SPIRIT DSP
Embedded Voice Experience

SPIRIT Mixer
User's Guide

Version 1.0

October, 2010

Copyright Information

Read this first

The document describes mixer software modules and detailed software interface definitions, contains flow-charts and testing procedures description. Document contains examples of the software integration and usage.

To read this document you are not required to have any special knowledge. However, prior experience with C-programming is desirable.

The mixer software performs weighted time-domain mixing of two PCM sources. It can be used for implementing cross-fade functionality as well as for overlapping two audio sources.

The document includes parametric mixer algorithm overview, recommendations on the software applications, API description, and testing procedure.

Contents

1. Introduction

The purpose of this document is to describe API, integration process and test procedures of mixer software.

1.1. Product overview

The mixer software performs linear averaging of data coming two sources with up to 2 channels in each. Algorithm operates on "frame by frame" basis. The frame length is a user-defined parameter.

Specification of the mixer software product is presented in **Table 1**.

Algorithm	Weighted sum of two signals, with dynamically changing weights.
Channels number	1,2
Maximum Frame size	Arbitrary
Algorithmic delay	No extra delay, except framing delay.
Signal gain range	-inf − 0 dB
Input and output signal format	Interleaved, linear 16-bit PCM, little-endian format.
Runtime adjustment	Gain value, transition length

Table 1. Specifications

For more information about other SPIRIT audio processing technologies visit www.spiritDSP.com

SPIRIT reserves the right to make changes to its products to improve the products' technical characteristics without any notice. Customers are advised to obtain the latest version of relevant information to verify that the data is up-to-date before placing the orders.

SPIRIT warrants performance of its products to current specifications in accordance with SPIRIT's standard warranty. Testing and other quality control techniques are utilized to the extent deemed necessary to support this warranty.

1.2. Related products

The software can be efficiently used in conjunction with other products of SPIRIT:

- MPEG-4 AAC Decoder
- MPEG-1 Layer 3 Codec
- Dynamic Range Control
- Sample Rate Converter
- Automatic Gain Control

Since data format is very common, it can be easily interfaced with other Third Party products.

1.3. Document Overview

This document is organized as follows.

Section 2 explains a functional description of the software.

Section 3 describes integration flow, descriptions of structures and interface functions.

2. Functional Description

This section describes mixer algorithm and provides several recommendations on mixer usage.

2.1. Algorithm overview

2.1.1. Overall mixer scheme

The mixer consists of two controlled amplifiers which process input PCM data stream with *Gain* and 1-Gain multipliers as depicted in Figure 1. Gain value is linearly increased or decreased toward the user defined value during user the user defined transition period. Output signal is obtained by adding results of two amplifiers. Source 1 and Source 2 can be one-channel or dual-channel signals.

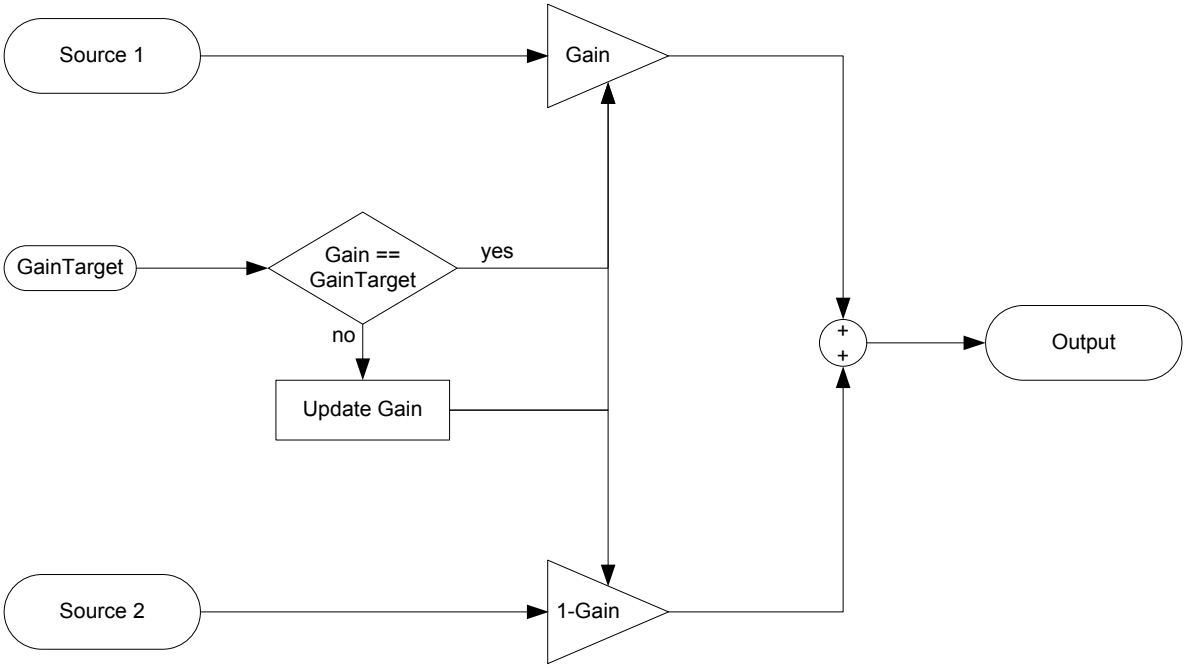

Figure 1. Block diagram of mixer algorithm

The mixer performs the following operation:

$$S_{out} = \alpha S_1 + (1-\alpha) S_2,$$

where S_1 and S_1 - input PCM samples, α - time varying gain value, S_{out} - output PCM sample.

2.2. Features and recommendations on mixer usage

2.2.1. Side tones

Figure 2 demonstrates an example of the mixer software usage where mixer is used to provide audible key press feedback while playing audio. When user presses a key, while playing music, the tone generator produces tonal signal. The tone added to music with the mixer, such that user can hear it on background, without music interrupt.

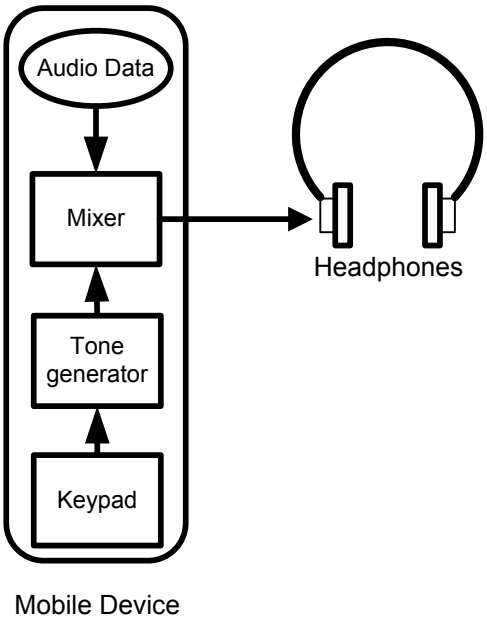

Figure 2. Side tones playback using mixer

2.2.2. Cross-fade

Figure 3 demonstrates example of the mixer software usage when implementing smooth transition between two songs (“cross-fade”). The “cross-fade” is a technique of smooth transition from one song to another. Old song is gradually reduced in volume (“fade-out”), while new song is gradually increased in volume (“fade-in”). Both songs are mixed and reproduced simultaneously. The Mixer software can be used for both mixing and gradual volume change, and is sufficient to implement cross-fade effect.

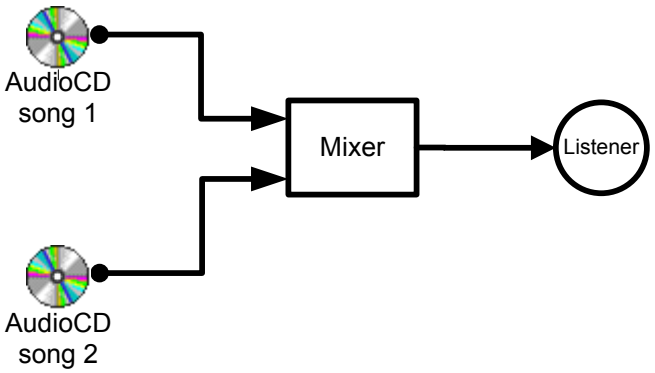

Figure 3. Cross-fade between two songs

3. API description

This section describes software implementation and its interface.

3.1. Integration flow

In order to integrate the mixer into user's framework, the user should:

- Set mixer parameters using *SpiritMixer_SetPrms()* function.
- Feed each new block of input data to *SpiritMixer_Apply()* function.

3.2. Predefined constants

SPIRIT_MIXER_PERSIST_SIZE_IN_BYTES	Persistent memory size in bytes.
SPIRIT_MIXER_MAX_CH	Maximal channel number.
SPIRIT_MIXER_MAX_GAIN_Q15	Maximal gain value (in Q15 format).

3.3. TSpiritMixer

Syntax

```
typedef void TSpiritMixer;
```

Remarks

Alias of the persistent memory buffer. Only a pointer on this type has sense; all API functions expect it points to a 4 bytes aligned memory region.

3.4. TSpiritMixer_Prms

Syntax

```
typedef struct {
    short nChannels;
    short gainQ15;
    long  transLen;
} TSpiritMixer_Prms;
```

Members

nChannels	Number of channels in source material, 1 or 2.
gainQ15	Desired gain in Q15.
transLen	Gain transition length in samples.

