Voxygen Expressive Speech

C/C++ SDK 8.1

Developer Guide for Windows, Mac OS and Linux platforms

Build version: 8.1_1

Reference	:	VOX167_SDK_C_8.1_1.0_EN			
Doc. version	:	1.0	**Status**	:	Release
Date	:	22/05/2017	**Diffusion**	:	restricted

Document review

Version	Date	Author	Verified by	Modification
1.0	22/05/2017	ER		Creation for ES C/C++ SDK 8.1_1

Summary

1 Overview

1.1 About this document

This document describes how to use the Voxygen Expressive Speech Core in your applications, using the low level 'C/C++' API and explains how to control the TTS engine.

WARNING: For debugging your application please read the section 10.2

1.2 Operating system requirements

This C/C++ SDK core, build , is intended for the following operating systems:

- Windows 7 minimum / i386 and x86_64 architectures
- Linux RHEL 6+ (or equivalent CentOS version) / i386 architecture
- Linux RHEL 7+ / x86_64 architecture
- Linux DEBIAN 6+ (or equivalent Ubuntu version) / i386 and x86_64 architectures
- Linux / ARM architecture (example : Raspberry Pi2, Odroid...)
- MacOS 10.7 minimum / i386 and x86_64

1.3 Naming convention and terminology

The Voxygen Expressive Speech C/C++ SDK package you received is intended for your particular platform: Windows, Mac OS, Linux or ARM, as mentioned in the archive name.

In this document, the following terms are used as shortcuts:

- `<baratinoo>` represents the main directory of the installed Voxygen Expressive Speech C/C++ SDK package. It is depending of the platform and build version. For example:
  - Voxygen_ES_SDK_C_linux_8.1_1 for Linux,
  - Voxygen_ES_SDK_C_windows_8.1_1 for Windows,
  - or Voxygen_ES_SDK_C_linux_armv8-a53_8.1_1 for Linux ARM v8.
- `<platform>` represents the platform. Replace this tag according to your platform with one of the following values:
  - `win32` for Windows 32 bits platforms
  - `win64` for windows 64 bits platforms
  - `linx86` for Linux 32 bits platforms / i386 architecture
  - `linx64` for Linux 64 bits platforms / x86_64 architecture
  - `linux_arm` for Linux platforms / ARM architecture
  - `darwin` for MacOS platforms multi architecture

Some naming terms specific to a platform are prefixed in the document with the following tags :

```
[For Linux]
[For MacOS]
[For Windows]
```

Abbreviation	Description
SSML	*Speech Synthesis Markup Language*
TTS	*Text To Speech*
Baratinoo	*Abbreviation for the Voxygen Expressive Speech core*
activation code	*Set of characters given to client when purchase is made. This activation code is used to retrieve an activation file.*
activation file	*Secure file which contains one or more activation keys. This file has a .lic extension.*
activation key	*Line in activation file which unlocks a feature (voices are associated with a feature)*

1.4 Components version

The following table lists the version of the main components in the distribution package `8.1_1`:

Component	Version
Voxygen Expressive Speech Core (Baratinoo) and utilities	*8.1*
Voxygen C/C++ applications	*1.6*

1.5 Build change history

Version	Date	Modifications
8.1_1	22/05/2017	Baratinoo 8.1. Default frequency is now 24kHz in examples.
8.0 1	10/01/2017	Baratinoo 8.0c
7.6_3	22/06/2016	Baratinoo 7.6c, No UID when a UID is not readable
7.6_2	02/06/2016	Baratinoo 7.6b, Windows, Linux, Mac OS, Linux armv7-a7 packages
7.6_1	27/05/2016	Baratinoo 7.6a, Linux armv7-a5 package
7.5_ 5	18/02/2016	Baratinoo 7.5b, utilities in 64bits mode
7.5_ 3	22/01/2016	Baratinoo 7.5a, Linux packages
7.5_ 2	22/01/2016	Baratinoo 7.5a, Windows and Mac OS packages
7.5_1	08/12/2015	Dynamic libraries **C++ API changes :** constructor and destructor replaced by newInstance() and deleteInstance() for BaratinooEngine, InputTextBuffer, OutputSignalBuffer objects.
7.4_2	24/06/2015	Baratinoo7.4a, Error reading UID on Windows
7.4_1	20/03/2015	Baratinoo 7.4
7.3_2	06/01/2015	Baratinoo 7.3a
7.3_1	01/12/2014	Baratinoo 7.3
7.1_6	11/2013	Baratinoo 7.1f

1.6 Reference documents

All documents, except this one, are located in the sub-directory ‘`doc`’ of the distribution.

