NXP Semiconductors
User Guide

Document identifier: VSAPIDUG
Rev. 0, 06 April 2022

VoiceSpot API Documentation

Contents

Chapter 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 to interface 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.

Chapter 2
Overview of functionality and interfaces

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

- The Detection Engine takes audio samples as an input and computes a matching score.
- Use a threshold on the matching score to determine whether a target utterance was spoken. The threshold may be either provided manually or computed automatically using an adaptive thresholding mechanism in response to the audio environment.
- The optional gating of the detection engine is executed if no speech is detected for low-power always-on applications.
- Estimate the start and stop time for a detection.
- Multiple instances are supported.

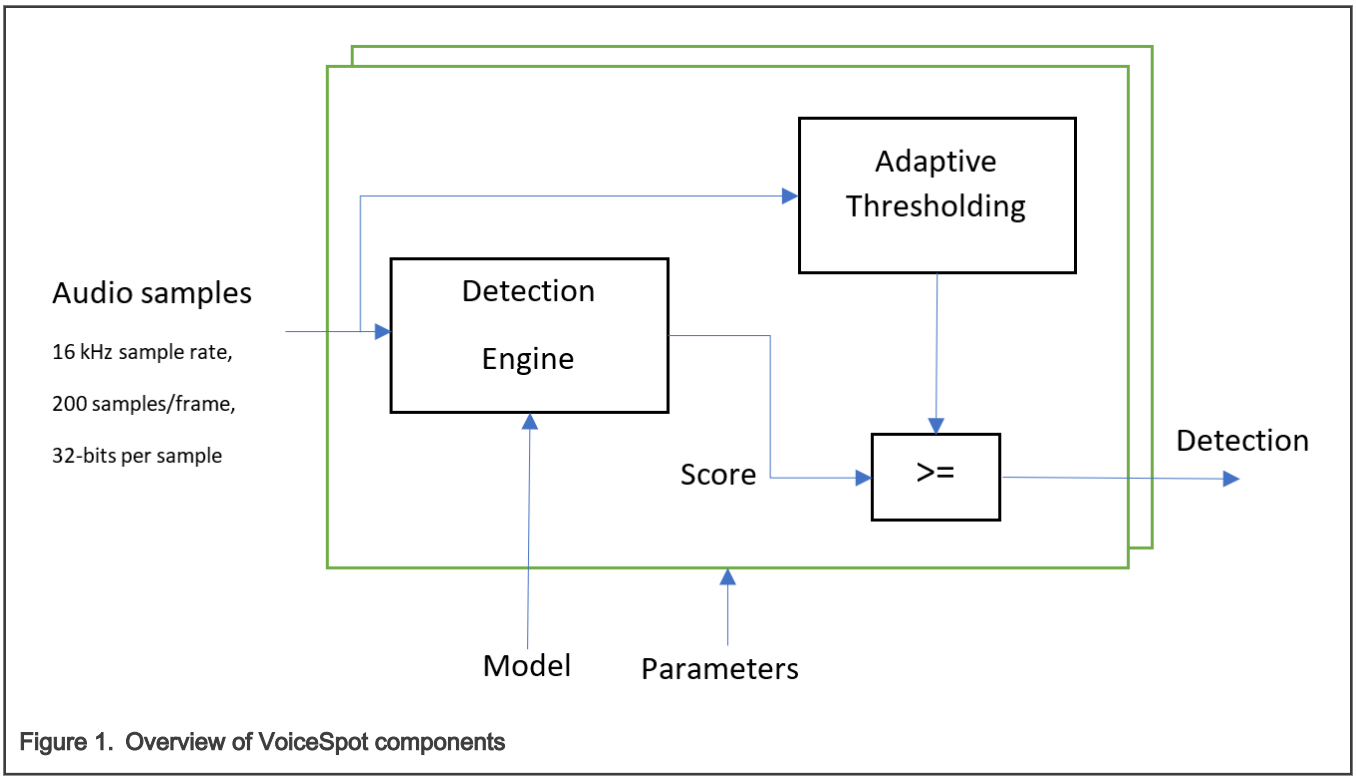

Figure 1. Overview of VoiceSpot components

The audio input provided to VoiceSpot must use a 16-kHz sample rate and it must be delivered in frames, each consisting of 200 samples per frame. The audio must have 32 bits per sample, because it is often possible to get more than 16 effective bits from the microphone. The data types 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 computes 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 is provided. If a matching score is available, it can be compared to the used threshold to decide whether a detection was made. The used threshold may be either a fixed threshold or an automatically computed threshold based on the Adaptive Thresholding mechanism.

