VS10xx DAC Mode Application

VSMPG "VLSI Solution Audio Decoder"

Project Code:
Project Name: VSMPG

Revision History

Rev.	Date	Author	Description
1.2	2009-07-09	PO	Documentation updated
1.1	2009-03-13	PO	Added VS1002 version
1.0	2009-02-18	PO	Initial version

1 Description

DAC Mode Application provides a low-delay 16-bit mono or stereo PCM sample output. SDI FIFO fill states can be monitored to adjust the samplerate for streaming applications.

There are two first-in-first-out (FIFO) buffers in VS10xx: a data FIFO for the received SDI bytes and an audio FIFO for samples going to the digital-to-analog converter (DAC). Both FIFO's are required in normal decoder use to even out the consumption of the bitstream data and to spread out the cycles needed for decoding. The audio buffer can receive a whole audio frame immediately when it has been decoded, so work on the the next frame can be started and cycles are not lost waiting for the audio buffer to empty.

However, the FIFO's also cause a considerable delay from the data input to the DAC, especially with small samplerates and bitrates. This delay in unnecessary with uncompressed data.

The DAC application minimizes the SDI to DAC delay by keeping the audio buffer as empty as possible and letting the user read the SDI FIFO fill state from AICTRL3 to control the input delay.

Note: Other application code can not be running at the same time.

Chip	File	IRAM	Description
VS1053B	dacmode1053-plg.h	0x50 .. 0x101	new compressed plugin file
VS1053B	dacmode1053-ctab.h	0x50 .. 0x101	old atab+dtab array format
VS1033C	dacmode1033c-plg.h	0x50 .. 0x111	new compressed plugin file
VS1033C	dacmode1033c-ctab.h	0x50 .. 0x111	old atab+dtab array format
VS1033D	dacmode1033d-plg.h	0x50 .. 0x111	new compressed plugin file
VS1033D	dacmode1033d-ctab.h	0x50 .. 0x111	old atab+dtab array format
VS1003B	dacmode1003-plg.h	0x50 .. 0x137	new compressed plugin file
VS1003B	dacmode1003-ctab.h	0x50 .. 0x137	old atab+dtab array format
VS1002D	dacmode1002-plg.h	0x50 .. 0x111	new compressed plugin file
VS1002D	dacmode1002-ctab.h	0x50 .. 0x111	old atab+dtab array format
VS1011E	dacmode1011e-plg.h	0x50 .. 0x111	new compressed plugin file
VS1011E	dacmode1011e-ctab.h	0x50 .. 0x111	old atab+dtab array format

- Give a software reset if you have other user code loaded.
- Wait for DREQ to rise and load the plugin. It will be started automatically.
- Set the playback rate by writing the samplerate to SCI_AUDATA.
- Select the correct linear 16-bit playback format by sending "RAWm" (mono little-endian), "RAWM" (mono big-endian), "RAWs" (stereo little-endian), or "RAWS" (stereo big-endian) to SDI. After receiving any of these strings the plugin will go to decoding mode and copy the low 16 bits of mode to SCI_HDAT1.
- Send data at the proper rate.

- Volume control is available during play.
- Bass and treble controls are available except in the VS1011e and VS1002d version.
- SCI_AICTRL3 contains the fill state of the SDI FIFO. If it starts to fill up, you can increase the samplerate temporarily by writing to SCI_AUDATA.
- SCI_AICTRL2 contains the number of stereo samples in the audio FIFO. Normally you don't need to monitor this register. Zero samples are automatically inserted to the audio FIFO if there is not enough data in SDI FIFO. In VS1053 the signal is faded towards zero.
- You can return to normal decoding mode by giving a software reset.

If you are not sending samples from a file, send samples only when the stream buffer is sufficiently empty. If the samples are received from a real-time source instead (such as USB audio or from Internet), you need to adjust the vs10xx samplerate dynamically to keep audio delay in control.

Check the stream fill state either before or after sending an audio packet to vs10xx, and try to keep the value to for example three times the audio packet size.

If the fill state is below your threshold, slow down the samplerate, if the fill state is sufficiently above, speed up the samplerate. In the middle set the nominal rate.

The rate change can also be bigger the further away from your selected value the fill state is. In VS1000 when we use USB audio, we have allowed the tuning to be +-0.2%, i.e. from 44011Hz to 44188Hz for 44100Hz nominal rate. For other applications the allowed range may need to be higher.

The best place to check is each time you have received an Audio packet. Either before or after you have sent the data to SDI.

An example tuning algorithm:

```
streamFill = Mp3ReadReg(SCI_AICTRL3);
control = streamFill - (AUDIO_DELAY_FRAMES*AUDIO_PACKET_WORDS);
newRate = nominalRate + control * scale;
if (newRate < nominalRate * 0.998 || newRate > nominalRate * 1.002) {
  newRate = nominalRate;
}
if (newRate != oldRate) {
  Mp3WriteReg(SCI_AUDATA, newRate);
  oldRate = newRate;
}
```

"scale" depends on AUDIO_DELAY_FRAMES. The larger the delay, the smaller the scale can be. If may be a good idea to put a debug printout after the write to SCI_AUDATA and adjust the scale until the operation becomes stable.

For stereo 16-bit 44.1kHz USB audio AUDIO_PACKET_WORDS would be 2*44100/1000, i.e. approximately 88 words every ms.

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()`.