

VoiceSpot API Documentation

Document history

Date	Version	Change
2020.11.17	0.1	Initial document covering VoiceSpot API v0.16.x.
2020.11.24	0.2	Updates to chapter on Adaptive Power State.
2020.12.16	0.3	Updated to VoiceSpot API v0.17.x + Added chapter on solutions combining a wake word and command model.
2021.01.05	0.4	Updated to VoiceSpot API v0.18.x
2021.02.12	0.5	Updated to VoiceSpot API v0.19.x
2021.04.08	0.6	Updated to VoiceSpot API v0.20.x
2021.07.16	0.7	Updated to VoiceSpot API v0.21.x, which adds model integrity checking as a separate function call.
2021.10.06	0.8	Updated to VoiceSpot API v0.22.x, which adds a callback signal to indicate that catch-up has ended.
2022.02.28	0.9	Updated to VoiceSpot API v0.23.x, which adds support for running with the model placed in read-only memory for all platforms.

Abbreviations

API	Application Programming Interface
MCPS	Mega Cycles Per Second
VAD	Voice Activity Detector

Contents

1	Introduction	5
2	Overview of Functionality and Interfaces.....	6
3	Initializing and running VoiceSpot	8
3.1	Initializing VoiceSpot	8
3.1.1	Creating the Common Control Structure.....	8
3.1.2	Creating a Single Instance	8
3.1.3	Checking and Opening a Model using an Instance	9
3.1.4	Thresholding Scores for Event Detection	10
3.1.5	Setting Parameters	11
3.1.6	Getting Library and Model Information.....	11
3.2	Running VoiceSpot	12
3.2.1	Processing of input audio	12
3.2.2	Checking for a Detection Event	13
3.2.3	Estimating start and stop time of detected utterances	14
3.2.4	Time-varying processing time	16
3.2.5	Resetting an Instance.....	17
3.3	Handling Multiple Instances.....	17
3.4	Closing and/or Releasing an Instance	17
3.5	Releasing the Control Structure	18
4	Adaptive Power State for Low-Power Always-On Operation.....	19
4.1	The Circular Buffer Struct.....	20
4.2	The Frame Callback Function	20
4.3	The Status Callback Function	21
4.4	Enabling Adaptive Power State	22
4.5	Speech Detection Without Using a Model.....	22
5	Solutions Combining a Wake Word Model and a Command Model.....	23
5.1	Implementation for a Memory Constrained System	23
6	List of Error Codes.....	24
7	Contact.....	26

1 Introduction

VoiceSpot is a compact speech detection engine that can be used for applications such as wake word detection and small vocabulary command detection. It is implemented as a C99 library and exposes an Application Programming Interface (API) used for interfacing to the library. This document is intended for developers looking to integrate and use VoiceSpot in a product and describes the available functionality and how to use it via the API.

2 Overview of Functionality and Interfaces

The VoiceSpot solution consists of the following components illustrated in Figure 1:

- The Detection Engine takes audio samples as input and computes a matching score.
- Using a threshold on the matching score to determine if a target utterance has been spoken. The threshold may be either provided manually or computed automatically using an adaptive thresholding mechanism in response to the audio environment.
- Optional gating of the detection engine execution if no speech is detected for low-power always-on applications.
- Estimation of start and stop time for a detection
- Support for multiple instances

