx_int32_t`, default is off)

SPEEX_GET_VBR† Get variable bit-rate (VBR) status (`spx_int32_t`)

SPEEX_SET_VBR_QUALITY† Set the encoder VBR speech quality (float 0.0 to 10.0, default is 8.0)

SPEEX_GET_VBR_QUALITY† Get the current encoder VBR speech quality (float 0 to 10)

SPEEX_SET_COMPLEXITY† Set the CPU resources allowed for the encoder (`spx_int32_t` from 1 to 10, default is 2)

SPEEX_GET_COMPLEXITY† Get the CPU resources allowed for the encoder (`spx_int32_t` from 1 to 10, default is 2)

SPEEX_SET_BITRATE† Set the bit-rate to use the closest value not exceeding the parameter (`spx_int32_t` in bits per second)

SPEEX_GET_BITRATE Get the current bit-rate in use (`spx_int32_t` in bits per second)

SPEEX_SET_SAMPLING_RATE Set real sampling rate (`spx_int32_t` in Hz)

SPEEX_GET_SAMPLING_RATE Get real sampling rate (`spx_int32_t` in Hz)

SPEEX_RESET_STATE Reset the encoder/decoder state to its original state, clearing all memories (no argument)

SPEEX_SET_VAD† Set voice activity detection (VAD) to on (1) or off (0) (`spx_int32_t`, default is off)

SPEEX_GET_VAD† Get voice activity detection (VAD) status (`spx_int32_t`)

SPEEX_SET_DTX† Set discontinuous transmission (DTX) to on (1) or off (0) (`spx_int32_t`, default is off)

SPEEX_GET_DTX† Get discontinuous transmission (DTX) status (`spx_int32_t`)

SPEEX_SET_ABR† Set average bit-rate (ABR) to a value n in bits per second (`spx_int32_t` in bits per second)

SPEEX_GET_ABR† Get average bit-rate (ABR) setting (`spx_int32_t` in bits per second)

SPEEX_SET_PLC_TUNING† Tell the encoder to optimize encoding for a certain percentage of packet loss (`spx_int32_t` in percent)

SPEEX_GET_PLC_TUNING† Get the current tuning of the encoder for PLC (`spx_int32_t` in percent)

SPEEX_SET_VBR_MAX_BITRATE† Set the maximum bit-rate allowed in VBR operation (`spx_int32_t` in bits per second)

SPEEX_GET_VBR_MAX_BITRATE† Get the current maximum bit-rate allowed in VBR operation (`spx_int32_t` in bits per second)

SPEEX_SET_HIGHPASS Set the high-pass filter on (1) or off (0) (`spx_int32_t`, default is on)

SPEEX_GET_HIGHPASS Get the current high-pass filter status (`spx_int32_t`)

† applies only to the encoder

‡ applies only to the decoder

5.4 Mode queries

Speex modes have a query system similar to the speex_encoder_ctl and speex_decoder_ctl calls. Since modes are read-only, it is only possible to get information about a particular mode. The function used to do that is:

```
void speex_mode_query(SpeexMode *mode, int request, void *ptr);
```

The admissible values for request are (unless otherwise note, the values are returned through *ptr*):

SPEEX_MODE_FRAME_SIZE Get the frame size (in samples) for the mode

SPEEX_SUBMODE_BITRATE Get the bit-rate for a submode number specified through *ptr* (integer in bps).

5.5 Packing and in-band signalling

Sometimes it is desirable to pack more than one frame per packet (or other basic unit of storage). The proper way to do it is to call speex_encode *N* times before writing the stream with speex_bits_write. In cases where the number of frames is not determined by an out-of-band mechanism, it is possible to include a terminator code. That terminator consists of the code 15 (decimal) encoded with 5 bits, as shown in Table 9.2. Note that as of version 1.0.2, calling speex_bits_write automatically inserts the terminator so as to fill the last byte. This doesn't involves any overhead and makes sure Speex can always detect when there is no more frame in a packet.

It is also possible to send in-band "messages" to the other side. All these messages are encoded as "pseudo-frames" of mode 14 which contain a 4-bit message type code, followed by the message. Table 5.1 lists the available codes, their meaning and the size of the message that follows. Most of these messages are requests that are sent to the encoder or decoder on the other end, which is free to comply or ignore them. By default, all in-band messages are ignored.

Code	Size (bits)	Content
0	1	Asks decoder to set perceptual enhancement off (0) or on(1)
1	1	Asks (if 1) the encoder to be less "agressive" due to high packet loss
2	4	Asks encoder to switch to mode N
3	4	Asks encoder to switch to mode N for low-band
4	4	Asks encoder to switch to mode N for high-band
5	4	Asks encoder to switch to quality N for VBR
6	4	Request acknowloedge (0=no, 1=all, 2=only for in-band data)
7	4	Asks encoder to set CBR (0), VAD(1), DTX(3), VBR(5), VBR+DTX(7)
8	8	Transmit (8-bit) character to the other end
9	8	Intensity stereo information
10	16	Announce maximum bit-rate acceptable (N in bytes/second)
11	16	reserved
12	32	Acknowledge receiving packet N
13	32	reserved
14	64	reserved
15	64	reserved

Table 5.1: In-band signalling codes

Finally, applications may define custom in-band messages using mode 13. The size of the message in bytes is encoded with 5 bits, so that the decoder can skip it if it doesn't know how to interpret it.

6 Speech Processing API (*libspeexdsp*)

As of version 1.2beta3, the non-codec parts of the Speex package are now in a separate library called *libspeexdsp*. This library includes the preprocessor, the acoustic echo canceller, the jitter buffer, and the resampler. In a UNIX environment, it can be linked into a program by adding *-lspeexdsp -lm* to the compiler command line. Just like for libspeex, **libspeexdsp calls are reentrant, but not thread-safe**. That means that it is fine to use calls from many threads, but **calls using the same state from multiple threads must be protected by mutexes**.

6.1 Preprocessor

In order to use the Speex preprocessor, you first need to:

```
#include <speex/speex_preprocess.h>
```

Then, a preprocessor state can be created as:

```
SpeexPreprocessState *preprocess_state = speex_preprocess_state_init(frame_size,
    sampling_rate);
```

and it is recommended to use the same value for `frame_size` as is used by the encoder (20 *ms*).

For each input frame, you need to call:

```
speex_preprocess_run(preprocess_state, audio_frame);
```

where `audio_frame` is used both as input and output. In cases where the output audio is not useful for a certain frame, it is possible to use instead:

```
speex_preprocess_estimate_update(preprocess_state, audio_frame);
```

This call will update all the preprocessor internal state variables without computing the output audio, thus saving some CPU cycles.

The behaviour of the preprocessor can be changed using:

```
speex_preprocess_ctl(preprocess_state, request, ptr);
```

which is used in the same way as the encoder and decoder equivalent. Options are listed in Section 6.1.1.

The preprocessor state can be destroyed using:

```
speex_preprocess_state_destroy(preprocess_state);
```

6.1.1 Preprocessor options

As with the codec, the preprocessor also has options that can be controlled using an ioctl()-like call. The available options are:

SPEEX_PREPROCESS_SET_DENOISE Turns denoising on(1) or off(2) (`spx_int32_t`)

SPEEX_PREPROCESS_GET_DENOISE Get denoising status (`spx_int32_t`)

SPEEX_PREPROCESS_SET_AGC Turns automatic gain control (AGC) on(1) or off(2) (`spx_int32_t`)

SPEEX_PREPROCESS_GET_AGC Get AGC status (`spx_int32_t`)

SPEEX_PREPROCESS_SET_VAD Turns voice activity detector (VAD) on(1) or off(2) (`spx_int32_t`)

SPEEX_PREPROCESS_GET_VAD Get VAD status (`spx_int32_t`)

SPEEX_PREPROCESS_SET_AGC_LEVEL

SPEEX_PREPROCESS_GET_AGC_LEVEL

SPEEX_PREPROCESS_SET_DEREVERB Turns reverberation removal on(1) or off(2) (`spx_int32_t`)

SPEEX_PREPROCESS_GET_DEREVERB Get reverberation removal status (`spx_int32_t`)

