UNIVERSITY OF APPLIED SCIENCES
KIEL

MASTER THESIS

Open-Source Audio Platform for Embedded Systems

Author:
Yasmeen SULTANA

Examiner:
Prof. Dr. Robert MANZKE

A thesis submitted for the degree of Master of Science

in Informaton Technology

May 1, 2018

Declaration of Authorship

I, Yasmeen SULTANA, declare that this thesis titled, "Open-Source Audio Platform for Embedded Systems" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

Date:

Abstract

The CTAG face 2 | 4 sound card is a hardware platform developed at Kiel University of Applied Sciences for all audio applications. This board is based on Analog devices AD1938 audio codec which contains 4 input ADC channels and 8 DAC output channels. The Linux based audio driver for this board has previously been developed on open source development platforms like the beagle bone and raspberry pi.

In this thesis, a further audio driver for the Teensyduino embedded platform has been developed using the open source Teensy audio library. At the time of this writing, the Teensy audio library offered two stereo I2S input / output channels, however, the sound card offers four stereo I2S audio outputs channels and two stereo inputs I2S channels. The AD1938 codec uses TDM encoding and decoding for transmitting and receiving multiple channels on a serial line. The development of the new audio driver includes the configuration of AD1938 codec via SPI driver and configuring the Teensy 3.6 micro controller registers to support I2S TDM streaming.

The Audio driver has been developed and tested to receive 8 channel input data and transmits 4 channel output data at 48kHz sample rate. By connecting two CTAG sound cards in daisy chain mode with a Teensy, the system can be expanded to 8 input channels and 16 output channels. The audio driver has also been extended to support these additional channels as well as supporting both master and slave mode.

The second part of the thesis deals with adding a reverberation effect for audio received from the audio codec. The Schroeder-Moorer based freeverb algorithm developed by Jezar is an often used reverb in open source platforms. The freeverb algorithm is suitable for the embedded system because of its small memory footprint as the decay time is mostly influenced by algorithmic parameters. A fixed-point implementation of the freeverb algorithm has been developed for the Teensy audio library for stereo channels and is tested in real-time.

Acknowledgements

I would like to express my special gratitude to my professor Dr. Robert Manzke, who motivated and introduced me to the embedded system world. Special thanks for his continuous advice, encouragement and technical support throughout the course of this thesis. Furthermore, technical equipment, such as oscilloscope, and facilities were provided in the University of Applied Sciences Kiel. Thank you very much. Also, I would like to thank Mr. Henrik Langer, co-student who was helpful with the information regarding daisy chain connections. I must thank my two sweet daughters, Nabila and Inaaya for their splendid support and for standing with me during my studies with patience and tolerance.

Contents

List of Figures

List of Tables

List of Abbreviations

PCM	**P**ulse **C**ode **M**odulation
I2S	**I**nter **IC** **S**ound
TDM	**T**ime **D**ivision **M**ultiplexing
SPI	**S**erial **P**eripheral **I**nterface
DMA	**D**irect **M**emory**A**ccess
ADC	**A**nalog to **D**igital **C**onverter
DAC	**D**igital to **A**nalog **C**onverter
TCD	**T**ransfer **C**ontrol **D**escriptor
SoC	**S**ystem-on-a-**C**hip
BSD	**B**erkeley **S**oftware **D**istribution
GPL	**G**eneral **P**ublic **L**icense
OSS	**O**pen **S**ource **S**oftware
CPU	**c**entral **P**rocessing **U**nit
IDE	**I**ntegrated **D**evelopment **E**nvironment
FIR	**F**inite **I**mpluse **R**esponse
IIR	**I**nfinite **I**mpluse **R**esponse
FPU	**F**loating **P**oint **U**nit
LPCF	**L**ow **P**ass **C**omb **F**ilter
RTT	**R**ound **T**rip **T**ime

Chapter 1

Introduction

1.1 Open source Eco-System

An Open source Ecosystem is a system that can be freely used, modified and shared by anyone. This system is developed, controlled, modified, reviewed, by general people and distributed under licenses that comply with open source definition[1].

Open source Initiative (OSI) published a document that describes the open source definition. The terms mentioned by OSI are [2] free redistribution, the author or license holder cannot collect royalty on the distribution of program, this program must make available to all the use , license should allow changes and derived works and must allow them to distribute under same terms as the original license, no person or group must be denied to access the code, License must not specific to a product, licensed software cannot restrict other software, license must be technology neutral and software must follow any non restricted licenses standards (like GNU General Public License (GPL), Apache Software License, Intel Open Source License, Berkeley Software Distribution (BSD) license etc).

The main purpose behind this approach is that a large group of people collaboratively without any financial gain wants to produce more useful and bug free product for everyone to use. Open source software (OSS) supporters argue that this kind of approach is faster and efficient and results are better quality. Now a days, even commercial software industries are also attracted to this approach. The Internet has brought students, hobbyist, developers, and users across the world together as a community to exchange ideas, discuss the problems and provide the solutions .

Due to a layered software architecture, drivers and middle wares libraries are hiding the hardware details and their configuration details; any user without hardware knowledge can develop an application using open source hardware platform and can contribute to open source community.

1.1.1 Open source operating system

The open source software(OSS) movement started by academicians like Donald Knuth, Richard Stallman against the commercialization of software has gained momentum in early 80's and led to the development of open source GNU operating system. With co-operation from developers, collaborating over the internet, Linus Torvalds as a student released the first version of the Linux kernel. The merger of GNU operating system with Linux kernel formed complete open source operating system. The University of California, Berkeley developed a UNIX derivative operating system and distributed as Berkeley Software Distribution(BSD).

The table contains few open source operating system and their licenses

Name	**Licenses**	**Operating family**
Linux	GPL/LGPL (GNU General Public License / GNU Lesser General Public License)	Unix-like
FreeBSD	BSD; GPL, LGPL software usually included	BSD Unix-like
OpenBSD	BSD (Berkeley Software Distribution)	Unix-like
FreeRTOS	modified GPL	real-time operating system kernel
FreeDOS	GPL	disk operating system (DOS)
GNU	GPL	Unix-like
Darwin, OpenDarwin, PureDarwin	APSL(Apple Public Source License)	BSD, Unix, Unix-like, OS X
Haiku	MIT (permissive free software license developed by the Massachusetts Institute of Technology (MIT))	BeOS(operating system developed by Be Inc.)

TABLE 1.1: Teensy 3.6 Open source Operating system and their licenses

Open source Linux based platform

The high-speed CPU processors and memory chips availability at a lower price has encouraged the companies to develop open source hardware platforms like Beagle bone, Raspberry pi for Linux operating system.The following are the mostly used open source hardware platforms by the student community.

1. **BeagleBoard**: Texas Instrument along with the electronics parts distributors like Digi-key and element14 has produced a low power open source single board computer. The primary goal of this educational board is promoting open source hardware in colleges and universities. This board has Texas Instrument's OMAP3530 system-on-a-chip(SoC) which, includes an ARM Cortex-A8 CPU(720 MHz), Imagination Technologies PowerVR SGX 2D/3D graphics processor and TMS320C64x+ core (520 MHz). This board can also run on different operating systems like Linux, Ubuntu, Android and Windows CE. The Beagle board has been updated with CPU, memory, and peripherals over the time. The Beagle board versions are BeagleBoard-xM, BeagleBone, BeagleBone Black, and BeagleBoard-X15 respectively.

2. **Raspberry Pi**: It is a single-board computer developed by the Raspberry Pi Foundation with the intent to promote the teaching of basic computer science. It comes in different models. All the models contain BCM2836 Broadcom SoC with an integrated ARM CPU and an on-chip graphical processing unit. Recently launched Raspberry pi 3 brings more powerful and ten time faster ARM Cortex-A53 processor compare to first generation of Raspberry pi. Additional features like WiFi and Blue-tooth makes it most compact and standalone computer [3]. With 1GB RAM, it can run bigger and powerful applications. It contains four USB and 40 GPIO pins, by using this one can connect many peripheral device.

   It runs on **Raspbian**, a Debian-based Linux operating system. Many other operating sytems like FreeBSD, Android, Windows 10 IoT Core can also run on Raspberry pi. Raspberry pi with low cost and high capabilities have become the choice of all levels of programmer.

3. **Arduino Yún**: The Arduino Yún is a microcontroller board based on the MIPS32 24K and ATmega32U4 CPU and the Atheros AR9331 processor with 400MHz speed and 64MB (AR9331) and 2.5KB (ATmega) memory. The Atheros processor supports a Linux distribution based on OpenWrt named Linino OS [4]. It contains features like Ethernet and WiFi support, USB Port, MicroSD port for external storage, 20 Input/Output pins. This is a type of Arduino board but with an ability to communicate with the Linux on board thus Arduni Yún offers powerful network mini computer.

4. **Inetl Galileo**: It is the first Arduino- boards based on Intel x86 architecture. This board combines the best of both Linux and an arduino emulator. Galileo is a microcontroller board based on the Intel® Quark SoC X1000 Application Processor, a 32-bit Intel Pentium-class system on a chip. [5]. Galileo is truely an open source, as schematic diagram and source code is available online and can be downloaded without any software license.

Open source microcontroller platforms without operating system

There are open source smaller microcontroller hardware platforms like Arduino based boards and Teensy for new generation sensors, wearable, low power applications which don't need higher memory and have fast booting requirement.

1. **Arduino**: Arduino is an open-source hardware and software platform company which produces micro controller boards, add-on boards(shield) and kits for a wide range of applications. There are Arduino boards for the Internet of thing, 3D printing, wearable, and educational. The Arduino Software (IDE) is a cross-platform software and it runs on Windows, Macintosh OSX, and Linux operating systems. There is a very big open source community developing and supporting various boards.

2. **Teensy**: A detailed information about teensy is discussed in section 1.2 (Teensy Eco-System).

3. **Adafruit Flora**: Adafruit Flora is a wearable electronic platform. It is thin, round, sewable based on Arduino microcontroller and commonly used in the wearable project. The latest version of the Flora features with micro-USB and Neopixel LEDs. It is easy to program and testing. Flora is based on Atmega 32u4 microcontroller, which powers Arduino Mega and Leonardo.[6] Since it is compatibility with Arduino, no much modification required and can use the same Arduino IDE.

4. **LightBlue Bean**: It is low energy Bluetooth Arduino-compatible microcontroller board. It also contains RGB LED, temperature sensor, and an accelerometer.It is programmed wirelessly and can be paired with Android or iOS devices. So it is well suitable for smartphones. It is powered by an ATmega328p microcontroller with 32KB Flash memory and 2KB SRAM and contains CR2032 coin cell battery [7].

1.2 Teensy Eco-System

Teensy is a complete USB-based microcontroller development system, in a very small footprint, capable of implementing many types of projects created by PJRC and designed by the co-owner, Paul Stoffregen. [8]. The Programming is done by means of the USB port.The teensy board is compatible with Arduino software & libraries. The open source teensy audio library is distributed under MIT-like licenses. It has few conditions like the copyright notice, development funding notice and the permission notice should be included in all files of the software.

Key features[9]

- USB can be any type of device
- Single pushbutton programming
- Easy to use Teensy Loader application
- Free software development tools
- Works with Mac OS X, Linux & Windows
- Tiny size, perfect for many projects

The table 1.2 list the packages [8] available in teensy.

Name	Description
framework-arduinoteensy	Arduino Wiring-based Framework
framework-mbed	mbed Framework
tool-teensy	Teensy Loader
toolchain-atmelavr	avr-gcc
toolchain-gccarmnoneeabi	gcc-arm-embedded

TABLE 1.2: Packages in Teensy

History of Teensy

Teensy has developed many powerful micro controller functions in the open source environment. The teensy's hardware and advance libraries of Arduino can be used to develop more powerful features. Similar to any other platform, teensy has also went through the iteration and now it is up to the 3rd generation of teensy board.

Teensy 1.0: It was introduced in late 2008 and was the first Arduino compatible board developed with USB communication feature. It offers 12 Mbps native USB.

Teensy 2.x: The Teensy 2.0 contains an AVR ATMEGA32U4 8bit microcontroller, 16 MHz clocks (16 MIPS), 25 Input/output lines and a USB client port. It added a new feature to support USB Keyboard, Mouse, and MIDI and hence became very popular for many enthusiast keyboard projects, either as a keyboard controller (eg the Phantom, tenkeyless keyboard). By using teensy 2.0, Paul created a code of stream USB package for LEDs and this was used to build great LED projects. The Teensy++ 2.0 with an AT90USB1286 chip that has 46 I/O lines.

Teensy 3.x: The Teensy 3.0, Teensy 3.1 and Teensy 3.2 have microcontrollers with ARM Cortex M-series CPUs (48Mhz, 72MHz, 72MHz clock respectively). Compare to other teensy's, this version of teensy has faster processor and memory increased to four times. Because of increased memory size, many audio applications are developed as an

audio library. The latest version is teensy 3.6 and details are explained in section 2.8.

Teensy LC: It has impressive capabilities and makes the project must simpler and it is ideal for 'Internet of Things' projects. It features an ARM Cortex-M0+ processor at 48 MHz, 62K Flash, 8K RAM, 12-bit analog input and output, hardware Serial, SPI & I2C, USB, and a total of 27 I/O pins.

1.2.1 Teensy Audio Library

The Teensy audio library consists of a set of objects that enable recording, processing, and playback of audio sampled at 44.1 kHz[10]. When the teensy is operating in I2S slave mode, the audio library can be configured to work at sampling frequencies like 48 KHz, 96 KHz by changing the define AUDIO SAMPLE RATE EXACT. The Objects instantiate particular audio functionalities, e.g., a waveform synthesizer, reverb, I2S input, I2S output and finite-impulse-response (FIR) filters etc. By creating a new object, a new functionality can be added. A cascade of objects forms a processing chain that performs a set of operations on inputs to produce the desired output[10]. For example, two or three I2S objects can be given to mixer, which produces the desired output (as shown in figure 1.1). The audio library defines each object in the chain to operate on 128 audio buffered samples.

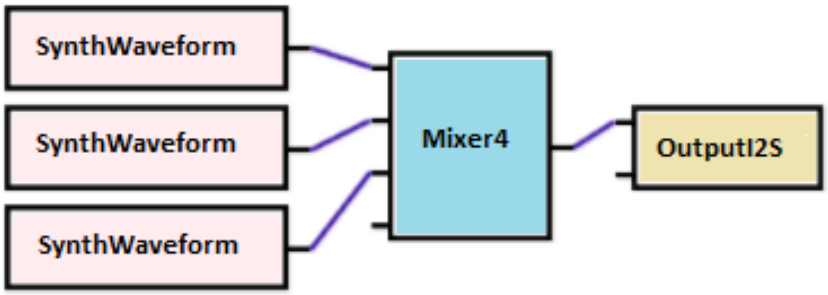

FIGURE 1.1: Cascading of objects

1.2.2 Audio Connections and memory

The important function of Teensy audio library related to audio stream connects are described in the table 1.3 [11]

Function	Description
AudioConnection myConnection(source, sourcePort, destination, destinationPort)	Route audio data from the source object's sourcePort, to the destination object's destinationPort.
AudioConnection myConnection(source, destination)	Route audio data from the source object's output 0, to the destination object's input 0.
AudioMemory(numberBlocks)	Allocate the memory for all audio connections. The numberBlocks input specifies how much memory to reserve for audio data. Each block holds 128 audio samples
AudioMemoryUsage()	Returns the number of blocks currently in use.
AudioMemoryUsageMax()	Return the maximum number of blocks that have ever been used. This is by far the most useful function for monitoring memory usage.
AudioMemoryUsageMaxReset()	Reset the maximum reported by AudioMemoryUsageMax. If you wish to discover the worst case usage over a period of time, this may be used at the beginning and then the maximum can be read.

TABLE 1.3: Audio Connections and Memory

1.3 Scope of Project

This thesis is divided into two parts. First part is to develop a C++ control class to interface CTAG face 2/4 sound card with Teensy 3.6 using the Teensy audio library. It also extends the existing I2S TDM class present in the teensy audio library to receive and transmit 16 channels in Teensy I2S slave mode.

The AD1938 codec based CTAG face 2/4 sound card can receive 4 input channels and transmit 8 output channels. The AD1938 codec registers can be configured using the SPI interface. The CTAG audio data is transmitted and received to and from the Teensy using I2S interface. The CTAG sound card has a crystal oscillator which is generating a clock at 24.576MHz. This generated clock is used to generate I2S clock signals when configured as I2S master. When two CTAG face 2/4 sound cards are connected in daisy chain mode the system can receive 8 input channels and transmit 16 output channels. For the test purpose the input signal from the PC is fed to any one of ADC input of CTAG sound card and output from any one of DAC's is connected to the speaker. The PCM pass-through program controls the routing of which ADC input shall go to which DAC output.

The second part of the thesis deals with implementation of a Freeverb algorithm and testing this reverb effect on real-time audio data that is received from the audio driver. Due to the memory limitation on the Teensy 3.6 board, we can test the reverb effect on only two channels.

The Freeverb build blocks like comb filters and all-pass filter are implemented in fixed-point arithmetic's to be compatiable with older teensy boards, reduce the CPU computational and increase the precision. The final mixing of the reverb output (wet) and direct(dry) signal is implemented using the existing mixer function present in the audio library. The reverberation effect is analysed by varying the feedback, damping factors of combo filter and gain of all pass filter.

The performance, latency of audio driver and the memory consumption and CPU performance of the freebverb is measured.

Chapter 2

Technical Background

2.1 Pulse-Code-Modulation (PCM)

Pulse coded modulation is the simplest form of pulse modulation. The PCM stream is a conversion of analog signal both in time and amplitude into a discrete both time and amplitude digital signal.The operations performed in the modulation of PCM are sampling, quantization, and encoding. The original signal can be reconstructed from the quantized samples by a demodulator. The accuracy with which the analog signal can be reproduced depends in part on the number of bits used to encode the original signal.

2.2 Inter-IC Sound (I^2S)

The Integrated Inter-IC Sound (I2S) serial bus interface standard was developed by Phillips Semiconductor especially for the transmission of digital audio between two IC's. It is a synchronous bus with three serial lines consisting of bit clock, word clock, and audio data.

As the transmitter and receiver have the same clock signal for data transmission, the transmitter as the master has to generate the bit clock, word-select signal, and data. In a complex system where there are more transmitters and receivers, it is difficult to define the master. So system master controls the audio data flow between the ICs. In such condition, the transmitter generates the data and act as a slave. Figure 2.1 illustrates some simple system configurations and the basic interface timing.

According to the I^2S specification [12], the bus consists of basic three lines: continuous serial clock(SCK), word select(WS), serial data(SD) (see Figure 2.1).

2.3 Time Division Multiplexed (TDM) Mode

Time Division Multiplexed method that allows multiple channels of data to be transmitted on a single data line. The equipment that combines and transmits the signals

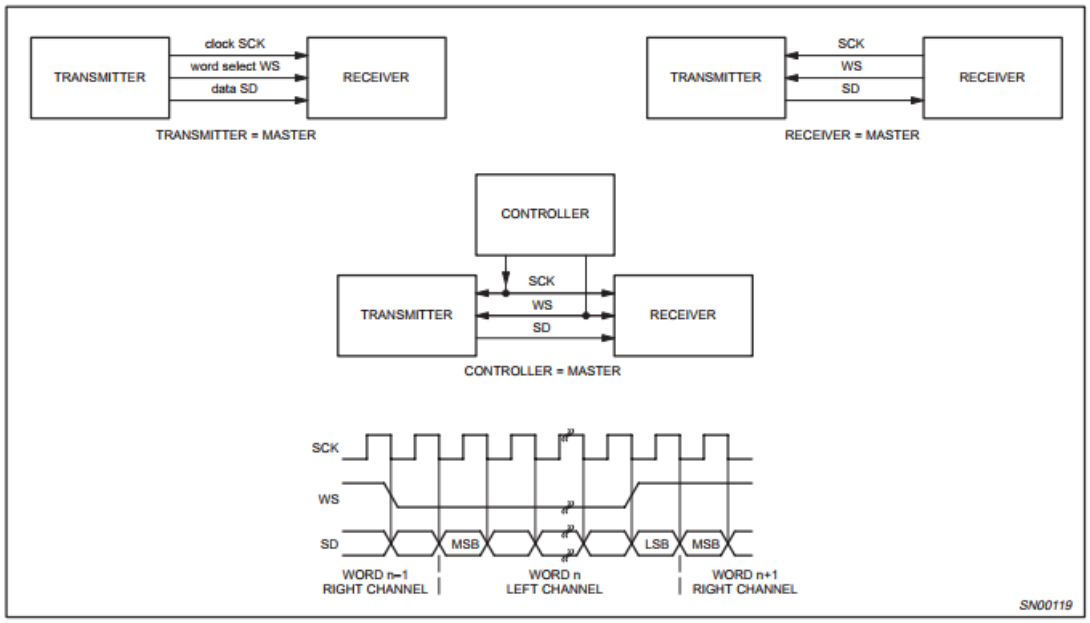

FIGURE 2.1: Simple System Configurations and Basic Interface Timing [12, S.2 Figure 1]

Name	**Function**
SCK	Its the output serial clock
WS	Defines the period of a sample, either left or right channel. This value can be set to: 16, 32 (default), 48, or 64. The word select line indicates the channel being transmitted: • WS = 0; channel 1 (left); • WS = 1; channel 2 (right). It acts like a frame sync in case of multi channel transmission.
SD	Serial data is transmitted in two's complement (Two time-multiplexed data channels)with the MSB first

TABLE 2.1: I^2S signal details

over a signal data line is known as Multiplexer. It accepts the input (data streams) from each source and divide each signal into units of the same size, and assign the units to the composite data in a rotating, repeated sequence. At the receiving end, this composite signal is separated by a equipment called De-multiplexer. The prerequisite for the successful TDM process is that the transmitter and the receiver have the same number of audio channels to be transmitted, their sampling depth or the TDMS slot width and the sampling rate.

The TDM interface is similar to the 2-Channel Serial Audio Interface, with the exception that more channels, typically 4, 6 or 8, are transmitted within a sample frame or sample period[13] as shown in Figure 2.2. As with the 2-Channel Serial Audio Interface, the TDM interface is consists of two control clocks, a frame synchronization pulse (FSYNC) and serial clock (SCLK), and the serial audio data line (SDATA).

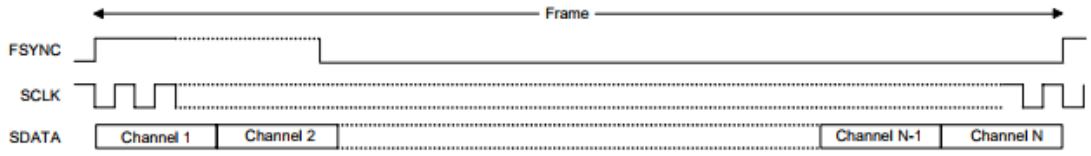

FIGURE 2.2: Generic TDM Interface [13, S.1, Figure 1]

Frame Synchronization Pulse - The FSYNC pulse is simply to identify the beginning of a TDM frame (at the rising edge or falling edge of the pulse). This frame rate is always at the audio sample rate, such as 44.1 kHz, 48 kHz, etc.

Channel Block - Each channel block consists of the audio data word succeeded by a sufficient number of zero data bits to complete the N-bit channel block. For example for a channel block of 32 bit with 24 bit audio data appended by 8 zero data bit.

Serial Clock - The main purpose of the serial clock is to shift the audio data into or out of the serial audio ports[13, pg.2]. For example, 8 TDM slots each with a width of 32 bits and a sampling rate of 48 kHz, a minimum clock frequency fCLK = $8 \cdot 32 \cdot 48$kHz = 12, 288MHz required.

2.4 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is an interface bus developed by Motorola to provide full-duplex synchronous serial communication between master and slave devices [14]. This Serial interface was developed to replace the parallel interface inside in any suitable system. Using this serial interface instead of parallel interfaces would inevitably reduce the volume of wires needed for the routing without reducing the data transfer speeds[15]. The synchronous data transfer is using only one clock provided by the SPI-master unit so the SPI-slave units can be designed without expensive precision oscillators.

A standard SPI bus is a form of four wired communication interface, these wire carry the information needed for the communication between SPI-master and SPI-slave. The signal wires include Master Out Slave In (MOSI), Master In Slave Out (MISO), the clock (SCK), and Slave Select (SS) (as shown in Figure 2.3). The SPI bus has one master and one or more slaves, master activates the slave via the chip select

emergency input (CS) to communicate with each other.

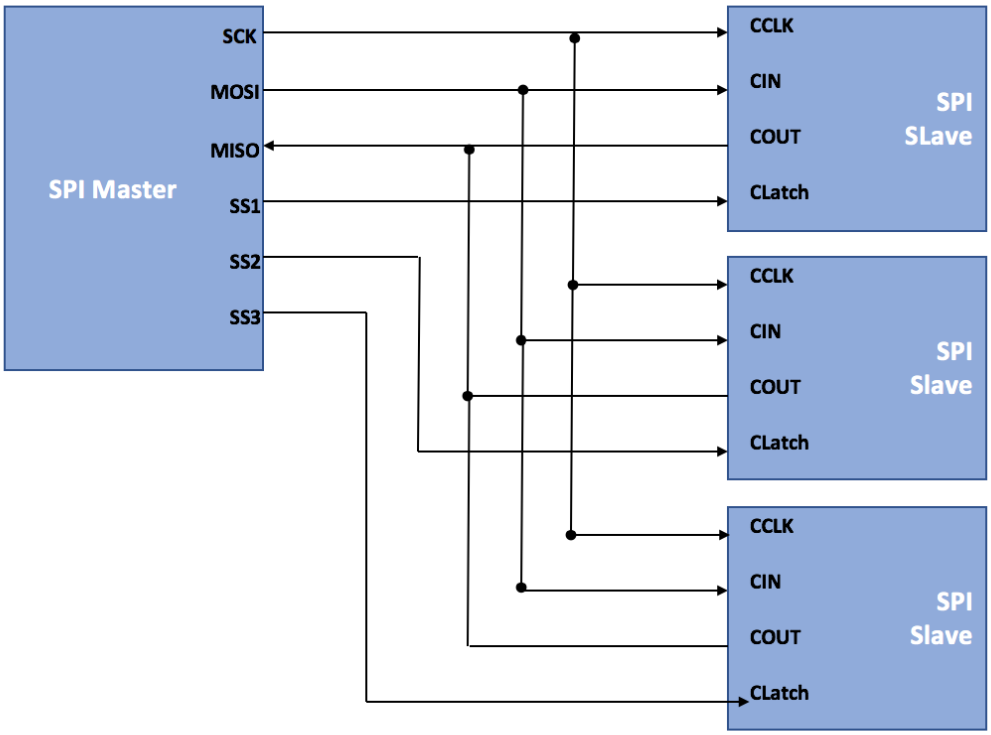

FIGURE 2.3: 4-wire SPI bus configuration with multiple slaves

2.4.1 Polarity and Clock Phase

The SPI bus operate in 4 different modes.These unique modes are defined by Clock polarity (CPOL) and clock phase (CPHA) (specified as "0", "1") as shown in figure 2.4. The detailed description of the available modes and data capture are summarized in the table 2.2.

