Opus

1.0.3

Generated by Doxygen 1.8.3.1

Thu Dec 5 2013 10:22:59

Contents

Chapter 1

Opus

The Opus codec is designed for interactive speech and audio transmission over the Internet.It is designed by the IETF Codec Working Group and incorporates technology from Skype's SILK codec and Xiph.Org's CELT codec.

The Opus codec is designed to handle a wide range of interactive audio applications, including Voice over IP, videoconferencing, in-game chat, and even remote live music performances. It can scale from low bit-rate narrowband speech to very high quality stereo music. Its main features are:

- Sampling rates from 8 to 48 kHz
- Bit-rates from 6 kb/s to 510 kb/s
- Support for both constant bit-rate (CBR) and variable bit-rate (VBR)
- Audio bandwidth from narrowband to full-band
- Support for speech and music
- Support for mono and stereo
- Support for multichannel (up to 255 channels)
- Frame sizes from 2.5 ms to 60 ms
- Good loss robustness and packet loss concealment (PLC)
- Floating point and fixed-point implementation

Documentation sections:

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Module Documentation

4.1 Opus Encoder

This page describes the process and functions used to encode Opus.

Typedefs

- typedef struct OpusEncoder OpusEncoder

  Opus encoder state.

Functions

- int opus_encoder_get_size (int channels)

  Gets the size of an `OpusEncoder` structure.
- OpusEncoder * opus_encoder_create (opus_int32 Fs, int channels, int application, int *error)

  Allocates and initializes an encoder state.
- int opus_encoder_init (OpusEncoder *st, opus_int32 Fs, int channels, int application)

  Initializes a previously allocated encoder state The memory pointed to by st must be at least the size returned by opus_encoder_get_size().
- opus_int32 opus_encode (OpusEncoder *st, const opus_int16 *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes an Opus frame.
- opus_int32 opus_encode_float (OpusEncoder *st, const float *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes an Opus frame from floating point input.
- void opus_encoder_destroy (OpusEncoder *st)

  Frees an `OpusEncoder` allocated by opus_encoder_create().
- int opus_encoder_ctl (OpusEncoder *st, int request,...)

  Perform a CTL function on an Opus encoder.

4.1.1 Detailed Description

This page describes the process and functions used to encode Opus. Since Opus is a stateful codec, the encoding process starts with creating an encoder state. This can be done with:

```
int          error;
OpusEncoder *enc;
enc = opus_encoder_create(Fs, channels, application, &error);
```

From this point, `enc` can be used for encoding an audio stream. An encoder state **must not** be used for more than one stream at the same time. Similarly, the encoder state **must not** be re-initialized for each frame.

While opus_encoder_create() allocates memory for the state, it's also possible to initialize pre-allocated memory:

```
int          size;
int          error;
OpusEncoder *enc;
size = opus_encoder_get_size(channels);
enc = malloc(size);
error = opus_encoder_init(enc, Fs, channels, application);
```

where opus_encoder_get_size() returns the required size for the encoder state. Note that future versions of this code may change the size, so no assuptions should be made about it.

The encoder state is always continuous in memory and only a shallow copy is sufficient to copy it (e.g. memcpy())

It is possible to change some of the encoder's settings using the opus_encoder_ctl() interface. All these settings already default to the recommended value, so they should only be changed when necessary. The most common settings one may want to change are:

```
opus_encoder_ctl(enc, OPUS_SET_BITRATE(bitrate));
opus_encoder_ctl(enc, OPUS_SET_COMPLEXITY(complexity));
opus_encoder_ctl(enc, OPUS_SET_SIGNAL(signal_type));
```

where

- bitrate is in bits per second (b/s)
- complexity is a value from 1 to 10, where 1 is the lowest complexity and 10 is the highest
- signal_type is either OPUS_AUTO (default), OPUS_SIGNAL_VOICE, or OPUS_SIGNAL_MUSIC

See Encoder related CTLs and Generic CTLs for a complete list of parameters that can be set or queried. Most parameters can be set or changed at any time during a stream.

To encode a frame, opus_encode() or opus_encode_float() must be called with exactly one frame (2.5, 5, 10, 20, 40 or 60 ms) of audio data:

```
len = opus_encode(enc, audio_frame, frame_size, packet, max_packet);
```

where

- audio_frame is the audio data in opus_int16 (or float for opus_encode_float())
- frame_size is the duration of the frame in samples (per channel)
- packet is the byte array to which the compressed data is written
- max_packet is the maximum number of bytes that can be written in the packet (4000 bytes is recommended). Do not use max_packet to control VBR target bitrate, instead use the OPUS_SET_BITRATE CTL.

opus_encode() and opus_encode_float() return the number of bytes actually written to the packet. The return value **can be negative**, which indicates that an error has occurred. If the return value is 1 byte, then the packet does not need to be transmitted (DTX).

Once the encoder state if no longer needed, it can be destroyed with

```
opus_encoder_destroy(enc);
```

If the encoder was created with opus_encoder_init() rather than opus_encoder_create(), then no action is required aside from potentially freeing the memory that was manually allocated for it (calling free(enc) for the example above)

4.1.2 Typedef Documentation

4.1.2.1 typedef struct OpusEncoder OpusEncoder

Opus encoder state.

This contains the complete state of an Opus encoder. It is position independent and can be freely copied.

See Also

opus_encoder_create,opus_encoder_init

4.1.3 Function Documentation

4.1.3.1 opus_int32 opus_encode (OpusEncoder * *st*, const opus_int16 * *pcm*, int *frame_size*, unsigned char * *data*, opus_int32 *max_data_bytes*)

Encodes an Opus frame.

Parameters

in	*st*	OpusEncoder*: Encoder state
in	*pcm*	opus_int16*: Input signal (interleaved if 2 channels). length is frame_size*channels*sizeof(opus_int16)
in	*frame_size*	int: Number of samples per channel in the input signal. This must be an Opus frame size for the encoder's sampling rate. For example, at 48 kHz the permitted values are 120, 240, 480, 960, 1920, and 2880. Passing in a duration of less than 10 ms (480 samples at 48 kHz) will prevent the encoder from using the LPC or hybrid modes.
out	*data*	unsigned char*: Output payload. This must contain storage for at least *max_data_bytes*.
in	*max_data_bytes*	opus_int32: Size of the allocated memory for the output payload. This may be used to impose an upper limit on the instant bitrate, but should not be used as the only bitrate control. Use OPUS_SET_BITRATE to control the bitrate.

Returns

The length of the encoded packet (in bytes) on success or a negative error code (see Error codes) on failure.

4.1.3.2 opus_int32 opus_encode_float (OpusEncoder * *st*, const float * *pcm*, int *frame_size*, unsigned char * *data*, opus_int32 *max_data_bytes*)

Encodes an Opus frame from floating point input.

Parameters

in	*st*	OpusEncoder*: Encoder state
in	*pcm*	float*: Input in float format (interleaved if 2 channels), with a normal range of +/-1.0. Samples with a range beyond +/-1.0 are supported but will be clipped by decoders using the integer API and should only be used if it is known that the far end supports extended dynamic range. length is frame_size*channels*sizeof(float)
in	*frame_size*	int: Number of samples per channel in the input signal. This must be an Opus frame size for the encoder's sampling rate. For example, at 48 kHz the permitted values are 120, 240, 480, 960, 1920, and 2880. Passing in a duration of less than 10 ms (480 samples at 48 kHz) will prevent the encoder from using the LPC or hybrid modes.
out	*data*	unsigned char*: Output payload. This must contain storage for at least *max_data_bytes*.
in	*max_data_bytes*	opus_int32: Size of the allocated memory for the output payload. This may be used to impose an upper limit on the instant bitrate, but should not be used as the only bitrate control. Use OPUS_SET_BITRATE to control the bitrate.

Returns

The length of the encoded packet (in bytes) on success or a negative error code (see Error codes) on failure.

4.1.3.3 OpusEncoder* opus_encoder_create (opus_int32 *Fs,* int *channels,* int *application,* int * *error*)

Allocates and initializes an encoder state.

There are three coding modes:

OPUS_APPLICATION_VOIP gives best quality at a given bitrate for voice signals. It enhances the input signal by high-pass filtering and emphasizing formants and harmonics. Optionally it includes in-band forward error correction to protect against packet loss. Use this mode for typical VoIP applications. Because of the enhancement, even at high bitrates the output may sound different from the input.

OPUS_APPLICATION_AUDIO gives best quality at a given bitrate for most non-voice signals like music. Use this mode for music and mixed (music/voice) content, broadcast, and applications requiring less than 15 ms of coding delay.

OPUS_APPLICATION_RESTRICTED_LOWDELAY configures low-delay mode that disables the speech-optimized mode in exchange for slightly reduced delay. This mode can only be set on an newly initialized or freshly reset encoder because it changes the codec delay.

This is useful when the caller knows that the speech-optimized modes will not be needed (use with caution).

Parameters

in	*Fs*	opus_int32: Sampling rate of input signal (Hz) This must be one of 8000, 12000, 16000, 24000, or 48000.
in	*channels*	int: Number of channels (1 or 2) in input signal
in	*application*	int: Coding mode (OPUS_APPLICATION_VOIP/OPUS_APPLICATION_AUDIO/OPUS_APPLICATION_RESTRICTED_LOWDELAY)
out	*error*	int*: Error codes

Note

Regardless of the sampling rate and number channels selected, the Opus encoder can switch to a lower audio bandwidth or number of channels if the bitrate selected is too low. This also means that it is safe to always use 48 kHz stereo input and let the encoder optimize the encoding.

4.1.3.4 int opus_encoder_ctl (OpusEncoder * *st*, int *request*, ...)

Perform a CTL function on an Opus encoder.

Generally the request and subsequent arguments are generated by a convenience macro.

Parameters

st	`OpusEncoder*`: Encoder state.
request	This and all remaining parameters should be replaced by one of the convenience macros in Generic CTLs or Encoder related CTLs.

See Also

Generic CTLs
Encoder related CTLs

4.1.3.5 void opus_encoder_destroy (OpusEncoder * *st*)

Frees an `OpusEncoder` allocated by opus_encoder_create().

Parameters

`in`	*st*	`OpusEncoder*`: State to be freed.

4.1.3.6 int opus_encoder_get_size (int *channels*)

Gets the size of an `OpusEncoder` structure.

Parameters

`in`	*channels*	`int`: Number of channels. This must be 1 or 2.

Returns

The size in bytes.

4.1.3.7 int opus_encoder_init (OpusEncoder * *st*, opus_int32 *Fs*, int *channels*, int *application*)

Initializes a previously allocated encoder state The memory pointed to by st must be at least the size returned by opus_encoder_get_size().

This is intended for applications which use their own allocator instead of malloc.

See Also

opus_encoder_create(),opus_encoder_get_size() To reset a previously initialized state, use the OPUS_RESET_STATE CTL.

Parameters

in	*st*	`OpusEncoder*`: Encoder state
in	*Fs*	`opus_int32`: Sampling rate of input signal (Hz) This must be one of 8000, 12000, 16000, 24000, or 48000.
in	*channels*	`int`: Number of channels (1 or 2) in input signal
in	*application*	`int`: Coding mode (OPUS_APPLICATION_VOIP/OPUS_APPLICATION_AUDIO/OPUS_APPLICATION_RESTRICTED_LOWDELAY)

Return values

OPUS_OK	Success or Error codes

4.2 Opus Decoder

This page describes the process and functions used to decode Opus.

Typedefs

- typedef struct OpusDecoder OpusDecoder

  Opus decoder state.

Functions

- int opus_decoder_get_size (int channels)

  Gets the size of an `OpusDecoder` structure.
- OpusDecoder * opus_decoder_create (opus_int32 Fs, int channels, int *error)

  Allocates and initializes a decoder state.
- int opus_decoder_init (OpusDecoder *st, opus_int32 Fs, int channels)

  Initializes a previously allocated decoder state.
- int opus_decode (OpusDecoder *st, const unsigned char *data, opus_int32 len, opus_int16 *pcm, int frame_size, int decode_fec)

  Decode an Opus packet.
- int opus_decode_float (OpusDecoder *st, const unsigned char *data, opus_int32 len, float *pcm, int frame_size, int decode_fec)

  Decode an Opus packet with floating point output.
- int opus_decoder_ctl (OpusDecoder *st, int request,...)

  Perform a CTL function on an Opus decoder.
- void opus_decoder_destroy (OpusDecoder *st)

  Frees an `OpusDecoder` allocated by opus_decoder_create().
- int opus_packet_parse (const unsigned char *data, opus_int32 len, unsigned char *out_toc, const unsigned char *frames[48], opus_int16 size[48], int *payload_offset)

  Parse an opus packet into one or more frames.
- int opus_packet_get_bandwidth (const unsigned char *data)

  Gets the bandwidth of an Opus packet.
- int opus_packet_get_samples_per_frame (const unsigned char *data, opus_int32 Fs)

  Gets the number of samples per frame from an Opus packet.
- int opus_packet_get_nb_channels (const unsigned char *data)

  Gets the number of channels from an Opus packet.
- int opus_packet_get_nb_frames (const unsigned char packet[], opus_int32 len)

  Gets the number of frames in an Opus packet.
- int opus_packet_get_nb_samples (const unsigned char packet[], opus_int32 len, opus_int32 Fs)

  Gets the number of samples of an Opus packet.
- int opus_decoder_get_nb_samples (const OpusDecoder *dec, const unsigned char packet[], opus_int32 len)

  Gets the number of samples of an Opus packet.

4.2.1 Detailed Description

This page describes the process and functions used to decode Opus. The decoding process also starts with creating a decoder state. This can be done with:

```
int          error;
OpusDecoder *dec;
dec = opus_decoder_create(Fs, channels, &error);
```

where

- Fs is the sampling rate and must be 8000, 12000, 16000, 24000, or 48000
- channels is the number of channels (1 or 2)
- error will hold the error code in case of failure (or OPUS_OK on success)
- the return value is a newly created decoder state to be used for decoding

While opus_decoder_create() allocates memory for the state, it's also possible to initialize pre-allocated memory:

```
int          size;
int          error;
OpusDecoder *dec;
size = opus_decoder_get_size(channels);
dec = malloc(size);
error = opus_decoder_init(dec, Fs, channels);
```

where opus_decoder_get_size() returns the required size for the decoder state. Note that future versions of this code may change the size, so no assuptions should be made about it.

The decoder state is always continuous in memory and only a shallow copy is sufficient to copy it (e.g. memcpy())

To decode a frame, opus_decode() or opus_decode_float() must be called with a packet of compressed audio data:

```
frame_size = opus_decode(dec, packet, len, decoded, max_size, 0);
```

where

- packet is the byte array containing the compressed data
- len is the exact number of bytes contained in the packet
- decoded is the decoded audio data in opus_int16 (or float for opus_decode_float())
- max_size is the max duration of the frame in samples (per channel) that can fit into the decoded_frame array

opus_decode() and opus_decode_float() return the number of samples (per channel) decoded from the packet. If that value is negative, then an error has occurred. This can occur if the packet is corrupted or if the audio buffer is too small to hold the decoded audio.

Opus is a stateful codec with overlapping blocks and as a result Opus packets are not coded independently of each other. Packets must be passed into the decoder serially and in the correct order for a correct decode. Lost packets can be replaced with loss concealment by calling the decoder with a null pointer and zero length for the missing packet.

A single codec state may only be accessed from a single thread at a time and any required locking must be performed by the caller. Separate streams must be decoded with separate decoder states and can be decoded in parallel unless the library was compiled with NONTHREADSAFE_PSEUDOSTACK defined.

