STEMS EFFICIENT 1.0.0

by zplane.development

September 21, 2022

Contents

1 STEMS EFFICIENT Documentation **2**
- 1.1 Introduction . . . 2
- 1.2 API Documentation . . . 3
  - 1.2.1 Latency Modes . . . 3
  - 1.2.2 Memory Allocation . . . 3
  - 1.2.3 Naming Conventions . . . 3
  - 1.2.4 Instance Handling Functions . . . 3
  - 1.2.5 Process Functions . . . 4
  - 1.2.6 Parameter Retrieving and Setting Functions . . . 4
  - 1.2.7 C++ Usage example demonstrating the functionalities of a single instance . . . 5
  - 1.2.8 C++ Usage example demonstrating the combined use of low and high latency instances . . . 7
- 1.3 Support . . . 9

2 Namespace Index **9**
- 2.1 Namespace List . . . 9

3 Class Index **9**
- 3.1 Class List . . . 9

4 File Index **10**
- 4.1 File List . . . 10

5 Namespace Documentation **10**
- 5.1 zplane Namespace Reference . . . 10

6 Class Documentation **10**
- 6.1 zplane::Stems Class Reference . . . 10
  - 6.1.1 Detailed Description . . . 11
  - 6.1.2 Member Enumeration Documentation . . . 11
  - 6.1.3 Constructor & Destructor Documentation . . . 13
  - 6.1.4 Member Function Documentation . . . 13

7 File Documentation **16**
- 7.1 /work/project/docs/docugen.txt File Reference . . . 16
- 7.2 Stems/Stems.h File Reference . . . 16
  - 7.2.1 Detailed Description . . . 17

Index **18**

1 STEMS EFFICIENT Documentation

1.1 Introduction

STEMS EFFICIENT is a source separation and stem generation algorithm that is capable of separating any stereo mixture into vocals, bass, drums, and ”other” (4 stereo stems in total).

The underlying algorithm in STEMS EFFICIENT is based on a single deep neural network designed to be most efficient while retaining a maximum of quality. Internally, STEMS EFFICIENT works by analyzing small chunks of audio one at a time and estimating all instruments for that audio chunk. The STEMS EFFICIENT interface allows the user to set the output block size according to their needs. Additionally, post-processing can be applied to further enhance separation. An overview of the algorithm is shown below.

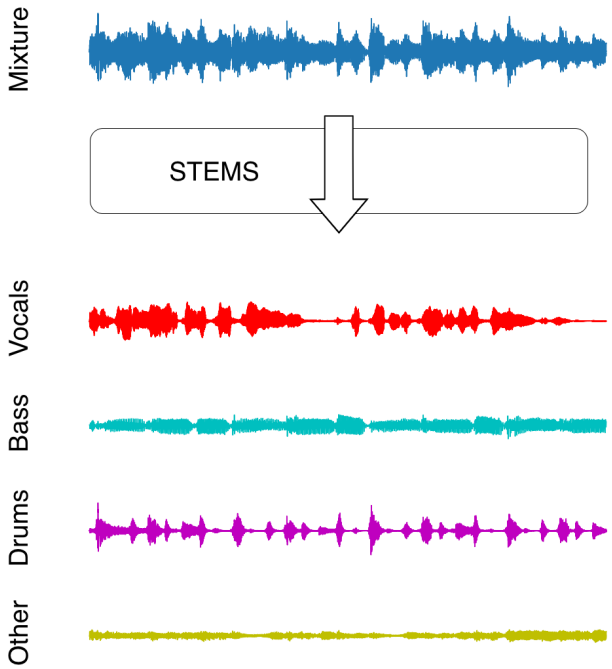


Figure 1: Overview of the source separation algorithm

This document contains all information you need to get started and use STEMS EFFICIENT to the best of its abilities. At its core are in-depth descriptions of the API through usage examples provided in the StemsClMain.cpp and StemsClCrossfade↩ Example.cpp files. These examples demonstrate all the functionalities of STEMS E↩ FFICIENT and can be used to get started as quickly and easily as possible. What follows is a detailed documentation of the STEMS EFFICIENT interface along with all of the methods and structs contained within.

