Introduction

The Audio Engine post-processing on STM32F4xx - Smart volume control user manual describes the software interface and requirements of the Smart volume control (SVC) module. It details all necessary interface functions, parameters, integration constraints for the programmer to integrate the SVC library into his own software.

It has been designed for the audio application developers who integrate this module into a main program. It provides a rough understanding of the underlying algorithm.

The SVC library is part of the STM32-AUDIO100A firmware package.
# Contents

1 Module overview ...................................................... 6  
   1.1 Algorithm function ........................................... 6  
   1.2 Module configuration .......................................... 6  
   1.3 Resource summary ............................................ 7  

2 Module interfaces .................................................... 8  
   2.1 APIs .......................................................... 8  
      2.1.1 svc_reset function ....................................... 8  
      2.1.2 svc_setParam function .................................... 8  
      2.1.3 svc_getParam function .................................... 9  
      2.1.4 svc_setConfig function ................................... 9  
      2.1.5 svc_getConfig function ................................... 10  
      2.1.6 svc_process function ...................................... 10  
   2.2 External definitions and types ................................. 10  
      2.2.1 Input and output buffers ................................ 10  
      2.2.2 Returned error values .................................... 11  
   2.3 Static parameters structure ................................... 12  
   2.4 Dynamic parameters structure ................................ 12  

3 Algorithm description ................................................ 14  
   3.1 Processing steps .............................................. 14  
   3.2 Data formats .................................................. 14  
   3.3 Performance assessment ....................................... 15  
      3.3.1 Compression gain ......................................... 15  
      3.3.2 Total Harmonic Distortion (THD) ......................... 17  
      3.3.3 Amplitude statistics ..................................... 18  

4 Application description .............................................. 19  
   4.1 System requirements and hardware setup ...................... 19  
   4.2 Recommendations for an optimal setup ....................... 19  
      4.2.1 Memory allocation ....................................... 19  
      4.2.2 Module API calls ....................................... 20  
      4.2.3 Module integration summary ............................. 22
5 How to tune and run the application ........................................ 24

6 Revision history ................................................................. 25
List of tables

Table 1. Resource summary ................................................................. 7
Table 2. svc_reset .............................................................................. 8
Table 3. svc_setParam ........................................................................ 9
Table 4. svc_getParam ......................................................................... 9
Table 5. svc_setConfig ......................................................................... 9
Table 6. svc_getConfig ......................................................................... 10
Table 7. svc_process ........................................................................... 10
Table 8. Input and output buffers ......................................................... 11
Table 9. Returned error values .............................................................. 11
Table 10. Static parameters structure .................................................. 12
Table 11. Dynamic parameters structure .............................................. 13
Table 12. Amplitude statistics ............................................................... 18
Table 13. Document revision history ..................................................... 25
List of figures

Figure 1. Block diagram of SVC module ................................................................. 14
Figure 2. SVC compression curve with 6 dB input volume ..................................... 15
Figure 3. SVC compression curve with 20 dB input volume ................................... 16
Figure 4. SVC compression curve and THD .............................................................. 17
Figure 5. API call procedure .................................................................................. 22
1 Module overview

1.1 Algorithm function

The SVC module allows to digitally modify the volume of the input signal in the range of [-80:+36] dB.

It is based on the dynamic range compressor function, which maps the amplitude range of the audio signal to a smaller range: reducing the signal level of the higher peaks while leaving the quieter parts untreated.

1.2 Module configuration

The SVC module supports mono, stereo and multichannel interleaved 16-bit and 32-bit I/O data.

The SVC library, lib_svc_m4.a, has been optimized for digital volume variations. It supports the dynamic configuration:

- **High quality**
  This configuration computes the gain to apply for each input sample, and gives the best possible results of the module. This has to be compared with the "Standard" version which computes the gain to apply for a block of 16 input samples, considering their mean value. The "Standard" configuration is less reactive and can increase the clipping samples percentage as compared to the "High quality" configuration; but it consumes twice less MHz.
1.3 Resource summary

Table 1 contains the module requirements for the Flash, stack and RAM memories and the frequency (MHz).

