VSRV VSOS AUDIO SUBSYSTEM

VSRV1

Revision History

Rev.	Date	Author	Description
0.04	2025-06-17	HV	Added auieth driver, first public prerelease.
0.03	2025-06-09	HH	Initial internal prerelease.

Contents

List of Figures

1 Introduction

There are two sides to the VSRVES01 prototype chip: the VSDSP^6 side running VLSI Solution's own real-time operating system VSOS, and the RISC-V side capable of running Linux. VSDSP^6 acts as the boot processor and can load Linux to the RISC-V side.

The audio subsystem on the VSRVES01 prototype chip is handled by the VSDSP^6/VSOS side. This document describes the versatile audio drivers that the VSPDSP^6/VSOS offers and explains how to use the drivers to your best advantage.

After the disclaimer and definitions in Chapters 2 and 3, an overview of the Audio subsystem is given in Chapter 4, *Overview*, followed by requirements in Chapter 5, *Requirements*.

The VSRV VSOS audio subsystem is presented in Chapter 6, *The VSRV VSOS Audio Subsystem*.

The currently existing audio drivers are presented in Chapter 7, *Audio Drivers*, followed by a presentation of the currently existing filters in Chapter 8, *Audio Driver Filters*, and control programs in Chapter 9, *Audio Control Programs*,

Some examples on how to start audio drivers from startup.txt or the VSOS Shell are shown in Chapter 10, *Configuration Examples*.

Chapter 12 shows how to control some aspects on audio using UiMessages, even if the program that is currently running doesn't have any audio controls.

Audio Decoders are presented in Chapter 13, and Audio Encoders in Chapter 14.

The document ends with Chapter 15, *Latest Document Version Changes*, and Chapter 16, *Contact Information*.

2 Disclaimer

VLSI Solution makes everything it can to make this documentation as accurate as possible. However, no warranties or guarantees are given for the correctness of this documentation.

3 Definitions

DSP Digital Signal Processor.

I-mem Instruction Memory.

LSW Least Significant (16-bit) Word.

MSW Most Significant (16-bit) Word.

RISC Reduced Instruction Set Computer.

VS_DSP^6 VLSI Solution's DSP core.

VSIDE VLSI Solution's Integrated Development Environment.

VSOS VLSI Solution's Operating System.

X-mem X Data Memory.

Y-mem Y Data Memory.

4 Overview

The VSOS Audio Subsystem provides numerous drivers to handle the many audio Input/Output options of VSRV. The audio drivers can be controlled either with ioctl() calls from the C language, or from VSOS Shell control program.

While instructions for how to use each audio driver are provided in the README.TXT or documentation .PDF files of the drivers, this document will provide an overview of the capabilities of the drivers. However, for details, refer to documentation of the audio drivers themselves.

5 Requirements

To test the audio drivers in this document, you need to have the following building blocks:

- VSRV1 CAT Board or something similar.
- Latest version of VSOS files downloaded from the VSDSP Forum web site.
- UART or USB->UART cable connected between DevBoard and PC for using the UART interface. Data speed is 115200 bps, format is 8N1.
- Your favorite UART Terminal Emulation program installed on the PC. Read the document "VSRV VSOS Shell" for further details.

When all of this is in order, you are ready to test the VSOS Audio Subystem.

6 The VSRV VSOS Audio Subsystem

At startup, and without loading any driver, VSOS does not offer audio capabilities. However, by loading drivers like AUIADC.DR3 (Chapter 7.4.1) and AUODAC.DR3 (Chapter 7.2.1), it is easy to add audio capabilities to VSOS.

When audio drivers have been installed, VSOS Audio makes it very easy to produce sound with its standard-C-like standard audio interface (Chapter 6.1, *Standard Audio*). Instead of being forced to use audio-specific I/O routines, audio looks just like files.

More complex audio operations and redirections can be done using a combination of audio drivers, described in Chapter 7, *Audio Drivers*.

6.1 Standard Audio

VSOS offers the user a standard audio source and destination, although the audio source is only activated if an appropriate audio input driver is loaded (Chapter 7). Called *stdaudioin* and *stdaudioout*, standard audio file handles are to sound much like *stdin* and *stdout* are to standard input and output in standard C. It is not allowed for the user to close standard audio input or output files, but the user may modify their parameters.

Both standard audio input and output open in stereo, 16-bit, 48 kHz mode. These parameters can be changed by the user, with driver and hardware dependent limitations.

The user may use all standard read and write operations to read from and write to standard audio. It is, however, required that fread() / fwrite() functions are used instead of character-based operations like fgetc() and fprintf(). For efficiency reasons, it is recommended to handle larger chunks of samples, like 32, at a time.

Stereo samples are stored in an interleaved fashion. In 32-bit mode, the least significant word is stored first. This is the same as the native VSDSP 32-bit word order.

Audio sample buffer 16-bit word order				
Audio format	**Word 0**	**Word 1**	**Word 2**	**Word 3**
16-bit stereo	Left	Right		
32-bit stereo	Left LSW	Left MSW	Right LSW	Right MSW

6.2 VSOS Audio Output Example Program

The following audio program example creates a low-intensity sine wave to the left channel, then outputs the samples.

```
#include <vo_stdio.h>
#include <stdlib.h>
#include <math.h>
#include <saturate.h>
#include <apploader.h>

#define SIN_TAB_SIZE 96
#define SIN_AMPLITUDE 1000 /* Max 32767 */

static const s_int16 __y sinTab[SIN_TAB_SIZE];

int main(void) {
  // Remember to never allocate buffers from stack space. So, if you
  // allocate the space inside your function, never forget "static"!
  static s_int16 myBuf[2*SIN_TAB_SIZE];
  int i;

  /* Build sine table */
  for (i=0; i<SIN_TAB_SIZE; i++) {
    sinTab[i] = (s_int16)(sin(i*2.0*M_PI/SIN_TAB_SIZE)*SIN_AMPLITUDE);
  }

  while (1) {
    // Clear buffer
    memset(myBuf, 0, sizeof(myBuf));

    // Create sine wave to the left channel.
    for (i=0; i<SIN_TAB_SIZE; i++) {
      myBuf[i*2] = sinTab[i];
    }

    // Write result
    fwrite(myBuf, sizeof(s_int16), 2*SIN_TAB_SIZE, stdaudioout);
  }

  // Not really needed because there was a while(1) before
  return EXIT_SUCCESS;
}
```

6.3 VSOS Audio Input/Output Example Program

The following audio program reads audio from the default input, and sends it to the default output, until the user pushes Ctrl-C in the VSOS Shell Environment.

```
#include <vo_stdio.h>
#include <apploader.h> // Contains LoadLibrary() and DropLibrary()
#include <consolestate.h>

#define BUFSIZE 128

ioresult main(char *parameters) {
  static s_int16 myBuf[BUFSIZE];

  if (!stdaudioin || !stdaudioout) {
    printf("E: NO AUDIO IN OR OUT!\n");
    return S_ERROR;
  }

  while (!(appFlags & APP_FLAG_QUIT)) { /* Until Ctrl-C is pushed */
    fread(myBuf, sizeof(s_int16), BUFSIZE, stdaudioin);
    fwrite(myBuf, sizeof(s_int16), BUFSIZE, stdaudioout);
  }

  return S_OK;
}
```

7 Audio Drivers

VSRV has multiple hardware audio paths, and software Audio Drivers that are meant to interface between the user and hardware. The Audio Drivers offer a consistent interface to the user so that most of the time the User Application doesn't need to know which audio driver it is interfacing with.

This Chapter will explain which driver you will need to attach each audio driver to your software.

7.1 General

With few exceptions, VSRV VSOS audio drivers offer a 2-channel (stereo) input, and/or a 2-channel (stereo) output.

Most if not all audio drivers may be set either to 16-bit or 32-bit mode. The default is 16 bits. If the driver cannot receive or send its word length status, it is operating in 16-bit mode.

Many audio drivers allow for their sample rate to be set. If applicable for the audio driver, 48 kHz is typical default sample rate. If the driver cannot set or return its sample rate, it is typically operating at 48 kHz.

Audio drivers are named using the following format:
`AUdyyyyyz.DR3`
where

Symbol	**Description**
d	Driver direction: I = input, O = output, X = Input+Output
yyyyy	Driver name, max. 5 characters
z	Optional M or S if e.g. I2S driver is Master or Slave

There may be more than one audio driver in use at the time. Most audio drivers may be started as *system* drivers in which case they connect to file handles stdaudioin and/or stdaudioout for easy access from the User Application. If not started as system drivers, they must be accessed through explicit file handles.