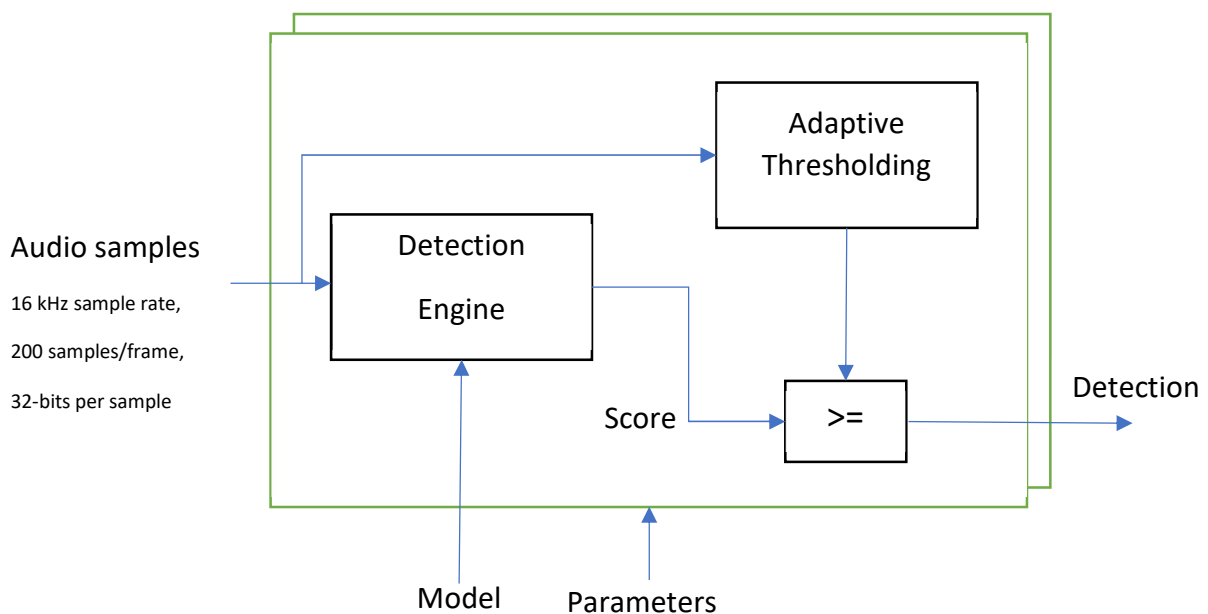


Figure 1: Overview of VoiceSpot components

The audio input provided to VoiceSpot must be using 16 kHz sample rate and usually delivered in frames each consisting of 200 samples per frame. The audio must be 32-bits per sample as it is often possible to get more than 16 effective bits from the microphone. The data type of the individual samples are either fixed-point or floating-point depending on the target platform.

The audio samples are fed to the Detection Engine and the Adaptive Thresholding mechanism. For each frame of input audio data, the detection engine will compute a number of scores. The number of scores available for a given input frame may be either 0 or 1 indicating if a matching score value has been provided or not. If a matching score is available, it can be compared to the used threshold in order to take decision on whether or

not a detection has been made. The used threshold may be chosen to be either a fixed threshold or an automatically computed threshold based on the Adaptive Thresholding mechanism.

3 Initializing and running VoiceSpot

This chapter will explain the steps needed to initialize and run one or more instances of VoiceSpot. Each instance acts as a container for the context of the Detection Engine and the Adaptive Threshold mechanism. To see it all put together in actual code, see the `basic_main.c` example application.

3.1 Initializing VoiceSpot

This section covers the steps involved in initializing VoiceSpot.

3.1.1 Creating the Common Control Structure

To initialize and run one or more instances of VoiceSpot, first we need to create the control structure for the instances. This is done by:

```
#include "public/rdsp_voicespot.h"

rdsp_voicespot_control* voicespot_control;

int32_t data_type = RDSP_DATA_TYPE__FLOAT32;

int32_t voicespot_status = rdspVoiceSpot_CreateControl(&voicespot_control,
data_type);
```

where `voicespot_control` is a pointer to the control struct that will be allocated from within the library. The variable `data_type` is the input data type to use, here

`RDSP_DATA_TYPE__FLOAT32` is used indicating 32-bit floating-point input, whereas `RDSP_DATA_TYPE__INT32` is available for 32-bit fixed-point input. The call to `rdspVoiceSpot_CreateControl()` will initialize the control struct `voicespot_control` and return an error code given here by `voicespot_status`. If the error code is not `RDSP_VOICESPOT_OK`, which has a value of 0, then something has likely gone wrong. For a list of possible error codes, see Chapter 23.

3.1.2 Creating a Single Instance

An instance is created by:

```
int32_t voicespot_handle;

int32_t enable_highpass_filter = 1;

int32_t generate_output = 0;

voicespot_status = rdspVoiceSpot_CreateInstance(voicespot_control,
&voicespot_handle, enable_highpass_filter, generate_output);
```


where `voicespot_control` is the pointer to the control structure and `voicespot_handle` is the handle provided by the library that identifies the instance.

The variable `enable_highpass_filter` configures if the built-in high-pass filter before the Detection Engine should be enabled (`=1`) or disabled (`=0`). It is generally recommended to have a high-pass filter before the Detection Engine, but if the audio stream has already been high-pass filtered elsewhere there is no need for enabling this. Notice that if the high-pass filter is enabled, the filtering will be done in-place on the input audio frame.

The variable `generate_output` configure if audio output must be provided from the instance. This is only used for debugging potential audio problems and should generally not be enabled for normal usage.

3.1.3 Checking and Opening a Model using an Instance

Before opening a model for the first time, it is good practice to check the integrity of the model. This will help guard against scenarios where the content of the model has been corrupted, which can otherwise be difficult to detect as it may or may not influence performance. The integrity of the model may be checked by the function call:

```
rdspVoiceSpot_CheckModelIntegrity(model_blob_size, model_blob);
```

Next, a VoiceSpot model must be used to open an instance. A VoiceSpot model is a binary blob provided as a binary model file that contains the acoustic fingerprints used to detect the utterances specified by the model. A model is opened using an instance as:

```
voicespot_status = rdspVoiceSpot_OpenInstance(voicespot_control, voicespot_handle,  
model_blob_size, model_blob, 0, 0);
```

where `model_blob_size` is the size of the model blob in bytes and `model_blob` is a `uint8_t` pointer to the binary model blob data. Notice that `model_blob` must be 16-byte aligned.

3.1.3.1 Opening a Model Stored in a Read-Only Location

On some target platforms, the VoiceSpot implementation will do in-place permutations of the internal model weights in order to optimize the target platform execution. However, this complicates placing the model in a read-only location such as flash memory since changes are made to the model blob during the opening operation.