Chapter 3
Initializing and running VoiceSpot

This chapter describes 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 describes the steps to initialize VoiceSpot.

3.1.1 Creating the common control structure

To initialize and run one or more instances of VoiceSpot, create the control structure for the instances. This is done as follows:

```
#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 is allocated from within the library. The variable `data_type` is the input data type to use. Here, `RDSP_DATA_TYPE__FLOAT32` is used to indicate a 32-bit floating-point input, whereas `RDSP_DATA_TYPE__INT32` is available for a 32-bit fixed-point input. The call to `rdspVoiceSpot_CreateControl()` initializes the `voicespot_control` control structure and returns an error code given by `voicespot_status`. If the error code is not `RDSP_VOICESPOT_OK`, which has a value of 0, then something likely went wrong. For a list of possible error codes, see List of error codes.

3.1.2 Creating a single instance

An instance is created as follows:

```
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 `enable_highpass_filter` variable configures whether 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. If the audio stream has already been high-pass filtered elsewhere, there is no need to enable this. Note that if the high-pass filter is enabled, the filtering is done in-place on the input audio frame.

The `generate_output` variable configures whether the audio output must be provided from the instance. This is only used to debug potential audio problems and it should 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, check the integrity of the model. This helps to protect against scenarios where the content of the model is corrupted, which can otherwise be difficult to detect, because 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);
```

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 follows:

```
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 perform in-place permutations of the internal model weights to optimize the target platform execution. However, this complicates placing the model in a read-only location (such as flash memory), because changes are made to the model blob during the opening operation.

To support this mode of operation, the `model_blob_is_already_open` argument allows using a modified model blob that is already permuted. To use this functionality, open the model as in the target platform development environment and save the resulting model blob to a read-only memory location (called `model_blob_open`). The modified model blob can now be opened solely from the read-only location as follows:

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

The permuted model blob can completely replace the original model blob in a customer implementation. Note that the integrity of the permuted model blob cannot be checked by "rdspVoiceSpot_CheckModelIntegrity()", because it contains changes. To check the integrity of the permuted model blob, compare it to the result of opening the equivalent unpermuted model blob.

3.1.4 Thresholding scores for event detection

To maximize the performance of the VoiceSpot solution, the used threshold can be adaptively adjusted based on the time since the last detection event (called adaptive sensitivity) and the amount of speech present in the environment (called 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 a wake word increases with the amount of speech in the environment.

Enable these adaptive modes of operation as follows:

```
// 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, the fixed threshold values are used instead. If needed, you may provide your own thresholds during runtime (see Checking for a detection event).

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 is provided as a parameter file together with the model file. The parameters can be applied to an open instance as follows:

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

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

The parameter blob contains parameters for the whole 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 (for example, the Adaptive Threshold mechanism), this is indicated by the error code. If this is the case, the parameter setting is only carried out for the enabled features. It is generally recommended to set the parameters after all features that should be enabled are enabled.

3.1.6 Getting library and model information

Getting information about the library can be done as follows:

```
rdsp_voicespot_version voicespot_version;

rdspVoiceSpot_GetLibVersion(voicespot_control, &voicespot_version);
```

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

The information about the model that is open can be determined as follows:

```
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.

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 is now 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, because the execution time may be unpredictable. Perform all allocation during the initialization.

3.2.1 Processing of input audio

The processing of an audio frame is done as follows:

```
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` or `processing_level = RDSP_PROCESSING_LEVEL__SKIP_OUTPUT` to only update the internal state with no output produced (it gives lower computational complexity).

The pointer to the input audio frame buffer (32 bits per sample) is given by `frame_buffer`. The frame size is usually 200 samples, but it should be determined from the instance (see Getting library and model information), because it may vary. Note that in-place processing may occur on the input frame buffer. Make sure that the content of the buffer is not needed after the function call.

The `num_scores` integer is 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`, the `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 ranging from 0 to 1024 (both included) and represent the probability of a match in the range from 0 % to 100 %.

3.2.2 Checking for a detection event

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

3.2.2.1 Using automatic thresholding

Check whether a detection occurred using Automatic Thresholding as follows:

```
int32_t allow_trigger;

int32_t *event_thresholds = NULL;

int32_t processing_period = 4;

int32_t score_index = rdspVoiceSpot_CheckIfTriggered(voicespot_control, voicespot_handle, scores,
allow_trigger, event_thresholds, processing_period);
```

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

The `scores` input is the array of matching scores computed by the `rdspVoiceSpot_Process()` function call.

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

The `event_thresholds` input array 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 `processing_period` input 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, the direct thresholding of the matching scores available in `scores` is also possible. To do so, compare the elements of `scores` with the selected thresholds. If the manual thresholding is done in this way, there is no need to call `rdspVoiceSpot_CheckIfTriggered()`.

3.2.3 Estimating start and stop time of detected utterances

Whenever a target utterance is detected, it is possible to estimate the start and stop location of the utterance as follows:

```
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 `score_index` input 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 to trade 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. Some estimates may fall outside this range. It is expected that virtually all timing estimates are within +/- 100 ms of the ground truth.

The `start_offset_samples` and `stop_offset_samples` outputs 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 shown in Figure 2.

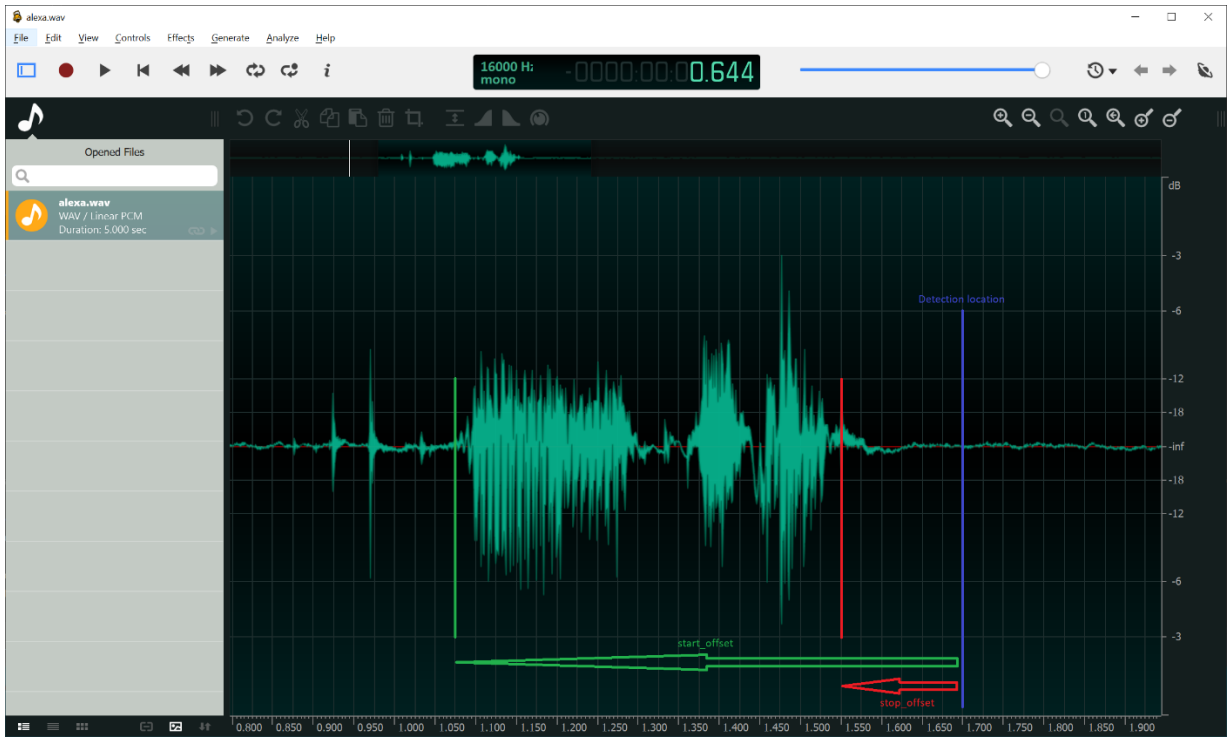

Figure 2. Example showing how the estimated start and stop time is reported as offset values

