VS1053B PCM MIXER

VSMPG “VLSI Solution Audio Decoder”

Project Code:
Project Name: VSMPG

Revision History			
Rev.	**Date**	**Author**	**Description**
1.3	2015-01-26	PO	Flexible rates, more like vs1063a.
1.2	2011-05-30	PO	GPIO configurable to reflect fill state.
1.1	2011-04-27	PO	Rate configured from AICTRL0
1.0	2011-03-14	PO	Initial version

1 Description

The new VS1053 audio path provides a lot of tricks not previously possible. For example it allows audio mixed into output while performing normal decoding. This way you can mix PCM samples with other audio.

The PCM mixer allows a mono linear PCM stream from 8 kHz to 48 kHz (and above) to be played back during any audio format playback. The desired rate is configured from AICTRL0 before starting the code. This audio signal is upsampled to at least 24kHz and played through the SDM side channel of vs1053.

With 12.288MHz XTALI you get exact rates of 8, 12, 16, 24, and 48 kHz. Other rates or rates with a different XTALI may not be exact.

The PCM data is sent through the SCI_AICTRL1 register, SCI_AICTRL0 tells how much space is in the PCM FIFO (you can send upto this many words). Note that SCI multiple write (see vs1053 datasheet) can be used to write multiple words with minimal overhead.

During startup (when you activate the mixer) AICTRL0 specifies the sample rate and AICTRL1 specifies which GPIO pin(s) (if any) to use for flow control. The value is a mask, so if you don't want any GPIO pin used, write 0 to AICTRL1. Use 0x02 for GPIO1, 0x04 for GPIO2, 0x08 for GPIO3 etc.

SCI_AICTRL2 controls volume independently of the normal playback volume. SCI_AICTRL2 value from 0 to 181 controls PCM volume in 0.5dB steps. Note: only the low 8 bits of SCI_AICTRL2 control the volume.

Note: to prevent sigma-delta modulator overflow, SCI_VOL should be at least 2dB (0x0404), and the sum of SCI_VOL and SCI_AICTRL2 attenuations at least 6dB (12). If you have not set large enough attenuations, the PCM Mixer adjusts the registers automatically to have at least these values. To have absolutely safe scaling, have 6dB (0x0c0c) or more in both SCI_VOL and SCI_AICTRL2.

The PCM mixer takes about 4.0 MHz of processing power when in use with 8 kHz rate. Processing will be automatically disabled after a 0.125-second timeout when samples are not being written to SCI_AICTRL1. The processing is resumed when there are at least 64 samples in the PCM FIFO (1/4 full).

The PCM mixer works with the basic VS1053b Patches Package versions that do not include the FLAC decoder. The versions with the FLAC decoder are larger, and the PCM mixer overlaps their code area. If you use the VS1053b Patches package, load it first, then the VS1053 PCM Mixer (PCM Mixer uses the application hook).

Chip	File	IRAM	Description
VS1053B	vs1053pcm.plg	0xd00 .. 0xe9e	compressed plugin
VS1053B	vs1053pcm.cmd	0xd00 .. 0xe9e	old atab+dtab format

This patch uses the SDM interrupt vector, so it can not be used with other encoding modes or with ADMixer (but ADMixer can be loaded into memory at the same time). However, it can be used with the vs1053b-patches package.

PCM Mixer uses X data RAM 0x3580..0x35a1 (interpolation) and 0x1858..0x1863 (pointers and counters) and Y data RAM 0xeb00..0xebff (PCM FIFO).

1.1 Control

Register	Description
SCI_AICTRL0	Free space in the buffer, configures rate at startup
SCI_AICTRL1	Write to send samples
SCI_AICTRL2	Write to set volume attenuation for PCM (0..181)

Perform the the normal system startup (including loading of the vs1053 patches package if you are using it), then load vs1053pcm.plg, then write your samplerate 8000, 12000, 16000, or 24000 to AICTRL0, you can also write the suitable volume settings to SCI_VOL and AICTRL2, then write 0x0d00 to AIADDR to start the PCM Mixer.
- read AICTRL0: tells how many samples can be sent
- write AICTRL1 to send the samples
- write AICTRL2 to set volume attenuation for the PCM stream (only low byte)

The PCM Mixer will deactivate in a software reset, so load it again and write 0x0d00 to AIADDR to restart.

```
/*Initialization*/
Mp3WriteReg(SCI_AICTRL0, 8000);   /*set rate to 8kHz*/
Mp3WriteReg(SCI_AICTRL1, 0x10);   /*GPIO4 is high if space for 32 samples*/
Mp3WriteReg(SCI_VOL,     0x0c0c); /*lower volume*/
Mp3WriteReg(SCI_AICTRL2, 0x0c);   /*set pcm volume*/
Mp3WriteReg(SCI_AIADDR,  0x0d00); /*start pcm mixer*/

/*Check fill state and send PCM data*/
s_int16 samples[32];

s_int16 availSpace = Mp3ReadReg(SCI_AICTRL0);
if (availSpace > 32) {
    ReadSamples(samples, 32);
    Mp3WriteRegMultiple(SCI_AICTRL1, samples, 32);
}
```

2 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 array is in `plugin[]`, a full decoder in C language is provided below:

```
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);
      }
    }
  }
}
```

3 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()`.