VS1103B TRADITIONAL USAGE PATCH

VSMPG “VLSI Solution Audio Decoder”

Project Code: -
Project Name: VSMPG

All information in this document is provided as-is without warranty. Features are subject to change without notice.

Revision History			
Rev.	**Date**	**Author**	**Description**
1.02	2014-07-24	HH	Added plugin format.
1.01	2007-06-20	HH	Updated for VS1103b.
1.00	2007-03-23	HH	Initial version for VS1103a.

Contents

1 Introduction

1.1 Description

VS1103b is a versatile MIDI + WAV PCM/ADPCM codec chip that supports complex playback and mixing multiple audio sources. However, its usage is quite different from older, single-stream VS10XX chips as VS1003.

The goal of the VS1103b Trad Patch is to make using the VS1103b behave as similarly as possible to the way VS1003 is used. It supports playback of 8-bit and 16-bit mono and stereo PCM WAV files as well as 4-bit mono and stereo IMA ADPCM WAV files. It also supports ADPCM recording from a microphone or line input at a sample rate selected by the user.

1.2 SCI Commands

SCI commands are presented in one of the following formats: "`w x yyyy`" or "`w 2 x yyyy`". In this format you write hexadecimal number `yyyy` to SCI register `x`, which is also in hexadeciman notation. E.g. "`w 0 0c00`" means that 0x0c00 is written to SCI register 0x0, or SCI_MODE.

2 Using the Patch

2.1 Loading and Activating the Patch

First load the patch code to RAM memory. See Chapter 4 for details on how to do it. Then perform the following SCI command that starts the application from the address where it was loaded, 0x30:

```
# Sets application address to 0x30
w a 30
```

2.2 Playing MIDI or WAV PCM/ADPCM Files

After activating the patch, you may execute the following sequence to enable MIDI and WAV PCM/IMA ADPCM playback:

```
w 3 c000    # SCI_CLOCKF   4X clock, XTALI = 12.288 MHz
w 0 0c04    # SCI_MODE     Reset software
```

If you are playing anything else than MIDI files, you may set CLOCKF to 2X (0x4000). If XTALI $\neq$ 12.288 MHz, set SCI_CLOCKF accordingly.

After performing these commands, you may feed MIDI and WAV PCM/IMA ADPCM files as usual through SDI.

2.3 Recording IMA ADPCM Files

2.3.1 Activating IMA ADPCM Recording

To activate IMA ADPCM recording, perform the following sequence:

```
w 3 8000    # SCI_CLOCKF   0x8000 = 3X clock, XTALI = 12.288 MHz
w c 0012    # SCI_AICTRL0  At 3x12.288 MHz, result is 8 kHz (after downsample)
w d 0000    # SCI_AICTRL1  constMultip, 1024=1.0, if 0 then autogain is used
w e 0000    # SCI_AICTRL2  maxMultip,   1024=1.0, special value 0=4.0
w 0 8c04    # SCI_MODE     Record from mic, new+sharedmode, sw reset
```

First SCI_CLOCKF must be set. 3X is enough for ADPCM encoding. Again, if XTALI $\neq$ 12.288 MHz, set SCI_CLOCKF accordingly.

Before activating ADPCM recording, user should write a clock divider value to SCI_AICTRL0. The sampling frequency is calculated from the following formula: $f_s = \frac{F_c}{256 \times d}$, where F_c is the internal clock (CLKI) and d is the divider value in SCI_AICTRL0. The lowest valid value for d is 4. If SCI_AICTRL0 contains 0, the default divider value 12 is used.

