YOUR VOICE HAS NO LIMIT

Getting Started with VSDK (C++ Edition)

Getting Started with VSDK (C++ Edition)

The Vivoka SDK is meant to greatly facilitate the creation of voice-enabled applications, regardless of who's providing the underlying technologies!

Installation

Upon getting your access through your sales representative, the SDK is installable through Conan, the C++ package manager. Please refer to the accompanying document `vdk_tutorials_vsdk_compiling_samples_with_conan.pdf` for dedicated setup instructions.
In case Conan is already configured but you don't want to go through installing a sample, you can directly install libraries like so:

```
# List the packages you have access to
conan search -r vivoka-customer 'vsdk-*/*@vivoka/customer'
# Install a specific version
conan install <package_reference>
```

Configuration

All VSDK engines are to be initialized with a JSON configuration file. This file contains a version, then each engine will declare its own configuration block:

```
{
  "version": "2.0",
  "vasr": {
    ...
  },
  ...
}
```

We **strongly** recommend you put this file under a `config` directory because most engines will generate additional configuration files or use a cache (configurable).

Library versions

You can access the versions of both VSDK and the engine like so:

```
#include <vsdk/global.hpp>

// Engine creation is explained later
fmt::print("VSDK v{} ; Engine v{}\n", Vsdk::version(), engine->version());
```

Error Handling

The SDK will throw exceptions as a way to report errors and avoid having to check every single function call. An exception stack is made to help track the origin of the error, so the following base program is recommended to print the whole stack of errors:

```
#include <vsdk/Exception.hpp>

int main() try
{
    // use VSDK here
    return EXIT_SUCCESS;
}
catch (std::exception const & e)
{
    fmt::print(stderr, "A fatal error occured:\n");
    Vsdk::printExceptionStack(e);
    return EXIT_FAILURE;
}
```

Please note that some part of the SDK might run on another thread of execution, and exceptions

can't cross thread boundaries. You have to protect your threads from exceptions by either catching them inside or sending tasks to the main thread.

Audio Management

VSDK being a SDK around voice technologies, audio is a central component. Starting from version 6, **audio pipelining** has been added for greater power and simplicity.

Pipeline

An audio pipeline is composed of 3 types of audio modules: one **Producer**, zero or more **Modifiers** and zero or more **Consumers**. Simply put: a producer sends audio (either synchronously or asynchronously) into modifiers (if any), and the resulting audio is finally given to consumers (if any).

```
#include <vsdk/audio/Pipeline.hpp>
#include <vsdk/audio/producers/File.hpp>
#include <vsdk/audio/consumers/File.hpp>

using namespace Vsdk::Audio;

void copyAudioFile(std::string const & inPath, std::string const & outPath)
{
    Pipeline p;

    // Here we construct the modules directly into the pipeline but you can do so outside!
    p.setProducer<Producer::File>(inPath);
    p.pushBackConsumer<Consumer::File>(outPath);
    p.run(); // blocking call

    // Asynchronous execution would rather call p.start() and p.stop() instead
    // but not all ProducerModule support it!
}
```

If you would like to create your own Audio Module, inherit `ProducerModule`, `ModifierModule` or `ConsumerModule` from `Pipeline.hpp`.

ASR

VDK 3 features three different ASR libraries: **CSDK**, **TNL** and our very own: **VASR**.

Basics

You will need to manipulate 2 concepts: Recognizers & Models. Both need to be configured but first let's explain who's who.
Models are fed to the Recognizer and describe the range of words and utterances that can be recognized. They will either be pre-compiled by the provider (like “free speech” models), or compiled from a grammar that you've written beforehand in the VDK Studio.
There are 3 types of models:

Type	Description
static	Static models embedd all possible vocabulary inside a single file or folder.
dynamic	Dynamic models have “holes” where you can plug new vocabulary at runtime. These need to be prepared and compiled at runtime before installing it on a recognizer.
free-speech	Free-Speech models are very large vocabulary static models. They often require additional files and are not supported by all engines.

Recognizers inherit `Audio::ConsumerModule` and report results as they receive audio and compare it to the current models data.