To support this mode of operation, the argument `model_blob_is_already_open` is provided which allows using a modified model blob that has already been permuted. To use this functionality, open the model like normal in the target platform development environment

and save the resulting model blob to a read-only memory location, here termed `model_blob_open`. The modified model blob can now be opened solely from the read-only location as:

```
voicespot_status = rdspVoiceSpot_OpenInstance(voicespot_control, voicespot_handle,  
model_blob_size, model_blob_open, 1, 0);
```

The permuted model blob can therefore completely replace the original model blob in a customer implementation. However, notice that the integrity of the permuted model blob can not be checked by `rdspVoiceSpot_CheckModelIntegrity()` since it contains changes. To check the integrity of the permuted model blob, compare it against the result of opening the equivalent unpermuted model blob.

3.1.4 Thresholding Scores for Event Detection

In order to maximize the performance of the VoiceSpot solution, the used threshold can be adaptively adjusted based on the time since the last detection event (termed adaptive sensitivity) and the amount of speech present in the environment (termed adaptive threshold). The reason for doing so is that it is expected that users are more likely to interact with the device immediately after having interacted with the device and that the risk of falsely detecting e.g. a wake word increases with the amount of speech in the environment.

Enabling these adaptive modes of operation is done by:

```
// Adaptive threshold modes  
// 0: fixed threshold  
// 1: adaptive threshold  
// 2: adaptive sensitivity  
// 3: adaptive threshold + adaptive sensitivity  
int32_t adapt_threshold_mode = 3;  
voicespot_status = rdspVoiceSpot_EnableAdaptiveThreshold(voicespot_control,  
voicespot_handle, adapt_threshold_mode);
```

where `adapt_threshold_mode` is the mode used for thresholding. Adaptive threshold and adaptive sensitivity may be enabled/disabled separately. If neither is enabled, fixed threshold values is used instead. If desired, the user may provide their own thresholds during runtime, see Section 3.2.2.

The recommended mode of operation is to use `adapt_threshold_mode = 3` for wake word detection and `adapt_threshold_mode = 0` for command detection.

3.1.5 Setting Parameters

For each VoiceSpot model, a set of associated parameters are typically provided as a parameter file together with the model file. The parameters can be applied to an open instance by:

```
voicespot_status = rdspVoiceSpot_SetParametersFromBlob(voicespot_control,  
voicespot_handle, param_blob);
```

where `param_blob` is a `uint8_t` pointer to the binary parameter blob.

The parameter blob contains parameters for all the functionality of the VoiceSpot solution. If the parameter blob contains parameters for a feature that is not enabled at the time of the function call, e.g. the Adaptive Threshold mechanism, this will be indicated by the error code. If this is the case, the parameter setting will only be carried out for enabled features. It is therefore generally recommended that setting of parameters be carried out after all features that should be enabled have been enabled.

3.1.6 Getting Library and Model Information

Getting information about the library can be done by:

```
rdsp_voicespot_version voicespot_version;  
  
rdspVoiceSpot_GetLibVersion(voicespot_control, &voicespot_version);
```

where the struct `voicespot_version` contains the version of the VoiceSpot library being used in the form `voicespot_version.major`, `voicespot_version.minor`, `voicespot_version.patch`, `voicespot_version.build`.

Information about the model that is open can be determined by:

```
char* voicespot_model_string;  
  
char** voicespot_class_string;  
  
int32_t num_samples_per_frame;  
  
int32_t num_outputs;  
  
rdspVoiceSpot_GetModelInfo(voicespot_control, voicespot_handle, &voicespot_version,  
&voicespot_model_string, &voicespot_class_string, &num_samples_per_frame,  
&num_outputs);
```

where `voicespot_model_string` is a pointer to a UTF-8 string describing the model, `num_samples_per_frame` is the number of input audio samples expected per frame and `num_outputs` is the number of output classes being detected by the Detection Engine.

In order to know what utterance a given output class corresponds to, `voicespot_class_string` contains an array of pointers to a UTF-8 string, one for each output class.

3.2 Running VoiceSpot

The instance has now been opened and the parameters configured, so the instance is now ready to receive audio for detection. To do so, the function calls described in this Section are all used in the real-time pipeline of frame-based audio processing.

No dynamic memory allocation should be performed in the real-time pipeline since the execution time may be unpredictable. Instead all allocation should have been done during initialization.

3.2.1 Processing of input audio

Processing a frame of audio is done by:

```
int32_t num_scores;

int32_t scores[num_outputs];    // Allocate during initialization

int32_t processing_level = RDSP_PROCESSING_LEVEL_FULL;

voicespot_status = rdspVoiceSpot_Process(voicespot_control, voicespot_handle,
processing_level, (uint8_t*) frame_buffer, &num_scores, scores, NULL);
```

where `processing_level` is the processing level that controls how much processing is done internally in the VoiceSpot library. Use `processing_level = RDSP_PROCESSING_LEVEL_FULL` to generate output or use `processing_level = RDSP_PROCESSING_LEVEL_SKIP_OUTPUT`

if only the internal state should be updated but no output produced (will give lower computational complexity).