SPEEX_PREPROCESS_SET_DEREVERB_LEVEL Not working yet, do not use

SPEEX_PREPROCESS_GET_DEREVERB_LEVEL Not working yet, do not use

SPEEX_PREPROCESS_SET_DEREVERB_DECAY Not working yet, do not use

SPEEX_PREPROCESS_GET_DEREVERB_DECAY Not working yet, do not use

SPEEX_PREPROCESS_SET_PROB_START

SPEEX_PREPROCESS_GET_PROB_START

SPEEX_PREPROCESS_SET_PROB_CONTINUE

SPEEX_PREPROCESS_GET_PROB_CONTINUE

SPEEX_PREPROCESS_SET_NOISE_SUPPRESS Set maximum attenuation of the noise in dB (negative `spx_int32_t`)

SPEEX_PREPROCESS_GET_NOISE_SUPPRESS Get maximum attenuation of the noise in dB (negative `spx_int32_t`)

SPEEX_PREPROCESS_SET_ECHO_SUPPRESS Set maximum attenuation of the residual echo in dB (negative `spx_int32_t`)

SPEEX_PREPROCESS_GET_ECHO_SUPPRESS Set maximum attenuation of the residual echo in dB (negative `spx_int32_t`)

SPEEX_PREPROCESS_SET_ECHO_SUPPRESS_ACTIVE Set maximum attenuation of the echo in dB when near end is active (negative `spx_int32_t`)

SPEEX_PREPROCESS_GET_ECHO_SUPPRESS_ACTIVE Set maximum attenuation of the echo in dB when near end is active (negative `spx_int32_t`)

SPEEX_PREPROCESS_SET_ECHO_STATE Set the associated echo canceller for residual echo suppression (pointer or NULL for no residual echo suppression)

SPEEX_PREPROCESS_GET_ECHO_STATE Get the associated echo canceller (pointer)

6.2 Echo Cancellation

The Speex library now includes an echo cancellation algorithm suitable for Acoustic Echo Cancellation (AEC). In order to use the echo canceller, you first need to

```
#include <speex/speex_echo.h>
```

Then, an echo canceller state can be created by:

```
SpeexEchoState *echo_state = speex_echo_state_init(frame_size, filter_length);
```

where `frame_size` is the amount of data (in samples) you want to process at once and `filter_length` is the length (in samples) of the echo cancelling filter you want to use (also known as *tail length*). It is recommended to use a frame size in the order of 20 ms (or equal to the codec frame size) and make sure it is easy to perform an FFT of that size (powers of two are better than prime sizes). The recommended tail length is approximately the third of the room reverberation time. For example, in a small room, reverberation time is in the order of 300 ms, so a tail length of 100 ms is a good choice (800 samples at 8000 Hz sampling rate).

Once the echo canceller state is created, audio can be processed by:

```
speex_echo_cancellation(echo_state, input_frame, echo_frame, output_frame);
```

where `input_frame` is the audio as captured by the microphone, `echo_frame` is the signal that was played in the speaker (and needs to be removed) and `output_frame` is the signal with echo removed.

One important thing to keep in mind is the relationship between `input_frame` and `echo_frame`. It is important that, at any time, any echo that is present in the input has already been sent to the echo canceller as `echo_frame`. In other words, the echo canceller cannot remove a signal that it hasn't yet received. On the other hand, the delay between the input signal and the echo signal must be small enough because otherwise part of the echo cancellation filter is inefficient. In the ideal case, you code would look like:

```
write_to_soundcard(echo_frame, frame_size);
read_from_soundcard(input_frame, frame_size);
speex_echo_cancellation(echo_state, input_frame, echo_frame, output_frame);
```

If you wish to further reduce the echo present in the signal, you can do so by associating the echo canceller to the preprocessor (see Section 6.1). This is done by calling:

```
speex_preprocess_ctl(preprocess_state, SPEEX_PREPROCESS_SET_ECHO_STATE,echo_state);
```

in the initialisation.

As of version 1.2-beta2, there is an alternative, simpler API that can be used instead of *speex_echo_cancellation()*. When audio capture and playback are handled asynchronously (e.g. in different threads or using the *poll()* or *select()* system call), it can be difficult to keep track of what input_frame comes with what echo_frame. Instead, the playback comtext/thread can simply call:

```
speex_echo_playback(echo_state, echo_frame);
```

every time an audio frame is played. Then, the capture context/thread calls:

```
speex_echo_capture(echo_state, input_frame, output_frame);
```

for every frame captured. Internally, *speex_echo_playback()* simply buffers the playback frame so it can be used by *speex_echo_capture()* to call *speex_echo_cancel()*. A side effect of using this alternate API is that the playback audio is delayed by two frames, which is the normal delay caused by the soundcard. When capture and playback are already synchronised, *speex_echo_cancellation()* is preferable since it gives better control on the exact input/echo timing.

The echo cancellation state can be destroyed with:

```
speex_echo_state_destroy(echo_state);
```

It is also possible to reset the state of the echo canceller so it can be reused without the need to create another state with:

```
speex_echo_state_reset(echo_state);
```

6.2.1 Troubleshooting

There are several things that may prevent the echo canceller from working properly. One of them is a bug (or something suboptimal) in the code, but there are many others you should consider first

- Using a different soundcard to do the capture and plaback will **not** work, regardless of what you may think. The only exception to that is if the two cards can be made to have their sampling clock "locked" on the same clock source. If not, the clocks will always have a small amount of drift, which will prevent the echo canceller from adapting.

- The delay between the record and playback signals must be minimal. Any signal played has to "appear" on the playback (far end) signal slightly before the echo canceller "sees" it in the near end signal, but excessive delay means that part of the filter length is wasted. In the worst situations, the delay is such that it is longer than the filter length, in which case, no echo can be cancelled.

- When it comes to echo tail length (filter length), longer is *not* better. Actually, the longer the tail length, the longer it takes for the filter to adapt. Of course, a tail length that is too short will not cancel enough echo, but the most common problem seen is that people set a very long tail length and then wonder why no echo is being cancelled.

- Non-linear distortion cannot (by definition) be modeled by the linear adaptive filter used in the echo canceller and thus cannot be cancelled. Use good audio gear and avoid saturation/clipping.

Also useful is reading *Echo Cancellation Demystified* by Alexey Frunze[1], which explains the fundamental principles of echo cancellation. The details of the algorithm described in the article are different, but the general ideas of echo cancellation through adaptive filters are the same.

As of version 1.2beta2, a new `echo_diagnostic.m` tool is included in the source distribution. The first step is to define DUMP_ECHO_CANCEL_DATA during the build. This causes the echo canceller to automatically save the near-end, far-end and output signals to files (aec_rec.sw aec_play.sw and aec_out.sw). These are exactly what the AEC receives and outputs. From there, it is necessary to start Octave and type:

```
echo_diagnostic('aec_rec.sw', 'aec_play.sw', 'aec_diagnostic.sw', 1024);
```

The value of 1024 is the filter length and can be changed. There will be some (hopefully) useful messages printed and echo cancelled audio will be saved to aec_diagnostic.sw . If even that output is bad (almost no cancellation) then there is probably problem with the playback or recording process.

6.3 Jitter Buffer

The jitter buffer can be enabled by including:

```
#include <speex/speex_jitter.h>
```

and a new jitter buffer state can be initialised by:

```
JitterBuffer *state = jitter_buffer_init(step);
```

where the `step` argument is the default time step (in timestamp units) used for adjusting the delay and doing concealment. A value of 1 is always correct, but higher values may be more convenient sometimes. For example, if you are only able to do concealment on 20ms frames, there is no point in the jitter buffer asking you to do it on one sample. Another example is that for video, it makes no sense to adjust the delay by less than a full frame. The value provided can always be changed at a later time.

The jitter buffer API is based on the `JitterBufferPacket` type, which is defined as:

```
typedef struct {
   char          *data;       /* Data bytes contained in the packet */
   spx_uint32_t len;          /* Length of the packet in bytes */
   spx_uint32_t timestamp;    /* Timestamp for the packet */
   spx_uint32_t span;         /* Time covered by the packet (timestamp units) */
} JitterBufferPacket;
```

As an example, for audio the timestamp field would be what is obtained from the RTP timestamp field and the span would be the number of samples that are encoded in the packet. For Speex narrowband, span would be 160 if only one frame is included in the packet.