The required core frequency (MHz) is estimated using EWARM v6.50 profiler, while in parenthesis values have been measured on real hardware, running on STM32F407IG chipset.

<table>
<thead>
<tr>
<th>I/O data</th>
<th>Version</th>
<th>Flash code (.text)</th>
<th>Flash data (.rodata)</th>
<th>Stack</th>
<th>Static RAM</th>
<th>Dynamic RAM</th>
<th>Required frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits I/O data stereo</td>
<td>SVC high quality</td>
<td>6276 Bytes</td>
<td>8 Bytes</td>
<td>74</td>
<td>1368 Bytes</td>
<td>2880 Bytes</td>
<td>10.7 MHz</td>
</tr>
<tr>
<td>16 bits I/O data multichannel 3.1</td>
<td>SVC high quality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>with 10 ms buffers at 48 kHz measured on hardware</td>
</tr>
<tr>
<td></td>
<td>SVC standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.9 MHz</td>
</tr>
<tr>
<td></td>
<td>SVC standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15.6 MHz</td>
</tr>
<tr>
<td></td>
<td>SVC standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9.9 MHz</td>
</tr>
<tr>
<td>32 bits I/O data stereo</td>
<td>SVC high quality</td>
<td>6152 Bytes</td>
<td>8 Bytes</td>
<td>74</td>
<td>2648 Bytes</td>
<td>4800 Bytes</td>
<td>10.9 MHz</td>
</tr>
<tr>
<td></td>
<td>SVC standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.2 MHz</td>
</tr>
<tr>
<td></td>
<td>SVC standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14.2 MHz</td>
</tr>
<tr>
<td></td>
<td>SVC standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8.3 MHz</td>
</tr>
</tbody>
</table>
## 2 Module interfaces

Two files are needed to integrate the SVC module: `lib_svc_m4.a` library and the `svc_glo.h` header file which contains all definitions and structures to be exported to the software integration framework.

*Note:* The `audio_fw_glo.h` file is a generic header file common to all audio modules; it must be included in the audio framework.

### 2.1 APIs

Six generic functions have a software interface to the main program. They allow the developer to initialize, reset, set and get parameters.

#### 2.1.1 svc_reset function

This procedure initializes the static memory of the SVC module and initializes the static and dynamic parameters with default values.

**API description:**

```c
int32_t svc_reset(void *static_mem_ptr, void *dynamic_mem_ptr);
```

**Table 2. svc_reset**

<table>
<thead>
<tr>
<th>I/O</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>static_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal static memory</td>
</tr>
<tr>
<td>Input</td>
<td>dynamic_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal dynamic memory</td>
</tr>
<tr>
<td>Returned value</td>
<td>-</td>
<td>int32_t</td>
<td>Error value</td>
</tr>
</tbody>
</table>

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

#### 2.1.2 svc_setParam function

This procedure writes module static parameters from the main framework to the module’s internal memory. It can be called after the reset routine and before the start of the real time processing. It handles the static parameters, i.e. the parameters with values which cannot be changed during the module processing (frame after frame).

*Note:* Static parameters cannot be changed dynamically after the start of the module processing, while dynamic parameters can be modified during processing (through `svc_setConfig()` API described below).

**API description:**

```c
int32_t svc_setParam(svc_static_param_t *input_static_param_ptr, void *static_mem_ptr);
```
2.1.3 svc_getParam function

This procedure gets the module static parameters from the module internal memory to the main framework. It can be called after the reset routine and before the start of the real time processing. It handles the static parameters, that is the parameters with values which cannot be changed during the module processing (frame after frame).

