VS1053B PARAMETRIC EQ

VSMPG “VLSI Solution Audio Decoder”

Project Code: VS1053
Project Name: Support

Revision History			
Rev.	**Date**	**Author**	**Description**
0.7	2017-09-11	PO	Version with common filters for both channels.
0.6	2017-07-31	PO	Initial version.

Contents

List of Figures

1 Description

When bass and treble controls or the 5-band EQ are not enough, this PEQ plugin package provides you with up to 20 parametric filters independently for both outputs channels. Another version provides you with up to 40 parametric filters common for both channels.

This plugin uses the Application address hook.

File	IRAM	Description
vs1053b-peq.plg	0x800..0xf29	independent, compressed plugin
vs1053b-peq.c	0x800..0xf29	independent, old array format
vs1053b-peq-c.plg	0x800..0xec7	common filters, compressed plugin
vs1053b-peq-c.c	0x800..0xec7	common filters, old array format

These plugins also use memory areas
X:0x1ed7..0x1eef, X:0x35b7..0x36f6, X:0x3700..0x3a4c,
Y:0x1800..0x187d, Y:0xfc57..0xff2a, Y:0xff80..0xffcb.

These plugins use the application address to start automatically (the last entry in the patch tables writes to SCI_AIADDR), and also use it afterwards to process the samples.

Both old loading tables and the new compressed plugin format is available. The new plugin format is recommended, because it saves data space and future plugins, patches, and application will be using the new format.

These plugins are compatible with the normal (`vs1053b-patches.plg`) and -latm (`vs1053b-patches-latm.plg`) versions of the vs1053b patches package.
Load the vs1053b Patches package first (including starting it), then `vs1053b-peq` or `vs1053b-peq-c`.

To disable the PEQ plugins write 0 to SCI_AIADDR.

1.1 Control

The independent-filter PEQ is configured through user RAM at addresses Y:0x1800..0x187e.
Locations Y:0x1800 to 0x183e control filters for the left channel.
Locations Y:0x183f to 0x187e control filters for the right channel.

	Address	Field	Description
Left channel filters	0x1800	update	Set to 1 to update filters.
	0x1801	rate	Remembers the samplerate.
	0x1802	num filters	Number of filters that follow
	0x1803 ..	filter data	up to 20 filters, 3 words each
Right channel filters	0x183f	update	Set to 1 to update filters.
	0x1840	rate	Remembers the samplerate.
	0x1841	num filters	Number of filters that follow
	0x1842 ..	filter data	up to 20 filters, 3 words each

The common-filter PEQ is configured through user RAM at addresses Y:0x1800..0x187e.
Locations Y:0x1800 to 0x187b control filters for both channels.

	Address	Field	Description
Common filters	0x1800	update	Set to 1 to update filters.
	0x1801	rate	Remembers the samplerate.
	0x1802	num filters	Number of filters that follow
	0x1803 ..	filter data	up to 40 filters, 3 words each

Parametric Filter

Each filter is fully parametric and defined by their center frequency, gain (positive or negative), and Q value. The bandwidth of a filter is its center frequency divided by Q value. A higher Q value produces a narrower bandwidth.

Offset	Field	Description
1	frequency	The center frequency of the filter, in Hz.
2	gainx10	The gain of the filter in 10ths of dB. Positive for emphasizing, negative for a suppressive filter. For example -25 is -2.5dB, 30 is +3.0dB .
3	qvaluex10	The Q value of the filter times 10. For example 15 is Q=1.5, 35 is Q=3.5 .

Note: currently there is no automatic allocation of headroom, so you must be careful when defining emphasizing filters.

Write the number of filters and the filter data first, then set the `update` field(s) to 1. When the filters have been updated, the `update` field(s) are set to 0.

Note that when writing through SCI_WRAM, add 0x4000 to the WRAMADDR address to write to Y memory.