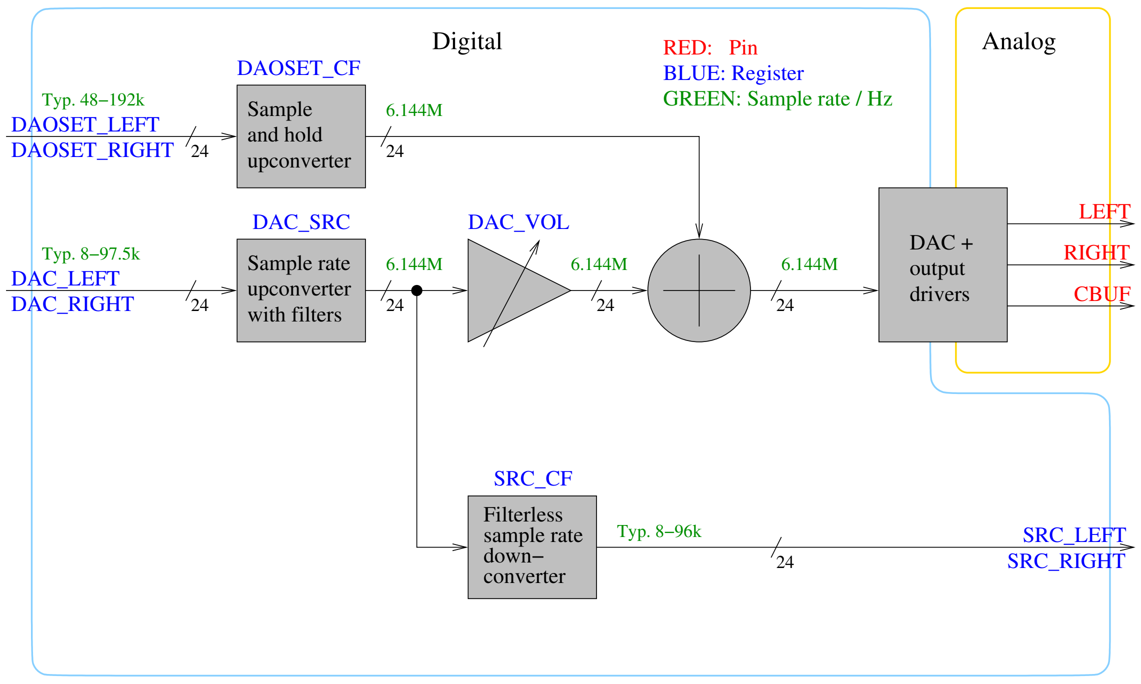

Figure 1: VSRVES01 playback (DA) audio paths

Figure 1 shows the VSRV hardware output audio paths. Most of these have a driver controlling them.

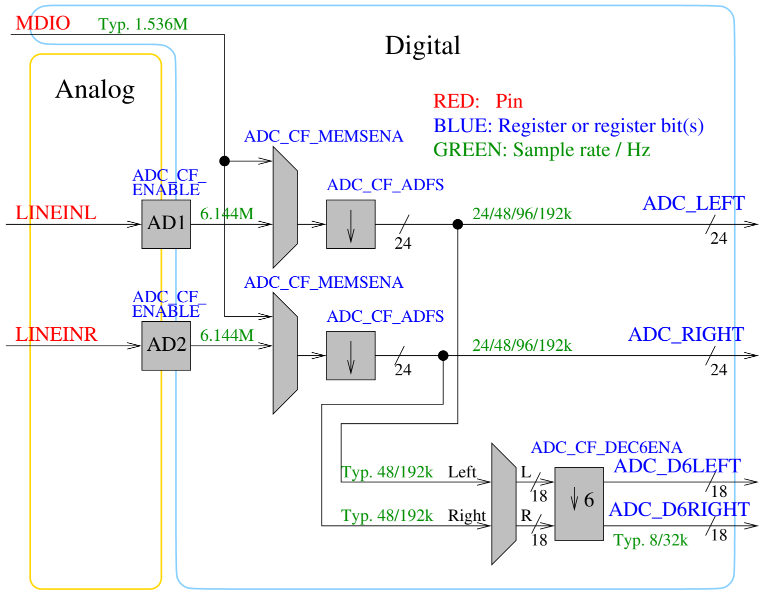

Figure 2: VSRVES01 recording (AD) signal paths

Figure 2 shows the VSRV hardware input audio paths. Most of these have a driver controlling them.

7.1.1 Examples of Loading and Unloading Audio Drivers

To load and activate the Analog Output DAC Audio Driver AUODAC.DR3 (Chapter 7.2), you may add the following line to S:STARTUP.TXT:

```
Driver +AUODAC s
```

Alternatively you may enter the same command on the VSOS Shell command line:

```
S:>driver +auodac s
```

Both of these methods will load and activate the audio driver. The parameter “s” stands for *system*, and it will make the audio driver automatically connect to stdaudioout. If this was an input driver, it would have connected to stdaudioin.

To verify that the driver has actually loaded and is running, you may run the following command on the VSOS Shell command line:

```
S:>auoutput
stdaudioout:      0x2056, auodac::audioFile=0x0c43(3139)
  ->Identify():   0x3c4f, auodac::Identify returns "AUODAC"
  ->op:           0x205f, auodac::audioFileOps=0x0000(0)
    ->Ioctl():    0x3b06, auodac::AudioIoctl
    ->Write():    0x3c05, auodac::AudioWrite
Sample rate:      48000
Bits per sample: 16
Channels:         unknown (assuming 2)
Buffer size:      512 16-bit words (256 16-bit stereo samples)
Buffer fill:      4 16-bit words (2 16-bit stereo samples)
Sample counter:   2102830
Underflows:       2100532
Volume:           +0.0 dB of maximum level
```

If you need to load another output audio driver, you can do it and check its status separately. Note that we already have a system driver for stdaudioout, so this driver will be started without the “s” system option. Below is a VSOS Shell command line example:

```
S:>driver +auooset
S:>auoutput -dauooset
audioFP:          0x241c, auooset::audioFile=0x0c43(3139)
  ->Identify():   0x436a, auooset::Identify returns "AUOOSET"
  ->op:           0x2423, auooset::audioFileOps=0x0000(0)
    ->Ioctl():    0x423c, auooset::AudioIoctl
    ->Write():    0x4311, auooset::AudioWrite
Sample rate:      48000
Bits per sample: 16
Channels:         unknown (assuming 2)
Buffer size:      512 16-bit words (256 16-bit stereo samples)
Buffer fill:      4 16-bit words (2 16-bit stereo samples)
Sample counter:   351032
Underflows:       350780
Volume:           +0.0 dB of maximum level
```

To remove the previous driver, you may enter the following command:

```
S:>driver -auooset
S:>auoutput -dauooset
E: Library auooset was not already in memory
```

More examples on how to load and unload audio driver are provided in Chapter 10, *Configuration Examples*.

7.2 Analog Output DAC Audio Driver

7.2.1 Driver AUODAC.DR3

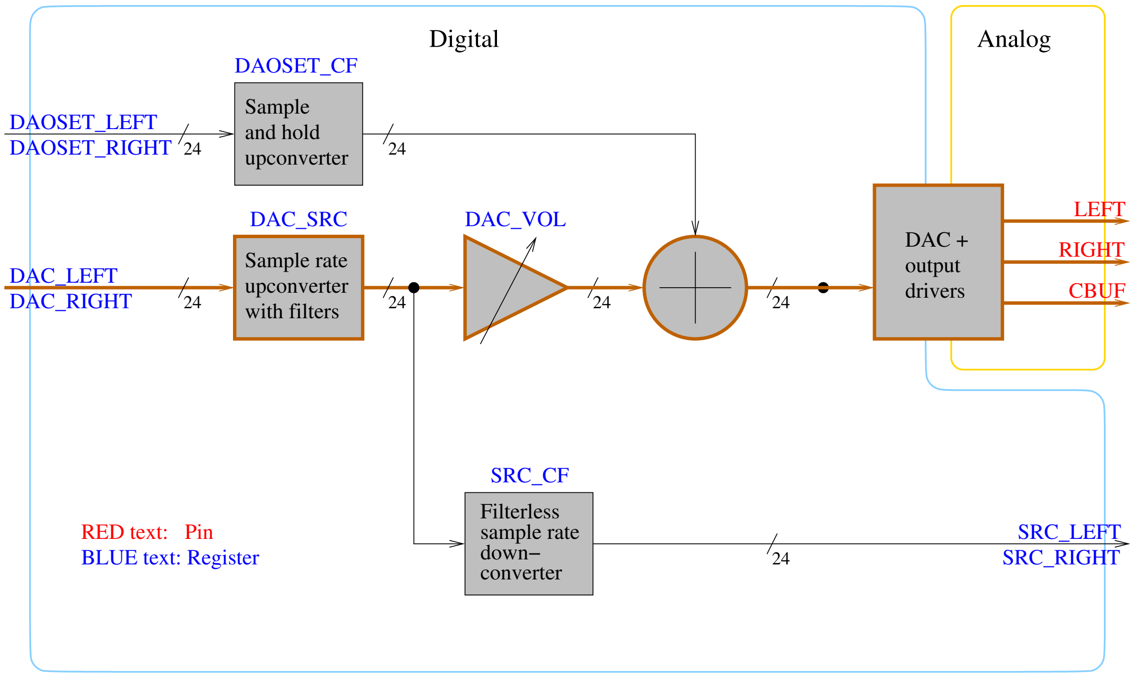

Figure 3: AUODAC.DR3 signal paths shown in **bold brown**

Figure 3 shows the VSRV high-quality, fully filtered analog output main audio path.