4.2.2 Typedef Documentation

4.2.2.1 typedef struct OpusDecoder OpusDecoder

Opus decoder state.

This contains the complete state of an Opus decoder. It is position independent and can be freely copied.

See Also

opus_decoder_create,opus_decoder_init

4.2.3 Function Documentation

4.2.3.1 int opus_decode (OpusDecoder * *st*, const unsigned char * *data*, opus_int32 *len*, opus_int16 * *pcm*, int *frame_size*, int *decode_fec*)

Decode an Opus packet.

Parameters

in	*st*	OpusDecoder*: Decoder state
in	*data*	char*: Input payload. Use a NULL pointer to indicate packet loss
in	*len*	opus_int32: Number of bytes in payload*
out	*pcm*	opus_int16*: Output signal (interleaved if 2 channels). length is frame_size*channels*sizeof(opus_int16)
in	*frame_size*	Number of samples per channel of available space in *pcm*. If this is less than the maximum packet duration (120ms; 5760 for 48kHz), this function will not be capable of decoding some packets. In the case of PLC (data==NULL) or FEC (decode_fec=1), then frame_size needs to be exactly the duration of audio that is missing, otherwise the decoder will not be in the optimal state to decode the next incoming packet. For the PLC and FEC cases, frame_size **must** be a multiple of 2.5 ms.
in	*decode_fec*	int: Flag (0 or 1) to request that any in-band forward error correction data be decoded. If no such data is available, the frame is decoded as if it were lost.

Returns

Number of decoded samples or Error codes

4.2.3.2 int opus_decode_float (OpusDecoder * *st*, const unsigned char * *data*, opus_int32 *len*, float * *pcm*, int *frame_size*, int *decode_fec*)

Decode an Opus packet with floating point output.

Parameters

in	*st*	OpusDecoder*: Decoder state
in	*data*	char*: Input payload. Use a NULL pointer to indicate packet loss
in	*len*	opus_int32: Number of bytes in payload
out	*pcm*	float*: Output signal (interleaved if 2 channels). length is frame_size*channels*sizeof(float)

in	*frame_size*	Number of samples per channel of available space in *pcm*. If this is less than the maximum packet duration (120ms; 5760 for 48kHz), this function will not be capable of decoding some packets. In the case of PLC (data==NULL) or FEC (decode_fec=1), then frame_size needs to be exactly the duration of audio that is missing, otherwise the decoder will not be in the optimal state to decode the next incoming packet. For the PLC and FEC cases, frame_size **must** be a multiple of 2.5 ms.
in	*decode_fec*	`int`: Flag (0 or 1) to request that any in-band forward error correction data be decoded. If no such data is available the frame is decoded as if it were lost.

Returns

Number of decoded samples or Error codes

4.2.3.3 OpusDecoder* opus_decoder_create (opus_int32 *Fs*, int *channels*, int * *error*)

Allocates and initializes a decoder state.

Parameters

in	*Fs*	`opus_int32`: Sample rate to decode at (Hz). This must be one of 8000, 12000, 16000, 24000, or 48000.
in	*channels*	`int`: Number of channels (1 or 2) to decode
out	*error*	`int*`: OPUS_OK Success or Error codes

Internally Opus stores data at 48000 Hz, so that should be the default value for Fs. However, the decoder can efficiently decode to buffers at 8, 12, 16, and 24 kHz so if for some reason the caller cannot use data at the full sample rate, or knows the compressed data doesn't use the full frequency range, it can request decoding at a reduced rate. Likewise, the decoder is capable of filling in either mono or interleaved stereo pcm buffers, at the caller's request.

4.2.3.4 int opus_decoder_ctl (OpusDecoder * *st*, int *request*, ...)

Perform a CTL function on an Opus decoder.

Generally the request and subsequent arguments are generated by a convenience macro.

Parameters

st	`OpusDecoder*`: Decoder state.
request	This and all remaining parameters should be replaced by one of the convenience macros in Generic CTLs or Decoder related CTLs.

See Also

Generic CTLs
Decoder related CTLs

4.2.3.5 void opus_decoder_destroy (OpusDecoder * *st*)

Frees an `OpusDecoder` allocated by opus_decoder_create().

Parameters

in	*st*	`OpusDecoder*`: State to be freed.

4.2.3.6 int opus_decoder_get_nb_samples (const OpusDecoder * *dec*, const unsigned char *packet[]*, opus_int32 *len*)

Gets the number of samples of an Opus packet.

Parameters

in	*dec*	`OpusDecoder*`: Decoder state
in	*packet*	`char*`: Opus packet
in	*len*	`opus_int32`: Length of packet

Returns

Number of samples

Return values

OPUS_BAD_ARG	Insufficient data was passed to the function
OPUS_INVALID_PACKET	The compressed data passed is corrupted or of an unsupported type

4.2.3.7 int opus_decoder_get_size (int *channels*)

Gets the size of an `OpusDecoder` structure.

Parameters

in	*channels*	`int`: Number of channels. This must be 1 or 2.

Returns

The size in bytes.

4.2.3.8 int opus_decoder_init (OpusDecoder * *st*, opus_int32 *Fs*, int *channels*)

Initializes a previously allocated decoder state.

The state must be at least the size returned by opus_decoder_get_size(). This is intended for applications which use their own allocator instead of malloc.

See Also

opus_decoder_create,opus_decoder_get_size To reset a previously initialized state, use the OPUS_RESET_STATE CTL.

Parameters

in	*st*	OpusDecoder*: Decoder state.
in	*Fs*	opus_int32: Sampling rate to decode to (Hz). This must be one of 8000, 12000, 16000, 24000, or 48000.
in	*channels*	int: Number of channels (1 or 2) to decode

Return values

OPUS_OK	Success or Error codes

4.2.3.9 int opus_packet_get_bandwidth (const unsigned char * *data*)

Gets the bandwidth of an Opus packet.

Parameters

in	*data*	char*: Opus packet

Return values

OPUS_BANDWIDTH_NARROWBAND	Narrowband (4kHz bandpass)
OPUS_BANDWIDTH_MEDIUMBAND	Mediumband (6kHz bandpass)
OPUS_BANDWIDTH_WIDEBAND	Wideband (8kHz bandpass)
OPUS_BANDWIDTH_SUPERWIDEBAND	Superwideband (12kHz bandpass)
OPUS_BANDWIDTH_FULLBAND	Fullband (20kHz bandpass)
OPUS_INVALID_PACKET	The compressed data passed is corrupted or of an unsupported type

4.2.3.10 int opus_packet_get_nb_channels (const unsigned char * *data*)

Gets the number of channels from an Opus packet.

Parameters

in	*data*	char*: Opus packet

Returns

Number of channels

Return values

OPUS_INVALID_PACKET	The compressed data passed is corrupted or of an unsupported type

4.2.3.11 int opus_packet_get_nb_frames (const unsigned char *packet[],* opus_int32 *len*)

Gets the number of frames in an Opus packet.

Parameters

in	*packet*	`char*`: Opus packet
in	*len*	`opus_int32`: Length of packet

Returns

Number of frames

Return values

OPUS_BAD_ARG	Insufficient data was passed to the function
OPUS_INVALID_PACKET	The compressed data passed is corrupted or of an unsupported type

4.2.3.12 int opus_packet_get_nb_samples (const unsigned char *packet[],* opus_int32 *len,* opus_int32 *Fs*)

Gets the number of samples of an Opus packet.

Parameters

in	*packet*	`char*`: Opus packet
in	*len*	`opus_int32`: Length of packet
in	*Fs*	`opus_int32`: Sampling rate in Hz. This must be a multiple of 400, or inaccurate results will be returned.

Returns

Number of samples

Return values

OPUS_BAD_ARG	Insufficient data was passed to the function
OPUS_INVALID_PACKET	The compressed data passed is corrupted or of an unsupported type

4.2.3.13 int opus_packet_get_samples_per_frame (const unsigned char * *data,* opus_int32 *Fs*)

Gets the number of samples per frame from an Opus packet.

Parameters

in	*data*	`char*`: Opus packet. This must contain at least one byte of data.
in	*Fs*	`opus_int32`: Sampling rate in Hz. This must be a multiple of 400, or inaccurate results will be returned.

Returns

Number of samples per frame.

4.2.3.14 int opus_packet_parse (const unsigned char * *data*, opus_int32 *len*, unsigned char * *out_toc*, const unsigned char * *frames[48]*, opus_int16 *size[48]*, int * *payload_offset*)

Parse an opus packet into one or more frames.

Opus_decode will perform this operation internally so most applications do not need to use this function. This function does not copy the frames, the returned pointers are pointers into the input packet.

Parameters

in	*data*	`char*`: Opus packet to be parsed
in	*len*	`opus_int32`: size of data
out	*out_toc*	`char*`: TOC pointer
out	*frames*	`char*[48]` encapsulated frames
out	*size*	`opus_int16[48]` sizes of the encapsulated frames
out	*payload_offset*	`int*`: returns the position of the payload within the packet (in bytes)

Returns

number of frames

4.3 Repacketizer

The repacketizer can be used to merge multiple Opus packets into a single packet or alternatively to split Opus packets that have previously been merged.

Typedefs

- typedef struct OpusRepacketizer OpusRepacketizer

Functions

- int opus_repacketizer_get_size (void)

  Gets the size of an `OpusRepacketizer` structure.
- OpusRepacketizer * opus_repacketizer_init (OpusRepacketizer *rp)

  (Re)initializes a previously allocated repacketizer state.
- OpusRepacketizer * opus_repacketizer_create (void)

  Allocates memory and initializes the new repacketizer with opus_repacketizer_init().
- void opus_repacketizer_destroy (OpusRepacketizer *rp)

  Frees an `OpusRepacketizer` allocated by opus_repacketizer_create().
- int opus_repacketizer_cat (OpusRepacketizer *rp, const unsigned char *data, opus_int32 len)

  Add a packet to the current repacketizer state.
- opus_int32 opus_repacketizer_out_range (OpusRepacketizer *rp, int begin, int end, unsigned char *data, opus_int32 maxlen)

  Construct a new packet from data previously submitted to the repacketizer state via opus_repacketizer_cat().
- int opus_repacketizer_get_nb_frames (OpusRepacketizer *rp)

  Return the total number of frames contained in packet data submitted to the repacketizer state so far via opus_repacketizer_cat() since the last call to opus_repacketizer_init() or opus_repacketizer_create().
- opus_int32 opus_repacketizer_out (OpusRepacketizer *rp, unsigned char *data, opus_int32 maxlen)

  Construct a new packet from data previously submitted to the repacketizer state via opus_repacketizer_cat().

4.3.1 Detailed Description

The repacketizer can be used to merge multiple Opus packets into a single packet or alternatively to split Opus packets that have previously been merged. Splitting valid Opus packets is always guaranteed to succeed, whereas merging valid packets only succeeds if all frames have the same mode, bandwidth, and frame size, and when the total duration of the merged packet is no more than 120 ms. The repacketizer currently only operates on elementary Opus streams. It will not manipualte multistream packets successfully, except in the degenerate case where they consist of data from a single stream.

The repacketizing process starts with creating a repacketizer state, either by calling opus_repacketizer_create() or by allocating the memory yourself, e.g.,

```
OpusRepacketizer *rp;
rp = (OpusRepacketizer*)malloc(opus_repacketizer_get_size());
if (rp != NULL)
    opus_repacketizer_init(rp);
```

Then the application should submit packets with opus_repacketizer_cat(), extract new packets with opus_repacketizer_out() or opus_repacketizer_out_range(), and then reset the state for the next set of input packets via opus_repacketizer_init().

For example, to split a sequence of packets into individual frames:

```
unsigned char *data;
int len;
while (get_next_packet(&data, &len))
{
  unsigned char out[1276];
  opus_int32 out_len;
  int nb_frames;
  int err;
  int i;
  err = opus_repacketizer_cat(rp, data, len);
  if (err != OPUS_OK)
  {
    release_packet(data);
    return err;
  }
  nb_frames = opus_repacketizer_get_nb_frames(rp);
  for (i = 0; i < nb_frames; i++)
  {
    out_len = opus_repacketizer_out_range(rp, i, i+1, out, sizeof(out));
    if (out_len < 0)
    {
       release_packet(data);
       return (int)out_len;
    }
    output_next_packet(out, out_len);
  }
  opus_repacketizer_init(rp);
  release_packet(data);
}
```

Alternatively, to combine a sequence of frames into packets that each contain up to TARGET_DURATION_MS milliseconds of data:

```
// The maximum number of packets with duration TARGET_DURATION_MS occurs
// when the frame size is 2.5 ms, for a total of (TARGET_DURATION_MS*2/5)
// packets.
unsigned char *data[(TARGET_DURATION_MS*2/5)+1];
opus_int32 len[(TARGET_DURATION_MS*2/5)+1];
int nb_packets;
unsigned char out[1277*(TARGET_DURATION_MS*2/2)];
opus_int32 out_len;
int prev_toc;
nb_packets = 0;
while (get_next_packet(data+nb_packets, len+nb_packets))
{
  int nb_frames;
  int err;
  nb_frames = opus_packet_get_nb_frames(data[nb_packets], len[nb_packets]);
  if (nb_frames < 1)
  {
    release_packets(data, nb_packets+1);
    return nb_frames;
  }
  nb_frames += opus_repacketizer_get_nb_frames(rp);
  // If adding the next packet would exceed our target, or it has an
  // incompatible TOC sequence, output the packets we already have before
  // submitting it.
  // N.B., The nb_packets > 0 check ensures we've submitted at least one
  // packet since the last call to opus_repacketizer_init(). Otherwise a
  // single packet longer than TARGET_DURATION_MS would cause us to try to
  // output an (invalid) empty packet. It also ensures that prev_toc has
  // been set to a valid value. Additionally, len[nb_packets] > 0 is
  // guaranteed by the call to opus_packet_get_nb_frames() above, so the
  // reference to data[nb_packets][0] should be valid.
  if (nb_packets > 0 && (
      ((prev_toc & 0xFC) != (data[nb_packets][0] & 0xFC)) ||
      opus_packet_get_samples_per_frame(data[nb_packets], 48000)*nb_frames
       >
      TARGET_DURATION_MS*48))
  {
    out_len = opus_repacketizer_out(rp, out, sizeof(out));
    if (out_len < 0)
    {
       release_packets(data, nb_packets+1);
       return (int)out_len;
    }
    output_next_packet(out, out_len);
    opus_repacketizer_init(rp);
    release_packets(data, nb_packets);
```

```
    data[0] = data[nb_packets];
    len[0] = len[nb_packets];
    nb_packets = 0;
  }
  err = opus_repacketizer_cat(rp, data[nb_packets], len[nb_packets]);
  if (err != OPUS_OK)
  {
    release_packets(data, nb_packets+1);
    return err;
  }
  prev_toc = data[nb_packets][0];
  nb_packets++;
}
// Output the final, partial packet.
if (nb_packets > 0)
{
  out_len = opus_repacketizer_out(rp, out, sizeof(out));
  release_packets(data, nb_packets);
  if (out_len < 0)
    return (int)out_len;
  output_next_packet(out, out_len);
}
```

An alternate way of merging packets is to simply call opus_repacketizer_cat() unconditionally until it fails. At that point, the merged packet can be obtained with opus_repacketizer_out() and the input packet for which opus_repacketizer_cat() needs to be re-added to a newly reinitialized repacketizer state.

