UM2184 User manual

Sound detector library software expansion for STM32Cube

Introduction

This document reviews the software interface and requirements for the sound detector library. It describes how to integrate the sound detector module into a main program such as the audio STM32Cube expansion software, and provides a basic understanding of the underlying algorithm.

The sound detector library is part of X-CUBE-AUDIO firmware package.

Contents

1 **Module overview** **5**

1.1 Algorithm functionality 5

1.2 Module configuration 5

1.3 Resources summary 6

2 **Module interfaces** **7**

2.1 APIs 7

2.1.1 sdr_reset function 7

2.1.2 sdr_setParam function 7

2.1.3 sdr_getParam function 8

2.1.4 sdr_setConfig function 8

2.1.5 sdr_getConfig function 9

2.1.6 sdr_process function 9

2.2 External definitions and types 9

2.2.1 Input and output buffers 9

2.2.2 Returned error values 10

2.3 Static parameters structure 11

2.4 Dynamic parameters structure 11

3 **Algorithm description** **13**

3.1 Processing steps 13

3.2 Data formats 13

4 **System requirement and hardware setup** **14**

4.1 Recommendations for optimal setup 14

4.1.1 Module integration example 14

4.1.2 Module APIs calls 15

5 **How to tune and run the application** **17**

6 **Revision history** **19**

List of tables

Table 1. Summary of resources 6
Table 2. Parameters for function sdr_reset 7
Table 3. Parameters for function sdr_setParam 8
Table 4. Parameters for function sdr_getParam 8
Table 5. Parameters for function sdr_setConfig 8
Table 6. Parameters for function sdr_getConfig 9
Table 7. Parameters for function sdr_process 9
Table 8. Input and output buffers 10
Table 9. Error values 10
Table 10. Static parameters 11
Table 11. Dynamic parameters. 12
Table 12. Document revision history 19

List of figures

Figure 1. SDR block diagram 13
Figure 2. SDR in the audio processing chain 14
Figure 3. Sequence of APIs 16
Figure 4. SDR excessive sensitivity 17
Figure 5. SDR poor sensitivity 17
Figure 6. SDR noise_pwr_min_dB too high 18
Figure 7. SDR noise_pwr_min_dB too low 18
Figure 8. SDR hangover_nb_frame too low 18
Figure 9. SDR hangover_nb_frame too high 18

1 Module overview

1.1 Algorithm functionality

The SoundDetectoR (SDR) module is in charge of detecting an audio signal for its use as a trigger for signal processing (such as speech recognition). It is based on a noise estimator, on the incoming signal energy measurement, and on two configurable thresholds. Based on these metrics, the decision - ACTIVE (presence of signal) or INACTIVE (presence of noise or low-level signal) - is taken.

The current implementation is using 32-bit resolution for all computations, and can be used with 16- or 24- bit input/output format.

1.2 Module configuration

The SDR module supports mono 16- or 24- bit I/O data, and is not memory limited to a maximum input frame size. Nevertheless, the estimators work at best with input frame sizes in the [10 ms ; 30 ms] range. This corresponds to a range of 160 samples to 480 samples at 16 kHz sampling rate.

Several versions of the module are available depending on the I/O format, on the core, and on the used tool chain:

- *SDR_CM4_IAR.a / SDR_CM4_GCC.a / SDR_CM4_Keil.lib*:
  16-bit input/output buffers. It runs on any STM32 microcontroller featuring an Arm® core with Cortex®-M4 instruction set
- *SDR_32b_CM4_IAR.a / SDR_CM4_GCC.a / SDR_CM4_Keil*:
  24-bit input/output buffers. It runs on any STM32 microcontroller featuring an Arm® core with Cortex®-M4 instruction set
- *SDR_CM7_IAR.a / SDR_CM7_GCC.a / SDR_CM7_Keil.lib*:
  16-bit input/output buffers. It runs on any STM32 microcontroller featuring an Arm® core with Cortex®-M7 instruction set
- *SDR_32b_CM7_IAR.a / SDR_CM7_GCC.a / SDR_CM7_Keil*:
  24-bit input/output buffers. It runs on any STM32 microcontroller featuring an Arm® core with Cortex®-M7 instruction set

1.3 Resources summary

Table 1 lists the requirements for memories and frequency. The footprints are measured on board, using IAR Embedded Workbench® for Arm® v7.40 (IAR Embedded Workbench® common components v7.2).