AUODAC.DR3 is the basic DAC output driver. It takes over the VSOS default driver and offers a lot of funtionality over it, like 16-bit and 32-bit data transfers. It takes over *stdaudioout*, so all software that writes to standard output will send audio to this driver.

The driver offers setting the sample rate with an approximately 0.01 Hz accuracy between 100 and 97500 Hz on VSRV. Audio is upconverted to an extremely high rate of 6.144 MHz by a high-quality hardware sample rate upconverter.

Playback volume can be set with 0.5 dB accuracy between full level volume (-0 dB) and -127 dB.

7.3 Analog Output Side Path Audio Driver

7.3.1 Driver AUOOSET.DR3

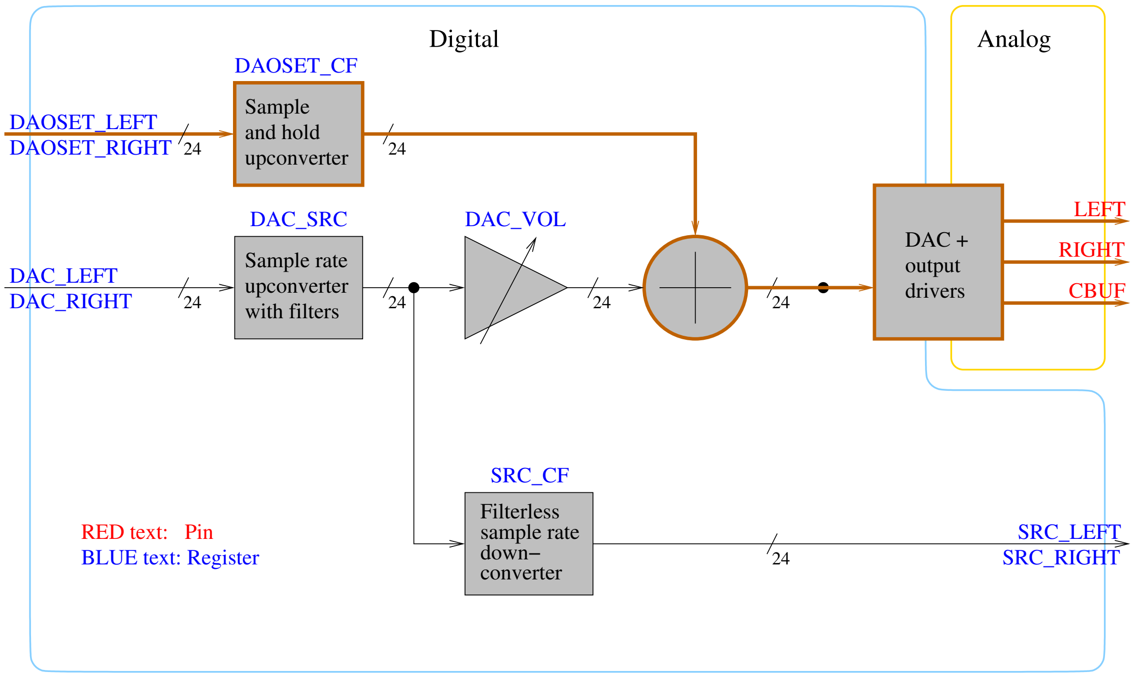

Figure 4: AUOOSET.DR3 signal paths shown in **bold brown**

Figure 4 shows the VSRV analog output audio side path. This audio path is not filtered; it is only put through a Sample and hold upconverter. As such, audible aliasing distortion may be heard if low sample rates are used. This audio path is best suitable for different kinds of alarm and effects sounds that may easily be independently overlayed on top of the audio of the main audio path (see Chapter 7.2.1).

The sample rate of the side audio path is independent from the main audio path. While it may be set to up to 192 kHz, all sample rates cannot be set accurately. While certain sample rates like 24, 48, and 96 kHz can be played accurately, some others, like 44.1 kHz, may have an up to 150 Hz error. While not a problem for effects sounds, this may be create issue with accurate timing when playing longer audio passages.

While there is no hardware volume control for the side audio path, the driver offers an equivalent software volume control.

7.4 Analog Input ADC Audio Driver

7.4.1 Driver AUIADC.DR3

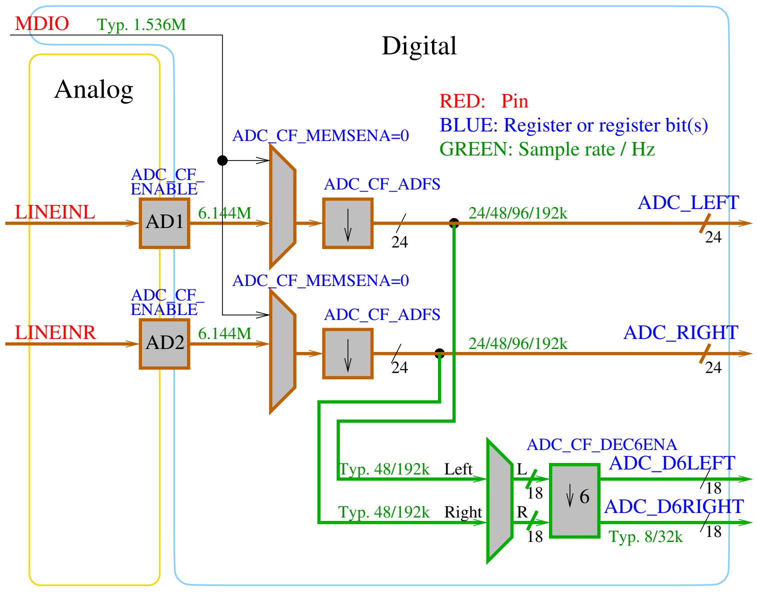

Figure 5: Default AUIADC.DR3 input signal path shown in **bold brown**. Additional, optional output path that goes through an additional decimator-by-6 shown in **bold green**.

By default, the AUIADC.DR3 driver reads an analog input, outputs its data to either one or two destinations, as shown in Figure 5.

Supported sample rates are 192, 96, 48, and 24 kHz. However, it is also possible to use a high-quality down-by-6 decimator for sample rates 32, 16, 8, and 4 kHz. When such a sample rate is selected, the driver automatically redirects itself to read its samples from registers ADC_D6LEFT/ADC_D6RIGHT instead of the default ADC_LEFT/ADC_RIGHT.

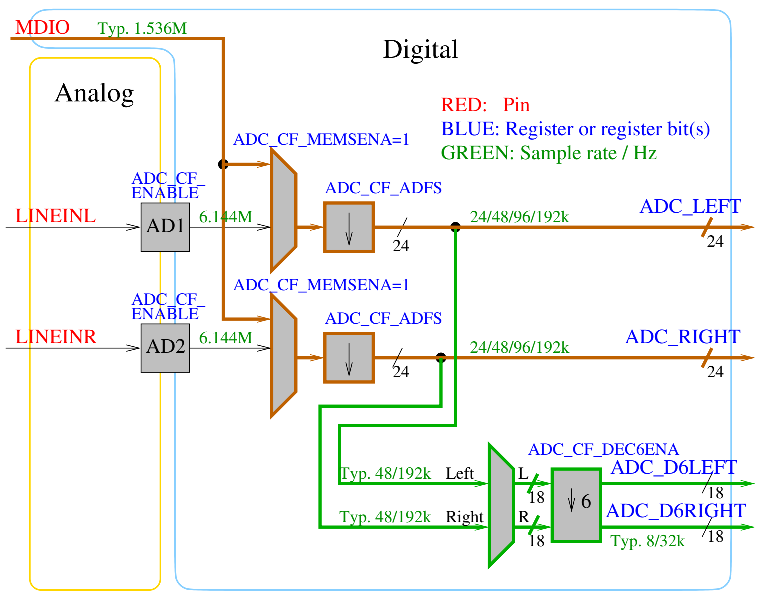

Figure 6: MEMS Mic AUIADC.DR3 input signal path shown in **bold brown**. Additional, optional output path that goes through an additional decimator-by-6 shown in **bold green**.

Alternatively, the driver can take its input from digital MEMS microphones, as shown in Figure 6. Selecting the MEMS input is not supported by the driver, but must be set by the user program by setting or clearing register ADC_CF bit ADC_CF_MEMSENA.

Input parameters can be controlled using the AUINPUT program (Chapter 9.1).

AUIADC.DR3 Command Line Parameters

If “s” is provided as the first command line parameter, then the driver will become a *system* driver, i.e. it will connect to the standard file handle *stdaudioin* so that all software may automatically connect to it.

