VS1003 8kHz AEC System

VSMPG "VLSI Solution Audio Decoder"

Project Code:
Project Name: VSMPG

Revision History			
Rev.	**Date**	**Author**	**Description**
0.6	2009-12-22	PO	Max gain for AGC, 80 Hz highpass filter added.
0.5	2009-12-02	PO	Initial version

Table of Contents

1 System Overview

In addition to DAC and earphone driver, the VS1003 contains a microphone input that can be used in two-way communications in mobile phones or general head-set applications.

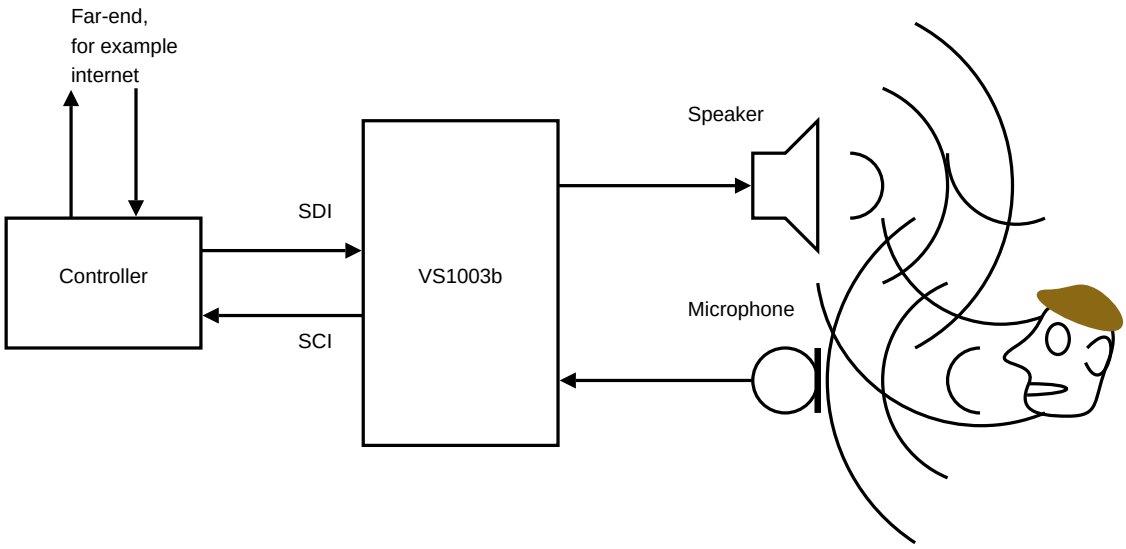

Figure 1.1: Speaker-Phone Application

In speaker-phone applications the microphone also picks up the speaker signal reflected by the room and people in it, creating a loopback and you will hear your own voice from the other end. This is very distracting and may make it impossible to have a conversation. The speaker sound can also travel through the unit chassis to the microphone.

To reduce this problem an acoustic echo cancellation (AEC) algorithm is used. AEC removes from the mic as much of the speaker signal as possible to reduce this loopback effect.

Note: Also see the application note "VoIP Acoustic Design" from http://www.vlsi.fi/en/support/evaluationboards/voipspeakerphone.html . The chassis must be designed so that the minimum of sound reaches the mic from the speaker through air or the chassis material. This may require either isolating the speaker from the PCB and the chassis or connecting it securely to the chassis, depending on the situation. (See the document for more discussion and details.) Especially the mic must not overflow from the speaker signal, nor should the speaker signal arrive distorted to the mic.

1.1 VS1003 8 kHz AEC System

The VS1003 8 kHz AEC system block diagram is shown in figure 1.2.