When a packet arrives, it need to be inserter into the jitter buffer by:

```
JitterBufferPacket packet;
/* Fill in each field in the packet struct */
jitter_buffer_put(state, &packet);
```

When the decoder is ready to decode a packet the packet to be decoded can be obtained by:

```
int start_offset;
err = jitter_buffer_get(state, &packet, desired_span, &start_offset);
```

If `jitter_buffer_put()` and `jitter_buffer_get()` are called from different threads, then **you need to protect the jitter buffer state with a mutex**.

Because the jitter buffer is designed not to use an explicit timer, it needs to be told about the time explicitly. This is done by calling:

```
jitter_buffer_tick(state);
```

This needs to be done periodically in the playing thread. This will be the last jitter buffer call before going to sleep (until more data is played back). In some cases, it may be preferable to use

[1] http://www.embeddedstar.com/articles/2003/7/article20030720-1.html

```
jitter_buffer_remaining_span(state, remaining);
```

The second argument is used to specify that we are still holding data that has not been written to the playback device. For instance, if 256 samples were needed by the soundcard (specified by `desired_span`), but `jitter_buffer_get()` returned 320 samples, we would have `remaining=64`.

6.4 Resampler

Speex includes a resampling modules. To make use of the resampler, it is necessary to include its header file:

```
#include <speex/speex_resampler.h>
```

For each stream that is to be resampled, it is necessary to create a resampler state with:

```
SpeexResamplerState *resampler;
resampler = speex_resampler_init(nb_channels, input_rate, output_rate, quality, &
   err);
```

where nb_channels is the number of channels that will be used (either interleaved or non-interleaved), input_rate is the sampling rate of the input stream, output_rate is the sampling rate of the output stream and quality is the requested quality setting (0 to 10). The quality parameter is useful for controlling the quality/complexity/latency tradeoff. Using a higher quality setting means less noise/aliasing, a higher complexity and a higher latency. Usually, a quality of 3 is acceptable for most desktop uses and quality 10 is mostly recommended for pro audio work. Quality 0 usually has a decent sound (certainly better than using linear interpolation resampling), but artifacts may be heard.

The actual resampling is performed using

```
err = speex_resampler_process_int(resampler, channelID, in, &in_length, out, &
   out_length);
```

where channelID is the ID of the channel to be processed. For a mono stream, use 0. The *in* pointer points to the first sample of the input buffer for the selected channel and *out* points to the first sample of the output. The size of the input and output buffers are specified by *in_length* and *out_length* respectively. Upon completion, these values are replaced by the number of samples read and written by the resampler. Unless an error occurs, either all input samples will be read or all output samples will be written to (or both). For floating-point samples, the function speex_resampler_process_float() behaves similarly.

It is also possible to process multiple channels at once.

To be continued...

6.5 Ring Buffer

Put some stuff there...

7 Formats and standards

Speex can encode speech in both narrowband and wideband and provides different bit-rates. However, not all features need to be supported by a certain implementation or device. In order to be called "Speex compatible" (whatever that means), an implementation must implement at least a basic set of features.

At the minimum, all narrowband modes of operation MUST be supported at the decoder. This includes the decoding of a wideband bit-stream by the narrowband decoder[1]. If present, a wideband decoder MUST be able to decode a narrowband stream, and MAY either be able to decode all wideband modes or be able to decode the embedded narrowband part of all modes (which includes ignoring the high-band bits).

For encoders, at least one narrowband or wideband mode MUST be supported. The main reason why all encoding modes do not have to be supported is that some platforms may not be able to handle the complexity of encoding in some modes.

7.1 RTP Payload Format

The RTP payload draft is included in appendix C and the latest version is available at `http://www.speex.org/drafts/latest`. This draft has been sent (2003/02/26) to the Internet Engineering Task Force (IETF) and will be discussed at the March 18th meeting in San Francisco.

7.2 MIME Type

For now, you should use the MIME type audio/x-speex for Speex-in-Ogg. We will apply for type `audio/speex` in the near future.

7.3 Ogg file format

Speex bit-streams can be stored in Ogg files. In this case, the first packet of the Ogg file contains the Speex header described in table 7.1. All integer fields in the headers are stored as little-endian. The `speex_string` field must contain the "`Speex   `" (with 3 trailing spaces), which identifies the bit-stream. The next field, `speex_version` contains the version of Speex that encoded the file. For now, refer to speex_header.[ch] for more info. The *beginning of stream* (`b_o_s`) flag is set to 1 for the header. The header packet has `packetno=0` and `granulepos=0`.

The second packet contains the Speex comment header. The format used is the Vorbis comment format described here: http://www.xiph.org/ogg/vorbis/doc/v-comment.html . This packet has `packetno=1` and `granulepos=0`.

The third and subsequent packets each contain one or more (number found in header) Speex frames. These are identified with `packetno` starting from 2 and the `granulepos` is the number of the last sample encoded in that packet. The last of these packets has the *end of stream* (`e_o_s`) flag is set to 1.

[1] The wideband bit-stream contains an embedded narrowband bit-stream which can be decoded alone

Field	Type	Size
speex_string	char[]	8
speex_version	char[]	20
speex_version_id	int	4
header_size	int	4
rate	int	4
mode	int	4
mode_bitstream_version	int	4
nb_channels	int	4
bitrate	int	4
frame_size	int	4
vbr	int	4
frames_per_packet	int	4
extra_headers	int	4
reserved1	int	4
reserved2	int	4

Table 7.1: Ogg/Speex header packet

8 Introduction to CELP Coding

Do not meddle in the affairs of poles, for they are subtle and quick to leave the unit circle.

Speex is based on CELP, which stands for Code Excited Linear Prediction. This section attempts to introduce the principles behind CELP, so if you are already familiar with CELP, you can safely skip to section 9. The CELP technique is based on three ideas:

1. The use of a linear prediction (LP) model to model the vocal tract
2. The use of (adaptive and fixed) codebook entries as input (excitation) of the LP model
3. The search performed in closed-loop in a “perceptually weighted domain”

This section describes the basic ideas behind CELP. This is still a work in progress.

8.1 Source-Filter Model of Speech Prediction

The source-filter model of speech production assumes that the vocal cords are the source of spectrally flat sound (the excitation signal), and that the vocal tract acts as a filter to spectrally shape the various sounds of speech. While still an approximation, the model is widely used in speech coding because of its simplicity.Its use is also the reason why most speech codecs (Speex included) perform badly on music signals. The different phonemes can be distinguished by their excitation (source) and spectral shape (filter). Voiced sounds (e.g. vowels) have an excitation signal that is periodic and that can be approximated by an impulse train in the time domain or by regularly-spaced harmonics in the frequency domain. On the other hand, fricatives (such as the "s", "sh" and "f" sounds) have an excitation signal that is similar to white Gaussian noise. So called voice fricatives (such as "z" and "v") have excitation signal composed of an harmonic part and a noisy part.

The source-filter model is usually tied with the use of Linear prediction. The CELP model is based on source-filter model, as can be seen from the CELP decoder illustrated in Figure 8.1.