Example; Start the driver and connect it to stdaudioin:

```
S:>Driver +AUIADC s
```

7.5 Ethernet Input Audio Driver

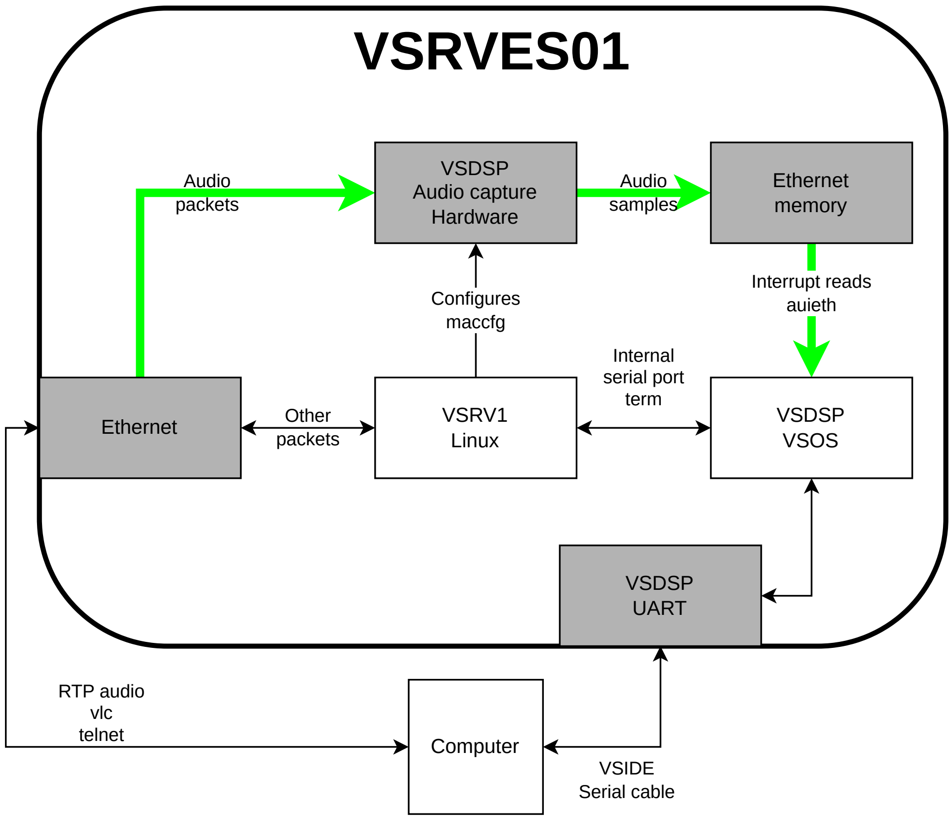

Figure 7: Simplified VSDSP audio capture hardware operation

The ethernet audio input requires co-operation from Linux side. Linux has to configure the ethernet hardware to pass specific packets as audio packets to VSDSP. The operation of the ethernet hardware is shown in figure 7. The computer to control the chip is also shown.

The usage process is

1. Load auieth driver
2. Load Linux using DDRLoad
3. Get access to Linux side of the chip
4. Configure ethernet on Linux side
5. Consume receiver audio

The first thing is to load the auieth driver with the Driver program before loading Linux.

After Linux has loaded, connect to it for example with telnet or Term and as a root user configure the ethernet with maccfg

Consult RISC-V Linux documentation how to configure the network to work with VSDP. For a usage example please refer to *VSRV User's Guide* section *Running demo programs on the CAT Board*.

7.5.1 Driver AUIETH.DR3

Initialization parameters

If “s” is provided as the first command line parameter, then the driver will become a *system* driver, i.e. it will connect to the standard file handle *stdaudioin* so that all software may automatically connect to it.

```
S:>Driver +AUIETH s
```

Monitor parameters

The auieth driver has some debugging aids. After it has been loaded into memory as a driver, you can also run it on the command line with different command line parameters and get information what is happening if audio playback has problems.

The monitor parameters are opionated order from user friendly to hard problem solving.

-v Print reception statistics once and exit

-m Monitor statistics counter.

-l Read and calculate packets as they are received. Consume the audio and calculate samplerate.

-d Dump the latest packets information

-r Dump the header in raw words in a loop

-w Write the output to s:pckts.txt file

The -w parameter is used in combination with -d and -r usually.

The statistics fields on the line are:

IC interrupt count

S sample count

Bad Not good packets

LE Last error in packet

ROF Receiver over flow

OF Input buffer overflow

IH Interrupt to handler clock cycle delay (interrupt duration in COPY_AUDIO_INTERRUPT configuration)

CT Receiver to buffer copy time in clock cycles

F Input buffer fill

The -l parameter prints little bit different information giving calculated samplerate, adjustment to it and how many RTP packets are received out of order.

Out of order packets are sent to audiobuffer in the order they are receivede and will make noise if there are plenty of them.

Known issues

2025-06-16

Receiving audio packets while auieth driver hasn't been loaded, can stop the ethernet audio reception. The sympton of this problem is that only two interrupts are received and then the ethernet hardware jams. A workaround is to load the auieth driver when starting up (for example in startup.txt) and keep it in memory.

Dependencies

The auieth driver uses ParamSpl program to handle its parameters.

The auieth driver also requires running Linux system on VSRV. After the audio path has been set up, Linux is not actively needed.

7.6 Slave Audio Input Synchronization Driver

When inputting audio data in slave mode (using for example the I2S audio input slave driver AUII2SS.DR3 (not available on VSRVES01)), the exact sample rate of the audio is usually not known. Even if the nominal sample rate is known, mismatches between master transmitter and the VSRV receiver clock crystals causes there to always be a mismatch between them (example: transmitter nominally sends 48000 Hz, but because of a clock mismatch the receiver sees the data at 48002.3 Hz).

This speed mismatch will eventually cause an audio buffer underflow or overflow, which may cause audible clicks or other kinds of distortion.

The slave audio input synchronization drivers are intended to remove the synchronization issue.

7.6.1 Driver AUXSYNCS.DR3

The Slave Audio Input Synchronization Driver AUXSYNCS.DR3 synchronizes a slave audio input driver with the analog Earphone/Line Out driver AUODAC.DR3.

Before starting the Sync Driver, the user must first load and connect a slave audio input driver to *stdaudioin*, and the analog output driver to *stdaudioout*. When the driver is loaded, it will automatically adjust the analog output sample rate according to the input. The adjustment range is up to 97500 Hz, so standard sample rates up to 96 kHz can be received. The Sync Driver can dynamically change its sample rate if the input sample rate changes.

Example startup.txt file clip (AUII2SS.DR3 not available on VSRVES01):

```
# Load I2S Slave Input driver and make it stdaudioin
driver +AUII2SS s
# Load Line Out / Earphone output driver and make it stdaudioout
driver +AUODAC s
# Connect and synchronize stdaudioout with stdaudioin slave
driver +AUXSYNCS
```

The same can be done using the VSOS Shell using the following commands:

```
S:>driver +auii2ss s
S:>driver +auodac s
S:>driver +auxsyncs
```

AUXSYNCS.DR3 has been tested with the I2S Slave Input drivers, but it is designed to be usable with any generic slave input driver that offers a near-constant data rate. It may not work properly with input drivers with large data bursts, like what for example VLC creates when in Ethernet streaming mode. For that, there exists AuXPlayB, presented in Chapter 7.7.2.

7.7 Audio Input to Output Copying Drivers

Sometimes it's useful to play back audio data from an input to an output in the background. This can be done by an audio copying driver.

7.7.1 Loopback Driver AUXPLAY.DR3

The AUXPLAY.DR3 driver reads data from *stdaudioin* and copies it to *stdaudioout*. While seemingly trivial, it does so in the background, allowing the user to do other operations while sound is being played back.

Normally the driver reports to *stdout* if there are input buffer overflows or output buffer underflows. The amount of the overflows/underflows are given in stereo samples (so e.g. +4800 at a sample rate of 48000 means 1/10s). The reports use the following format:

```
AUXPLAY: In overflow +4088
AUXPLAY: Out underflow +4034
```

To disable overflow and underflow reporting, give the 'q' parameter when loading and starting AUXPLAY.DR3.

7.7.2 Loopback Slave Driver AUXPLAYB.DR3 with Big Buffer

Like the AUXPLAY.DR3 driver (Chapter 7.7.1), AUXPLAYB.DR3 reads data from *stdaudioin* and copies it to *stdaudioout*. However, at startup it will allocate itself large read and write buffers so as to be able to play audio back from a source that is not continuous. The buffer allocated is 49152 16-bit words, or just over 0.5 seconds when playback parameters are 48 kHz, stereo, 16-bit.

The driver always starts with a nominal sample rate of 48 kHz. Then, depending on the fill state of its audio buffer, it will minutely adjust the sample rate so as not to cause underflows or overflows. This adjustment is done in such small steps, with an accuracy of around 1/100 Hz, that it is absolutely beyond anyone's hearing threshold.

Normally the driver reports to *stdout* if there are input buffer overflows or output buffer underflows. The amount of the overflows/underflows are given in stereo samples (so e.g. +4800 at a sample rate of 48000 means 1/10s). The reports use the following format:

```
AUXPLAYB: In overflow +4088
AUXPLAYB: Out underflow +4034
```

To disable overflow and underflow reporting, give the 'q' parameter when loading and starting AUXPLAYB.DR3.

8 Audio Filter Drivers

Audio filter drivers connect to an audio source or sink, and offer additional functionality, like filtering.