4.3.2 Typedef Documentation

4.3.2.1 typedef struct OpusRepacketizer OpusRepacketizer

4.3.3 Function Documentation

4.3.3.1 int opus_repacketizer_cat (OpusRepacketizer * *rp*, const unsigned char * *data*, opus_int32 *len*)

Add a packet to the current repacketizer state.

This packet must match the configuration of any packets already submitted for repacketization since the last call to opus_repacketizer_init(). This means that it must have the same coding mode, audio bandwidth, frame size, and channel count. This can be checked in advance by examining the top 6 bits of the first byte of the packet, and ensuring they match the top 6 bits of the first byte of any previously submitted packet. The total duration of audio in the repacketizer state also must not exceed 120 ms, the maximum duration of a single packet, after adding this packet.

The contents of the current repacketizer state can be extracted into new packets using opus_repacketizer_out() or opus_repacketizer_out_range().

In order to add a packet with a different configuration or to add more audio beyond 120 ms, you must clear the repacketizer state by calling opus_repacketizer_init(). If a packet is too large to add to the current repacketizer state, no part of it is added, even if it contains multiple frames, some of which might fit. If you wish to be able to add parts of such packets, you should first use another repacketizer to split the packet into pieces and add them individually.

See Also

opus_repacketizer_out_range
opus_repacketizer_out
opus_repacketizer_init

Parameters

	rp	`OpusRepacketizer*`: The repacketizer state to which to add the packet.
in	*data*	`const unsigned char*`: The packet data. The application must ensure this pointer remains valid until the next call to opus_repacketizer_init() or opus_repacketizer_destroy().
	len	`opus_int32`: The number of bytes in the packet data.

Returns

An error code indicating whether or not the operation succeeded.

Return values

OPUS_OK	The packet's contents have been added to the repacketizer state.
OPUS_INVALID_PACKET	The packet did not have a valid TOC sequence, the packet's TOC sequence was not compatible with previously submitted packets (because the coding mode, audio bandwidth, frame size, or channel count did not match), or adding this packet would increase the total amount of audio stored in the repacketizer state to more than 120 ms.

4.3.3.2 OpusRepacketizer* opus_repacketizer_create (void)

Allocates memory and initializes the new repacketizer with opus_repacketizer_init().

4.3.3.3 void opus_repacketizer_destroy (OpusRepacketizer * *rp*)

Frees an `OpusRepacketizer` allocated by opus_repacketizer_create().

Parameters

in	*rp*	`OpusRepacketizer*`: State to be freed.

4.3.3.4 int opus_repacketizer_get_nb_frames (OpusRepacketizer * *rp*)

Return the total number of frames contained in packet data submitted to the repacketizer state so far via opus_repacketizer_cat() since the last call to opus_repacketizer_init() or opus_repacketizer_create().

This defines the valid range of packets that can be extracted with opus_repacketizer_out_range() or opus_repacketizer_out().

Parameters

rp	`OpusRepacketizer*`: The repacketizer state containing the frames.

Returns

The total number of frames contained in the packet data submitted to the repacketizer state.

4.3.3.5 int opus_repacketizer_get_size (void)

Gets the size of an `OpusRepacketizer` structure.

Returns

The size in bytes.

4.3.3.6 OpusRepacketizer* opus_repacketizer_init (OpusRepacketizer * *rp*)

(Re)initializes a previously allocated repacketizer state.

The state must be at least the size returned by opus_repacketizer_get_size(). This can be used for applications which use their own allocator instead of malloc(). It must also be called to reset the queue of packets waiting to be repacketized, which is necessary if the maximum packet duration of 120 ms is reached or if you wish to submit packets with a different Opus configuration (coding mode, audio bandwidth, frame size, or channel count). Failure to do so will prevent a new packet from being added with opus_repacketizer_cat().

See Also

opus_repacketizer_create
opus_repacketizer_get_size
opus_repacketizer_cat

Parameters

rp	`OpusRepacketizer*`: The repacketizer state to (re)initialize.

Returns

A pointer to the same repacketizer state that was passed in.

4.3.3.7 opus_int32 opus_repacketizer_out (OpusRepacketizer * *rp*, unsigned char * *data*, opus_int32 *maxlen*)

Construct a new packet from data previously submitted to the repacketizer state via opus_repacketizer_cat().

This is a convenience routine that returns all the data submitted so far in a single packet. It is equivalent to calling

```
opus_repacketizer_out_range(rp, 0,
      opus_repacketizer_get_nb_frames(rp),
                          data, maxlen)
```

Parameters

	rp	`OpusRepacketizer*`: The repacketizer state from which to construct the new packet.
`out`	*data*	`const unsigned char*`: The buffer in which to store the output packet.

	maxlen	`opus_int32`: The maximum number of bytes to store in the output buffer. In order to guarantee success, this should be at least `1277*opus_repacketizer_get_nb_frames(rp)`. However, `1*opus_repacketizer_get_nb_frames(rp)` plus the size of all packet data submitted to the repacketizer since the last call to opus_repacketizer_init() or opus_repacketizer_create() is also sufficient, and possibly much smaller.

Returns

The total size of the output packet on success, or an error code on failure.

Return values

OPUS_BUFFER_TOO_SMALL	*maxlen* was insufficient to contain the complete output packet.

4.3.3.8 opus_int32 opus_repacketizer_out_range (OpusRepacketizer * *rp*, int *begin*, int *end*, unsigned char * *data*, opus_int32 *maxlen*)

Construct a new packet from data previously submitted to the repacketizer state via opus_repacketizer_cat().

Parameters

	rp	`OpusRepacketizer*`: The repacketizer state from which to construct the new packet.
	begin	`int`: The index of the first frame in the current repacketizer state to include in the output.
	end	`int`: One past the index of the last frame in the current repacketizer state to include in the output.
`out`	*data*	`const unsigned char*`: The buffer in which to store the output packet.
	maxlen	`opus_int32`: The maximum number of bytes to store in the output buffer. In order to guarantee success, this should be at least `1276` for a single frame, or for multiple frames, `1277*(end-begin)`. However, `1*(end-begin)` plus the size of all packet data submitted to the repacketizer since the last call to opus_repacketizer_init() or opus_repacketizer_create() is also sufficient, and possibly much smaller.

Returns

The total size of the output packet on success, or an error code on failure.

Return values

OPUS_BAD_ARG	`[begin,end)` was an invalid range of frames (begin < 0, begin >= end, or end > opus_repacketizer_get_nb_frames()).
OPUS_BUFFER_TOO_SMALL	*maxlen* was insufficient to contain the complete output packet.

4.4 Error codes

Macros

- #define OPUS_OK

  No error.
- #define OPUS_BAD_ARG

  One or more invalid/out of range arguments.
- #define OPUS_BUFFER_TOO_SMALL

  The mode struct passed is invalid.
- #define OPUS_INTERNAL_ERROR

  An internal error was detected.
- #define OPUS_INVALID_PACKET

  The compressed data passed is corrupted.
- #define OPUS_UNIMPLEMENTED

  Invalid/unsupported request number.
- #define OPUS_INVALID_STATE

  An encoder or decoder structure is invalid or already freed.
- #define OPUS_ALLOC_FAIL

  Memory allocation has failed.

4.4.1 Detailed Description

4.4.2 Macro Definition Documentation

4.4.2.1 #define OPUS_ALLOC_FAIL

Memory allocation has failed.

4.4.2.2 #define OPUS_BAD_ARG

One or more invalid/out of range arguments.

4.4.2.3 #define OPUS_BUFFER_TOO_SMALL

The mode struct passed is invalid.

4.4.2.4 #define OPUS_INTERNAL_ERROR

An internal error was detected.

4.4.2.5 #define OPUS_INVALID_PACKET

The compressed data passed is corrupted.

4.4.2.6 #define OPUS_INVALID_STATE

An encoder or decoder structure is invalid or already freed.

4.4.2.7 #define OPUS_OK

No error.

4.4.2.8 #define OPUS_UNIMPLEMENTED

Invalid/unsupported request number.

4.5 Pre-defined values for CTL interface

Macros

- #define OPUS_AUTO

  Auto/default setting.
- #define OPUS_BITRATE_MAX

  Maximum bitrate.
- #define OPUS_APPLICATION_VOIP

  Best for most VoIP/videoconference applications where listening quality and intelligibility matter most.
- #define OPUS_APPLICATION_AUDIO

  Best for broadcast/high-fidelity application where the decoded audio should be as close as possible to the input.
- #define OPUS_APPLICATION_RESTRICTED_LOWDELAY

  Only use when lowest-achievable latency is what matters most.
- #define OPUS_SIGNAL_VOICE 3001

  Signal being encoded is voice.
- #define OPUS_SIGNAL_MUSIC 3002

  Signal being encoded is music.
- #define OPUS_BANDWIDTH_NARROWBAND

  4 kHz bandpass
- #define OPUS_BANDWIDTH_MEDIUMBAND

  6 kHz bandpass
- #define OPUS_BANDWIDTH_WIDEBAND

  8 kHz bandpass
- #define OPUS_BANDWIDTH_SUPERWIDEBAND

  12 kHz bandpass
- #define OPUS_BANDWIDTH_FULLBAND

  20 kHz bandpass

4.5.1 Detailed Description

See Also

Generic CTLs, Encoder related CTLs

4.5.2 Macro Definition Documentation

4.5.2.1 #define OPUS_APPLICATION_AUDIO

Best for broadcast/high-fidelity application where the decoded audio should be as close as possible to the input.

4.5.2.2 #define OPUS_APPLICATION_RESTRICTED_LOWDELAY

Only use when lowest-achievable latency is what matters most.

Voice-optimized modes cannot be used.

4.5.2.3 #define OPUS_APPLICATION_VOIP

Best for most VoIP/videoconference applications where listening quality and intelligibility matter most.

4.5.2.4 #define OPUS_AUTO

Auto/default setting.

4.5.2.5 #define OPUS_BANDWIDTH_FULLBAND

20 kHz bandpass

4.5.2.6 #define OPUS_BANDWIDTH_MEDIUMBAND

6 kHz bandpass

4.5.2.7 #define OPUS_BANDWIDTH_NARROWBAND

4 kHz bandpass

4.5.2.8 #define OPUS_BANDWIDTH_SUPERWIDEBAND

12 kHz bandpass

4.5.2.9 #define OPUS_BANDWIDTH_WIDEBAND

8 kHz bandpass

4.5.2.10 #define OPUS_BITRATE_MAX

Maximum bitrate.

4.5.2.11 #define OPUS_SIGNAL_MUSIC 3002

Signal being encoded is music.

4.5.2.12 #define OPUS_SIGNAL_VOICE 3001

Signal being encoded is voice.

4.6 Encoder related CTLs

These are convenience macros for use with the `opus_encode_ctl` interface.

Macros

- #define OPUS_SET_COMPLEXITY(x)

  Configures the encoder's computational complexity.
- #define OPUS_GET_COMPLEXITY(x)

  Gets the encoder's complexity configuration.
- #define OPUS_SET_BITRATE(x)

  Configures the bitrate in the encoder.
- #define OPUS_GET_BITRATE(x)

  Gets the encoder's bitrate configuration.
- #define OPUS_SET_VBR(x)

  Enables or disables variable bitrate (VBR) in the encoder.
- #define OPUS_GET_VBR(x)

  Determine if variable bitrate (VBR) is enabled in the encoder.
- #define OPUS_SET_VBR_CONSTRAINT(x)

  Enables or disables constrained VBR in the encoder.
- #define OPUS_GET_VBR_CONSTRAINT(x)

  Determine if constrained VBR is enabled in the encoder.
- #define OPUS_SET_FORCE_CHANNELS(x)

  Configures mono/stereo forcing in the encoder.
- #define OPUS_GET_FORCE_CHANNELS(x)

  Gets the encoder's forced channel configuration.
- #define OPUS_SET_MAX_BANDWIDTH(x)

  Configures the maximum bandpass that the encoder will select automatically.
- #define OPUS_GET_MAX_BANDWIDTH(x)

  Gets the encoder's configured maximum allowed bandpass.
- #define OPUS_SET_BANDWIDTH(x)

  Sets the encoder's bandpass to a specific value.
- #define OPUS_SET_SIGNAL(x)

  Configures the type of signal being encoded.
- #define OPUS_GET_SIGNAL(x)

  Gets the encoder's configured signal type.
- #define OPUS_SET_APPLICATION(x)

  Configures the encoder's intended application.
- #define OPUS_GET_APPLICATION(x)

  Gets the encoder's configured application.
- #define OPUS_GET_SAMPLE_RATE(x)

  Gets the sampling rate the encoder or decoder was initialized with.
- #define OPUS_GET_LOOKAHEAD(x)

  Gets the total samples of delay added by the entire codec.
- #define OPUS_SET_INBAND_FEC(x)

  Configures the encoder's use of inband forward error correction (FEC).
- #define OPUS_GET_INBAND_FEC(x)

Gets encoder's configured use of inband forward error correction.

- #define OPUS_SET_PACKET_LOSS_PERC(x)

  Configures the encoder's expected packet loss percentage.
- #define OPUS_GET_PACKET_LOSS_PERC(x)

  Gets the encoder's configured packet loss percentage.
- #define OPUS_SET_DTX(x)

  Configures the encoder's use of discontinuous transmission (DTX).
- #define OPUS_GET_DTX(x)

  Gets encoder's configured use of discontinuous transmission.
- #define OPUS_SET_LSB_DEPTH(x)

  Configures the depth of signal being encoded.
- #define OPUS_GET_LSB_DEPTH(x)

  Gets the encoder's configured signal depth.
- #define OPUS_GET_LAST_PACKET_DURATION(x)

  Gets the duration (in samples) of the last packet successfully decoded or concealed.

4.6.1 Detailed Description

These are convenience macros for use with the `opus_encode_ctl` interface. They are used to generate the appropriate series of arguments for that call, passing the correct type, size and so on as expected for each particular request.

Some usage examples:

```
int ret;
ret = opus_encoder_ctl(enc_ctx, OPUS_SET_BANDWIDTH(
    OPUS_AUTO));
if (ret != OPUS_OK) return ret;

opus_int32 rate;
opus_encoder_ctl(enc_ctx, OPUS_GET_BANDWIDTH(&rate));

opus_encoder_ctl(enc_ctx, OPUS_RESET_STATE);
```

See Also

Generic CTLs, Opus Encoder

4.6.2 Macro Definition Documentation

4.6.2.1 #define OPUS_GET_APPLICATION(*x*)

Gets the encoder's configured application.

See Also

OPUS_SET_APPLICATION

Parameters

out	*x*	`opus_int32 *`: Returns one of the following values: **OPUS_APPLICATION_VOIP** Process signal for improved speech intelligibility. **OPUS_APPLICATION_AUDIO** Favor faithfulness to the original input. **OPUS_APPLICATION_RESTRICTED_LOWDELAY** Configure the minimum possible coding delay by disabling certain modes of operation.

4.6.2.2 #define OPUS_GET_BITRATE(*x*)

Gets the encoder's bitrate configuration.

See Also

OPUS_SET_BITRATE

Parameters

out	*x*	opus_int32 *: Returns the bitrate in bits per second. The default is determined based on the number of channels and the input sampling rate.

4.6.2.3 #define OPUS_GET_COMPLEXITY(*x*)

Gets the encoder's complexity configuration.

See Also

OPUS_SET_COMPLEXITY

Parameters

out	*x*	opus_int32 *: Returns a value in the range 0-10, inclusive.