8.2 Linear Prediction (LPC)

Linear prediction is at the base of many speech coding techniques, including CELP. The idea behind it is to predict the signal $x[n]$ using a linear combination of its past samples:

$$y[n] = \sum_{i=1}^{N} a_i x[n-i]$$

where $y[n]$ is the linear prediction of $x[n]$. The prediction error is thus given by:

$$e[n] = x[n] - y[n] = x[n] - \sum_{i=1}^{N} a_i x[n-i]$$

The goal of the LPC analysis is to find the best prediction coefficients a_i which minimize the quadratic error function:

$$E = \sum_{n=0}^{L-1} [e[n]]^2 = \sum_{n=0}^{L-1} \left[x[n] - \sum_{i=1}^{N} a_i x[n-i] \right]^2$$

That can be done by making all derivatives $\frac{\partial E}{\partial a_i}$ equal to zero:

$$\frac{\partial E}{\partial a_i} = \frac{\partial}{\partial a_i} \sum_{n=0}^{L-1} \left[x[n] - \sum_{i=1}^{N} a_i x[n-i] \right]^2 = 0$$

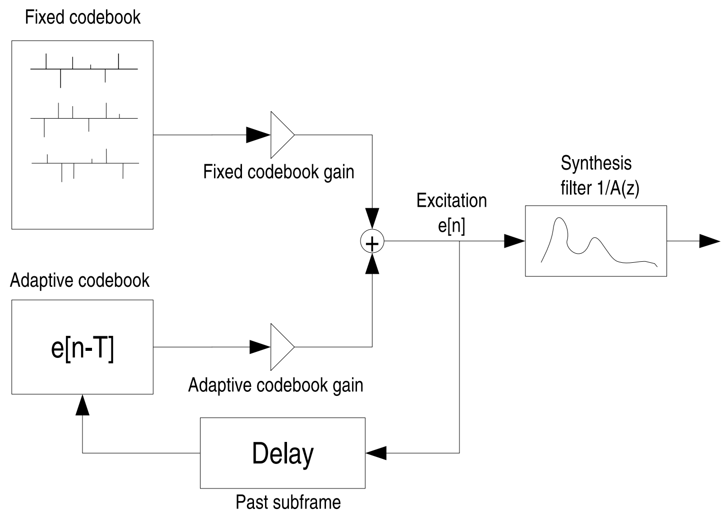

Figure 8.1: The CELP model of speech synthesis (decoder)

For an order N filter, the filter coefficients a_i are found by solving the system $N \times N$ linear system $\mathbf{Ra} = \mathbf{r}$, where

$$\mathbf{R} = \begin{bmatrix} R(0) & R(1) & \cdots & R(N-1) \\ R(1) & R(0) & \cdots & R(N-2) \\ \vdots & \vdots & \ddots & \vdots \\ R(N-1) & R(N-2) & \cdots & R(0) \end{bmatrix}$$

$$\mathbf{r} = \begin{bmatrix} R(1) \\ R(2) \\ \vdots \\ R(N) \end{bmatrix}$$

with $R(m)$, the auto-correlation of the signal $x[n]$, computed as:

$$R(m) = \sum_{i=0}^{N-1} x[i]x[i-m]$$

Because $\mathbf{R}$ is Hermitian Toeplitz, the Levinson-Durbin algorithm can be used, making the solution to the problem $\mathcal{O}\left(N^2\right)$ instead of $\mathcal{O}\left(N^3\right)$. Also, it can be proven that all the roots of $A(z)$ are within the unit circle, which means that $1/A(z)$ is always stable. This is in theory; in practice because of finite precision, there are two commonly used techniques to make sure we have a stable filter. First, we multiply $R(0)$ by a number slightly above one (such as 1.0001), which is equivalent to adding noise to the signal. Also, we can apply a window to the auto-correlation, which is equivalent to filtering in the frequency domain, reducing sharp resonances.

8.3 Pitch Prediction

During voiced segments, the speech signal is periodic, so it is possible to take advantage of that property by approximating the excitation signal $e[n]$ by a gain times the past of the excitation:

$$e[n] \simeq p[n] = \beta e[n-T] \ ,$$

where T is the pitch period, β is the pitch gain. We call that long-term prediction since the excitation is predicted from $e[n-T]$ with $T \gg N$.

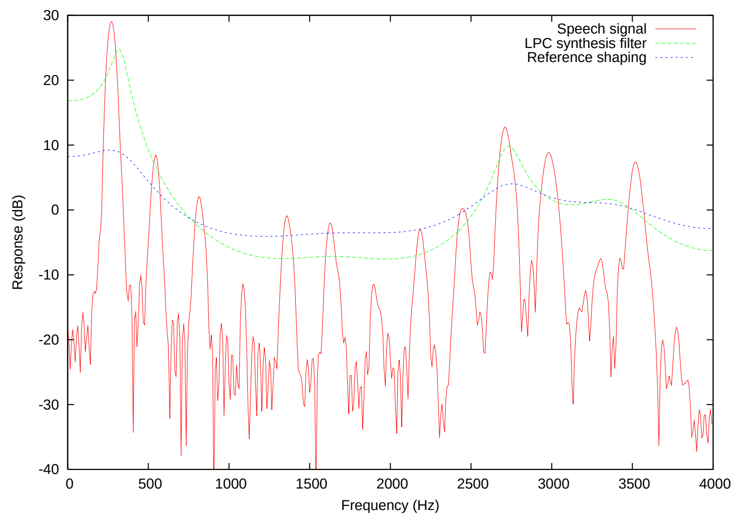

Figure 8.2: Standard noise shaping in CELP. Arbitrary y-axis offset.

8.4 Innovation Codebook

The final excitation $e[n]$ will be the sum of the pitch prediction and an *innovation* signal $c[n]$ taken from a fixed codebook, hence the name *Code* Excited Linear Prediction. The final excitation is given by

$$e[n] = p[n] + c[n] = \beta e[n-T] + c[n] \, .$$

The quantization of $c[n]$ is where most of the bits in a CELP codec are allocated. It represents the information that couldn't be obtained either from linear prediction or pitch prediction. In the z-domain we can represent the final signal $X(z)$ as

$$X(z) = \frac{C(z)}{A(z)\left(1-\beta z^{-T}\right)}$$

8.5 Noise Weighting

Most (if not all) modern audio codecs attempt to "shape" the noise so that it appears mostly in the frequency regions where the ear cannot detect it. For example, the ear is more tolerant to noise in parts of the spectrum that are louder and *vice versa*. In order to maximize speech quality, CELP codecs minimize the mean square of the error (noise) in the perceptually weighted domain. This means that a perceptual noise weighting filter $W(z)$ is applied to the error signal in the encoder. In most CELP codecs, $W(z)$ is a pole-zero weighting filter derived from the linear prediction coefficients (LPC), generally using bandwidth expansion. Let the spectral envelope be represented by the synthesis filter $1/A(z)$, CELP codecs typically derive the noise weighting filter as

$$W(z) = \frac{A(z/\gamma_1)}{A(z/\gamma_2)} \, , \tag{8.1}$$

where $\gamma_1 = 0.9$ and $\gamma_2 = 0.6$ in the Speex reference implementation. If a filter $A(z)$ has (complex) poles at p_i in the z-plane, the filter $A(z/\gamma)$ will have its poles at $p'_i = \gamma p_i$, making it a flatter version of $A(z)$.

The weighting filter is applied to the error signal used to optimize the codebook search through analysis-by-synthesis (AbS). This results in a spectral shape of the noise that tends towards $1/W(z)$. While the simplicity of the model has been an important reason for the success of CELP, it remains that $W(z)$ is a very rough approximation for the perceptually optimal noise weighting function. Fig. 8.2 illustrates the noise shaping that results from Eq. 8.1. Throughout this paper, we refer to $W(z)$ as the noise weighting filter and to $1/W(z)$ as the noise shaping filter (or curve).

8.6 Analysis-by-Synthesis

One of the main principles behind CELP is called Analysis-by-Synthesis (AbS), meaning that the encoding (analysis) is performed by perceptually optimising the decoded (synthesis) signal in a closed loop. In theory, the best CELP stream would be produced by trying all possible bit combinations and selecting the one that produces the best-sounding decoded signal. This is obviously not possible in practice for two reasons: the required complexity is beyond any currently available hardware and the “best sounding” selection criterion implies a human listener.

In order to achieve real-time encoding using limited computing resources, the CELP optimisation is broken down into smaller, more manageable, sequential searches using the perceptual weighting function described earlier.

9 Speex narrowband mode

This section looks at how Speex works for narrowband (8 kHz sampling rate) operation. The frame size for this mode is 20 ms, corresponding to 160 samples. Each frame is also subdivided into 4 sub-frames of 40 samples each.