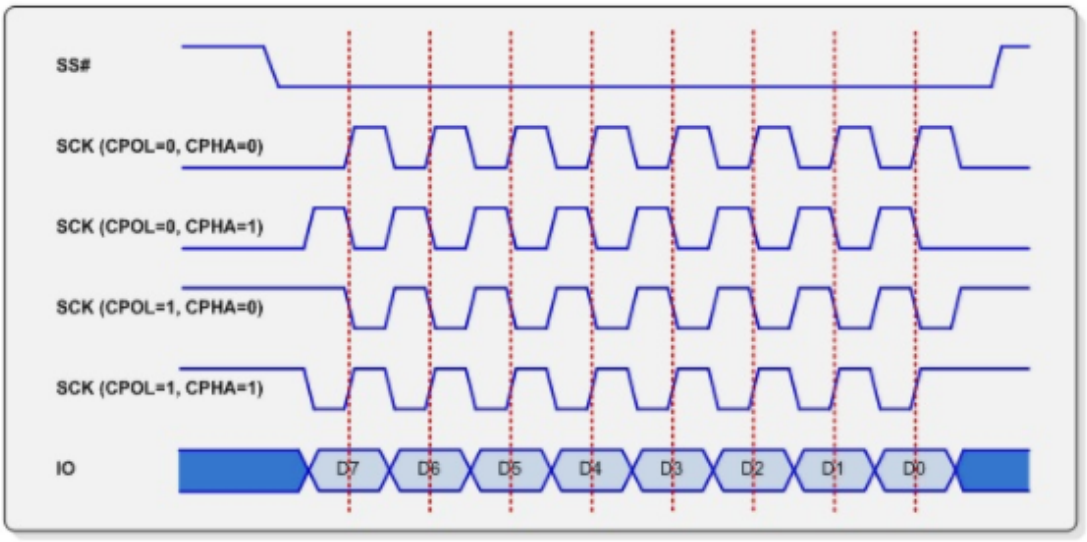

FIGURE 2.4: SPI bus timing [14, Figure 2]

MODE	CPOL	CPHA	Definition
0	0	0	data is sampled at the leading rising edge of the clock
1	0	1	data sampled at on the trailing falling edge
2	1	0	data is sampled at the leading falling edge of the clock
3	1	1	data sampled on the trailing rising edge

TABLE 2.2: SPI mode definitions

2.5 Direct Memory Access(DMA)

DMA is the generic term used to refer to a transfer protocol where a peripheral device transfers information directly to or from memory, without the system processor being required to perform the transaction [16]. DMA has several advantages and are listed below:

- DMA transfers the data with maximum speed thus is useful for high-speed data acquisition devices
- DMA also minimizes latency in the device as the dedicated hardware responds more quickly than interrupts, and also transfer time is short. Minimizing latency means, reducing the amount of temporary storage (memory) required on an I/O device[17].

- DMA also off-loads the processor, therefore, the processor is not used for handling the data transfer activity and is available for other processing activity.
- DMA also increases overall system utilization as data transfer actually occurs in parallel.

2.6 CTAG

CTAG face 2|4 is an I2S sound card based on AD1938 audio codec by Analog Devices Inc[18]. It is 4 layer PCB which also include ground and power panel. Soundcard contains 3 logical section, there are: 1. powder supply(digital 3.3V and 3.3v+5v analog), 2. Codec consists of reset circuitry and digital I/O and 3. Analog I/O.

2.7 AD1938 Codec

The AD1938 codec is a single chip high-performance audio codec. It contains (24 bit) four Analog-to-digital converts (ADCs) with inputs and eight Digital-to-Analog converters (DACs) arranged as single-ended output. The AD1938 is controlled through an SPI interface as shown in Functional block diagram figure 2.5. The onboard PLL is used to derive the master clock either from the LR clock or from an external crystal and thus the AD1938 eliminates the need for a separate high-frequency master clock [19].

The Following describes the summary of some characteristics and features from the AD1938 data-sheet [19]:

- Low EMI design
- operates from 3.3 V digital and analog supplies.
- 24bits Word Width
- Support 8khz to 192khz sampling rate
- SPI controllable for flexibility
- PLL generated or direct master clock
- Right-justified, left-justified, I2S-justified, and TDM modes
- 94 dB THD + N
- Log volume control with autoramp function

Functional Block Diagram (as shown in Figure 2.5) and Description of each block:

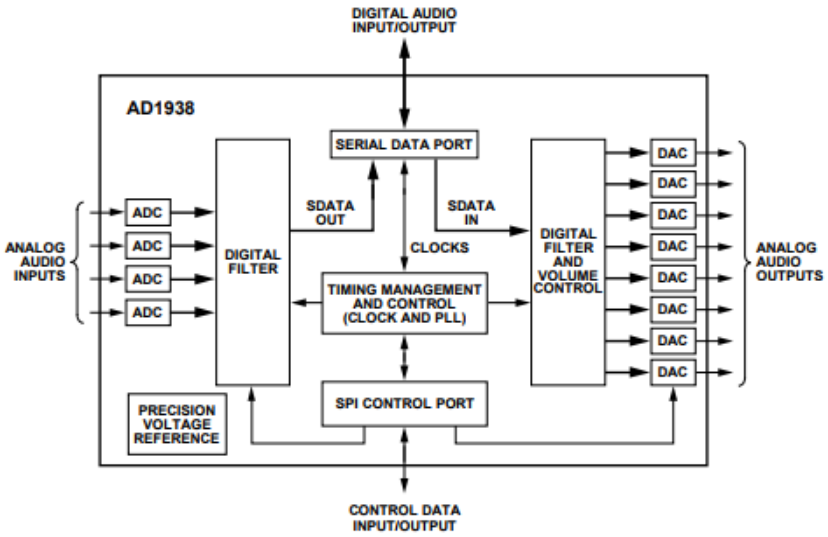

FIGURE 2.5: Functional Block Diagram [19, Figure 1]

2.7.1 Analog to digital converters(ADC)

In the AD1938 there are four ADC channels that are configured as two stereo pairs with differential input. The ADCs operates at a different sampling rate of 48kHz, 96kHz or 192kHz. It includes digital filters with 79db stopband attenuation. The Digital outputs are provided by the two serial data output pin, one for each stereo pair and a LR clock(ALRCLK) and bit clock (ABCLK).

ADC Control registers: There are three ADC control registers. ADC control register 0 configures bits related to the power-down mode, High-pass filter and output sample rate . ADC control register 1 defines ADC serial formats, word width, serial delay clock and bit clock (BCLK) active edge bits. similarly, programming ADC control register 2, configures LRCLK polarity and format, BCLK source, polarity and format and operating in either master or slave mode.

2.7.2 Digital to Analog Converter(DAC)

The AD1938 DAC channels configured as four stereo pairs(8 analog output). It includes digital filters with 70db stopband attenuation. Digital inputs are supplied through four serial data input pins (one for each stereo pair), a common frame clock (DLRCLK), and a bit clock (DBCLK)[19].

DAC Control registers: There are five DAC control registers. DAC control register 0 defines power-down mode, Sample rate, serial data delay, serial format. DAC control register 1 configures LRCLK Polarity , BCLK per frame, BCLK source, BCLK Polarity, LRCLK and BCLK master/slave mode bits. DAC control register 2 programs master mute, word width, output polarity. DAC Individual Channel Mutes register, as name

implies mutes individual DAC channels. DAC Volume Controls register control the output volume.

2.7.3 Clock signal

The onboard phase locked loop (PLL) can be selected to reference the input sample rate either from the LRCLK pins or 256, 384, 512, or 768 times the sample rate from the MCLKI/XI pin (48 kHz mode)[19]. In an AD193x family, if a device is programmed at 256xf_s, the frequency of master clock input is 256 x 48 KHz= 12.288 MHz. This frequency of the master clock should remain same even the device operates at 96 kHz or 192KHz (which become 128xf_s and 64xf_s respectively).

By default, onboard PLL generates the internal master clock from an external clock. This internal master clock for ADC is 256 x f_s for all clock modes but for DAC it varies according to the modes: 512xf_s(48KHz mode), 256xf_s(96KHz mode) and 128xf_s(192KHz mode). By setting the correct values in PLL and control register 1, a direct master clock 512 xf_s can be used either for ADC or DAC.

PLL and Clock control registers: There are two PLL and Clock control registers. PLL and Clock control register 0 defines PLL power-down mode, master clock rate setting, PLL input and internal master clock enable. PLL and clock control register 1 configures ADC and DAC clock source selection either from PLL or master clock.

2.8 Teensy 3.6

The Teensy 3.6 USB Development Board is an Arduino compatible USB-based microcontroller development system from PJRC [20]. All of the standard Arduino functions (digitalWrite, pinMode, analogRead, etc) works on Teensy. It has powerful MK66FX1M0VMD18[21], 32 bit 180 MHz ARM Cortex-M4 processor. This high performance ARM chip has faster clock speed, more memory, and flash. It also contains hardware peripherals such as SD card functionality, Ethernet, extended serial communication ports etc. The below figure 2.6 is the pin diagram of the Teensy 3.6. The table 2.3 summarizes the features specific to Teensy 3.6.

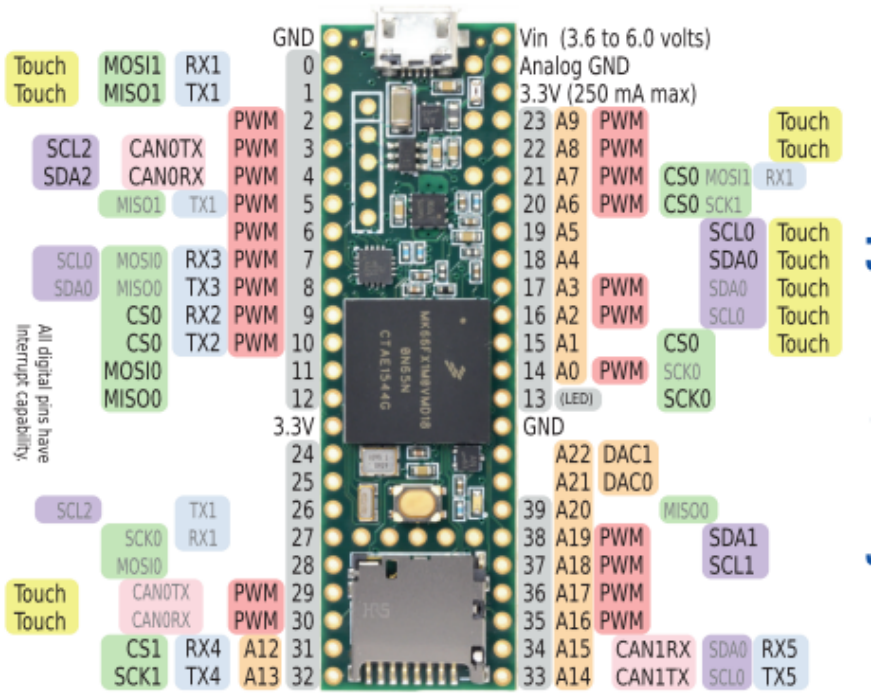

FIGURE 2.6: Teensy 3.6 pin configuration

Specification	**Teensy 3.6**
Flash Memory	1.25 MB
RAM Memory	256 KB
EEPROM	4KB(4096 (bytes)
I/O Pins	62 (42 breadboard friendly)
Tolerance On All Digital I/O Pins	5volts
PWM	22
peripherals	USB High Speed (480 Mbit/sec) Port 2 CAN Bus Ports 4 I2C Ports 11 Touch Sensing Inputs 3 SPI Ports 6 Serial Ports 14 Hardware Timers Native (4 bit SDIO) micro SD card port Hardware single precision floating point

TABLE 2.3: Teensy 3.6 hardware details

2.8.1 Teensy clock generation:

The Figure 2.7 describes the clock generation logic of the teensy's processor. The clock generation logic divides the clock sources for the various domains like system clock bus clock, flash memory and for peripheral module-specific clock.

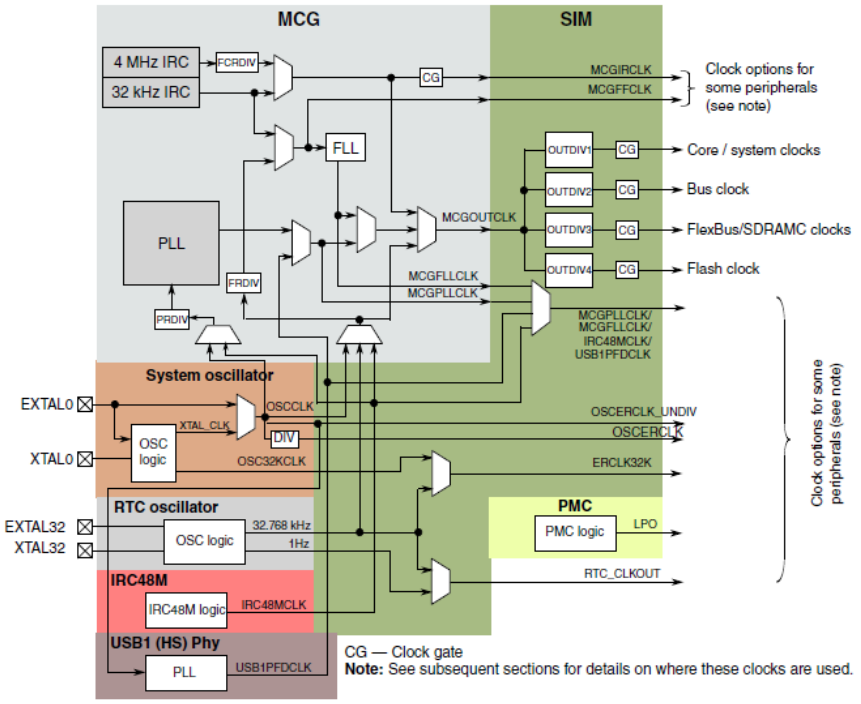

FIGURE 2.7: Clocking diagram [21, Figure 6-1]

The following table 2.4 describes the important clocks needed for the I2S clock generation from the previous block diagram

Clock name	Description
System clock	generated by MCGOUTCLK divided by OUTDIV1, clocks the bus masters directly.
MCGFLLCLK	MCG output derived from FLL.
MCGPLLCLK	MCG module output of the Phase lock loop(PLL). MCGFLLCLK and MCGPLLCLK may clock some modules.
USB1PFDCLK	USB Phy output of the PLL Fractional Divider.
IRC48MCLK	Internal 48 MHz oscillator that can be used as a reference to the MCG and also may clock some on-chip modules.
OSCERCLK	System oscillator output sourced from OSCCLKthat may clock some on-chip modules. Dividable by 1,2, and 4.

TABLE 2.4: Kinetic

The I2S audio master clock (MCLK) can be generated either from system clock (SYSCLK) or system oscillator clock (OSCERCLK) or peripheral clock(MCGPLLCLK, MCGFLLCLK, IRC48MCLK, USB1PFDCLK). MCLK is used to generate the bit clock and the frame sync of I2S/SAI. When using I2S as slave, the bit clock and frame sync are generated externally. Figure 2.8 illustrates I2S clock generation of the teensy3.6.

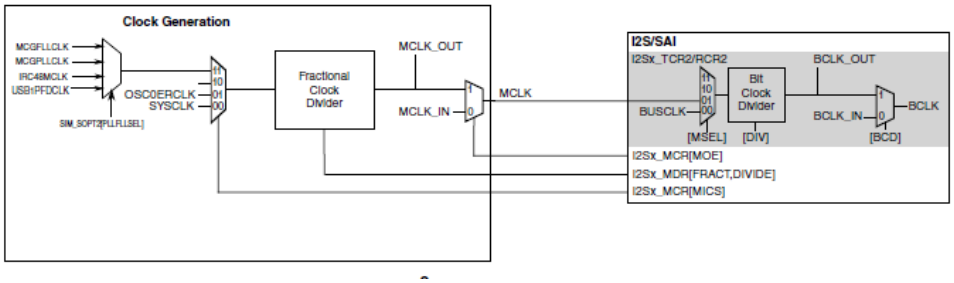

FIGURE 2.8: I^2S/SAI clock generation [21, Figure 6-12]

2.9 Audio Library Architecture

FIGURE 2.9: Teensy Software Architecture

The Teensy software can be visualized into four layers (as shown in Figure 2.9.

1. **Drivers**
   This layer contains the basic driver classes that are needed to communicate to external devices like SPI, USB, Serial Terminal, GPIO pin.

2. **Teensy Core Library**
   This layer contains implementation of Arduino driver base class and also defines the base classes for audio stream, usb, gpio and audio library

3. **Audio Library**
   This library contains the utilities for audio analysis like, fft, peek detection, filter, mixer and audio data capture, audio data playback from the SD card. There are utilities to generate various waveforms like sine, sweep and ramp signal.

4. **Audio Application**
   The audio application can be developed using the audio library and drivers.

2.9.1 Naming conventions of audio library

Teensy audio library development group has set of rules for creating new objects and their naming conventions. Audio objects should follow consistent naming conventions and adhere to common rules to make audio library usable by users of all skill levels. Every object in the audio library begins with "Audio" followed by category name, which is followed by a name that describes the function. Each name begins with capital letter.

The category names that are defined by teensy audio library are followed [22]

- **Input**: receives audio signals or real time data
- **Output**: transmits signal or real time data
- **Play**: Read data from SD card and play
- **Record**: Write data to SD card
- **Mixer**: Add multiple audio streams with individual gain
- **Analyze**: Analyze audio samples and provide information
- **Synth**: create audio
- **Filter**: Filter audio sample
- **Effect**: modifying audio, other than filtering
- **Control**: control external hardware which processes the audio.

For example from the object name"AudioInputI2S" one can deduce that it is a object which receives input data in I2S format.

2.9.2 Creating new objects in Teensy audio Library

By using Teensy audio library, creating the own audio processing objects very easy [23].

C++ Object definition All the new objects should be derived from the "AudioStream" base class. This base class provides all memory management (allocate, release), communication with other audio objects(receive, transmit, connect) and CPU/memory tracking functions. The basic C++ Template using AudioStream is shown below

```
#include "AudioStream.h"

class AudioEffectTemplate : public AudioStream
{
public:
        AudioEffectTemplate() : AudioStream(1, inputQueueArray);
        virtual void update(void);
private:
        audio_block_t *inputQueueArray[1];
};
```

In this thesis, AudioControlAD1938, AudioInputTDMSlave, AudioOutputTDMSlave and AudioEffectFreeverb classes are developed by following the rules mentioned above and further details about these classes can be found in Implementation chapter.

2.10 DMA on K66144M180FS

2.10.1 Enhanced direct memory access:

The enhanced direct memory access (eDMA) controller is a second-generation module capable of performing complex information exchanges with negligible intervention from a host processor. The main purpose of this module is to allow the CPU to perform other operations rather than data transfer, this will increase system performance. The eDMA module works in conjunction with the direct memory access multiplexer (DMAMUX), which routes DMA sources called slots to any of the DMA channels[21]. This device contains 32 DMA channel, which allows maximum 63 DMA request signals (59 peripheral slots and up to four always-on slots).The eDMA's programming model is separated into two regions(as shown in Figure 2.10:

- The eDMA engine (defines the calculation of source and destination address , and data movement operations)
- The transfer-control descriptor local memory (consists of transfer control descriptor (TCD) for each eDMA channel)

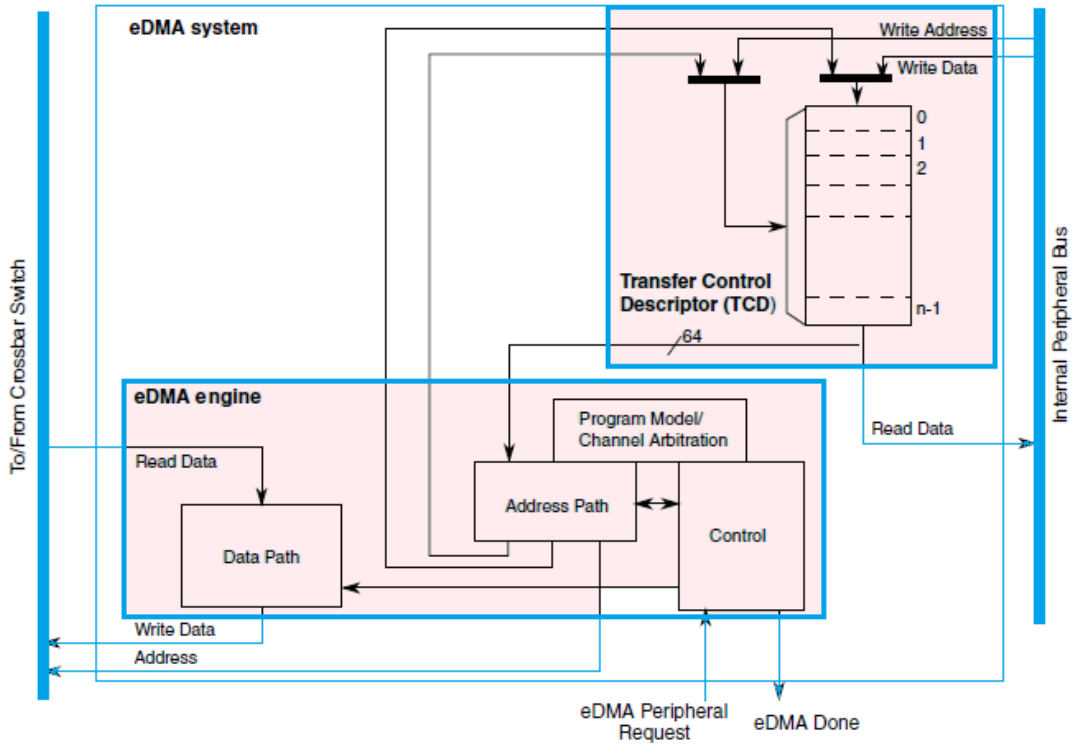

FIGURE 2.10: eDMA module

2.10.2 eDMA transfer control descriptor (TCD)

The TCD contains all the information about the data movement. This includes the source and destination addresses, the address increment after each transfer and the transfer size [24]. It operates in minor and major nested transfer loops

2.10.3 TCD memory:

Each channel requires a 32-byte transfer control descriptor for defining the desired data movement operation. The channel descriptors are stored in the local memory in sequential order: channel 0, channel 1, ... channel 31. Each TCDn definition is presented as 11 registers of 16 or 32 bits. Prior to activating a channel, you must initialize its TCD with the appropriate transfer profile. The table 2.5 summarizes the TCD Register description

eDMA	TCDn Register Name	Abbreviation	Width(bits)
0x00	Source Address	TCDn_SADDR	32
0x04	Transfer Attributes	TCDn_ATTR	16
0x06	Signed Sources Address Offset	TCDn_SOFF	16
0x08	Minor Byte Count	TCDn_NBYTES	32
0x0C	Last Source Address Adjustment	TCDn_SLAST	32
0x10	Destination Address	TCDn_DADDR	32
0x14	Current Minor Loop Link, Major Loop Count	TCDn_CITER	16
0x16	Signed Destination Address Offset	TCDn_DOFFR	16
0x18	Last Destination Address Adjustment/Scatter Gather Address	TCDn_DLAST_SGA	32
0x1C	Beginning Minor Loop Link, Major Loop Count	TCDn_BITER	16
0x1E	Control and Status	TCDn_CSR	10

TABLE 2.5: eDMA transfer control descriptor (TCD) register

2.10.4 Nested Loop Example

Each DMA request will move the number of bytes configured in the NBYTES (minor byte count) register. This corresponds to a minor loop [24]. When each minor loop is transferred CITER(major loop count) register is decremented. When major loop count reaches its half value DMA Interrupt is generated and data is copied. Similarly when major loop count reaches zero(CITER register is equal to zero), once again DMA interrupt is generated and data is transferred in to buffer.

Minor loop -NBYTES
Major loop- CITER minor loop
The Figure 2.11 shown the Nested Loop example.

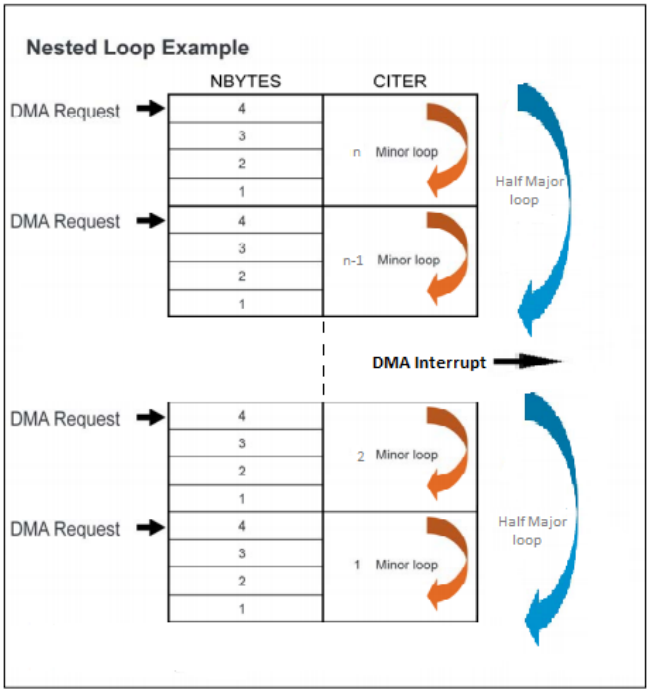

FIGURE 2.11: Nested Loop

2.11 Daisy chain

In daisy-chain mode, multiple devices are connected such that each device receives the previous devices' data and transfers it along with its own data[25]. Consider a circumstance where N devices are connected in the chain.The second device gets information from the first device and forwards this information along with its information to the third device. Similarly, the Nth device receives the collective information along with all the devices.

Finally, one receiver is connected to the Nth device, which collects all the data from the Nth device. Figure 2.12 shows how three (N = 3) devices work in a daisy chain. A daisy-chained network can take two basic forms: linear and ring[26]. An advantage of the ring topology over linear is that the number of transmitters and receivers can be cut in half since there is a loop connection back from the last device to the first. The main advantage of the daisy chain is its simplicity. Another advantage is scalability[27].