Audio filter drivers are named using the following format:
`FTdyyyyy.DR3`
where

Symbol	Description
d	Driver direction: I = input, O = output, X = Input/Output
yyyyy	Driver name, max 5 characters

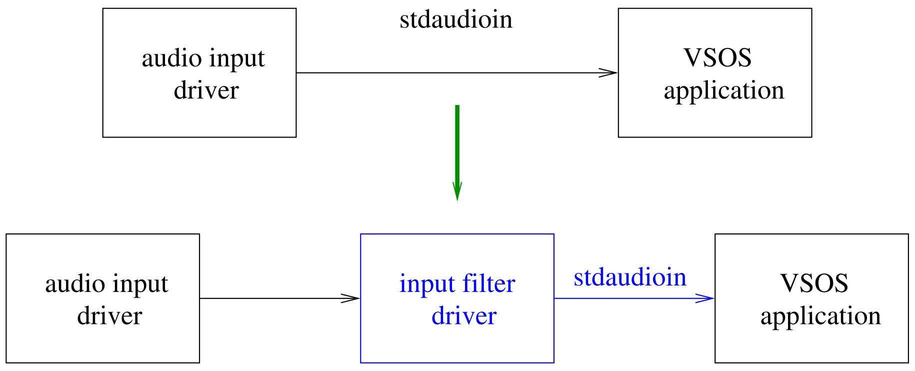

Figure 8: An input filter driver connects to the stdaudioin chain

All filter input drivers connect directly between the current *stdaudioin* program chain and the user program, as shown in Figure 8. The base driver responsible for *stdaudioin* (e.g. AUIADC.DR3) must be loaded before the filter driver.

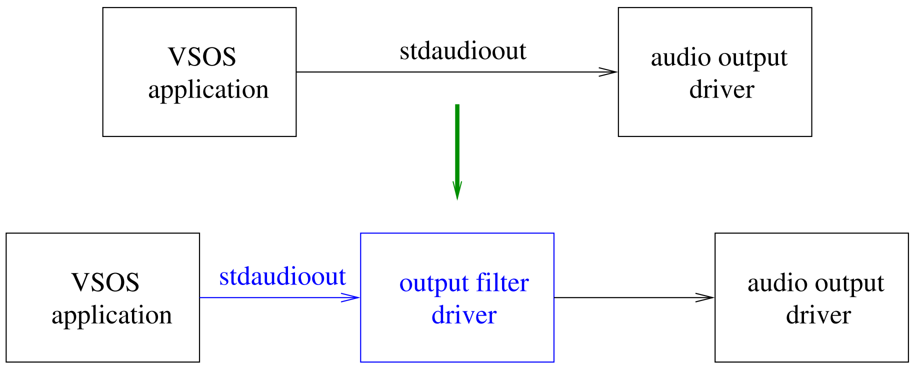

Figure 9: An output filter driver connects to the *stdaudioout* chain

All filter output drivers connect directly between the user program and the current *stdaudioout* program chain, as shown in Figure 9. The base driver responsible for *stdaudioout* (e.g. AUODAC.DR3) must be loaded before the filter driver.

Audio filters must be removed in the reverse order of loading.

Below is a command line example of audio and filter driver allocation and removal:

```
S:>auinput
E: No stdaudioin or NULL ptr
S:>driver +auiadc s
S:>auinput
stdaudioin:       0x2392, auiadc::audioFile=0x0c63(3171)
  ->Identify():   0x414e, auiadc::Identify returns "AUIADC"
  ->op:           0x2399, auiadc::audioFileOps=0x0000(0)
    ->Ioctl():    0x4023, auiadc::AudioIoctl
    ->Read():     0x4104, auiadc::AudioRead
Sample rate:      48000
Bits per sample: 16
Channels:         unknown (assuming 2)
Buffer size:      512 16-bit words (256 16-bit stereo samples)
Buffer fill:      508 16-bit words (254 16-bit stereo samples)
Sample counter:   51484
Overflows:        51232
S:>driver +ftiagc
S:>auinput
stdaudioin:       0x2541, FTIAGC::audioFile=0x0863(2147)
  ->Identify():   0x4644, FTIAGC::Identify returns "FTIAGC32"
  ->op:           0x2569, FTIAGC::audioFileOps=0x0000(0)
    ->Ioctl():    0x4508, FTIAGC::AudioIoctl
    ->Read():     0x4597, FTIAGC::AudioRead
[... rest of auinput printouts cut for brevity ...]
S:>driver -ftiagc
S:>auinput
stdaudioin:       0x2392, auiadc::audioFile=0x0c63(3171)
  ->Identify():   0x414e, auiadc::Identify returns "AUIADC"
  ->op:           0x2399, auiadc::audioFileOps=0x0000(0)
    ->Ioctl():    0x4023, auiadc::AudioIoctl
    ->Read():     0x4104, auiadc::AudioRead
[... rest of auinput printouts cut for brevity ...]
S:>driver -auiadc
S:>auinput
E: No stdaudioin or NULL ptr
```

8.1 Equalizer Filter Driver

The Equalizer Filter Drivers implements a high-quality, multiband equalizer to VSRV's output audio path.

The package itself contains detailed PDF documentation; please read it for details.

8.1.1 Driver FTOEQU.DR3

TBD.

8.1.2 Control Program SETEQU.DR3

TBD.

8.2 DC Offset / AGC Filter Drivers

When audio is digitized, two technical issues are DC Offset and Large Dynamic Range.

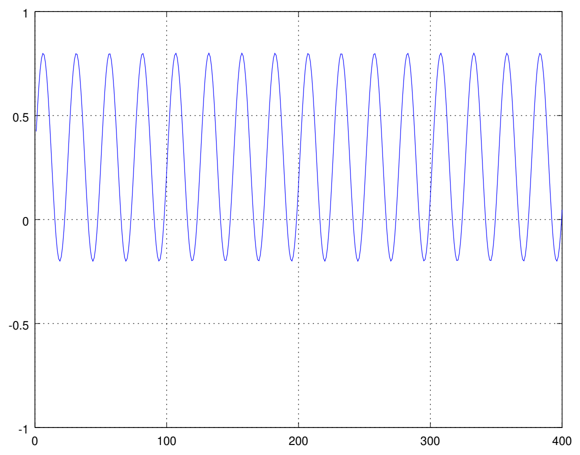

Figure 10: Audio with exaggerated DC offset

In an ideal world DC Offset wouldn't happen. However, in the real world, signals almost always have a slight DC offset. Note, how the sine wave in Figure 10 does not move evenly around the center point, but has an offset of about +0.35. While the figure has been greatly exaggerated, this is a real phenomenon caused by a myriad of different reasons.

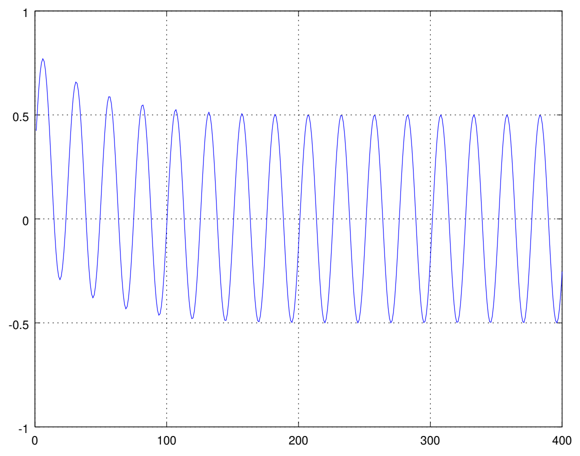

Figure 11: Audio with DC blocking

DC offset may cause many issues, including increased power consumption, audible cracks and pops, waring down of speaker elements, and non-ideal audio compression. Because of this, it is best to remove the audio offset with a DC Blocker algorithm, as shown in Figure 11. Notice how the offset disappear after a little while (in this case, it has vanished practically completely by sample 150).

Another issue in audio is excessive dynamic range. This is not a problem when recording well-mixed, pre-recorded music, but it may be a big issue when recording speech from the microphone. To compensate for the audio level differences of close and faw away speakers, and Automatic Gain Control (AGC) unit may often be useful. Note, however, that AGC should not be used for HiFi recording applications!

8.2.1 Driver FTIDCBL.DR3

TBD.

8.2.2 Driver FTIAGC.DR3

TBD.

8.2.3 Control Program SETAGC.DR3

TBD.

8.3 Pitch Shifter / Speed Shifter Filter Driver

The FtPitch package offers a pitch and speed shifter that connects to stdout. The speed shifter can nominally be controlled to speeds between 0.68x and 1.64x of realtime, and the pitch shifter can nominally be controlled between 0.61x and 1.47x of normal pitch.

Features and limitations:

- Speed divided by pitch (speed/pitch) must be between 0.68 and 1.64.
- Pitch shifting alters sample rate. If the resulting sample exceeds 96 kHz, playback will be at incorrect speed.
- The shifter has been optimized to work best for audio where sample rate is between 32 and 48 kHz.

8.3.1 Driver FTOPITCH.DR3

TBD.

8.3.2 Control Program SETPITCH.DR3

TBD.