Reference	Document name	Component version	Document version
VOX167	Developer Guide for Windows, Mac OS and Linux platforms	8.1_1	1.0
VOX31	SSML reference manual	8.1	1.0
VOX32	BARATINOO tags reference manual	8.1	1.0
VOX170	Exceptions lexicon manual	8.1	1.0
VOX341	BARATINOO normaliser manual	8.1	1.0
VOX349	Phonemes and visemes reference manual	8.1	1.0

2 Package overview

Here are the components of the Voxygen Expressive Speech C/C++ SDK:

- The `c_api/` directory contains the C++ API:
  - baratinoo.h which describes the API
  - baratinooio.h which describes an implementation of the input and output objects needed by the engine.
  - baratinooio.cpp which is an implementation of the input and output objects needed by the engine.

- The `bin/<platform>` directory contains binaries which are necessary for the link edition and runtime usage.

- The `bin/<platform>_debug` directory contains the dynamic library to use with a debugger.

- The `c_api/barademo_CPP/` directory contains an example application using the C++ API.

- The `c_api/barademo_C/` directory contains an example application using the C API.

- The `config/` directory contains data needed by the TTS engine: voice data, language data, and the activation file.

- The `doc/` directory contains documentation about how to customize your data to improve the synthesis rendering.

3 Installation, activation and configuration

3.1 Installation process

Note that for the Voxygen TTS engine to run you have to install at least one voice and a valid activation file.

Here are the main steps to install the Voxygen Expressive Speech C/C++ SDK package.

Install the C/C++ SDK package

Extract the data from the archive file (without changing the directories structure) in the directory of your choice, like `/opt` under Linux MacOs or `c:\` under Windows.

Install one or more voices

This part is described in the next chapter.

Get and install activation file for voices

In order to use a given voice, you must have the associated activation file with the extension .lic.
The procedure to get the .lic activation file is as follows:

1. Open a command window
2. Go into the installation directory `<baratinoo>/bin/<platform>` by using the command `cd`
3. `[For Linux]` Type: `./lic_id`
   `[For MacOS]` Type: `./lic_id`
   `[For Windows]` Type: `lic_id.exe`
4. Then press 'Enter'
5. Send this baratinoo`.uid` file by mail to Voxygen Support: support@voxygen.fr
6. Once the support team has sent you back the .lic activation file, copy it into the directory `<baratinoo>/config/`

Checking activation file

You can check the status of yours activation keys by executing the command:

```
<baratinoo>/bin/<platform>/lic_list[.exe] <baratinoo>/config/baratinoo.cfg
```

The activation file depends the Voxygen TTS engine version, the voice and the voice version. The activation file is associated to your device.

Notes:

- *For Linux or Windows platforms the 32 bits or 64 bits versions of the tools produce the same result.*
- *For 64 bits Linux platform please ensure you have installed the 32 bits compatibility libraries before running the tools compiled in 32 bits mode.*

3.2 Building and running the example applications

Two example applications are provided for Linux Mac OS and Microsoft Visual Studio:
- an example application using the C++ API, located in the `c_api/barademo_CPP/` directory
- an example application using the C API, located in the `c_api/barademo_C/` directory

A script is provided for building each of these example applications.

1. Open a console terminal
2. Check your build environment:
   `[For Linux]` you need gcc compiler and 'make' tool
   `[For Mac OS]` you need gcc compiler and 'make' tool
   `[For Windows]` the included command script is intended for Microsoft Visual Studio compiler (for CPP example, see remark about Visual Studio in chapter 7). Adapt the commands in the script if you use another compiler like MinGW.
3. Go in the `c_api/barademo_CPP/<platform>` or `c_api/barademo_C/<platform>` directory
4. Run the following command:
   `[For Linux] make`
   `[For MacOS] make`
   `[For Windows] compil_demo.bat`
   If successful, an executable file is created in the `bin/<platform>` directory.
5. Go to the `bin/<platform>` directory and run the program `barademo_CPP[.exe]` or `barademo_C[.exe]`:
   `-barademo_C[.exe]` synthesizes the content of the texte.txt file and, in case of success, it outputs a `barademo_C.wav` audio file. Baratinoo logs are written in the `barademo_C.log` files.
   - `barademo_CPP[.exe]` synthesizes the content of the texte.txt file if it exists or a default sentence, and in case of success, it outputs a `barademo_CPP.wav` audio file. Baratinoo logs are written in the `barademo_CPP.log` files.