API description:

```
int32_t svc_getParam(svc_static_param_t *input_static_param_ptr, void *static_mem_ptr);
```

Table 3. svc_setParam

<table>
<thead>
<tr>
<th>I/O</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>input_static_param_ptr</td>
<td>svc_static_param_t *</td>
<td>Pointer to static parameters structure</td>
</tr>
<tr>
<td>Input</td>
<td>static_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal static memory</td>
</tr>
<tr>
<td>Returned value</td>
<td>–</td>
<td>int32_t</td>
<td>Error value</td>
</tr>
</tbody>
</table>

2.1.4 svc_setConfig function

This procedure sets the module dynamic parameters from the main framework to the module internal memory. It can be called at any time during processing (after reset and setParam routines).

API description:

```
int32_t svc_setConfig(svc_dynamic_param_t *input_dynamic_param_ptr, void *static_mem_ptr);
```

Table 4. svc_getParam

<table>
<thead>
<tr>
<th>I/O</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>input_static_param_ptr</td>
<td>svc_static_param_t *</td>
<td>Pointer to static parameters structure</td>
</tr>
<tr>
<td>Input</td>
<td>static_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal static memory</td>
</tr>
<tr>
<td>Returned value</td>
<td>–</td>
<td>int32_t</td>
<td>Error value</td>
</tr>
</tbody>
</table>

Table 5. svc_setConfig

<table>
<thead>
<tr>
<th>I/O</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>input_dynamic_param_ptr</td>
<td>svc_dynamic_param_t *</td>
<td>Pointer to dynamic parameters structure</td>
</tr>
<tr>
<td>Input</td>
<td>static_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal static memory</td>
</tr>
<tr>
<td>Returned value</td>
<td>–</td>
<td>int32_t</td>
<td>Error value</td>
</tr>
</tbody>
</table>
2.1.5 svc_getConfig function

This procedure gets the module dynamic parameters from the internal static memory to the main framework. It can be called at any time during the module processing (after reset and setParam routines).

API description:
```c
int32_t svc_getConfig(svc_dynamic_param_t *input_dynamic_param_ptr, void *static_mem_ptr);
```

<table>
<thead>
<tr>
<th>I/O</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>input_dynamic_param_ptr</td>
<td>svc_dynamic_param_t *</td>
<td>Pointer to dynamic parameters structure</td>
</tr>
<tr>
<td>Input</td>
<td>static_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal static memory</td>
</tr>
<tr>
<td>Returned value</td>
<td>–</td>
<td>int32_t</td>
<td>Error value</td>
</tr>
</tbody>
</table>

2.1.6 svc_process function

This procedure is the main processing routine of the module. It should be called at any time, to process each frame.

API description:
```c
int32_t svc_process(buffer_t *input_buffer, buffer_t *output_buffer, void *static_mem_ptr);
```

<table>
<thead>
<tr>
<th>I/O</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>input_buffer</td>
<td>buffer_t *</td>
<td>Pointer to input buffer structure</td>
</tr>
<tr>
<td>Output</td>
<td>output_buffer</td>
<td>buffer_t *</td>
<td>Pointer to output buffer structure</td>
</tr>
<tr>
<td>Input</td>
<td>static_mem_ptr</td>
<td>void *</td>
<td>Pointer to internal static memory</td>
</tr>
<tr>
<td>Returned value</td>
<td>–</td>
<td>int32_t</td>
<td>Error value</td>
</tr>
</tbody>
</table>

Note: This process routine can run in place; the input_buffer and the output_buffer are identical.

2.2 External definitions and types

For genericity reasons and to facilitate the integration in main frameworks, some types and definitions are adopted.

2.2.1 Input and output buffers

The SVC library is using 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, as described below, must be respected each time before calling the processing routine; else, errors will be 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb_channels</td>
<td>int32_t</td>
<td>Number of channels in data: 1 for mono, 2 for stereo, 4 for 3.1 multichannel,...</td>
</tr>
<tr>
<td>nb_bytes_per_Sample</td>
<td>int32_t</td>
<td>Dynamic of data in number of bytes (16-bit = 2, 24-bit = 3, 32-bit = 4)</td>
</tr>
<tr>
<td>data_ptr</td>
<td>void *</td>
<td>Pointer to data buffer (must be allocated by the main framework)</td>
</tr>
<tr>
<td>buffer_size</td>
<td>int32_t</td>
<td>Number of samples per channel in the data buffer</td>
</tr>
<tr>
<td>mode</td>
<td>int32_t</td>
<td>In case of stereo stream, left and right channels can be interleaved. 0 = not interleaved, 1 = interleaved.</td>
</tr>
</tbody>
</table>

2.2.2 Returned error values

Table 9 describes possible returned error values.