8.4 Reverb Generator Filter Drivers

The FtRev package offers a reverb-style echo generator that connects to stdaudioin or stdaudioout. Many parameters of the Reverb Generator may be modified.

8.4.1 Driver FTIREV.DR3

TBD.

8.4.2 Driver FTOREV.DR3

TBD.

8.4.3 Control Program SETREV.DR3

TBD.

8.5 Noise Killer Filter Driver

Because of the way stereo information is transmitted on FM radio, stereo reception is always more suscept to white noise and other artifacts than mono reception. A way to reduce or remove the noise is to either dampen the stereo effect at the receiver, or to just turn FM stereo reception off. The FtNoiseKiller package offers an adaptive FM stereo radio noise killer algorithm that doesn't destroy the stereo image. This will help in creating a purely noiseless stereo radio experience.

8.5.1 Driver FTINOISE.DR3

TBD.

8.5.2 Control Program SETNOISE.DR3

TBD.

8.6 Mono / Differential Filter Drivers

TBD.

9 Audio Control Programs

These programs are useful for displaying and changing audio parameters, as well as debugging audio interfaces. They are parts of the AuControl solution.

9.1 Control Program AUINPUT.DR3

```
Usage: AuInput [-ddrv|-pfp|-rrate|-bbits|-sbufsize|-cch|chconf|-v|+v|-h]
-ddrv Connect to audio driver DRV.DR3
-pfp Set input audio driver pointer to fp (use with caution!)
-rrate Set sample rate to rate
-bbits Number of bits (16 or 32)
-sbufSz Set buffer size to bufSz 16-bit words
-cch Set number of channels to ch
-v|+v Verbose on|off
-h Show this help
```

AUINPUT lets the user display control several parameters of *stdaudioin*, or any unlocked audio input driver, or file pointer if it is known.

If called without any command line arguments that change a value, AUINPUT will display the status of the audio driver as shown below

```
S:>auinput
stdaudioin:       0x203a, auii2ss::audioFile=3171(0xc63)
  ->Identify():   0x3b4f, auxsyncs::Identify returns "AUXSYNCS"
  ->op:           0x2041, auii2ss::audioFileOps=0(0x0)
    ->Ioctl():    0x3992, auxsyncs::AudioIoctl
    ->Read():     0x38cf, auii2ss::AudioRead
Sample rate:      48000
Bits per sample:  16
Channels:         unknown (assuming 2)
Buffer size:      512 16-bit words (256 16-bit stereo samples)
Buffer fill:      508 16-bit words (254 16-bit stereo samples)
Sample counter:   235803492
Overflows:        123022
```

In this example, slave audio synchronization driver AUXSYNCS.DR3 (Chapter 7.6.1) has been loaded on top of AUII2SS.DR3, replacing two of its methods, Identify() and Ioctl().

Note: To display symbol information, AUINPUT requires library TRACE.DR3.

9.2 Control Program AUOUTPUT.DR3

```
Usage: AuOutput [-ddrv|-pfp|-rrate|-bbits|-sbufSize|-cch|-lvol|-v|+v|-h]
```

```
-ddrv Connect to audio driver DRV.DR3
-pfp Set output audio file pointer to fp (use with caution!)
-rrate Set sample rate to rate
-bbits Number of bits (16 or 32)
-sbufSz Set buffer size to bufSz 16-bit words
-cch Set number of channels to ch
-lvol Volume Level of maximum (vol = -128 .. 127.5)
-v|+v Verbose on|off
-h Show this help
```

AUOUTPUT lets the user display control several parameters of *stdaudioout*, or any unlocked audio input driver, or file pointer if it is known.

If called without any command line arguments that change a value, AUOUTPUT will display the status of the audio driver as shown below

```
S:>auoutput
stdaudioout:      0x1fea, auodac::audioFile=3139(0xc43)
  ->Identify():   0x3b4f, auxsyncs::Identify returns "AUXSYNCS"
  ->op:           0x1ff1, auodac::audioFileOps=0(0x0)
    ->Ioctl():    0x355b, auodac::AudioIoctl
    ->Write():    0x39fb, auxsyncs::AudioWrite
Sample rate:      47793
Bits per sample: 16
Channels:         unknown (assuming 2)
Buffer size:      4096 16-bit words (2048 16-bit stereo samples)
Buffer fill:      4 16-bit words (2 16-bit stereo samples)
Sample counter:   235977115
Underflows:       177796
Volume:           +0.0 dB of maximum level
```

In this example, slave audio synchronization driver AUXSYNCS.DR3 (Chapter 7.6.1) has been loaded on top of AUODAC.DR3 (Chapter 7.2.1), replacing two of its methods, Identify() and Write().

Note: To display symbol information, AUINPUT requires library TRACE.DR3.

10 Configuration Examples

Here are some configuration examples for loading different audio drivers.

For full options for each of these programs, have a look at the README.TXT / PDF file for each of the drivers.

10.1 Minimal startup.txt for Playback

```
# Audio DAC out driver
Driver +AUODAC s
```

Note the “s” parameter after the AUODAC.DR3 driver. This “s” parameter marks that the driver should become a *system* driver. In other words, it should connect to the system file stdaudioout so that all subsequent writes to stdaudioout will go through this driver.

10.2 Basic startup.txt for Recording

```
# Audio DAC out driver
Drover +AUODAC s
# Audio ADC in driver
Driver +AUIADC s
# DC Block; use at least this with analog input even if not using AGC
Driver +FTIDCBL
```

In this example, both AUODAC.DR3 and AUIADC.DR3 have the *system* “s” option, so they will occupy standard audio handles stdaudioout and stdaudioin, respectively.

10.3 Loading/Unloading Drivers Using the VSOS Shell

Using the VSOS Shell Environment, you can use the DRIVER.DR3 program to load drivers to memory, and to later unload them.

If possible, you should always unload drivers in the reverse order of loading them. This is particularly true with drivers that connect to other drivers, like AUXSYNCS which connects to both the *stdaudioin* and *stdaudioout* drivers (in this case AUII2SS and AUODAC, respectively), and AUXPLAY which also uses *stdaudioin* and *stdaudioout*

Example: to load and inload some drivers, run the following commands:

```
S:>driver +auii2ss s
S:>driver +auodac s
S:>driver +auxsyncs
S:>driver +auxplay
```

To unload the drivers, enter the following commands:

```
S:>driver -auxplay
S:>driver -auxsyncs
S:>driver -auodac
S:>driver -auii2ss
```

11 VSOS Audio ioctl() Controls

VSOS Audio Drivers can be controlled from C language using ioctl() controls declared in <aucommon.h>.

There are many more definitions in the #include file <aucommon.h>. Refer to the documentation of the specific drivers you use for exact details on what of these functions they support and how to get access to a file pointer for that driver.

The ioctl() function prototype is
`s_int16 ioctl(void *p, register int request, register char *arg);`
where `p` is the file or device pointer (e.g. `stdaudioin` or `stdaudioout`), `request` is the type of the request, and `arg` is the optional argument.

ioctl() returns `S_ERROR` (-1) for an error (there was an error in the parameters, or the ioctl() for the `request` doesn't exist in this driver), any other value for success.

Generally, for functions that set a value, if `arg` is a pointer or a 16-bit value, it is casted to c `char *` and passed to the function (e.g. IOCTL_AUDIO_SET_BITS in Chapter 11.2.5). If `arg` is a larger entity (e.g. 32-bit number), a pointer to the value is passed instead (e.g. IOCTL_AUDIO_SET_ORATE in Chapter 11.2.3).

Again, generally, for functions that return a 16-bit value where `S_ERROR` (-1) isn't included in the valid value range, the value is returned directly (e.g. IOCTL_AUDIO_GET_BITS in Chapter 11.2.4). Otherwise, the user needs to transmit a pointer to the return value in `arg` (e.g. IOCTL_AUDIO_GET_ORATE in Chapter 11.2.2). Not that in both cases ioctl() returns `S_ERROR` (-1) if there was an error in the call.

11.1 Resetting a Driver

11.1.1 IOCTL_RESTART

Restart driver. Normally this needs never be done.

Example:
```
  ioctl(fp, IOCTL_RESTART, NULL);
```

11.2 Controlling Sample Rate and Bit Width

11.2.1 IOCTL_AUDIO_SET_RATE_AND_BITS

Set sample rate times 256 and number of bits. This is the recommended way of setting the sample rate and bit width with drivers like e.g I2S where there is a limit to sample rate and bit width combinations. Note that the sample rate / bit width argument doesn't fit into 16 bits, so it needs to be passed through a pointer.

Some drivers have very restricted number of sample rates supported. If you want to see what sample rate actually was set by the hardware, it is recommended to do a IOCTL_AUDIO_GET_IRATE or IOCTL_AUDIO_GET_ORATE call to see what you actually got.

