VS1033D Patches

VSMPG "VLSI Solution Audio Decoder"

Project Code:
Project Name: VSMPG

Revision History			
Rev.	**Date**	**Author**	**Description**
1.1	2011-02-02	PO	AAC: fix for MP4 format with unused data at the start of the mdat atom.
1.0	2008-08-15	PO	Initial version layer III fix

1 Description

There are some known problems in the VS1033d firmware that this patch addresses.

- MP3: frame ending during huffman data causes audio to be muted.
- AAC: MP4 format with unused data at the start of the mdat atom are not played.

This patch uses the application address to start automatically (the last entry in the patch tables writes to SCI_AIADDR), but does not use it afterwards. So, you must load any patch that actually uses the application address after this patch or it will be deactivated.

Chip	File	IRAM	Description
VS1033D	vs1033d-patches.plg	0x260 .. 0x2bf	compressed plugin
VS1033D	vs1033d-patches-tab.c	0x260 .. 0x2bf	old array format

Hardware or software reset will deactivate the patch. You must reload the patch after each hardware and software reset.

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

1.1 Fixes in Detail

MP3

VS1033D contains totally new full-accuracy MP3 decoder. Unfortunately, the new decoder interprets bits running out during huffman decoding as an error and clears the already decoded part of the data. This causes part of the sound to vanish. This seems to happen predominantly with 48 kHz MP3 files and in the left channel only.

This patch corrects the problem.

AAC: MP4 format

Unused data at the start of the **mdat** atom is now skipped so the start of audio data is located correctly.

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()`.