VS1063A SCIDECODER

"VLSI Solution Audio Decoder/Encoder"

Project Code: VS1063
Project Name: Support

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

Revision History			
Rev.	**Date**	**Author**	**Description**
0.8	2011-10-24	POj	Initial version.

Contents

List of Figures

1 VS1063a SDI Encoder

1.1 Description

VS1063 decodes multiple audio formats to its built-in DAC's and to I2S output, but sometimes you want to decode audio without playing it. This application is created for you. This application receives any decodable file from SDI and decodes it to be read through SCI in WAV format.

Several WAV output formats are supported: PCM, μlaw, Alaw, G.722 ADPCM, and IMA ADPCM.

Note that the output data rate does not in any way depend on the samplerate of the original audio. It only depends on how fast the audio can be decoded (and encoded) and how quickly you can provide data to SDI and read from SCI. If you have a high internal clock configured in CLOCKF, you can usually decode faster than real-time.

1.2 Files

Chip	File	IRAM	Description
VS1063A	vs1063a-scidecoder.c	0x50..0x3cb	old array format
VS1063A	vs1063a-scidecoder.plg	0x50..0x3cb	compressed plugin
VS1063A	vs1063a-scidecoder.cmd	0x50..0x3cb	legacy command format

The application only works with VS1063A.

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.

Unless otherwise specified, this patch is **not** compatible with other vs1063a plugins. You need to give a software reset and load the right code when you switch between different plugins.

1.3 Additional Information

If you find a bug, a curious feature, or are unsure how to use VS1063, let us know.

VSDSP Forum is at **http://www.vsdsp-forum.com/** .

Support e-mail address is **support@vlsi.fi** .

1.4 Usage

SCI Registers Used	
Register	**Description**
AICTRL0	Use to read the WAV file data
AICTRL1	Tells the number of available words you can read from AICTRL0.
AICTRL2	Incremented if samplerate changes during decoding.
AICTRL3	Selects channel configuration and the WAV encoding format to use.

AICTRL3 Bits Supported		
Bits	**Name**	**Description**
0..3	Channel mode	0 is stereo, 2 is left, 3 is right, 4 is mono downmix, 7 is automatic mode
4..7	Encoding mode	0 to 4 are valid (ADPCM, PCM, μlaw, Alaw, G722)
10	AICTRL3_NORIFF	When set does not create WAV header(s).

How to Decode to WAV Formats

1. Send the application from .plg or .c file.
2. Write the encoding format to SCI_AICTRL3.
3. Start plugin by writing 0x50 to SCI_AIADDR.
4. Then send the audio file to SDI.
   - Send data to SDI 32 bytes at a time whenever DREQ is high.
   - SCI_AICTRL1 becomes non-zero when encoded data is available. Then read the indicated number of encoded words from SCI_AICTRL0.
5. Use the normal end-of-file procedure, then read the remaining words.
6. Fix the RIFF WAV file size and data size fields.

Note: if samplerate changes during decoding, a new WAV header is generated and AICTRL2 is incremented. So, when you see AICTRL2 changing value, you can start to monitor the data for “RIFF” and begin a new file, read and set a new rate for your output, or something else appropriate depending on your application.

Example Pseudo-Code (untested)

```
unsigned char buffer[512], outbuffer[512];

Mp3WriteReg(SCI_MODE, Mp3ReadReg(SCI_MODE) | SM_RESET);
while (!DREQ)
  ;
LoadPlugin();
Mp3WriteReg(SCI_AICTRL3, ENCMODE_PCM | 7); /*PCM, automatic channel mode*/
Mp3WriteReg(SCI_AIADDR, 0x50);             /*start plugin*/
while (1) {
  unsigned short avail;
  if (DREQ /*is high -- send more data to decode*/) {
    if (pos < filesize) {           /*if file data left*/
      Mp3SendData(buffer+cnt, 32);
      pos += 32;
      cnt += 32;
      if (cnt >= 512) {
        cnt = 0;
        GetNextSector(buffer);
      }
    } else {
      if (cnt == 0) {
        Mp3WriteReg(SCI_WRAMADDR, 0x1e29); /*resync*/
        Mp3WriteReg(SCI_WRAM, 0);          /* off */
      }
      Mp3SendEndFillBytes(32);  /*send endfillbytes to flush SDI FIFO*/
      cnt += 32;
      if (cnt == 2048) {        /*when all file data flushed, send cancel */
        Mp3WriteReg(SCI_MODE, Mp3ReadReg(SCI_MODE) | SM_CANCEL);
      }
    }
  }
  avail = Mp3ReadReg(SCI_AICTRL1); /*check available encoded words*/
  while (avail) {
    unsigned short data = Mp3ReadReg(SCI_AICTRL0);
    avail--;
    outbuffer[outcnt++] = data>>8;
    outbuffer[outcnt++] = data;
    if (outcnt >= 512) {
      outcnt = 0;
      SendNextSector(outbuffer); /* save a full sector of data */
    }
  }
  if (pos >= filesize && cnt > 14000 && Mp3ReadReg(SCI_HDAT1) == 0) {
    /*encoded the whole file, leave while loop.*/
    SendNextSector(outbuffer); /* save the partially filled sector */
    break;
  }
}
FixWavHeader(); /* If needed, fix RIFF size and data chunk size fields. */
```

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.