Also many design decisions were based on the original goals and assumptions:

- Minimizing the amount of information extracted from past frames (for robustness to packet loss)
- Dynamically-selectable codebooks (LSP, pitch and innovation)
- sub-vector fixed (innovation) codebooks

9.1 Whole-Frame Analysis

In narrowband, Speex frames are 20 ms long (160 samples) and are subdivided in 4 sub-frames of 5 ms each (40 samples). For most narrowband bit-rates (8 kbps and above), the only parameters encoded at the frame level are the Line Spectral Pairs (LSP) and a global excitation gain g_{frame}, as shown in Fig. 9.1. All other parameters are encoded at the sub-frame level.

Linear prediction analysis is performed once per frame using an asymmetric Hamming window centered on the fourth sub-frame. Because linear prediction coefficients (LPC) are not robust to quantization, they are first are converted to line spectral pairs (LSP). The LSP's are considered to be associated to the 4^{th} sub-frames and the LSP's associated to the first 3 sub-frames are linearly interpolated using the current and previous LSP coefficients. The LSP coefficients and converted back to the LPC filter $\hat{A}(z)$. The non-quantized interpolated filter is denoted $A(z)$ and can be used for the weighting filter $W(z)$ because it does not need to be available to the decoder.

To make Speex more robust to packet loss, no prediction is applied on the LSP coefficients prior to quantization. The LSPs are encoded using vector quantizatin (VQ) with 30 bits for higher quality modes and 18 bits for lower quality.

9.2 Sub-Frame Analysis-by-Synthesis

The analysis-by-synthesis (AbS) encoder loop is described in Fig. 9.2. There are three main aspects where Speex significantly differs from most other CELP codecs. First, while most recent CELP codecs make use of fractional pitch estimation with a single gain, Speex uses an integer to encode the pitch period, but uses a 3-tap predictor (3 gains). The adaptive codebook contribution $e_a[n]$ can thus be expressed as:

$$e_a[n] = g_0 e[n-T-1] + g_1 e[n-T] + g_2 e[n-T+1] \tag{9.1}$$

where g_0, g_1 and g_2 are the jointly quantized pitch gains and $e[n]$ is the codec excitation memory. It is worth noting that when the pitch is smaller than the sub-frame size, we repeat the excitation at a period T. For example, when $n-T+1 \geq 0$, we use $n-2T+1$ instead. In most modes, the pitch period is encoded with 7 bits in the $[17,144]$ range and the β_i coefficients are vector-quantized using 7 bits at higher bit-rates (15 kbps narrowband and above) and 5 bits at lower bit-rates (11 kbps narrowband and below).

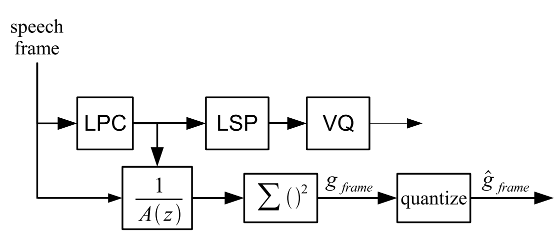

Figure 9.1: Frame open-loop analysis

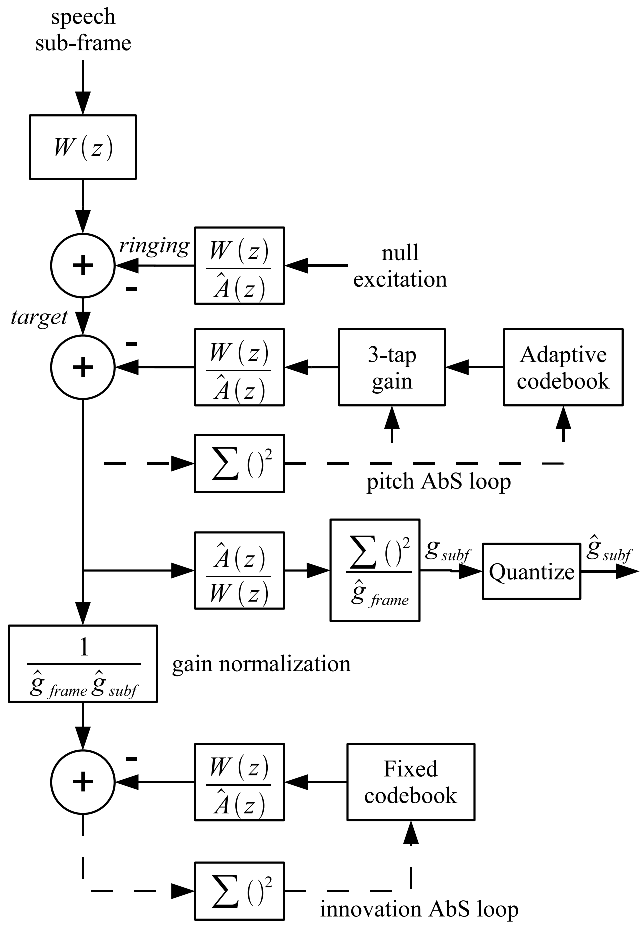

Figure 9.2: Analysis-by-synthesis closed-loop optimization on a sub-frame.

Many current CELP codecs use moving average (MA) prediction to encode the fixed codebook gain. This provides slightly better coding at the expense of introducing a dependency on previously encoded frames. A second difference is that Speex encodes the fixed codebook gain as the product of the global excitation gain g_{frame} with a sub-frame gain corrections g_{subf}. This increases robustness to packet loss by eliminating the inter-frame dependency. The sub-frame gain correction is encoded before the fixed codebook is searched (not closed-loop optimized) and uses between 0 and 3 bits per sub-frame, depending on the bit-rate.

The third difference is that Speex uses sub-vector quantization of the innovation (fixed codebook) signal instead of an algebraic codebook. Each sub-frame is divided into sub-vectors of lengths ranging between 5 and 20 samples. Each sub-vector is chosen from a bitrate-dependent codebook and all sub-vectors are concatenated to form a sub-frame. As an example, the 3.95 kbps mode uses a sub-vector size of 20 samples with 32 entries in the codebook (5 bits). This means that the innovation is encoded with 10 bits per sub-frame, or 2000 bps. On the other hand, the 18.2 kbps mode uses a sub-vector size of 5 samples with 256 entries in the codebook (8 bits), so the innovation uses 64 bits per sub-frame, or 12800 bps.

9.3 Bit allocation

There are 7 different narrowband bit-rates defined for Speex, ranging from 250 bps to 24.6 kbps, although the modes below 5.9 kbps should not be used for speech. The bit-allocation for each mode is detailed in table 9.1. Each frame starts with the mode ID encoded with 4 bits which allows a range from 0 to 15, though only the first 7 values are used (the others are reserved). The parameters are listed in the table in the order they are packed in the bit-stream. All frame-based parameters are packed before sub-frame parameters. The parameters for a certain sub-frame are all packed before the following sub-frame is packed. Note that the "OL" in the parameter description means that the parameter is an open loop estimation based on the whole frame.

Parameter	Update rate	0	1	2	3	4	5	6	7	8
Wideband bit	frame	1	1	1	1	1	1	1	1	1
Mode ID	frame	4	4	4	4	4	4	4	4	4
LSP	frame	0	18	18	18	18	30	30	30	18
OL pitch	frame	0	7	7	0	0	0	0	0	7
OL pitch gain	frame	0	4	0	0	0	0	0	0	4
OL Exc gain	frame	0	5	5	5	5	5	5	5	5
Fine pitch	sub-frame	0	0	0	7	7	7	7	7	0
Pitch gain	sub-frame	0	0	5	5	5	7	7	7	0
Innovation gain	sub-frame	0	1	0	1	1	3	3	3	0
Innovation VQ	sub-frame	0	0	16	20	35	48	64	96	10
Total	frame	5	43	119	160	220	300	364	492	79

Table 9.1: Bit allocation for narrowband modes

So far, no MOS (Mean Opinion Score) subjective evaluation has been performed for Speex. In order to give an idea of the quality achievable with it, table 9.2 presents my own subjective opinion on it. It sould be noted that different people will perceive the quality differently and that the person that designed the codec often has a bias (one way or another) when it comes to subjective evaluation. Last thing, it should be noted that for most codecs (including Speex) encoding quality sometimes varies depending on the input. Note that the complexity is only approximate (within 0.5 mflops and using the lowest complexity setting). Decoding requires approximately 0.5 mflops in most modes (1 mflops with perceptual enhancement).