Table 9. Returned error values

<table>
<thead>
<tr>
<th>Definition</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVC_ERROR_NONE</td>
<td>0</td>
<td>No error</td>
</tr>
<tr>
<td>SVC_UNSUPPORTED_DELAY_LENGTH</td>
<td>-1</td>
<td>Wrong delay length setting</td>
</tr>
<tr>
<td>SVC_UNSUPPORTED_VOLUME</td>
<td>-2</td>
<td>Wrong volume setting</td>
</tr>
<tr>
<td>SVC_UNSUPPORTED_MUTE_MODE</td>
<td>-3</td>
<td>Wrong mute setting</td>
</tr>
<tr>
<td>SVC_UNSUPPORTED_QUALITY_MODE</td>
<td>-4</td>
<td>Wrong quality setting</td>
</tr>
<tr>
<td>SVC_UNSUPPORTED_JOINT_STEREO_MODE</td>
<td>-5</td>
<td>Wrong joint stereo setting</td>
</tr>
<tr>
<td>SVC_UNSUPPORTED_NUMBER_OF_BYTEPERSAMPLE</td>
<td>-6</td>
<td>Input data is not 16-bit sample format</td>
</tr>
<tr>
<td>SVC_BAD_HW</td>
<td>-7</td>
<td>May happen if the library is not used with the right hardware</td>
</tr>
</tbody>
</table>
2.3 Static parameters structure

There are some static parameter to be set before calling the process routine.

```c
struct svc_static_param {
    int16_t   delay_len;
    int16_t joint_stereo;
};
typedef struct svc_static_param svc_static_param_t;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay_len</td>
<td>int16_t</td>
<td>Delay introduced in number of samples in the range [0:320]</td>
</tr>
<tr>
<td>joint_stereo</td>
<td>int16_t</td>
<td>1 to enable joint_stereo, 0 to disable it</td>
</tr>
</tbody>
</table>

- "Delay_len" parameter represents the delay applied between the gain computed from an input sample and this input sample.
  For tuning, it is advised to smoothen the signal peaks and to use the following value:
  \[
  \text{delay\_len} = \text{attack\_time} \times F_s \times 2/3
  \]

- "Joint_stereo" parameter allows to apply the same amount of gain reduction to all the channels: left, right, and possibly other channels in case of multichannel, in order to preserve the stereo image of the input signal.
  As it almost divides the MHz consumption of the algorithm by 2, it is advised to set it.

The default settings are:
- Delay_len is set to 100 samples
- Joint_stereo is enabled.

2.4 Dynamic parameters structure

There are two dynamic parameters to be used.