- labs(rateBitsX256) = sampleRateX256
- if rateBits < 0, then use 32-bit I/O
- Sets both input and output sample rate, if applicable
- Not available with Slave Mode drivers

Example:
```
s_int32 rateBitsX256 = -48000*256L; /* Set to 48000 Hz, 32 bits */
if (ioctl(fp, IOCTL_AUDIO_SET_RATE_AND_BITS, (char *)(&rateBits))) {
  printf("Couldn't set sample rate and bits\n");
}
```

11.2.2 IOCTL_AUDIO_GET_IRATE, IOCTL_AUDIO_GET_ORATE

Get current sample rate times 256. Note that sample rate doesn't fit into 16 bits, so it needs to be passed through a 32-bit pointer.

Some drivers have very restricted number of sample rates supported. If you want to see what sample rate actually was set by the hardware, it is recommended to do a IOCTL_AUDIO_GET_IRATE or IOCTL_AUDIO_GET_ORATE call to see what you actually got.

- Not available with Slave Mode drivers

Example for input driver:
```
s_int32 sampleRateX256;
if (ioctl(fp, IOCTL_AUDIO_GET_IRATE, (char *)(&sampleRateX256))) {
  printf("Couldn't get sample rate\n");
}
```

Example for output driver:
```
s_int32 sampleRateX256;
if (ioctl(fp, IOCTL_AUDIO_GET_ORATE, (char *)(&sampleRateX256))) {
  printf("Couldn't get sample rate\n");
}
```

11.2.3 IOCTL_AUDIO_SET_IRATE, IOCTL_AUDIO_SET_ORATE

Set sample rate times 256. Note that sample rate doesn't fit into 16 bits, so it needs to be passed through a 32-bit pointer.

- Only for Master Mode drivers
- It is recommended to use IOCTL_AUDIO_SET_RATE_AND_BITS instead

Example for input driver:

```
s_int32 sampleRateX256 = 48000*256L;
if (ioctl(fp, IOCTL_AUDIO_SET_IRATE, (char *)(&sampleRateX256))) {
  printf("Couldn't set sample rate\n");
}
```

Example for output driver:

```
s_int32 sampleRateX256 = 48000*256L;
if (ioctl(fp, IOCTL_AUDIO_SET_ORATE, (char *)(&sampleRateX256))) {
  printf("Couldn't set sample rate\n");
}
```

11.2.4 IOCTL_AUDIO_GET_BITS

Get number of bits for driver. If the driver only supports 16-bit mode, it may not implement this function call.

Example:

```
s_int16 bits = ioctl(fp, IOCTL_AUDIO_GET_BITS, NULL);
if (bits < 0) bits = 16;
```

11.2.5 IOCTL_AUDIO_SET_BITS

Set number of bits for driver.

Example:

- bits may be 16 or 32
- With Master Mode drivers it is recommended to use IOCTL_AUDIO_SET_RATE_AND_BITS instead

Example:

```
if (ioctl(fp, IOCTL_AUDIO_SET_BITS, (char *)(32))) {
  printf("Couldn't set bits\n");
}
```

11.3 Controlling Number of Audio Channels

These functions are relevant to drivers that can handle other than normal 2-channel audio. 2-channel drivers don't need to implement these function calls.

11.3.1 IOCTL_AUDIO_GET_ICHANNELS, IOCTL_AUDIO_GET_OCHANNELS

Get number of audio channels of a driver. If the driver only supports 2-channel audio, it may not implement this function call.

Example for input driver:

```
s_int16 channels = ioctl(fp, IOCTL_AUDIO_GET_ICHANNELS, NULL);
if (channels < 0) channels = 2;
```

Example for output driver:

```
s_int16 channels = ioctl(fp, IOCTL_AUDIO_GET_OCHANNELS, NULL);
if (channels < 0) channels = 2;
```

11.3.2 IOCTL_AUDIO_SET_ICHANNELS, IOCTL_AUDIO_SET_OCHANNELS

Set number of bits audio channels. As of the writing of this (2023-02-28) no drivers support this function.

Example for input driver:

```
if (ioctl(fp, IOCTL_AUDIO_SET_ICHANNELS, (char *)(6))) {
  printf("Couldn't set number of channels\n");
}
```

Example for output driver:

```
if (ioctl(fp, IOCTL_AUDIO_SET_OCHANNELS, (char *)(6))) {
  printf("Couldn't set number of channels\n");
}
```

11.4 Controlling Audio Buffers

11.4.1 IOCTL_AUDIO_GET_INPUT_BUFFER_FILL

Get input buffer fill state in 16-bit words.

- Only for drivers with input capability

Example:

```
iBufFill = ioctl(fp, IOCTL_AUDIO_GET_INPUT_BUFFER_FILL, NULL);
```

11.4.2 IOCTL_AUDIO_GET_INPUT_BUFFER_SIZE

Get input buffer size in 16-bit words.

- Only for drivers with input capability

Example:

```
iBufSize = ioctl(fp, IOCTL_AUDIO_GET_INPUT_BUFFER_SIZE, NULL);
```

11.4.3 IOCTL_AUDIO_SET_INPUT_BUFFER_SIZE

Set input buffer size in 16-bit words.

- Only for drivers with input capability

Example:

```
if (ioctl(fp, IOCTL_AUDIO_SET_INPUT_BUFFER_SIZE, (char *)(1024))) {
  printf("Couldn't set input buffer size\n");
}
```

11.4.4 IOCTL_AUDIO_GET_OUTPUT_BUFFER_FREE

Get how many 16-bit words there are free in the output buffer.

- Only for drivers with DSP output capability

Example:

```
iBufFill = ioctl(fp, IOCTL_AUDIO_GET_OUTPUT_BUFFER_FREE, NULL);
```

11.4.5 IOCTL_AUDIO_GET_OUTPUT_BUFFER_SIZE

Get output buffer size in 16-bit words.

- Only for drivers with DSP output capability

Example:

```
oBufSize = ioctl(fp, IOCTL_AUDIO_GET_OUTPUT_BUFFER_SIZE, NULL);
```

11.4.6 IOCTL_AUDIO_SET_OUTPUT_BUFFER_SIZE

Set output buffer size in 16-bit words.

- Only for drivers with DSP output capability

Example:

```
if (ioctl(fp, IOCTL_AUDIO_SET_OUTPUT_BUFFER_SIZE, (char *)(1024))) {
  printf("Couldn't set output buffer size\n");
}
```

11.5 Volume Control

11.5.1 IOCTL_AUDIO_GET_VOLUME

Get volume. Volume is a number between 0 - 511 where 256 is full-scale, and each successive number represents a volume gain step of -0.5 dB. See table below:

IOCTL_AUDIO_GET_VOLUME argument table

Argument	Amplification	Description
0	+128.0 dB	Insane amplification
1	+127.5 dB	Insane amplification minus 0.5 dB
...	...	...
255	+0.5 dB	Slightly louder than full-scale volume
256	0.0 dB	Full-scale volume
257	-0.5 dB	Almost full-scale volume
...	...	...
509	-126.0 dB	Very silent
510	-∞ dB	No sound, may not turn off driver
511	-∞ dB	No sound, may turn off driver

A driver may limit the range it actually accepts for its volume settings. E.g. the analog output driver AUODAC only supports the range between 256 (0.0 dB) and 511 (analog driver power-down). As another example, the S/PDIF driver supports the range between 208 (+24.0 dB) and 511 (silence). If a driver does not support the whole range, it will automatically limit itself so you can still call it with the extreme values.

511 is a special value that allows e.g. the audio driver to turn itself off (supported by e.g. AUODAC). Use with caution!

To get the maximum value the driver accepts as its setting, you may set the third parameter to IOCTL_PARAM_MAX. The get the minimum value use IOCTL_PARAM_MIN.

Example:

```
volume = ioctl(fp, IOCTL_AUDIO_GET_VOLUME, NULL);
maxVolArgVal = ioctl(fp, IOCTL_AUDIO_GET_VOLUME, IOCTL_ARGVAL_MAX);
minVolArgVal = ioctl(fp, IOCTL_AUDIO_GET_VOLUME, IOCTL_ARGVAL_MIN);
```

11.5.2 IOCTL_AUDIO_SET_VOLUME

Set volume. Scale for volume is the same as for IOCTL_AUDIO_GET_VOLUME (Chapter 11.5.1).

Example:

```
/* Set full scale volume */
if (ioctl(fp, IOCTL_AUDIO_SET_VOLUME, (char *)(256))) {
  printf("Couldn't set volume\n");
}
```

11.6 Miscellaneous Controls

11.6.1 IOCTL_AUDIO_GET_SAMPLE_COUNTER

Get sample counter. This value may be used to synchronize input and output (e.g. by the driver AUXSYNCS, Chapter 7.6.1).

Example:
```
s_int32 sampleCounter;
if (ioctl(fp, IOCTL_AUDIO_GET_SAMPLE_COUNTER, (char *)(&sampleCounter))) {
  printf("Couldn't get sample counter\n");
}
```

11.6.2 IOCTL_AUDIO_GET_OVERFLOWS

Get overflow sample counter for the input buffer.

If this number changes while an audio input program is running, this is an indication of a program performance or input/output buffer size issue.

If nobody cosumes samples from the input audio driver, this value increases at the rate of the sample counter that can be read with IOCTL_AUDIO_GET_SAMPLE_COUNTER.