Note: for 64 bits Linux platform please ensure you have installed the 32 bits compatibility libraries before running the example compiled in 32 bits mode.

4 Voxygen Expressive Speech voice

4.1 Presentation

A Voxygen Expressive Speech voice is delivered separately in a compressed archive file.
The speech synthesis voices are supplied at a 24kHz sampling frequency. The Voxygen Expressive Speech core contains a resampling functionality, so it can produce speech synthesis at any rate (in the range [6000Hz, 48 000Hz]).
The use of a voice requires the acquisition of an activation file. The procedure for getting such an activation file is described in chapter 3.1.

4.2 Data Installation

Just extract the voice archive in the `<baratinoo>/config` directory without changing the directory structure.

5 Adjusting the rendering of Voxygen synthesis

5.1 Mark-up language

A mark-up language allows you to dynamically influence the behaviour of the speech synthesis system, by changing voice, regulating flow control, changing volume...
Two mark-up formats are available:

- The baratinoo in-house format . It is described in the document [VOX32] located in the directory `<baratinoo>/doc/`
- The standardized SSML format . It is described in the document [VOX31] located in the directory `<baratinoo>/doc/`

The API allows you to tell the TTS engine how to deal with these mark-up formats:

- Not to accept mark-ups. In this case, the mark-ups are interpreted as plain text.
- or to accept only one of the formats: in-house or SSML
- or to accept the two formats. You can then use them alternatively in a text.

5.2 Exceptions file

An exception file allows to improve the speech synthesis results for certain words (such as acronyms).
An empty exception file is provided per language in each language sub-directory :
<baratinoo>`/config/data/<language_dir>/<language>`**`.pls+xml`**

You can also create your own exceptions files and you can use them with setting the appropriate mark-ups in your texts (see SSML element <lexicon> in VOX31 document).
The [VOX170] document presents the format of an exception file and the way to personalize it in order to improve the speech synthesis result.

5.3 Text pre-processing

A normaliser rules file can be used to pre-process the input text at the beginning of the synthesis pipeline of the text-to-speech engine. It may be used for any generic transformation you want to apply on the text to read.
An empty normaliser rules file **textprocess.rgx+xml** is provided in each language sub-directory (in each <baratinoo>`/config/data/<language>`). Add your own rules in it for any transformation you want to apply on the text to read.
You can also create your own normaliser rules files and you can use them with setting the appropriate mark-ups in your texts (see SSML element <lexicon> in VOX31 document).
The [VOX341] document presents the format of an RGX rules file and the way to personalize it in order to improve the speech synthesis result.

6 TTS engine functional overview

This API allows you to create one text-to-speech engine and control it finely.
You can interact with the engine in many ways:

- The engine can process a whole entry text without stop.
- The engine can stop each time a predefined event occurs: voice changes, end of a word or a sentence, on a silence or for each phoneme…
- The engine can stop each time it gives a signal buffer.
- The engine will stop after a predefined number of processing loops.

Each time the engine stops, you can do anything else, purge the engine or run it again to resume the processing.
Before running the engine, you can choose the synthesized signal coding and frequency, the encoding of the input and the authorized parsing modes.

7 C++ API use

NOTE that this C++ API can be only used with **Visual Studio** on Windows due to name mangling compatibility. With other compilers please use the C API.