1.2 API Documentation

1.2.1 Latency Modes

The `STEMS EFFICIENT` API has two latency modes, ”high” and ”low”, as defined in **enum Stems::LatencyMode**. The normal and recommended mode is ”high”. For even more efficiency and when playback needs to happen quickly, the user can choose the ”low” latency mode. In this case, `STEMS EFFICIENT` processes smaller chunks of audio at a time and therefore outputs to the stream quicker and more often. In low latency mode, the quality is slightly deteriorated. The latency mode is set during initialization of `STEMS EFFICIENT` and can't be changed afterwards. The following figure illustrates this behavior. Note that the output block size can be set to be the same in both cases.

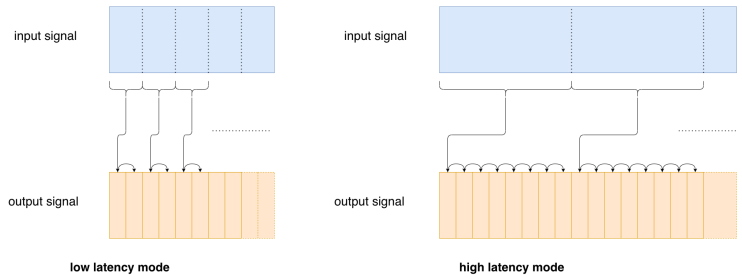


Figure 2: Illustration of the latency modes

1.2.2 Memory Allocation

The `STEMS EFFICIENT` SDK does not allocate buffers handled by the calling application; the input buffers must be allocated by the calling application. Input audio buffers are allocated as double arrays of size [numChannels][maxBlockSize] where maxBlockSize should be the value returned by **Stems::getMaxFramesNeeded()**. Note that the value returned by **Stems::getMaxFramesNeeded()** depends on the latency mode as well as the input sampling rate. The 4 output buffers (one for each instrument), are allocated as double arrays of size [numChannels][blockSize] where blockSize is the output block size chosen by the user.

1.2.3 Naming Conventions

When talking about **frames**, the number of audio samples per channel is meant. For example, 512 stereo frames correspond to 1024 float values (samples). If the sample size is 32bit float, one sample has a memory usage of 4 bytes. An audio **block** is a sequence of consecutive frames with a given length.

1.2.4 Instance Handling Functions

- **ErrorType Stems::initialize (int numChannels, float sampleRate, int max↩ Outputblocksize, LatencyMode latencyMode);**
  - Initalizes a `STEMS EFFICIENT` instance.
- **ErrorType Stems::reset() ;**

- Resets all internal variables and buffers to the default state. The return value indicates whether an error occurred or not.

1.2.5 Process Functions

- **ErrorType Stems::process(float const* const* const ppfInputBlock, std::size_t numInputFrames, float* const* const ppfOutputBlock)**
  - Performs the actual stems separation processing if the number of frames provided is as retrieved by **Stems::getFramesNeeded()**.
- **ErrorType Stems::finishProcessing(float const* const* const ppfInputBlock, std::size_t numInputFrames)**
  - Signals the end of the processing loop. Input contains the remaining samples of the input signal that should be processed. numInputFrames must be less than the length reported by **Stems::getFramesNeeded()**, otherwise process can still be called. This function must only be called once. After a call to this function, no further calls to process() are possible.
- **ErrorType Stems::flushBuffer (float* const* const ppfOutputBlock, std::size_t& numOutputFrames)**
  - Gets all the remaining internal frames when no more input data is available and writes them into the buffer ppfOutputBlock. Returns the number of written samples. The use of this function is optional.

1.2.6 Parameter Retrieving and Setting Functions

- **ErrorType Stems::setOutputBlockSize (std::size_t outputBlockSize)**
  - Sets the output block size.
- **std::size_t Stems::getFramesNeeded();**
  - Returns the required number of sample frames in order to obtain a full output block during the next call to Stems::process()
- **std::size_t Stems::getMaxFramesNeeded();**
  - Returns the maximum required number of frames needed.
- **void Stems::setEnablePostProcessingOnInstrument (Instrument instrument, bool activatePostProcessing);**
  - Applies post-processing on a selected instrument. The post-processing here reduces interferences from other instruments and improves separation for the selected instrument. Post-processing can be activated or deactivated during processing. Post-processing can also be activated for multiple instruments.
- **bool Stems::getEnablePostProcessingOnInstrument (Instrument instrument);**
  - Returns information on the current activation state of post-processing for a given instrument.