Table 1. Summary of resources

I/O	Cortex® core	Flash code (.text)	Flash data (.rodata)	Stack	Persistent RAM	Scratch RAM(1)	Frequency (MHz)
16 bits	M4	1060 Bytes	8	100 Bytes	112 Bytes	4 Bytes	0.05
	M7(2)	1228 Bytes					0.03
24 bits	M4	1270 Bytes					0.07
	M7(2)	1470 Bytes					0.05

1. Scratch RAM is the memory that can be shared with other processes running on the same priority level. This memory is not used from one frame to another by SDR routines.
2. Footprints on STM32F7 (based on M7 core) are measured on boards with stack and heap sections located in DTCM memory, with a 10 ms framing.

2 Module interfaces

Two files are needed to integrate the SDR module: *SDR_xxx_CMy_zzz.a/.lib* library and the *sdr_glo.h* header file, which contains all definitions and structures to be exported to the software integration framework.

Note also that *audio_fw_glo.h* file is a generic header file common to all audio modules and must be included in the audio framework.

2.1 APIs

Six generic functions have a software interface to the main program:

- sdr_reset function
- sdr_setParam function
- sdr_getParam function
- sdr_setConfig function
- sdr_getConfig function
- sdr_process function

Each of these functions is described in the following sections.

2.1.1 sdr_reset function

This procedure initializes the persistent memory of the sound detector module, and initializes static parameters with default values.

```
int32_t sdr_reset(void *persistent_mem_ptr, void *scratch_mem_ptr);
```

Table 2. Parameters for function sdr_reset

I/O	Name	Type	Description
Input	*persistent_mem_ptr*	*void **	Pointer to internal persistent memory
Input	*scratch_mem_ptr*	*void **	Pointer to internal scratch memory
Returned value	-	*int32_t*	Error value

This routine must be called at least once at initialization time, when the real time processing has not yet started.

2.1.2 sdr_setParam function

This procedure writes the module static parameters from the main framework to the module internal memory. It can be called after reset routine and before real time processing starts. It handles static parameters (i.e. the parameter values cannot be changed during the module processing).

```
int32_t sdr_setParam(sdr_static_param_t *input_static_param_ptr, void
*persistent_mem_ptr);
```

Table 3. Parameters for function sdr_setParam

I/O	Name	Type	Description
Input	*input_static_param_ptr*	*sdr_static_param_t**	Pointer to static parameters structure
Input	*persistent_mem_ptr*	*void **	Pointer to internal persistent memory
Returned value	-	*int32_t*	Error value

2.1.3 sdr_getParam function

This procedure gets the module static parameters from the module's internal memory to main framework.

It can be called after reset routine and before real time processing starts. It handles static parameters (i.e. the parameter values that cannot be changed during module processing).

```
int32_t sdr_getParam(sdr_static_param_t *input_static_param_ptr, void
*persistent_mem_ptr);
```

Table 4. Parameters for function sdr_getParam

I/O	Name	Type	Description
Input	*input_static_param_ptr*	*sdr_static_param_t**	Pointer to static parameters structure
Input	*persistent_mem_ptr*	*void **	Pointer to internal persistent memory
Returned value	-	*int32_t*	Error value

2.1.4 sdr_setConfig function

This procedure sets module dynamic parameters from main framework to module internal memory.

It can be called at any time during processing.

```
int32_t sdr_setConfig(sdr_dynamic_param_t *input_dynamic_param_ptr, void
*persistent_mem_ptr);
```

Table 5. Parameters for function sdr_setConfig

I/O	Name	Type	Description
Input	*input_dynamic_param_ptr*	*sdr_dynamic_param_t**	Pointer to dynamic parameters structure
Input	*persistent_mem_ptr*	*void **	Pointer to internal persistent memory
Returned value	-	*int32_t*	Error value

2.1.5 sdr_getConfig function

This procedure gets the module dynamic parameters from internal persistent memory to the main framework.

It can be called at any time during processing.

```
int32_t sdr_getConfig(sdr_dynamic_param_t *input_dynamic_param_ptr, void
*persistent_mem_ptr);
```

Table 6. Parameters for function sdr_getConfig

I/O	Name	Type	Description
Input	*input_dynamic_param_ptr*	*sdr_dynamic_param_t**	Pointer to dynamic parameters structure
Input	*persistent_mem_ptr*	*void **	Pointer to internal persistent memory
Returned value	-	*int32_t*	Error value

2.1.6 sdr_process function

This procedure is the module main processing routine.

It should be called at any time, to process each frame.

```
int32_t sdr_process(buffer_t *input_buffer, buffer_t *output_buffer, void
*persistent_mem_ptr);
```

Table 7. Parameters for function sdr_process

I/O	Name	Type	Description
Input	*input_buffer*	*buffer_t**	Pointer to input buffer structure
Output	*output_buffer*	*buffer_t*	Pointer to output buffer structure
Input	*persistent_mem_ptr*	*void **	Pointer to internal persistent memory
Output	-	*int32_t*	Error value

This routine can run in place, meaning that the same buffer can be used for input and output at the same time.