When the library is initialized with a call to the **`baratinooInit()`** function, it is possible to create one BaratinooEngine object.
When created, the BaratinooEngine must be initialized with a call to the `init()` member function. This function takes one parameter which is the file name of a Baratinoo configuration file. Before initialization, the BaratinooEngine is in UNINITIALIZED state. After that, it becomes INITIALIZED.
One input object (a specific implementation of the InputText class) must be given to BaratinooEngine through a call to `setInput()` after which the BaratinooEngine object is in READY state. You can use the basic implementation `InputTextBuffer` provided in this API (see baratinooio.h and .cpp files) or you can implement your own input class inherited from the `InputText` class object.
One output object (a specific implementation of the OutputSignal class) can be given to BaratinooEngine through a call to `setOutput()`. This call is not mandatory for the BaratinooEngine to run, but it is the only way to get the signal. You can use the basic implementation `OutputSignalBuffer` provided in this API (see baratinooio.h and .cpp files) or you can implement your own input class inherited from the `OutputSignal` class object.
It is then possible to call the `processLoop()` member which is the function where BaratinooEngine actually works and transforms input text into output signal. Optional parameter 'count' of `processLoop()` is the number of internal steps after which `processLoop()` should stop. If omitted, the whole text is processed. It is not possible to know *a priori* how many steps are needed to process a given text. All steps do not last the same time. The value for 'count' depends on the application and how responsive it should be. An appropriate value is generally in the range 10 to 100.
When the whole text is processed, BaratinooEngine goes into INITIALIZED state again, where another call to `setInput()` is expected if another text is to be processed.
A call to `purge()` set the BaratinooEngine to INITIALIZED state without finishing current input text.
Destruction of BaratinooEngine can occur in all states. It implicitly calls `purge()` before deleting.
Following functions return an enumeration of type BARATINOO_STATE which reflects the current state of BaratinooEngine: `init()`, `setInput()`, `processLoop()`, `purge()`.
Delete all engines objects when you have finished.
When all Baratinoo engines are deleted, call **`BaratinooTerminate()`** to clear memory and close the library.

Events
In parallel with output signal, a BaratinooEngine produces some events, synchronized with signal. Events are listed in enumeration BARATINOO_EVENT_TYPE.
An event is represented by a class BaratinooEvent which contains the associated values of event (in a union) and the timestamp of the event in the output signal. Timestamp unit is the sample. Timestamp is reset to 0 for each input text.
Possible events and their associated values can be fetched with a call to `getEvent()` (only in EVENT state).
Default behavior of BaratinooEngine is to be silent (no event activated).
Events can individually activated with a call to `setWantedEvent()`, individually deactivated with

a call to `unsetWantedEvent()` or globally activated with a call to `setAllWantedEvents()`, or deactivated with a call to `unsetAllWantedEvents()`. Those four functions can be called from any state but UNINITIALIZED.
Here are the possible events:

BARATINOO_MARKER_EVENT	when a marker tag is reached in the input data.
BARATINOO_WAITMARKER_EVENT	when a waitmarker tag is reached in the input data.
BARATINOO_PARAGRAPHE_EVENT	when processing a paragraph
BARATINOO_SENTENCE_EVENT	when processing a sentence
BARATINOO_WORD_EVENT	when processing a word
BARATINOO_PUNCTUATION_EVENT	when processing a punctuation
BARATINOO_SEPARATOR_EVENT	when processing a word separator like a space
BARATINOO_SYLLABLE_EVENT	when processing a syllable
BARATINOO_PHONEME_EVENT	when processing a phoneme *(see note 1)*
BARATINOO_VISEME_EVENT	when processing a viseme
BARATINOO_SILENCE_EVENT	when processing a silence
BARATINOO_NEW_VOICE_EVENT	when there is a voice change
BARATINOO_RAW_EVENT	when a raw tag is reached in the input data

Note 1: The BARATINOO_PHONEME_EVENT is not available in the standard packaging.

State diagram

Here are the 7 possible states of a BaratinooEngine:

- UNINITIALIZED. This state is the state of a BaratinooEngine after creation. The only allowed function call in this state is `init()`.
- INITIALIZED. This state is reach after a call to `init()` and after `processLoop()` processed the whole input text. In this state a call to `setInput()` is expected.
- READY. This state is reached from INITIALIZED or INPUT_ERROR states after a call to `setInput()`. A call to `processLoop()` is expected at that time.
- RUNNING. This state is reached after a call to `processLoop()` when input text is partially processed. This happens if the optional value 'count' of `processLoop()` is not negative or if the callback `OutputSignal::writeSignal()` returns a non zero value. In this state another call to `processLoop()` is expected.
- EVENT. This state is reached after a call to `processLoop()` when an activated event occurs. Associated value of current event can be fetch with a call to `getEvent()`. Another call to `processLoop()` is expected.
- INPUT_ERROR. `processLoop()` returns this value when an error occurs in input, probably because of a parsing error (not valid XML document for example). Like in INITIALIZED state, another call to `setInput()` is necessary to change state to READY.
- ENGINE_ERROR. This state is reached when a fatal error occurs. Probably because of lack of memory. More detail is given in an ERROR trace. In that case, engine cannot run anymore and destruction is the only allowed action.