Configuration

Each engine has its own configuration quirks and tweaks, but here is a common (though incomplete) pattern using VSDK-CSDK, which supports all 3 types of models:

```
{
  "version": "2.0",
  "csdk": {
    "paths": {
      "data_root": "../data"
    },
    "asr": {
      "recognizers": {
        "rec": { ... }
      },
      "models": {
        "static_example": {
          "type": "static",
          "file": "<model_name>.fcf"
        },
        "dynamic_example": {
          "type": "dynamic",
          "file": "<base_model_name>.fcf",
          "slots": {
            "firstname": { ... },
            "lastname": { ... }
          },
          ...
        },
        "free-speech_example": {
          "type": "free-speech",
          "file": "<base_model_name>.fcf",
          "extra_models": { ... }
        }
      }
    }
  }
}
```

Starting the engine

```
#include <vsdk/asr/csdk.hpp> // underlying ASR engine, here we choose CSDK
using AsrEngine = Vsdk::Asr::Csdk::Engine;

Vsdk::Asr::EnginePtr const engine = Vsdk::Asr::Engine::make<AsrEngine>("config/vsdk.json");
// engine is a std::shared_ptr, copy it around as needed but don't let it go out of scope while you need it!
// const here means the pointer is const, not the pointee (the Engine)
```

You can't create two separate instances of the same engine! Attempting to create a second one will get you another pointer to the existing engine. Terminate the first engine (i.e. let it go out of scope) then you can make a new instance.

That's it! If no exception was thrown your engine is ready to be used.
Each engine has its own configuration document, check it out for further details, as well as the ASR samples to get started with actual, production-ready code.

Creating a Recognizer

```
auto const rec = engine->recognizer("rec"); // Instantiate the recognizer we configured above
```

You can then plug yourself to the reporting mechanism:

```
rec->subscribe([] (Vsdk::Asr::Recognizer::Event  const & e) { ... });
rec->subscribe([] (Vsdk::Asr::Recognizer::Error  const & e) { ... });
rec->subscribe([] (Vsdk::Asr::Recognizer::Result const & r) { ... });
```

And finally, apply a model to actually recognize vocabulary:

```
rec->setModel("static_example"); // same call whether the model is static, dynamic or free-speech!
```

Also, don't forget to insert it in the pipeline or nothing's going to happen by itself:

```
p.pushBackConsumer(rec);
```

Dynamic Models

Only dynamic models need to be manipulated explicitely to add the missing data at runtime:

```
auto const model = engine->dynamicModel("dynamic_example");
model->addData("firstname", "André");
model->addData("lastname", "Lemoine");
model->compile();
// We can now apply it to a recognizer!
rec->setModel("dynamic_example"); // Or use setModel(model->name())
```

TTS

VDK 3 features three TTS libraries: **CSDK**, **Baratinoo** and **VtApi**.

Configuration

TTS engines must be configured before the program starts. Here is a complete setup with 2 channels, one for each language possible (this time using the Baratinoo engine):

```
{
  "tts": {
    "data_path": "../data", // This is relative to vsdk.json itself, NOT the program's working dir!
    "channels": {
      "channel_fr": { "voices": ["Arnaud_neutre"] },
      "channel_en": { "voices": ["Laura"] }
    }
  }
}
```

An empty channel list will trigger an error, as well as an empty voice list!

Voice format

Each engine has its own voice format, described in the following table:

Engine	Format	Example
vsdk-csdk	`<language>,<name>,<quality>`	`enu,evan,embedded-pro`
vsdk-vtapi	`<name>,<quality>`	`alice,d22`
vsdk-baratinoo	`<name>`	`Arnaud_neutre`

Starting the engine

```
#include <vsdk/tts/baratinoo.hpp>
using TtsEngine = Vsdk::Tts::Baratinoo::Engine;

auto const engine = Vsdk::Tts::Engine::make<TtsEngine>("config/vsdk.json");
```

Listing the configured channels and voices

```cpp
// With C++17 or higher
for (auto const & [channel, voices] : engine->availableVoices())
    fmt::print("Available voices for '{}': ['{}']\n", channel, fmt::join(voices, "'; '"));

// With C++11 or higher
for (auto const & it : engine->availableVoices())
    fmt::print("Available voices for '{}': ['{}']\n", it.first, fmt::join(it.second, "'; '"));
```