The pointer to the input audio frame buffer (32-bits per sample) is given by `frame_buffer`. Usually the frame size is 200 samples, but this should be determined from the instance (see Section 3.1.6) as it may vary. Notice that in-place processing may occur on the input frame buffer, so be sure that the content of the buffer is not needed after the function call.

The integer `num_scores` is being output from the library and contains the number of output scores available for the current input audio frame with the value being either 0 or 1. For `num_scores = 1`, then `num_outputs` score values are available in the array `scores`

corresponding to the matching score computed for each of the output classes for the present input audio frame. The matching scores are integers from the range 0 to 1024 (both included) and represent the probability of a match in the range 0% to 100%.

3.2.2 Checking for a Detection Event

Whenever a frame has been processed and `num_scores = 1`, the score value should be compared with a threshold to determine if a detection has occurred. This can either be done using the Automatic Thresholding mechanism (recommended) or manually.

3.2.2.1 Using Automatic Thresholding

Checking if a detection has occurred using Automatic Thresholding is done by:

```
int32_t allow_trigger;  
  
int32_t *event_thresholds = NULL;  
  
int32_t processing_period = 4;  
  
int32_t score_index = rdspVoiceSpot_CheckIfTriggered(voicspot_control,  
voicspot_handle, scores, allow_trigger, event_thresholds, processing_period);
```

where the output `score_index` indicates the index of the class for which a detection has occurred. If `score_index = -1`, no detection has occurred.

The input `scores` is the array of matching scores computed by the function call

```
rdspVoiceSpot_Process().
```

The input `allow_trigger` is a binary variable indicating if a detection should be allowed currently. If for example a detection has been found in the previous frame, the detection is very likely to also be present in the next. Detections may therefore be ignored for some time as indicated by the variable `allow_trigger`.

The input array `event_thresholds` holds any manually specified thresholds to use for the current frame. If `event_thresholds = NULL`, then the Automatic Thresholding mechanism is used to compute the thresholds to use.

The input `processing_period` is used to indicate the period (in frames) with which `rdspVoiceSpot_CheckIfTriggered()` is called. For example, `processing_period = 4` is the correct value to use if `processing_level = RDSP_PROCESSING_LEVEL_FULL` is used for every frame in `rdspVoiceSpot_Process()` and `rdspVoiceSpot_CheckIfTriggered()` is called whenever `num_scores = 1`.

3.2.2.2 Manual Thresholding

If desired, direct thresholding of the matching scores available in `scores` is also possible. To do so, simply compare the elements of `scores` with the selected thresholds. If manual thresholding is done in this manner, there is no need to call

```
rdspVoiceSpot_CheckIfTriggered().
```

3.2.3 Estimating start and stop time of detected utterances

Whenever a target utterance has been detected, it is possible to estimate the start and stop location of the utterance by:

```
int32_t start_offset_samples = 0;
int32_t stop_offset_samples = 0;
int32_t timing_accuracy = 4;
int32_t score_threshold = -1;

voicespot_status = rdspVoiceSpot_EstimateStartAndStop(voicespot_control,
voicespot_handle, score_index, score_threshold, timing_accuracy,
&start_offset_samples, &stop_offset_samples)
```

where the input `score_index` holds the class index of the detected utterance.

The input `score_threshold` is the threshold to use for the timing estimate. If Automatic Thresholding is used, simply input `score_threshold = -1`.

The input `timing_accuracy` indicates the requested timing accuracy in frames. The default is to use `timing_accuracy = 4`, but values of 8, 12 and 16 are available for trading off lower computational cost for reduced timing accuracy. When using `timing_accuracy = 4`, it is expected that many of the timing estimates are within +/- 50 ms of the ground truth, but some estimates may fall outside of this range. However, it is expected that virtually all timing estimates are within +/- 100 ms of the ground truth.

The outputs `start_offset_samples` and `stop_offset_samples` contain the estimated start and stop times represented as an offset compared to the current location in time where the detection occurred. The offsets are represented as positive numbers of samples back in time from the detection location as illustrated in Figure 2.