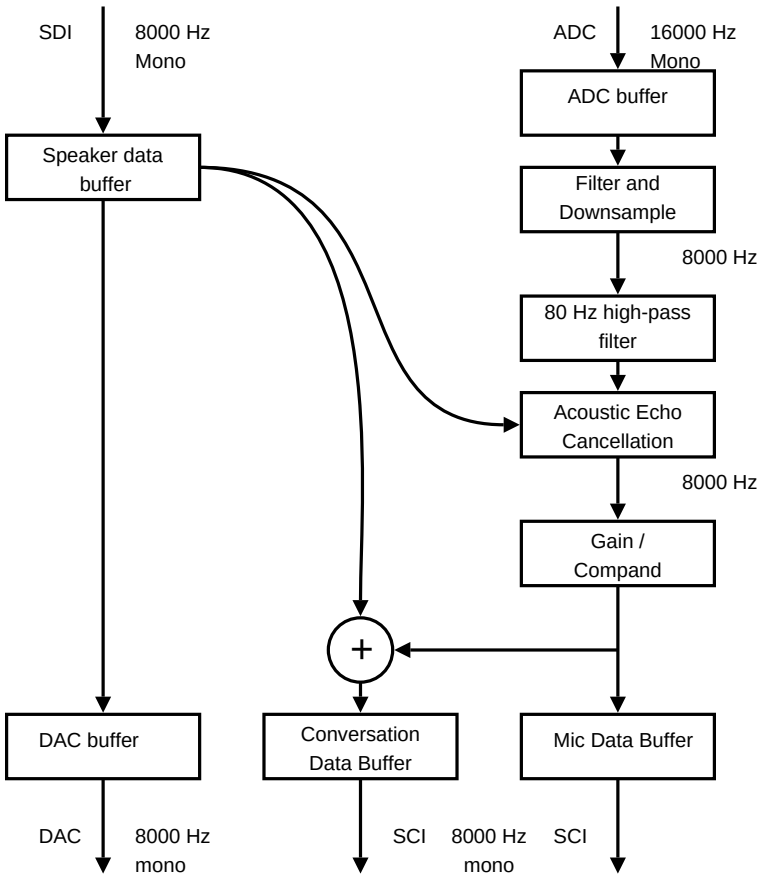

Figure 1.2: VS1003 8 kHz AEC System

There are two main data paths with first-in-first-out buffer (FIFO's). One FIFO is for the speaker signal, another is for the mic signal. An additional FIFO can be used to read out the conversation (both AEC-processed mic signal and speaker signal added together).

The speaker signal is transferred through the serial data interface (SDI). DREQ operates normally (rises when the FIFO starts to become full), but in real-time applications you can also read the buffer fill state from a SCI register.

The other FIFO's are read through SCI. The fill status registers should be read first to see how many data words are available.

2 Usage

2.1 Initialization

The application code is loaded to VS1003 using the SCI_WRAMADDR and SCI_WRAM registers. See chapter 3 for how to load the code using the plugin file. The AEC application is started by writing 0x0030 to SCI_AIADDR. After this the system can be controlled through SCI registers.

Files		
Chip	**12.288 MHz**	**Description**
VS1003B	aec12288.plg	8 kHz system for 12.288 MHz clock
VS1003B	aec12288.c	8 kHz system for 12.288 MHz clock

12.288 MHz input clock is assumed. With VS1003 the PLL will be set to 4.0× mode automatically.

Normal mode is restored by setting the software reset bit in the SCI_MODE register.

Note that this user application takes full control over the whole system. If any other user applications or patches are active, you should perform software reset before loading and activating this code.

2.2 Control

The following SCI registers control the behavior of the 8 kHz system. When the 8 kHz system software has been started, no other SCI registers should be used.

Register	Controls	Explanation
MODE	Mode	Software Reset detected
DECODE_TIME	SpkFill	Indicates speaker data (SDI) buffer fill state
AUDATA	ConvData	Conversation to PC (both Mic and Speaker data)
WRAMADDR		Lowest bit is MIC mute
HDAT0	MicData	Microphone data to PC
HDAT1	MicFill	Fill state of mic data buffer
VOL	Volume	Digital volume setting
AICTRL0	Adapt	Sets information about speaker-to-mic gain, use default: 2
AICTRL1	Gain	Microphone gain (0 = automatic gain control)
AICTRL2	MaxAGC	Maximum gain for AGC mode
AICTRL3	ConvFill	Fill state of conversation buffer

2.2.1 SCI_MODE

The SCI_MODE register uses the software reset bit (SM_RESET). When this bit is set by the user, the 8 kHz system will return control to the normal firmware code by performing a software reset.

In addition SCI_MODE is used to select ADC source between microphone and line input.

2.2.2 SCI_VOL

Only digital volume control is available.

Note that if you change volume, it may affect the AEC operation.

2.2.3 SCI_WRAMADDR

Bit 0 is mic mute, bit 1 is AEC disable. To disable MIC, write 1 to SCI_WRAMADDR. If you want to disable AEC, write 2 to SCI_WRAMADDR. This is useful when you want to test the AEC operation (but remember to let the system adapt when you change the state). For normal operation write 0 to SCI_WRAMADDR.