2.2 External definitions and types

2.2.1 Input and output buffers

The SDR library uses extended I/O buffers, which contain, in addition to the samples, some useful information on the stream, such as the number of channels, the number of bytes per sample, and the interleaving mode.

An I/O buffer structure type, like the one described below, must be respected each time, before calling the processing routine; otherwise, errors are returned:

```
typedef struct {
   int32_t    nb_channels;
   int32_t    nb_bytes_per_Sample;
   void       *data_ptr;
   int32_t    buffer_size;
   int32_t    mode;
} buffer_t;
```

Table 8. Input and output buffers

Name	Type	Description
nb_channels	*int32_t*	Number of channels in data (1 for mono). SDR only supports mono channel.
nb_bytes_per_Sample	*int32_t*	Dynamic of data (2 for 16 bits; 3 for 24 bits). SDR supports audio samples in 16- bit and 24- bit format.
data_ptr	*void **	Pointer to data buffer (must be allocated by the main framework)
buffer_size	*int32_t*	Number of samples per channel in the data buffer
mode	*int32_t*	In case of stereo stream, left and right channels can be interleaved: 0 = not interleaved, 1 = interleaved.

2.2.2 Returned error values

Table 9 lists possible returned error values:

Table 9. Error values

Definition	Value	Description
SDR_ERROR_NONE	0	No error.
SDR_UNSUPPORTED_INTERLEAVING_MODE	-1	Input data is not interleaved.
SDR_UNSUPPORTED_NUMBER_OF_CHANNELS	-2	Input data is not mono.
SDR_UNSUPPORTED_NB OF BYTEPERSAMPLES	-3	Input data is not 16-bit or 24-bit sample format.
SDR_BAD_LEARNING_FRAME_NB	-4	`learning_frame_nb` is not in the [2 : 20] range.
SDR_BAD_THR_VALUE	-5	Thresholds should be in the [1 : 40] range.
SDR_BAD_NOISE_PWR_MIN	-6	`noise_pwr_min_dB` is not in the [-87 : 0] range.
SDR_BAD_HW	-7	Unsupported HW for the library.
SDR_LOOPSIZE_BUFFERSIZE_DIFFERENT	-8	`buffer_size` parameter is not identical to the input buffer size to process.

2.3 Static parameters structure

The SDR initial parameters are set using the corresponding static parameter structure before calling the `sdr_setParam()` function.

```
struct sdr_static_param {
   int32_t sampling_rate;
   int32_t buffer_size;
   int32_t learning_frame_nb;
}
```

Table 10. Static parameters

Name	Type	Description
sampling_rate	int32_t	Input buffer sampling rate in Hz. All standard audio sampling rates are supported. Default value is 16 kHz. This value is used in the calculation of some constant times.
buffer_size	int32_t	Input buffer size in samples. This value is used in the calculation of some constant times.
learning_frame_nb	int32_t	Number of frames on which the SDR initializes the noise estimator. Initialization happens once after the `sdr_reset()` function and during the call to the `sdr_process()` function.

2.4 Dynamic parameters structure

It is possible to change the SDR configuration by setting new values in the dynamic parameter structure before calling the `sdr_setConfig()` function.

The SDR library proposes input and output parameters in its dynamic parameters structure.

For input parameters, it is possible to set or change the SDR configuration by setting the dynamic parameter structure before calling the `sdr_setConfig()` function.

For output parameters, the `sdr_getConfig()` function must be called before accessing to the updated output dynamic parameters.

The input and output parameters, as per the structure presented below, are described in *Table 11*.

```
struct sdr_dynamic_param {
   int32_t enable;               /* input variable  */
   int32_t ratio_thr1_dB;        /* input variable  */
   int32_t ratio_thr2_dB;        /* input variable  */
   int32_t noise_pwr_min_dB;     /* input variable  */
   int32_t hangover_nb_frame;    /* input variable  */
   int32_t output_state;         /* output variable */
   int32_t output_nrj;           /* output variable */
   int32_t output_noise_pwr;     /* output variable */
   int32_t output_thr1;          /* output variable */
   int32_t output_thr2;          /* output variable */
}
```