9.4 Perceptual enhancement

This section was only valid for version 1.1.12 and earlier. It does not apply to version 1.2-beta1 (and later), for which the new perceptual enhancement is not yet documented.

This part of the codec only applies to the decoder and can even be changed without affecting inter-operability. For that reason, the implementation provided and described here should only be considered as a reference implementation. The enhancement system is divided into two parts. First, the synthesis filter $S(z) = 1/A(z)$ is replaced by an enhanced filter:

$$S'(z) = \frac{A(z/a_2)A(z/a_3)}{A(z)A(z/a_1)}$$

Mode	Quality	Bit-rate (bps)	mflops	Quality/description
0	-	250	0	No transmission (DTX)
1	0	2,150	6	Vocoder (mostly for comfort noise)
2	2	5,950	9	Very noticeable artifacts/noise, good intelligibility
3	3-4	8,000	10	Artifacts/noise sometimes noticeable
4	5-6	11,000	14	Artifacts usually noticeable only with headphones
5	7-8	15,000	11	Need good headphones to tell the difference
6	9	18,200	17.5	Hard to tell the difference even with good headphones
7	10	24,600	14.5	Completely transparent for voice, good quality music
8	1	3,950	10.5	Very noticeable artifacts/noise, good intelligibility
9	-	-	-	reserved
10	-	-	-	reserved
11	-	-	-	reserved
12	-	-	-	reserved
13	-	-	-	Application-defined, interpreted by callback or skipped
14	-	-	-	Speex in-band signaling
15	-	-	-	Terminator code

Table 9.2: Quality versus bit-rate

where a_1 and a_2 depend on the mode in use and $a_3 = \frac{1}{r}\left(1 - \frac{1-ra_1}{1-ra_2}\right)$ with $r = .9$. The second part of the enhancement consists of using a comb filter to enhance the pitch in the excitation domain.

10 Speex wideband mode (sub-band CELP)

For wideband, the Speex approach uses a *q*uadrature *m*irror *f*ilter (QMF) to split the band in two. The 16 kHz signal is thus divided into two 8 kHz signals, one representing the low band (0-4 kHz), the other the high band (4-8 kHz). The low band is encoded with the narrowband mode described in section 9 in such a way that the resulting "embedded narrowband bit-stream" can also be decoded with the narrowband decoder. Since the low band encoding has already been described, only the high band encoding is described in this section.

10.1 Linear Prediction

The linear prediction part used for the high-band is very similar to what is done for narrowband. The only difference is that we use only 12 bits to encode the high-band LSP's using a multi-stage vector quantizer (MSVQ). The first level quantizes the 10 coefficients with 6 bits and the error is then quantized using 6 bits, too.

10.2 Pitch Prediction

That part is easy: there's no pitch prediction for the high-band. There are two reasons for that. First, there is usually little harmonic structure in this band (above 4 kHz). Second, it would be very hard to implement since the QMF folds the 4-8 kHz band into 4-0 kHz (reversing the frequency axis), which means that the location of the harmonics is no longer at multiples of the fundamental (pitch).

10.3 Excitation Quantization

The high-band excitation is coded in the same way as for narrowband.

10.4 Bit allocation

For the wideband mode, the entire narrowband frame is packed before the high-band is encoded. The narrowband part of the bit-stream is as defined in table 9.1. The high-band follows, as described in table 10.1. For wideband, the mode ID is the same as the Speex quality setting and is defined in table 10.2. This also means that a wideband frame may be correctly decoded by a narrowband decoder with the only caveat that if more than one frame is packed in the same packet, the decoder will need to skip the high-band parts in order to sync with the bit-stream.

Parameter	Update rate	0	1	2	3	4
Wideband bit	frame	1	1	1	1	1
Mode ID	frame	3	3	3	3	3
LSP	frame	0	12	12	12	12
Excitation gain	sub-frame	0	5	4	4	4
Excitation VQ	sub-frame	0	0	20	40	80
Total	frame	4	36	112	192	352

Table 10.1: Bit allocation for high-band in wideband mode

Mode/Quality	Bit-rate (bps)	Quality/description
0	3,950	Barely intelligible (mostly for comfort noise)
1	5,750	Very noticeable artifacts/noise, poor intelligibility
2	7,750	Very noticeable artifacts/noise, good intelligibility
3	9,800	Artifacts/noise sometimes annoying
4	12,800	Artifacts/noise usually noticeable
5	16,800	Artifacts/noise sometimes noticeable
6	20,600	Need good headphones to tell the difference
7	23,800	Need good headphones to tell the difference
8	27,800	Hard to tell the difference even with good headphones
9	34,200	Hard to tell the difference even with good headphones
10	42,200	Completely transparent for voice, good quality music

Table 10.2: Quality versus bit-rate for the wideband encoder

A Sample code

This section shows sample code for encoding and decoding speech using the Speex API. The commands can be used to encode and decode a file by calling:
`% sampleenc in_file.sw | sampledec out_file.sw`
where both files are raw (no header) files encoded at 16 bits per sample (in the machine natural endianness).

A.1 sampleenc.c

sampleenc takes a raw 16 bits/sample file, encodes it and outputs a Speex stream to stdout. Note that the packing used is **not** compatible with that of speexenc/speexdec.

Listing A.1: Source code for sampleenc

```
#include <speex/speex.h>
#include <stdio.h>

/*The frame size in hardcoded for this sample code but it doesn't have to be*/
#define FRAME_SIZE 160
int main(int argc, char **argv)
{
   char *inFile;
   FILE *fin;
   short in[FRAME_SIZE];
   float input[FRAME_SIZE];
   char cbits[200];
   int nbBytes;
   /*Holds the state of the encoder*/
   void *state;
   /*Holds bits so they can be read and written to by the Speex routines*/
   SpeexBits bits;
   int i, tmp;

   /*Create a new encoder state in narrowband mode*/
   state = speex_encoder_init(&speex_nb_mode);

   /*Set the quality to 8 (15 kbps)*/
   tmp=8;
   speex_encoder_ctl(state, SPEEX_SET_QUALITY, &tmp);

   inFile = argv[1];
   fin = fopen(inFile, "r");

   /*Initialization of the structure that holds the bits*/
   speex_bits_init(&bits);
   while (1)
   {
      /*Read a 16 bits/sample audio frame*/
      fread(in, sizeof(short), FRAME_SIZE, fin);
      if (feof(fin))
         break;
      /*Copy the 16 bits values to float so Speex can work on them*/
```

```
      for (i=0;i<FRAME_SIZE;i++)
         input[i]=in[i];

      /*Flush all the bits in the struct so we can encode a new frame*/
      speex_bits_reset(&bits);

      /*Encode the frame*/
      speex_encode(state, input, &bits);
      /*Copy the bits to an array of char that can be written*/
      nbBytes = speex_bits_write(&bits, cbits, 200);

      /*Write the size of the frame first. This is what sampledec expects but
       it's likely to be different in your own application*/
      fwrite(&nbBytes, sizeof(int), 1, stdout);
      /*Write the compressed data*/
      fwrite(cbits, 1, nbBytes, stdout);

   }

   /*Destroy the encoder state*/
   speex_encoder_destroy(state);
   /*Destroy the bit-packing struct*/
   speex_bits_destroy(&bits);
   fclose(fin);
   return 0;
}
```

A.2 sampledec.c

sampledec reads a Speex stream from stdin, decodes it and outputs it to a raw 16 bits/sample file. Note that the packing used is **not** compatible with that of speexenc/speexdec.