```c
struct svc_dynamic_param {
    int16_t   target_volume_dB;
    int16_t   mute;
    int16_t   enable_compr;
    int32_t   attack_time;
    int32_t   release_time;
    int16_t   quality;
};
typedef struct svc_dynamic_param svc_dynamic_param_t;
```
Table 11. Dynamic parameters structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target_volume_dB</td>
<td>int16_t</td>
<td>Volume dB input value, in 1/2 dB steps</td>
</tr>
<tr>
<td>mute</td>
<td>int16_t</td>
<td>1 to enable mute, 0 to disable it</td>
</tr>
<tr>
<td>enable_compr</td>
<td>int16_t</td>
<td>1 to enable compression, 0 to disable it</td>
</tr>
<tr>
<td>attack_time</td>
<td>int32_t</td>
<td>Attack time coefficient in Q31 format</td>
</tr>
<tr>
<td>release_time</td>
<td>int32_t</td>
<td>Release time coefficient in Q31 format</td>
</tr>
<tr>
<td>quality</td>
<td>int16_t</td>
<td>1 to enable HIGH_Q mode, 0 to enable standard mode</td>
</tr>
</tbody>
</table>

Attack and release times

Conversion formula: Attack_time $\alpha_A$ and release time $\alpha_R$ are the coefficients given as input of the algorithm, in Q31 format, corresponding to the following formula:

$$\alpha = \left( e^{-1}/ (\tau \cdot Fs) \right) \cdot 2^{31}$$

Where:

- $Fs = 48$ kHz and $\tau$ is the attack/release time in seconds.
- Value Range: $\tau_{MIN} = 1$ sample @$Fs = 48$ kHz = 0.02 ms (physical limit)
- $\tau_{MAX}$: there is no maximum limitation, whereas values exceeding tens of seconds do not seem realistic. The implementation limit is $2^{31}$ because the parameter is a 32-bit format.

The conversion between $\tau$ and $\alpha$ must be done outside the algorithm, by the framework. $\alpha_A$ and $\alpha_R$ must be given as "attack_time" and "release_time" inputs to the algorithm.

*Note:* Considering $\tau$ range described above, $\alpha_A$ and $\alpha_R$ should be in the following range: [790015084, 2147483647].
3 Algorithm description

3.1 Processing steps

Figure 1. Block diagram of SVC module

Abs block: computes the absolute value of the input signal.
Level detector block: smoothenes the signal depending on attack and release times.
dB block: converts the signal to dB format.
Gain computer block: computes in real time the optimized compression curve depending on the targeted volume.
Lin block: converts the signal back to the linear format.

3.2 Data formats

The SVC module supports fixed point mono, stereo and multichannel interleaved 16-bit and 32-bit input data.

In case of multichannel 3.1, the interleaved input data is expected to be within the following scheme: L1/R1/C1/LFE1// L2/R2/C2/LFE2// etc.

The module delivers output data following the same interleaved pattern and the 16-bit or 32-bit resolution as the input data.
3.3 Performance assessment

3.3.1 Compression gain

The compression gain curve design, which is calculated from the input volume, is an essential part of the SVC module. Therefore, its real time computation must be very accurate.

Find below the compression gain curve obtained with the Matlab model and with the SVC module for 6 dB (see Figure 2) and 20 dB (see Figure 3) input volumes. The x-axis represents the dB volume of the input sample, and the y-axis represents the dB volume applied to the output sample and computed by the algorithm. You can notice two parts in the red curve: a linear part for low volume inputs, and a compressed part for higher volume inputs which is needed to prevent the output from clipping.

Note: The blue curve noted "Input" represents the compression gain = 1, i.e. when output = input.

Figure 2. SVC compression curve with 6 dB input volume
Figure 3. SVC compression curve with 20 dB input volume
3.3.2 Total Harmonic Distortion (THD)

THD, Total Harmonic Distortion, is a measurement used to characterize the linearity of audio systems.

When the input is a pure sine wave, the measurement is most commonly the ratio of the sum of the powers of all harmonic frequencies to the power of the fundamental frequency:

\[
\text{THD} = \frac{P_2 + P_3 + \ldots + P_{\infty}}{P_1} = \sum_{i=2}^{\infty} \frac{P_i}{P_1}
\]

As compressors are non-linear modules, it is useful to know which distortion the SVC module can add to the signal.