Table 11. Dynamic parameters

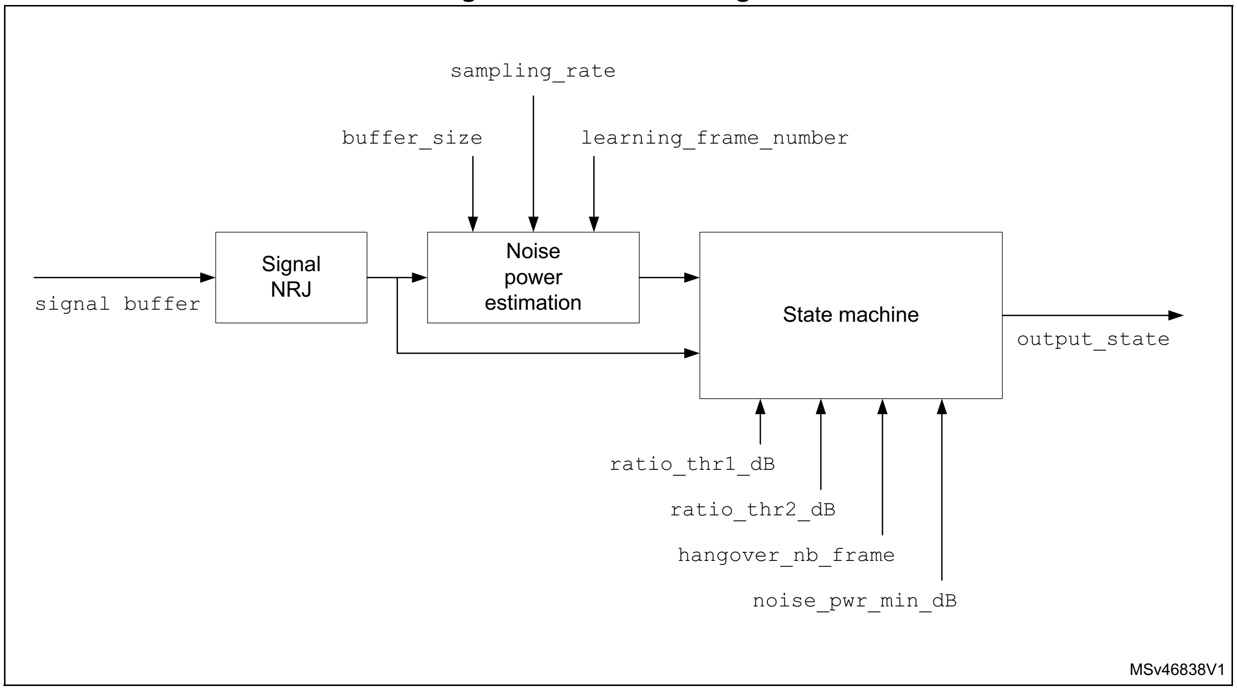
Name	Type	Description
enable	int32_t	1: enable the processing of the SDR. 0: disable the processing of the SDR. SDR output state is not reliable.
ratio_thr1_dB	int32_t	Threshold (in dB) above which SDR state goes from SDR_STATE_INACTIVE to SDR_STATE_ACTIVE. This threshold is added to the noise power estimation and compared to signal energy. Default value is 9 dB. It must be in the [1 : 40] range.
ratio_thr2_dB	int32_t	Threshold (in dB) below which SDR state goes from SDR_STATE_ACTIVE to SDR_STATE_INACTIVE. This threshold is added to the noise power estimation and compared to signal energy. Default value is 7 dB. It must be in the [1 : 40] range.
noise_pwr_min_dB	int32_t	Minimum noise power (in dBFS) below which noise power estimation cannot run. Default value is -60 dBFS. It must be in the [-87 : 0] range.
hangover_nb_frame	int32_t	Number of frames of hangover for state going from SDR_STATE_ACTIVE to SDR_STATE_INACTIVE. Default value is 4. It must be in the [2 : 20] range.
output_state	int32_t	SDR decision. `state = SDR_STATE_INACTIVE` means no speech is detected. `state = SDR_STATE_ACTIVE` means speech is detected.
output_nrj	int32_t	For debug. Current frame energy.
output_noise_pwr	int32_t	For debug. Current frame noise power estimation.
output_thr1	int32_t	For debug. Current frame THR1 estimation (from SDR_STATE_INACTIVE to SDR_STATE_ACTIVE).
output_thr2	int32_t	For debug. Current frame THR2 estimation (from SDR_STATE_ACTIVE to SDR_STATE_INACTIVE).

3 Algorithm description

3.1 Processing steps

The SDR block diagram presented in *Figure 1*.

Figure 1. SDR block diagram



The algorithm starts with the computation of the energy of the current frame continues with the update of the noise power estimation. During the learning phase (corresponding to the `learning_frame_nb` parameter), the algorithm assumes that the incoming signal is noise in order to get a reference noise level. This estimation is then used to compute the two thresholds used for deciding if the current frame is ACTIVE or INACTIVE. Finally the hangover mechanism prevents the SDR state from excessive toggling.

The output variables are then updated in the persistent memory structure, and can be accessed through the call to the `sdr_getConfig()` function and the dynamic parameters structure.

The output buffer contains the copy of the input buffer.