Listing A.2: Source code for sampledec

```
#include <speex/speex.h>
#include <stdio.h>

/*The frame size in hardcoded for this sample code but it doesn't have to be*/
#define FRAME_SIZE 160
int main(int argc, char **argv)
{
   char *outFile;
   FILE *fout;
   /*Holds the audio that will be written to file (16 bits per sample)*/
   short out[FRAME_SIZE];
   /*Speex handle samples as float, so we need an array of floats*/
   float output[FRAME_SIZE];
   char cbits[200];
   int nbBytes;
   /*Holds the state of the decoder*/
   void *state;
   /*Holds bits so they can be read and written to by the Speex routines*/
   SpeexBits bits;
   int i, tmp;

   /*Create a new decoder state in narrowband mode*/
   state = speex_decoder_init(&speex_nb_mode);
```

```

   /*Set the perceptual enhancement on*/
   tmp=1;
   speex_decoder_ctl(state, SPEEX_SET_ENH, &tmp);

   outFile = argv[1];
   fout = fopen(outFile, "w");

   /*Initialization of the structure that holds the bits*/
   speex_bits_init(&bits);
   while (1)
   {
      /*Read the size encoded by sampleenc, this part will likely be
        different in your application*/
      fread(&nbBytes, sizeof(int), 1, stdin);
      fprintf (stderr, "nbBytes:␣%d\n", nbBytes);
      if (feof(stdin))
         break;

      /*Read the "packet" encoded by sampleenc*/
      fread(cbits, 1, nbBytes, stdin);
      /*Copy the data into the bit-stream struct*/
      speex_bits_read_from(&bits, cbits, nbBytes);

      /*Decode the data*/
      speex_decode(state, &bits, output);

      /*Copy from float to short (16 bits) for output*/
      for (i=0;i<FRAME_SIZE;i++)
         out[i]=output[i];

      /*Write the decoded audio to file*/
      fwrite(out, sizeof(short), FRAME_SIZE, fout);
   }

   /*Destroy the decoder state*/
   speex_decoder_destroy(state);
   /*Destroy the bit-stream truct*/
   speex_bits_destroy(&bits);
   fclose(fout);
   return 0;
}
```

B Jitter Buffer for Speex

Listing B.1: Example of using the jitter buffer for Speex packets

```
#include <speex/speex_jitter.h>
#include "speex_jitter_buffer.h"

#ifndef NULL
#define NULL 0
#endif


void speex_jitter_init(SpeexJitter *jitter, void *decoder, int sampling_rate)
{
   jitter->dec = decoder;
   speex_decoder_ctl(decoder, SPEEX_GET_FRAME_SIZE, &jitter->frame_size);

   jitter->packets = jitter_buffer_init(jitter->frame_size);

   speex_bits_init(&jitter->current_packet);
   jitter->valid_bits = 0;

}

void speex_jitter_destroy(SpeexJitter *jitter)
{
   jitter_buffer_destroy(jitter->packets);
   speex_bits_destroy(&jitter->current_packet);
}

void speex_jitter_put(SpeexJitter *jitter, char *packet, int len, int timestamp)
{
   JitterBufferPacket p;
   p.data = packet;
   p.len = len;
   p.timestamp = timestamp;
   p.span = jitter->frame_size;
   jitter_buffer_put(jitter->packets, &p);
}

void speex_jitter_get(SpeexJitter *jitter, spx_int16_t *out, int *current_timestamp
    )
{
   int i;
   int ret;
   spx_int32_t activity;
   char data[2048];
   JitterBufferPacket packet;
   packet.data = data;

   if (jitter->valid_bits)
   {
```

```
      /* Try decoding last received packet */
      ret = speex_decode_int(jitter->dec, &jitter->current_packet, out);
      if (ret == 0)
      {
         jitter_buffer_tick(jitter->packets);
         return;
      } else {
         jitter->valid_bits = 0;
      }
   }

   ret = jitter_buffer_get(jitter->packets, &packet, jitter->frame_size, NULL);

   if (ret != JITTER_BUFFER_OK)
   {
      /* No packet found */

      /*fprintf (stderr, "lost/late frame\n");*/
      /*Packet is late or lost*/
      speex_decode_int(jitter->dec, NULL, out);
   } else {
      speex_bits_read_from(&jitter->current_packet, packet.data, packet.len);
      /* Decode packet */
      ret = speex_decode_int(jitter->dec, &jitter->current_packet, out);
      if (ret == 0)
      {
         jitter->valid_bits = 1;
      } else {
         /* Error while decoding */
         for (i=0;i<jitter->frame_size;i++)
            out[i]=0;
      }
   }
   speex_decoder_ctl(jitter->dec, SPEEX_GET_ACTIVITY, &activity);
   if (activity < 30)
      jitter_buffer_update_delay(jitter->packets, &packet, NULL);
   jitter_buffer_tick(jitter->packets);
}

int speex_jitter_get_pointer_timestamp(SpeexJitter *jitter)
{
   return jitter_buffer_get_pointer_timestamp(jitter->packets);
}
```

C IETF RTP Profile

AVT G. Herlein
Internet-Draft
Intended status: Standards Track J. Valin
Expires: October 24, 2007 University of Sherbrooke
A. Heggestad
April 22, 2007

RTP Payload Format for the Speex Codec
draft-ietf-avt-rtp-speex-01 (non-final)

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on October 24, 2007.

Copyright Notice

Abstract

Speex is an open-source voice codec suitable for use in Voice over IP (VoIP) type applications. This document describes the payload format for Speex generated bit streams within an RTP packet. Also included here are the necessary details for the use of Speex with the Session Description Protocol (SDP).

Editors Note

All references to RFC XXXX are to be replaced by references to the RFC number of this memo, when published.

Table of Contents

1. Introduction

Speex is based on the CELP [CELP] encoding technique with support for either narrowband (nominal 8kHz), wideband (nominal 16kHz) or ultra-wideband (nominal 32kHz). The main characteristics can be summarized as follows:

- o Free software/open-source
- o Integration of wideband and narrowband in the same bit-stream
- o Wide range of bit-rates available
- o Dynamic bit-rate switching and variable bit-rate (VBR)
- o Voice Activity Detection (VAD, integrated with VBR)
- o Variable complexity

To be compliant with this specification, implementations MUST support 8 kHz sampling rate (narrowband)" and SHOULD support 8 kbps bitrate. The sampling rate MUST be 8, 16 or 32 kHz.

2. Terminology

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 RFC2119 [RFC2119] and indicate requirement levels for compliant RTP implementations.

3. RTP usage for Speex

3.1. RTP Speex Header Fields

The RTP header is defined in the RTP specification [RFC3550]. This section defines how fields in the RTP header are used.

Payload Type (PT): The assignment of an RTP payload type for this packet format is outside the scope of this document; it is specified by the RTP profile under which this payload format is used, or signaled dynamically out-of-band (e.g., using SDP).

Marker (M) bit: The M bit is set to one to indicate that the RTP packet payload contains at least one complete frame

Extension (X) bit: Defined by the RTP profile used.

Timestamp: A 32-bit word that corresponds to the sampling instant for the first frame in the RTP packet.

3.2. RTP payload format for Speex

The RTP payload for Speex has the format shown in Figure 1. No additional header fields specific to this payload format are required. For RTP based transportation of Speex encoded audio the standard RTP header [RFC3550] is followed by one or more payload data blocks. An optional padding terminator may also be used.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         RTP Header                            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                 one or more frames of Speex ....              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        one or more frames of Speex ....       |    padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: RTP payload for Speex

3.3. Speex payload

For the purposes of packetizing the bit stream in RTP, it is only necessary to consider the sequence of bits as output by the Speex encoder [speexenc], and present the same sequence to the decoder. The payload format described here maintains this sequence.

A typical Speex frame, encoded at the maximum bitrate, is approx. 110

octets and the total number of Speex frames SHOULD be kept less than the path MTU to prevent fragmentation. Speex frames MUST NOT be fragmented across multiple RTP packets,

An RTP packet MAY contain Speex frames of the same bit rate or of varying bit rates, since the bit-rate for a frame is conveyed in band with the signal.

The encoding and decoding algorithm can change the bit rate at any 20 msec frame boundary, with the bit rate change notification provided in-band with the bit stream. Each frame contains both "mode" (narrowband, wideband or ultra-wideband) and "sub-mode" (bit-rate) information in the bit stream. No out-of-band notification is required for the decoder to process changes in the bit rate sent by the encoder.

Sampling rate values of 8000, 16000 or 32000 Hz MUST be used. Any other sampling rates MUST NOT be used.