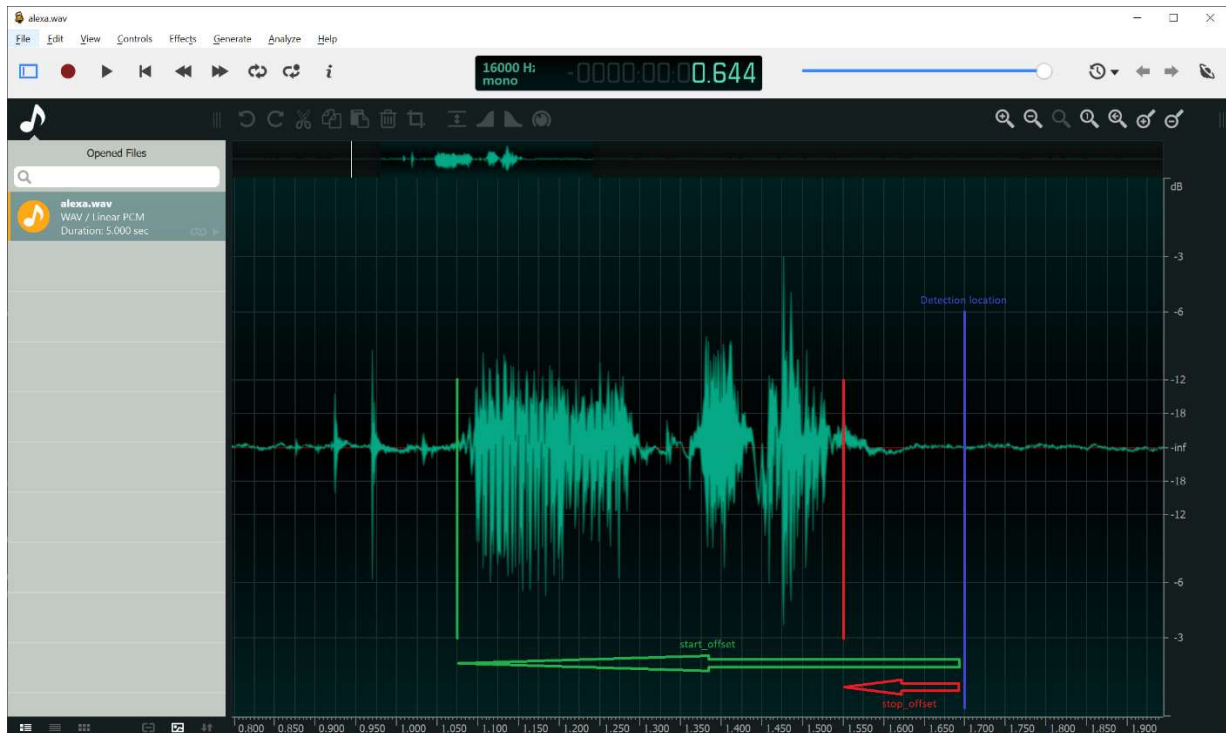


Figure 2: Example showing how the estimated start and stop time is reported as offset values.

If only the stop time is required, this function call should be used instead as it is less complex to compute:

```
voicespot_status = rdspVoiceSpot_EstimateStop(voicespot_control, voicespot_handle,
score_index, score_threshold, &stop_offset_samples)
```

If only the start time is required, this function call may be used instead:

```
int32_t rdspVoiceSpot_EstimateStart(rdsp_voicespot_control* voicespot_control,
int32_t voicespot_handle, int32_t score_index, int32_t score_threshold, int32_t
timing_accuracy, int32_t* start_offset);
```

As estimating the start time can take a significant number of cycles on some platforms, it is possible to break the computations into a number of equally sized parts by calling:

```
int32_t rdspVoiceSpot_EstimateStartSplit(rdsp_voicespot_control* voicespot_control,
int32_t voicespot_handle, int32_t score_index, int32_t score_threshold, int32_t
timing_accuracy, int32_t max_num_parts, int32_t* start_offset);
```

where `max_num_parts` is the maximum number of equal size parts the computation should be split into. As long as `*start_offset < 0`, the computation must then be continued by calling the following function:

```
int32_t rdspVoiceSpot_EstimateStartSplitContinue(rdsp_voicespot_control*
voicespot_control, int32_t voicespot_handle, int32_t timing_accuracy, int32_t*
start_offset);
```

3.2.4 Time-varying processing time

Since VoiceSpot typically only outputs a matching score for every 4th input frame, the computational load, as measured e.g. by the number of Mega Cycles Per Second (MCPS) required to run VoiceSpot, is not constant per frame. Instead, it varies in a periodic manner as the computational load is higher in the frame where the output is generated, see Figure 3. The exact ratio between the computational load for the low-MCPS frames and the high-MCPS frames varies from one target implementation to another and will therefore have to be tested on the used library.

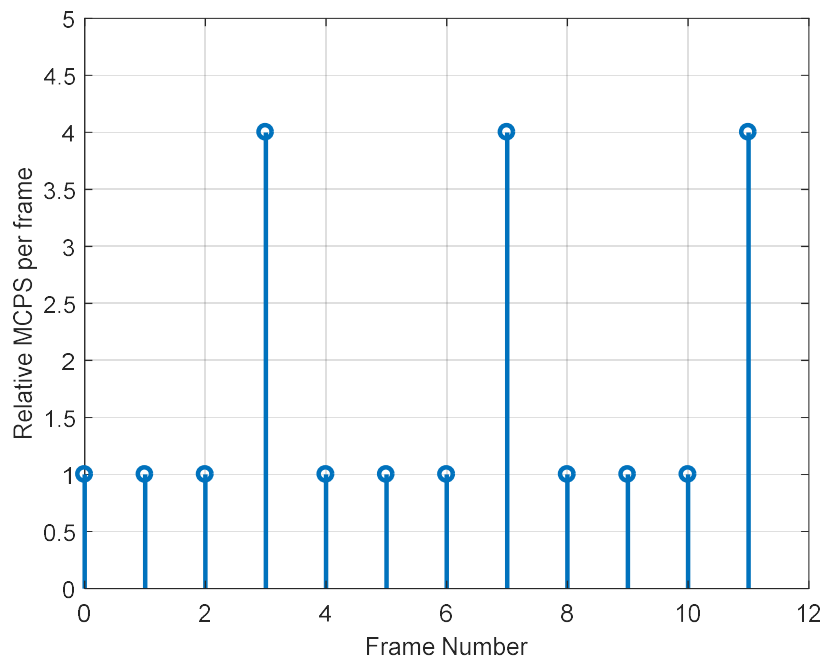


Figure 3: Typical time-varying computational load of `rdspVoiceSpot_Process()`.