2.2.4 SCI_AICTRL0

SCI_AICTRL0 is used to indicate to the AEC software the gain of speaker compared to mic in 3 dB steps. This value is only read at AEC startup. Use 2 as a default value. You may need to use a higher value if you use speaker with high amplification.

2.2.5 SCI_AICTRL1

SCI_AICTRL1 controls the microphone gain. Valid values are from one (muted) through 1024 (1× gain) to 65535 (64×, i.e. +36 dB). If the value is 0, automatic gain control is used. The default value after startup is 0.

2.2.6 SCI_AUDATA / SCI_AICTRL3

SCI_AUDATA can be used to read out the conversation, i.e. both mic and speaker data mixed together. This is useful when you want to save the conversation to disk.

SCI_AICTRL3 gives the fill state of the conversation data buffer.

2.2.7 SDI / SCI_DECODE_TIME

Speaker data should be sent to the serial data inteface (SDI) in 125 μs intervals (i.e. 8 kHz).

A circular buffer allows you to send upto 255 samples beforehand. If the system runs out of data (you send samples too slowly), a zero-sample is inserted. If you send samples too fast, they are accumulated until the input data buffer overruns. If you do not have a 8 kHz timebase, you can use SCI_DECODE_TIME to see how many samples are waiting to be played.

SCI_DECODE_TIME is the speaker data buffer fullness indicator. If you have an exact 8 kHz timebase and are sending samples at the correct frequency, the number of samples in the input buffer should remain fairly low.

Also the DREQ pin reflects the state of the speaker data buffer. If there is no space, DREQ stays low, otherwise DREQ is high.

If you have a need to resynchronize, stop sending new samples until the speaker data buffer is empty.

2.2.8 SCI_HDAT0 / SCI_HDAT1

Microphone data is read from the SCI_HDAT0 register. The number of available samples is in SCI_HDAT1. If you read too much data (or too fast), the previous sample value is returned. If you read too little data (or too slowly), it is accumulated until the mic buffer overruns.

Because the software updates the contents of SCI_HDAT0, you should read the register with enough wait in-between (7.5 μs). Otherwise the contents of both SCI_HDAT1 and SCI_HDAT0 may not be correct.

SCI_HDAT1 contains the number of words waiting in the output buffer.

2.3 SCI Timing

Because the software must react to SCI register reads to update the register contents, there must be at least a 7.5 μs delay between reads/writes.

2.3.1 Main Loop Example

The following shows the basic operation of a main loop. If samples have been received, they are sent to vs1003 using SDI. If vs1003 has samples ready, the are read out and sent. The save to file example shows how to save data to a file so that it works correctly in both big-endian and little-endian systems.

```
typedef unsigned short u_int16;
..
void main(void) {
  SoftwareReset(); /*and wait for DREQ to rise*/
  LoadUserCode();
  WriteVS10xxRegister(SCI_AICTRL1/*0x0d*/, 0);/*autogain*/
  /* Activate User Code */
  WriteVS10xxRegister(SCI_MODE   /*0x00*/, 0x0800);/*new mode, select mic*/
  WriteVS10xxRegister(SCI_AICTRL0/*0x0c*/, 0x0002);/*default*/
  WriteVS10xxRegister(SCI_WRAMADDR/*0x07*/, 0x0000);/*flags off*/
  WriteVS10xxRegister(SCI_AIADDR /*0x0a*/, 0x0030);/*start application*/

  while (1) {
    u_int16 reg;
    /* Check for speaker data available */
    if (SamplesFromFarEnd()) {
      reg = GetSampleFromFarEnd();
      WriteVS10xxData(reg); /* Send data to Speaker through SDI */
    }
    /* Check for data from Mic */
    reg = ReadVS10xxReg(SCI_HDAT1);
    if (reg >= 2) { /* read two values at a time to reduce overheads */
      u_int16 dat[2];
      dat[0] = ReadVS10xxReg(SCI_HDAT0);
      dat[1] = ReadVS10xxReg(SCI_HDAT0);
#if 0 /*save mic data to disk*/
      {
        unsigned char tmp[4];
        tmp[0] = dat[0]>>8;
        tmp[1] = dat[0];
        tmp[2] = dat[1]>>8;
        tmp[3] = dat[1];
        if (fp)
          fwrite(tmp, 1, 4, fp);
      }
#else
      /* then send mic data */
      SendDataToFarEnd(dat, 2);
#endif
    }
```

```
    if (saveConversation) {
      reg = ReadVS10xxReg(SCI_AICTRL3); /* conversation fill state */
      if (reg >= 2) { /* read two at a time to reduce overheads */
        u_int16 dat[2];
        dat[0] = ReadVS10xxReg(SCI_AUDATA);
        dat[1] = ReadVS10xxReg(SCI_AUDATA);
#if 0   /*save conversation data to disk*/
        {
          unsigned char tmp[4];
          tmp[0] = dat[0]>>8;
          tmp[1] = dat[0];
          tmp[2] = dat[1]>>8;
          tmp[3] = dat[1];
          if (fp)
            fwrite(tmp, 1, 4, fp);
        }
#else
        /* then save conversation data */
        SaveConversationData(dat, 2);
#endif
      }
    }
    if (volumeChanged) {
      volumeChanged = 0;
      WriteVS10xxRegister(SCI_VOL, volume);
    }
  }
}
```