- Only for drivers with input

Example:
```
s_int32 overFlow;
if (ioctl(fp, IOCTL_AUDIO_GET_OVERFLOWS, (char *)(&overFlow))) {
  printf("Couldn't get overflow counter\n");
}
```

11.6.3 IOCTL_AUDIO_GET_UNDERFLOWS

Get underflow sample counter for the output buffer.

If this number changes while an audio output program is running, this is an indication of a program performance or input/output buffer size issue.

If nobody produces samples for the output audio driver, this value increases at the rate of the sample counter that can be read with IOCTL_AUDIO_GET_SAMPLE_COUNTER.

- Only for drivers with output

Example:
```
s_int32 underFlow;
if (ioctl(fp, IOCTL_AUDIO_GET_UNDERFLOWS, (char *)(&underFlow))) {
  printf("Couldn't get underflow counter\n");
}
```

12 Controlling Audio from VSOS Shell with UiMessages

When using the VSOS Shell, some audio functions may be controlled even if running a VSOS program that doesn't take audio controls. If the TTY is not in RAW mode, the following escape sequences defined in <uimessages.h> may be sent to the shell.

12.1 Setting Volume anywhere from VSOS Shell

Note that <B> here means sending ASCII code 2, invoked in most terminal emulation programs by pushing Ctrl-B.

Volume up by 1/2 dB:
<B>111ms

Volume down by 1/2 dB:
<B>112ms

Set attenuation to -HH/2 dB, where HH is a hexadecimal number:
<B>206mHHs

Example:
To set volume to -20 dB, you need to send 40 = 0x28:
<B>206m28s

12.2 Sending Equalizer Controls from VSOS Shell

The filters are accessed with UiMessages that have the following format, where X is the filter number (0..f), and YY is the 16-bit signed value presented as an unsigned 16-bit hexadecimal number. <B>21XmYYs

Example:
Let's assume we have the following configuration lines in config.txt:

```
RUN SETEQU 1 3 100 0 0.7
RUN SETEQU 2 3 10000 0 0.7
```

Now, to set bass (filter channel 1) to +6 dB (6), send the following command:
<B>210m6s

To set treble (filter channel 2) to -12 dB (0xfff4), send the following command:
<B>211mfff4s

Up To 16 channels may be accesses with messages ranging from 0x210 to 0x21f.

13 Audio Decoders

Library audiodec automatically chooses between many Audio Decoders when presented an audio file. The libraries, their respective decode audio formats, and their clock rate requirements, are presented below.

NOTE: The clock speeds have been tested with a Class 4 SD card and “typical test files”. While VLSI believes the information to be accurate, clock rates should still be interpreted as estimates. The estimates are given for a system with a typical interrupt load and with a 8 KiW audio output buffer. An example `S:CONFIG.TXT` configuration file that sets the system up for best playback audio performance is provided in file config_audio_decoders.txt in the *VSOS Root and Libraries Source Code* package.

Audio Decoders		
LibName	**Format**	**Description**
decaac	AAC	AAC in ADTS and MP4 containers (.aac, .m4a, .mp4, .3gpp)
decac3	AC3	AC3 decoded into stereo
decaiff	AIFF	Apple uncompressed PCM format
decalac	ALAC	Apple lossless in MP4 (.mp4, .m4a) or CAFF (.caf) container
decape	APE	Monkey’s audio
decdsd	DSD	DSD bitstream files in .DSF and .DFF container, LSb first only
decflac	FLAC	Free Lossless Audio Codec
decmp3	MP3	MPEG audio layer 3
decvorb	Ogg Vorbis	Vorbis audio in Ogg container
decwav	RIFF WAV	Many RIFF WAV subformats
decwma	WMA	Windows Media Audio
mp4file	-	Determines if MP4 file contains ALAC or AAC

NOTE: For formats where there may be more than 2 audio channels, only files up to 2 audio channels are supported.

Library decaac subformats and clock requirements	
Clock	**Description**
30 MHz	AAC up to 48 kHz, 280 kbit/s

Library decaiff subformats and clock requirements	
Clock	**Description**
12 MHz	AIFF up to 96 kHz 16-bit
18 MHz	AIFF up to 96 kHz 24-bit
37 MHz	AIFF up to 192 kHz 24-bit
61 MHz	AIFF up to 352 kHz 24-bit

Library decac3 subformats and clock requirements	
Clock	**Description**
– MHz	5.1 channels at 48 kHz, – kbit/s

Library decalac subformats and clock requirements	
Clock	**Description**
37 MHz	Apple lossless up to 48 kHz 16-bit
74 MHz	Apple lossless up to 96 kHz 24-bit

Library decape subformats and clock requirements	
Clock	**Description**
61 MHz	Monkey's Audio up to 48 kHz 24-bit, profile Fast
67 MHz	Monkey's Audio up to 48 kHz 24-bit, profile Normal
79 MHz	Monkey's Audio up to 48 kHz 24-bit, profile High
N/A	Monkey's Audio, profiles Extra High and Insane

Library decdsd subformats and clock requirements	
Clock	**Description**
49 MHz	DSD64 (2.8 MHz, 1-bit)
86 MHz	DSD128 (5.6 MHz, 1-bit)
92 MHz	DSD256 (11.3 MHz, 1-bit)

Library decflac subformats and clock requirements	
Clock	**Description**
12 MHz	FLAC up to 16 kHz, 16-bit
18 MHz	FLAC up to 32 kHz, 16-bit
25 MHz	FLAC up to 48 kHz, 16-bit
37 MHz	FLAC up to 96 kHz, 16-bit
43 MHz	FLAC up to 96 kHz, 24-bit

Library decmp3 subformats and clock requirements	
Clock	**Description**
12 MHz	MP3 at 8 kHz, 8 kbit/s
31 MHz	MP3 up to 48 kHz, 320 kbit/s

Library decvorb subformats and clock requirements	
Clock	**Description**
12 MHz	Ogg Vorbis up to 16 kHz, 73 kbit/s
18 MHz	Ogg Vorbis up to 32 kHz, 151 kbit/s
37 MHz	Ogg Vorbis up to 48 kHz, 346 kbit/s
55 MHz	Ogg Vorbis up to 96 kHz, 362 kbit/s

Library decwav subformats and clock requirements	
Clock	**Description**
12 MHz	RIFF WAV up to 96 kHz 16-bit
18 MHz	RIFF WAV up to 96 kHz 24-bit
31 MHz	RIFF WAV up to 192 kHz 24-bit (e.g. DXD format)
55 MHz	RIFF WAV up to 352 kHz 24-bit (e.g. DXD format)
68 MHz	RIFF WAV up to 352 kHz 32-bit floating-point (e.g. DXD format)

Library decwma subformats and clock requirements	
Clock	**Description**
61 MHz	All WMA files in VLSI Solution's internal test suite

13.1 Decoder Loop Functionality

Some of the audio decoders include a chance to play a part of the audio file in a loop.

Depending on the decoder, there may or may not be support for the

The list of audio decoders that contains loop functionality, and the level of support, is provided in the following table:

Audio decoders with loop functionality				
LibName	**Set timing**[1]	**Sample accurate**[2]	**Smooth**[3]	**Comments**
decvorb	Yes	Yes	No	-
decwav	Yes	Yes[4]	No	-

[1] If this feature is not available, the decoder is only able to loop the complete audio file. To make sure user software is compatible with potential future versions of the driver which may start supporting the Set Timing feature, Loop structure should be set as follows:

```
loop->startSeconds = loop->endSeconds = loop->endSamples = 0;
loop->endSeconds = 0xFFFFFFFFU;
```

[2] If this feature is available, looping is sample-accurate. If not available, loop start and stop points may vary slightly.

[3] If this feature is available, loop supports the CFL_DECLICK flag which declicks the loop but is not sample accurate. If this feature is missing from the decoder, flag CFL_DECLICK is ignored.

[4] Exception: IMA ADPCM is not sample accurate.

An example of how to use the loop feature is provided in solution PlayFileLoop in the *VSOS Root and Libraries Source Code* package. Read the README.TXT file for details.

14 Audio Encoders

TBD.

14.1 ENCVORB.DR3 - Ogg Vorbis Encoder

TBD.

14.2 ENCMP3.DR3 - MP3 Encoder (VS1205 only)

TBD.

14.3 ENCOPUS.DR3 - Opus Raw Encoder

TBD.

14.4 ENCFLAC.DR3 - FLAC Encoder

TBD.

15 Latest Document Version Changes

This chapter describes the latest changes to this document.

Version 0.04, 2025-06-17

- First public prerelease.
- Added Chapter 7.5, *Ethernet Input Audio Driver*, including documentation for the AUIETH.DR3 driver.

Version 0.03, 2025-06-09

First internal prerelease.

16 Contact Information

VLSI Solution Oy
Entrance G, 2nd floor
Hermiankatu 8
FI-33720 Tampere
FINLAND

URL: http://www.vlsi.fi/
Phone: +358-50-462-3200
Commercial e-mail: sales@vlsi.fi

For technical support or suggestions regarding this document, please participate at
http://www.vsdsp-forum.com/
For confidential technical discussions, contact
support@vlsi.fi