4.6.2.4 #define OPUS_GET_DTX(*x*)

Gets encoder's configured use of discontinuous transmission.

See Also

OPUS_SET_DTX

Parameters

out	*x*	opus_int32 *: Returns one of the following values: **0** DTX disabled (default). **1** DTX enabled.

4.6.2.5 #define OPUS_GET_FORCE_CHANNELS(*x*)

Gets the encoder's forced channel configuration.

See Also

OPUS_SET_FORCE_CHANNELS

Parameters

out	*x*	opus_int32 *: **OPUS_AUTO** Not forced (default) **1** Forced mono **2** Forced stereo

4.6.2.6 #define OPUS_GET_INBAND_FEC(*x*)

Gets encoder's configured use of inband forward error correction.

See Also

OPUS_SET_INBAND_FEC

Parameters

out	*x*	opus_int32 *: Returns one of the following values: **0** Inband FEC disabled (default). **1** Inband FEC enabled.

4.6.2.7 #define OPUS_GET_LAST_PACKET_DURATION(*x*)

Gets the duration (in samples) of the last packet successfully decoded or concealed.

Parameters

out	*x*	opus_int32 *: Number of samples (at current sampling rate).

4.6.2.8 #define OPUS_GET_LOOKAHEAD(*x*)

Gets the total samples of delay added by the entire codec.

This can be queried by the encoder and then the provided number of samples can be skipped on from the start of the decoder's output to provide time aligned input and output. From the perspective of a decoding application the real data begins this many samples late.

The decoder contribution to this delay is identical for all decoders, but the encoder portion of the delay may vary from implementation to implementation, version to version, or even depend on the encoder's initial configuration. Applications needing delay compensation should call this CTL rather than hard-coding a value.

Parameters

out	*x*	opus_int32 *: Number of lookahead samples

4.6.2.9 #define OPUS_GET_LSB_DEPTH(x)

Gets the encoder's configured signal depth.

See Also

OPUS_SET_LSB_DEPTH

Parameters

out	x	opus_int32 *: Input precision in bits, between 8 and 24 (default: 24).

4.6.2.10 #define OPUS_GET_MAX_BANDWIDTH(x)

Gets the encoder's configured maximum allowed bandpass.

See Also

OPUS_SET_MAX_BANDWIDTH

Parameters

out	x	opus_int32 *: Allowed values: **OPUS_BANDWIDTH_NARROWBAND** 4 kHz passband **OPUS_BANDWIDTH_MEDIUMBAND** 6 kHz passband **OPUS_BANDWIDTH_WIDEBAND** 8 kHz passband **OPUS_BANDWIDTH_SUPERWIDEBAND** 12 kHz passband **OPUS_BANDWIDTH_FULLBAND** 20 kHz passband (default)

4.6.2.11 #define OPUS_GET_PACKET_LOSS_PERC(x)

Gets the encoder's configured packet loss percentage.

See Also

OPUS_SET_PACKET_LOSS_PERC

Parameters

out	x	opus_int32 *: Returns the configured loss percentage in the range 0-100, inclusive (default: 0).

4.6.2.12 #define OPUS_GET_SAMPLE_RATE(x)

Gets the sampling rate the encoder or decoder was initialized with.

This simply returns the Fs value passed to opus_encoder_init() or opus_decoder_init().

Parameters

out	*x*	opus_int32 *: Sampling rate of encoder or decoder.

4.6.2.13 #define OPUS_GET_SIGNAL(*x*)

Gets the encoder's configured signal type.

See Also

OPUS_SET_SIGNAL

Parameters

out	*x*	opus_int32 *: Returns one of the following values: **OPUS_AUTO** (default) **OPUS_SIGNAL_VOICE** Bias thresholds towards choosing LPC or Hybrid modes. **OPUS_SIGNAL_MUSIC** Bias thresholds towards choosing MDCT modes.

4.6.2.14 #define OPUS_GET_VBR(*x*)

Determine if variable bitrate (VBR) is enabled in the encoder.

See Also

OPUS_SET_VBR
OPUS_GET_VBR_CONSTRAINT

Parameters

out	*x*	opus_int32 *: Returns one of the following values: **0** Hard CBR. **1** VBR (default). The exact type of VBR may be retrieved via OPUS_GET_VBR_CONSTRAINT.

4.6.2.15 #define OPUS_GET_VBR_CONSTRAINT(*x*)

Determine if constrained VBR is enabled in the encoder.

See Also

OPUS_SET_VBR_CONSTRAINT
OPUS_GET_VBR

Parameters

out	*x*	opus_int32 *: Returns one of the following values: **0** Unconstrained VBR. **1** Constrained VBR (default).

4.6.2.16 #define OPUS_SET_APPLICATION(*x*)

Configures the encoder's intended application.

The initial value is a mandatory argument to the encoder_create function.

See Also

OPUS_GET_APPLICATION

Parameters

in	*x*	opus_int32: Returns one of the following values: **OPUS_APPLICATION_VOIP** Process signal for improved speech intelligibility. **OPUS_APPLICATION_AUDIO** Favor faithfulness to the original input. **OPUS_APPLICATION_RESTRICTED_LOWDELAY** Configure the minimum possible coding delay by disabling certain modes of operation.

4.6.2.17 #define OPUS_SET_BANDWIDTH(*x*)

Sets the encoder's bandpass to a specific value.

This prevents the encoder from automatically selecting the bandpass based on the available bitrate. If an application knows the bandpass of the input audio it is providing, it should normally use OPUS_SET_MAX_BANDWIDTH instead, which still gives the encoder the freedom to reduce the bandpass when the bitrate becomes too low, for better overall quality.

See Also

OPUS_GET_BANDWIDTH

Parameters

in	*x*	opus_int32: Allowed values: **OPUS_AUTO** (default) **OPUS_BANDWIDTH_NARROWBAND** 4 kHz passband **OPUS_BANDWIDTH_MEDIUMBAND** 6 kHz passband **OPUS_BANDWIDTH_WIDEBAND** 8 kHz passband **OPUS_BANDWIDTH_SUPERWIDEBAND** 12 kHz passband **OPUS_BANDWIDTH_FULLBAND** 20 kHz passband

4.6.2.18 #define OPUS_SET_BITRATE(*x*)

Configures the bitrate in the encoder.

Rates from 500 to 512000 bits per second are meaningful, as well as the special values OPUS_AUTO and OPUS_BITRATE_MAX. The value OPUS_BITRATE_MAX can be used to cause the codec to use as much rate as it can, which is useful for controlling the rate by adjusting the output buffer size.

See Also

OPUS_GET_BITRATE

Parameters

in	*x*	`opus_int32`: Bitrate in bits per second. The default is determined based on the number of channels and the input sampling rate.

4.6.2.19 #define OPUS_SET_COMPLEXITY(*x*)

Configures the encoder's computational complexity.

The supported range is 0-10 inclusive with 10 representing the highest complexity.

See Also

OPUS_GET_COMPLEXITY

Parameters

in	*x*	`opus_int32`: Allowed values: 0-10, inclusive.

4.6.2.20 #define OPUS_SET_DTX(*x*)

Configures the encoder's use of discontinuous transmission (DTX).

Note

This is only applicable to the LPC layer

See Also

OPUS_GET_DTX

Parameters

in	*x*	`opus_int32`: Allowed values: **0** Disable DTX (default). **1** Enabled DTX.

4.6.2.21 #define OPUS_SET_FORCE_CHANNELS(*x*)

Configures mono/stereo forcing in the encoder.

This can force the encoder to produce packets encoded as either mono or stereo, regardless of the format of the input audio. This is useful when the caller knows that the input signal is currently a mono source embedded in a stereo stream.

See Also

OPUS_GET_FORCE_CHANNELS

Parameters

in	*x*	opus_int32: Allowed values: **OPUS_AUTO** Not forced (default) **1** Forced mono **2** Forced stereo

4.6.2.22 #define OPUS_SET_INBAND_FEC(*x*)

Configures the encoder's use of inband forward error correction (FEC).

Note

This is only applicable to the LPC layer

See Also

OPUS_GET_INBAND_FEC

Parameters

in	*x*	opus_int32: Allowed values: **0** Disable inband FEC (default). **1** Enable inband FEC.

4.6.2.23 #define OPUS_SET_LSB_DEPTH(*x*)

Configures the depth of signal being encoded.

This is a hint which helps the encoder identify silence and near-silence.

See Also

OPUS_GET_LSB_DEPTH

Parameters

in	*x*	opus_int32: Input precision in bits, between 8 and 24 (default: 24).

4.6.2.24 #define OPUS_SET_MAX_BANDWIDTH(*x*)

Configures the maximum bandpass that the encoder will select automatically.

Applications should normally use this instead of OPUS_SET_BANDWIDTH (leaving that set to the default, OPUS_AUTO). This allows the application to set an upper bound based on the type of input it is providing, but still gives the encoder the freedom to reduce the bandpass when the bitrate becomes too low, for better overall quality.

See Also

OPUS_GET_MAX_BANDWIDTH

Parameters

in	*x*	opus_int32: Allowed values: **OPUS_BANDWIDTH_NARROWBAND** 4 kHz passband **OPUS_BANDWIDTH_MEDIUMBAND** 6 kHz passband **OPUS_BANDWIDTH_WIDEBAND** 8 kHz passband **OPUS_BANDWIDTH_SUPERWIDEBAND** 12 kHz passband **OPUS_BANDWIDTH_FULLBAND** 20 kHz passband (default)

4.6.2.25 #define OPUS_SET_PACKET_LOSS_PERC(*x*)

Configures the encoder's expected packet loss percentage.

Higher values with trigger progressively more loss resistant behavior in the encoder at the expense of quality at a given bitrate in the lossless case, but greater quality under loss.

See Also

OPUS_GET_PACKET_LOSS_PERC

Parameters

in	*x*	opus_int32: Loss percentage in the range 0-100, inclusive (default: 0).

4.6.2.26 #define OPUS_SET_SIGNAL(*x*)

Configures the type of signal being encoded.

This is a hint which helps the encoder's mode selection.

See Also

OPUS_GET_SIGNAL

Parameters

in	*x*	opus_int32: Allowed values: **OPUS_AUTO** (default) **OPUS_SIGNAL_VOICE** Bias thresholds towards choosing LPC or Hybrid modes. **OPUS_SIGNAL_MUSIC** Bias thresholds towards choosing MDCT modes.

4.6.2.27 #define OPUS_SET_VBR(*x*)

Enables or disables variable bitrate (VBR) in the encoder.

The configured bitrate may not be met exactly because frames must be an integer number of bytes in length.

Warning

Only the MDCT mode of Opus can provide hard CBR behavior.

See Also

OPUS_GET_VBR
OPUS_SET_VBR_CONSTRAINT

Parameters

in	*x*	opus_int32: Allowed values: **0** Hard CBR. For LPC/hybrid modes at very low bit-rate, this can cause noticeable quality degradation. **1** VBR (default). The exact type of VBR is controlled by OPUS_SET_VBR_CONSTRAINT.

4.6.2.28 #define OPUS_SET_VBR_CONSTRAINT(*x*)

Enables or disables constrained VBR in the encoder.

This setting is ignored when the encoder is in CBR mode.

Warning

Only the MDCT mode of Opus currently heeds the constraint. Speech mode ignores it completely, hybrid mode may fail to obey it if the LPC layer uses more bitrate than the constraint would have permitted.

See Also

OPUS_GET_VBR_CONSTRAINT
OPUS_SET_VBR

Parameters

in	*x*	opus_int32: Allowed values: **0** Unconstrained VBR. **1** Constrained VBR (default). This creates a maximum of one frame of buffering delay assuming a transport with a serialization speed of the nominal bitrate.

4.7 Generic CTLs

These macros are used with the `opus_decoder_ctl` and `opus_encoder_ctl` calls to generate a particular request.

Macros

- #define OPUS_RESET_STATE

  Resets the codec state to be equivalent to a freshly initialized state.
- #define OPUS_GET_FINAL_RANGE(x)

  Gets the final state of the codec's entropy coder.
- #define OPUS_GET_PITCH(x)

  Gets the pitch of the last decoded frame, if available.
- #define OPUS_GET_BANDWIDTH(x)

  Gets the encoder's configured bandpass or the decoder's last bandpass.

4.7.1 Detailed Description

These macros are used with the `opus_decoder_ctl` and `opus_encoder_ctl` calls to generate a particular request. When called on an `OpusDecoder` they apply to that particular decoder instance. When called on an `OpusEncoder` they apply to the corresponding setting on that encoder instance, if present.

Some usage examples:

```
int ret;
opus_int32 pitch;
ret = opus_decoder_ctl(dec_ctx, OPUS_GET_PITCH(&pitch));
if (ret == OPUS_OK) return ret;

opus_encoder_ctl(enc_ctx, OPUS_RESET_STATE);
opus_decoder_ctl(dec_ctx, OPUS_RESET_STATE);

opus_int32 enc_bw, dec_bw;
opus_encoder_ctl(enc_ctx, OPUS_GET_BANDWIDTH(&enc_bw));
opus_decoder_ctl(dec_ctx, OPUS_GET_BANDWIDTH(&dec_bw));
if (enc_bw != dec_bw) {
  printf("packet bandwidth mismatch!\n");
}
```

See Also

Opus Encoder, opus_decoder_ctl, opus_encoder_ctl, Decoder related CTLs, Encoder related CTLs

4.7.2 Macro Definition Documentation

4.7.2.1 #define OPUS_GET_BANDWIDTH(*x*)

Gets the encoder's configured bandpass or the decoder's last bandpass.

See Also

OPUS_SET_BANDWIDTH

Parameters

out	*x*	opus_int32 *: Returns one of the following values: **OPUS_AUTO** (default) **OPUS_BANDWIDTH_NARROWBAND** 4 kHz passband **OPUS_BANDWIDTH_MEDIUMBAND** 6 kHz passband **OPUS_BANDWIDTH_WIDEBAND** 8 kHz passband **OPUS_BANDWIDTH_SUPERWIDEBAND** 12 kHz passband **OPUS_BANDWIDTH_FULLBAND** 20 kHz passband

4.7.2.2 #define OPUS_GET_FINAL_RANGE(*x*)

Gets the final state of the codec's entropy coder.

This is used for testing purposes, The encoder and decoder state should be identical after coding a payload (assuming no data corruption or software bugs)

Parameters

out	*x*	opus_uint32 *: Entropy coder state

4.7.2.3 #define OPUS_GET_PITCH(*x*)

Gets the pitch of the last decoded frame, if available.

This can be used for any post-processing algorithm requiring the use of pitch, e.g. time stretching/shortening. If the last frame was not voiced, or if the pitch was not coded in the frame, then zero is returned.

This CTL is only implemented for decoder instances.

Parameters

out	*x*	opus_int32 *: pitch period at 48 kHz (or 0 if not available)

4.7.2.4 #define OPUS_RESET_STATE

Resets the codec state to be equivalent to a freshly initialized state.

This should be called when switching streams in order to prevent the back to back decoding from giving different results from one at a time decoding.

4.8 Decoder related CTLs

Macros

- #define OPUS_SET_GAIN(x)

  Configures decoder gain adjustment.
- #define OPUS_GET_GAIN(x)

  Gets the decoder's configured gain adjustment.

4.8.1 Detailed Description

See Also

Generic CTLs, Encoder related CTLs, Opus Decoder

4.8.2 Macro Definition Documentation

4.8.2.1 #define OPUS_GET_GAIN(*x*)

Gets the decoder's configured gain adjustment.