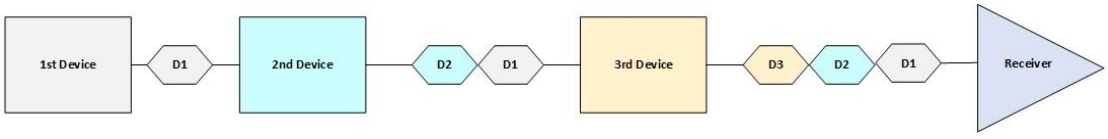

FIGURE 2.12: Three Devices in Daisy Chain

2.12 Digital signal processing

2.12.1 Digital Filters

The digital filter is a complicated device that performs mathematical operations on a sampled and discrete signal to enhance or reduce the necessary aspects of the signal. Digital filters are expensive when compare to analog filters and mostly used in signal processing. The behavior and response of the digital filter are very important. Digital filters have only handful of components that must be taken care and just by arranging these few component in a complex form a complicated filter can be made.

There are two types of filters in the digital realm: Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters.

Finite Impulse Response (FIR) Filter:- FIR Filter is a filter whose impulse response is of finite duration as it settles to zero at finite time [28]. In other terms: if the input signal has fallen to zero, the filter output will also fall to zero after a given number of sampling periods. The Figure 2.13 shows an FIR filter structure of order N + 1. For an FIR filter of order N, output is computed as a weighted sum all values of the filter input, i.e.

$$y(n) = b_0x(n) + b_1x(n-1) + ... + b_Nx(n-N) \tag{2.1}$$

$$y(n) = \sum_{i=0}^{N}(b(i) * x(n-i) \tag{2.2}$$

where:

- x(n) is the input signal
- y(n) is the output signal (given by linear combination of the last input samples x(n-i)
- b(i) is the coefficient of the filter(i.e. give the weight for the combination)

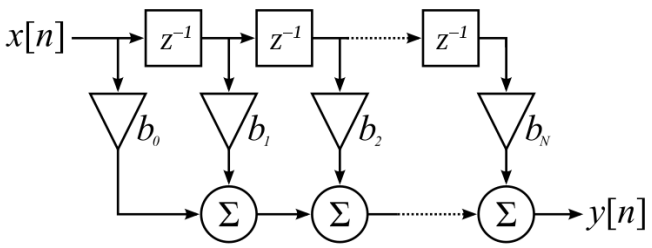

FIGURE 2.13: Discrete-time FIR filter of order N [28]

FIR Filters have few useful properties that make it sometimes important. The advantage of use FIR filter over IIR filter are:

- no feedback loop, no rounding error occurred by summed iteration. So implementation of such filter are simpler.
- very stable, the output is the summation of finite number of finite multiple of input.
- With symmetric coefficient FIR filter providers linear phase response, sometime this is very important in some applications like data communications, cross filtering etc
- Also used in designing all-pass filter to suppress the phase response of standard IIR filter.

The disadvantages of FIR filter are high CPU power consumption, low efficiency. By designing the specialized hardware for FIR filters, one can make it as efficient as IIR for many applications.

Infinite Impulse Response (IIR) Filter:- In DSP, IIR is one of the primary type of digital filter. As name implies the "impulse response" is infinite, because of the feedback in the filter. If the impulse signal given to filter, an infinite number of non zero values will be obtained, i.e it continues infinitely.

The figure 2.14 show the IIR filter example and the difference equation [29] which explains how the output signal is compared to input (equation 2.3)

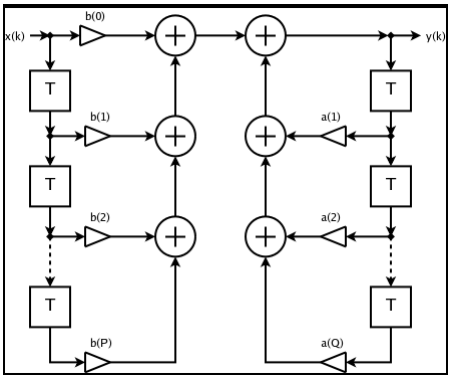

FIGURE 2.14: Sample IIR filter structure

$$y(n) = \frac{1}{a_0}(b_0x(n) + b_1x(n-1) + ... + b_px(n-D)$$

$$-a_1y(n-1) - a_2y(n-2) - .... - a_Qy(n-Q)) \tag{2.3}$$

Where

- P is the feedforward filter order
- b_i are the feedforward filter coefficients
- Q is the feedback filter order
- a_i are the feedback filter coefficients
- x(n) is the input signal
- y(n) is the output signal.

The advantage of IIR filter over FIR is, the given filter characteristic can be achieved with less memory and calculation. It also as many disadvantages like,

- If there is any error in output computation due to fixed point operations, this error is propagated back through the feedback and it continuously effect the output.
- less stable and nonlinear phase response

2.12.2 Filters response Type

There are many types of filters under this category like high pass filter, low pass, band pass, stop band, notch, comb filter and all-pass filter. For this thesis comb and all-pass filters are required, so description of these filters will be explained below.

Comb Filter:
A comb filter is generated by mixing the delayed version of the input signal to itself, causing constructive and destructive interface. A comb filter can be generated either from FIR or IIR filters. The frequency response of a comb filter appears as series of regularly spaced notches, giving the appearance of a comb. Thus the name comb is coined.

There are two different types of Comb filters depending on the direction in which delayed signal added to input like Feedforward and Feedback.

The advantages of the comb filters are:

- It has linearphase and guaranteed stability only for FIR form.
- It is used to replace expensive multipliers in the filter
- It reduces the artifacts such as dot crawl
- It can be used to model the effect of acoustic standing waves in a cylindrical cavity multiplierless filters

Feedforward form:- Feedforward comb filter is one of the simplest FIR filter [30] and the general structure is shown in the Figure 2.15. The frequency response of this filter is periodic, its drops periodically to minimum value and rises periodically to maximum value.

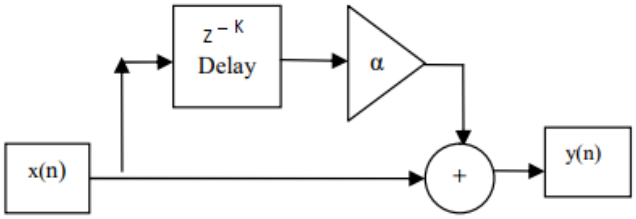

FIGURE 2.15: Feed Forward comb filter

It can be described by the difference equation, where K is the delay length, and α is a scaling factor applied to the delayed signal.

$$y(n) = x(n) + \alpha x(n - K) \tag{2.4}$$

If we take Z Transform on both sides, we define the transfer function as:

$$H(z) = \frac{Y(z)}{X(z)} \tag{2.5}$$

$$H(z) = 1 + \alpha z^{-K} \tag{2.6}$$

$$H(z) = \frac{z^K + \alpha}{z^K} \tag{2.7}$$

Feedback form:- The feedback comb filter is a simple type of infinite impulse response (IIR) filter [30] and the general structure is shown in the Figure 2.16

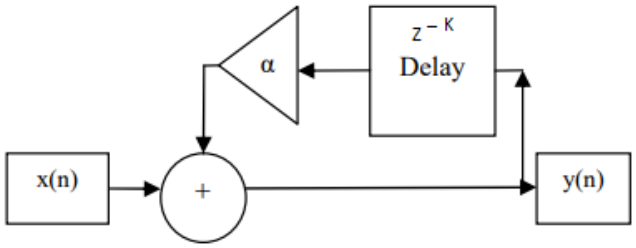

FIGURE 2.16: Feedback comb filter

General equation for feedback comb filter is

$$y(n) = x(n) + \alpha y(n - K) \tag{2.8}$$

The transfer function of feedback form is:

$$H(z) = \frac{Y(z)}{X(z)} \tag{2.9}$$

$$H(z) = \frac{z^K}{z^K - \alpha} \tag{2.10}$$

All-pass filter

The All-pass filter is one of the important blocks while building the digital audio signal processing system. It is a filter where the amplitude is unity for all frequency but introduces the phase shift [31]. It behaves as phase equalizer. The main purpose of using an all-pass filter is to add a phase shift to the response of the system. The all-pass filters are commonly used in an audio application like filter banks, speaker crossover, and reverberators. For artificial reverberators, the basic structure of the all-pass filter is shown in Figure 2.17. It is constructed by using delay with feedforward and feedback paths (This structure, known as the Schroeder all-pass comb filter).

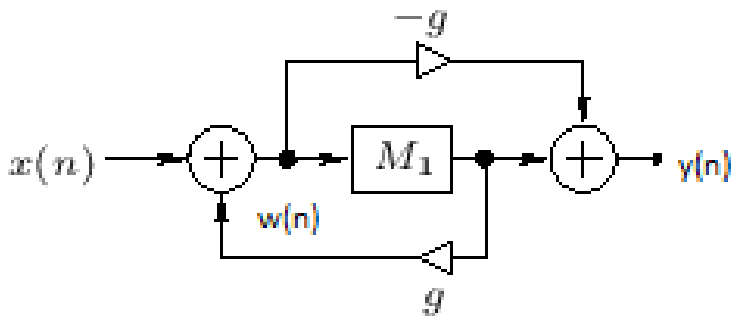

FIGURE 2.17: All-pass filter

The difference equation is :

$$w(n) = x(n) + g * w(n - M_1) \tag{2.11}$$

$$y(n) = -g * w(n) + w(n - M_1) \tag{2.12}$$

2.12.3 Numerical Precision

Floating-point representation

Floating point represent a number in a wide range from very small (2^{-17}) to very large (2^{70}). IEEE 754 defines the floating point numbers with three fields: **S** sine bit, **e** an exponent, **f** mantissa(fractional) [32].

IEEE 754 defines single and double precisions.

- single precision: it contains total 32bits, which includes one sign bit (positive or negative) , 8 bits exponent and 23 bits fractional.
- double precision: it contains total 64bits, which includes one sign bit (positive or negative) , 11 bits exponent and 52 bits fractional.

Advantages:

- It represent values between integers.
- because of the scaling factor, it can represent wide range of values.

Disadvantage:

- Processing time is more compare to fixed point number

Fixed point representation

Earlier low-cost microcontrollers and DSP processors don't have a Floating-Point Unit (FPU) to handle the floating-point operations. They have only Arithmetic Logic Unit (ALU) which can handle only integer arithmetics.

The floating-point numbers are multiplied by a scaling factor and make floating numbers as integers. The scaling factor depends upon how many fractional positions are absorbed in the integer. The scaling factor can be a power of 10(decimal) or power of 2 (binary). When the scaling factor is a power of 2 then it is called binary scaling. The number of bits of scaling factor determines the virtual binary point.

After the arithmetic operations, the integer is divided by the same scale factor (re-scaling) to remove the scaling. In binary scaling division can be achieved by right shifting the number.

Q format: The fixed points are represented by Q m.n format.

Where m is a number of bits for the signed integer part, n is a number of bits for the fractional part(scaling factor 2^n), m+n is the total word length, it can be 16 bits or 32 bits. For example, Q1.15 means 1 bit signed integer part and 15 bits for the fractional part.

Pros

- The fixed point has more precision than float point
- scaling and rescaling can be done using shift operations

Cons

- Due to the scaling, care must be taken to avoid the over flow.

2.13 Reverberation

Natural reverberation is the combined effect of multiple sound reflections within a room[33]. Once the source of the sound stops, the reverberation in a room causes the perceived sound to decay at a smooth and gradual rate. In the real world, reverberation characteristics are influenced by the dimensions of the actual room size, the construction of the room (wall materials), objects found in the room(i.e furniture) and diffusion.

Normally there is always a direct path from source to the listener but the sound waves travel by bouncing at the walls and surfaces in the room to the listener. These waves take longer time and path, with less energy. In the figure 2.18, as sound waves may be reflected several times before reaching the listener, the reflections attenuate over time [34].

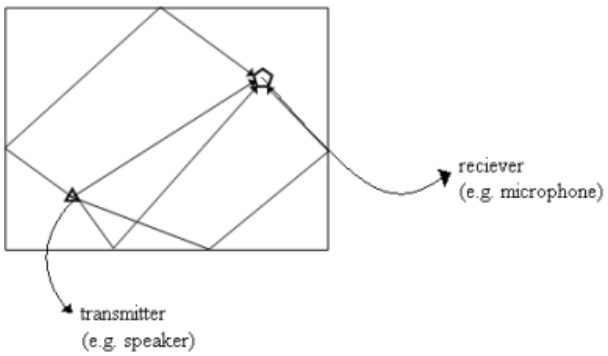

FIGURE 2.18: Sound wave traveling back and forth in a closed-space environment [34, Figure 1.2.1]

The figure 2.19 shows the impulse response plots for two very different spaces; a large concert hall and a cathedral [35]. As the sound waves travel, it reflects the nearby structures thus causing the first echoes. These initial echoes are called as Early Reflections. Since reflections are direct reflection it has high energy and difficult to implement but important for the simulation of reverb. The sound waves continue to expand and generate more reflections, with the reflected signals adding up with other reflected signals while decaying in energy. The resulting reverb "tail" is called late reverberation or subsequent reverberation [35]. The late reflections have a denser characteristic with lower energy and produce a sound with more persistence. The persistence is more because one can hear the small version of an original sound source for long after that source has fadeout. The time taken for sound energy to get below audible thresholds for human hearing is termed reverb or reverberation times [36].

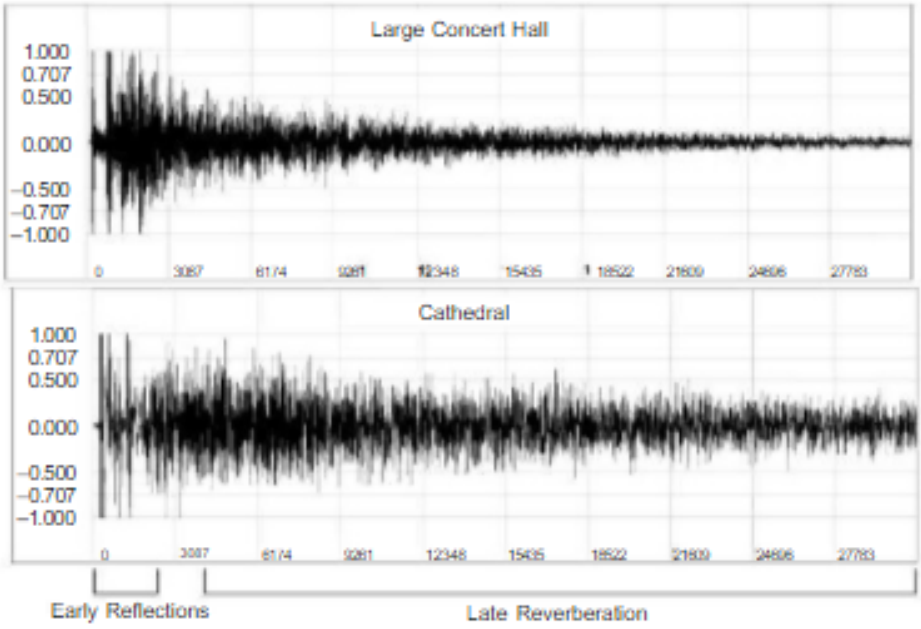

FIGURE 2.19: The impulse responses of a large hall and cathedral. [35, Figure 11.1]

2.13.1 The Reverb Time: RT_{60}

RT_{60} reverb time is one of the most well-known measurements for reverb. The amount of time required for reflections of a direct sound to decay 60 dB is reverb time or RT_{60}. Sabine worked on reverberation and established a relationship between the RT60 of a room, its volume, and its total absorption (in sabins) [37]. and derived the following formula in Equation 2.13 :

$$RT_{60} = 0.5\frac{V_R}{S_R A_{RAve}} \tag{2.13}$$

Where

$$V_R = \text{volume of room in } ft^3$$
$$S_R = \text{surface area of room in } ft^2$$
$$A_{RAve} = \text{average absorption coefficient}$$

2.13.2 Artificial Reverberation

In addition to natural reverb, one can also reproduce the reverberation effect using software (developed using audio cards, synthesizers, processors, and digital audio applications). In 1920, the need for the artificial reverberation first raised especially for the music studio and recordings [38].

Schroeder Reverberation

In 1960, Manfred Schroeder developed the first digital reverberation algorithm by using four parallel comb filter with two delay-based all-pass filters in series as shown in figure 2.20. In this structure, the comb filters produce the long echoes that occur between the walls in a hall or room. but these parallel comb filters do not build up the echoes as in the realistic situation. To increase the echo density the parallel comb filter output is fed to all-pass filters connected in series. These all-pass filter multiplies the echoes to generate the reverberation effect and avoid coloration as all-pass filters have the flat frequency response.

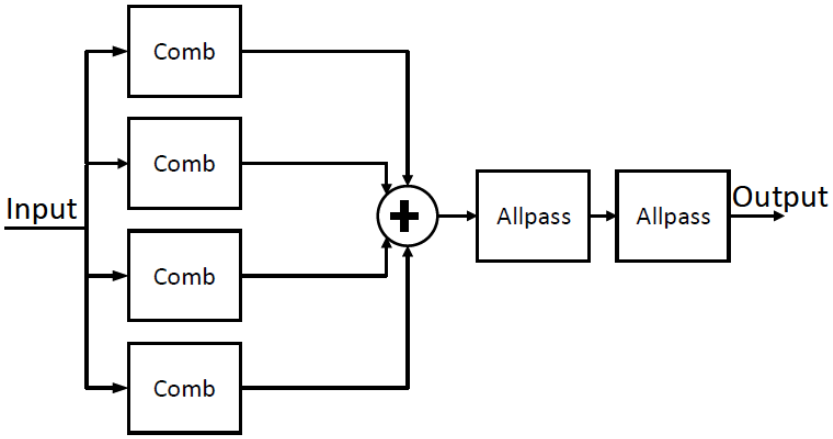

FIGURE 2.20: Schroeder's original reverb design

Schoerder suggested few properties for selecting comb filters and all-pass filter as follows: [35]

Comb filter properties:

- The ratio of smallest to largest delay should be 1:1.5
- choose the delay lengths that are typically mutually prime and spanning successive orders of magnitude, e.g., 1051, 337, 113 .
- Gain should be calculated according the below equation (2.14)

$$g = 10^{\frac{-3DT_s}{RT_{60}}} \tag{2.14}$$

where

T_s = sample period

D = the delay length
g = feedback gain

All-pass filter properties:

- Select the delay that are smaller than comb filters (1ms to 5ms)
- both the gain must be same (in between 0.5-0.707)

Moorer's Reverberation

In the real room, the high frequencies decay much faster than low frequency, this cannot be achieved using Schroeder reverberator. Other problems with Schroeder suggested structure is that the echo density doesn't adds-up sufficiently as required and for impulsive sounds, the response is fluttery and metallic. To overcome the above problems James Moorer suggested improvements (as shown in figure 2.21) to Schroeder reverberator structure by adding two additional comb filters with Low pass filter in the feedback path and only one all-pass on the output. These additional comb filters increase the echo density and with the addition of low pass filter generates the frequency dependent air absorption [39]. However, the Moorer's reverberator produces much better reverberation effect as compare to Schroeder but still sensitive to impulsive sounds.

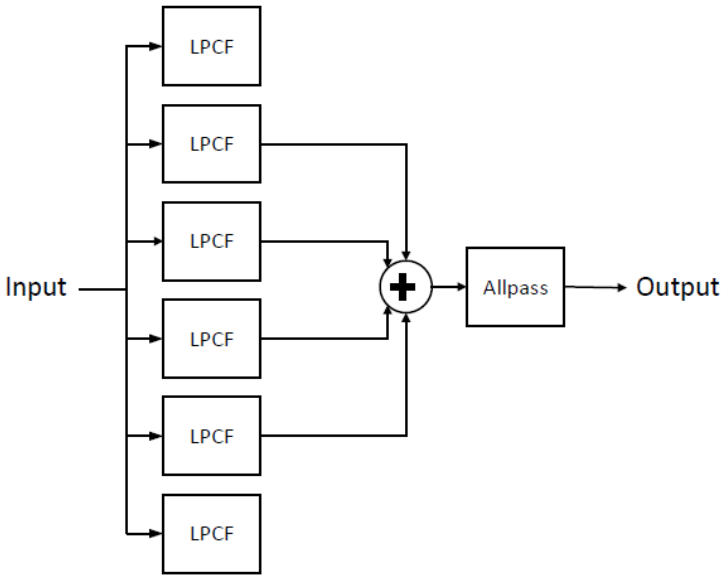

FIGURE 2.21: Moorer's reverb design

Low pass comb filter Reverberation

The figure 2.21 is the block diagram of low pass comb filter. By placing low pass filter in the feedback loop not only remove the high frequencies but also generate the short impulsive sounds by smearing the echoes [40].

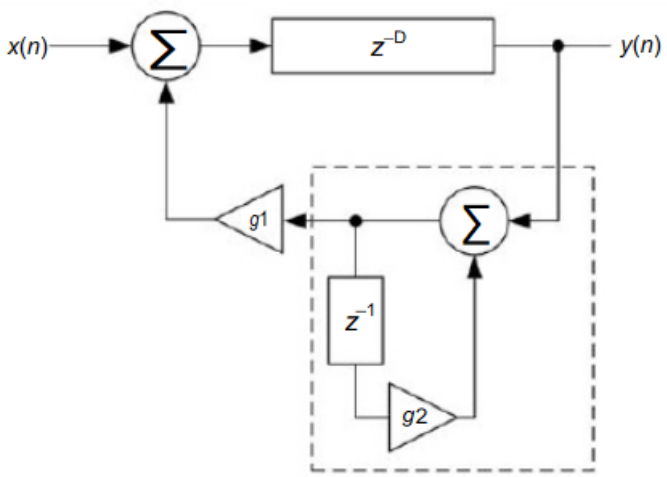

FIGURE 2.22: Low pass comb filter design

The difference equation of comb feed farword filter (equation 2.15) and Low pass filter(LPF) (equation 2.17) is as follows

$$H_C(z) = \frac{z^{-D}}{1 - g_1 z^{-D}} \tag{2.15}$$

$$H_{LP}(z) = \frac{1}{1 - g_2 z^{-1}} \tag{2.16}$$

According to th figure 2.22 the Lowpass comb filter equation can obtain by multiplying the LPF transfer equation to feedback g_1 Z^{-z}

$$H(z) = \frac{z^{-D}}{1 - H_{LP}(z) g_1 z^{-D}} \tag{2.17}$$

After solving the equation 2.18 the final output is

$$y(n) = x(n - D) - g_2 x(n - D - 1) + g_2 y(n - 1) + g_1 y(n - D) \tag{2.18}$$

2.13.3 Freeverb Algorithm

Freeverb is modifed implementation of Schroeder reverberation programmed by "Jezar at Dreampoint" and popularly used in free software world [41]. The basic model of the freeverb algorithm is shown in the figure 2.23. The two-channel input is combined to create the mono signal by multiplying with gain and is fed to left and right channel reverberator. The output from reverberators is given to a mixer which combines both direct signal and reverberator signals to produce the desired stereo output. More explanation on generation stereo output is explained in the coming sections.

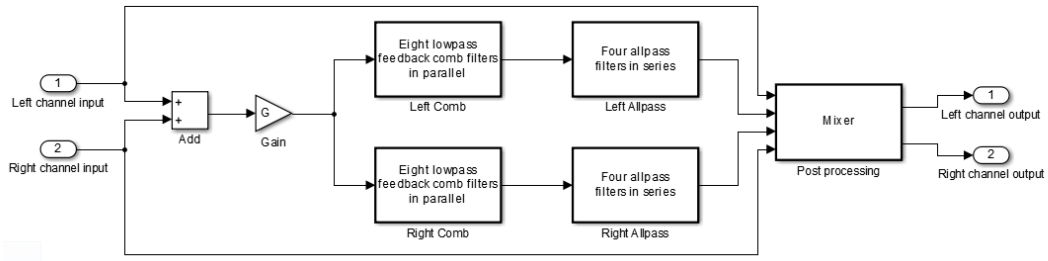

FIGURE 2.23: Freeverb algorithm Model [42]

The basic block diagram of each reverberator channel consists of 8 parallel Schroeder-Moorer lowpass comb-filters and four series Schroeder all-pass filter. The figure 2.24 shows the block diagram of the freeverb left audio channel with filter delay lengths. As per the suggestion by *Jezar*, the freeverb right stereo channel can be obtained by adding an integer value 23 to delay lines of all 12 filters. This integer is known as **stereospread** [41].

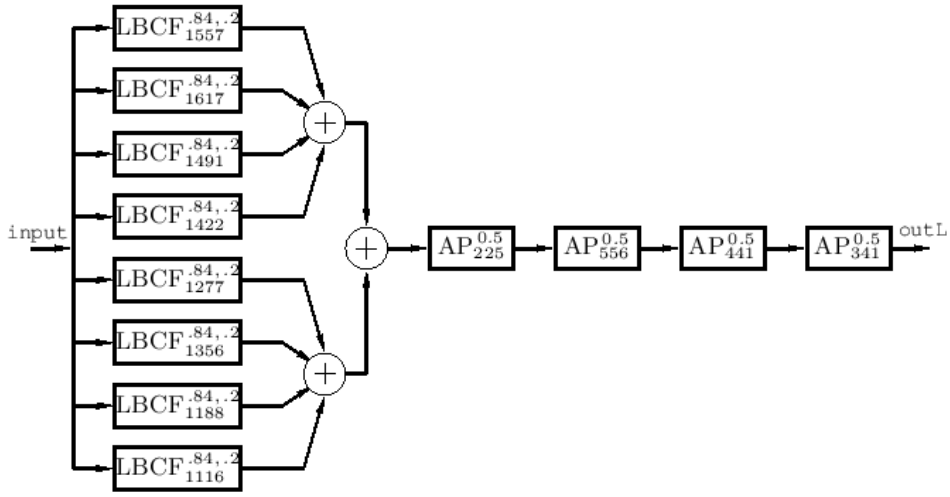

FIGURE 2.24: Freeverb algorithm for mono channel [41]

In the Mixer block the directed signals and the reflected signals are mixed to provide reverberation effect.

$$\begin{bmatrix} outputL \\ outputR \end{bmatrix} = dry \quad \begin{bmatrix} inputL \\ inputR \end{bmatrix} + \quad \begin{bmatrix} wet1 & wet2 \\ wet2 & wet1 \end{bmatrix} \quad \begin{bmatrix} outputL \\ outputR \end{bmatrix}$$

where

- *dry* = this parameter defines how much original signal should be mixed to output.
- *wet1 and wet2* = this parameter defines how much reverberation signal is mixed with output.

