VS1053B ADC Mix

VSMPG "VLSI Solution Audio Decoder"

Project Code:
Project Name: VSMPG

Revision History			
Rev.	**Date**	**Author**	**Description**
1.3	2011-11-07	PO	Added channel swap and mono versions. Does not turn ADC off to avoid DECIM-restart bug.
1.2	2010-05-27	PO	Small bugfix for left and right input versions.
1.1	2009-11-05	PO	Separate versions for left and right input. Gain control added.
1.0	2009-06-29	PO	Initial version

1 Description

The new VS1053 audio path provides a lot of tricks not previously possible. For example it allows ADC monitoring while performing normal decoding. This way you can mix stereo line inputs with other audio.

This plugin has several versions. One takes both ADC channels and routes them to left and right outputs. The second one swaps channels. Two other versions take either left or right ADC input and plays them using both left and right outputs. The mono version mixes both left and right inputs and plays them using both left and right outputs.

There plugins are distributed in two formats: the new compressed plugin, and the old loading tables. The new compressed plugin format is recommended, because it saves data space and future plugins, patches, and application will be using the new format.

Chip	File	IRAM	Description
VS1053B	admix-stereo.plg	0xf00 .. 0xf37	Stereo version
VS1053B	admix-swap.plg	0xf00 .. 0xf37	Stereo channel swap version
VS1053B	admix-left.plg	0xf00 .. 0xf31	MIC/LINE1 version
VS1053B	admix-right.plg	0xf00 .. 0xf31	LINE2 version
VS1053B	admix-mono.plg	0xf00 .. 0xf34	LINE1+LINE2 version

Chip	File	IRAM	Description
VS1053B	admix-stereo.c	0xf00 .. 0xf37	old atab+dtab format
VS1053B	admix-swap.c	0xf00 .. 0xf37	old atab+dtab format
VS1053B	admix-left.c	0xf00 .. 0xf31	old atab+dtab format
VS1053B	admix-right.c	0xf00 .. 0xf31	old atab+dtab format
VS1053B	admix-mono.c	0xf00 .. 0xf34	old atab+dtab format

This patch uses the ADC interrupt vector, so it can not be used with other encoding modes. However, it can be used with the vs1053b-patches package.

1.1   Control

Register	Description
SCI_MODE	Clear SM_LINE1 bit for MIC, set for LINE1
SCI_AICTRL0	Gain: -3 (0xfffd, max volume) to -31 (quiet).
SCI_AIADDR	0x0f00 to activate, 0x0f01 to deactivate

First load the appropriate plugin using the .plg file.

Then set the SCI_MODE register value. The reset value for SCI_MODE has SM_LINE1 bit set, so if you want the MIC input to be used, remember to clear it.

Then set the suitable monitoring gain by writing to SCI_AICTRL0. The gain is set in 3 dB steps. -3 (0xfffd) is the highest value you should use. -4 is -3 dB, -5 is -6 dB, etc. upto -31. Gain can also be changed when the plugin is active.

And finally, activate the plugin by writing 0x0f00 to SCI_AIADDR. SCI_AIADDR will be cleared automatically when the plugin has started.

You can deactivate the plugin by writing 0xf01 to SCI_AIADDR. SCI_AIADDR will be cleared automatically when the plugin has deactivated itself. You can also deactivate the plugin by giving a software reset.

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:

```c
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()`.