1.2.7 C++ Usage example demonstrating the functionalities of a single instance

The complete code referenced here can be found in the example source file StemsClMain.cpp. An overview of the processing steps and buffer management is illustrated below.

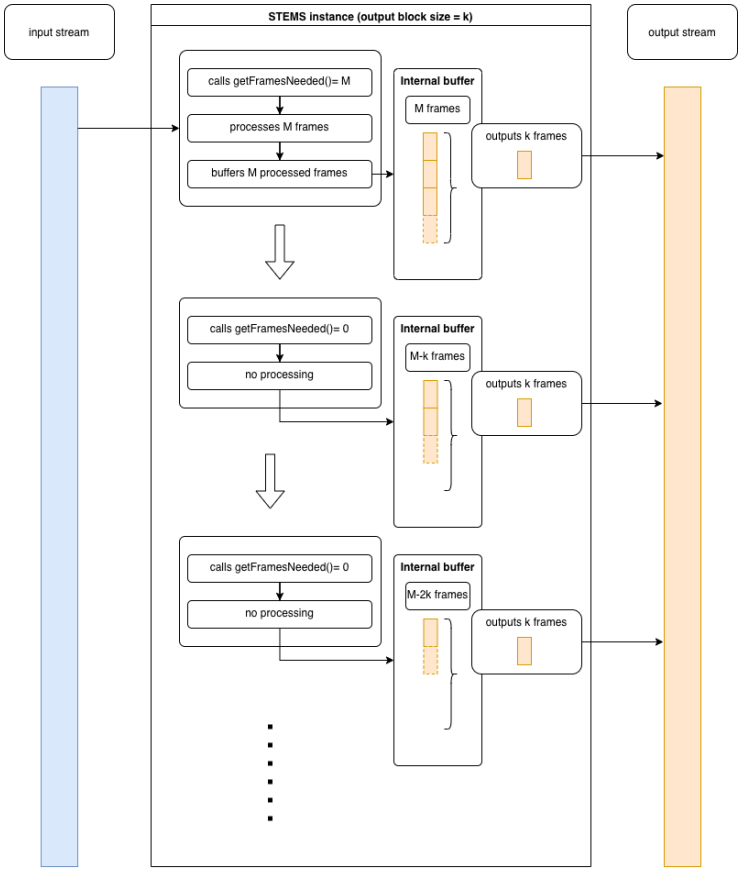


Figure 3: Flow chart illustration

In the first step, an instance of `STEMS EFFICIENT` must be created. **Note that this is different to how you would create an instance of previous zplane SDKs**. After instantiation, we use the **Stems::initialize()** method to prepare `STEMS EFFI↩ CIENT` to be used.

```
zplane::Stems stems;
error = stems.initialize(inputFile.GetNumOfChannels(), inputFile.GetSampleRate(), kBlockSize,
   zplane::Stems::LatencyMode::high);
```

In this case, we initialize the instance of `STEMS EFFICIENT` just with the channels and samplerate from the audio file to be processed. We also have to set the desired output block size here as well as the latency mode. After initializing, we can optionally set the post-processing to certain instruments. This can also be done later on.

```
stems.setEnablePostProcessingOnInstrument (
```

```
    zplane::Stems::vocals, true);
```

Now that we have initialized our instance of `STEMS EFFICIENT`, we can now allocate our input buffer according to the maximum of frames needed by the instance:

```
    ppfInput[i] = new float[stems.getMaxFramesNeeded()];
```

The output buffers are allocated with the desired output block size, and with the total number of channels of all our stems. In this case, we have 4 stereo stems, so the output buffer has 8 channels.

```
auto numOutputChannels = vocalsFile.GetNumOfChannels() + bassFile.GetNumOfChannels() + drumsFile.
  GetNumOfChannels() + otherFile.GetNumOfChannels();
```

And:

```
    ppfOutput[i] = new float[kBlockSize];
```

Now that we have allocated all our buffers, we can start processing. We start the process loop by asking our `STEMS EFFICIENT` instance how many frames are needed for processing.

```
    size_t inputSize = stems.getFramesNeeded();
```

We then read the audio file and fill the input buffer:

```
    numFramesRead = inputFile.Read (ppfInput, inputSize);
```