Lowpass-Feedback Comb Filter in freeverb

The lowpass comb filter in freeverb is similar to lowpass comb filter suggested by Moorer. But lowpass used is a unity gain one-pole as shown in figure 2.21. The lowpass filter gain was selected as g_1 = 1-d so that filter has unity gain at zero frequency and at high frequencies is a relatively small gain. The transfer function of the low pass filter is given below

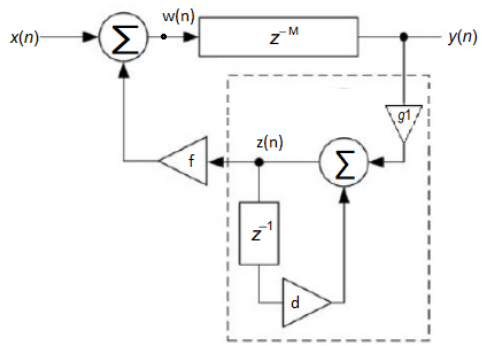

FIGURE 2.25: Lowpass comb filter in freeverb

$$H(z) = \frac{1-d}{1-dz^{-1}} \tag{2.19}$$

Therefore by substituting the lowpass filter equation 2.19 in the comb filter transfer function equation 2.15 , the approximate lowpass comb filter transfer function is then

$$LBCF_M^{f,d} = \frac{z^{-M}}{1 - f\frac{1-d}{1-dz^{-1}}z^{-M}} \tag{2.20}$$

The difference equation of the freeverb lowpass comb filter is

$$y(n) = w(n-M) \tag{2.21}$$
$$z(n) = y(n) * (1-d) + z(n-1) * d \tag{2.22}$$
$$w(n) = x(n) + z(n) * f \tag{2.23}$$

where

x(n) = input signal
y(n) = output signal
w(n-M) = delay line of comb filter
z_1 = delay line of lowpass filter
f = feedback gain (f < 1)
d = damping factor of lowpass (d < 0.5)

plugin parameter:

This section explains each user controllable effect parameters and their purposes [43]

Wet Mix:- Wet signals are the type of signals coming from reverberation and that undergo modification and process. Dry signals are the raw /direct signals coming from the source. The mixer controls the signal level by mixing the original dry signal with the altered wet signal. It determines how much stereo separation occur in the reverberation.

Roomsize (f):- In freeverb algorithm roomsize is related to the feedback gain of low pass comb filter. By increasing the roomsize (f increases) the reverberation time increases. For more stability the f must be less than one.

$$f = \text{roomsize} = \text{initialroom} * \text{scaleroom} + \text{offsetroom} \text{ [41]}$$
$$= 0.5*0.28 + 0.7 = 0.84$$

Damping (d):- It gives the decay factor of the reverberation. Higher the damping value, less reverb and decays fast. For stability the d must be less than one.

$$d = \text{damp} = \text{initialdamp} * \text{scaledamp} \text{ [41]}$$
$$= 0.5*0.4 = 0.2$$

2.14 Literature Review

The Teensy Audio library has support for the control of various codecs like Asahi KASEI AK4558, Cirrus Logic CS4272, Cirrus Logic CS42448, NXP SGTL5000 and Wolfson WS8731 etc. All the codec are controlled using Inter-Integrated Circuit (I2C) bus and codecs are configured as I2S slave and teensy as I2S master with maximum support of 8 channels in TDM mode.

This thesis developed the control class for the AD1938 codec in Teensy Audio library with SPI control. It also extended the TDM I2S input and output class for the both 8 and 16 channels and Teensy as I2S slave.

This thesis adds the reverb effect to the teensy audio library by using the freeverb algorithm, developed by Jezar. During the development process, many digital reverberation algorithms are studied. The first artificial reverberation was designed by Schroeder using four parallel comb filters and followed by two series all-pass filters [44]. The Schroeder reverberator does not have sufficient echo density and had very poor response (Metallic) for the impulsive sounds [39].

Moorer improved the Schroeder algorithm by adding more comb filters to increase the echo density [40].A low pass filter is added to the feedback loop of the comb filters to simulate the frequency dependent air absorption [39]. But still has poor response to impulsive sounds.

For exponential buildup of the echoes, Gardner suggested using nested all-pass filters, where an all-pass filter is embedded to the delay line in another all-pass filter. The echoes generated by inner all-pass filter are recirculated by the outer feedback path.As a result, the echo density increases with the time [45] and reduced the metallic sound of the all-pass filter. The nested all-pass filter design is more complex to implement.

convolution reverb convolves the input signal with pre-recorded or approximated or simulated room impulse response(RIR) [46]. The convolution operation can be realized by passing the input signal through the FIR filter whose coefficient are the room impulse response. This reverb has the best sound but unfortunately more complex operations and computationally expensive. There is no chance to change the reverb parameter.

There much more reverb algorithm, this thesis implements the freeverb algorithm. The freeverb algorithm is based on Schroeder and Moorer reverberator, which is programmed in C++ by Jezar at Dreampoint is widely used in the open source community. The teensy 3.6 has only 256KB RAM, a Freeverb algorithm for two-channel audio stream can be implemented within this available memory.

Chapter 3

Implementation and Hardware Setup

This chapter explains about hardware and software tool setup and details of implementation of individual components.

3.1 Hardware Setup

3.1.1 Installing the teensyduino application

The Arduino IDE doesn't come with built-in support for Teensy devices, so Paul Stoffregen (the genius behind Teensy) created a simple application called Teensyduino which allows you to program your Teensy directly from the Arduino IDE, as well as adjust the clock speed, and USB device functions (serial, HID, MIDI etc. . .)[47]. Below are few steps that have to follow for the installation of teensyduino on windows PC.

1. Install Arduino from https://www.arduino.cc/en/Main/Software The current release build is Arduino-1.8.5
2. Install Teensyduino https://www.pjrc.com/teensy/td_140/TeensyduinoInstall.exe
3. Select the Teensyduino installation directory as Arduino directory (C:/Program Files (x86)/Arduino)
4. Follow the installation step 3 given in https://www.pjrc.com/teensy/td_download.html The Teensy audio library is installed in the "C:/Program Files (x86)/Arduino/hardware/avr/libraries/Audio" folder. Now open the arduino

Once the installation is finished, restart the application. Now navigate to Tools>Board: the list of Teensy Boards are displayed and select the teensy 3.6. Set Clock speed to 196MHz Optimize speed and ensure USB device type selected as 'serial'.

3.1.2 Setup

The two CTAG board connected to the teensy 3.6 forming daisy chain mode. The Figure 3.1 explains the hardware setup and how the connections are made physically. The

table 3.1 and 3.2 explains the Teensy 3.6 connects to the pins on the Master AD1938 sound card and Slave AD1938 sound card respectively.

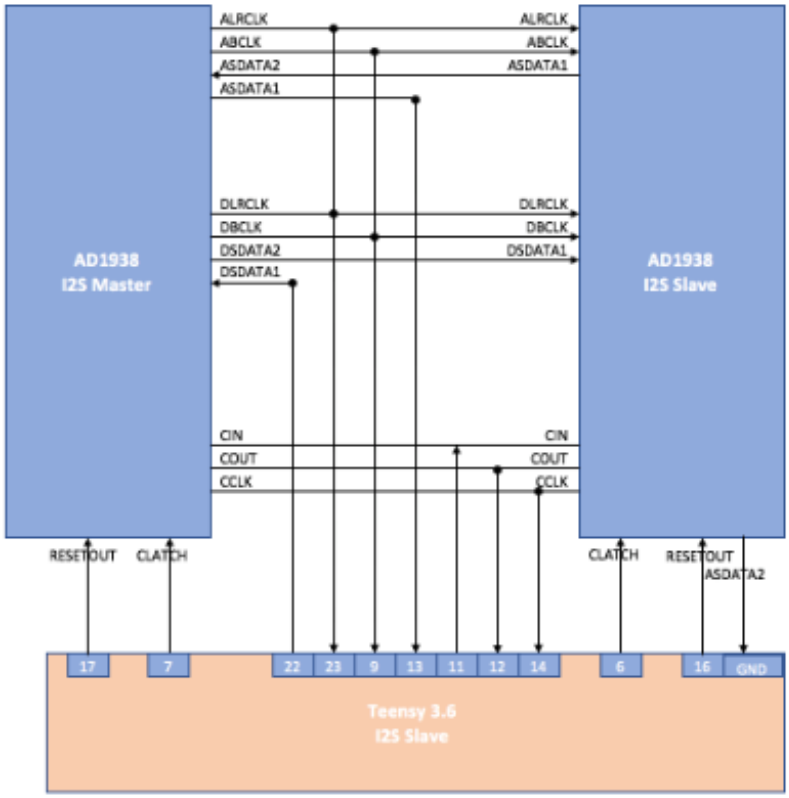

FIGURE 3.1: Hardware Block Diagram

Master AD1938 AudioCard Pin Name	**Teensy 3.6**
I2S_MCLK	Not Used
I2S_DAC_LRCLK	24
I2S_DAC_BCLK	9
I2S_ADC_LRCLK	24
I2S_ADC_BCLK	9
I2S_DAC_DATA1	22
I2S_ADC_DATA1	13
SPI_CLATCH	7
SPI_CCLK	14
SPI_COUT	12
SPI_CIN	11
RESETOUT	17

Slave AD1938 AudioCard Pin Name	**Teensy 3.6**
I2S_MCLK	Not Used
I2S_DAC_LRCLK	24
I2S_DAC_BCLK	9
I2S_ADC_LRCLK	24
I2S_ADC_BCLK	9
I2S_DAC_DATA1	-
I2S_ADC_DATA1	-
SPI_CLATCH	6
SPI_CCLK	14
SPI_COUT	12
SPI_CIN	11
RESETOUT	16
I2S_ADC_DATA2	GND

TABLE 3.1: Pin connections of Master and Slave AD1938 AudioCard and Teensy 3.6

Master AD1938 Audio-Card Pin Name	**Slave AD1938 Audio-Card Pin Name**
I2S_DAC_DATA2	I2S_DAC_DATA1
I2S_ADC_DATA2	I2S_ADC_DATA1

TABLE 3.2: Daisy chain Pin connections between Master and Slave AD1938 codec

3.1.3 Data Sheets

Teensy 3.6

Detailed hardware information is available in the K66P144M180SF5RMV2 Datasheet[21].

AD1938

The summary of characteristic and features of Codec AD1938 is available in the Datasheet[19]

3.2 Daisy chain implementation

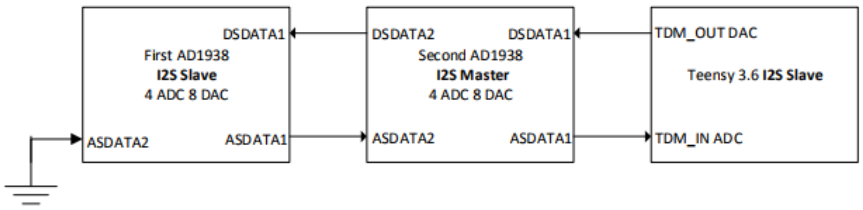

FIGURE 3.2: Daisy Chain

AD1938 codec support the daisy chain mode configuration for both ADC and DAC data transfer.

Two AD1938 are connected in daisy chain mode to expand the system to 8 ADCs and 16 DACs. The second AD1938 in the above diagram which acts as I2S master provides the bit clock(BLCK) and frame sync (LRCLK)for both Teensy and first AD1938 codec. The second AD9138 codec registers are configured to use its ADC clock as an I2S master for the DAC unit (as shown in figure 3.2).

3.2.1 ADC data flow

The first AD1938 codec ADC output data (4 ADC) is pushed into the second AD1938 codec. The second AD1938 combines this received ADC data with its own ADC data (4 ADC) and outputs combined ADC data (8 ADC) to Teensy TDM_IN (explained in Figure 3.3).

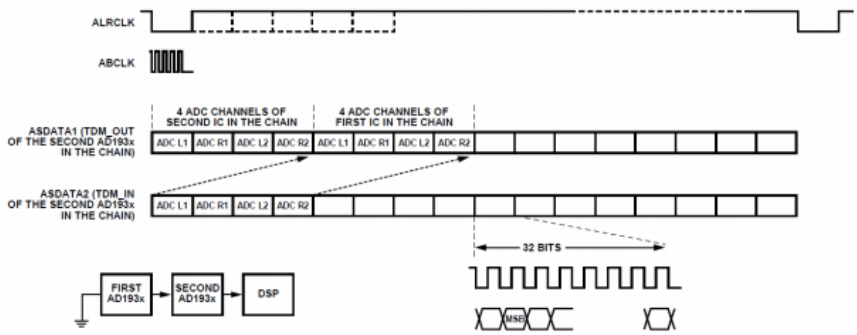

FIGURE 3.3: ADC TDM Daisy-Chain Mode (512 fS BCLK, Two-AD193x Daisy Chain) [19, Figure 22]

3.2.2 DAC data flow

When Teensy sends 16 channels through serial port (TDM out) to second AD1938, the second AD1938 separates the first AD1938 DAC data and its own DAC data. The first AD1938 data is pushed through DSDATA2 to the first AD1938 (see Figure 3.4).

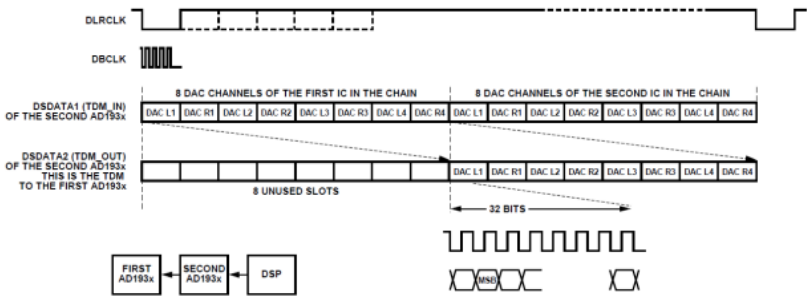

FIGURE 3.4: Single-Line DAC TDM Daisy-Chain Mode [19, Figure 18]

3.3 Audio channels and Sample rate support

The following table list the Audio channel and sample rate available for various ADC daisy chain configurations.

Sampling rate	Audio Channel
48KHz	8 channels at 256 fs single line TDM
48KHz	16 channels at 512 fs single line TDM(The last 8 channels are empty)

TABLE 3.3: ADC daisy chain configuration

These following are various DAC daisy chain configurations which are possible.

Sampling rate	Audio Channel
48KHz	16 channels at 512 fs single line TDM
96KHz	16 channels at 256fs dual line TDM
192KHz	16 channels at 256fs dual line TDM

TABLE 3.4: DAC daisy chain configuration

3.4 Audio Control Class for AD1938 codec

The Teensy Audio library contains the controls class for various codecs like Asahi KASEI AK4558, Cirrus Logic CS4272, Cirrus Logic CS42448, NXP SGTL5000 and Wolfson WS8731. All the codec are controlled using the I2C bus. There is no support for the AD1938 codec in Teensy Audio library.

The AD1938 audio control class is derived from the base class of "AudioControl". This base AudioControl class contains functions for all the codecs and some functions may not be needed for AD193x. In AD1938 there is no gain control for ADC and no support input source selection, for this reason, inputLevel() and inputSelect() function not needed. The following section gives the details how to implement AD1938 control class.

The Audio control class can be divided in two parts

1. SPI port control
2. AD1938 codec control

3.4.1 SPI port control

The AD1938 has an SPI control port that allows writing and reading internal control registers of ADCs, DACs, and clock system. The high-level SPI port write and SPI port read function for AD1938 are developed using the existing SPI library and wire library functions.

1. Initialize SPI pins and reset ad1938

Configure the Teensy pins, clatch and reset out pins as output. To reset the Ad1938 by pulling the reset out pin to low and hold for some time and then pull it back to high using the pinMode function of wiring library.

```
pinMode(ad1938_clatch, OUTPUT);
pinMode(ad1938_reset, OUTPUT);

digitalWrite(ad1938_reset, LOW);
delay(200);
digitalWrite(ad1938_reset, HIGH);
delay(400); //wait for 300ms to load the code
```

After the reset, most of the registers in the Ad1938 codec will initialize with default values. The initialization will run and PLL acquires the lock state.

2. Configuration Teensy SPI pins

The teensy SPI library uses some pins as default SPI pins(10,11,12,13) as shown in the pin diagram figure 2.6. If the default pins are used for another purpose, one can reconfigure alternate pins as SPI pins using the SPI library functions. The function begin() will initialize the Teensy SPI port registers with configured clock, MISO (master in slave out), MOSI (master out slave in) pin information.

```
/*SPI clock pin set*/
SPI.setMOSI(cin);
SPI.setMISO(cout);
SPI.setSCK(cclk);/*SPI clock alternate pin*/
SPI.begin();
```

3. Configuration Teensy SPI clock and mode

The SPI clock frequency, transmission byte order, and the SPI clock mode are configured using the function SPISetting from the library. .

SPI Settings(AD1938_SPI_CLK_FREQ, MSBFIRST, SPI_MODE3);

The AD1938 can operate up to maximum 10 MHz SPI clock frequency. In this project, 1 MHz clock is used. SPI library has two additional bus protection functions to avoid corrupting the ongoing SPI operations.

```
/*disables the SPI interrupts and apply the settings*/
SPI.beginTransaction(SPISettings(AD1938_SPI_CLK_FREQ, MSBFIRST, SPI_MODE3));

/*enables the SPI interrupts*/
SPI.endTransaction();
```

4. Teensy SPI Slave selection

The SPI slave device is selected by pulling the GPIO pin to low. During the start of the SPI communication, this pin is pulled to LOW and kept low until all the bytes are transmitted. At the end of the SPI transfer once again this pin is pulled to HIGH.

```
digitalWrite(ad1938_clatch, LOW); /*slave select*/

digitalWrite(ad1938_clatch, HIGH);
```

5. SPI data formation for AD1938 codec

AD1938 codec uses the following SPI protocol (as shown in figure 3.5) to exchange register value with SPI master. The global address of AD1938 is 0x4 as per the data sheet. This global address is shifted by one bit and appended with read (1)/write (0) bit. The MSB byte is 0x9 while reading the data and 0x8 while writing the data.

	Global Address	R/$\overline{W}$	Register Address	Data
Bit	23:17	16	15:8	7:0

FIGURE 3.5: SPI data formation

6. Teensy SPI write function

The AD1938 codec register can be configured by sending the 3 bytes using SPI library transfer function.

```
unsigned char data[3];
data[0] = 0x8;
data[1] = 0;
data[2] = 0x3c;
SPI.transfer(&data[0], 3);
```

7. Teensy SPI reading function

To read a register from AD1938 , first send the address of the register along with the read command and then AD1938 will answer the register value. This reading one byte can be achieved by passing 0 to SPI transfer function.

```
unsigned char data[2],reg;
data[0] = 0x9;
data[1] = 0;
SPI.transfer(&data[0], 2);
reg = SPI.transfer(0x00);
```

8. SPI read and write functions

Combing SPI setting, slave select we can develop spi port read function and write functions.

```
/*-------------------------------------------------------------------------*/
       spi_read_reg
/*-------------------------------------------------------------------------*/
unsigned char spi_read_reg(unsigned char reg)
{
        unsigned char result = 0;
        unsigned char data[AD1938_SPI_WRITE_BYTE_COUNT];

        data[0] = AD1938_READ_ADDRESS;
        data[1] = reg;
        data[2] = 0x0;

        // and configure settings
        SPI.beginTransaction(SPISettings(AD1938_SPI_CLK_FREQ,
                                         MSBFIRST,
                                         SPI_MODE3));
        // take the chip select low to select the device:
        digitalWrite(ad1938_clatch, LOW);

    //Send register location , read byte is 2
       SPI.transfer(&data[0], AD1938_SPI_READ_BYTE_COUNT);


     // send a value of 0 to read the first byte returned:
       result =(unsigned char) SPI.transfer(0x00);

       // take the chip select high to de-select:
       digitalWrite(ad1938_clatch, HIGH);
       // release control of the SPI port
       SPI.endTransaction();
```

```
        return(result);

}

/*
    ------------------------------------------------------------------------
    */
/*        spi_write_reg                                                    */
/*
    ------------------------------------------------------------------------
    */
bool spi_write_reg(unsigned char reg, unsigned char val)
{
        unsigned char data[AD1938_SPI_WRITE_BYTE_COUNT];

        /*fill the buffer as per AD1938 format*/
        data[0] = AD1938_WRITE_ADDRESS;
        data[1] = reg;
        data[2] = val;

        // and configure settings
        SPI.beginTransaction(SPISettings(AD1938_SPI_CLK_FREQ, MSBFIRST,
            SPI_MODE3)); // take the chip select low to select the device:
        digitalWrite(ad1938_clatch, LOW);

        SPI.transfer(&data[0], AD1938_SPI_WRITE_BYTE_COUNT);
        // take the chip select high to de-select:
        digitalWrite(ad1938_clatch, HIGH);
        // release control of the SPI port
        SPI.endTransaction();

        return true;

}
```

3.4.2 AD1938 codec control

The AD1938 codec contains 17 registers to configure PLL, ADC, DAC and also for muting and volume increase. Each register is 8 bit length and valid values have to be set according the Ad1938 data sheet.

The high level functions are

1. Config
2. Enable
3. Disable
4. Volume

5. DAC mute
6. ADC mute

1. Config

This function is generalized and take the input parameters like sample rate, number of channels and number of bits per channel similar to ALSA driver. There is one additional parameter to configure each AD1938 codec either I2S master or slave.

CTAG face 2 4 board contains crystal oscillator operating at 512fs (24.576 MHz). This will be used to generate clock in master mode. In slave mode the I2S clock is generated from the ADC LR clock. This function uses the received input parameter to set the AD1938 codec register values.

AD1938 Master register config:

```
/*0 PLL and Clock Control 0 register, Initally internal master clock is
    diabled, master clock rate settin at input 512 (* 44.1KHz or 48KHz),
    master clock is generated by crystal oscilator. */
spi_write_reg(AD1938_PLL_CLK_CTRL0, (DIS_ADC_DAC | INPUT512 | PLL_IN_MCLK |
    MCLK_OUT_XTAL |PLL_PWR_DWN));

/*1 PLL and Clock Control 1 register, DAC clock source select master clock
    MCLK*/
spi_write_reg(AD1938_PLL_CLK_CTRL1, (DAC_CLK_MCLK | ADC_CLK_MCLK | ENA_VREF))
    ;

/*2 DAC Control 0 register, sample rate is set 48khz, Serial data delay is 1,
     Serial format is TDM in daisy chain mode */
spi_write_reg(AD1938_DAC_CTRL0, (DAC_FMT_TDM | DAC_BCLK_DLY_1 | DAC_SR_48K |
    DAC_PWR_UP));

/*3 DAC Control 1 register, Bit clock is internally generated , DAC's LRCLCK
    and BCLK are in slave mode. LCLK polarity is left low, Bit clock per
    frame is 512 (16 channels), BCLK active edge Latch in mid cycle (normal)
    */
spi_write_reg( AD1938_DAC_CTRL1, DAC_BCLK_SRC_INTERNAL |DAC_BCLK_SLAVE|
    DAC_LRCLK_SLAVE |DAC_LRCLK_POL_NORM | DAC_CHANNELS_16 | DAC_LATCH_MID));

/*14 ADC Control 0 register, Output sample rate is 48KHz */
spi_write_reg(AD1938_ADC_CTRL0, ADC_SR_48K);

/*15 ADC Control 1 register, In TDM mode BCLK active edge Latch in mid cycle
    (normal), Serial data delay is 1,16 bit word width */
spi_write_reg(AD1938_ADC_CTRL1, (ADC_LATCH_MID | ADC_FMT_TDM | ADC_BCLK_DLY_1
     | ADC_WIDTH_16));
```

```
/*16 ADC Control 2 register, BCLK and LRCLK is in master mode with internally
     generated bit clcok, for 16 channels,BCLK polarity  Drive out on falling
     edge, LRCLK polarity is left low*/
spi_write_reg(AD1938_ADC_CTRL2, ( ADC_BCLK_SRC_INTERNAL|ADC_BCLK_MASTER |
    ADC_CHANNELS_16 |
                ADC_LRCLK_MASTER   | ADC_LRCLK_FMT_50_50|ADC_LRCLK_POL_NORM|
                    ADC_BCLK_POL_NORM));
```

AD1938 Slave register config:

```
/*0 PLL and Clock Control 0 register, Initally internal master clock is
    diabled, master clock rate settin at input 512 (* 44.1khz or 48khz)*/
spi_write_reg(AD1938_PLL_CLK_CTRL0, (DIS_ADC_DAC | INPUT512 | PLL_IN_ALRCLK |
     MCLK_OUT_OFF |PLL_PWR_DWN));

/*1 PLL and Clock Control 1 register, DAC and ADC clock source select PLL
    clock, On-chip voltage reference is enabled*/
spi_write_reg(AD1938_PLL_CLK_CTRL1, (DAC_CLK_PLL | ADC_CLK_PLL | ENA_VREF));

/*2 DAC Control 0 register, sample rate is set 48khz, Serial data delay is 1,
     Serial format is TDM in daisy chain mode */
spi_write_reg(AD1938_DAC_CTRL0, (DAC_FMT_TDM | DAC_BCLK_DLY_1 | DAC_SR_48K |
    DAC_PWR_UP));

/*3 DAC Control 1  register, take the BCLK source from DBCLK pin, Slave mode,
     LCLK polarity is left low, */
spi_write_reg(AD1938_DAC_CTRL1, ( DAC_BCLK_SRC_PIN|DAC_BCLK_SLAVE|
    DAC_LRCLK_SLAVE  |DAC_LRCLK_POL_NORM | dac_channels | DAC_LATCH_MID));

/*4 DAC Control 2*/
spi_write_reg(AD1938_DAC_CTRL2, DAC_WIDTH_16);

/*14 ADC Control 0 register, Output sample rate is 48KHz*/
spi_write_reg(AD1938_ADC_CTRL0, ADC_SR_48K);

/*15 ADC Control 1 register, In TDM mode BCLK active edge Latch in mid cycle
    (normal), Serial data delay is 0,16 bit word width */
spi_write_reg(AD1938_ADC_CTRL1, (ADC_LATCH_MID | ADC_FMT_TDM | ADC_BCLK_DLY_0
     | ADC_WIDTH_16));

/*16 ADC Control 2 register, BCLK source is ABCLK pin, slave mode, rest are
    similar to master registers*/
spi_write_reg(AD1938_ADC_CTRL2, ( ADC_BCLK_SRC_PIN|ADC_BCLK_SLAVE |
    ADC_CHANNELS_16 | ADC_LRCLK_SLAVE   | ADC_LRCLK_FMT_50_50|
    ADC_LRCLK_POL_NORM|ADC_BCLK_POL_NORM));
```

2. Enable

This function enables the PLL mode to normal mode and set the ADC and DAC units active by setting the PLL Control register

AD1938 Master:

```
// set ENA_ADC_DAC and PLL_PWR_UP
spi_write_reg(AD1938_PLL_CLK_CTRL0, (ENA_ADC_DAC | INPUT512 | PLL_IN_DLRCLK |
    MCLK_OUT_OFF | PLL_PWR_UP));
```

AD1938 Slave:

```
// set ENA_ADC_DAC and PLL_PWR_UP
spi_write_reg(AD1938_PLL_CLK_CTRL0, (ENA_ADC_DAC | INPUT512 | PLL_IN_MCLK |
    MCLK_OUT_XTAL | PLL_PWR_UP));
```

3. Disable

To reduce the power consumption system this functions pull PLL, ADC and DAC units into power down mode and disables the internal clocks of ADC and DAC by writing to the registers.

```
spi_write_reg(AD1938_PLL_CLK_CTRL0, (DIS_ADC_DAC |  reg_value| PLL_PWR_DWN));

spi_write_reg(AD1938_DAC_CTRL0, ((reg_value&0xfe)|DAC_PWR_DWN));

spi_write_reg(AD1938_DAC_CTRL0, ((reg_value&0xfe)|ADC_PWR_DWN));
```

4. Volume

The codec has a precision of 3/8 db for each step. The input parameter is converted accordingly and value is get for all the 8 DAC registers .