See Also

OPUS_SET_GAIN

Parameters

out	*x*	opus_int32 *: Amount to scale PCM signal by in Q8 dB units.

4.8.2.2 #define OPUS_SET_GAIN(*x*)

Configures decoder gain adjustment.

Scales the decoded output by a factor specified in Q8 dB units. This has a maximum range of -32768 to 32767 inclusive, and returns OPUS_BAD_ARG otherwise. The default is zero indicating no adjustment. This setting survives decoder reset.

gain = pow(10, x/(20.0*256))

Parameters

in	*x*	opus_int32: Amount to scale PCM signal by in Q8 dB units.

4.9 Opus library information functions

Functions

- const char * opus_strerror (int error)

  Converts an opus error code into a human readable string.
- const char * opus_get_version_string (void)

  Gets the libopus version string.

4.9.1 Detailed Description

4.9.2 Function Documentation

4.9.2.1 const char* opus_get_version_string (void)

Gets the libopus version string.

Returns

Version string

4.9.2.2 const char* opus_strerror (int *error*)

Converts an opus error code into a human readable string.

Parameters

in	*error*	int: Error number

Returns

Error string

4.10 Multistream specific encoder and decoder CTLs

These are convenience macros that are specific to the opus_multistream_encoder_ctl() and opus_multistream_decoder_ctl() interface.

Macros

- #define OPUS_MULTISTREAM_GET_ENCODER_STATE(x, y)

  Gets the encoder state for an individual stream of a multistream encoder.
- #define OPUS_MULTISTREAM_GET_DECODER_STATE(x, y)

  Gets the decoder state for an individual stream of a multistream decoder.

4.10.1 Detailed Description

These are convenience macros that are specific to the opus_multistream_encoder_ctl() and opus_multistream_decoder_ctl() interface. The CTLs from Generic CTLs, Encoder related CTLs, and Decoder related CTLs may be applied to a multistream encoder or decoder as well. In addition, you may retrieve the encoder or decoder state for an specific stream via OPUS_MULTISTREAM_GET_ENCODER_STATE or OPUS_MULTISTREAM_GET_DECODER_STATE and apply CTLs to it individually.

4.10.2 Macro Definition Documentation

4.10.2.1 #define OPUS_MULTISTREAM_GET_DECODER_STATE(*x*, *y*)

Gets the decoder state for an individual stream of a multistream decoder.

Parameters

in	*x*	opus_int32: The index of the stream whose decoder you wish to retrieve. This must be non-negative and less than the streams parameter used to initialize the decoder.
out	*y*	OpusDecoder**: Returns a pointer to the given decoder state.

Return values

OPUS_BAD_ARG	The index of the requested stream was out of range.

4.10.2.2 #define OPUS_MULTISTREAM_GET_ENCODER_STATE(*x*, *y*)

Gets the encoder state for an individual stream of a multistream encoder.

Parameters

in	*x*	opus_int32: The index of the stream whose encoder you wish to retrieve. This must be non-negative and less than the streams parameter used to initialize the encoder.
out	*y*	OpusEncoder**: Returns a pointer to the given encoder state.

Return values

OPUS_BAD_ARG	The index of the requested stream was out of range.

4.11 Opus Multistream API

The multistream API allows individual Opus streams to be combined into a single packet, enabling support for up to 255 channels.

Typedefs

- typedef struct OpusMSEncoder OpusMSEncoder

  Opus multistream encoder state.
- typedef struct OpusMSDecoder OpusMSDecoder

  Opus multistream decoder state.

Multistream encoder functions

- opus_int32 opus_multistream_encoder_get_size (int streams, int coupled_streams)

  Gets the size of an OpusMSEncoder structure.
- opus_int32 opus_multistream_surround_encoder_get_size (int channels, int mapping_family)
- OpusMSEncoder * opus_multistream_encoder_create (opus_int32 Fs, int channels, int streams, int coupled_streams, const unsigned char *mapping, int application, int *error)

  Allocates and initializes a multistream encoder state.
- OpusMSEncoder * opus_multistream_surround_encoder_create (opus_int32 Fs, int channels, int mapping_family, int *streams, int *coupled_streams, unsigned char *mapping, int application, int *error)
- int opus_multistream_encoder_init (OpusMSEncoder *st, opus_int32 Fs, int channels, int streams, int coupled_streams, const unsigned char *mapping, int application)

  Initialize a previously allocated multistream encoder state.
- int opus_multistream_surround_encoder_init (OpusMSEncoder *st, opus_int32 Fs, int channels, int mapping_family, int *streams, int *coupled_streams, unsigned char *mapping, int application)
- int opus_multistream_encode (OpusMSEncoder *st, const opus_int16 *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes a multistream Opus frame.
- int opus_multistream_encode_float (OpusMSEncoder *st, const float *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes a multistream Opus frame from floating point input.
- void opus_multistream_encoder_destroy (OpusMSEncoder *st)

  Frees an `OpusMSEncoder` allocated by opus_multistream_encoder_create().
- int opus_multistream_encoder_ctl (OpusMSEncoder *st, int request,...)

  Perform a CTL function on a multistream Opus encoder.

Multistream decoder functions

- opus_int32 opus_multistream_decoder_get_size (int streams, int coupled_streams)

  Gets the size of an `OpusMSDecoder` structure.
- OpusMSDecoder * opus_multistream_decoder_create (opus_int32 Fs, int channels, int streams, int coupled_streams, const unsigned char *mapping, int *error)

  Allocates and initializes a multistream decoder state.
- int opus_multistream_decoder_init (OpusMSDecoder *st, opus_int32 Fs, int channels, int streams, int coupled_streams, const unsigned char *mapping)

  Intialize a previously allocated decoder state object.

- int opus_multistream_decode (OpusMSDecoder *st, const unsigned char *data, opus_int32 len, opus_int16 *pcm, int frame_size, int decode_fec)

  Decode a multistream Opus packet.

- int opus_multistream_decode_float (OpusMSDecoder *st, const unsigned char *data, opus_int32 len, float *pcm, int frame_size, int decode_fec)

  Decode a multistream Opus packet with floating point output.

- int opus_multistream_decoder_ctl (OpusMSDecoder *st, int request,...)

  Perform a CTL function on a multistream Opus decoder.

- void opus_multistream_decoder_destroy (OpusMSDecoder *st)

  Frees an `OpusMSDecoder` allocated by opus_multistream_decoder_create().

4.11.1 Detailed Description

The multistream API allows individual Opus streams to be combined into a single packet, enabling support for up to 255 channels. Unlike an elementary Opus stream, the encoder and decoder must negotiate the channel configuration before the decoder can successfully interpret the data in the packets produced by the encoder. Some basic information, such as packet duration, can be computed without any special negotiation.

The format for multistream Opus packets is defined in the `Ogg encapsulation specification` and is based on the self-delimited Opus framing described in Appendix B of `RFC 6716`. Normal Opus packets are just a degenerate case of multistream Opus packets, and can be encoded or decoded with the multistream API by setting `streams` to `1` when initializing the encoder or decoder.

Multistream Opus streams can contain up to 255 elementary Opus streams. These may be either "uncoupled" or "coupled", indicating that the decoder is configured to decode them to either 1 or 2 channels, respectively. The streams are ordered so that all coupled streams appear at the beginning.

A `mapping` table defines which decoded channel `i` should be used for each input/output (I/O) channel `j`. This table is typically provided as an unsigned char array. Let `i = mapping[j]` be the index for I/O channel `j`. If `i < 2*coupled_streams`, then I/O channel `j` is encoded as the left channel of stream `(i/2)` if `i` is even, or as the right channel of stream `(i/2)` if `i` is odd. Otherwise, I/O channel `j` is encoded as mono in stream `(i - coupled_streams)`, unless it has the special value 255, in which case it is omitted from the encoding entirely (the decoder will reproduce it as silence). Each value `i` must either be the special value 255 or be less than `streams + coupled_streams`.

The output channels specified by the encoder should use the `Vorbis channel ordering`. A decoder may wish to apply an additional permutation to the mapping the encoder used to achieve a different output channel order (e.g. for outputing in WAV order).

Each multistream packet contains an Opus packet for each stream, and all of the Opus packets in a single multistream packet must have the same duration. Therefore the duration of a multistream packet can be extracted from the TOC sequence of the first stream, which is located at the beginning of the packet, just like an elementary Opus stream:

```
int nb_samples;
int nb_frames;
nb_frames = opus_packet_get_nb_frames(data, len);
if (nb_frames < 1)
  return nb_frames;
nb_samples = opus_packet_get_samples_per_frame(data, 48000) * nb_frames;
```

The general encoding and decoding process proceeds exactly the same as in the normal Opus Encoder and Opus Decoder APIs. See their documentation for an overview of how to use the corresponding multistream functions.

4.11.2 Typedef Documentation

4.11.2.1 typedef struct OpusMSDecoder OpusMSDecoder

Opus multistream decoder state.

This contains the complete state of a multistream Opus decoder. It is position independent and can be freely copied.

See Also

opus_multistream_decoder_create
opus_multistream_decoder_init

4.11.2.2 typedef struct OpusMSEncoder OpusMSEncoder

Opus multistream encoder state.

This contains the complete state of a multistream Opus encoder. It is position independent and can be freely copied.

See Also

opus_multistream_encoder_create
opus_multistream_encoder_init

4.11.3 Function Documentation

4.11.3.1 int opus_multistream_decode (OpusMSDecoder * *st*, const unsigned char * *data*, opus_int32 *len*, opus_int16 * *pcm*, int *frame_size*, int *decode_fec*)

Decode a multistream Opus packet.

Parameters

	st	`OpusMSDecoder*`: Multistream decoder state.
`in`	*data*	`const unsigned char*`: Input payload. Use a `NULL` pointer to indicate packet loss.
	len	`opus_int32`: Number of bytes in payload.
`out`	*pcm*	`opus_int16*`: Output signal, with interleaved samples. This must contain room for `frame_size*channels` samples.
	frame_size	`int`: The number of samples per channel of available space in *pcm*. If this is less than the maximum packet duration (120 ms; 5760 for 48kHz), this function will not be capable of decoding some packets. In the case of PLC (data==NULL) or FEC (decode_fec=1), then frame_size needs to be exactly the duration of audio that is missing, otherwise the decoder will not be in the optimal state to decode the next incoming packet. For the PLC and FEC cases, frame_size **must** be a multiple of 2.5 ms.
	decode_fec	`int`: Flag (0 or 1) to request that any in-band forward error correction data be decoded. If no such data is available, the frame is decoded as if it were lost.

Returns

Number of samples decoded on success or a negative error code (see Error codes) on failure.

4.11.3.2 int opus_multistream_decode_float (OpusMSDecoder * *st*, const unsigned char * *data*, opus_int32 *len*, float * *pcm*, int *frame_size*, int *decode_fec*)

Decode a multistream Opus packet with floating point output.

Parameters

	st	`OpusMSDecoder*`: Multistream decoder state.
in	*data*	`const unsigned char*`: Input payload. Use a `NULL` pointer to indicate packet loss.
	len	`opus_int32`: Number of bytes in payload.
out	*pcm*	`opus_int16*`: Output signal, with interleaved samples. This must contain room for `frame_size*channels` samples.
	frame_size	`int`: The number of samples per channel of available space in *pcm*. If this is less than the maximum packet duration (120 ms; 5760 for 48kHz), this function will not be capable of decoding some packets. In the case of PLC (data==NULL) or FEC (decode_fec=1), then frame_size needs to be exactly the duration of audio that is missing, otherwise the decoder will not be in the optimal state to decode the next incoming packet. For the PLC and FEC cases, frame_size **must** be a multiple of 2.5 ms.
	decode_fec	`int`: Flag (0 or 1) to request that any in-band forward error correction data be decoded. If no such data is available, the frame is decoded as if it were lost.

Returns

Number of samples decoded on success or a negative error code (see Error codes) on failure.

4.11.3.3 OpusMSDecoder* opus_multistream_decoder_create (opus_int32 *Fs*, int *channels*, int *streams*, int *coupled_streams*, const unsigned char * *mapping*, int * *error*)

Allocates and initializes a multistream decoder state.

Call opus_multistream_decoder_destroy() to release this object when finished.

Parameters

	Fs	`opus_int32`: Sampling rate to decode at (in Hz). This must be one of 8000, 12000, 16000, 24000, or 48000.
	channels	`int`: Number of channels to output. This must be at most 255. It may be different from the number of coded channels (`streams + coupled_streams`).
	streams	`int`: The total number of streams coded in the input. This must be no more than 255.
	coupled_streams	`int`: Number of streams to decode as coupled (2 channel) streams. This must be no larger than the total number of streams. Additionally, The total number of coded channels (`streams + coupled_streams`) must be no more than 255.
in	*mapping*	`const unsigned char[channels]`: Mapping from coded channels to output channels, as described in Opus Multistream API.
out	*error*	`int *`: Returns OPUS_OK on success, or an error code (see Error codes) on failure.

4.11.3.4 int opus_multistream_decoder_ctl (OpusMSDecoder * *st*, int *request*, ...)

Perform a CTL function on a multistream Opus decoder.

Generally the request and subsequent arguments are generated by a convenience macro.

Parameters

st	`OpusMSDecoder*`: Multistream decoder state.
request	This and all remaining parameters should be replaced by one of the convenience macros in Generic CTLs, Decoder related CTLs, or Multistream specific encoder and decoder CTLs.

See Also

Generic CTLs
Decoder related CTLs
Multistream specific encoder and decoder CTLs

4.11.3.5 void opus_multistream_decoder_destroy (OpusMSDecoder * *st*)

Frees an `OpusMSDecoder` allocated by opus_multistream_decoder_create().

Parameters

st	`OpusMSDecoder`: Multistream decoder state to be freed.

4.11.3.6 opus_int32 opus_multistream_decoder_get_size (int *streams*, int *coupled_streams*)

Gets the size of an `OpusMSDecoder` structure.

Parameters

streams	`int`: The total number of streams coded in the input. This must be no more than 255.
coupled_streams	`int`: Number streams to decode as coupled (2 channel) streams. This must be no larger than the total number of streams. Additionally, The total number of coded channels (`streams + coupled_streams`) must be no more than 255.

Returns

The size in bytes on success, or a negative error code (see Error codes) on error.

4.11.3.7 int opus_multistream_decoder_init (OpusMSDecoder * *st*, opus_int32 *Fs*, int *channels*, int *streams*, int *coupled_streams*, const unsigned char * *mapping*)

Intialize a previously allocated decoder state object.

The memory pointed to by *st* must be at least the size returned by opus_multistream_encoder_get_size(). This is intended for applications which use their own allocator instead of malloc. To reset a previously initialized state, use the OPUS_RESET_STATE CTL.

See Also

opus_multistream_decoder_create
opus_multistream_deocder_get_size

Parameters

	st	`OpusMSEncoder*`: Multistream encoder state to initialize.
	Fs	`opus_int32`: Sampling rate to decode at (in Hz). This must be one of 8000, 12000, 16000, 24000, or 48000.
	channels	`int`: Number of channels to output. This must be at most 255. It may be different from the number of coded channels (`streams + coupled_streams`).
	streams	`int`: The total number of streams coded in the input. This must be no more than 255.
	coupled_streams	`int`: Number of streams to decode as coupled (2 channel) streams. This must be no larger than the total number of streams. Additionally, The total number of coded channels (`streams + coupled_streams`) must be no more than 255.
`in`	*mapping*	`const unsigned char[channels]`: Mapping from coded channels to output channels, as described in Opus Multistream API.