Remarks

This structure carries used for reading and setup mixer parameters using *SpiritMixer_SetPrms()* and *SpiritMixer_GetPrms()*. The *gainQ15* is applied to first source signal while the second is amplified by *(1_{Q15} - gainQ15)* gain.

Both of source signals must have the same number of channels.

3.5. SpiritMixer_SetPrms()

Syntax

```
int SpiritMixer_SetPrms(
          TSpiritMixer *mx,
          const TSpiritMixer_Prms *prms
);
```

Parameters

mx	Pointer to initialized persistent memory buffer.
prms	Initialization parameters.

Return value

Error code.

Remarks

This function can be invoked in run-time. It is recommended to receive current parameters first and then setup new. The proper call sequence is the following:

```
TSpiritMixer_Prms prms;

SpiritMixer_SetPrms(obj, &prms);  // get current parameters

prms.xxx = xxx;                   // modify
prms.yyy = yyy;

SpiritMixer_SetPrms(obj, &prms);  // set new parameters
```

3.6. SpiritMixer_GetPrms()

Syntax

```
int SpiritMixer_GetPrms(
          const TSpiritMixer *mx,
          TSpiritMixer_Prms *prms
);
```

Parameters

mx	Pointer to initialized persistent memory buffer.
prms	Mixer parameters storage.

Return value

Error code.

Remarks

This function returns actual mixer parameters.

3.7. SpiritMixer_Apply()

Syntax

```
int SpiritMixer_Apply(
          TSpiritMixer *mx,
          const short *in0,
          const short *in1,
          int nSamplesPerCh,
          short *out
);
```

Parameters

mx	Pointer to initialized persistent memory buffer.
in0	First source PCM buffer.
in1	Second source PCM buffer.
nSamplesPerCh	Number of input frame samples per channel.
out	Output PCM buffer.

Return value

Error code.

Remarks

This function process input PCM data with given parameters (gain and transition length). The output buffer has the same length and structure as the input. It is allowed the origin of input and output buffers coincide.

3.8. Error codes

Enumerated name	Description
SPIRIT_MIXER_S_OK	Operation completed successfully.
SPIRIT_MIXER_E_INVALIDARG	Bad arguments for a function.
SPIRIT_MIXER_E_CH	Bad number of channels.
SPIRIT_MIXER_E_GAIN	Bad gain value.
SPIRIT_MIXER_E_BADALIGN_PERSIST	Bad persistent memory alignment.

3.9. Application example

This is an example of Mixer utility usage; it can be linked with mixer library and run.

```
#include <stdio.h>
#include "spiritMixer.h"

static long mixer[SPIRIT_MIXER_PERSIST_SIZE_IN_BYTES/4];

#define PCM_BUFFER_SIZE_IN_SAMPLES 256
static short bufIn0[PCM_BUFFER_SIZE_IN_SAMPLES];
static short bufIn1[PCM_BUFFER_SIZE_IN_SAMPLES];
static short bufOut[PCM_BUFFER_SIZE_IN_SAMPLES];

void Process_SingleFile (
    const char *szInputName1,
    const char *szInputName2,
    const char *szOutputName,
    short gainStart,
    short gainEnd,
    long  transitionLen
)
{
    FILE *pFileIn1 = fopen(szInputName1,  "rb");
    FILE *pFileIn2 = fopen(szInputName2,  "rb");
    FILE *pFileOut = fopen(szOutputName,  "wb");
    unsigned long  lHz = 48000;
    unsigned int   nCh = 2;
    TSpiritMixer_Prms prms;

    // Initialize
    SpiritMixer_Init(mixer);

    // Set initial params
    prms.nChannels = nCh;
    prms.gainQ15   = gainStart;
    prms.transLen  = 0;
    SpiritMixer_SetPrms(mixer,  &prms);

    // Set final gain and transition length
    prms.nChannels = nCh;
    prms.gainQ15   = gainEnd;
    prms.transLen  = transitionLen;
    SpiritMixer_SetPrms(mixer,  &prms);

    while (1) {
        int nSamples0 = fread(bufIn0, sizeof(short), PCM_BUFFER_SIZE_IN_SAMPLES, pFileIn1);
        int nSamples1 = fread(bufIn1, sizeof(short), PCM_BUFFER_SIZE_IN_SAMPLES, pFileIn2);
        int nSamplesPerCh = nSamples0 < nSamples1 ? nSamples0 : nSamples1;

        nSamplesPerCh >>= (nCh==2 ? 1:0);

        if(nSamplesPerCh == 0) {
            break;
        }

        // Run mixer
        SpiritMixer_Apply(mixer, bufIn0, bufIn1, nSamplesPerCh, bufOut);

        // Write output data
        fwrite(bufOut, sizeof(short), nSamplesPerCh*nCh, pFileOut);
    }

    fclose(pFileIn1);
    fclose(pFileIn2);
    fclose(pFileOut);
}
```