The process function can then be called with the required number of input frames.

```
    stems.process(ppfInput, numFramesRead, ppfOutput);
```

If there was no error, we can simply write out the output buffers for each instrument. Note that the output buffer has 8 channels for each of the 4 stereo stems.

```
    vocalsFile.Write (&ppfOutput[0], kBlockSize);
    bassFile.Write (&ppfOutput[2], kBlockSize);
    drumsFile.Write (&ppfOutput[4], kBlockSize);
    otherFile.Write (&ppfOutput[6], kBlockSize);
```

Once all available blocks have been fed to `STEMS EFFICIENT` and the last section of audio is smaller than number of frames required by the instance, the process loop is exited.

We can now call **Stems::finishProcessing()** to process the last section of audio.

```
stems.finishProcessing (ppfInput, numFramesRead);
```

Afterwards, to get the remaining samples in the internal buffer of `STEMS EFFI↩ CIENT`, we need to call the **Stems::flushBuffer()** method and write the samples to the output files.

```
    stems.flushBuffer(ppfOutput, numOfOutputFrames);
```

After successful processing, we can simply close the audio files.

1.2.8 C++ Usage example demonstrating the combined use of low and high latency instances

This example aims to showcase the combined use of ”low” and ”high” latency instances of `STEMS EFFICIENT`. In this first example, the low-latency instance is used to process the first few audio blocks (at a faster rate) while the high-latency instance is processing a later block in parallel. To bridge the gap smoothly between the outputs of both instances, a simple crossfade is applied. A schematic illustration of the processing flow is provided below:

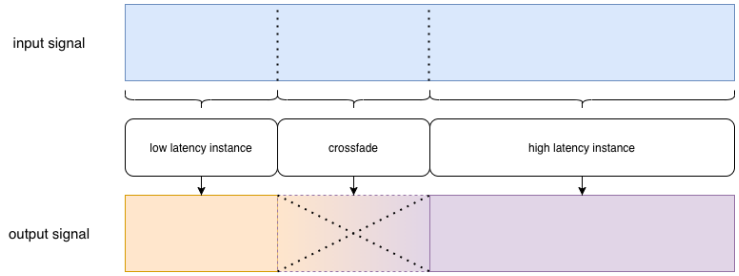


Figure 4: Illustration of the combined use of low and high latency instances

Much of this example is similar to StemsClMain.cpp but, in this particular example, only two instances of `STEMS EFFICIENT` are initialized:

```
zplane::Stems stemsHigh;
zplane::Stems stemsLow;


errorHigh = stemsHigh.initialize(inputFile.GetNumOfChannels(), inputFile.GetSampleRate(),
  kBlockSize, zplane::Stems::LatencyMode::high);
errorLow = stemsLow.initialize(inputFile.GetNumOfChannels(), inputFile.GetSampleRate(),
  kBlockSize, zplane::Stems::LatencyMode::low);
```

Accordingly, each instances requires a different number of frames for processing:

```
size_t maxFramesNeededHigh = stemsHigh.getMaxFramesNeeded();
size_t maxFramesNeededLow = stemsLow.getMaxFramesNeeded();
```

The input buffer thus has to be allocated with the maximum frame length it might need:

```
    ppfInput[i] = new float[std::max(maxFramesNeededHigh, maxFramesNeededLow)];
```

Both instances of `STEMS EFFICIENT` can use the same input buffer. But in this case, we have to allocate output buffers for each instance of `STEMS EFFICIENT` and the crossfade, or sum of the two outputs:

```
    ppfOutputLow[i] = new float[kBlockSize];
    ppfOutputHigh[i] = new float[kBlockSize];
    ppfOutputSum[i] = new float[kBlockSize];
```

Also, because both instances read from the same input file, read positions for each instance have to be saved and set individually before reading and processing the input file. The idea here is that we can decide beforehand at which frame the high-latency instance will start processing; appropriately, the low-latency instance will process from the beginning up until that point. So the initial read position of the high-latency instance serves as threshold for the low-latency instance. We also have to make sure that the initial read position of the high-latency instance is divisible by the output block size defined by the user so that the instance starts processing at the start of a block.

```
int readIndexLow (0);
int readIndexHigh (128000);
```

We also defined the length in frames of the crossfade section following the initial processing of the low-latency instance.

```
int crossFadeLength (3072);
```

In the next step, the process loop of the low-latency instance can be executed. This loop is very similar to the StemsClMain.cpp example.