Returns

OPUS_OK on success, or an error code (see Error codes) on failure.

4.11.3.8 int opus_multistream_encode (OpusMSEncoder * *st*, const opus_int16 * *pcm*, int *frame_size*, unsigned char * *data*, opus_int32 *max_data_bytes*)

Encodes a multistream Opus frame.

Parameters

	st	`OpusMSEncoder*`: Multistream encoder state.
`in`	*pcm*	`const opus_int16*`: The input signal as interleaved samples. This must contain `frame_size*channels` samples.
	frame_size	`int`: Number of samples per channel in the input signal. This must be an Opus frame size for the encoder's sampling rate. For example, at 48 kHz the permitted values are 120, 240, 480, 960, 1920, and 2880. Passing in a duration of less than 10 ms (480 samples at 48 kHz) will prevent the encoder from using the LPC or hybrid modes.
`out`	*data*	`unsigned char*`: Output payload. This must contain storage for at least *max_data_bytes*.
`in`	*max_data_bytes*	`opus_int32`: Size of the allocated memory for the output payload. This may be used to impose an upper limit on the instant bitrate, but should not be used as the only bitrate control. Use OPUS_SET_BITRATE to control the bitrate.

Returns

The length of the encoded packet (in bytes) on success or a negative error code (see Error codes) on failure.

4.11.3.9 int opus_multistream_encode_float (OpusMSEncoder * *st*, const float * *pcm*, int *frame_size*, unsigned char * *data*, opus_int32 *max_data_bytes*)

Encodes a multistream Opus frame from floating point input.

Parameters

	st	`OpusMSEncoder*`: Multistream encoder state.
in	*pcm*	`const float*`: The input signal as interleaved samples with a normal range of +/-1.0. Samples with a range beyond +/-1.0 are supported but will be clipped by decoders using the integer API and should only be used if it is known that the far end supports extended dynamic range. This must contain `frame_size*channels` samples.
	frame_size	`int`: Number of samples per channel in the input signal. This must be an Opus frame size for the encoder's sampling rate. For example, at 48 kHz the permitted values are 120, 240, 480, 960, 1920, and 2880. Passing in a duration of less than 10 ms (480 samples at 48 kHz) will prevent the encoder from using the LPC or hybrid modes.
out	*data*	`unsigned char*`: Output payload. This must contain storage for at least *max_data_bytes*.
in	*max_data_bytes*	`opus_int32`: Size of the allocated memory for the output payload. This may be used to impose an upper limit on the instant bitrate, but should not be used as the only bitrate control. Use OPUS_SET_BITRATE to control the bitrate.

Returns

The length of the encoded packet (in bytes) on success or a negative error code (see Error codes) on failure.

4.11.3.10 OpusMSEncoder* opus_multistream_encoder_create (opus_int32 *Fs*, int *channels*, int *streams*, int *coupled_streams*, const unsigned char * *mapping*, int *application*, int * *error*)

Allocates and initializes a multistream encoder state.

Call opus_multistream_encoder_destroy() to release this object when finished.

Parameters

	Fs	`opus_int32`: Sampling rate of the input signal (in Hz). This must be one of 8000, 12000, 16000, 24000, or 48000.
	channels	`int`: Number of channels in the input signal. This must be at most 255. It may be greater than the number of coded channels (`streams + coupled_streams`).
	streams	`int`: The total number of streams to encode from the input. This must be no more than the number of channels.
	coupled_streams	`int`: Number of coupled (2 channel) streams to encode. This must be no larger than the total number of streams. Additionally, The total number of encoded channels (`streams + coupled_streams`) must be no more than the number of input channels.

in	*mapping*	`const unsigned char[channels]`: Mapping from encoded channels to input channels, as described in Opus Multistream API. As an extra constraint, the multistream encoder does not allow encoding coupled streams for which one channel is unused since this is never a good idea.
	application	`int`: The target encoder application. This must be one of the following: **OPUS_APPLICATION_VOIP** Process signal for improved speech intelligibility. **OPUS_APPLICATION_AUDIO** Favor faithfulness to the original input. **OPUS_APPLICATION_RESTRICTED_LOWDELAY** Configure the minimum possible coding delay by disabling certain modes of operation.
out	*error*	`int *`: Returns OPUS_OK on success, or an error code (see Error codes) on failure.

4.11.3.11 int opus_multistream_encoder_ctl (OpusMSEncoder * *st*, int *request*, ...)

Perform a CTL function on a multistream Opus encoder.

Generally the request and subsequent arguments are generated by a convenience macro.

Parameters

st	`OpusMSEncoder*`: Multistream encoder state.
request	This and all remaining parameters should be replaced by one of the convenience macros in Generic CTLs, Encoder related CTLs, or Multistream specific encoder and decoder CTLs.

See Also

Generic CTLs
Encoder related CTLs
Multistream specific encoder and decoder CTLs

4.11.3.12 void opus_multistream_encoder_destroy (OpusMSEncoder * *st*)

Frees an `OpusMSEncoder` allocated by opus_multistream_encoder_create().

Parameters

st	`OpusMSEncoder*`: Multistream encoder state to be freed.

4.11.3.13 opus_int32 opus_multistream_encoder_get_size (int *streams*, int *coupled_streams*)

Gets the size of an OpusMSEncoder structure.

Parameters

streams	`int`: The total number of streams to encode from the input. This must be no more than 255.
coupled_streams	`int`: Number of coupled (2 channel) streams to encode. This must be no larger than the total number of streams. Additionally, The total number of encoded channels (`streams + coupled_streams`) must be no more than 255.

Returns

The size in bytes on success, or a negative error code (see Error codes) on error.

4.11.3.14 int opus_multistream_encoder_init (OpusMSEncoder * *st*, opus_int32 *Fs*, int *channels*, int *streams*, int *coupled_streams*, const unsigned char * *mapping*, int *application*)

Initialize a previously allocated multistream encoder state.

The memory pointed to by *st* must be at least the size returned by opus_multistream_encoder_get_size(). This is intended for applications which use their own allocator instead of malloc. To reset a previously initialized state, use the OPUS_RESET_STATE CTL.

See Also

opus_multistream_encoder_create
opus_multistream_encoder_get_size

Parameters

	st	`OpusMSEncoder*`: Multistream encoder state to initialize.
	Fs	`opus_int32`: Sampling rate of the input signal (in Hz). This must be one of 8000, 12000, 16000, 24000, or 48000.
	channels	`int`: Number of channels in the input signal. This must be at most 255. It may be greater than the number of coded channels (`streams + coupled_streams`).
	streams	`int`: The total number of streams to encode from the input. This must be no more than the number of channels.
	coupled_streams	`int`: Number of coupled (2 channel) streams to encode. This must be no larger than the total number of streams. Additionally, The total number of encoded channels (`streams + coupled_streams`) must be no more than the number of input channels.
`in`	*mapping*	`const unsigned char[channels]`: Mapping from encoded channels to input channels, as described in Opus Multistream API. As an extra constraint, the multistream encoder does not allow encoding coupled streams for which one channel is unused since this is never a good idea.
	application	`int`: The target encoder application. This must be one of the following: **OPUS_APPLICATION_VOIP** Process signal for improved speech intelligibility. **OPUS_APPLICATION_AUDIO** Favor faithfulness to the original input. **OPUS_APPLICATION_RESTRICTED_LOWDELAY** Configure the minimum possible coding delay by disabling certain modes of operation.

Returns

OPUS_OK on success, or an error code (see Error codes) on failure.

4.11.3.15 OpusMSEncoder* opus_multistream_surround_encoder_create (opus_int32 *Fs*, int *channels*, int *mapping_family*, int * *streams*, int * *coupled_streams*, unsigned char * *mapping*, int *application*, int * *error*)

4.11.3.16 opus_int32 opus_multistream_surround_encoder_get_size (int *channels*, int *mapping_family*)

4.11.3.17 int opus_multistream_surround_encoder_init (OpusMSEncoder * *st*, opus_int32 *Fs*, int *channels*, int *mapping_family*, int * *streams*, int * *coupled_streams*, unsigned char * *mapping*, int *application*)

4.12 Opus Custom

Opus Custom is an optional part of the Opus specification and reference implementation which uses a distinct API from the regular API and supports frame sizes that are not normally supported. Use of Opus Custom is discouraged for all but very special applications for which a frame size different from 2.5, 5, 10, or 20 ms is needed (for either complexity or latency reasons) and where interoperability is less important.

Typedefs

- typedef struct OpusCustomEncoder OpusCustomEncoder

  Contains the state of an encoder.
- typedef struct OpusCustomDecoder OpusCustomDecoder

  State of the decoder.
- typedef struct OpusCustomMode OpusCustomMode

  The mode contains all the information necessary to create an encoder.

Functions

- OpusCustomMode ∗ opus_custom_mode_create (opus_int32 Fs, int frame_size, int ∗error)

  Creates a new mode struct.
- void opus_custom_mode_destroy (OpusCustomMode ∗mode)

  Destroys a mode struct.
- int opus_custom_encoder_get_size (const OpusCustomMode ∗mode, int channels)

  Gets the size of an OpusCustomEncoder structure.
- OpusCustomEncoder ∗ opus_custom_encoder_create (const OpusCustomMode ∗mode, int channels, int ∗error)

  Creates a new encoder state.
- int opus_custom_encoder_init (OpusCustomEncoder ∗st, const OpusCustomMode ∗mode, int channels)

  Initializes a previously allocated encoder state The memory pointed to by st must be the size returned by opus_custom_encoder_get_size.
- void opus_custom_encoder_destroy (OpusCustomEncoder ∗st)

  Destroys a an encoder state.
- int opus_custom_encode_float (OpusCustomEncoder ∗st, const float ∗pcm, int frame_size, unsigned char ∗compressed, int maxCompressedBytes)

  Encodes a frame of audio.
- int opus_custom_encode (OpusCustomEncoder ∗st, const opus_int16 ∗pcm, int frame_size, unsigned char ∗compressed, int maxCompressedBytes)

  Encodes a frame of audio.
- int opus_custom_encoder_ctl (OpusCustomEncoder ∗OPUS_RESTRICT st, int request,...)

  Perform a CTL function on an Opus custom encoder.
- int opus_custom_decoder_get_size (const OpusCustomMode ∗mode, int channels)

  Gets the size of an OpusCustomDecoder structure.
- OpusCustomDecoder ∗ opus_custom_decoder_create (const OpusCustomMode ∗mode, int channels, int ∗error)

  Creates a new decoder state.
- int opus_custom_decoder_init (OpusCustomDecoder ∗st, const OpusCustomMode ∗mode, int channels)

  Initializes a previously allocated decoder state The memory pointed to by st must be the size returned by opus_custom_decoder_get_size.

- void opus_custom_decoder_destroy (OpusCustomDecoder *st)

  Destroys a an decoder state.
- int opus_custom_decode_float (OpusCustomDecoder *st, const unsigned char *data, int len, float *pcm, int frame_size)

  Decode an opus custom frame with floating point output.
- int opus_custom_decode (OpusCustomDecoder *st, const unsigned char *data, int len, opus_int16 *pcm, int frame_size)

  Decode an opus custom frame.
- int opus_custom_decoder_ctl (OpusCustomDecoder *OPUS_RESTRICT st, int request,...)

  Perform a CTL function on an Opus custom decoder.

4.12.1 Detailed Description

Opus Custom is an optional part of the Opus specification and reference implementation which uses a distinct API from the regular API and supports frame sizes that are not normally supported. Use of Opus Custom is discouraged for all but very special applications for which a frame size different from 2.5, 5, 10, or 20 ms is needed (for either complexity or latency reasons) and where interoperability is less important. In addition to the interoperability limitations the use of Opus custom disables a substantial chunk of the codec and generally lowers the quality available at a given bitrate. Normally when an application needs a different frame size from the codec it should buffer to match the sizes but this adds a small amount of delay which may be important in some very low latency applications. Some transports (especially constant rate RF transports) may also work best with frames of particular durations.

Libopus only supports custom modes if they are enabled at compile time.

The Opus Custom API is similar to the regular API but the opus_encoder_create and opus_decoder_create calls take an additional mode parameter which is a structure produced by a call to opus_custom_mode_create. Both the encoder and decoder must create a mode using the same sample rate (fs) and frame size (frame size) so these parameters must either be signaled out of band or fixed in a particular implementation.

Similar to regular Opus the custom modes support on the fly frame size switching, but the sizes available depend on the particular frame size in use. For some initial frame sizes on a single on the fly size is available.

4.12.2 Typedef Documentation

4.12.2.1 typedef struct OpusCustomDecoder OpusCustomDecoder

State of the decoder.

One decoder state is needed for each stream. It is initialized once at the beginning of the stream. Do *not* re-initialize the state for every frame. Decoder state

4.12.2.2 typedef struct OpusCustomEncoder OpusCustomEncoder

Contains the state of an encoder.

One encoder state is needed for each stream. It is initialized once at the beginning of the stream. Do *not* re-initialize the state for every frame. Encoder state

4.12.2.3 typedef struct OpusCustomMode OpusCustomMode

The mode contains all the information necessary to create an encoder.

Both the encoder and decoder need to be initialized with exactly the same mode, otherwise the output will be corrupted.
Mode configuration

4.12.3 Function Documentation

4.12.3.1 int opus_custom_decode (OpusCustomDecoder * *st*, const unsigned char * *data*, int *len*, opus_int16 * *pcm*, int *frame_size*)

Decode an opus custom frame.

Parameters

in	*st*	OpusCustomDecoder*: Decoder state
in	*data*	char*: Input payload. Use a NULL pointer to indicate packet loss
in	*len*	int: Number of bytes in payload
out	*pcm*	opus_int16*: Output signal (interleaved if 2 channels). length is frame_size*channels*sizeof(opus_int16)
in	*frame_size*	Number of samples per channel of available space in *pcm.

Returns

Number of decoded samples or Error codes

4.12.3.2 int opus_custom_decode_float (OpusCustomDecoder * *st*, const unsigned char * *data*, int *len*, float * *pcm*, int *frame_size*)

Decode an opus custom frame with floating point output.

Parameters

in	*st*	OpusCustomDecoder*: Decoder state
in	*data*	char*: Input payload. Use a NULL pointer to indicate packet loss
in	*len*	int: Number of bytes in payload
out	*pcm*	float*: Output signal (interleaved if 2 channels). length is frame_size*channels*sizeof(float)
in	*frame_size*	Number of samples per channel of available space in *pcm.

Returns

Number of decoded samples or Error codes

4.12.3.3 OpusCustomDecoder* opus_custom_decoder_create (const OpusCustomMode * *mode*, int *channels*, int * *error*)

Creates a new decoder state.

Each stream needs its own decoder state (can't be shared across simultaneous streams).

Parameters

in	*mode*	OpusCustomMode: Contains all the information about the characteristics of the stream (must be the same characteristics as used for the encoder)
in	*channels*	int: Number of channels
out	*error*	int*: Returns an error code

Returns

Newly created decoder state.

4.12.3.4 int opus_custom_decoder_ctl (OpusCustomDecoder *OPUS_RESTRICT *st*, int *request*, ...)

Perform a CTL function on an Opus custom decoder.

Generally the request and subsequent arguments are generated by a convenience macro.

See Also

Generic CTLs

4.12.3.5 void opus_custom_decoder_destroy (OpusCustomDecoder * *st*)

Destroys a an decoder state.