Exceptions

BaratinooEngine does not throw exceptions.

Multithread

The BARATINOO TTS system is multithread safe in a sense that many BaratinooEngine objects can exist in a same process, working in different threads (limitation coming from resources: memory or license).
But BaratinooEngine objects themselves are not multithread safe. If two member functions

are called on a same object from two different threads, result is undefined.

8 C++ API description

8.1 Library functions

baratinooInit()

```
BARATINOO_INIT_RETURN baratinooInit( BaratinooTrace *trace)
```

Description:

Initialize the API.

Arguments:

`trace` : pointer to a trace object derived from the BaratinooTrace class.

Return code

`BARATINOO_INIT_OK`
`BARATINOO_INIT_ERROR`

baratinooInitWithKey()

```
BARATINOO_INIT_RETURN baratinooInitWithKey( BaratinooTrace *trace,
                                            BaratinooKey *key);
```

Description:

Initialize the API with a shared secret key (specific use case).

Arguments:

`trace` : pointer to a trace object derived from the BaratinooTrace class.
`key` : pointer to a key object.

Return code

`BARATINOO_INIT_OK`
`BARATINOO_INIT_ERROR`

baratinooTerminate()

```
void baratinooTerminate()
```

Description:

Close the API.

Arguments:

None.

Return code:

None.

getBaratinooVersion()

```
const char * getBaratinooVersion()
```

Description:

Return a string that contains the Baratinoo engine version.

Arguments:

None.

Return code:

A pointer to a static string that contains the Baratinoo engine version.

8.2 Class `BaratinooTrace`

BaratinooTrace is a pure virtual class. You must provide your own implementation of this class. Your implementation must have a `write()` member function which is called by the BARATINOO library and its objects. Arguments are:

- level: 5 categories of trace (see TraceLevel enumeration)
- engineNumber: 0 when trace is issued from library
- source: part of engine which generated the trace
- format + args: formated print, same syntax as printf() function

You can also provide another implementation of the `write()` member function which has one more argument `privatedata` :

- level: 5 categories of trace (see TraceLevel enumeration)
- engineNumber: 0 when trace is issued from library
- source: part of engine which generated the trace
- privatedata : value you passed at the engine initialization
- format + args: formated print, same syntax as printf() function

This implementation is useful if you create more than one engine: you can know what engine is calling the write method. To do so, you need to set private data at the engine initialization.

8.3 Class BaratinooKey

BaratinooKey is a pure virtual class. You must provide your own implementation of this class. Your implementation must have a `get()` member function which is called by the BARATINOO library and its objects. Argument is:

- index: request the 'index+1'th key (>=0)

8.4 Class `InputText`

This is a virtual class that you must derived from to create an input object which will be passed to the engine in the `setInput()` method.
You must implement the virtual following method:
`virtual int readText(void *address, int length)`
When the engine calls this method, this method must fill the buffer `address` with some text data without exceeding `length` bytes length. This method returns the number of bytes copied in the buffer or 0 if there is no more text data.
In the constructor of your object, you can pass a parsing and an encoding mode and a voice index.
An implementation of this class is provided in this API, see the `InputTextBuffer` class below.

8.5 Class `OutputSignal`

This is a virtual class that you must derived from to create an output object which will be passed to the engine in the `setOutput()` method.
You must implement the virtual following method:
`int writeSignal(void *address, int length)`
When the engine calls this method, this method processes the signal contained in the buffer `address`. The length of the signal is `length` bytes. This method should copy the data instead of keeping a pointer to the buffer because Baratinoo engine takes ownership of the buffer, and it will be deleted when the function exits. If you want that the engine returns immediately from the `processLoop()` method, this method must return 1. If not, this method must return 0.
In the constructor of your object, you can pass a signal coding format and an output signal frequency.
An implementation of this class is provided in this API, see the `OutputSignalBuffer` class below.

8.6 Class `BaratinooEngine`

This is the class for the text-to-speech engine. You must create a new instance of this class to get an engine.

Public class methods overview

`BaratinooEngine::newInstance(const void *privatedata = NULL)`
Constructor
Return a new instance of BaratinooEngine object

`BaratinooEngine::deleteInstance(BaratinooEngine *instance)`
Destructor
Instance is a pointer to the instance of BaratinooEngine object to delete

Public BaratinooEngine methods overview

`BARATINOO_STATE init(const char *config)`
Initialize the engine. `config` argument is a configuration filename.