We break the loop when the threshold is reached, and save the current read position of the low-latency instance:

```
if(currentOutputBlock * kBlockSize >= readIndexHigh)
{
    readIndexLow = inputFile.GetFilePos();
    assert(readIndexHigh == vocalsFile.GetFilePos());
    break;
}
```

Next, we can start the crossfade section. Both instances process the same audio section and the outputs from both are mixed together using a simple linear ramp. First, the audio data is read for the low-latency instance:

```
size_t inputSizeLow = stemsLow.getFramesNeeded();
// read audio data
inputFile.SetFilePos(readIndexLow);
framesRead = inputFile.Read (ppfInput, inputSizeLow);
readIndexLow = inputFile.GetFilePos();
```

Note that the read position is set before reading and saved afterwards. Then processing is applied:

```
stemsLow.process (ppfInput, framesRead, ppfOutputLow);
```

Same is done for the high-latency instance:

```
size_t inputSizeHigh = stemsHigh.getFramesNeeded();
// read audio data
inputFile.SetFilePos(readIndexHigh);
framesRead = inputFile.Read (ppfInput, inputSizeHigh);
readIndexHigh = inputFile.GetFilePos();
```

And again processing is applied:

```
stemsHigh.process (ppfInput, framesRead, ppfOutputHigh);
```

Note that, despite the two instances having different input sizes, the output blocks have the same lengths. Therefore, now that we have processed the same output block with both instances, we can combine them together using a simple linear crossfade:
Simplified the sum is

```
X = (1-f(x)) * A  + f(x) * B
```

Where `X` is the result of the crossfade, `A` is the low-latency output, `B` the high-latency output, and `f(x)` is a ramp with `x` in the range [0, crossFadeLength].

The result can now but written to the files.

```
vocalsFile.Write (&ppfOutputSum[0], kBlockSize);
bassFile.Write (&ppfOutputSum[2], kBlockSize);
drumsFile.Write (&ppfOutputSum[4], kBlockSize);
otherFile.Write (&ppfOutputSum[6], kBlockSize);
```

We break the loop when the crossfade length is reached.

```
if (currentCrossFadeOutputBlock * kBlockSize >= crossFadeLength)
    break;
```

Finally, we can continue processing the rest of the audio file only with the high-latency instance in the same way as in StemsClMain.cpp.

1.3 Support

Support for the source code is - within the limits of the agreement - available from:

zplane.development
Grunewaldstr. 83
d-10823 berlin
Germany

fon: +49.30.854 09 15.0
fax: +49.30.854 09 15.5

@: info@zplane.de

2 Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

zplane **10**

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

zplane::Stems **10**

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

Stems/Stems.h
Interface of the Stems class **16**

5 Namespace Documentation

5.1 zplane Namespace Reference

Classes

- class Stems

6 Class Documentation

6.1 zplane::Stems Class Reference

`#include <Stems/Stems.h>`

Public Types

- enum ErrorType {
  noError, memError, notInitializedError, alreadyInitializedError,
  invalidFunctionParamError, invalidFunctionCallError, unknownError, numError↩
  Types }
- enum VersionType {
  major, minor, patch, revision,
  numVersionTypes }
- enum LatencyMode { low, high }
- enum Instrument { vocals, bass, drums, other }

Public Member Functions

- Stems ()
- ~Stems ()
- ErrorType initialize (int numChannels, float sampleRate, int maxOutputblocksize, LatencyMode latencyMode)
- bool isInitialized ()
- ErrorType setOutputBlockSize (std::size_t outputBlockSize)
- std::size_t getFramesNeeded ()
- std::size_t getMaxFramesNeeded ()
- ErrorType process (float const *const *const ppfInputBlock, std::size_t num↩
  InputFrames, float *const *const ppfOutputBlock)

- ErrorType finishProcessing (float const *const *const ppfInputBlock, std::size_t numInputFrames)
- ErrorType flushBuffer (float *const *const ppfOutputBlock, std::size_t &numOutputFrames)
- ErrorType reset ()
- void setEnablePostProcessingOnInstrument (Instrument instrument, bool activatePostProcessing)
- bool getEnablePostProcessingOnInstrument (Instrument instrument)

Static Public Member Functions

- static const char * getVersion ()
- static const char * getBuildDate ()

6.1.1 Detailed Description

Definition at line 40 of file Stems.h.