3.2 Data formats

Input of SDR module is expected to be an audio stream, mono, in 16- or 32-bit format. All operations are done with 32 bits resolution. The algorithm can run with almost any buffer size. Nevertheless, like every speech processing algorithm, it will perform at best with input frame sizes ranging from 10 ms to 200 ms. All operations are done with 32-bit resolution. The output format is the same as the format of the input buffer.

4 System requirement and hardware setup

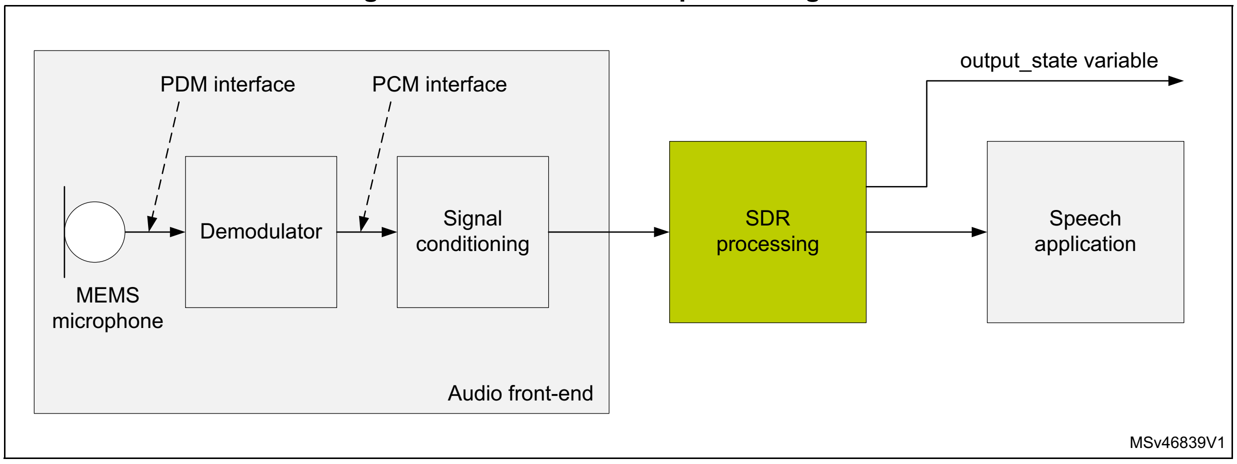
SDR libraries are built to run either on a Cortex®-M4 or on a Cortex®-M7 core. They can be integrated and run on corresponding STM32F4/STM32L4 or STM32F7 family devices. There is no other hardware dependency.

4.1 Recommendations for optimal setup

The SDR module can be executed at any place in an audio processing chain. It generally takes place in an audio front-end together with sound capture mechanisms, and is used as a speech or audio detector to trigger other modules or applications.

Figure 2 shows the optimal setup.