Parameters

in	*st*	OpusCustomDecoder*: State to be freed.

4.12.3.6 int opus_custom_decoder_get_size (const OpusCustomMode * *mode*, int *channels*)

Gets the size of an OpusCustomDecoder structure.

Parameters

in	*mode*	OpusCustomMode *: Mode configuration
in	*channels*	int: Number of channels

Returns

size

4.12.3.7 int opus_custom_decoder_init (OpusCustomDecoder * *st*, const OpusCustomMode * *mode*, int *channels*)

Initializes a previously allocated decoder state The memory pointed to by st must be the size returned by opus_custom_decoder_get_size.

This is intended for applications which use their own allocator instead of malloc.

See Also

opus_custom_decoder_create(),opus_custom_decoder_get_size() To reset a previously initialized state use the OPUS_RESET_STATE CTL.

Parameters

in	*st*	OpusCustomDecoder*: Decoder state
in	*mode*	OpusCustomMode *: Contains all the information about the characteristics of the stream (must be the same characteristics as used for the encoder)
in	*channels*	int: Number of channels

Returns

OPUS_OK Success or Error codes

4.12.3.8 int opus_custom_encode (OpusCustomEncoder * *st*, const opus_int16 * *pcm*, int *frame_size*, unsigned char * *compressed*, int *maxCompressedBytes*)

Encodes a frame of audio.

Parameters

in	*st*	OpusCustomEncoder*: Encoder state
in	*pcm*	opus_int16*: PCM audio in signed 16-bit format (native endian). There must be exactly frame_size samples per channel.
in	*frame_size*	int: Number of samples per frame of input signal
out	*compressed*	char *: The compressed data is written here. This may not alias pcm and must be at least maxCompressedBytes long.
in	*maxCompressedBytes*	int: Maximum number of bytes to use for compressing the frame (can change from one frame to another)

Returns

Number of bytes written to "compressed". If negative, an error has occurred (see error codes). It is IMPORTANT that the length returned be somehow transmitted to the decoder. Otherwise, no decoding is possible.

4.12.3.9 int opus_custom_encode_float (OpusCustomEncoder * *st*, const float * *pcm*, int *frame_size*, unsigned char * *compressed*, int *maxCompressedBytes*)

Encodes a frame of audio.

Parameters

in	*st*	OpusCustomEncoder*: Encoder state
in	*pcm*	float*: PCM audio in float format, with a normal range of +/-1.0. Samples with a range beyond +/-1.0 are supported but will be clipped by decoders using the integer API and should only be used if it is known that the far end supports extended dynamic range. There must be exactly frame_size samples per channel.
in	*frame_size*	int: Number of samples per frame of input signal
out	*compressed*	char *: The compressed data is written here. This may not alias pcm and must be at least maxCompressedBytes long.

in	*max-Compressed-Bytes*	int: Maximum number of bytes to use for compressing the frame (can change from one frame to another)

Returns

Number of bytes written to "compressed". If negative, an error has occurred (see error codes). It is IMPORTANT that the length returned be somehow transmitted to the decoder. Otherwise, no decoding is possible.

4.12.3.10 OpusCustomEncoder* opus_custom_encoder_create (const OpusCustomMode * *mode,* int *channels,* int * *error*)

Creates a new encoder state.

Each stream needs its own encoder state (can't be shared across simultaneous streams).

Parameters

in	*mode*	OpusCustomMode*: Contains all the information about the characteristics of the stream (must be the same characteristics as used for the decoder)
in	*channels*	int: Number of channels
out	*error*	int*: Returns an error code

Returns

Newly created encoder state.

4.12.3.11 int opus_custom_encoder_ctl (OpusCustomEncoder *OPUS_RESTRICT *st,* int *request,* ...)

Perform a CTL function on an Opus custom encoder.

Generally the request and subsequent arguments are generated by a convenience macro.

See Also

Encoder related CTLs

4.12.3.12 void opus_custom_encoder_destroy (OpusCustomEncoder * *st*)

Destroys a an encoder state.

Parameters

in	*st*	OpusCustomEncoder*: State to be freed.

4.12.3.13 int opus_custom_encoder_get_size (const OpusCustomMode * *mode,* int *channels*)

Gets the size of an OpusCustomEncoder structure.

Parameters

in	*mode*	OpusCustomMode *: Mode configuration
in	*channels*	int: Number of channels

Returns

size

4.12.3.14 int opus_custom_encoder_init (OpusCustomEncoder * *st*, const OpusCustomMode * *mode*, int *channels*)

Initializes a previously allocated encoder state The memory pointed to by st must be the size returned by opus_custom_encoder_get_size.

This is intended for applications which use their own allocator instead of malloc.

See Also

opus_custom_encoder_create(),opus_custom_encoder_get_size() To reset a previously initialized state use the OPUS_RESET_STATE CTL.

Parameters

in	*st*	OpusCustomEncoder*: Encoder state
in	*mode*	OpusCustomMode *: Contains all the information about the characteristics of the stream (must be the same characteristics as used for the decoder)
in	*channels*	int: Number of channels

Returns

OPUS_OK Success or Error codes

4.12.3.15 OpusCustomMode* opus_custom_mode_create (opus_int32 *Fs*, int *frame_size*, int * *error*)

Creates a new mode struct.

This will be passed to an encoder or decoder. The mode MUST NOT BE DESTROYED until the encoders and decoders that use it are destroyed as well.

Parameters

in	*Fs*	int: Sampling rate (8000 to 96000 Hz)
in	*frame_size*	int: Number of samples (per channel) to encode in each packet (64 - 1024, prime factorization must contain zero or more 2s, 3s, or 5s and no other primes)
out	*error*	int*: Returned error code (if NULL, no error will be returned)

Returns

A newly created mode

4.12.3.16 void opus_custom_mode_destroy (OpusCustomMode * *mode*)

Destroys a mode struct.

Only call this after all encoders and decoders using this mode are destroyed as well.

Parameters

in	*mode*	OpusCustomMode*: Mode to be freed.

Chapter 5

File Documentation

5.1 opus.h File Reference

Opus reference implementation API.

```
#include "opus_types.h"
#include "opus_defines.h"
```

Typedefs

- typedef struct OpusEncoder OpusEncoder

  Opus encoder state.
- typedef struct OpusDecoder OpusDecoder

  Opus decoder state.
- typedef struct OpusRepacketizer OpusRepacketizer

Functions

- int opus_encoder_get_size (int channels)

  Gets the size of an `OpusEncoder` structure.
- OpusEncoder * opus_encoder_create (opus_int32 Fs, int channels, int application, int *error)

  Allocates and initializes an encoder state.
- int opus_encoder_init (OpusEncoder *st, opus_int32 Fs, int channels, int application)

  Initializes a previously allocated encoder state The memory pointed to by st must be at least the size returned by opus_encoder_get_size().
- opus_int32 opus_encode (OpusEncoder *st, const opus_int16 *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes an Opus frame.
- opus_int32 opus_encode_float (OpusEncoder *st, const float *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes an Opus frame from floating point input.
- void opus_encoder_destroy (OpusEncoder *st)

  Frees an `OpusEncoder` allocated by opus_encoder_create().
- int opus_encoder_ctl (OpusEncoder *st, int request,...)

Perform a CTL function on an Opus encoder.

- int opus_decoder_get_size (int channels)

  Gets the size of an `OpusDecoder` structure.
- OpusDecoder * opus_decoder_create (opus_int32 Fs, int channels, int *error)

  Allocates and initializes a decoder state.
- int opus_decoder_init (OpusDecoder *st, opus_int32 Fs, int channels)

  Initializes a previously allocated decoder state.
- int opus_decode (OpusDecoder *st, const unsigned char *data, opus_int32 len, opus_int16 *pcm, int frame_size, int decode_fec)

  Decode an Opus packet.
- int opus_decode_float (OpusDecoder *st, const unsigned char *data, opus_int32 len, float *pcm, int frame_size, int decode_fec)

  Decode an Opus packet with floating point output.
- int opus_decoder_ctl (OpusDecoder *st, int request,...)

  Perform a CTL function on an Opus decoder.
- void opus_decoder_destroy (OpusDecoder *st)

  Frees an `OpusDecoder` allocated by opus_decoder_create().
- int opus_packet_parse (const unsigned char *data, opus_int32 len, unsigned char *out_toc, const unsigned char *frames[48], opus_int16 size[48], int *payload_offset)

  Parse an opus packet into one or more frames.
- int opus_packet_get_bandwidth (const unsigned char *data)

  Gets the bandwidth of an Opus packet.
- int opus_packet_get_samples_per_frame (const unsigned char *data, opus_int32 Fs)

  Gets the number of samples per frame from an Opus packet.
- int opus_packet_get_nb_channels (const unsigned char *data)

  Gets the number of channels from an Opus packet.
- int opus_packet_get_nb_frames (const unsigned char packet[], opus_int32 len)

  Gets the number of frames in an Opus packet.
- int opus_packet_get_nb_samples (const unsigned char packet[], opus_int32 len, opus_int32 Fs)

  Gets the number of samples of an Opus packet.
- int opus_decoder_get_nb_samples (const OpusDecoder *dec, const unsigned char packet[], opus_int32 len)

  Gets the number of samples of an Opus packet.
- int opus_repacketizer_get_size (void)

  Gets the size of an `OpusRepacketizer` structure.
- OpusRepacketizer * opus_repacketizer_init (OpusRepacketizer *rp)

  (Re)initializes a previously allocated repacketizer state.
- OpusRepacketizer * opus_repacketizer_create (void)

  Allocates memory and initializes the new repacketizer with opus_repacketizer_init().
- void opus_repacketizer_destroy (OpusRepacketizer *rp)

  Frees an `OpusRepacketizer` allocated by opus_repacketizer_create().
- int opus_repacketizer_cat (OpusRepacketizer *rp, const unsigned char *data, opus_int32 len)

  Add a packet to the current repacketizer state.
- opus_int32 opus_repacketizer_out_range (OpusRepacketizer *rp, int begin, int end, unsigned char *data, opus_int32 maxlen)

  Construct a new packet from data previously submitted to the repacketizer state via opus_repacketizer_cat().
- int opus_repacketizer_get_nb_frames (OpusRepacketizer *rp)

  Return the total number of frames contained in packet data submitted to the repacketizer state so far via opus_repacketizer_cat() since the last call to opus_repacketizer_init() or opus_repacketizer_create().
- opus_int32 opus_repacketizer_out (OpusRepacketizer *rp, unsigned char *data, opus_int32 maxlen)

  Construct a new packet from data previously submitted to the repacketizer state via opus_repacketizer_cat().

5.1.1 Detailed Description

Opus reference implementation API.

5.2 opus_custom.h File Reference

Opus-Custom reference implementation API.

```
#include "opus_defines.h"
```

Macros

- #define OPUS_CUSTOM_EXPORT
- #define OPUS_CUSTOM_EXPORT_STATIC

Typedefs

- typedef struct OpusCustomEncoder OpusCustomEncoder

  Contains the state of an encoder.
- typedef struct OpusCustomDecoder OpusCustomDecoder

  State of the decoder.
- typedef struct OpusCustomMode OpusCustomMode

  The mode contains all the information necessary to create an encoder.

Functions

- OpusCustomMode * opus_custom_mode_create (opus_int32 Fs, int frame_size, int *error)

  Creates a new mode struct.
- void opus_custom_mode_destroy (OpusCustomMode *mode)

  Destroys a mode struct.
- int opus_custom_encoder_get_size (const OpusCustomMode *mode, int channels)

  Gets the size of an OpusCustomEncoder structure.
- OpusCustomEncoder * opus_custom_encoder_create (const OpusCustomMode *mode, int channels, int *error)

  Creates a new encoder state.
- int opus_custom_encoder_init (OpusCustomEncoder *st, const OpusCustomMode *mode, int channels)

  Initializes a previously allocated encoder state The memory pointed to by st must be the size returned by opus_custom_encoder_get_size.
- void opus_custom_encoder_destroy (OpusCustomEncoder *st)

  Destroys a an encoder state.
- int opus_custom_encode_float (OpusCustomEncoder *st, const float *pcm, int frame_size, unsigned char *compressed, int maxCompressedBytes)

  Encodes a frame of audio.
- int opus_custom_encode (OpusCustomEncoder *st, const opus_int16 *pcm, int frame_size, unsigned char *compressed, int maxCompressedBytes)

  Encodes a frame of audio.

- int opus_custom_encoder_ctl (OpusCustomEncoder *OPUS_RESTRICT st, int request,...)

  Perform a CTL function on an Opus custom encoder.
- int opus_custom_decoder_get_size (const OpusCustomMode *mode, int channels)

  Gets the size of an OpusCustomDecoder structure.
- OpusCustomDecoder * opus_custom_decoder_create (const OpusCustomMode *mode, int channels, int *error)

  Creates a new decoder state.
- int opus_custom_decoder_init (OpusCustomDecoder *st, const OpusCustomMode *mode, int channels)

  Initializes a previously allocated decoder state The memory pointed to by st must be the size returned by opus_custom_decoder_get_size.
- void opus_custom_decoder_destroy (OpusCustomDecoder *st)

  Destroys a an decoder state.
- int opus_custom_decode_float (OpusCustomDecoder *st, const unsigned char *data, int len, float *pcm, int frame_size)

  Decode an opus custom frame with floating point output.
- int opus_custom_decode (OpusCustomDecoder *st, const unsigned char *data, int len, opus_int16 *pcm, int frame_size)

  Decode an opus custom frame.
- int opus_custom_decoder_ctl (OpusCustomDecoder *OPUS_RESTRICT st, int request,...)

  Perform a CTL function on an Opus custom decoder.

5.2.1 Detailed Description

Opus-Custom reference implementation API.

5.2.2 Macro Definition Documentation

5.2.2.1 #define OPUS_CUSTOM_EXPORT

5.2.2.2 #define OPUS_CUSTOM_EXPORT_STATIC

5.3 opus_defines.h File Reference

Opus reference implementation constants.

```
#include "opus_types.h"
```

Macros

- #define OPUS_OK

  No error.
- #define OPUS_BAD_ARG

  One or more invalid/out of range arguments.
- #define OPUS_BUFFER_TOO_SMALL

  The mode struct passed is invalid.
- #define OPUS_INTERNAL_ERROR

An internal error was detected.

- #define OPUS_INVALID_PACKET

  The compressed data passed is corrupted.

- #define OPUS_UNIMPLEMENTED

  Invalid/unsupported request number.

- #define OPUS_INVALID_STATE

  An encoder or decoder structure is invalid or already freed.

- #define OPUS_ALLOC_FAIL

  Memory allocation has failed.