6.1.2 Member Enumeration Documentation

ErrorType `enum zplane::Stems::ErrorType`

Enumerator

noError	no error occurred
memError	memory allocation failed
notInitializedError	instance has not been initialized yet
alreadyInitializedError	instance has already been initialized
invalidFunctionParamError	one or more function parameters are not valid
invalidFunctionCallError	function call is not allowed
unknownError	unknown error occurred
numErrorTypes	

Definition at line 43 of file Stems.h.

```
44         {
45             noError,
46             memError,
47             notInitializedError,
48             alreadyInitializedError,
49             invalidFunctionParamError,
50             invalidFunctionCallError,
51             unknownError,
52             numErrorTypes
53         };
```

Instrument `enum zplane::Stems::Instrument`

Enumerator

vocals	
bass	
drums	
other	

Definition at line 70 of file Stems.h.

```
71         {
72             vocals,
73             bass,
74             drums,
75             other
76         };
```

LatencyMode `enum zplane::Stems::LatencyMode`

Enumerator

low	
high	

Definition at line 64 of file Stems.h.

```
65         {
66             low,
67             high
68         };
```

VersionType `enum zplane::Stems::VersionType`

Enumerator

major	
minor	
patch	
revision	
numVersionTypes	

Definition at line 55 of file Stems.h.

```
56         {
57             major,
58             minor,
59             patch,
60             revision,
61             numVersionTypes
62         };
```

6.1.3 Constructor & Destructor Documentation

Stems() `zplane::Stems::Stems ( )`

~Stems() `zplane::Stems::~Stems ( )`

6.1.4 Member Function Documentation

finishProcessing() `ErrorType zplane::Stems::finishProcessing (`
`float const *const *const ppfInputBlock,`
`std::size_t numInputFrames )`

Signals the end of the processing loop. Input contains the remaining samples of the input signal that should be processed. numInputFrames must be less than the length reported by Stems::getFramesNeeded(), otherwise process can still be called. This function must only be called once. After a call to this function no further calls to Stems::process() are possible.

Parameters

ppfInputBlock	: input sample buffer [channels][samples]
numInputFrames	: the number of input frames

Returns

Stems::ErrorType : Returns an error flag

flushBuffer() `ErrorType zplane::Stems::flushBuffer (`
`float *const *const ppfOutputBlock,`
`std::size_t & numOutputFrames )`

Gets all the remaining internal frames when no more input data is available and writes them into the buffer ppfOutputBlock. Returns the number of written samples. The use of this function is optional.

Parameters

ppfOutputBlock	output sample buffer [channels][samples]
numOutputFrames	the number of output frames

Returns

Stems::ErrorType : Returns an error flag

getBuildDate() `static const char* zplane::Stems::getBuildDate ( ) [static]`

Returns the build date string.

getEnablePostProcessingOnInstrument() `bool zplane::Stems::getEnablePost↩ ProcessingOnInstrument (`
`Instrument instrument )`

Gives back information on the current activation state of post-processing for a given instrument.

Parameters

instrument	Stems::Instrument, instrument to apply post-processing to

Returns

activatePostProcessing: bool, whether or not post-processing is activated for selected track

getFramesNeeded() `std::size_t zplane::Stems::getFramesNeeded ( )`

Returns the required number of sample frames in order to obtain a full output block during the next call to Stems::process()

Returns

size_t : required number of sample frames

getMaxFramesNeeded() `std::size_t zplane::Stems::getMaxFramesNeeded ( )`

Returns the maximum required number of frames needed. This value is dependent on the output block size.

Returns

size_t : required number of sample frames

getVersion() `static const char* zplane::Stems::getVersion ( ) [static]`

Returns major version, minor version, patch and build number of this Stems version.

initialize() `ErrorType zplane::Stems::initialize (`
`int numChannels,`
`float sampleRate,`
`int maxOutputblocksize,`
`LatencyMode latencyMode )`

Initialize an instance of Stems. Must be called before using any of Stems functionality.