Figure 2. SDR in the audio processing chain



4.1.1 Module integration example

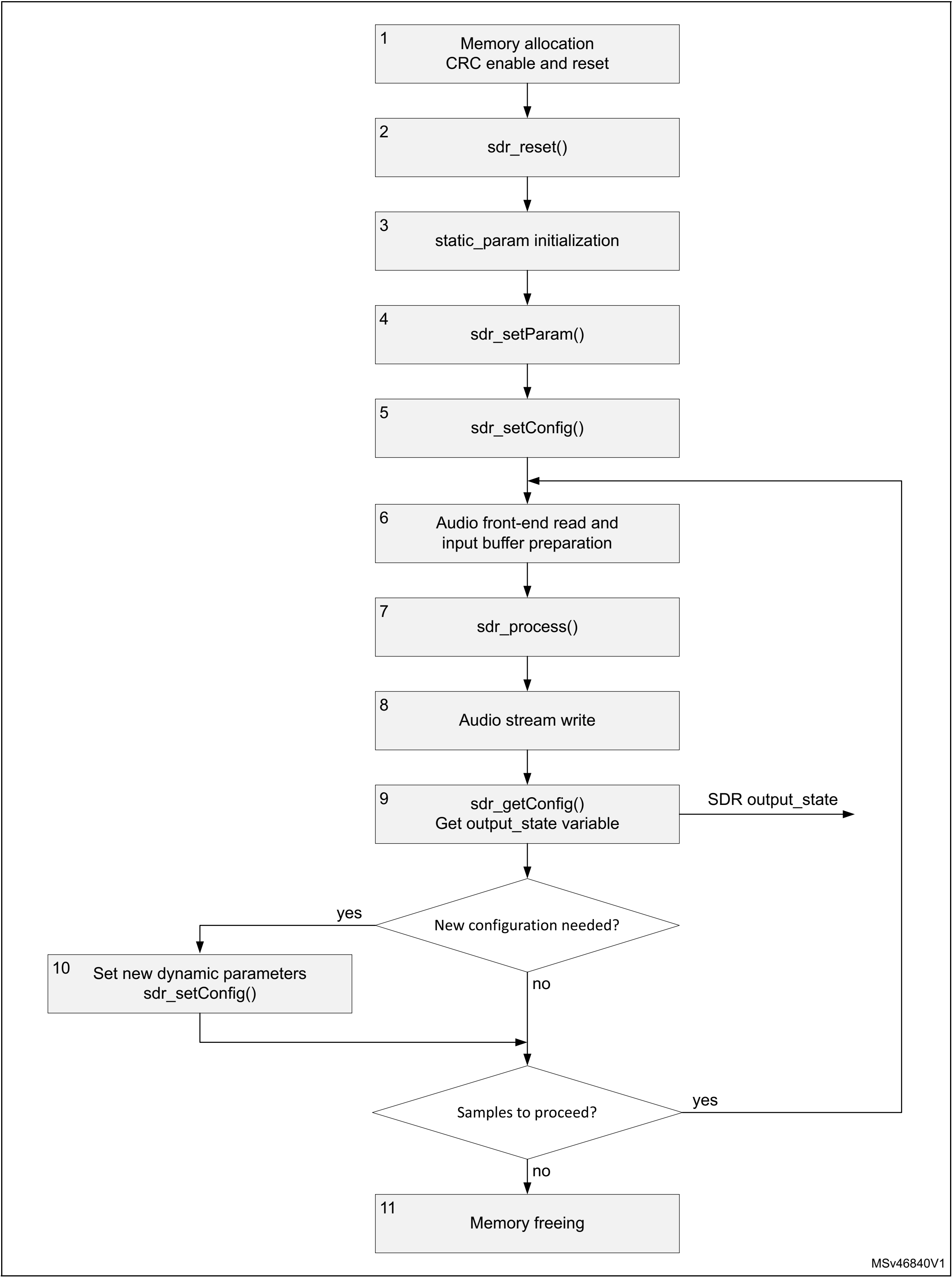
Cube expansion SDR integration examples are provided on STM32F746G-Discovery and STM32F469I-Discovery boards. Refer to the provided integration code for more details.

4.1.2 Module APIs calls

The sequence is shown in *Figure 3 on page 16*. Each step is described in the list below:

1. As explained in *Chapter 2: Module interfaces on page 7*, SDR scratch and persistent memories have to be allocated, as well as input and output buffer according to the structures defined in *Section 2.3* and in *Section 2.4*. The CRC HW block must be enabled to unlock the library.
2. Once the memory is allocated, the call to the `sdr_reset()` function initializes internal variables.
3. The SDR static parameters structure can then be set.
4. Call the `sdr_setParam()` routine to apply static parameters.
5. The dynamic parameters need to be set before going to the processing step. Configure the SDR dynamic parameters as desired and execute the `sdr_setConfig()` function.
6. The audio stream is read from the audio front-end interface and the `input_buffer` structure has to be filled according to the stream characteristics (number of channels, sample rate, interleaving and data pointer). The output buffer structure has to be set as well.
7. Calling the `sdr_process()` function executes the SDR algorithm.
8. The output audio stream can then be written in the proper interface. In the case of the SDR, the output buffer is the same as the input buffer.
9. In order to get the SDR `output_state` variable updated, the `sdr_getConfig()` function has to be called before accessing the dynamic parameter structure.
10. If needed, the user can set new dynamic parameters and call the `sdr_setConfig()` function to update module configuration.
11. If the application is still running and provides new input samples to be processed, execuion must branch to step 6 and continue from step 6 onward; otherwise, the processing loop is over.
12. Once the processing loop is over, the allocated memory must be released.

Figure 3. Sequence of APIs


1 Memory allocation CRC enable and reset
2 sdr_reset()
3 static_param initialization
4 sdr_setParam()
5 sdr_setConfig()
6 Audio front-end read and input buffer preparation
7 sdr_process()
8 Audio stream write
9 sdr_getConfig() Get output_state variable
SDR output_state
New configuration needed?
yes
no
10 Set new dynamic parameters sdr_setConfig()
Samples to proceed?
yes
no
11 Memory freeing
MSv46840V1