0 mean full attenuation
1 means no attenuation

5. DAC mute

Instead of reducing volume step by step , one can mute the all DAC by configuring registers.

```
spi_write_reg(AD1938_DAC_CHNL_MUTE, 0xff);/*mute*/

spi_write_reg(AD1938_DAC_CHNL_MUTE, 00);/*unmute*/
```

6. ADC mute

The ADC input can be muted by configuring the register.

```
spi_write_reg(AD1938_ADC_CTRL0, (reg_value&0xc3)|0x3c);/*mute*/

spi_write_reg(AD1938_ADC_CTRL0, (reg_value&0xc3));/*unmute*/
```

7. Complete AD1938 control class header file

This class contains all the necessary private variable and hence it can be used in multiple instances.

```
class AudioControlAD1938 : public AudioControl
  {
  public:
    bool spiInit(int clatch,int reset,int cout,int cin,int cclk);
    bool config(Te_samplingRate sampleRate,
                Te_bitsPerSample wordLen,
                Te_i2sNumChannels numChannels,
                Te_i2sClockMode mode);
    bool enable(void);
    bool disable(void);
    bool volume(float volume);
    bool adcMute(bool mute);
    bool dacMute(bool mute);
    void readAllreg(void);
    bool inputSelect(int n) {
      return (n == 0) ? true : false;
      }
    bool inputLevel( float volume) {
      return false;
      }
```

```
};
```

3.5 Example Application using AudioControl Ad1938 class

The example Arduino sketch to configure the ad1938 codec using audiocontrolad1938 class is as shown below.

```
#include"control_ad1938.h"
AudioControlAD1938        ad1938master; /*define the class*/

void setup() {

  /*reset codec and init spi pins*/
  ad1938master.spiInit( 7, 17, 12, 11, 14);
  /*configure codec*/
  ad1938master.config(FS_48000,BITS_16,I2S_TDM_16CH,AD1938_I2S_MASTER);
  /*adjust the volume*/
  ad1938master.volume(1);
  /*power up the adc */
  ad1938master.enable();
  }

void loop() {
  // put your main code here, to run repeatedly:

  }
```

3.6 Audio Library TDM Slave Extension

The Audio library of Teensy contains objects classes (as shown in figure 3.6) for receiving and transmitting the audio. The Audio library contains TDM input and output object class for 8 channel input and 8 channel output when Teensy is I2S master.

For the current project of daisy chain, we need 8 channel input and 16 channel output and Teensy as I2S slave. To meet the requirement criteria the AduioOuputTDMSlave and AudioInputTDMSlave classes are developed and they are derived from AduioOuputTDM and AduioInputTDM classes respectively.

3.6.1 AudioOutTDMSlave class

The teensy registers are configured for receiving data in slave mode. The function config_tdm configures the kinetic register.

1. Clock configuration

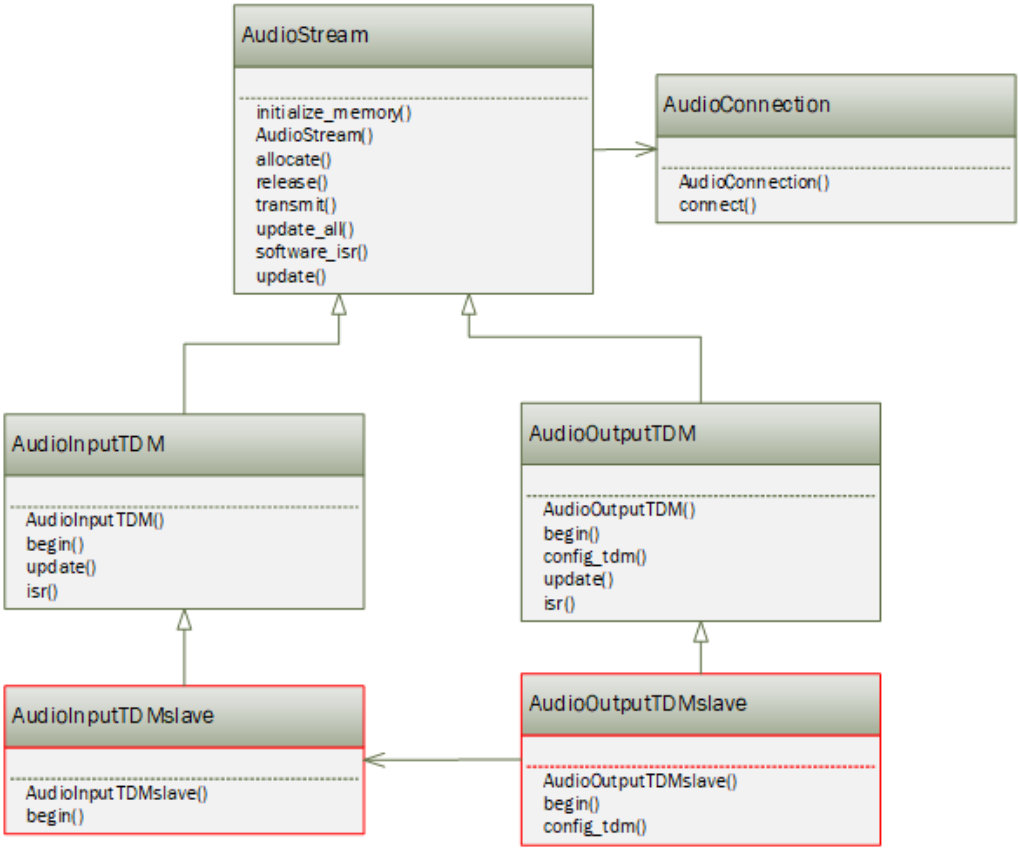

FIGURE 3.6: Audio TDM Class Diagram

- Enable clock for I2S module, DMA mux and DMA by setting the registers below

```
SIM_SCGC6 |= SIM_SCGC6_I2S;
SIM_SCGC6 |= SIM_SCGC6_DMAMUX;
SIM_SCGC7 |= SIM_SCGC7_DMA;
```

- Configure Master clock as slave, selects the input clock zero to the MCLK divider and set the division register to zero

```
I2S0_MCR = I2S_MCR_MICS(0);
I2S0_MDR = 0;
```

2. Transmitter configuration :

```
I2S0_TMR = 0; // Enable all the bits in a word
I2S0_TCR1 = I2S_TCR1_TFW(8) //set the water mark
I2S0_TCR2 = I2S_TCR2_SYNC(0) | I2S_TCR2_BCP; //Configured for
    asynchronous mode of operation and Bit clock is active low with
    drive outputs on falling edge and sample inputs on rising edge.
I2S0_TCR3 = I2S_TCR3_TCE;  // A channel is enabled before FIFO
    operations
```

```
I2S0_TCR4 = I2S_TCR4_FRSZ(15) | I2S_TCR4_SYWD(0) | I2S_TCR4_MF
                | I2S_TCR4_FSE | I2S_TCR4_FSP; // Configure number of
                    channels/ words in a frame , Sync Width, MSB is
                    transmitted first , Frame Sync Early , Frame sync is
                    active low

      I2S0_TCR5 = I2S_TCR5_WNW(31) | I2S_TCR5_W0W(31) | I2S_TCR5_FBT(31);

       // Configure each Word width (except first word of the frame),
           first word width and MSB First Bit Shifted
```

3. Receiver Configuration

```
I2S0_RMR = 0; // configure receive word is enable
I2S0_RCR1 = I2S_RCR1_RFW(8);  //set the water mark
I2S0_RCR2 = I2S_RCR2_SYNC(1) | I2S_TCR2_BCP ;  //Configured for
    Synchronous with transmitter and Clock Polarity is similar to
    transmitter register
I2S0_RCR3 = I2S_RCR3_RCE; // Receive data channel N is enabled.
I2S0_RCR4 = I2S_RCR4_FRSZ(15) | I2S_RCR4_SYWD(0) | I2S_RCR4_MF
                | I2S_RCR4_FSE | I2S_RCR4_FSP| I2S_RCR4_FSD; //
                    Configure number of channels/ words in a frame ,
                    Sync Width, MSB is transmitted first , Frame Sync
                    Early , Frame sync is active low
 I2S0_RCR5 = I2S_RCR5_WNW(31) | I2S_RCR5_W0W(31) | I2S_RCR5_FBT(31); //
      Configure each Word width (except first word of the frame), first
     word width and MSB First Bit Shifted
```

The following function initializes the transmitter DMA configuration and configures the TDM registers

```
void AudioOutputTDMslave::begin(void)
{
        Serial.print("\n AudioOutputTDMslave:begin ");
        dma.begin(true); // Allocate the DMA channel first

        for (int i=0; i < MAX_CHANNELS; i++) {
                block_input[i] = NULL;
        }

        AudioOutputTDMslave::config_tdm();

    // pin 22, PTC1, I2S0_TXD0
        CORE_PIN22_CONFIG = PORT_PCR_MUX(6);

     //Memory address pointing to the transmitter buffer (source data).
        dma.TCD->SADDR = tdm_tx_buffer;

    //  Offset value added to current source address to get the next state
        value, once the source read is completed
```

```
    dma.TCD->SOFF = 4;

//Source and destination transfer size is set to 32-bit
    dma.TCD->ATTR = DMA_TCD_ATTR_SSIZE(2) | DMA_TCD_ATTR_DSIZE(2);

//Defines the number of bytes to transfer per request
 dma.TCD->NBYTES_MLNO = 4;

//once major iteration count completes, a value is added to source
    address. This applied value restore the source address to the initial
     value.
    dma.TCD->SLAST = -sizeof(tdm_tx_buffer);

//points the destination data i.e I2S transmit register
    dma.TCD->DADDR = &I2S0_TDR0;

//offset value of destination address is zero for the memory map register
    .
    dma.TCD->DOFF = 0;

//The channel-to-channel linking is disabled. CITER represent the Current
     Major Iteration Count and it is  decremented for each minor loop
    completion.
    dma.TCD->CITER_ELINKNO = sizeof(tdm_tx_buffer) / 4;

//no linked or chained TCD
    dma.TCD->DLASTSGA = 0;

//Starting Major Iteration Count.Initially BITER has same value as CITER
    and BITER value is added in to CITER once major iteration count is
    exhausted
    dma.TCD->BITER_ELINKNO = sizeof(tdm_tx_buffer) / 4;

//Enables the interrupt once the major iteration count is half or full
    completed
    dma.TCD->CSR = DMA_TCD_CSR_INTHALF | DMA_TCD_CSR_INTMAJOR;

//Connects the source DMA to I2S transmit
    dma.triggerAtHardwareEvent(DMAMUX_SOURCE_I2S0_TX);
     update_responsibility = update_setup();

// Once TCD registers are defined , DMA is enabled
    dma.enable();

    //reset the transfer control register
    I2S0_TCSR = I2S_TCSR_SR;

//Enable the tranmitter, Bit clock and DMA request
    I2S0_TCSR = I2S_TCSR_TE | I2S_TCSR_BCE | I2S_TCSR_FRDE;

//Map the Interupt service routine to DMA
dma.attachInterrupt(isr);
```

```
}
```

3.6.2 AudioInputTDMSlave class

The following function initializes the receiver DMA configuration and configures the TDM registers

```
void AudioInputTDMslave::begin(void)
{
        //Serial.print("\n AudioInputTDMslave:begin ");
        dma.begin(true); // Allocate the DMA channel first

        AudioOutputTDMslave::config_tdm();

// pin 13, PTC5, I2S0_RXD0
        CORE_PIN13_CONFIG = PORT_PCR_MUX(4);

    //Memory address pointing to the I2S Reciever register
        dma.TCD->SADDR = &I2S0_RDR0;

   //offset value of source address is zero for the memory map register.
        dma.TCD->SOFF = 0;

    //Source and destination transfer size is set to 32-bit
        dma.TCD->ATTR = DMA_TCD_ATTR_SSIZE(2) | DMA_TCD_ATTR_DSIZE(2);

   //four bytes are recieved per request
        dma.TCD->NBYTES_MLNO = 4;

    //the value for memory map is zero
        dma.TCD->SLAST = 0;

    //points the destination data i.e rceiver buffer
        dma.TCD->DADDR = tdm_rx_buffer;

   //   Offset value added to current destination address to get the next
       state value, once the recieved read is completed
        dma.TCD->DOFF = 4;

    //The channel-to-channel linking is disabled. CITER represent the Current
         Major Iteration Count andit is  decremented for each minor loop
        completion.
        dma.TCD->CITER_ELINKNO = sizeof(tdm_rx_buffer) / 4;

    //Adjustment value added to destination address at the completion of
       major cont iteration and set it to initial value
        dma.TCD->DLASTSGA = -sizeof(tdm_rx_buffer);
```

```
  //Starting Major Iteration Count.Initially BITER has same value as CITER
      and BITER value is added in to CITER once major iteration count is
      exhausted
        dma.TCD->BITER_ELINKNO = sizeof(tdm_rx_buffer) / 4;

   //Enables the interrupt once the major iteration count is half or full
       completed
        dma.TCD->CSR = DMA_TCD_CSR_INTHALF | DMA_TCD_CSR_INTMAJOR;

     //Connects the source DMA to I2S receiver
        dma.triggerAtHardwareEvent(DMAMUX_SOURCE_I2S0_RX);
        update_responsibility = update_setup();

    // Once TCD registers are defined , DMA is enabled
    dma.enable();

        //Enable receiver, Bit Clock, DMA Request and Reset FIFO
        I2S0_RCSR |= I2S_RCSR_RE | I2S_RCSR_BCE | I2S_RCSR_FRDE | I2S_RCSR_FR
            ;

   // TX clock enable, because sync'd to TX
        I2S0_TCSR |= I2S_TCSR_TE | I2S_TCSR_BCE;

     //Map the Interupt service routine to
     DMAdma.attachInterrupt(isr);

}
```

3.7 TDM audio data flow

The below diagram 3.7 explains the TDM audio data flow in Teensy Audio application

The AudioInputTDM class receives the data from the I2S_RDR0 register to rx_buffer through the eDMA. When rx_buffer is half full the DMA generates interrupt. In the interrupt service routine (ISR) the data is separated into different channels and copied into individual buffers. The Audio library of Teensy contains "AudioConnect" function which copies data between two objects.

The AudioOputTDM class transmit the data from tx_buffer to I2S_TDR0 through the eDMA. When tx_buffer is half emptied the DMA generates interrupt. In the interrupt service routine (ISR) the data is combined from different channels and copied to tx_buffer. When there is no data available then zeros are copied to the tx_buffer.

The PCM pass through application is explained in the form of a flow chart below.(see Figure 3.8)

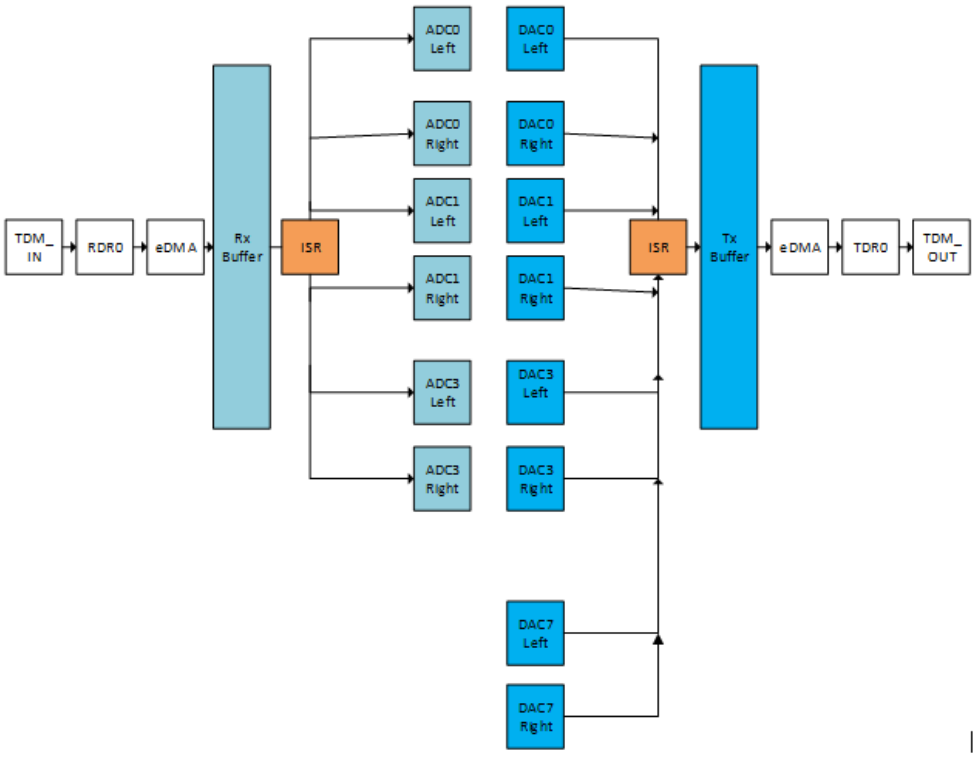

FIGURE 3.7: TDM audio flow class

3.8 Reverb Algorithm

This thesis also includes implementation of freeverb algorithm (reverberation effect) for the teensy audio library in c++. The figure 3.9 shows the implemenation of freeverb algorithm using teensy objects.

This implementation uses the input signal from I2S data coming from audio driver **AudioInputTDMslave** class. This signal is fed to mixture AudioMixer4 class (mixer1) which combines left and right channel with gain multiplication. The output of the mixer1 is given to two freeverb objects (**AudioEffectFreeverb** class), one for left channel reverb and other for the right channel reverb. The Left output can be derived from mixing the direct input signal (left dry), reverberated left signal (wet1) and reverberated right signal (wet2). Similarly the right output can be derived from mixing the direct input signal (right dry), reverberated right signal (wet1) and reverberated left signal (wet2) with appropriate gains received from the user plugin. The left output from mixer2 and right output from mixer3 are given to output of audio driver **AudioOutputTDMslave** class to obtain the stereo output signal.

The freeverb objects is drived form the audio stream base class as shown in figure 3.10.

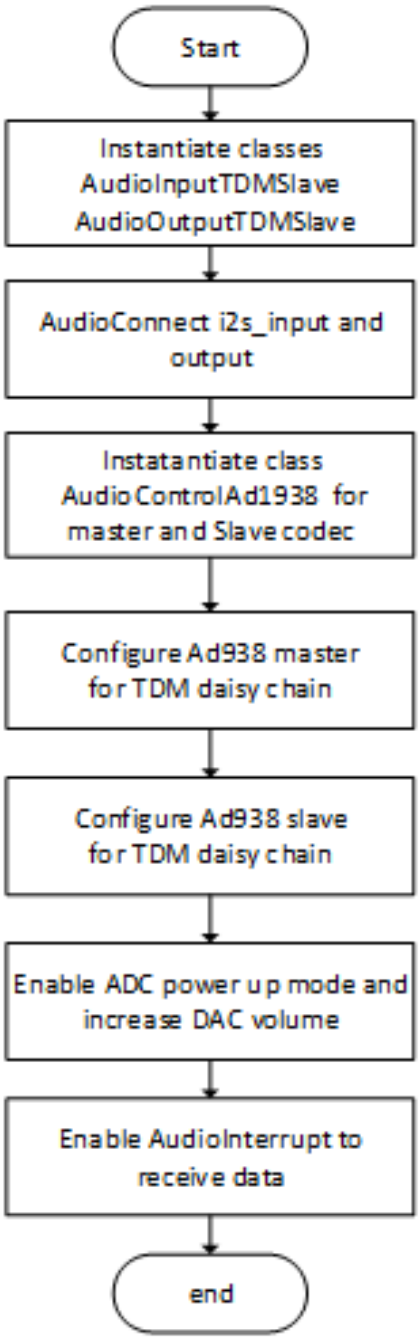

FIGURE 3.8: Flow chart: The PCM pass through application

The AudioEffectFreeverb class contains initialization and the processing for all-pass filter, lowpass comb filter. The flow of the freeverb is presented in the update function. All the filter implementations are done using fixed point arithmetics. The functions are explained below

prcess_lbcf function (Lowpass com filter implementation)

This function implements the difference equations 2.21 of lowpass comb filter.

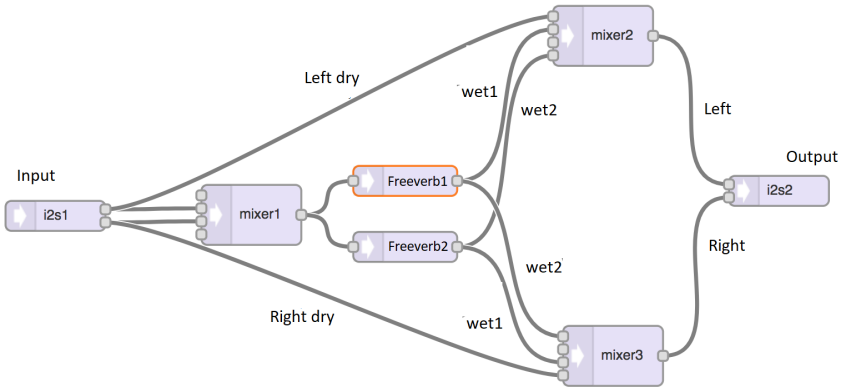

FIGURE 3.9: Freeverb using Teensy objects

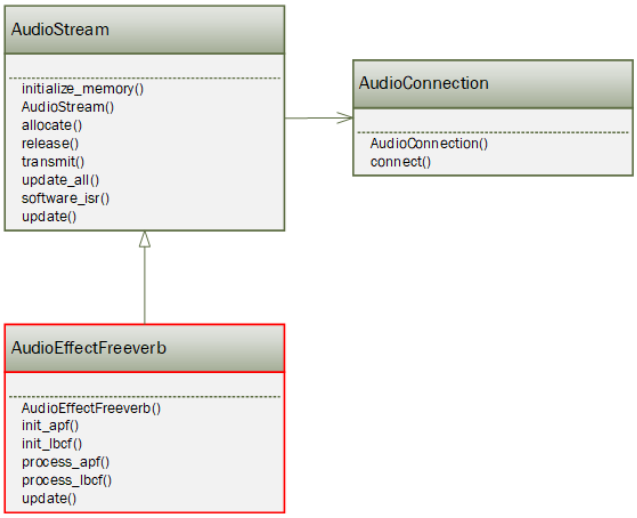

FIGURE 3.10: Freeverb object AudioEffectFreeverb

```
/*
output = w(n-M);
z1 = (output * (1-d)) + (z1 * d);
w(n-M) = input + (z1 * f);
*/
for (n = 0; n < AUDIO_BLOCK_SAMPLES; n++)
{
        bufout = lbcf->pbuffer[lbcf->bufferIndex];
        input = in_buf[n];

        sum = multiply_32x32_rshift32_rounded( bufout, lbcf->damp2);
        sum2= multiply_32x32_rshift32_rounded(lbcf->state_z1, lbcf->damp1);
        lbcf->state_z1 = ((sum + sum2) << 2);
```

```
        sum = multiply_32x32_rshift32_rounded(lbcf->state_z1, lbcf->feedback)
            ;
        sum2 = input + (sum << 2);
        lbcf->pbuffer[lbcf->bufferIndex] = sum2;
        out_buf[n] = bufout;
        lbcf->bufferIndex++;
        if (lbcf->bufferIndex >= lbcf->delay)
        {
                lbcf->bufferIndex = 0;
        }
}
```

prcess_apf function (All-pass filter implementation)

This function implements the difference equations 2.11 of all-pass filter.

```
/*
bufout = w(n-M);
w(n-M) = input + bufout * g;
output = -w(n-M) * g + bufout;
*/
for (n = 0; n < AUDIO_BLOCK_SAMPLES; n++)
{
        bufout = apf->pbuffer[apf->bufferIndex];
        input = in_buf[n];
        z1 = multiply_32x32_rshift32_rounded( bufout, apf->gain);
        input += (z1 << 2);
        w= multiply_32x32_rshift32_rounded(-input, apf->gain);
        output =bufout +(w << 2);
        out_buf[n] = output;
        apf->pbuffer[apf->bufferIndex] = input;
        apf->bufferIndex++;
        if (apf->bufferIndex >= apf->delay)
        {
                apf->bufferIndex = 0;
        }
}
```

update

The update function combines eight parallel lowpass comb filter and four cascaded all-pass filter. Each filter is operated on 128 samples as per the requirement of teensy library.

```
/*input 16 bit is convereted to 32 bit */
arm_q15_to_q31(block->data, q31_buf, AUDIO_BLOCK_SAMPLES);

/*eight parallel lowpass comb filters*/
process_lbcf(&lbcf[0], q31_buf, sum_buf);
```

```
process_lbcf(&lbcf[1], q31_buf, aux_buf);

arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

process_lbcf(&lbcf[2], q31_buf, aux_buf);
arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

process_lbcf(&lbcf[3], q31_buf, aux_buf);
arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

process_lbcf(&lbcf[4], q31_buf, aux_buf);
arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

process_lbcf(&lbcf[5], q31_buf, aux_buf);
arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

process_lbcf(&lbcf[6], q31_buf, aux_buf);
arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

process_lbcf(&lbcf[7], q31_buf, aux_buf);
arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

/*four cascaded all-pass filters*/
process_apf(&apf[0], sum_buf, q31_buf);
process_apf(&apf[1], q31_buf, q31_buf);
process_apf(&apf[2], q31_buf, q31_buf);
process_apf(&apf[3], q31_buf, q31_buf);

/* filter output 32 bit is converted to 16 bit */
arm_q31_to_q15(q31_buf, block->data, AUDIO_BLOCK_SAMPLES);
```

Chapter 4

Evaluation

4.1 Latency

Definition Audio latency is the delay between sound being triggered and then actually perceived. The reasons for the cause of latency in the audio system are ADCs, DACs, buffering, digital signal transmission, transmission time, the speed of sound and the transmission medium. Round-trip time (RTT), also called round-trip delay, is the time required for a signal pulse or packet to travel from a specific source to a specific destination and back again. [48].