If only the stop time is required, the following function call should be used, because 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, the following 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);
```

Because 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 that the computation should be split into. As long as `*start_offset < 0`, the computation must then continue 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

Because VoiceSpot typically only outputs a matching score for every 4th input frame, the computational load, as measured 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 must be tested on the used library.

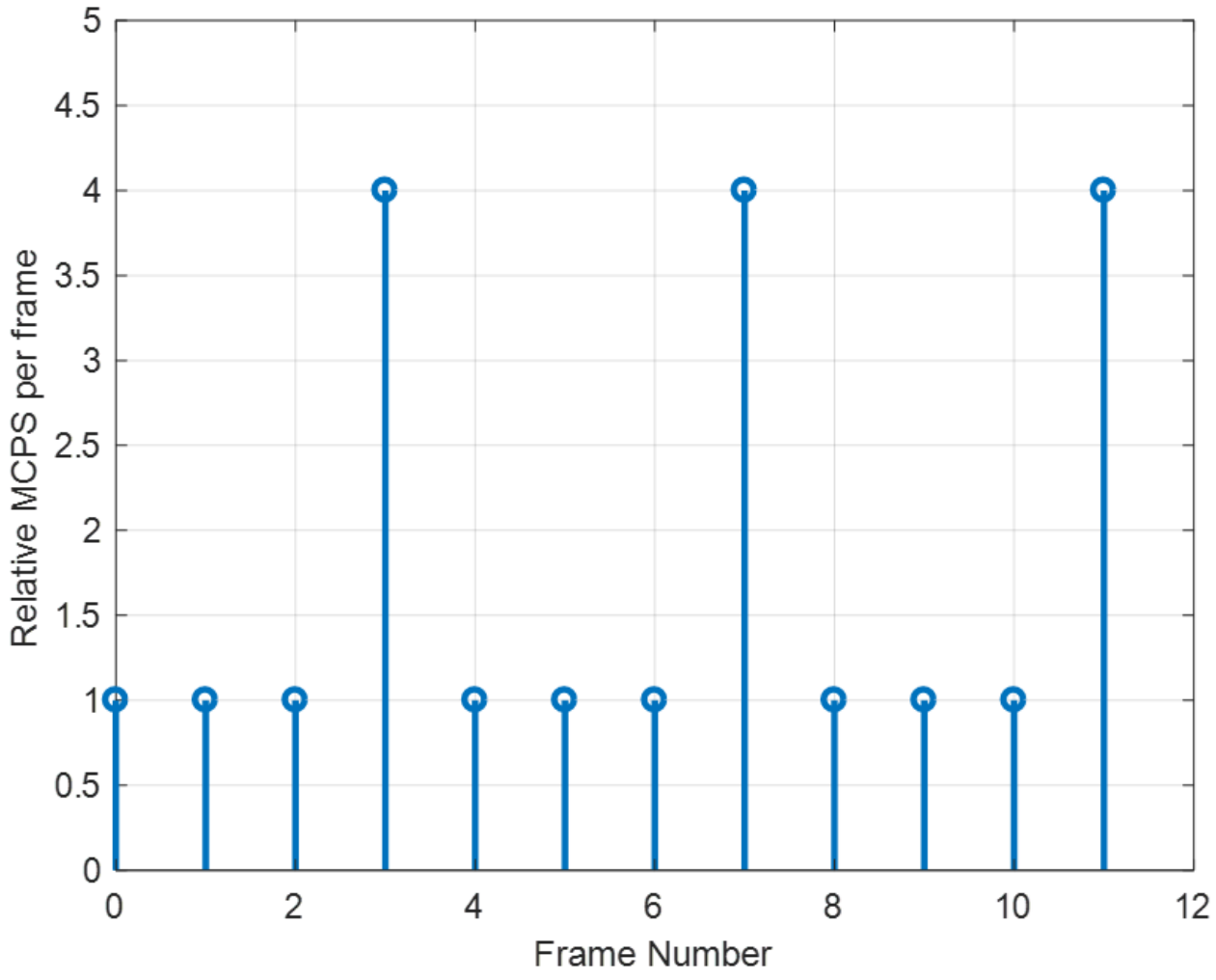

Figure 3. Typical time-varying computational load of `rdspVoiceSpot_Process()`

It is 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 cannot be met, an audio buffer must be inserted into the incoming audio stream to make sure that audio is not being lost while processing. In this manner, the processing unit can be clocked based on the average MCPS, not the peak MCPS.

Do not miss any real-time deadlines when estimating the start time, (see Estimating start and stop time of detected utterances) because this function can take some time to complete, depending on the target platform.

3.2.5 Resetting an instance

To reset an instance back to the state that it was in right after being initialized, call `rdspVoiceSpot_ResetProcessing(voicespot_control, voicespot_handle)`. Note that after resetting an instance, it takes around 1 second to 2 seconds of audio input before the instance is ready to perform the next detection.