5 How to tune and run the application

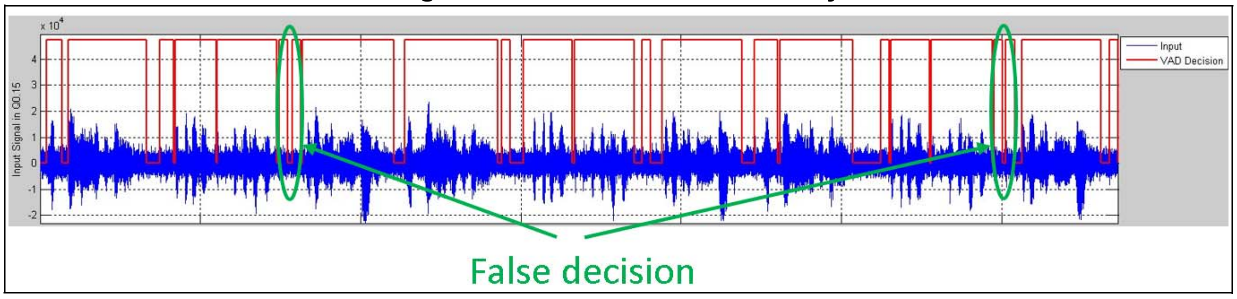
The SDR is designed to process preferably a signal coming from a speech front-end, with an input sampling rate of 16 kHz or 8 kHz. SDR also behaves properly with higher sampling rates.

The learning phase is important. During this phase, the SDR averages the signal during the configured number of frames and uses it as its starting point for noise estimation.

`ratio_thr1_dB` and `ratio_thr2_dB` have to be properly tuned.

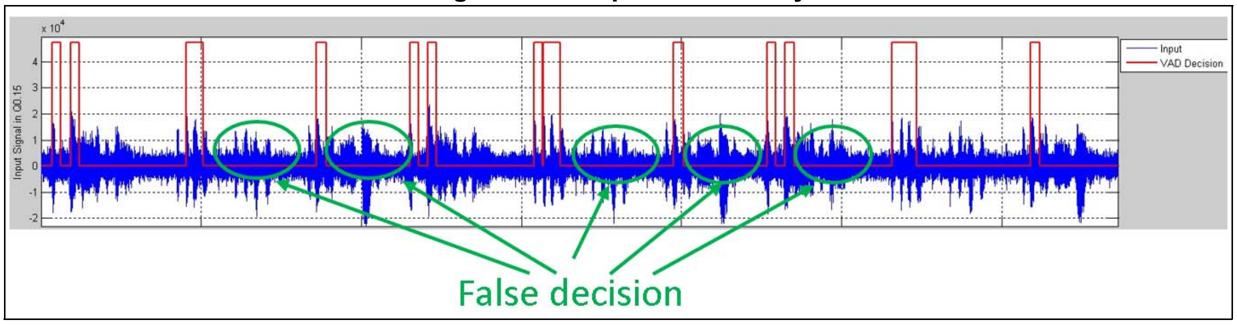
Too low values (e.g. `ratio_thr1_dB = 4`) can lead to an excessive sensitivity of the SDR algorithm to fluctuating noise as illustrated in *Figure 4*.

Figure 4. SDR excessive sensitivity



Conversely, too high values (e.g. `ratio_thr1_dB = 12`) can lead to have the SDR algorithm not sensitive enough. As a result, speech utterance can be detected as noise as shown in *Figure 5*.

Figure 5. SDR poor sensitivity

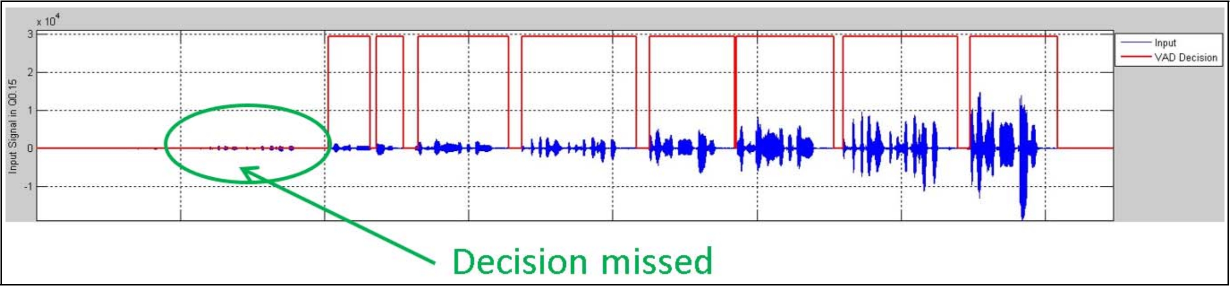


It is strongly recommended to tune `ratio_thr2_dB` so that it is always 2 to 4 dB smaller than `ratio_rhr1_dB`.

The `noise_pwr_min_dB` value must reflect the minimum level under which the input signal is not speech (background level or noise floor).