Evaluation:

The teensy library has audio objects works on the block of 128 samples of 16 bits per sample. In the receiver interrupt the audio samples are copied when the receiver DMA buffer is half filled and in the transmitter interrupt audio samples are sent out when transmitter DMA buffer is half filled.

The round-trip time in the driver is Rx DMA half buffer delay + Tx DMA half buffer delay. The receiver and transmitter DMA buffer of type 32 bits and it is not changed to make it compatible with other configurations.

RTT = 2*128 +2*128=512 samples (16 bits per sample)

RTT =512/48 = 10.666 ms

By using oscilloscope the latency can be measured. Oscilloscope captures the transmitted and received signal, the figure 4.1 shows the delay between signal and it is around **9ms**.

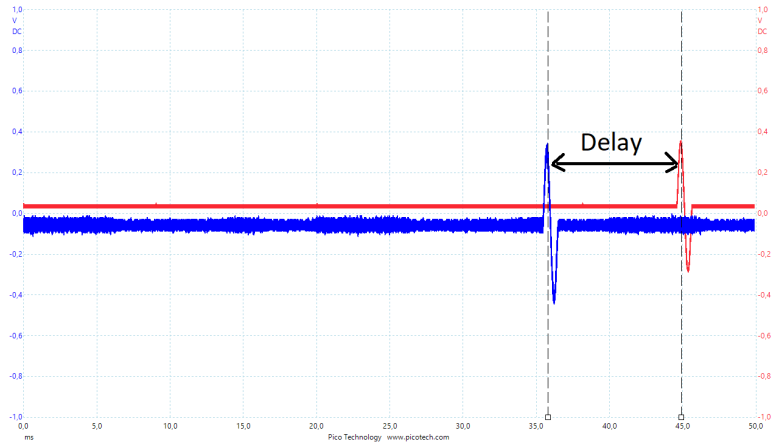

FIGURE 4.1: Round trip time

4.2 Buffer Size

Definition:

When recording audio into your computer, your sound card needs some time to process the incoming information. The amount of time allotted for processing is called the Buffer Size[49]. In general low buffer size is preferred as it limits the latency but sometimes it results in a higher burden on the system as it has very little time to process the audio. When introducing the larger buffer size, the audio delay is generated. So it is important to find the appropriate buffer size for the session as this can vary depending on the number of tracks, plug-ins, audio files etc..

Evaluation:

During testing the DMA buffer size is increased from 128 to 256 samples per channel and by using the oscilloscope the delay between input and the ouput signal is measured (as shown in figure 4.2).The measured delay is around 18ms. As observed when the DMA buffer size has increased the delay also increases. Smaller DMA size has less latency but this will increase the interrupts and CPU load. Hence optimal DMA buffer size must be chosen.

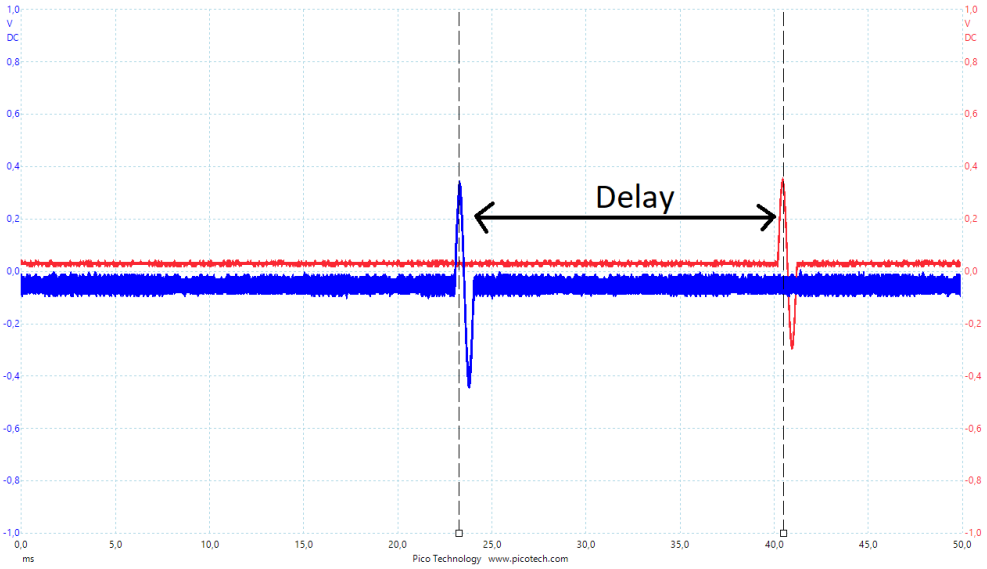

FIGURE 4.2: Input and output pluse signal with delay

4.3 DSP Benchmark using CPU

CPU Benchmark is very complex to define, this can be measured by a full application or a single operation [50]. The CPU performance is measured by number of MAC (multiply and accumulate) operations. These MAC operations are limited by memory access, pipeline latencies, algorithm feedback requirements, application tasks, operations system Kernel system calls.

DSP Benchmarks are measured for the DSP operations like FFT, DCT, FIR. They typically measured in MIPS (Million instructions per second) or DMIPS (Dhrystone-MIPS). For the ARM processor, DMIPS are used.

Evaluation

The CPU consumption and memory usage can be measured using the available Teensy Audio library API's listed below

- processorUsage
- processorUsageMax
- AudioMemoryUsage
- AudioMemoryUsageMax

These APIs calculates the number of CPU cycles taken for processing 128 samples at the 44.1khz sample rate. The figure shows the CPU consumption and memory utilization by I2S Driver and Freeverb Algorithm.

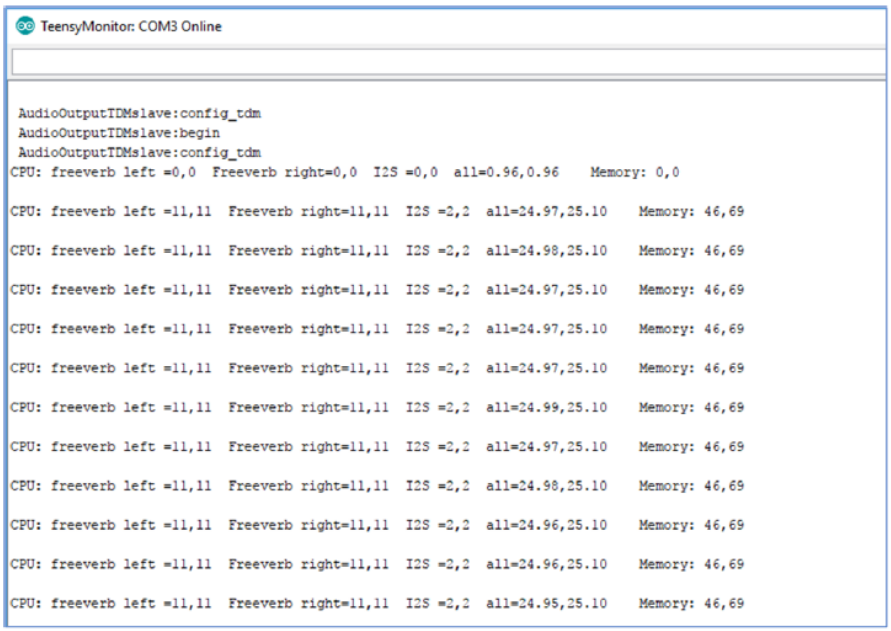

FIGURE 4.3: Serial terminal log

4.4 Benefits of DMA

- DMA permits the peripherals, to transfer data directly to or from memory without CPU involvement
- Kinetic DMA has very flexible control over the data width, data size, and minor and major loops.
- DMA also generates interrupts at a regular interval (ping pong) during the transfer of data. In this thesis, interrupt are generated when half of the buffer is transferred.

4.5 Memory foot-print

The following table shows the measured memory details

Component	Code	Data	Details
SPI	1.15kB	0.05kB	master and slave Instances
Audio Driver	3.6kB	97.8kB	16 channel PCM Pass-through
Freeverb	3.6kB	104.6kB	Stereo freeverb

TABLE 4.1: Memory consumption

4.6 Power Consumption

In electrical engineering, power consumption is the amount of input energy consumed by the electrical appliances, usually expressed in units of watts (W) or kilowatts (kW).

Evaluation Using multimeter the current and the voltage for the each component is measured between the Vcc and GND in the full operating mode. The measured values are summarized in the table 4.2

Component	**Voltage (V)**	**Current (mA)**	**Power (watts)**
Teensy	3.3	80 (180 MHz)	0.264
AD1938 (master)	5	100	0.5
AD1938 (slave)	5	100	0.5

TABLE 4.2: power consumption

Chapter 5

Discussion / Limitations

5.1 Discuss results and findings

This section discusses the results and findings of this thesis. The discussion is presented in two primary parts. First, development of Audio Driver. Second, Implementation of Freeverb Algorithm.

During the study of this thesis, I did research on open source ecosystem. The open source software has to be developed in time, must be more generic, it should be freely available to all the programmer with non-restricted or limited restriction license and must provide freedom to modify, review and redistribute the software or code. This kind of approach is must faster and more efficient and results in better quality.

I got familiar with the open source operating system and their licenses. Did the basic study on a linux based platform like BeagleBoard, Raspberry Pi, Arduino Yún and Intel Galileo. Also studied non-Linux based embedded devices like Arduino, Adafruit Flora, LightBlue Bean.

During the course of the thesis, I have studied the teensy data specifications and various generations of teensy hardwares. I found that teensy 3.6 very powerful microcontroller and compatible with Arduino software and libraries. The open source Teensy audio library is distributed under MIT-like license.

Through my literature study, I found out that the Teensy Audio library has support for the control of various multi-channel audio codecs. All the codec are controlled using Inter-Integrated Circuit (I2C) bus and codecs are configured as Inter-IC Sound (I2S) slave and teensy as Inter-IC Sound master with the maximum support of 8 channels in Time-division multiplexing mode.

Since teensy audio library provides no support for the AD1938 codec, I have developed the AD1938 audio control class and it is derived from the base class "AudioControl" which is present in the audio library. This class has two function, SPI port

control and AD1938 codec control. I have developed API's of SPI read and write functions for AD1938. The AD1938 codec contains 17 registers to configure Phase-locked loops (PLL), Analog to digital converter (ADC), Digital to Analog converter (DAC). I have configured these registers to achieve high-level function like configure, Enable, Disable, Volume, Mute and Unmute.

To support daisy chain mode, I have developed multi-instance Serial Peripheral Interface(SPI) control class to configure two AD1938 codecs simultaneously.

Extensively studied Teensy audio library data flow and I2S input/ output classes and also studied the MK66FX1M0VMD18 Kinetics processor control registers to configure Direct memory access and Inter-IC Sound clocks.

The audio library of teensy contains object classes for receiving and transmitting the audio. when teensy is master, this library has support for 8 input channels and 8 output channels. The two main objectives of this thesis are to make teensy as slave and to connect two codecs and teensy in daisy chain mode as shown in figure 3.2 and generate 8 input channel and 16 output channels. To achieve the above objectives, the AduioOuputTDMSlave and AudioInputTDMSlave classes are developed which configures clock, transmitter, receiver and enhanced Direct Memory Access. By using Pulse-code modulation (PCM) pass through code and synthesized sine tone, the developed I2S driver is tested and verified for 8 input channels and 16 output channels.

In the second part of the thesis, the reverb effect is added to the teensy audio library. During the study phase, many artificial digital reverberation algorithms like Schroeder, Moorer, Gardner and others were studied. It is observed that the comb filters produce the long echoes that occur between the walls. but parallel comb filter doesn't produce enough echo density as in realistic solution. To increase the echo density the parallel comb filter output is fed to all-pass filters connected in series. These all-pass filter multipliers the echoes. The low pass filter is added to the comb filter to simulate the frequency dependent air absorption and increase echo density. The freeverb open source algorithm, developed by Jezar has similar structure like Schroeder-Moorer reverberator. This algorithm has eight lowpass comb filters in parallel and four all-pass filters in series.

The Freeverb Algorithm for Teensy is implemented using Fixed point arithmetics so that the source code developed can be used with all teensy variants, which doesn't have floating processing unit. The fixed point implementation also reduces the CPU computational and increases the precision for both left and right channels. The reverberation effect is analyzed by varying the feedback, damping factors of comb filter and gain of the all-pass filter. The freeverb Algorithm is tested in real time by giving I2S input and verified the effects by listening to the I2S output channel.

5.2 what could not be achieved

There are few limitations of this thesis. First, this thesis are the audio driver is tested with 48KHz sampling rate only, higher sampling rates like 96KHz, 128KHz more are not tested. Second, freeverb is implemented with the predefined fixed delay lengths, rather than variable delay length so the memory consumption for various reverberation time cannot be evaluated. Third, in freeverb, the precision between the fixed point and the floating point implementation are not evaluated. and the last, it was difficult to publish the source code in the teensy open source audio library, as the moderators did not respond to my mail.

Chapter 6

Conclusion

6.1 Conclusion

The main purpose of this thesis was to develop an audio driver for teensy using soundcard CTAG 2/4 and implement the freeverb algorithm. The Audiocontrol class using SPI drivers was developed to initialize and configure the AD1938 codec for various I2S modes. This Audiocontrol class also supports volume control and mute. Analyzed the existing I2S input/ output TDM class for the I2S master and developed the derived I2S Input/output TDM in slave mode. Later, this class was extended to support 16 input channels and 16 output channels. By understanding the AD1938 control registers, two soundcards are connected in daisy chain mode which supports 8 input audio channels and 16 audio output channels. The I2S driver latency is measured using an oscilloscope as 10.66 ms. A PCM pass-through code with existing audio library objects is created and tested. As the system supports only 8 input channels, internally generated sine tone was used to verify the other output channels. After the PCM pass through, this thesis added reverb effect using the open source freeverb algorithm in fixed-point arithmetic for the fixed delay length. This algorithm is tested in the real time by giving I2s audio input and varied the reverberation effects.

As a future extension one can adjust the delay length to increase the reverberation time. In the current implementation, the internal delay buffers are of size 32 bits width, the buffer width can be carefully optimized to 16 bit. There is a scope for the optimization in the interrupt service routine(ISR) of transmitter and receiver.

Bibliography

[1] *Understanding and implementation of open source ecosystems final*. URL: https://www.slideshare.net/RachitTechnologyPvtL/understanding-and-implementation-of-open-source-ecosystems-final.

[2] *The Open Source Definition*. URL: https://opensource.org/osd-annotated.

[3] *Raspberry pi model B*. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[4] *Arduino Yún LininoOS*. URL: https://www.arduino.cc/en/Main/ArduinoBoardYun?from=Main.ArduinoYUN.

[5] *Intel Galileo*. URL: https://www.arduino.cc/en/ArduinoCertified/IntelGalileo.

[6] *FLORA - Wearable electronic platform: Arduino-compatible - v3*. URL: https://www.adafruit.com/product/659.

[7] *LightBlue Bean*. URL: https://www.adafruit.com/product/2732.

[8] PlatformIO. *Teensy*. Revision 9fc5aecb, 2014. URL: http://docs.platformio.org/en/latest/platforms/teensy.html.

[9] *Teensy USB Development Board*. URL: https://www.pjrc.com/teensy/.

[10] Tommaso Melodia G. Enrico Santagati. "A Software-Defined Ultrasonic Networking Framework for Wearable Devices". In: *IEEE/ACM TRANSACTIONS ON NETWORKING* (2016).

[11] *Audio Connections and Memory*. URL: https://www.pjrc.com/teensy/td_libs_AudioConnection.html.

[12] Philips Semiconductors. *I2S bus specification*. High Tech Campus 60 5656 AG Eindhoven, NoordBrabant, The Netherlands., june 5, 1996 edition. URL: https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf.

[13] Cirrus Logic. *Time Division Multiplexed Audio Interface*. 2006. URL: https://d3uzseaevmutz1.cloudfront.net/pubs/appNote/AN301REV1.pdf.

[14] Corelis An EWA Company. *Serial Peripheral Interface (SPI) Bus*. URL: https://www.corelis.com/whitepapers/BusPro-S_SPI_Tutorial.pdf.

[15] Leens F. "An introduction to I2C and SPI protocols". In: *IEEE Instrumentation & Measurement Magazine* (2009).

[16] *Direct Memory Access (DMA) Modes and Bus Mastering DMA*. URL: http://www.pcguide.com/ref/hdd/if/ide/modesDMA-c.html.

[17] A. F. Harvey and National Instruments Corporation Data Acquisition Division Staff. *DMA Fundamentals on Various PC Platforms*. April 1991. URL: http://cires1.colorado.edu/jimenez-group/QAMSResources/Docs/DMAFundamentals.pdf.

[18] *Linux-Based Low-Latency Multichannel Audio System (CTAG face2|4)*. URL: http://www.creative-technologies.de/linux-based-low-latency-multichannel-audio-system-2/.

[19] Analog Devices. *AD1938 Datasheet*. Inc., P.O. Box 9106, Norwood, MA 02062-9106, U.S.A., rev. e edition, 2013. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/AD1938.pdf.

[20] *Teensy 3.6 Website*. URL: https://www.pjrc.com/store/teensy36.html.

[21] Inc. Freescale Semiconductor. *K66 Sub-Family Reference Manual*. Core Electronics, 2015. URL: https://www.pjrc.com/teensy/K66P144M180SF5RMV2.pdf.

[22] *Naming Conventions for Audio Objects*. URL: https://www.pjrc.com/teensy/td_libs_AudioNamingConvention.html.

[23] *Creating New Audio Objects*. URL: https://www.pjrc.com/teensy/td_libs_AudioNewObjects.html.

[24] Inc. Freescale Semiconductor. *eDMA for Kinetis K series MCUs*. Core Electronics, Rev 1.0 2015. URL: https://www.nxp.com/docs/en/supporting-information/Enhanced-Direct-Memory-Access-Controller-Training.pdf.

[25] Vikash Sethia Bhaskar Goswami. "Using ADS8410/13 in Daisy-Chain Mode". In: *Application Report* (SLAA296–May 2006).

[26] *Network topology wiki*. URL: https://en.wikipedia.org/wiki/Network_topology#Daisy_chain.

[27] Margaret Rouse. *daisy chain*. TechTarget, 2000. URL: {http://searchnetworking.techtarget.com/definition/daisy-chain},.

[28] *Finite impulse response*. URL: https://en.wikipedia.org/wiki/Finite_impulse_response.

[29] *Infinite impulse response*. URL: https://en.wikipedia.org/wiki/Infinite_impulse_response.

[30] Rajesh Mehra Monika Singh Saroj Dogra. "IIR Filter Design and Analysis using Notch and Comb Filter". In: *International Journal of Scientific Research Engineering & Technology (IJSRET)* (September 2013).

[31] Hank Zumbahlen. *Allpass Filters*. Analog Device, One Technology Way • P.O. Box 9106 •Norwood, MA 02062-9106, U.S.A, 2012. URL: http://www.analog.com/media/en/training-seminars/tutorials/MT-202.pdf.

[32] Jun Yang. *Lecture 3 Floating Point Representations*. URL: http://www.pitt.edu/~juy9/142/slides/L3-FP_Representation.pdf.

[33] K. Brandenburg M. Kahrs. "Applications of Digital Signal Processing to audio and acoustics". In: *Kluwer Academic Publishers* (1998).

[34] Farzad Foroughi Abari. "Optimization of Audio Processing algorithms (Reverb) on ARMv6 family of processors". In: *Blekinge Institute of Technology* (October 2007).

[35] Will Pirkle. *Designing Audio Effect Plug-Ins in C++ With Digital Audio Signal Processing Theory*. Focal Press, 2013.

[36] DENNIS FOLEY. *Early Reflections Vs Reverb – Why Do They Matter?* JULY 9, 2014. URL: https://www.acousticfields.com/early-reflections-vs-reverb-why-do-they-matter/.

[37] *Reverberation*. URL: https://en.wikipedia.org/wiki/Reverberation.

[38] Vesa et al Välimäki. "Fifty Years of Artificial Reverberation". In: *IEEE Transactions on Audio, Speech, and Language Processing. Vol 20, No. 5, Pp. 1421-1448* ().

[39] William Grant Gardner. "The Virtual Acoustic Room". In: *S.B., Computer Science and Engineering, Massachusetts Institute of Technology, Cambridge* (August 10, 1992).

[40] James A. Moorer. "About this Reverberation Business". In: *Computer Music Journal, Vol. 3, No. 2, Pp. 13-28* (June 1979).

[41] Julius O. Smith. *Physical Audio Signal Processing*. online book, 2010 edition. http://-ccrma.stanford.edu/~jos/pasp/, accessed <date>.

[42] *Add Reverberation Using Freeverb Algorithm*. URL: https://www.mathworks.com/help/audio/examples/add-reverberation-using-freeverb-lgorithm.html?s_tid=gn_loc_drop.

[43] Alex Blair Whitler. "GatorVerb: Modeling Algorithmic Reverb in a Virtual Studio Technology Plugin". In: (spring 2011).

[44] M. R. Schroeder. "Natural Sounding Artificial Reverberation". In: *J. Audio Engineering Society* (Vol. 10, No 3 (1962)).

[45] William G. Gardner. *Applications of Digital Signal Processing to Audio and Acoustics. Reverberation Algorithms*. The International Series in Engineering and Computer Science, vol 437. Springer, Boston, MA, 2002.

[46] Julian; Savioja Lauri; Smith Julius O.; Abel Jonathan Välimäki Vesa; Parker. "More Than 50 Years of Artificial Reverberation". In: *AES Conference:60th International Conference: DREAMS* (January 27, 2016).

[47] Sam. *Using Teensy with Arduino IDE*. Core Electronics, 2016. URL: https://core-electronics.com.au/tutorials/using-teensy-with-arduino-ide.html.

[48] Margaret Rouse. *Round-trip time (RTT)*. TechTarget, April 2007. URL: http://searchnetworking.techtarget.com/definition/round-trip-time.

[49] Inc. iZotope. *What is Buffer Size and why is it important?* 2001-2017. URL: https://www.izotope.com/en/support/knowledge-base/what-is-buffer-size-and-why-is-it-important.html.

[50] *What to Look for in DSP Benchmarks*. URL: https://www.eetimes.com/document.asp?doc_id=1275300.

Appendix A

Appendix

The source code can be found on the github

1. Audio control class https://github.com/yasmeensultana/ad1938_codec
2. Freeverb https://github.com/yasmeensultana/freeverb

Control_ad1938.h

```
/* AD1938 Audio Codec  control library
*
* Copyright (c) 2017, Yasmeen Sultana
*
*
* Permission is hereby granted, free of charge, to any person obtaining a
    copy
* of this software and associated documentation files (the "Software"), to
    deal
* in the Software without restriction, including without limitation the
    rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice, development funding notice, and this permission
* notice shall be included in all copies or substantial portions of the
    Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*/

#ifndef _CONTROL_AD1938_H_
#define _CONTROL_AD1938_H_

#include "AudioControl.h"
```

```
/*SPI */
#define AD1938_SPI_CLK_FREQ       1000000
#define AD1938_SPI_CHIP_SEL       7  /*teensy 3.6 gpio*/
#define AD1938_SPI_SCK            14 /*teensy 3.6 gpio*/
#define AD1938_RESET              17 /*teensy 3.6 gpio*/

/*sampling rate*/
typedef enum
{
  FS_32000,
  FS_44100,
  FS_48000,
  FS_64000,
  FS_88200,
  FS_96000,
  FS_128000,
  FS_176400,
  FS_192000,
}Te_samplingRate;

/*number of bits per sample*/
typedef enum
{
  BITS_16,
  BITS_20,
  BITS_24,
}Te_bitsPerSample;

/*I2s clock mode*/
typedef enum
{
  AD1938_I2S_SLAVE,
  AD1938_I2S_MASTER,
}Te_i2sClockMode;

/*I2s number of channels */
typedef enum
{
  I2S_STEREO_2CH,
  I2S_TDM_8CH,
  I2S_TDM_16CH
}Te_i2sNumChannels;


class AudioControlAD1938 : public AudioControl
{
public:
        bool spiInit(int clatch,int reset,int cout,int cin,int cclk);
        bool config(Te_samplingRate sampleRate,Te_bitsPerSample wordLen,
            Te_i2sNumChannels numChannels, Te_i2sClockMode mode);
        bool enable(void);
```

```
        bool disable(void);
        bool volume(float volume);
        bool adcMute(bool mute);
        bool dacMute(bool mute);
        void readAllreg(void);
        bool inputSelect(int n) {
                return (n == 0) ? true : false;
        }
        bool inputLevel( float volume) {
                 return false;
        }
private:
        int ad1938_clatch;
        int ad1938_reset;
        int ad1938_cout;
        int ad1938_cin;
        int ad1938_cclk;
        Te_i2sClockMode    i2sMode;
        Te_bitsPerSample   wordLen;
        Te_i2sNumChannels numChannels;
        Te_samplingRate    samplingRate;

protected:
        bool spi_write_reg(unsigned char reg, unsigned char val);
        unsigned char spi_read_reg(unsigned char reg);
        bool isPllLocked();
        bool dacVolume(int dac_num,int volume);

};

#endif // !_CONTROL_AD1938_H_
```