It is therefore important that the application using VoiceSpot checks that the time taken by `rdspVoiceSpot_Process()` when processing the frame at which an output is generated, is compatible with the scheduling of tasks in the audio system. If the real-time deadline can not be met, an audio buffer will have to be inserted in the incoming audio stream to make

sure audio is not being lost while the processing is ongoing. In this manner, the processing unit can be clocked based on the average MCPS, not the peak MCPS.

Care should also be taken not to miss any real-time deadlines when estimating the start time (see Section 3.2.3) since this function can take some time to complete depending on the target platform.

3.2.5 Resetting an Instance

Resetting an instance back to the state it was in right after being initialized is possible by calling `rdspVoiceSpot_ResetProcessing(voicespot_control, voicespot_handle)`. Notice that after resetting an instance, it takes around 1 to 2 seconds of audio input before the instance is ready to perform the next detection.

Resetting only the state of the Adaptive Power State mechanism is possible by calling `rdspVoiceSpot_ResetAdaptivePowerState(voicespot_control, voicespot_handle)`.

3.3 Handling Multiple Instances

VoiceSpot supports running multiple instances side-by-side. This includes running multiple instances each using the same model and parameters as well as running multiple instances with different models and parameters. If the same model is used by multiple instances, the memory used for holding the model in memory will be reused internally between the instances.

Multi-threaded execution is currently not officially supported as it is not actively being tested, but running separate instances in separate threads is expected to work. If you see a need for this feature, please request the feature for official support.

To run multiple instances, use a common control structure for managing the instances and simply create, open and run the individual instances as described for the case of a single instance.

3.4 Closing and/or Releasing an Instance

An instance can be closed thereby partly deallocated by:

```
voicespot_status = rdspVoiceSpot_CloseInstance(voicespot_control,  
voicespot_handle);
```

An instance can be released and thereby fully deallocated by:

```
voicespot_status = rdspVoiceSpot_ReleaseInstance(voicespot_control,  
voicespot_handle);
```

3.5 Releasing the Control Structure

The control structure can be released by:

```
voicespot_status = rdspVoiceSpot_ReleaseControl(voicespot_control);
```

As the control structure manages all instances, releasing the control structure will automatically close and release all instances. The easiest way to perform a complete release of VoiceSpot is therefore to call this function.

4 Adaptive Power State for Low-Power Always-On Operation

In order to lower the computational complexity of running VoiceSpot for always-on operation, the Detection Engine may be turned off when no speech is detected in the environment. The computational load of the low-power mode is typically 10-20 times lower than that of the full processing mode, which can allow significant power savings at little to no cost in terms of missed detections. The gating of the Detection Engine is controlled by a Voice Activity Detector (VAD) operating on the input audio. The VAD controls whether an instance is in the low-power mode, where the Detection Engine is not executing, or in the full processing mode.

In order to not miss detecting a target utterance when a sudden onset of speech occurs, a circular buffer is needed for the input audio. This allows the Detection Engine to process audio also from before speech was detected when the state changes from low-power mode to full processing mode as illustrated in Figure 4. Here, the audio available to the Detection Engine after going from low-power mode to full processing mode is indicated by the Trigger Window.