To reset only the state of the Adaptive Power State mechanism, call `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 to hold the model in memory is reused internally between the instances.

Multi-threaded execution is currently not officially supported, because it is not actively being tested. Running separate instances in separate threads is expected to work. If you need this feature, 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 and partly deallocated as follows:

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

An instance can be released and fully deallocated as follows:

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

3.5 Releasing the control structure

The control structure can be released as follows:

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

Because the control structure manages all instances, releasing the control structure automatically closes and releases all instances. The easiest way to perform a complete release of VoiceSpot is to call this function.

Chapter 4
Adaptive power state for low-power always-on operation

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 the 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.

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 the speech was detected when the state changes from the low-power mode to the full processing mode, as shown in Figure 4. The audio available to the Detection Engine after going from the low-power mode to the full processing mode is indicated by the Trigger Window.

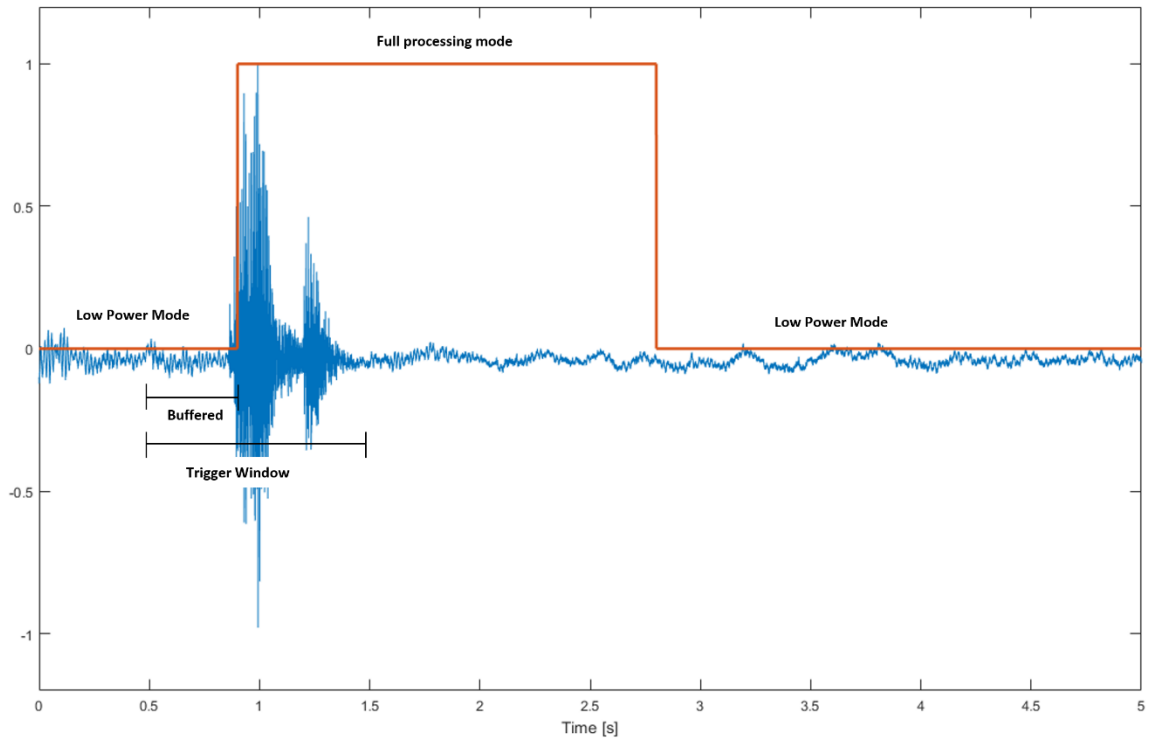

Figure 4. Illustration of adaptive power state change from low-power mode to full processing and back

4.1 Circular buffer structure

To manage the circular buffer, the following structure is defined in the `public/rdsp_voicespot_sleep_cbuffer.h` header file:

```
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, configure the content of this structure using `read_index_samples = 0` and `write_index_samples = 0`.

Whenever audio data is written into the circular buffer, update the `write_index_samples` variable accordingly.

4.2 Frame callback function

For the Detection Engine to be able to go back and process audio in the circular input audio buffer, define a callback function 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 `your_frame_callback_function()` callback function is called, the `voicespot_handle` variable contains 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 the format used by `rdspVoiceSpot_Process()`, allowing the data format used in the circular buffer to differ from the format used by `rdspVoiceSpot_Process()`. If needed, this allows compression to be applied to the audio in the circular buffer.

Register the `your_frame_callback_function` function as the frame callback function as follows:

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

4.3 Status callback function

If you want 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 `voicespot_handle` variable is the handle of the instance from which the callback is made and `rdsp_voicespot_processing_status` is an enumeration that can take on the following 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 wants to go into the full processing mode. This can be used to increase 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 the full processing mode.

A value of `RDSP_IS_PROCESSING_VERY_LOW` indicates that VoiceSpot is in the low-power mode. This can be used to lower the clock frequency of the device.

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

Register the `your_status_callback_function` function as the status callback function as follows:

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

4.4 Enabling adaptive power state

When the `voicespot_buffer_struct` circular buffer structure is initialized, the adaptive power state functionality may be enabled for a given instance as follows:

```
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 standalone 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 as the normal mode of operation.

Chapter 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. To reduce the amount of memory needed to run the two different models, the idea is to close and release the wake word model when a wake word is detected. This enables VoiceSpot to only require memory to run a single model at a time, because there are never two models open at the same time. Instead, it uses a sequence of models. Because 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 to not miss any audio. It is convenient and memory saving to utilize a single systemwide 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 standalone wake word model (for example, using the Adaptive Power State mechanism). When a wake word is 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 (for example, 0.5 seconds of audio or whatever length the audio input buffer has). Now go to processing state 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 processing state 1.

Chapter 6
List of error codes

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

Table 1. List of possible error codes

Error code	Value	Meaning
RDSP_VOICESPOT_OK	0	No errors detected
RDSP_VOICESPOT_MALLOC_FAIL	-1	Memory allocation failed
RDSP_VOICESPOT_INVALID_POINTER	-2	Invalid pointer detected
RDSP_VOICESPOT_UNSUPPORTED_DATA_TYPE	-3	Unsupported data type requested
RDSP_VOICESPOT_INVALID_HANDLE	-4	Invalid handle detected
RDSP_VOICESPOT_NO_FREE_HANDLE	-5	There are no free handles available
RDSP_VOICESPOT_OPEN_FAILED	-6	Opening of the model failed
RDSP_VOICESPOT_NOT_OPEN	-7	Model instance is not open
RDSP_VOICESPOT_NO_TRIGGER_FOUND	-8	No valid trigger found for timing estimation
RDSP_VOICESPOT_UNABLE_TO_CLOSE_MASTER	-9	Master cannot be closed before all slaves have been closed
RDSP_VOICESPOT_UNABLE_TO_OPEN_AS_SLAVE	-10	Slave model is not compatible with master
RDSP_VOICESPOT_BLOB_NOT_VECTOR_ALIGNED	-11	The model blob is not aligned to the vector boundaries (typically 16 bytes)
RDSP_VOICESPOT_LICENSE_EXPIRED	-12	Software license has expired (renew it)
RDSP_VOICESPOT_INVALID_ID	-13	Invalid ID was used
RDSP_VOICESPOT_PARAMETER_NOT_AVAILABLE	-14	Parameter is not available in the current state

Table continues on the next page...

Table 1. List of possible error codes (continued)

Error code	Value	Meaning
RDSP_VOICESPOT_VALID_THRESHOLD_NOT_AVAILABLE	-0x100	No valid threshold value is available for event detection
RDSP_VOICESPOT_BUFFER_UNDERFLOW	-0x101	Input buffer experienced underflow

Chapter 7
Contact

https://www.nxp.com/design/software/embedded-software/voicespot-wake-word-engine:VOICESPOT

Email: voice@nxp.com

Chapter 8
Revision history

Table 2. Revision history

Revision number	Date	Substantive changes
0	06 April 2022	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX,EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE, VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. M, M Mobileye and other Mobileye trademarks or logos appearing herein are trademarks of Mobileye Vision Technologies Ltd. in the United States, the EU and/or other jurisdictions.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 06 April 2022
Document identifier: VSAPIDUG