`int getEngineNumber()`
Each time an engine is created, it gets a rank. Return this value.

`BARATINOO_STATE setInput(InputText *input)`
Initialize the input object the engine will call to get text to process. The properties of the input object (encoding and parsing mode, voice index) are taken in account by the engine.
`BARATINOO_STATE setInput(InputBinary *input)`
Initialize the binary input object the engine will call to get binary item to process.

`void setOutput(OutputSignal *output)`
Initialize the output object the engine will call to send the synthesized signal. The properties of the output object (signal coding format, signal frequency) are taken in account by the engine.

`void setOutput(OutputBinary *output)`
Initialize the binary output object the engine will call to send the output items.

`BARATINOO_STATE processLoop()`

Run the engine. This method returns when all the input data are processed, or when there is an error or when a predefined event has occurred.

`BARATINOO_STATE processLoop(int count)`
Run the engine. This method returns when all the input data are processed, or when there is an error or when a predefined event has occurred or when the engine has done `count` loops.

`BARATINOO_STATE purge()`
Reset the engine.

`void setWantedEvent(BARATINOO_EVENT_TYPE type)`
Set the specified event on. The engine will stop if this event occurs.

`void unsetWantedEvent(BARATINOO_EVENT_TYPE type)`
Reset the specified event.

`void setAllWantedEvent()`
Set all events on.

`void unsetAllWantedEvent()`
Reset all events.

`BaratinooEvent &getEvent()`
Return the event that has occurred. You must call it only when the engine has stopped in the EVENT state.

`int getNumberOfVoices()`
Return the number of voices currently loaded.

`BaratinooVoiceInfo &getVoiceInfo(int voiceNumber)`
Return information about the specified voice.

```
BARATINOO_CANSPEAK canSpeak (int voiceNumber,
    const char *language_iso639, const char *region_iso3166, const
    char *variant)
```
Return information about the capabilities of a the specified voice.

8.7 Class `InputTextBuffer`

This class is an implementation of the virtual `InputText` class. Source files of this object are baratinooio.h and baratinooio.cpp.
This implementation is simple to use:

1- Use the class methods `newInstance()` to create an InputTextBuffer object (do not use the InputTextBuffer object constructor). You can set a specific encoding format for the text, a specific parsing mode and the voice to use. Default behaviors are: BARATINOO_DEFAULT_ENCODING, BARATINOO_ALL_PARSING, voice index 0.
2- Call `init()` method to set the text to synthetize. `init()` returns 0 if an error occurs (text is null or allocation error) or 1 if all is right. The text is stored in an internal buffer. Then, the `readText()` member function will be automatically called each time the engine needs datas.
3- When all the text is processed, you can reuse this object by calling the `init()` method again.
4- Use the class methods `deleteInstance()` to delete the InputTextBuffer object (do not use the InputTextBuffer object destructor).

8.8 Class `OutputSignalBuffer`

This class is an implementation of the virtual `OutputSignal` class. Source files of this object are baratinooio.h and baratinooio.cpp.
This implementation is simple to use:

1- Use the class methods `newInstance()` to create an OutputSignalBuffer object (do not use the OutputSignalBuffer object constructor). You can set the signal coding format and the synthesized signal frequency. Default behaviors are: BARATINOO_PCM, 24000Hz. This object has an `writeSignal()` method which stores all the signal data sent by the engine in an internal buffer.
2- When the engine has processed all the input text, check first if an error has occurred using `isError()` method. Then, you can get the signal data using `getSignalBuffer()` and `getSignalLength()` methods to get the address of signal buffer and the number of signal bytes. You can get an WAV header or an AU header corresponding to the signal buffer by calling `getSignalHeader()` method (call `deleteSignalHeader()` when you're done with it). It is useful if you want to create a WAV or AU file: you just need to get the header and put it in file and then get the signal and append it in file. The `detachSignal()` method is useful if you want keep the buffer without reallocate it.
3- Call the `resetSignal()` method to clear the buffer when you have finished. The object can now be reused for a new processing.
4- Use the class methods `deleteInstance()` to delete the OutputSignalBuffer object (do not use the OutputSignalBuffer object destructor).