control_ad1938.cpp

```
/* AD1938 Audio Codec  control library
*
* Copyright (c) 2017, Yasmeen Sultana
*
*
* Permission is hereby granted, free of charge, to any person obtaining a
    copy
* of this software and associated documentation files (the "Software"), to
    deal
* in the Software without restriction, including without limitation the
    rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice, development funding notice, and this permission
* notice shall be included in all copies or substantial portions of the
    Software.
*
```

```
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*/
#include"control_ad1938.h"

#include <SPI.h>


/*
reference
http://www.analog.com/media/en/technical-documentation/data-sheets/AD1938.pdf
http://www.analog.com/media/en/technical-documentation/application-notes/AN
    -1365.pdf
*/
/* SPI 3 byte register format
-------------------------------------------------
|Global Address |R/W |Register Address |Data |
-------------------------------------------------
|23:17          |16  |15:8             |  7:0|
-------------------------------------------------
Address Register
0               PLL and Clock Control 0
1               PLL and Clock Control 1
2               DAC Control 0
3               DAC Control 1
4               DAC Control 2
5               DAC individual channel mutes
6               DAC L1 volume control
7               DAC R1 volume control
8               DAC L2 volume control
9               DAC R2 volume control
10              DAC L3 volume control
11              DAC R3 volume control
12              DAC L4 volume control
13              DAC R4 volume control
14              ADC Control 0
15              ADC Control 1
16              ADC Control 2
*/

/*The global address for the AD1938 is 0x04, shifted left one bit due to the
    R/W bit.*/
#define AD1938_GLOBAL_ADDRESS    0x04

#define AD1938_WRITE_ADDRESS     (AD1938_GLOBAL_ADDRESS<<1)
#define AD1938_READ_ADDRESS      ((AD1938_GLOBAL_ADDRESS<<1)|1)
```

```
#define AD1938_SPI_WRITE_BYTE_COUNT    3
#define AD1938_SPI_READ_BYTE_COUNT     2

/*
    ---------------------------------------------------------------------------
    */
/*PLL and Clock Control 0
                                                                        */
/*
    ---------------------------------------------------------------------------
    */
#define AD1938_PLL_CLK_CTRL0     0x00

/*
Bit Value Function Description
0 0 Normal operation PLL power-down
1 Power-down
2:1 00 INPUT 256 ( 44.1 kHz or 48 kHz) MCLKI/XI pin functionality (PLL active
    ), master clock rate setting
    01 INPUT 384 (44.1 kHz or 48 kHz)
    10 INPUT 512 ( 44.1 kHz or 48 kHz)
    11 INPUT 768 ( 44.1 kHz or 48 kHz)
4:3 00 XTAL oscillator enabled MCLKO/XO pin, master clock rate setting
    01 256 * fS VCO output
    10 512 * fS VCO output
    11 Off
6:5 00 MCLKI/XI PLL input
    01 DLRCLK
    10 ALRCLK
    11 Reserved
7 0 Disable: ADC and DAC idle Internal master clock enable
  1 Enable: ADC and DAC active
*/

#define DIS_ADC_DAC          (0x00)
#define ENA_ADC_DAC          (0x80)

#define PLL_IN_MCLK          (0x00)
#define PLL_IN_DLRCLK        (0x20)
#define PLL_IN_ALRCLK        (0x40)

#define MCLK_OUT_XTAL        (0x00)
#define MCLK_OUT_256FS       (0x08)
#define MCLK_OUT_512FS       (0x10)
#define MCLK_OUT_OFF         (0x18)

#define INPUT256             (0x00)
#define INPUT384             (0x02)
#define INPUT512             (0x04)
#define INPUT768             (0x06)
```

```
#define PLL_PWR_UP              (0x00)
#define PLL_PWR_DWN             (0x01)

/*
    ------------------------------------------------------------------------------
    */
/*PLL and Clock Control 1
                                                                        */
/*
    ------------------------------------------------------------------------------
    */
#define AD1938_PLL_CLK_CTRL1        0x01
/*
Bit Value Function Description
0 0 PLL clock DAC clock source select
1 MCLK
1 0 PLL clock ADC clock source select
1 MCLK
2 0 Enabled On-chip voltage reference
1 Disabled
3 0 Not locked PLL lock indicator (read-only)
  1 Locked
7:4 0000 Reserved
*/
#define AD1938_PLL_LOCK         (0x08)

#define DIS_VREF                (0x04)
#define ENA_VREF                (0x00)

#define ADC_CLK_PLL             (0x00)
#define ADC_CLK_MCLK            (0x20)

#define DAC_CLK_PLL             (0x00)
#define DAC_CLK_MCLK            (0x01)

/*
    ------------------------------------------------------------------------------
    */
/*        DAC Control 0
                                                                    */
/*
    ------------------------------------------------------------------------------
    */
#define AD1938_DAC_CTRL0            0x02
/*
Bit Value Function Description
0 0 Normal Power-down
1 Power-down
2:1 00 32 kHz/44.1 kHz/48 kHz Sample rate
    01 64 kHz/88.2 kHz/96 kHz
    10 128 kHz/176.4 kHz/192 kHz
    11 Reserved
```

```
5:3 000 1 SDATA delay (BCLK periods)
    001 0
    010 8
    011 12
    100 16
    101 Reserved
    110 Reserved
    111 Reserved
7:6 00 Stereo (normal) Serial format
    01 TDM (daisy chain)
    10 DAC AUX mode (ADC-, DAC-, TDM-coupled)
    11 Dual-line TDM
*/
#define DAC_FMT_I2S          (0x00)
#define DAC_FMT_TDM          (0x40)
#define DAC_FMT_AUX          (0x80)
#define DAC_FMT_DUALTDM      (0xc0)


#define DAC_BCLK_DLY_1       (0x00)
#define DAC_BCLK_DLY_0       (0x08)
#define DAC_BCLK_DLY_8       (0x01)
#define DAC_BCLK_DLY_12      (0x18)
#define DAC_BCLK_DLY_16      (0x20)

#define DAC_SR_48K           (0x00)
#define DAC_SR_96K           (0x02)
#define DAC_SR_192K          (0x04)

#define DAC_PWR_UP           (0x00)
#define DAC_PWR_DWN          (0x01)
/*
    ------------------------------------------------------------------------
    */
/*        DAC Control 1
                                                                        */
/*
    ------------------------------------------------------------------------
    */
#define AD1938_DAC_CTRL1          0x03

/*
Bit Value Function Description
0   0 Latch in mid cycle (normal) BCLK active edge (TDM in)
1 Latch in at end of cycle (pipeline)
2:1 00 64 (2 channels) BCLKs per frame
    01 128 (4 channels)
    10 256 (8 channels)
    11 512 (16 channels)
3 0 Left low LRCLK polarity
  1 Left high
4 0 Slave LRCLK master/slave
```

```
  1 Master
5 0 Slave BCLK master/slave
  1 Master
6 0 DBCLK pin BCLK source
  1 Internally generated
7 0 Normal BCLK polarity
  1 Inverted
*/

#define DAC_BCLK_POL_NORM       (0x00)
#define DAC_BCLK_POL_INV        (0x80)

#define DAC_BCLK_SRC_PIN        (0x00)
#define DAC_BCLK_SRC_INTERNAL   (0x40)

#define DAC_BCLK_SLAVE          (0x00)
#define DAC_BCLK_MASTER         (0x20)

#define DAC_LRCLK_SLAVE         (0x00)
#define DAC_LRCLK_MASTER        (0x10)

#define DAC_LRCLK_POL_NORM      (0x00)
#define DAC_LRCLK_POL_INV       (0x08)

#define DAC_CHANNELS_2          (0x00)
#define DAC_CHANNELS_4          (0x02)
#define DAC_CHANNELS_8          (0x04)
#define DAC_CHANNELS_16         (0x06)

#define DAC_LATCH_MID           (0x00)
#define DAC_LATCH_END           (0x01)
/*
    ------------------------------------------------------------------------
    */
/*      DAC Control 2
                                                                        */
/*
    ------------------------------------------------------------------------
    */
/*
Bit Value Function Description
0   0 Unmute Master mute
1 Mute
2:1 00 Flat De-emphasis (32 kHz/44.1 kHz/48 kHz mode only)
    01 48 kHz curve
    10 44.1 kHz curve
    11 32 kHz curve
4:3 00 24 Word width
    01 20
    10 Reserved
    11 16
5 0 Noninverted DAC output polarity
```

```
  1 Inverted
7:6 00 Reserved
*/
#define AD1938_DAC_CTRL2          0x04
#define DAC_OUT_POL_NORM      (0x00)
#define DAC_OUT_POL_INV       (0x20)

#define DAC_WIDTH_24          (0x00)
#define DAC_WIDTH_20          (0x08)
#define DAC_WIDTH_16          (0x18)

#define DAC_DEEMPH_FLAT       (0x00)
#define DAC_DEEMPH_48K        (0x02)
#define DAC_DEEMPH_44_1K      (0x04)
#define DAC_DEEMPH_32K        (0x06)

#define DAC_UNMUTE_ALL        (0x00)
#define DAC_MUTE_ALL          (0x01)
/*
    ------------------------------------------------------------------------
    */
/* DAC individual channel mutes
                                                                        */
/*
    ------------------------------------------------------------------------
    */
#define AD1938_DAC_CHNL_MUTE      0x05
/*
Bit Value Function Description
0 0 Unmute DAC 1 left mute
  1 Mute
1 0 Unmute DAC 1 right mute
  1 Mute
2 0 Unmute DAC 2 left mute
  1 Mute
3 0 Unmute DAC 2 right mute
  1 Mute
4 0 Unmute DAC 3 left mute
  1 Mute
5 0 Unmute DAC 3 right mute
  1 Mute
6 0 Unmute DAC 4 left mute
  1 Mute
7 0 Unmute DAC 4 right mute
  1 Mute
*/
#define DACMUTE_R4      (0x80)
#define DACMUTE_L4      (0x40)
#define DACMUTE_R3      (0x20)
#define DACMUTE_L3      (0x10)
#define DACMUTE_R2      (0x08)
#define DACMUTE_L2      (0x04)
```

```
#define DACMUTE_R1      (0x02)
#define DACMUTE_L1      (0x01)

/*
Bit Value Function Description
7:0         0         No attenuation DAC volume control
1 to 254  -3/8 dB per step
255         Full attenuation

*/
#define DACVOL_MIN      (0xFF)
#define DACVOL_LOW      (0xC0)
#define DACVOL_MED      (0x80)
#define DACVOL_HI       (0x40)
#define DACVOL_MAX      (0x00) // 0db Volume
#define DACVOL_MASK     (0xFF)
/*-------------------------------------------------------------*/
/*DAC L1 volume control       */
/*-------------------------------------------------------------*/
#define AD1938_DAC_L1_VOL       0x06
/*-------------------------------------------------------------*/
/*DAC R1 volume control          */
/*-------------------------------------------------------------*/
#define AD1938_DAC_R1_VOL       0x07
/*-------------------------------------------------------------*/
/*DAC L2 volume control           */
/*-------------------------------------------------------------*/
#define AD1938_DAC_L2_VOL       0x08
/*-------------------------------------------------------------*/
/*DAC R2 volume control
                                                              */
/*-------------------------------------------------------------*/
#define AD1938_DAC_R2_VOL       0x09
/*-------------------------------------------------------------*/
/*DAC L3 volume control
                                                              */
/*-------------------------------------------------------------*/
#define AD1938_DAC_L3_VOL       0x0a

/*-------------------------------------------------------------*/
/*DAC R3 volume control
                                                              */
/*-------------------------------------------------------------*/
#define AD1938_DAC_R3_VOL       0x0b

/*-------------------------------------------------------------*/
/*DAC L4 volume control
                                                              */
/*-------------------------------------------------------------*/
#define AD1938_DAC_L4_VOL       0x0c
/*-------------------------------------------------------------*/
```

```
/*DAC R4 volume control
                                                                    */
/*-------------------------------------------------------------*/
#define AD1938_DAC_R4_VOL       0x0d

/*-------------------------------------------------------------*/
/*        ADC Control 0          */
/*-------------------------------------------------------------*/
#define AD1938_ADC_CTRL0        0x0e
/*
Bit Value Function Description
0 0 Normal Power-down
  1 Power down
1 0 Off High-pass filter
  1 On
2 0 Unmute ADC L1 mute
  1 Mute
3 0 Unmute ADC R1 mute
  1 Mute
4 0 Unmute ADC L2 mute
  1 Mute
5 0 Unmute ADC R2 mute
  1 Mute
7:6 00 32 kHz/44.1 kHz/48 kHz Output sample rate
    01 64 kHz/88.2 kHz/96 kHz
    10 128 kHz/176.4 kHz/192 kHz
    11 Reserved
*/
#define ADC_SR_48K              (0x00)
#define ADC_SR_96K              (0x40)
#define ADC_SR_192K             (0x80)

#define ADC_R2_UNMUTE           (0x00)
#define ADC_R2_MUTE             (0x20)

#define ADC_L2_UNMUTE           (0x00)
#define ADC_L2_MUTE             (0x10)

#define ADC_R1_UNMUTE           (0x00)
#define ADC_R1_MUTE             (0x08)

#define ADC_L1_UNMUTE           (0x00)
#define ADC_L1_MUTE             (0x04)

#define ADC_HP_FILT_OFF         (0x00)
#define ADC_HP_FILT_ON          (0x02)

#define ADC_PWR_UP              (0x00)
#define ADC_PWR_DWN             (0x01)
/*-------------------------------------------------------------*/
/*        ADC Control 01               */
/*-------------------------------------------------------------*/
```

```
#define AD1938_ADC_CTRL1          0x0f
/*
Bit Value Function Description
1:0 00 24 Word width
    01 20
    10 Reserved
    11 16
4:2 000 1 SDATA delay (BCLK periods)
    001 0
    010 8
    011 12
    100 16
    101 Reserved
    110 Reserved
    111 Reserved
6:5 00 Stereo Serial format
    01 TDM (daisy chain)
    10 ADC AUX mode (ADC-, DAC-, TDM-coupled)
    11 Reserved
7 0 Latch in mid cycle (normal) BCLK active edge (TDM in)
  1 Latch in at end of cycle (pipeline)
*/
#define ADC_LATCH_MID         (0x00)
#define ADC_LATCH_END         (0x80)

#define ADC_FMT_I2S           (0x00)
#define ADC_FMT_TDM           (0x20)
#define ADC_FMT_AUX           (0x40)

#define ADC_BCLK_DLY_1        (0x00)
#define ADC_BCLK_DLY_0        (0x04)
#define ADC_BCLK_DLY_8        (0x08)
#define ADC_BCLK_DLY_12       (0x0c)
#define ADC_BCLK_DLY_16       (0x10)

#define ADC_WIDTH_24          (0x00)
#define ADC_WIDTH_20          (0x01)
#define ADC_WIDTH_16          (0x03)
/*-------------------------------------------------------*/
/*         ADC Control 2                                 */
/*-------------------------------------------------------*/
#define AD1938_ADC_CTRL2          0x10
/*
Bit Value Function Description
0 0 50/50 (allows 32, 24, 20, or 16 bit clocks (BCLKs) per channel) LRCLK
    format
  1 Pulse (32 BCLKs per channel)
1 0 Drive out on falling edge (DEF) BCLK polarity
  1 Drive out on rising edge
2 0 Left low LRCLK polarity
  1 Left high
3 0 Slave LRCLK master/slave
```

```
  1 Master
5:4 00 64 BCLKs per frame
    01 128
    10 256
    11 512
6 0 Slave BCLK master/slave
  1 Master
7 0 ABCLK pin BCLK source
  1 Internally generated
*/

#define ADC_BCLK_SRC_PIN        (0x00)
#define ADC_BCLK_SRC_INTERNAL   (0x80)

#define ADC_BCLK_SLAVE          (0x00)
#define ADC_BCLK_MASTER         (0x40)

#define ADC_CHANNELS_2          (0x00)
#define ADC_CHANNELS_4          (0x10)
#define ADC_CHANNELS_8          (0x20)
#define ADC_CHANNELS_16         (0x30)

#define ADC_LRCLK_SLAVE         (0x00)
#define ADC_LRCLK_MASTER        (0x08)

#define ADC_LRCLK_POL_NORM      (0x00)
#define ADC_LRCLK_POL_INV       (0x04)

#define ADC_BCLK_POL_NORM       (0x00)
#define ADC_BCLK_POL_INV        (0x02)

#define ADC_LRCLK_FMT_50_50     (0x00)
#define ADC_LRCLK_FMT_PULSE     (0x01)

/*-------------------------------------------------------*/
/*        init(void)                                     */
/*-------------------------------------------------------*/
bool AudioControlAD1938::spiInit(int clatch,int reset,int cout,int cin,int
    cclk)
{
        ad1938_clatch = clatch;
        ad1938_reset  = reset;
        ad1938_cout   = cout;
        ad1938_cin    = cin;
        ad1938_cclk   = cclk;

        /**/
        pinMode(ad1938_clatch, OUTPUT);
        pinMode(ad1938_reset, OUTPUT);

        /*SPI clock pin set*/
       // SPI.setMOSI(cin);
```

```
        //SPI.setMISO(cout);
        SPI.setSCK(cclk);/*SPI clock alternate pin 14*/
        SPI.begin();
         /*reset codec*/
        digitalWrite(ad1938_reset, LOW);
        delay(200);
        digitalWrite(ad1938_reset, HIGH);
        delay(400);//wait for 300ms to load the code

        return true;

}
/*-------------------------------------------------------*/
/*        spi_read_reg                                   */
/*-------------------------------------------------------*/
unsigned char AudioControlAD1938::spi_read_reg(unsigned char reg)
{
        unsigned char result = 0;
        unsigned char data[AD1938_SPI_WRITE_BYTE_COUNT];

        data[0] = AD1938_READ_ADDRESS;
        data[1] = reg;
        data[2] = 0x0;

        // and configure settings
        SPI.beginTransaction(SPISettings(AD1938_SPI_CLK_FREQ, MSBFIRST,
            SPI_MODE3));
        // take the chip select low to select the device:
        digitalWrite(ad1938_clatch, LOW);

        SPI.transfer(&data[0], AD1938_SPI_READ_BYTE_COUNT); //Send register
            location , read byte is 2

    // send a value of 0 to read the first byte returned:
        result =(unsigned char) SPI.transfer(0x00);

        // take the chip select high to de-select:
        digitalWrite(ad1938_clatch, HIGH);
        // release control of the SPI port
        SPI.endTransaction();


        return(result);

}

/*-------------------------------------------------------*/
/*        spi_write_reg
                                                         */
/*-------------------------------------------------------*/
bool AudioControlAD1938::spi_write_reg(unsigned char reg, unsigned char val)
{
```

```
        unsigned char data[AD1938_SPI_WRITE_BYTE_COUNT];

        /*fill the buffer as per AD1938 format*/
        data[0] = AD1938_WRITE_ADDRESS;
        data[1] = reg;
        data[2] = val;

        // and configure settings
        SPI.beginTransaction(SPISettings(AD1938_SPI_CLK_FREQ, MSBFIRST,
            SPI_MODE3));
    // take the chip select low to select the device:
        digitalWrite(ad1938_clatch, LOW);

        SPI.transfer(&data[0], AD1938_SPI_WRITE_BYTE_COUNT);
        // take the chip select high to de-select:
        digitalWrite(ad1938_clatch, HIGH);
        // release control of the SPI port
        SPI.endTransaction();

        return true;

}

/*-------------------------------------------------------*/
/*        init(void)                                     */
                                                        */
/*-------------------------------------------------------*/
bool AudioControlAD1938::config(Te_samplingRate sampleRate,
                                Te_bitsPerSample wordLen,
                                                Te_i2sNumChannels

                                                    numChannels
                                                    ,
                                                Te_i2sClockMode
                                                    mode)
{

    unsigned char dac_fs = 0;
        unsigned char adc_fs = 0;
        unsigned char dac_mode = 0;
        unsigned char adc_mode = 0;
        unsigned char dac_wl = 0;
        unsigned char adc_wl = 0;
        unsigned char dac_channels = 0;
        unsigned char adc_channels = 0;


        i2sMode      = mode;
        wordLen      = wordLen;
        numChannels = numChannels;
        samplingRate = sampleRate;
```

```
    switch(sampleRate)
    {
     case FS_32000:
     case FS_44100:
     case FS_48000:
     {
            dac_fs = DAC_SR_48K;
            adc_fs = ADC_SR_48K;
     }
            break;
case FS_64000:
case FS_88200:
case FS_96000:
     {
            dac_fs = DAC_SR_96K;
            adc_fs = ADC_SR_96K;
     }
            break;
case FS_128000:
case FS_176400:
case FS_192000:
     {
            dac_fs = DAC_SR_192K;
            adc_fs = ADC_SR_192K;
     }
            break;
     default:
     {
            dac_fs = DAC_SR_48K;
            adc_fs = ADC_SR_48K;
     }
    }

    switch(wordLen)
    {
     case BITS_16:
     {
            dac_wl = DAC_WIDTH_16;
            adc_wl = ADC_WIDTH_16;


     }
    break;
case BITS_20:
     {
            dac_wl = DAC_WIDTH_20;
            adc_wl = ADC_WIDTH_20;

     }
     break;
case BITS_24:
     {
```

```
                dac_wl = DAC_WIDTH_24;
                adc_wl = ADC_WIDTH_24;

            }

        break;

            default:
            {
                dac_wl = DAC_WIDTH_24;
                adc_wl = ADC_WIDTH_24;

            }
        }

        switch(numChannels)
        {
            case I2S_STEREO_2CH:
            {
                dac_mode = DAC_FMT_I2S;
                adc_mode = ADC_FMT_I2S;
                dac_channels = DAC_CHANNELS_2;
            adc_channels = ADC_CHANNELS_2;

            }
                break;
    case I2S_TDM_8CH:
            {
                dac_mode = DAC_FMT_TDM;
                adc_mode = ADC_FMT_TDM;
                dac_channels = DAC_CHANNELS_8;
            adc_channels = ADC_CHANNELS_8;

            }
                break;
    case I2S_TDM_16CH:
            {
                dac_mode = DAC_FMT_TDM;
                adc_mode = ADC_FMT_TDM;
                dac_channels = DAC_CHANNELS_16;
            adc_channels = ADC_CHANNELS_16;

            }
        break;
            default:
            {
                dac_mode = DAC_FMT_I2S;
                adc_mode = ADC_FMT_I2S;
                dac_channels = DAC_CHANNELS_2;
            adc_channels = ADC_CHANNELS_2;
            }
        }
```

```
    if(mode == AD1938_I2S_SLAVE)
    {

    //0 PLL and Clock Control 0
    spi_write_reg(AD1938_PLL_CLK_CTRL0, (DIS_ADC_DAC | INPUT512 |
        PLL_IN_ALRCLK | MCLK_OUT_OFF |PLL_PWR_DWN));


//1 PLL and Clock Control 1
    spi_write_reg(AD1938_PLL_CLK_CTRL1, (DAC_CLK_PLL | ADC_CLK_PLL |
        ENA_VREF));

    //2 DAC Control 0
    spi_write_reg(AD1938_DAC_CTRL0, (dac_mode | DAC_BCLK_DLY_1 | dac_fs |
         DAC_PWR_UP));

//3 DAC Control 1
    spi_write_reg(AD1938_DAC_CTRL1, ( DAC_BCLK_SRC_PIN|DAC_BCLK_SLAVE|
        DAC_LRCLK_SLAVE  |DAC_LRCLK_POL_NORM | dac_channels |
        DAC_LATCH_MID));
    //4 DAC Control 2
    spi_write_reg(AD1938_DAC_CTRL2, dac_wl);

    //5 DAC individual channel mutes
    spi_write_reg(AD1938_DAC_CHNL_MUTE, 0x00);/*mute*/

    //6 DAC L1 volume control
    spi_write_reg(AD1938_DAC_L1_VOL, DACVOL_MAX);

    //7 DAC R1 volume control
    spi_write_reg(AD1938_DAC_R1_VOL, DACVOL_MAX);

    //8 DAC L2 volume control
    spi_write_reg(AD1938_DAC_L2_VOL, DACVOL_MAX);

    //9 DAC R2 volume control
    spi_write_reg(AD1938_DAC_R2_VOL, DACVOL_MAX);

    //10 DAC L3 volume control
    spi_write_reg(AD1938_DAC_L3_VOL, DACVOL_MAX);

    //11 DAC R3 volume control
    spi_write_reg(AD1938_DAC_R3_VOL, DACVOL_MAX);

    //12 DAC L4 volume control
    spi_write_reg(AD1938_DAC_L4_VOL, DACVOL_MAX);

    //13 DAC R4 volume control
    spi_write_reg(AD1938_DAC_R4_VOL, DACVOL_MAX);
```

```
        //14 ADC Control 0
        spi_write_reg(AD1938_ADC_CTRL0, adc_fs);

        //15 ADC Control 1
        spi_write_reg(AD1938_ADC_CTRL1, (ADC_LATCH_MID | adc_mode |
            ADC_BCLK_DLY_0 | adc_wl));

        //16 ADC Control 2
        spi_write_reg(AD1938_ADC_CTRL2, ( ADC_BCLK_SRC_PIN|ADC_BCLK_SLAVE |
            adc_channels | ADC_LRCLK_SLAVE  | ADC_LRCLK_FMT_50_50|
            ADC_LRCLK_POL_NORM|ADC_BCLK_POL_NORM));

        }
        else
        {
                //0 PLL and Clock Control 0
                spi_write_reg(AD1938_PLL_CLK_CTRL0, (DIS_ADC_DAC | INPUT512 |
                     PLL_IN_MCLK | MCLK_OUT_XTAL |PLL_PWR_DWN));

                //1 PLL and Clock Control 1
                spi_write_reg(AD1938_PLL_CLK_CTRL1, (DAC_CLK_MCLK |
                    ADC_CLK_MCLK | ENA_VREF));

                //2 DAC Control 0
                spi_write_reg(AD1938_DAC_CTRL0, (dac_mode | DAC_BCLK_DLY_1 |
                    dac_fs | DAC_PWR_UP));

                 //3 DAC Control 1
                spi_write_reg(AD1938_DAC_CTRL1, ( DAC_BCLK_SRC_INTERNAL|
                    DAC_BCLK_SLAVE| DAC_LRCLK_SLAVE  |DAC_LRCLK_POL_NORM |
                    dac_channels | DAC_LATCH_MID));

                //4 DAC Control 2
                spi_write_reg(AD1938_DAC_CTRL2, dac_wl);

                //5 DAC individual channel mutes
                spi_write_reg(AD1938_DAC_CHNL_MUTE, 0x00);/*unmute*/

                //6 DAC L1 volume control
                spi_write_reg(AD1938_DAC_L1_VOL, DACVOL_MAX);

                //7 DAC R1 volume control
                spi_write_reg(AD1938_DAC_R1_VOL, DACVOL_MAX);

                //8 DAC L2 volume control
                spi_write_reg(AD1938_DAC_L2_VOL, DACVOL_MAX);

                //9 DAC R2 volume control
                spi_write_reg(AD1938_DAC_R2_VOL, DACVOL_MAX);

                //10 DAC L3 volume control
                spi_write_reg(AD1938_DAC_L3_VOL, DACVOL_MAX);
```

```
                //11 DAC R3 volume control
                spi_write_reg(AD1938_DAC_R3_VOL, DACVOL_MAX);

                //12 DAC L4 volume control
                spi_write_reg(AD1938_DAC_L4_VOL, DACVOL_MAX);

                //13 DAC R4 volume control
                spi_write_reg(AD1938_DAC_R4_VOL, DACVOL_MAX);

                //14 ADC Control 0
                spi_write_reg(AD1938_ADC_CTRL0, adc_fs);

                //15 ADC Control 1
                spi_write_reg(AD1938_ADC_CTRL1, (ADC_LATCH_MID | adc_mode |
                    ADC_BCLK_DLY_1 | adc_wl));

                //16 ADC Control 2
                spi_write_reg(AD1938_ADC_CTRL2, ( ADC_BCLK_SRC_INTERNAL|
                    ADC_BCLK_MASTER | adc_channels | ADC_LRCLK_MASTER  |
                    ADC_LRCLK_FMT_50_50|ADC_LRCLK_POL_NORM|ADC_BCLK_POL_NORM)
                    );
        }
        return true;

}

/*-------------------------------------------------------*/
/*        isPllLocked(void)
                                                         */
/*-------------------------------------------------------*/
bool AudioControlAD1938::isPllLocked(void)
{
        return ((spi_read_reg(AD1938_PLL_CLK_CTRL1)>>3)&0x1);

}
/*-------------------------------------------------------*/
/*        enable(void)
                                                         */
/*-------------------------------------------------------*/
bool AudioControlAD1938::enable(void)
{

        if(i2sMode == AD1938_I2S_SLAVE)
        {
                spi_write_reg(AD1938_PLL_CLK_CTRL0, (ENA_ADC_DAC | INPUT512 |
                    PLL_IN_DLRCLK | MCLK_OUT_OFF | PLL_PWR_UP));
        }
        else
        {
                spi_write_reg(AD1938_PLL_CLK_CTRL0, (ENA_ADC_DAC | INPUT512 |
                    PLL_IN_MCLK | MCLK_OUT_XTAL |PLL_PWR_UP));
```

```
        }

        spi_write_reg(AD1938_DAC_CHNL_MUTE, 0);/*un mute*/

    return true;
}

/*-------------------------------------------------------*/
/*        disable(void)                                  */
/*-------------------------------------------------------*/
bool AudioControlAD1938::disable(void)
{
        unsigned char reg_value;
        reg_value = spi_read_reg(AD1938_PLL_CLK_CTRL0);
        reg_value = (reg_value&0x7e);/*mask the last and first bits*/
        spi_write_reg(AD1938_PLL_CLK_CTRL0, (DIS_ADC_DAC | reg_value|
            PLL_PWR_DWN));

        reg_value = spi_read_reg(AD1938_DAC_CTRL0);

        spi_write_reg(AD1938_DAC_CTRL0, ((reg_value&0xfe)|DAC_PWR_DWN));

        reg_value = spi_read_reg(AD1938_ADC_CTRL0);

        spi_write_reg(AD1938_DAC_CTRL0, ((reg_value&0xfe)|ADC_PWR_DWN));

   return true;
}
/*-------------------------------------------------------*/
/*        dacVolume(int dac_num, float volume)           */
/*-------------------------------------------------------*/
bool AudioControlAD1938::dacVolume(int dac_num, int volume)
{

                switch (dac_num)
                {
                case 0://DAC0
                        spi_write_reg(AD1938_DAC_L1_VOL, volume);
                        spi_write_reg(AD1938_DAC_R1_VOL, volume);
                        break;
                case 1://DAC1
                        spi_write_reg(AD1938_DAC_L2_VOL, volume);
                        spi_write_reg(AD1938_DAC_R2_VOL, volume);
                        break;
                case 2://DAC2
                        spi_write_reg(AD1938_DAC_L3_VOL, volume);
                        spi_write_reg(AD1938_DAC_R3_VOL, volume);
                        break;
                case 3://DAC3
                        spi_write_reg(AD1938_DAC_L4_VOL, volume);
```

```
                        spi_write_reg(AD1938_DAC_R4_VOL, volume);
                        break;

                }
        return true;

}
/*-------------------------------------------------------------*/
/*        volume( float volume)  */                            /*
   -------------------------------------------------------------*/
bool AudioControlAD1938::volume(float volume)
{
        int vol = 0;

        vol =(int)((1.0-volume)*255);
        if (vol<0)
                vol=0;
        if(vol>255)
                vol =255;

        dacVolume(0,vol);
        dacVolume(1,vol);
        dacVolume(2,vol);
        dacVolume(3,vol);

        return true;

}

/*-------------------------------------------------------------*/
/*        dacMute                                              */
/*-------------------------------------------------------------*/
bool AudioControlAD1938::dacMute(bool mute)
{
        if (mute == true)
        {
                spi_write_reg(AD1938_DAC_CHNL_MUTE, 0xff);/*mute*/
        }
        else
        {
                spi_write_reg(AD1938_DAC_CHNL_MUTE, 00);/*unmute*/
        }
        return true;

}

/*-------------------------------------------------------------*/
/*        adcMute                                              */
/*-------------------------------------------------------------*/
bool AudioControlAD1938::adcMute(bool mute)
```

```
{
        unsigned char reg_value;
        reg_value = spi_read_reg(AD1938_ADC_CTRL0);
        if (mute == true)
        {
                spi_write_reg(AD1938_ADC_CTRL0, (reg_value&0xc3)|0x3c);/*mute
                    */
        }
        else
        {
                spi_write_reg(AD1938_ADC_CTRL0, (reg_value&0xc3));/*unmute*/
        }
        return true;
}

/*-------------------------------------------------------*/
/*        readAllreg    */
/*-------------------------------------------------------*/
void AudioControlAD1938::readAllreg(void)
{
        int i =0;
        unsigned char reg_val=0;
         Serial.print("\n readAllreg\n");
        for (i = 0;i<17;i++)
        {
           reg_val = spi_read_reg(i);
           Serial.print("\n");
           Serial.print(i);
           Serial.print("\t");
           Serial.print(reg_val, HEX);
        }

}
/*-----------------------------------*/
/*        end of file     */
/*-------------------------------------------------------*/
```