- #define OPUS_GNUC_PREREQ(_maj, _min) 0
- #define OPUS_RESTRICT
- #define OPUS_WARN_UNUSED_RESULT
- #define OPUS_ARG_NONNULL(_x)
- #define OPUS_SET_APPLICATION_REQUEST 4000
- #define OPUS_GET_APPLICATION_REQUEST 4001
- #define OPUS_SET_BITRATE_REQUEST 4002
- #define OPUS_GET_BITRATE_REQUEST 4003
- #define OPUS_SET_MAX_BANDWIDTH_REQUEST 4004
- #define OPUS_GET_MAX_BANDWIDTH_REQUEST 4005
- #define OPUS_SET_VBR_REQUEST 4006
- #define OPUS_GET_VBR_REQUEST 4007
- #define OPUS_SET_BANDWIDTH_REQUEST 4008
- #define OPUS_GET_BANDWIDTH_REQUEST 4009
- #define OPUS_SET_COMPLEXITY_REQUEST 4010
- #define OPUS_GET_COMPLEXITY_REQUEST 4011
- #define OPUS_SET_INBAND_FEC_REQUEST 4012
- #define OPUS_GET_INBAND_FEC_REQUEST 4013
- #define OPUS_SET_PACKET_LOSS_PERC_REQUEST 4014
- #define OPUS_GET_PACKET_LOSS_PERC_REQUEST 4015
- #define OPUS_SET_DTX_REQUEST 4016
- #define OPUS_GET_DTX_REQUEST 4017
- #define OPUS_SET_VBR_CONSTRAINT_REQUEST 4020
- #define OPUS_GET_VBR_CONSTRAINT_REQUEST 4021
- #define OPUS_SET_FORCE_CHANNELS_REQUEST 4022
- #define OPUS_GET_FORCE_CHANNELS_REQUEST 4023
- #define OPUS_SET_SIGNAL_REQUEST 4024
- #define OPUS_GET_SIGNAL_REQUEST 4025
- #define OPUS_GET_LOOKAHEAD_REQUEST 4027
- #define OPUS_GET_SAMPLE_RATE_REQUEST 4029
- #define OPUS_GET_FINAL_RANGE_REQUEST 4031
- #define OPUS_GET_PITCH_REQUEST 4033
- #define OPUS_SET_GAIN_REQUEST 4034
- #define OPUS_GET_GAIN_REQUEST 4045 /∗ Should have been 4035 ∗/
- #define OPUS_SET_LSB_DEPTH_REQUEST 4036
- #define OPUS_GET_LSB_DEPTH_REQUEST 4037
- #define OPUS_GET_LAST_PACKET_DURATION_REQUEST 4039
- #define __opus_check_int(x) (((void)((x) == (opus_int32)0)), (opus_int32)(x))
- #define __opus_check_int_ptr(ptr) ((ptr) + ((ptr) - (opus_int32∗)(ptr)))
- #define __opus_check_uint_ptr(ptr) ((ptr) + ((ptr) - (opus_uint32∗)(ptr)))
- #define OPUS_AUTO

Auto/default setting.

- #define OPUS_BITRATE_MAX

  Maximum bitrate.

- #define OPUS_APPLICATION_VOIP

  Best for most VoIP/videoconference applications where listening quality and intelligibility matter most.

- #define OPUS_APPLICATION_AUDIO

  Best for broadcast/high-fidelity application where the decoded audio should be as close as possible to the input.

- #define OPUS_APPLICATION_RESTRICTED_LOWDELAY

  Only use when lowest-achievable latency is what matters most.

- #define OPUS_SIGNAL_VOICE 3001

  Signal being encoded is voice.

- #define OPUS_SIGNAL_MUSIC 3002

  Signal being encoded is music.

- #define OPUS_BANDWIDTH_NARROWBAND

  4 kHz bandpass

- #define OPUS_BANDWIDTH_MEDIUMBAND

  6 kHz bandpass

- #define OPUS_BANDWIDTH_WIDEBAND

  8 kHz bandpass

- #define OPUS_BANDWIDTH_SUPERWIDEBAND

  12 kHz bandpass

- #define OPUS_BANDWIDTH_FULLBAND

  20 kHz bandpass

- #define OPUS_SET_COMPLEXITY(x)

  Configures the encoder's computational complexity.

- #define OPUS_GET_COMPLEXITY(x)

  Gets the encoder's complexity configuration.

- #define OPUS_SET_BITRATE(x)

  Configures the bitrate in the encoder.

- #define OPUS_GET_BITRATE(x)

  Gets the encoder's bitrate configuration.

- #define OPUS_SET_VBR(x)

  Enables or disables variable bitrate (VBR) in the encoder.

- #define OPUS_GET_VBR(x)

  Determine if variable bitrate (VBR) is enabled in the encoder.

- #define OPUS_SET_VBR_CONSTRAINT(x)

  Enables or disables constrained VBR in the encoder.

- #define OPUS_GET_VBR_CONSTRAINT(x)

  Determine if constrained VBR is enabled in the encoder.

- #define OPUS_SET_FORCE_CHANNELS(x)

  Configures mono/stereo forcing in the encoder.

- #define OPUS_GET_FORCE_CHANNELS(x)

  Gets the encoder's forced channel configuration.

- #define OPUS_SET_MAX_BANDWIDTH(x)

  Configures the maximum bandpass that the encoder will select automatically.

- #define OPUS_GET_MAX_BANDWIDTH(x)

  Gets the encoder's configured maximum allowed bandpass.

- #define OPUS_SET_BANDWIDTH(x)

  Sets the encoder's bandpass to a specific value.
- #define OPUS_SET_SIGNAL(x)

  Configures the type of signal being encoded.
- #define OPUS_GET_SIGNAL(x)

  Gets the encoder's configured signal type.
- #define OPUS_SET_APPLICATION(x)

  Configures the encoder's intended application.
- #define OPUS_GET_APPLICATION(x)

  Gets the encoder's configured application.
- #define OPUS_GET_SAMPLE_RATE(x)

  Gets the sampling rate the encoder or decoder was initialized with.
- #define OPUS_GET_LOOKAHEAD(x)

  Gets the total samples of delay added by the entire codec.
- #define OPUS_SET_INBAND_FEC(x)

  Configures the encoder's use of inband forward error correction (FEC).
- #define OPUS_GET_INBAND_FEC(x)

  Gets encoder's configured use of inband forward error correction.
- #define OPUS_SET_PACKET_LOSS_PERC(x)

  Configures the encoder's expected packet loss percentage.
- #define OPUS_GET_PACKET_LOSS_PERC(x)

  Gets the encoder's configured packet loss percentage.
- #define OPUS_SET_DTX(x)

  Configures the encoder's use of discontinuous transmission (DTX).
- #define OPUS_GET_DTX(x)

  Gets encoder's configured use of discontinuous transmission.
- #define OPUS_SET_LSB_DEPTH(x)

  Configures the depth of signal being encoded.
- #define OPUS_GET_LSB_DEPTH(x)

  Gets the encoder's configured signal depth.
- #define OPUS_GET_LAST_PACKET_DURATION(x)

  Gets the duration (in samples) of the last packet successfully decoded or concealed.
- #define OPUS_RESET_STATE

  Resets the codec state to be equivalent to a freshly initialized state.
- #define OPUS_GET_FINAL_RANGE(x)

  Gets the final state of the codec's entropy coder.
- #define OPUS_GET_PITCH(x)

  Gets the pitch of the last decoded frame, if available.
- #define OPUS_GET_BANDWIDTH(x)

  Gets the encoder's configured bandpass or the decoder's last bandpass.
- #define OPUS_SET_GAIN(x)

  Configures decoder gain adjustment.
- #define OPUS_GET_GAIN(x)

  Gets the decoder's configured gain adjustment.

Functions

- const char * opus_strerror (int error)

  Converts an opus error code into a human readable string.

- const char * opus_get_version_string (void)

  Gets the libopus version string.

5.3.1 Detailed Description

Opus reference implementation constants.

5.3.2 Macro Definition Documentation

5.3.2.1 #define __opus_check_int(*x*) (((void)((x) == (opus_int32)0)), (opus_int32)(x))

5.3.2.2 #define __opus_check_int_ptr(*ptr*) ((ptr) + ((ptr) - (opus_int32∗)(ptr)))

5.3.2.3 #define __opus_check_uint_ptr(*ptr*) ((ptr) + ((ptr) - (opus_uint32∗)(ptr)))

5.3.2.4 #define OPUS_ARG_NONNULL(*_x*)

5.3.2.5 #define OPUS_GET_APPLICATION_REQUEST 4001

5.3.2.6 #define OPUS_GET_BANDWIDTH_REQUEST 4009

5.3.2.7 #define OPUS_GET_BITRATE_REQUEST 4003

5.3.2.8 #define OPUS_GET_COMPLEXITY_REQUEST 4011

5.3.2.9 #define OPUS_GET_DTX_REQUEST 4017

5.3.2.10 #define OPUS_GET_FINAL_RANGE_REQUEST 4031

5.3.2.11 #define OPUS_GET_FORCE_CHANNELS_REQUEST 4023

5.3.2.12 #define OPUS_GET_GAIN_REQUEST 4045 /∗ Should have been 4035 ∗/

5.3.2.13 #define OPUS_GET_INBAND_FEC_REQUEST 4013

5.3.2.14 #define OPUS_GET_LAST_PACKET_DURATION_REQUEST 4039

5.3.2.15 #define OPUS_GET_LOOKAHEAD_REQUEST 4027

5.3.2.16 #define OPUS_GET_LSB_DEPTH_REQUEST 4037

5.3.2.17 #define OPUS_GET_MAX_BANDWIDTH_REQUEST 4005

5.3.2.18 #define OPUS_GET_PACKET_LOSS_PERC_REQUEST 4015

5.3.2.19 #define OPUS_GET_PITCH_REQUEST 4033

5.3.2.20 #define OPUS_GET_SAMPLE_RATE_REQUEST 4029

5.3.2.21 #define OPUS_GET_SIGNAL_REQUEST 4025

5.3.2.22 #define OPUS_GET_VBR_CONSTRAINT_REQUEST 4021

5.3.2.23 #define OPUS_GET_VBR_REQUEST 4007

5.3.2.24 #define OPUS_GNUC_PREREQ(*_maj, _min*) 0

5.3.2.25 #define OPUS_RESTRICT

5.3.2.26 #define OPUS_SET_APPLICATION_REQUEST 4000

5.3.2.28 #define OPUS_SET_BITRATE_REQUEST 4002

```
#include "opus.h"
```

Macros

- #define __opus_check_encstate_ptr(ptr) ((ptr) + ((ptr) - (OpusEncoder**)(ptr)))
- #define __opus_check_decstate_ptr(ptr) ((ptr) + ((ptr) - (OpusDecoder**)(ptr)))
- #define OPUS_MULTISTREAM_GET_ENCODER_STATE_REQUEST 5120
- #define OPUS_MULTISTREAM_GET_DECODER_STATE_REQUEST 5122
- #define OPUS_MULTISTREAM_GET_ENCODER_STATE(x, y)

  Gets the encoder state for an individual stream of a multistream encoder.
- #define OPUS_MULTISTREAM_GET_DECODER_STATE(x, y)

  Gets the decoder state for an individual stream of a multistream decoder.

Typedefs

- typedef struct OpusMSEncoder OpusMSEncoder

  Opus multistream encoder state.
- typedef struct OpusMSDecoder OpusMSDecoder

  Opus multistream decoder state.

Functions

Multistream encoder functions

- opus_int32 opus_multistream_encoder_get_size (int streams, int coupled_streams)

  Gets the size of an OpusMSEncoder structure.
- opus_int32 opus_multistream_surround_encoder_get_size (int channels, int mapping_family)
- OpusMSEncoder * opus_multistream_encoder_create (opus_int32 Fs, int channels, int streams, int coupled_-streams, const unsigned char *mapping, int application, int *error)

  Allocates and initializes a multistream encoder state.
- OpusMSEncoder * opus_multistream_surround_encoder_create (opus_int32 Fs, int channels, int mapping_-family, int *streams, int *coupled_streams, unsigned char *mapping, int application, int *error)
- int opus_multistream_encoder_init (OpusMSEncoder *st, opus_int32 Fs, int channels, int streams, int coupled-_streams, const unsigned char *mapping, int application)

  Initialize a previously allocated multistream encoder state.
- int opus_multistream_surround_encoder_init (OpusMSEncoder *st, opus_int32 Fs, int channels, int mapping-_family, int *streams, int *coupled_streams, unsigned char *mapping, int application)
- int opus_multistream_encode (OpusMSEncoder *st, const opus_int16 *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes a multistream Opus frame.
- int opus_multistream_encode_float (OpusMSEncoder *st, const float *pcm, int frame_size, unsigned char *data, opus_int32 max_data_bytes)

  Encodes a multistream Opus frame from floating point input.
- void opus_multistream_encoder_destroy (OpusMSEncoder *st)

  Frees an `OpusMSEncoder` allocated by opus_multistream_encoder_create().
- int opus_multistream_encoder_ctl (OpusMSEncoder *st, int request,...)

  Perform a CTL function on a multistream Opus encoder.

Multistream decoder functions

- opus_int32 opus_multistream_decoder_get_size (int streams, int coupled_streams)

  Gets the size of an `OpusMSDecoder` structure.
- OpusMSDecoder * opus_multistream_decoder_create (opus_int32 Fs, int channels, int streams, int coupled_streams, const unsigned char *mapping, int *error)

  Allocates and initializes a multistream decoder state.
- int opus_multistream_decoder_init (OpusMSDecoder *st, opus_int32 Fs, int channels, int streams, int coupled_streams, const unsigned char *mapping)

  Intialize a previously allocated decoder state object.
- int opus_multistream_decode (OpusMSDecoder *st, const unsigned char *data, opus_int32 len, opus_int16 *pcm, int frame_size, int decode_fec)

  Decode a multistream Opus packet.
- int opus_multistream_decode_float (OpusMSDecoder *st, const unsigned char *data, opus_int32 len, float *pcm, int frame_size, int decode_fec)

  Decode a multistream Opus packet with floating point output.
- int opus_multistream_decoder_ctl (OpusMSDecoder *st, int request,...)

  Perform a CTL function on a multistream Opus decoder.
- void opus_multistream_decoder_destroy (OpusMSDecoder *st)

  Frees an `OpusMSDecoder` allocated by opus_multistream_decoder_create().

5.4.1 Detailed Description

Opus reference implementation multistream API.

5.4.2 Macro Definition Documentation

5.4.2.1 #define __opus_check_decstate_ptr(*ptr*) ((ptr) + ((ptr) - (OpusDecoder**)(ptr)))

5.4.2.2 #define __opus_check_encstate_ptr(*ptr*) ((ptr) + ((ptr) - (OpusEncoder**)(ptr)))

5.4.2.3 #define OPUS_MULTISTREAM_GET_DECODER_STATE_REQUEST 5122

5.4.2.4 #define OPUS_MULTISTREAM_GET_ENCODER_STATE_REQUEST 5120

5.5 opus_types.h File Reference

Opus reference implementation types.

Macros

- #define opus_int int /* used for counters etc; at least 16 bits */
- #define opus_int64 long long
- #define opus_int8 signed char
- #define opus_uint unsigned int /* used for counters etc; at least 16 bits */
- #define opus_uint64 unsigned long long
- #define opus_uint8 unsigned char

Typedefs

- typedef short opus_int16
- typedef unsigned short opus_uint16
- typedef int opus_int32
- typedef unsigned int opus_uint32

5.5.1 Detailed Description

Opus reference implementation types.

5.5.2 Macro Definition Documentation

5.5.2.1 #define opus_int int /* used for counters etc; at least 16 bits */

5.5.2.2 #define opus_int64 long long

5.5.2.3 #define opus_int8 signed char

5.5.2.4 #define opus_uint unsigned int /* used for counters etc; at least 16 bits */

5.5.2.5 #define opus_uint64 unsigned long long

5.5.2.6 #define opus_uint8 unsigned char

5.5.3 Typedef Documentation

5.5.3.1 typedef short opus_int16

5.5.3.2 typedef int opus_int32

5.5.3.3 typedef unsigned short opus_uint16

5.5.3.4 typedef unsigned int opus_uint32