The RTP payload MUST be padded to provide an integer number of octets as the payload length. These padding bits are LSB aligned in network octet order and consist of a 0 followed by all ones (until the end of the octet). This padding is only required for the last frame in the packet, and only to ensure the packet contents ends on an octet boundary.

3.4. Example Speex packet

In the example below we have a single Speex frame with 5 bits of padding to ensure the packet size falls on an octet boundary.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      RTP Header                               |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                        ..speex data..                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        ..speex data..         |0 1 1 1 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

3.5. Multiple Speex frames in a RTP packet

Below is an example of two Speex frames contained within one RTP packet. The Speex frame length in this example fall on an octet boundary so there is no padding.

Speex codecs [speexenc] are able to detect the bitrate from the

payload and are responsible for detecting the 20 msec boundaries between each frame.

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                         RTP Header                            |
    +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
    |                       ..speex frame 1..                       |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |       ..speex frame 1..       |      ..speex frame 2..        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                       ..speex frame 2..                       |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4. IANA Considerations

This document defines the Speex media type.

4.1. Media Type Registration

This section describes the media types and names associated with this payload format. The section registers the media types, as per RFC4288 [RFC4288]

4.1.1. Registration of media type audio/speex

Media type name: audio

Media subtype name: speex

Required parameters:

None

Optional parameters:

ptime: see RFC 4566. SHOULD be a multiple of 20 msec.

maxptime: see RFC 4566. SHOULD be a multiple of 20 msec.

Encoding considerations:

This media type is framed and binary, see section 4.8 in [RFC4288].

Security considerations: See Section 6

Interoperability considerations:

None.

Published specification: RFC XXXX [This RFC].

Applications which use this media type:

Audio streaming and conferencing applications.

Additional information: none

Person and email address to contact for further information :

Alfred E. Heggestad: aeh@db.org

Intended usage: COMMON

Restrictions on usage:

This media type depends on RTP framing, and hence is only defined for transfer via RTP [RFC3550]. Transport within other framing protocols is not defined at this time.

Author: Alfred E. Heggestad

Change controller:

IETF Audio/Video Transport working group delegated from the IESG.

5. SDP usage of Speex

When conveying information by SDP [RFC4566], the encoding name MUST be set to "speex". An example of the media representation in SDP for offering a single channel of Speex at 8000 samples per second might be:

```
m=audio 8088 RTP/AVP 97
a=rtpmap:97 speex/8000
```

Note that the RTP payload type code of 97 is defined in this media definition to be 'mapped' to the speex codec at an 8kHz sampling frequency using the 'a=rtpmap' line. Any number from 96 to 127 could have been chosen (the allowed range for dynamic types).

The value of the sampling frequency is typically 8000 for narrow band operation, 16000 for wide band operation, and 32000 for ultra-wide band operation.

If for some reason the offerer has bandwidth limitations, the client may use the "b=" header, as explained in SDP [RFC4566]. The following example illustrates the case where the offerer cannot receive more than 10 kbit/s.

```
m=audio 8088 RTP/AVP 97
b=AS:10
a=rtmap:97 speex/8000
```

In this case, if the remote part agrees, it should configure its Speex encoder so that it does not use modes that produce more than 10 kbit/s. Note that the "b=" constraint also applies on all payload types that may be proposed in the media line ("m=").

An other way to make recommendations to the remote Speex encoder is to use its specific parameters via the a=fmtp: directive. The following parameters are defined for use in this way:

ptime: duration of each packet in milliseconds.

sr: actual sample rate in Hz.

ebw: encoding bandwidth - either 'narrow' or 'wide' or 'ultra' (corresponds to nominal 8000, 16000, and 32000 Hz sampling rates).

```
      vbr: variable bit rate - either 'on' 'off' or 'vad' (defaults to
      off).  If on, variable bit rate is enabled.  If off, disabled.  If
      set to 'vad' then constant bit rate is used but silence will be
      encoded with special short frames to indicate a lack of voice for
      that period.


      cng: comfort noise generation - either 'on' or 'off'.  If off then
      silence frames will be silent; if 'on' then those frames will be
      filled with comfort noise.


      mode: Speex encoding mode.  Can be {1,2,3,4,5,6,any} defaults to 3
      in narrowband, 6 in wide and ultra-wide.


   Examples:

      m=audio 8008 RTP/AVP 97
      a=rtpmap:97 speex/8000
      a=fmtp:97 mode=4

   This examples illustrate an offerer that wishes to receive a Speex
   stream at 8000Hz, but only using speex mode 4.

   Several Speex specific parameters can be given in a single a=fmtp
   line provided that they are separated by a semi-colon:

      a=fmtp:97 mode=any;mode=1

   The offerer may indicate that it wishes to send variable bit rate
   frames with comfort noise:

             m=audio 8088 RTP/AVP 97
             a=rtmap:97 speex/8000
             a=fmtp:97 vbr=on;cng=on

   The "ptime" attribute is used to denote the packetization interval
   (ie, how many milliseconds of audio is encoded in a single RTP
   packet).  Since Speex uses 20 msec frames, ptime values of multiples
   of 20 denote multiple Speex frames per packet.  Values of ptime which
   are not multiples of 20 MUST be ignored and clients MUST use the
   default value of 20 instead.

   Implementations SHOULD support ptime of 20 msec (i.e. one frame per
   packet)
```

In the example below the ptime value is set to 40, indicating that

there are 2 frames in each packet.

```
m=audio 8008 RTP/AVP 97
a=rtpmap:97 speex/8000
a=ptime:40
```

Note that the ptime parameter applies to all payloads listed in the media line and is not used as part of an a=fmtp directive.

Values of ptime not multiple of 20 msec are meaningless, so the receiver of such ptime values MUST ignore them. If during the life of an RTP session the ptime value changes, when there are multiple Speex frames for example, the SDP value must also reflect the new value.

Care must be taken when setting the value of ptime so that the RTP packet size does not exceed the path MTU.

6. Security Considerations

RTP packets using the payload format defined in this specification are subject to the security considerations discussed in the RTP specification [RFC3550], and any appropriate RTP profile. This implies that confidentiality of the media streams is achieved by encryption. Because the data compression used with this payload format is applied end-to-end, encryption may be performed after compression so there is no conflict between the two operations.

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 be overloaded. However, this encoding does not exhibit any significant non-uniformity.

As with any IP-based protocol, in some circumstances a receiver may be overloaded simply by the receipt of too many packets, either desired or undesired. Network-layer authentication may be used to discard packets from undesired sources, but the processing cost of the authentication itself may be too high.

7. Acknowledgements

The authors would like to thank Equivalence Pty Ltd of Australia for their assistance in attempting to standardize the use of Speex in H.323 applications, and for implementing Speex in their open source OpenH323 stack. The authors would also like to thank Brian C. Wiles <brian@streamcomm.com> of StreamComm for his assistance in developing the proposed standard for Speex use in H.323 applications.

The authors would also like to thank the following members of the Speex and AVT communities for their input: Ross Finlayson, Federico Montesino Pouzols, Henning Schulzrinne, Magnus Westerlund.

Thanks to former authors of this document; Simon Morlat, Roger Hardiman, Phil Kerr

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

8.2. Informative References

[CELP] "CELP, U.S. Federal Standard 1016.", National Technical Information Service (NTIS) website http://www.ntis.gov/.

[RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures", BCP 13, RFC 4288, December 2005.

[speexenc]
Valin, J., "Speexenc/speexdec, reference command-line encoder/decoder", Speex website http://www.speex.org/.

Authors' Addresses

Greg Herlein
2034 Filbert Street
San Francisco, California 94123
United States

Email: gherlein@herlein.com

Jean-Marc Valin
University of Sherbrooke
Department of Electrical and Computer Engineering
University of Sherbrooke
2500 blvd Universite
Sherbrooke, Quebec J1K 2R1
Canada

Email: jean-marc.valin@usherbrooke.ca

Alfred E. Heggestad
Biskop J. Nilssonsgt. 20a
Oslo 0659
Norway

Email: aeh@db.org

Full Copyright Statement

Intellectual Property

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).

D Speex License

E GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.

- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- H. Include an unaltered copy of this License.

- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Index