Creating a channel

Remember, channel must be configured beforehand!

```cpp
Vsdk::Tts::ChannelPtr const channelFr = engine->channel("channel_fr");
channelFr->setCurrentVoice("Arnaud_neutre"); // mandatory before any synthesis can take place
```

You can also activate a voice right away:

```cpp
auto const channelEn = engine->makeChannel("channel_en", "laura");
// ChannelPtr is also a std::shared_ptr
```

The engine instance can't die while at least one channel instance is alive. Destruction order is important!

Speech Synthesis

Speech Synthesis is **synchronous**! That means the call will block the thread until the synthesis is done or an error occured. If you need to keep going, put that in another thread.

```cpp
Vsdk::Audio::Buffer const resultFr = channelFR->synthesizeFromText("Bonjour ! Je suis une voix synthétique.");

// Also works with SSML input
auto const ssml = R"(<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis" xml:lang="en-US">
                       Here is an <say-as interpret-as="characters">SSML</say-as> sample.
                     </speak>)";
auto const resultEn = channelEn->synthesisFromText(ssml);
```

You can load your text or SSML input from a file too:

```cpp
auto const result = channel->synthesisFromFile("path/to/file");
```

`Audio::Buffer` is **NOT** a pointer type! Avoid copying it around, prefer **move operations**.

Playing the result

VSDK provides a cross-platform player in the `vsdk-audio-portaudio` package. Playing the result is very easy:

```cpp
Vsdk::Audio::Consumer::PaPlayer player;
player.play(buffer.data(), buffer.sampleRate(), buffer.channelCount());
// Or more simply
player.play(buffer);
```

The audio data is a 16bit signed Little-Endian PCM buffer. Channel count is always 1 and sample rate varies depending on the engine:

Engine	Sample Rate (kHz)
csdk	`22050`
baratinoo	`24000`
vtapi	`22050`

Storing the result on disk

```
buffer.saveToFile("path/to/file.pcm");
```

Only PCM extension is available, which means the file has no audio header of any sort. You can play it by supplying the right parameter, i.e.:

```
aplay -f S16_LE -s <sample_rate> -c 1 file.pcm
```

Or add a WAV header:

```
ffmpeg -f s16le -ar <sample_rate> -ac 1 -i file.pcm file.wav
```

Voice Biometrics

VDK 3 features two Voice Biometrics libraries: **TSSV** and **IDVoice**.

Configuration

Voice biometrics engines must be configured before the program starts. Here is a complete setup for the TSSV provider:

```
{
    "version": "2.0",
    "tssv": {           // Contrary to other technologies,
        "biometrics": { // biometrics paths are relative to the program's working directory!
            "generated_models_path": "data/models",
            "background_model_TI": "data/text-independent-16kHz.ubm",
            "background_model_TD": "data/text-dependent-16kHz.ubm"
        }
    }
}
```

Starting the engine

```
#include <vsdk/biometrics/tssv.hpp>
using BioEngine = Vsdk::Biometrics::Tssv::Engine;

auto const engine = Vsdk::Biometrics::Engine::make<BioEngine>("./config/vsdk.json");
```

Creating a model

Models contain enrollment data that recognition operations need.

```
// To create a text independant model
auto const model = engine->makeModel("test_ti", Vsdk::Biometrics::ModelType::TEXT_INDEPENDANT);

// To create a text dependant model
auto const model = engine->makeModel("test_td", Vsdk::Biometrics::ModelType::TEXT_DEPENDANT);
```

Checking users in the model

If a model was previously created with the same name it will be loaded. You can check enrolled users with:

```
fmt::print("Enrolled users: '{}'", fmt::join(model->users(), "', '"));
```

Adding a user to the model

You can either add raw audio data directly or from a file. After adding all the data for a given user, compile the model to finalize the enrollment process:

```
model->addRecord("victorien", "data/victorienti.wav");
model->compile();
```

The more data you give the model better the result will be.
Prefer to register the data in the condition of the use case of the model.