3 How to Load a Plugin

A plugin file (.plg) contains a data file that contains one unsigned 16-bit array called plugin. The file is in an interleaved and RLE compressed format. An example of a plugin array is:

```
const unsigned short plugin[10] = { /* Compressed plugin */
  0x0007, 0x0001, 0x8260,
  0x0006, 0x0002, 0x1234, 0x5678,
  0x0006, 0x8004, 0xabcd,
};
```

The vector is decoded as follows:

1. Read register address number `addr` and repeat number `n`.
2. If `(n & 0x8000U)`, write the next word `n` times to register `addr`.
3. Else write next `n` words to register `addr`.
4. Continue until array has been exhausted.

The example array first tells to write 0x8260 to register 7. Then write 2 words, 0x1234 and 0x5678, to register 6. Finally, write 0xabcd 4 times to register 6.

Assuming the plugin array (`plugin[]`) is in file `aec12888.plg`, a full decoder in C language is provided below:

```
#include "aec12288.plg"
void WriteVS10xxRegister(unsigned short addr, unsigned short value);

void LoadUserCode(void) {
  int i = 0;

  while (i<sizeof(plugin)/sizeof(plugin[0])) {
    unsigned short addr, n, val;
    addr = plugin[i++];
    n = plugin[i++];
    if (n & 0x8000U) { /* RLE run, replicate n samples */
      n &= 0x7FFF;
      val = plugin[i++];
      while (n--) {
        WriteVS10xxRegister(addr, val);
      }
    } else {           /* Copy run, copy n samples */
      while (n--) {
        val = plugin[i++];
        WriteVS10xxRegister(addr, val);
      }
    }
  }
}
```

4 How to Use Old Loading Tables

Each patch contains two arrays: `atab` and `dtab`. `dtab` contains the data words to write, and `atab` gives the SCI registers to write the data values into. For example:

```
const unsigned char atab[] = { /* Register addresses */
    7, 6, 6, 6, 6
};
const unsigned short dtab[] = { /* Data to write */
    0x8260, 0x0030, 0x0717, 0xb080, 0x3c17
};
```

These arrays tell to write 0x8260 to SCI_WRAMADDR (register 7), then 0x0030, 0x0717, 0xb080, and 0x3c17 to SCI_WRAM (register 6). This sequence writes two 32-bit instruction words to instruction RAM starting from address 0x260. It is also possible to write 16-bit words to X and Y RAM. The following code loads the patch code into VS10xx memory.

```
/* A prototype for a function that writes to SCI */
void WriteVS10xxRegister(unsigned char sciReg, unsigned short data);

void LoadUserCode(void) {
  int i;
  for (i=0;i<sizeof(dtab)/sizeof(dtab[0]);i++) {
    WriteVS10xxRegister(atab[i]/*SCI register*/, dtab[i]/*data word*/);
  }
}
```

Patch code tables use mainly these two registers to apply patches, but they may also contain other SCI registers, especially SCI_AIADDR (10), which is the application code hook.

If different patch codes do not use overlapping memory areas, you can concatenate the data from separate patch arrays into one pair of `atab` and `dtab` arrays, and load them with a single `LoadUserCode()`.