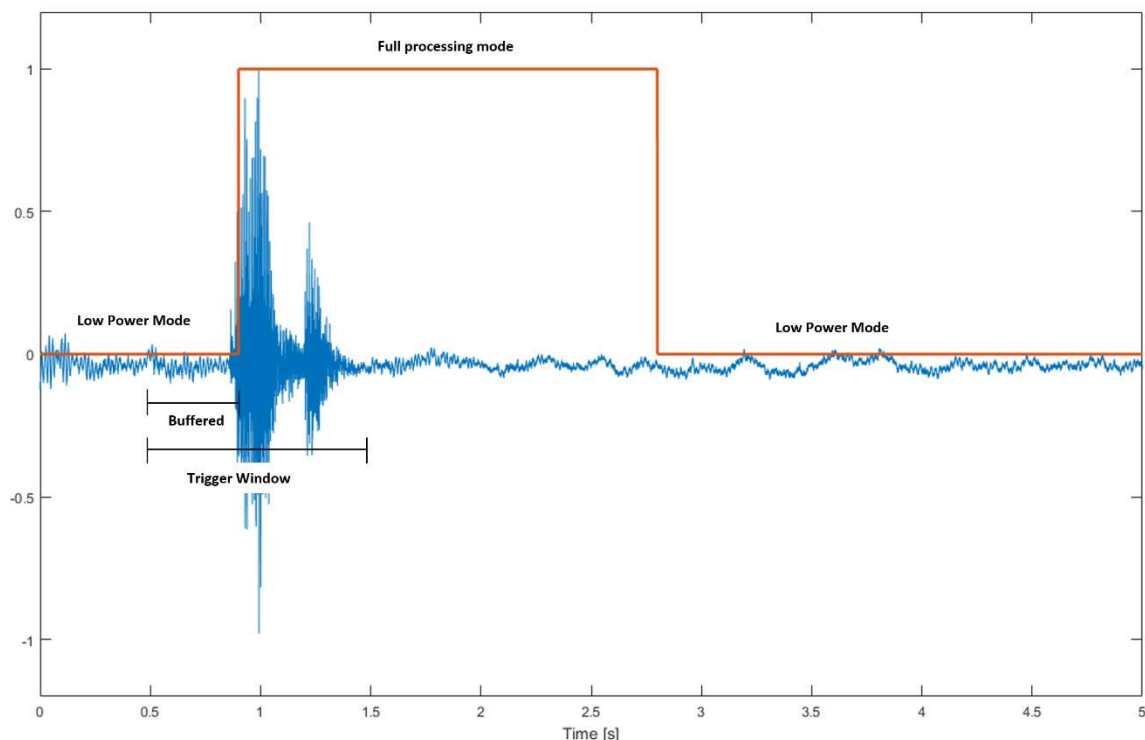


Figure 4: Illustration of adaptive power state change from low-power mode to full processing and back.

4.1 The Circular Buffer Struct

In order to manage the circular buffer, the following struct is defined in the header file

public/rdsp_voicespot_sleep_cbuffer.h:

```
typedef struct rdsp_voicespot_buffer_struct {  
    uint8_t* buffer;  
    int32_t read_index_samples;  
    volatile int32_t write_index_samples;  
    int32_t length_samples;  
} rdsp_voicespot_sleep_cbuffer;
```

Here `buffer` is the pointer to the start of the circular audio buffer, `read_index_samples` is the current read index in samples, `write_index_samples` is the current write index in samples and `length_samples` is the length of the circular audio buffer in samples. During initialization, the content of this struct should be configured using `read_index_samples = 0` and `write_index_samples = 0`.

Whenever audio data is written into the circular buffer, the variable `write_index_samples` must be updated accordingly.

4.2 The Frame Callback Function

In order for the Detection Engine to be able to go back and process audio in the circular input audio buffer, a callback function must be defined that provides the next input audio frame from the circular buffer.

The callback function must have the following function prototype:

```
void your_frame_callback_function(int32_t voicespot_handle, uint8_t**  
frame_pointer);
```

Whenever the callback function `your_frame_callback_function()` is called, the variable `voicespot_handle` will contain the handle of the instance from which the callback comes and `*frame_pointer` must contain a pointer to the next input audio frame as taken from the circular buffer. The data format of this next input audio frame must follow that used by `rdspVoiceSpot_Process()`, allowing the data format used in the circular buffer to differ from that used by `rdspVoiceSpot_Process()`. If desirable, this allows compression to be applied to the audio in the circular buffer.

Registering the function `your_frame_callback_function` as the frame callback function is done by:

```
voicespot_status = rdspVoiceSpot_RegisterReadFrameCallback(voicespot_control,  
voicespot_handle, your_frame_callback_function);
```

4.3 The Status Callback Function

If you would like to be notified whenever there is a change in the power state, you can make a status callback function that will be called every time the status changes. The status callback must have the following function prototype :

```
void your_status_callback_function(int32_t voicespot_handle,  
rdsp_voicespot_processing_status processing_status);
```

where the variable `voicespot_handle` is the handle of the instance from which the callback is made and `rdsp_voicespot_processing_status` is an enum that can take on the values:

```
{RDSP_IS_PROCESSING_FULL, RDSP_IS_PROCESSING_VERY_LOW,  
RDSP_REQUEST_FULL_PROCESSING, RDSP_IS_PROCESSING_NORMAL}.
```

Whenever the power state changes, the status callback function will be called with `processing_status` indicating the status.

A value of `RDSP_REQUEST_FULL_PROCESSING` indicates that VoiceSpot would like to go into full processing mode. This can be used for e.g. increasing the clock frequency of the device to match the increased computational load.

A value of `RDSP_IS_PROCESSING_FULL` indicates that VoiceSpot is now in full processing mode.

A value of `RDSP_IS_PROCESSING_VERY_LOW` indicates that VoiceSpot is now in low-power mode. This can be used for e.g. lowering the clock frequency of the device.

A value of `RDSP_IS_PROCESSING_NORMAL` indicates that VoiceSpot has completed the catch-up phase and is now in normal processing mode. This can be used for e.g. lowering the clock frequency of the device.