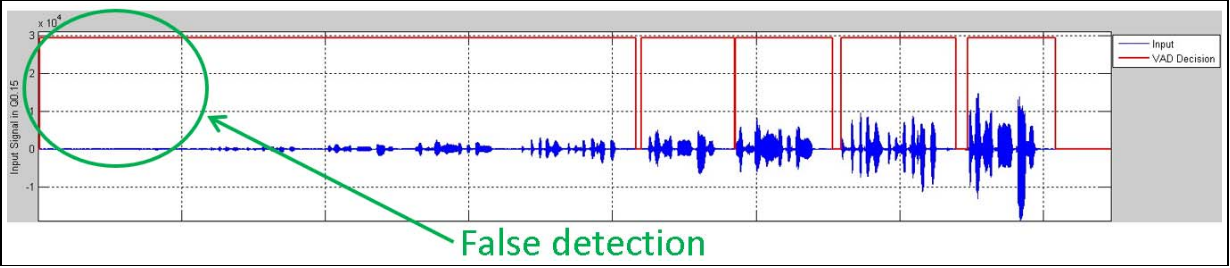
In the next two examples, the input signal has a background noise of about -58 dB. If the `noise_pwr_min_dB` parameter is much higher than the noise level (e.g. `noise_pwr_min_dB = -40dB`), SDR_STATE_INACTIVE state detection can occur even if there is speech present at low level. This is shown in *Figure 6*.

Figure 6. SDR noise_pwr_min_dB too high



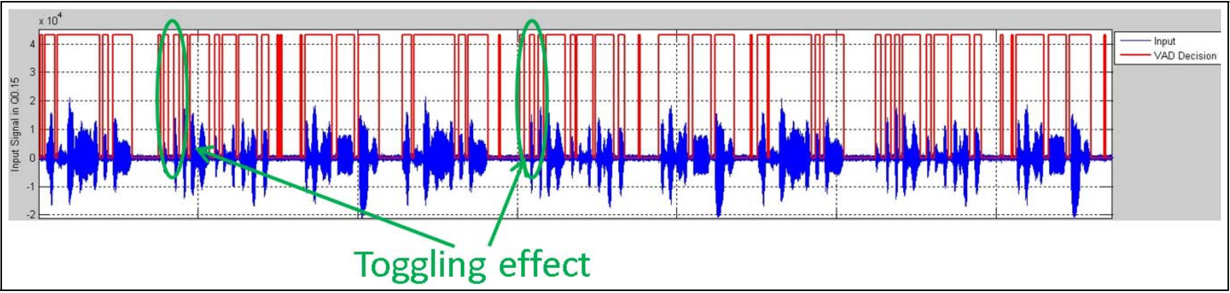
On the opposite, having `noise_pwr_min_dB` being much lower than noise floor (e.g. `noise_pwr_min_dB = -`65 dB) makes the SDR detect speech even if there is only low level fluctuating noise as shown in *Figure 7*.

Figure 7. SDR noise_pwr_min_dB too low



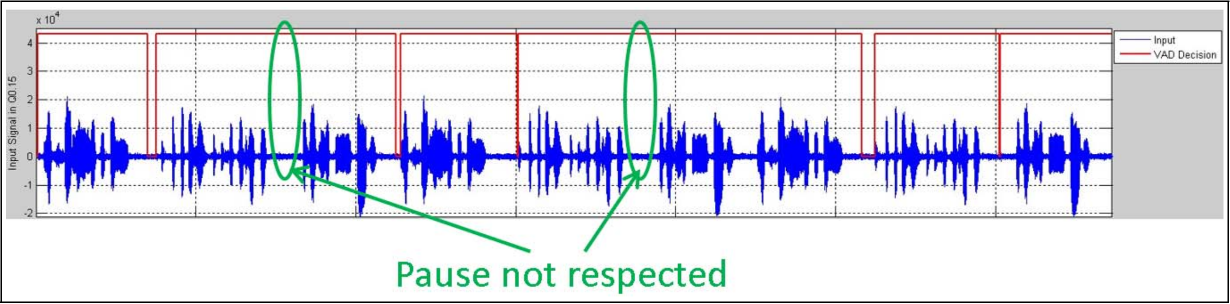
`hangover_nb_frame` must be tuned in order to avoid excessive toggling from SDR_STATE_INACTIVE to SDR_STATE_ACTIVE. Having this parameter tuned too low (ex. = 2) keeps the state toggling too much as shown in *Figure 8*.

Figure 8. SDR hangover_nb_frame too low



On the opposite, If `hangover_nb_frame` is too high (e.g. 50), the state will remain stuck to SDR_STATE_ACTIVE even in the absence of speech. Such occurrences are shown in *Figure 9.*

Figure 9. SDR hangover_nb_frame too high



6 Revision history

Table 12. Document revision history

Date	Revision	Changes
09-Jan-2018	1	Initial release.