Code example for loading filters to the independent-filter PEQ:

```
const short filters[] = {
  250, -3*10, 30, /* 20Hz, -3.0dB, Q=3.0 */
  1200, -3*10, 25, /* 1200Hz, -3.0dB, Q=2.5 */
  0, 0, 0 /* end marker */
};
/* Use the same filters for left and right */
void LoadFilters() {
  int i;
  WriteMp3Reg(SCI_WRAMADDR, 0x5803); /* Left filters start */
  i=0;
  do {
    WriteMp3Reg(SCI_WRAM, filters[i+0]); //frequency
    WriteMp3Reg(SCI_WRAM, filters[i+1]); //gain
    WriteMp3Reg(SCI_WRAM, filters[i+2]); //Q value
    i+=3;
  } while (filters[i]);
  WriteMp3Reg(SCI_WRAMADDR, 0x5842); /* Right filters start */
  i=0;
  do {
    WriteMp3Reg(SCI_WRAM, filters[i+0]); //frequency
    WriteMp3Reg(SCI_WRAM, filters[i+1]); //gain
    WriteMp3Reg(SCI_WRAM, filters[i+2]); //Q value
    i+=3;
  } while (filters[i]);
  /* Write the number of filters */
  WriteMp3Reg(SCI_WRAMADDR, 0x5802);
  WriteMp3Reg(SCI_WRAM, i/3);
  WriteMp3Reg(SCI_WRAMADDR, 0x5841);
  WriteMp3Reg(SCI_WRAM, i/3);
  /* Force update */
  WriteMp3Reg(SCI_WRAMADDR, 0x5800);
  WriteMp3Reg(SCI_WRAM, 1);
  WriteMp3Reg(SCI_WRAMADDR, 0x583f);
  WriteMp3Reg(SCI_WRAM, 1);
}
```

Examples

```
WRAMADDR=0x5802 WRAM=0x0003          left channel, 3 filters
WRAM=0x008a WRAM=0xffce WRAM=0x0014  138Hz, -5.0dB, Q=2.0
WRAM=0x03e8 WRAM=0xffc4 WRAM=0x0023  1000Hz, -6.0dB, Q=3.5
WRAM=0x27f1 WRAM=0xff9c WRAM=0x0032  10225Hz, -10.0dB, Q=5.0
WRAMADDR=0x5841 WRAM=0x0002          right channel, 2 filters
WRAM=0x0028 WRAM=0xfff6 WRAM=0x000f  40Hz, -1.0dB, Q=1.5
WRAM=0x03e8 WRAM=0xffc4 WRAM=0x0023  1000Hz, -6.0dB, Q=3.5
WRAMADDR=0x5800 WRAM=0x0001  updates filters for the left channel
WRAMADDR=0x583f WRAM=0x0001  updates filters for the right channel
```

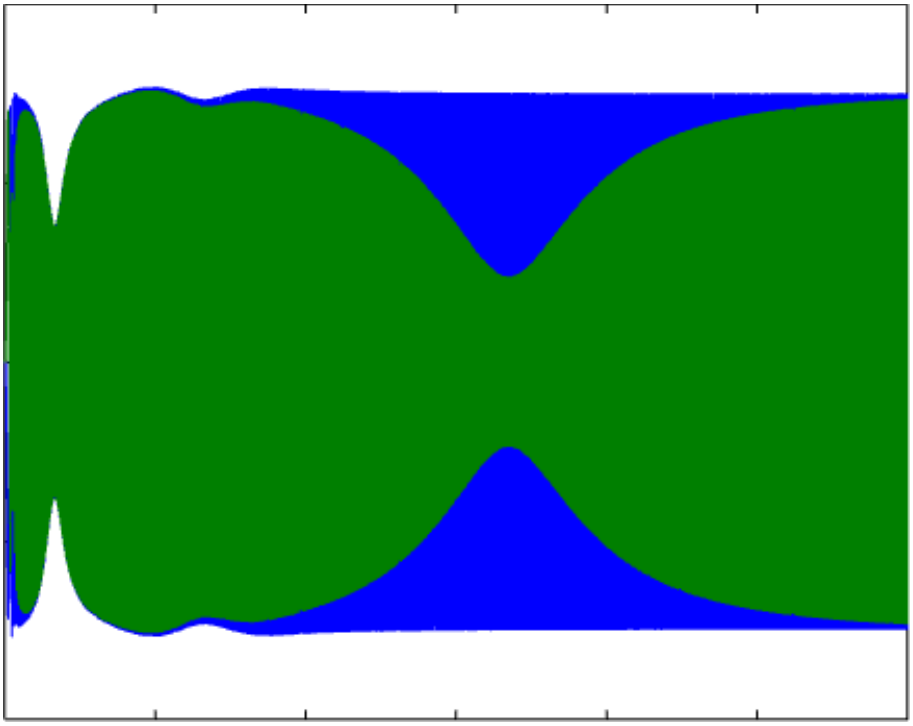

Figure 1: Example Response, Sine sweep

CPU Consumption

The CPU consumption varies depending on the center frequency and other parameters of the filters and the playback samplerate. The number of filters does not directly determine the CPU consumption.

The value of SCI_AICTRL1 gives feedback of the required CPU consumption in 100kHz steps. For example a value of 161 would correspond to 16.1MHz.

The maximum CPU consumption is about 30 MHz with 48 kHz samplerate.

SCI_AICTRL0 returns a non-zero value when there are active filters.

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);
      }
    }
  }
}
```