Examples:
$F_c = 3.0 \times 12.288\,\text{MHz}$, $d = 18$. Now $f_s = \frac{3.0 \times 12288000}{256 \times 18} = 8000\,\text{Hz}$.
$F_c = 2.5 \times 14.745\,\text{MHz}$, $d = 18$. Now $f_s = \frac{2.5 \times 14745000}{256 \times 18} = 8000\,\text{Hz}$.
$F_c = 3.0 \times 13.000\,\text{MHz}$, $d = 19$. Now $f_s = \frac{3.0 \times 13000000}{256 \times 19} = 8018\,\text{Hz}$.
$F_c = 3.0 \times 12.288\,\text{MHz}$, $d = 9$. Now $f_s = \frac{3.0 \times 12288000}{256 \times 9} = 16000\,\text{Hz}$.
$F_c = 2.5 \times 14.745\,\text{MHz}$, $d = 9$. Now $f_s = \frac{2.5 \times 14745000}{256 \times 9} = 16000\,\text{Hz}$.
$F_c = 3.5 \times 13.000\,\text{MHz}$, $d = 11$. Now $f_s = \frac{3.5 \times 13000000}{256 \times 11} = 16158\,\text{Hz}$.

Automatic Gain Control (AGC) may be controlled with SCI_AICTRL1 and SCI_AICTRL2. By setting SCI_AICTRL1 to a non-zero value AGC is turned off. When SCI_AICTRL1 is zero, SCI_AICTRL2 sets the maximum gain that is allowed for AGC. The default value of 4 (corresponds to register values 0 and 4096) gives 12 dB auto gain which usually doesn't generate too much noise for most applications. The best values depend on the application.

After performing these commands you should be able to hear the sound from the line input or microphone (depending on SCI_MODE bit SM_LINE_IN).

2.3.2 Reading IMA ADPCM Data

After IMA ADPCM recording has been activated, data is read through registers SCI_IN0 and SCI_IN1.

The IMA ADPCM sample buffer is 128 16-bit words as opposed to VS1003's 256 words. The fill status of the buffer can be read from 8 MSb's of SCI_HDAT1. If SCI_HDAT1[15:8] $\neq 0$, you can read `(SCI_HDAT1 >> 8) & 0xFF` 16-bit words from SCI_HDAT0. If the data is not read fast enough, the buffer overflows and returns to empty state.

Note: if SCI_HDAT1 ≥ 112, it may be better to wait for the buffer to overflow and clear before reading samples. That way you may avoid buffer aliasing.

Each IMA ADPCM block is 128 words, i.e. 256 bytes. If you wish to interrupt reading data and possibly continue later, please stop at a 128-word boundary. This way whole blocks are skipped and the encoded stream stays valid.

The 128 words (256 bytes) of an ADPCM block are read from SCI_IN0 and written into file as follows. The high 8 bits of SCI_IN0 should be written as the first byte to a file, then the low 8 bits. Note that this is contrary to the default operation of some 16-bit microcontrollers, and you may have to take extra care to do this right.

A way to see if you have written the file in the right way is to check bytes 2 and 3 (the first byte counts as byte 0) of each 256-byte block. Byte 3 should always be zero.

An example pseudo-code read loop is provided below.

```
static unsigned char b[256];
int blockAlign = 0;
FILE *fp = OpenWriteFile();
WriteHeader(fp);
while (!outOfWav) {
  int bytes = (ReadCmd(port, SCI_HDAT1) >> 8) << 1;
  if (bytes >= 128) {
    int i;
    for (i=0; i<128; i+=2) {
      int t = ReadCmd(port, SCI_HDAT0);
      b[blockAlign*128+i  ] = t >> 8;
      b[blockAlign*128+i+1] = t & 0xFF;
    }
    blockAlign++;
  }
  if (blockAlign >= 2) {
    fwrite(b, 1, 256, fp); // Write 256 bytes,
    blockAlign = 0;        // i.e. one IMA ADPCM block at a time
  }
}
```

2.3.3 Adding a RIFF Header

To make your IMA ADPCM file a RIFF / WAV file, you have to add a header before the actual data. Note that 2- and 4-byte values are little-endian (lowest byte first) in this format:

File Offset	Field Name	Size	Bytes	Description
0	ChunkID	4	`"RIFF"`	
4	ChunkSize	4	F0 F1 F2 F3	File size - 8
8	Format	4	`"WAVE"`	
12	SubChunk1ID	4	`"fmt "`	
16	SubChunk1Size	4	0x14 0x0 0x0 0x0	20
20	AudioFormat	2	0x11 0x0	0x11 for IMA ADPCM
22	NumOfChannels	2	0x1 0x0	Mono sound
24	SampleRate	4	R0 R1 R2 R3	8 kHz: 0x1f40, 16 kHz: 0x3e80
28	ByteRate	4	B0 B1 B2 B3	8 kHz: 0x0fd7, 16 kHz: 0x1faf
32	BlockAlign	2	0x0 0x1	0x100
34	BitsPerSample	2	0x4 0x0	4-bit ADPCM
36	ByteExtraData	2	0x2 0x0	2
38	ExtraData	2	0xf9 0x1	Samples per block (505)
40	SubChunk2ID	4	`"fact"`	
44	SubChunk2Size	4	0x4 0x0 0x0 0x0	4
48	NumOfSamples	4	S0 S1 S2 S3	
52	SubChunk3ID	4	`"data"`	
56	SubChunk3Size	4	D0 D1 D2 D3	Data size (File Size-60)
60	Block1	256		First ADPCM block
316	...			More ADPCM data blocks

If we have n audio blocks, the values in the table are as follows:
$F = n \times 256 + 52$
$R = F_s$ (see Chapter 2.3.1 to see how to calculate F_s)
$B = \frac{F_s \times 256}{505}$
$S = n \times 505$. $D = n \times 256$

If you know beforehand how much you are going to record, you may fill in the complete header before any actual data. However, if you don't know how much you are going to record, you have to fill in the header size datas F, S and D after finishing recording.

Note: These are exactly the same operations that are done with all VS10XX chips.

2.4 Exiting the Patch

To exit the patch, set register 0 (SCI_MODE) bit 4 (SM_PDOWN). Instead of setting VS1103b to software powerdown mode, this will deactivate the patch and perform a full soft reset. You may choose to perform this operation between songs or if playback for some reason fails (DREQ keeps constantly up or down).

To read how to re-activate the patch, see Chapter 2.1.

Example:
```
w 0 0c10    # SCI_MODE     Deactivate patch, keep newmode
```

3 SCI Register Usage

This chapter described how SCI registers are used while running the patch.

3.1 Register SM_MODE

The following bits of register SCI_MODE work as documented in the VS1103 datasheet: SM_DIFF, SM_DACT, SM_SDIORD, SM_SDISHARE, SM_SDINEW, SM_EARSPEAKER, SM_LINE_IN and SM_ADPCM.

SM_RECORD_PATH, SM_TESTS and SM_ICONF are not used.

SM_RESET does not perform a full sofware reset, but a more limited patch reset.

SM_OUTOFMIDI is used to cancel current playback regardless of whether a WAV PCM/IMA ADPCM or a MIDI file is played.

SM_PDOWN is used for exiting the patch and performing a full soft reset, see Chapter 2.4.

3.2 Other Registers

The following registers work as documented in the VS1103 datasheet: SCI_STATUS, SCI_BASS, SCI_CLOCKF, SCI_AUDATA, SCI_WRAM, SCI_WRAMADDR, SCI_AIADDR, SCI_VOL and SCI_AICTRL3.

Register SCI_DECODE_TIME shows decode time for both MIDI and ADPCM files. It is not automatically cleared between songs, but the user may clear it.

Registers SCI_IN0 and SCI_IN1 work described in Chapter 2.3.2.

Registers SCI_MIXERVOL/SCI_AICTRL0, SCI_ADPCMRECCTL/SCI_AICTRL1 and SCI_AICTRL2 work as described in Chapter 2.3.1.

4 How to Load a Plugin

4.1 How to Load a .PLG File

A plugin file (.plg) contains a data file that contains one unsigned 16-bit vector called plugin. The file is in an interleaved and RLE compressed format. An example of a plugin vector 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 table has been exhausted.

The example vector 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 vector is in vector `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);
      }
    }
  }
}
```