In Figure 4, the blue curve represents the THD value measured for SVC with 36 dB input gain. The THD increases as expected in the non-linearity part of the compression curve, and especially when input > -12 dB. The green curve represents the compression gain of the SVC with 36 dB input gain.

\textit{Note:} The value represented is THD +60 dB, thus a gain of 0 dB on the graph corresponds to -60 dB.

\textbf{Figure 4. SVC compression curve and THD}
3.3.3 Amplitude statistics

Amplitude statistics are important measures to evaluate the audio signal characteristics. In order to assess the performance of the SVC implementation, several measures have been identified: the number of clipped samples, the RMS power and the stereo image of each channel.

*Table 12* is an example of comparison between "Joint stereo" and "No Joint stereo" modes. It also shows the differences between the "high quality" and "standard" configurations for the same input gain = 36 dB.

The results shown in *Table 12* are obtained with the same input file: "miossec" for several input gains: 6, 12, 18, 24, and 36 dB.

You can observe that the RMS of the "Joint stereo" mode is lower than the "No joint stereo" one. The difference depends on the input gain (from 0.1 to 1 dB). Consequently, the clipping percentage is also lower for the "Joint stereo" mode.

Knowing that the input signal has a stereo image of 0.54 dB, the "Joint stereo" allows to preserve the stereo image for all input gains, whereas "No joint stereo" does not.

<table>
<thead>
<tr>
<th>Input file</th>
<th>No joint stereo M4</th>
<th>Joint Stereo M4</th>
<th>DIFF</th>
<th>Stereo Image</th>
<th>Stereo Image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clipped samples %</td>
<td>Total RMS Power</td>
<td>Clipped samples %</td>
<td>Total RMS Power</td>
<td>No Joint</td>
</tr>
<tr>
<td>Miossec_6dB_L</td>
<td>0.00809</td>
<td>-11.01</td>
<td>0.00549</td>
<td>-11.21</td>
<td>0.00256</td>
</tr>
<tr>
<td>Miossec_6dB_R</td>
<td>0.00738</td>
<td>-10.56</td>
<td>0.00573</td>
<td>-10.67</td>
<td>0.00166</td>
</tr>
<tr>
<td>Miossec_12dB_L</td>
<td>0.05446</td>
<td>-7.88</td>
<td>0.03308</td>
<td>-8.46</td>
<td>0.02138</td>
</tr>
<tr>
<td>Miossec_12dB_R</td>
<td>0.05624</td>
<td>-7.6</td>
<td>0.04094</td>
<td>-7.92</td>
<td>0.01731</td>
</tr>
<tr>
<td>Miossec_18dB_L</td>
<td>0.27259</td>
<td>-6.12</td>
<td>0.13296</td>
<td>-6.94</td>
<td>0.13963</td>
</tr>
<tr>
<td>Miossec_18dB_R</td>
<td>0.30844</td>
<td>-5.96</td>
<td>0.19793</td>
<td>-6.4</td>
<td>0.11051</td>
</tr>
<tr>
<td>Miossec_24dB_L</td>
<td>0.79945</td>
<td>-5.11</td>
<td>0.34902</td>
<td>-6.07</td>
<td>0.45043</td>
</tr>
<tr>
<td>Miossec_24dB_R</td>
<td>0.86478</td>
<td>-5.02</td>
<td>0.55031</td>
<td>-5.53</td>
<td>0.31446</td>
</tr>
<tr>
<td>Miossec_30dB_L</td>
<td>1.52968</td>
<td>-4.51</td>
<td>0.65362</td>
<td>-5.53</td>
<td>0.87606</td>
</tr>
<tr>
<td>Miossec_30dB_R</td>
<td>1.61686</td>
<td>-4.46</td>
<td>1.03725</td>
<td>-5.09</td>
<td>0.57961</td>
</tr>
<tr>
<td>Miossec_36dB_L</td>
<td>2.38731</td>
<td>-4.08</td>
<td>1.0495</td>
<td>-5.12</td>
<td>1.33777</td>
</tr>
<tr>
<td>Miossec_36dB_R</td>
<td>2.47107</td>
<td>-4.05</td>
<td>1.6307</td>
<td>-4.6</td>
<td>0.84038</td>
</tr>
<tr>
<td>Miossec_STANDARD_L</td>
<td>6.46295</td>
<td>-3.1</td>
<td>5.16298</td>
<td>-3.5</td>
<td>1.29997</td>
</tr>
<tr>
<td>Miossec_STANDARD_R</td>
<td>6.03980</td>
<td>-3.21</td>
<td>6.56466</td>
<td>-3.07</td>
<td>-0.52485</td>
</tr>
</tbody>
</table>
4 Application description

4.1 System requirements and hardware setup

The SVC library is built to run on a Cortex M4 core without FPU usage. It can be integrated and run on STM32F4 family devices.

4.2 Recommendations for an optimal setup

The SVC library should be executed close to the audio DAC, at the end of the chain, because of its non-linear effect on the signal.

Follow the steps below to integrate the SVC in an audio software framework.

4.2.1 Memory allocation

First of all, all the memory used by the module must be allocated.

The static and dynamic parameter structures are exported in the svc_glo.h file.

Example of allocation:

```c
/* svc static and dynamic parameters structures memory allocation */
svc_static_param_t *static_param_ptr = malloc(sizeof(svc_static_param_t));
svc_dynamic_param_t *dynamic_param_ptr = malloc(sizeof(svc_dynamic_param_t));
```

- `static_param_ptr` pointer is used by svc_setParam() and svc_getParam() routines.
- `dynamic_param_ptr` pointer is used by svc_setConfig() and svc_getConfig() routines.

Next, static and dynamic memory required by the SVC module must be allocated by the framework. Their sizes are exported as a constant in the svc_glo.h header. The memory allocation can be done as written below:

```c
/* SVC memory structure memory allocation */
void *static_mem_ptr = malloc(svc_static_mem_size);
void *dynamic_mem_ptr = malloc(svc_dynamic_mem_size);
```

- `dynamic_mem_ptr` pointer is a parameter of the svc_reset() routine,
- `static_mem_ptr` pointer is a parameter of all SVC exported APIs.

It is then necessary to allocate the memory for input and output audio buffers.
4.2.2 Module API calls

Once the memory has been allocated, the svc_reset() routine must be called to initialize the SVC module static memory.

- The svc_reset() routine should be called each time the audio processing has been stopped and started.
- The svc_getParam() routine can be called while the run time process has not started to extract current svc_mode used, if needed.
- The svc_setParam() routine must be called to configure the module's internal memory with the corresponding delay samples introduced to the processing, as in the example below. The memory used by the module must be allocated.

```c
/* SVC static parameters setting */
svc_static_param.delay_len = 100;
svc_static_param.joint_stereo = 1;
error = svc_setParam(&svc_static_param, svc_static_mem_ptr);
```

Now that the hardware has been configured and the SVC module has been initialized and configured, the run time process can start.

Every dynamic parameter needs to be set by user, and can be modified by user at runtime, through SvcCtrlStruct structure.

Note that, the AudioPlayer_SVC_alpha_compute function makes the conversion, described in the 2.4 Dynamic parameters structure, between attack_time in ms and SVC input coefficient in Q31 format.

Every time the dynamic parameters change, they are taken into account by the module via the svc_setConfig() routine:

```c
/* SVC dynamic parameters setting */
svc_dynamic_param.target_volume_dB = SvcCtrlStruct->target_volume_dB;
svc_dynamic_param.mute = SvcCtrlStruct->mute;
svc_dynamic_param.enable_compr = SvcCtrlStruct->enable_compr;

attack_us = SvcCtrlStruct->attack_time;
svc_dynamic_param.attack_time = AudioPlayer_SVC_alpha_compute(attack_us);
release_us = SvcCtrlStruct->release_time;
svc_dynamic_param.release_time = AudioPlayer_SVC_alpha_compute(release_us);

svc_dynamic_param.quality = SvcCtrlStruct->quality;
error = svc_setConfig(&svc_dynamic_param, svc_static_mem_ptr);
```

At each new frame, the input buffer structure fields must be filled in as in the example below, as well as the data address for the output buffer structure. Then, the svc_process() routine can be called:

```c
/* SVC component processing */
svc_input_buffer_t.data_ptr = input_buffer_ptr;
svc_input_buffer_t.buffer_size = input_buffer_size;
svc_input_buffer_t.mode = INTERLEAVED;
svc_input_buffer_t.nb_bytes_per_Sample = 2;
```
svc_input_buffer_t.nb_channels = 2;
svc_output_buffer_t.data_ptr = output_buffer_ptr;
svc_output_buffer_t.buffer_size = output_buffer_size;
svc_output_buffer_t.mode = INTERLEAVED;
svc_output_buffer_t.nb_bytes_per_Sample = 2;
svc_output_buffer_t.nb_channels = 2;
error = svc_process (&svc_input_buffer_t, &svc_output_buffer_t, svc_static_mem_ptr);
4.2.3 Module integration summary

Figure 5. API call procedure

1. Memory allocation
2. svc_reset() → static_param initialization
3. static_param initialization → svc_setParam()
4. svc_setParam() → audio stream read input_buffer preparation
5. audio stream read input_buffer preparation → svc_setConfig()
6. svc_setConfig() → svc_process()
7. svc_process() → Audio stream write
8. Audio stream write → New config needed?
9. svc_setConfig() → yes
10. Memory freeing → no
1. As explained above, the SVC static and dynamic structures have to be allocated, as well as the input and output buffer, according to the structures defined in Section 2.2.1: Input and output buffers.

2. Once the memory is allocated, the call to the svc_reset() function initializes the internal variables and sets the default configuration to the module.

3. The module static configuration can now be set by initializing the static_param structure.

4. Call the svc_setParam() routine to send static parameters from the audio framework to the module.

5. The audio stream is read from the proper interface and the input_buffer structure has to be filled in according to the stream characteristics (number of channels, sample rate, interleaving and data pointer). The output buffer structure has to be set as well.

6. Get dynamic parameters when they are updated, and call svc_setConfig() routine to send dynamic parameters from the audio framework to the module.

7. Call the main processing routine to apply the effect.

8. The output audio stream can now be written in the proper interface.

9. If needed, the user can set new dynamic parameters and call the svc_setConfig() routine again, to update the module configuration.

10. Once the processing loop is over, the allocated memory has to be freed.
5 How to tune and run the application

Once the module has been integrated into the audio framework to play samples at 48 kHz, just launch the Audio player and choose a .WAV or .MP3 file with a 48 kHz sampling frequency if there is no sampling rate conversion available.

The SVC "target_volume_dB" dynamic parameter represents the volume asked by the user, in half dB steps. The volume change is applied smoothly to avoid audible artifacts.

The SVC "attackTime" dynamic parameter represents the time in ms, to decrease the gain, when the input overshoots the threshold. The lowest the attack time is, the quickest the gain will follow the input signal when the input signal amplitude increases.

The SVC "releaseTime" dynamic parameter represents the time in ms, to bring back the gain to a normal value, once the signal falls below the threshold. The lowest the release time is, the quickest the gain will follow the input signal when the input signal amplitude decreases.

The SVC "enable_compr" dynamic parameter enables the effect when set to 1 or disables it when set to 0.

The SVC "mute" dynamic parameter mutes the output when set to 1 or has no influence on input signal when set to 0. When enabled, it allows to mute the signal smoothly over a frame, avoiding audible artifacts.

The SVC "quality" dynamic parameter, when enabled, allows to compute the gain for each sample. On the contrary, when disabled, the gain is computed with an average over 16 input samples, to reduce MHz consumption.
## Revision history

Table 13. Document revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18-Jul-2013</td>
<td>1</td>
<td>Initial release.</td>
</tr>
<tr>
<td>08-Nov-2013</td>
<td>2</td>
<td>Added 32-bit I/O data and multichannel support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Section 1.2: Module configuration, Section 2.3: Static parameters structure, Section 3.2: Data formats and Section 3.3.1: Compression gain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Table 1: Resource summary.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Figure 2: SVC compression curve with 6 dB input volume and Figure 3: SVC compression curve with 20 dB input volume.</td>
</tr>
<tr>
<td>28-Nov-2014</td>
<td>3</td>
<td>Updated RPN on cover page</td>
</tr>
<tr>
<td>10-Dec-2014</td>
<td>4</td>
<td>Updated Section 4.2.2 and Section 5</td>
</tr>
</tbody>
</table>