Registering the function `your_status_callback_function` as the status callback function is done by:

```
voicespot_status =  
rdspVoiceSpot_RegisterPowerStateStatusCallback(voicespot_control, voicespot_handle,  
your_status_callback_function);
```

4.4 Enabling Adaptive Power State

When the circular buffer struct `voicespot_buffer_struct` has been initialized, the Adaptive Power State functionality may be enabled for a given instance by:

```
rdsp_voicespot_sleep_cbuffer voicespot_buffer_struct;  
  
// Insert voicespot_buffer_struct initialization here  
  
int32_t adapt_power_state_mode = 1;    // 0 = Off, 1 = On  
  
voicespot_status = rdspVoiceSpot_EnableAdaptivePowerState(voicespot_control,  
voicespot_handle, adapt_power_state_mode, &voicespot_buffer_struct);
```

4.5 Speech Detection Without Using a Model

If desired, the VAD functionality of the Adaptive Power State mechanism may be used stand-alone without using the Detection Engine. This may be useful for speech detection without the need for utterance detection. To operate in this mode, create an instance without opening it using a model blob and use it like the normal mode of operation.

5 Solutions Combining a Wake Word Model and a Command Model

A wake word model and a command model may be combined into an overall solution capable of handling a wake word plus command in one utterance, for example “*Hey VoiceSpot, Volume up*” for turning up the playback volume. In this example, “*Hey VoiceSpot*” is the wake word recognized by the wake word model and “*Volume up*” is the command recognized by the command model. This chapter outlines the recommended way to implement such a solution in a memory constrained system.

5.1 Implementation for a Memory Constrained System

The first step of the implementation is to detect a wake word as described in the previous chapters. In order to reduce the amount of memory needed to run the two different models, the idea is to close and release the wake word model once a wake word has been detected. This enables VoiceSpot to only require memory for running a single model at a time as there is never two models open at the same time, but instead using a sequence of the models. Since there can be very little time from the end of the wake word and until the start of the command, an input audio buffer is required for the command model in order not to miss any audio. It is therefore convenient and memory saving to utilize a single system-wide audio buffer for this purpose, serving the purposes of buffering audio for VoiceSpot’s Adaptive Power State mechanism, a potential command model as well as for buffering of audio to be forwarded to a cloud service or host system.

The different processing states involved in the sequence are as follows:

1. Run the wake word model on the real-time audio while buffering the input audio just like for a stand-alone wake word model, e.g. using the Adaptive Power State mechanism. Once a wake word has been detected, go to processing state 2.
2. Close and release the wake word model instance. Create and open a command model instance. Feed all input audio stored in the input audio buffer into the command model instance, e.g. 0.5 seconds of audio or whatever length the audio input buffer has. Now go to state processing 3.
3. Run the command model instance on the real-time audio until a command is detected or a timeout for this command state is reached, then close and release the command model instance, create and open a wake word model instance and go back to state processing 1.

6 List of Error Codes

The possible error codes that may be returned by a function call are defined in the header files `public/rdsp_voicespot_defines.h` and `public/rdsp_model_defines.h` and are shown in Table 1.

Error code	Value	Meaning
<code>RDSP_VOICESPOT_OK</code>	0	No errors detected
<code>RDSP_VOICESPOT_MALLOC_FAIL</code>	-1	Memory allocation failed
<code>RDSP_VOICESPOT_INVALID_POINTER</code>	-2	An invalid pointer detected
<code>RDSP_VOICESPOT_UNSUPPORTED_DATA_TYPE</code>	-3	Unsupported data type requested
<code>RDSP_VOICESPOT_INVALID_HANDLE</code>	-4	An invalid handle detected
<code>RDSP_VOICESPOT_NO_FREE_HANDLE</code>	-5	There are no free handles available
<code>RDSP_VOICESPOT_OPEN_FAILED</code>	-6	Opening of the model failed
<code>RDSP_VOICESPOT_NOT_OPEN</code>	-7	The model instance is not open
<code>RDSP_VOICESPOT_NO_TRIGGER_FOUND</code>	-8	No valid trigger found for timing estimation
<code>RDSP_VOICESPOT_UNABLE_TO_CLOSE_MASTER</code>	-9	Master can not be closed before all slaves have been closed
<code>RDSP_VOICESPOT_UNABLE_TO_OPEN_AS_SLAVE</code>	-10	Slave model is not compatible with master
<code>RDSP_VOICESPOT_BLOB_NOT_VECTOR_ALIGNED</code>	-11	The model blob is not aligned to the vector boundaries, typically 16 bytes
<code>RDSP_VOICESPOT_LICENSE_EXPIRED</code>	-12	The SW license has expired, please renew
<code>RDSP_VOICESPOT_INVALID_ID</code>	-13	An invalid ID was used

RDSP_VOICESPOT_PARAMETER_NOT_AVAILABLE	-14	The parameter is not available in the current state
RDSP_VOICESPOT_VALID_THRESHOLD_NOT_AVAILABLE	-0x100	No valid threshold value available for event detection
RDSP_VOICESPOT_BUFFER_UNDERFLOW	-0x101	The input buffer has experienced underflow

Table 1: List of possible error codes.



7 Contact

Retune DSP ApS

Diplomvej 381

DK-2800 Kgs. Lyngby

Denmark

Email: info@retune-dsp.com

Phone: +45 2728 2817

<http://www.retune-dsp.com>