The format is preferred to be 16Khz mono-channel, you can create such wav file with the command:

```
arecord -c 1 -f S16_LE -r 16000 [filename].wav
```

Checking model info

Models report the status of the enrollment:

```
model->subscribe([] (Vsdk::Biometrics::Model::Event const & e) { ... });
model->subscribe([] (Vsdk::Biometrics::Model::Error const & e) { ... });
```

Performing authentication or identication

Both are covered in the same chapter as it is very similar:

```
// Identificator
auto authenticator = engine->makeIdentificator("ident", model, 5);
// Authenticator
auto identificator = engine->makeAuthenticator("auth", model, 5);
auto identificator->setUserToRecognize("victorien");
```

The only difference is that the authentication can only recognize user “victorien” and the identification can recognize every user enrolled in the model.
Note the third parameter: it is the confidence level you require. It ranges between 0 and 10 and act differently depending on your provider, 10 meaning you want your recognizer to be the strictest possible (you will have the lowest false positive but also the highest false negative).
They both inherit `Audio::ConsumerModule` and must be inserted into the pipeline.

Getting the result

```
#include <vsdk/biometrics/tssv/Constants.hpp>

identificator->subscribe([] (Vsdk::Biometrics::Identificator::Result const & result)
{
    namespace Key = Vsdk::Constants::Tssv::IdentResult;

    auto const user  = result.json[Key::id   ].get<std::string>();
    auto const score = result.json[Key::score].get<float>();
    fmt::print("Ident Result: '{}' (score: {})\n", user, score);
});

authenticator->subscribe([] (Vsdk::Biometrics::Authenticator::Result const & result)
{
    namespace Key = Vsdk::Constants::Tssv::AuthResult;

    auto const user  = result.json[Key::id   ].get<std::string>();
    auto const score = result.json[Key::score].get<float>();
    fmt::print("Auth Result: '{}' (score: {})\n", user, score);
});
```

Different providers will give you different results, for example IDVoice reports varying results as it

analyzes the audio, while TSSV only sends you result if the engine thinks it is acceptable (depending of the confidence level you set).
We recommend that you try it out the application in real situation to select your custom minimum score required to satisfy your need in false rejection and false acceptation. But by default you can just check if the score is above 0.

AFE

VDK 3 features only one Audio Front End: **VAFE**.

Configuration

Here is a template that needs to be added in the `vsdk.json` configuration file, using **VAFE**:

```
{
  "vafe": {
    "afe": {
      "analyzers": {
        "first":  { "type": "snr" },
        "second": { "type": "rt60", "bitspersample": 256 },
        "third":  { "type": "mos", "model": "path_to_mos_model.json" }
      },
      "filters": {
        "filter1": { "type": "bandpass", "lowfrequency": 80, "highfrequency": 400 },
        "filter2": { "type": "lowpass",  "frequency": 80 },
        "filter3": { "type": "highpass", "frequency": 400 }
      }
    }
  }
}
```

Includes

```
#include <vsdk/afe/vafe.hpp>
```

Creating an Analyzer

```
auto const snrAnalyzer = engine->makeAnalyzer(name);
snrAnalyzer->subscribe([] (Vsdk::Afe::Analyzer::Result const & r) { ... });
p.pushBackConsumer(snrAnalyzer);
```

Creating a Filter

```
auto const bandpassFilter = engine->makeFilter("filter1");
p.pushBackModifier(bandpassFilter);
```

Insertion Order

Multiple analyzers and filters can be registered. Order has an importance, especially the modifiers (here filters) as they are called sequentially. You can use various methods of inserting to control the order:

```
p.pushBackModifier(filter1);
auto it = p.pushBackModifier(filter3);
p.insertModifier(it, filter2); // inserts filter2 between 1 and 3
```

Analyzers

Every values are normalized in percentage, with 100% as best value possible.
Three analyzers type are available:

Type	Meaning	Description
SNR	Signal-Noise Ratio	Representation of the distance between speech and noise into the signal.

Type	Meaning	Description
RT60	Reverb-Time for 60dB	Algorithm used to evaluate echo present in the data.
MOS	Mean Opinion Score	Tries to evaluate audio as humans using machine learning.