Parameters

numChannels	Number of channels in the input signal.
sampleRate	Sample rate of the input signal in Hertz.
maxOutputblocksize	Blocksize of the output buffer in samples.
latencyMode	Selects the latency mode of the separation model. Can be either "high" or "low". This parameter will affect the output of getFramesNeeded.

Returns

Stems::ErrorType : Returns an error flag

isInitialized() `bool zplane::Stems::isInitialized ( )`

process() `ErrorType zplane::Stems::process (`
`float const *const *const ppfInputBlock,`
`std::size_t numInputFrames,`
`float *const *const ppfOutputBlock )`

Does the actual stems separation processing if the number of frames provided is as retrieved by Stems::getFramesNeeded(). Input needs to be stereo. Output is multichannel stereo with the stems stacked in channels. With vocals, bass, drums and other, output has 8 channels.

Parameters

ppfInputBlock	: input sample buffer [channels][samples]
numInputFrames	: the number of input frames
ppfOutputBlock	: output sample buffer [channels][samples]

Returns

Stems::ErrorType : Returns an error flag

reset() `ErrorType zplane::Stems::reset ( )`

Clears the internal buffers. Call this method to avoid the remaining samples in the process buffer being audible when you e.g. stop playback and start it again at different time position. Other parameters are not reset.

Returns

Stems::ErrorType : Returns an error flag

setEnablePostProcessingOnInstrument() `void zplane::Stems::setEnablePost↩ ProcessingOnInstrument (`
`Instrument instrument,`
`bool activatePostProcessing )`

Applies post-processing on selected instrument. The post-processing here reduces interferences from other instruments and improves separation for the selected instrument. Post-processing can be activated or turned off during processing. Post-processing can also be activated for different instruments.

Parameters

instrument	Stems::Instrument, instrument to apply post-processing to
activatePostProcessing	bool, turns the post-processing on or off for the selected instrument.

setOutputBlockSize() `ErrorType zplane::Stems::setOutputBlockSize (`
`std::size_t outputBlockSize )`

Sets the output block size.

Parameters

outputBlockSize	: the new output blocksize

Returns

Stems::ErrorType : Returns an error flag

The documentation for this class was generated from the following file:

- Stems/Stems.h

7 File Documentation

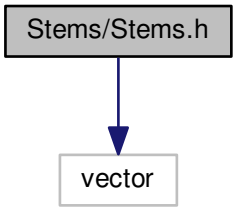
7.1 /work/project/docs/docugen.txt File Reference

7.2 Stems/Stems.h File Reference

interface of the Stems class.

```
#include <vector>
```

Include dependency graph for Stems.h:



Classes

- class zplane::Stems

Namespaces

- zplane

7.2.1 Detailed Description

interface of the Stems class.

:

Index

/work/project/docs/docugen.txt, 16
~Stems
    zplane::Stems, 13

ErrorType
    zplane::Stems, 11

finishProcessing
    zplane::Stems, 13
flushBuffer
    zplane::Stems, 13

getBuildDate
    zplane::Stems, 14
getEnablePostProcessingOnInstrument
    zplane::Stems, 14
getFramesNeeded
    zplane::Stems, 14
getMaxFramesNeeded
    zplane::Stems, 14
getVersion
    zplane::Stems, 14

initialize
    zplane::Stems, 14
Instrument
    zplane::Stems, 11
isInitialized
    zplane::Stems, 15

LatencyMode
    zplane::Stems, 12

process
    zplane::Stems, 15

reset
    zplane::Stems, 15

setEnablePostProcessingOnInstrument
    zplane::Stems, 16
setOutputBlockSize
    zplane::Stems, 16
Stems
    zplane::Stems, 13
Stems/Stems.h, 16

VersionType
    zplane::Stems, 12

zplane, 10
zplane::Stems, 10
    ~Stems, 13
    ErrorType, 11
    finishProcessing, 13
    flushBuffer, 13
    getBuildDate, 14
    getEnablePostProcessingOnInstrument, 14
    getFramesNeeded, 14
    getMaxFramesNeeded, 14
    getVersion, 14
    initialize, 14
    Instrument, 11
    isInitialized, 15
    LatencyMode, 12
    process, 15
    reset, 15
    setEnablePostProcessingOnInstrument, 16
    setOutputBlockSize, 16
    Stems, 13
    VersionType, 12