9 C API description

A C API is also available in the Voxygen Expressive Speech C/C++ SDK. It is iso-functional with its C++ counterpart: for each object of the C++ API there is a set of C functions. Each function of a set matches an equivalent method in the object.
These sets are listed below.

9.1 Library functions

```
BARATINOO_INIT_RETURN BCinitlib(BaratinooTraceCB traceCB)

BARATINOO_INIT_RETURN BCinitlibWithKey(BaratinooTraceCB traceCB,
                            BaratinooKeyCB keyCB)

void BCterminatelib()

const char *BcgetBaratinooVersion()
```

9.2 BaratinooTrace object

You must declare a function with the following prototype which will be passed to the `Bcinitlib or BCinitlibWithKey` function:

```
void (*BaratinooTraceCB)(BaratinooTraceLevel level,
       int engineNumber, const char *source,
       const void *privatedata, const char *format, va_list args)
```

9.3 InputText object

You must declare and implement a function with the following prototype:

```
int (*BaratinooInputTextCB)(void *privateData, void *address,
```

```
        int length)
```

9.4 OutputSignal object

You must declare and implement a function with the following prototype:

```
int (*BaratinooOutputSignalCB)(void *privateData,
       const void *address, int length)
```

9.5 BaratinooEngine object

You use C functions the same way as the C++ methods. You just need to pass the engine pointer at the C function.

Create the engine :

```
BCengine BCnew(const void *privatedata)
```

Initialize the engine: use one of the following functions:

```
void BCinit(BCengine engine, const char *config)
```

```
void BCinitWithVoices(BCengine engine, const char *config,
          const char *voicesFilter)
```

```
void BCinitWithFilters(BCengine engine, const char *config,
          const char *voicesFilter, const char *languagesFilter,
          const char *gendersFilter)
```

Delete the engine

```
void BCdelete(BCengine engine);
```

Manage the engine

```
int BCgetEngineNumber(BCengine engine);
```

```
BARATINOOC_STATE BCgetState(BCengine engine);
```

```
BARATINOOC_STATE BCsetInputText(BCengine engine, BaratinooInputTextCB cb,
       void *privateData, BARATINOO_PARSING parsing,
       BARATINOO_TEXT_ENCODING encoding, int voiceIndex, const char
       *voiceModules, const char *uri);
```

```
void BCsetOutputSignal(BCengine engine,
       BaratinooOutputSignalCB cb, void *privateData,
       BARATINOO_SIGNAL_CODING coding, int frequency);
```

```
BARATINOOC_STATE BCprocessLoop(BCengine engine, int count);
```

```
BARATINOOC_STATE BCpurge(BCengine engine);
```

Manage events

```
void BCsetWantedEvent(BCengine engine, BARATINOO_EVENT_TYPE type);
```

```
void BCunsetWantedEvent(BCengine engine,
       BARATINOO_EVENT_TYPE type);
```

```
void BCsetAllWantedEvent(BCengine engine);
```

```
void BCunsetAllWantedEvent(BCengine engine);
```

```
BaratinooEvent BCgetEvent(BCengine engine);
```

Check voices

```
int BCgetNumberOfVoices(BCengine engine);

BaratinooVoiceInfo BCgetVoiceInfo(BCengine engine,int voiceNumber);

BARATINOO_CANSPEAK BCcanSpeak (BCengine engine, int voiceNumber, const char
       *language_iso639, const char *region_iso3166, const char *variant);
```

9.6 InputTextBuffer object

Instead of calling `BCsetInputText` and its callback mechanism, you can use a `BCinputTextBuffer` :

Creating and Deleting InputTextBuffer :

```
BCinputTextBuffer BCinputTextBufferNew(BARATINOO_PARSING parsing,
       BARATINOO_TEXT_ENCODING encoding, int voiceIndex, char
       *voiceModules);

void BCinputTextBufferDelete(BCinputTextBuffer inputTextBuffer);
```

Set text :

```
int BCinputTextBufferInit(BCinputTextBuffer inputTextBuffer, const char
       *text);
```

Assign InputTextBuffer to Engine :

```
BARATINOOC_STATE BCinputTextBufferSetInEngine(BCinputTextBuffer
       inputTextBuffer, BCengine engine);
```

9.7 OutputSignalBuffer object

Instead of calling `BCsetOutputSignal` and its callback mechanism, you can use a `BCoutputSignalBuffer` :

Creating and Deleting OutputSignalBuffer :

```
BCoutputSignalBuffer BCoutputSignalBufferNew(BARATINOO_SIGNAL_CODING
coding, int frequency);

void BCoutputSignalBufferDelete(BCoutputSignalBuffer
outputSignalBuffer);
```

Assign OutputSignalBuffer to Engine :

```
void BCoutputTextBufferSetInEngine(BCoutputSignalBuffer
outputSignalBuffer, BCengine engine);


int BCoutputSignalBufferIsError(BCoutputSignalBuffer
outputSignalBuffer);
```

Getting signal and header :

```
char * BCoutputSignalBufferGetSignalBuffer(BCoutputSignalBuffer
outputSignalBuffer);

int BCoutputSignalBufferGetSignalLength(BCoutputSignalBuffer
outputSignalBuffer);

void BCoutputSignalBufferResetSignal(BCoutputSignalBuffer
outputSignalBuffer);

void * BCoutputSignalBufferDetachSignal(BCoutputSignalBuffer
outputSignalBuffer);
```

```
void BCoutputSignalBufferDeleteSignal(void *signal);

int BCoutputSignalBufferGetSignalHeader(BCoutputSignalBuffer
outputSignalBuffer, OUTPUTSIGNALBUFFER_HEADER_FORMAT format, char
**pHeader);

void BCoutputSignalBufferDeleteSignalHeader(char *header);
```

10 Build process

10.1 Header file

Include `baratinoo.h` file in your source code.
If you use the Baratinoo inputTextBuffer or outputSignalBuffer objects provided with this API, include `baratinooio.h` files in your source code.

10.2 Linking

Release mode

Link your application with the dynamic release/`libbaratinoo.lib` file.
IMPORTANT Because of the activation protection in this library **you cannot debug** your application. If you try to do it your program will exit at engine initialisation.

Debug mode

Use the <**platform_debug>/`libbaratinoo.dll`** library instead of the <platform>/ `libbaratinoo.dll`.
IMPORTANT: the <platform_debug>/`libbaratinoo.dll` must be used only in debug. You must not use this library for your production.
If you use this library outside the debugger the output signal will consist of garbage noise instead of speech.

Troubleshooting release-specific errors

If you desperately need to debug a release-specific issue, only found when using the <pl atform>/`libbaratinoo.dll` file, you can insert a breakpoint in your application after engine initialisation and then attach the debugger to your application.

UWP applications

When using Microsoft Visual Studio 2015 to build a UWP application, you have to add the extension *"Microsoft Visual C++ 2013 Runtime package for Windows Universal"* to the references of your project.

10.3 At runtime

You need to have at least a voice installed and also its activation file in order for the engine to start.
For 64 bits Linux platform please ensure you have installed the 32 bits compatibility libraries before running your application compiled in 32 bits mode.

11 Client deployment

11.1 What you need to deploy

In order for the synthesis to work fine on a client terminal, you must install the following data :

- the config subdirectory containing the baratinoo.cfg and at least data files for one voice and language
- a valid activation file in the config subdirectory for each deployed voice (may be either a single global file or a file per voice)

If your client can use mark-up for improving synthesis, you may also deliver the following documents :

- The baratinoo in-house format described in the document [VOX32] located in the directory baratinoo8.1_1/doc/
- The standardized SSML format described in the document [VOX31] located in the directory baratinoo8.1_1/doc/
-

11.2 How to get an activation file for a client terminal

You need to have a valid activation file installed on the client terminal.

You should follow the same process as described in section 3.1 Installation process : *« Get and install activation file for voices ».*

12 Third party softwares

12.1 OPUS

```
Copyright 2001-2011 Xiph.Org, Skype Limited, Octasic,
                    Jean-Marc Valin, Timothy B. Terriberry,
                    CSIRO, Gregory Maxwell, Mark Borgerding,
                    Erik de Castro Lopo

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

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

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

- Neither the name of Internet Society, IETF or IETF Trust, nor the
names of specific contributors, may be used to endorse or promote
products derived from this software without specific prior written
permission.

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

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

Opus is subject to the royalty-free patent licenses which are specified at:

Xiph.Org Foundation:
https://datatracker.ietf.org/ipr/1524/

Microsoft Corporation:
https://datatracker.ietf.org/ipr/1914/

Broadcom Corporation:
https://datatracker.ietf.org/ipr/1526/