input_tdm.h

```
/* Audio Library for Teensy 3.X
 * Copyright (c) 2017, Paul Stoffregen, paul@pjrc.com
 *
 * Development of this audio library was funded by PJRC.COM, LLC by sales of
 * Teensy and Audio Adaptor boards.  Please support PJRC's efforts to develop
 * open source software by purchasing Teensy or other PJRC products.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy
 * of this software and associated documentation files (the "Software"), to
    deal
 * in the Software without restriction, including without limitation the
    rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
```

```
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice, development funding notice, and this
    permission
 * notice shall be included in all copies or substantial portions of the
    Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#ifndef _input_tdm_h_
#define _input_tdm_h_

#include "Arduino.h"
#include "AudioStream.h"
#include "DMAChannel.h"
//#define MAX_CHANNELS  16
#define MAX_CHANNELS  32

class AudioInputTDM : public AudioStream
{
public:
        AudioInputTDM(void) : AudioStream(0, NULL) { begin(); }
        virtual void update(void);
        void begin(void);
protected:
     AudioInputTDM(int dummy): AudioStream(0, NULL) {} // to be used only
         inside AudioInputI2Sslave !!
        static bool update_responsibility;
        static DMAChannel dma;
        static void isr(void);
private:
        static audio_block_t *block_incoming[MAX_CHANNELS];
};
class AudioInputTDMslave : public AudioInputTDM
{
public:
        AudioInputTDMslave(void) : AudioInputTDM(0) { begin(); }
        void begin(void);
        friend void dma_ch1_isr(void);
};
#endif
```

input_tdm.cpp

```
/* Audio Library for Teensy 3.X
 * Copyright (c) 2017, Paul Stoffregen, paul@pjrc.com
 *
 * Development of this audio library was funded by PJRC.COM, LLC by sales of
 * Teensy and Audio Adaptor boards.  Please support PJRC's efforts to develop
 * open source software by purchasing Teensy or other PJRC products.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy
 * of this software and associated documentation files (the "Software"), to
    deal
 * in the Software without restriction, including without limitation the
    rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice, development funding notice, and this
    permission
 * notice shall be included in all copies or substantial portions of the
    Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#include "input_tdm.h"
#include "output_tdm.h"
#if defined(KINETISK)

DMAMEM static uint32_t tdm_rx_buffer[AUDIO_BLOCK_SAMPLES*MAX_CHANNELS];
audio_block_t * AudioInputTDM::block_incoming[MAX_CHANNELS] = {
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
};
bool AudioInputTDM::update_responsibility = false;
DMAChannel AudioInputTDM::dma(false);


void AudioInputTDM::begin(void)
{
        Serial.print("\n AudioInputTDM:begin ");
        dma.begin(true); // Allocate the DMA channel first
```

```
        // TODO: should we set & clear the I2S_RCSR_SR bit here?
        AudioOutputTDM::config_tdm();

        CORE_PIN13_CONFIG = PORT_PCR_MUX(4); // pin 13, PTC5, I2S0_RXD0
        dma.TCD->SADDR = &I2S0_RDR0;
        dma.TCD->SOFF = 0;
        dma.TCD->ATTR = DMA_TCD_ATTR_SSIZE(2) | DMA_TCD_ATTR_DSIZE(2);
        dma.TCD->NBYTES_MLNO = 4;
        dma.TCD->SLAST = 0;
        dma.TCD->DADDR = tdm_rx_buffer;
        dma.TCD->DOFF = 4;
        dma.TCD->CITER_ELINKNO = sizeof(tdm_rx_buffer) / 4;
        dma.TCD->DLASTSGA = -sizeof(tdm_rx_buffer);
        dma.TCD->BITER_ELINKNO = sizeof(tdm_rx_buffer) / 4;
        dma.TCD->CSR = DMA_TCD_CSR_INTHALF | DMA_TCD_CSR_INTMAJOR;
        dma.triggerAtHardwareEvent(DMAMUX_SOURCE_I2S0_RX);
        update_responsibility = update_setup();
        dma.enable();

        I2S0_RCSR |= I2S_RCSR_RE | I2S_RCSR_BCE | I2S_RCSR_FRDE | I2S_RCSR_FR
            ;
        I2S0_TCSR |= I2S_TCSR_TE | I2S_TCSR_BCE; // TX clock enable, because
            sync'd to TX
        dma.attachInterrupt(isr);
}

// TODO: needs optimization...
static void memcpy_tdm_rx(uint32_t *dest1, uint32_t *dest2, const uint32_t *
    src)
{
        uint32_t i, in1, in2;

        for (i=0; i < AUDIO_BLOCK_SAMPLES/2; i++) {
                in1 = *src;
                in2 = *(src+(MAX_CHANNELS>>1));
                src += MAX_CHANNELS;
                *dest1++ = (in1 >> 16) | (in2 & 0xFFFF0000);
                *dest2++ = (in1 << 16) | (in2 & 0x0000FFFF);
        }
        //Serial.print(*src,HEX);
}

void AudioInputTDM::isr(void)
{
        uint32_t daddr;
        const uint32_t *src;
        unsigned int i;
   //Serial.print("\n AudioInputTDMslave:isr ");
        //digitalWriteFast(35, HIGH);
        daddr = (uint32_t)(dma.TCD->DADDR);
        dma.clearInterrupt();
```

```
        if (daddr < (uint32_t)tdm_rx_buffer + sizeof(tdm_rx_buffer) / 2) {
                // DMA is receiving to the first half of the buffer
                // need to remove data from the second half
                src = &tdm_rx_buffer[AUDIO_BLOCK_SAMPLES*(MAX_CHANNELS>>1)];
        } else {
                // DMA is receiving to the second half of the buffer
                // need to remove data from the first half
                src = &tdm_rx_buffer[0];
        }
        if (block_incoming[0] != NULL) {
                for (i=0; i < MAX_CHANNELS; i += 2) {
                        uint32_t *dest1 = (uint32_t *)(block_incoming[i]->
                            data);
                        uint32_t *dest2 = (uint32_t *)(block_incoming[i+1]->
                            data);
                        memcpy_tdm_rx(dest1, dest2, src);
                        src++;
                }
        }
        if (update_responsibility) update_all();
        //digitalWriteFast(35, LOW);
}

void AudioInputTDM::update(void)
{
        unsigned int i, j;
        audio_block_t *new_block[MAX_CHANNELS];
        audio_block_t *out_block[MAX_CHANNELS];

        // allocate 16 new blocks.  If any fails, allocate none
        for (i=0; i < MAX_CHANNELS; i++) {
                new_block[i] = allocate();
                if (new_block[i] == NULL) {
                        for (j=0; j < i; j++) {
                                release(new_block[j]);
                        }
                        memset(new_block, 0, sizeof(new_block));
                        break;
                }
        }
        __disable_irq();
        memcpy(out_block, block_incoming, sizeof(out_block));
        memcpy(block_incoming, new_block, sizeof(block_incoming));
        __enable_irq();
        if (out_block[0] != NULL) {
                // if we got 1 block, all 16 are filled
                for (i=0; i < MAX_CHANNELS; i++) {
                        transmit(out_block[i], i);
                        release(out_block[i]);
                }
        }
```

```
}

void AudioInputTDMslave::begin(void)
{
        //Serial.print("\n AudioInputTDMslave:begin ");
        dma.begin(true); // Allocate the DMA channel first

        // TODO: should we set & clear the I2S_RCSR_SR bit here?
        AudioOutputTDMslave::config_tdm();

        CORE_PIN13_CONFIG = PORT_PCR_MUX(4); // pin 13, PTC5, I2S0_RXD0
        dma.TCD->SADDR = &I2S0_RDR0;
        dma.TCD->SOFF = 0;
        dma.TCD->ATTR = DMA_TCD_ATTR_SSIZE(2) | DMA_TCD_ATTR_DSIZE(2);
        dma.TCD->NBYTES_MLNO = 4;
        dma.TCD->SLAST = 0;
        dma.TCD->DADDR = tdm_rx_buffer;
        dma.TCD->DOFF = 4;
        dma.TCD->CITER_ELINKNO = sizeof(tdm_rx_buffer) / 4;
        dma.TCD->DLASTSGA = -sizeof(tdm_rx_buffer);
        dma.TCD->BITER_ELINKNO = sizeof(tdm_rx_buffer) / 4;
        dma.TCD->CSR = DMA_TCD_CSR_INTHALF | DMA_TCD_CSR_INTMAJOR;
        dma.triggerAtHardwareEvent(DMAMUX_SOURCE_I2S0_RX);
        update_responsibility = update_setup();
        dma.enable();

        I2S0_RCSR |= I2S_RCSR_RE | I2S_RCSR_BCE | I2S_RCSR_FRDE | I2S_RCSR_FR
            ;
        I2S0_TCSR |= I2S_TCSR_TE | I2S_TCSR_BCE; // TX clock enable, because
            sync'd to TX
        dma.attachInterrupt(isr);
}
#endif // KINETISK
```

output_tdm.h

```
/* Audio Library for Teensy 3.X
 * Copyright (c) 2017, Paul Stoffregen, paul@pjrc.com
 *
 * Development of this audio library was funded by PJRC.COM, LLC by sales of
 * Teensy and Audio Adaptor boards.  Please support PJRC's efforts to develop
 * open source software by purchasing Teensy or other PJRC products.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy
 * of this software and associated documentation files (the "Software"), to
    deal
 * in the Software without restriction, including without limitation the
    rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
```

```
 * The above copyright notice, development funding notice, and this
    permission
 * notice shall be included in all copies or substantial portions of the
    Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#ifndef output_tdm_h_
#define output_tdm_h_

#include "Arduino.h"
#include "AudioStream.h"
#include "DMAChannel.h"

//#define MAX_CHANNELS  16
#define MAX_CHANNELS  32
#define MAX_CHANNELS  32
class AudioOutputTDM : public AudioStream
{
public:
        AudioOutputTDM(void) : AudioStream(MAX_CHANNELS, inputQueueArray) {
            begin(); }
        virtual void update(void);
        void begin(void);
        friend class AudioInputTDM;
protected:
    AudioOutputTDM(int dummy) : AudioStream(MAX_CHANNELS, inputQueueArray) {
         }
        static void config_tdm(void);
        static audio_block_t *block_input[MAX_CHANNELS];
        static bool update_responsibility;
        static DMAChannel dma;
        static void isr(void);
private:
        audio_block_t *inputQueueArray[MAX_CHANNELS];
};

class AudioOutputTDMslave : public AudioOutputTDM
{
public:
        AudioOutputTDMslave(void) : AudioOutputTDM(0) { begin(); } ;
        void begin(void);
        friend class AudioInputTDMslave;
```

```
        friend void dma_ch0_isr(void);
protected:
        static void config_tdm(void);
};
#endif
```

output_tdm.cpp

```
/* Audio Library for Teensy 3.X
 * Copyright (c) 2017, Paul Stoffregen, paul@pjrc.com
 *
 * Development of this audio library was funded by PJRC.COM, LLC by sales of
 * Teensy and Audio Adaptor boards.  Please support PJRC's efforts to develop
 * open source software by purchasing Teensy or other PJRC products.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy
 * of this software and associated documentation files (the "Software"), to
    deal
 * in the Software without restriction, including without limitation the
    rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice, development funding notice, and this
    permission
 * notice shall be included in all copies or substantial portions of the
    Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

#include "output_tdm.h"
#include "memcpy_audio.h"

#if defined(KINETISK)

audio_block_t * AudioOutputTDM::block_input[MAX_CHANNELS] = {
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
};
bool AudioOutputTDM::update_responsibility = false;
static uint32_t zeros[AUDIO_BLOCK_SAMPLES/2];
DMAMEM static uint32_t tdm_tx_buffer[AUDIO_BLOCK_SAMPLES*MAX_CHANNELS];
```

```
DMAChannel AudioOutputTDM::dma(false);

void AudioOutputTDM::begin(void)
{
        Serial.print("\n AudioOutputTDM:begin ");
        dma.begin(true); // Allocate the DMA channel first

        for (int i=0; i < MAX_CHANNELS; i++) {
                block_input[i] = NULL;
        }

        // TODO: should we set & clear the I2S_TCSR_SR bit here?
        config_tdm();
        CORE_PIN22_CONFIG = PORT_PCR_MUX(6); // pin 22, PTC1, I2S0_TXD0

        dma.TCD->SADDR = tdm_tx_buffer;
        dma.TCD->SOFF = 4;
        dma.TCD->ATTR = DMA_TCD_ATTR_SSIZE(2) | DMA_TCD_ATTR_DSIZE(2);
        dma.TCD->NBYTES_MLNO = 4;
        dma.TCD->SLAST = -sizeof(tdm_tx_buffer);
        dma.TCD->DADDR = &I2S0_TDR0;
        dma.TCD->DOFF = 0;
        dma.TCD->CITER_ELINKNO = sizeof(tdm_tx_buffer) / 4;
        dma.TCD->DLASTSGA = 0;
        dma.TCD->BITER_ELINKNO = sizeof(tdm_tx_buffer) / 4;
        dma.TCD->CSR = DMA_TCD_CSR_INTHALF | DMA_TCD_CSR_INTMAJOR;
        dma.triggerAtHardwareEvent(DMAMUX_SOURCE_I2S0_TX);
        update_responsibility = update_setup();
        dma.enable();

        I2S0_TCSR = I2S_TCSR_SR;
        I2S0_TCSR = I2S_TCSR_TE | I2S_TCSR_BCE | I2S_TCSR_FRDE;
        dma.attachInterrupt(isr);
}

// TODO: needs optimization...
static void memcpy_tdm_tx(uint32_t *dest, const uint32_t *src1, const
    uint32_t *src2)
{
        uint32_t i, in1, in2, out1, out2;

        //Serial.print(*src2,HEX);
        for (i=0; i < AUDIO_BLOCK_SAMPLES/2; i++) {
                in1 = *src1++;
                in2 = *src2++;
            //Serial.print("\n");
            //Serial.print(in1,HEX);
                out1 = (in1 << 16) | (in2 & 0xFFFF);
                out2 = (in1 & 0xFFFF0000) | (in2 >> 16);
                //out1=((in1 & 0xFFFF)<<16);
                //out2=((in1 & 0xFFFF0000));
                *dest = out1;
```

```
                //Serial.print("\n");
                //Serial.print(out1,HEX);
                *(dest + (MAX_CHANNELS>>1)) = out2;
                dest += MAX_CHANNELS;
        }

        //Serial.print(*src,HEX);
}

void AudioOutputTDM::isr(void)
{
        uint32_t *dest;
        const uint32_t *src1, *src2;
        uint32_t i, saddr;
    //Serial.print("\n AudioOutputTDM:isr ");
        ///digitalWriteFast(35, HIGH);
        saddr = (uint32_t)(dma.TCD->SADDR);
        dma.clearInterrupt();
        if (saddr < (uint32_t)tdm_tx_buffer + sizeof(tdm_tx_buffer) / 2) {
                // DMA is transmitting the first half of the buffer
                // so we must fill the second half
                dest = tdm_tx_buffer + AUDIO_BLOCK_SAMPLES*(MAX_CHANNELS>>1);
        } else {
                // DMA is transmitting the second half of the buffer
                // so we must fill the first half
                dest = tdm_tx_buffer;
        }
        if (update_responsibility) AudioStream::update_all();
        for (i=0; i < MAX_CHANNELS; i += 2) {
                src1 = block_input[i] ? (uint32_t *)(block_input[i]->data) :
                    zeros;
                src2 = block_input[i+1] ? (uint32_t *)(block_input[i+1]->data
                    ) : zeros;
                memcpy_tdm_tx(dest, src1, src2);
                dest++;
        }
        for (i=0; i < MAX_CHANNELS; i++) {
                if (block_input[i]) {
                        release(block_input[i]);
                        block_input[i] = NULL;
                }
        }
        //digitalWriteFast(35, LOW);
}

void AudioOutputTDM::update(void)
{
        audio_block_t *prev[MAX_CHANNELS];
        unsigned int i;

        __disable_irq();
        for (i=0; i < MAX_CHANNELS; i++) {
```

```
                prev[i] = block_input[i];
                block_input[i] = receiveReadOnly(i);
        }
        __enable_irq();
        for (i=0; i < MAX_CHANNELS; i++) {
                if (prev[i]) release(prev[i]);
        }
}

// MCLK needs to be 48e6 / 1088 * 512 = 22.588235 MHz -> 44.117647 kHz sample
    rate
//
#if F_CPU == 96000000 || F_CPU == 48000000 || F_CPU == 24000000
  // PLL is at 96 MHz in these modes
  #define MCLK_MULT 4
  #define MCLK_DIV  17
#elif F_CPU == 72000000
  #define MCLK_MULT 16
  #define MCLK_DIV  51
#elif F_CPU == 120000000
  #define MCLK_MULT 16
  #define MCLK_DIV  85
#elif F_CPU == 144000000
  #define MCLK_MULT 8
  #define MCLK_DIV  51
#elif F_CPU == 168000000
  #define MCLK_MULT 16
  #define MCLK_DIV  119
#elif F_CPU == 180000000
  #define MCLK_MULT 32
  #define MCLK_DIV  255
  #define MCLK_SRC  0
#elif F_CPU == 192000000
  //#define MCLK_MULT 2
  //#define MCLK_DIV  17
  //++++
  #define MCLK_MULT 16
  #define MCLK_DIV  125
  //----
#elif F_CPU == 216000000
  #define MCLK_MULT 16
  #define MCLK_DIV  153
  #define MCLK_SRC  0
#elif F_CPU == 240000000
  #define MCLK_MULT 8
  #define MCLK_DIV  85
#else
  #error "This CPU Clock Speed is not supported by the Audio library";
#endif

#ifndef MCLK_SRC
#if F_CPU >= 20000000
```

```
  #define MCLK_SRC  3  // the PLL
#else
  #define MCLK_SRC  0  // system clock
#endif
#endif

void AudioOutputTDM::config_tdm(void)
{
        SIM_SCGC6 |= SIM_SCGC6_I2S;
        SIM_SCGC7 |= SIM_SCGC7_DMA;
        SIM_SCGC6 |= SIM_SCGC6_DMAMUX;

        // if either transmitter or receiver is enabled, do nothing
        if (I2S0_TCSR & I2S_TCSR_TE) return;
        if (I2S0_RCSR & I2S_RCSR_RE) return;

        // enable MCLK output
        I2S0_MCR = I2S_MCR_MICS(MCLK_SRC) | I2S_MCR_MOE;
        while (I2S0_MCR & I2S_MCR_DUF) ;
        I2S0_MDR = I2S_MDR_FRACT((MCLK_MULT-1)) | I2S_MDR_DIVIDE((MCLK_DIV-1)
            );

        // configure transmitter
        I2S0_TMR = 0;
        I2S0_TCR1 = I2S_TCR1_TFW(4);
        I2S0_TCR2 = I2S_TCR2_SYNC(0) | I2S_TCR2_BCP | I2S_TCR2_MSEL(1)
                | I2S_TCR2_BCD | I2S_TCR2_DIV(0);
        I2S0_TCR3 = I2S_TCR3_TCE;
        I2S0_TCR4 = I2S_TCR4_FRSZ(7) | I2S_TCR4_SYWD(0) | I2S_TCR4_MF
                | I2S_TCR4_FSE | I2S_TCR4_FSD;
        I2S0_TCR5 = I2S_TCR5_WNW(31) | I2S_TCR5_W0W(31) | I2S_TCR5_FBT(31);

        // configure receiver (sync'd to transmitter clocks)
        I2S0_RMR = 0;
        I2S0_RCR1 = I2S_RCR1_RFW(4);
        I2S0_RCR2 = I2S_RCR2_SYNC(1) | I2S_TCR2_BCP | I2S_RCR2_MSEL(1)
                | I2S_RCR2_BCD | I2S_RCR2_DIV(0);
        I2S0_RCR3 = I2S_RCR3_RCE;
        I2S0_RCR4 = I2S_RCR4_FRSZ(7) | I2S_RCR4_SYWD(0) | I2S_RCR4_MF
                | I2S_RCR4_FSE | I2S_RCR4_FSD;
        I2S0_RCR5 = I2S_RCR5_WNW(31) | I2S_RCR5_W0W(31) | I2S_RCR5_FBT(31);

        // configure pin mux for 3 clock signals
        CORE_PIN23_CONFIG = PORT_PCR_MUX(6); // pin 23, PTC2, I2S0_TX_FS (
            LRCLK)
        CORE_PIN9_CONFIG  = PORT_PCR_MUX(6); // pin  9, PTC3, I2S0_TX_BCLK
        CORE_PIN11_CONFIG = PORT_PCR_MUX(6); // pin 11, PTC6, I2S0_MCLK
}
void AudioOutputTDMslave::config_tdm(void)
{
        Serial.print("\n AudioOutputTDMslave:config_tdm ");
        SIM_SCGC6 |= SIM_SCGC6_I2S;
```

```
        SIM_SCGC7 |= SIM_SCGC7_DMA;
        SIM_SCGC6 |= SIM_SCGC6_DMAMUX;

        // if either transmitter or receiver is enabled, do nothing
        if (I2S0_TCSR & I2S_TCSR_TE) return;
        if (I2S0_RCSR & I2S_RCSR_RE) return;

        // Select input clock 0
        // Configure to input the bit-clock from pin, bypasses the MCLK
            divider
        I2S0_MCR = I2S_MCR_MICS(0);
        I2S0_MDR = 0;

        // configure transmitter
        I2S0_TMR = 0;
        I2S0_TCR1 = I2S_TCR1_TFW(8);
        I2S0_TCR2 = I2S_TCR2_SYNC(0) | I2S_TCR2_BCP;
        I2S0_TCR3 = I2S_TCR3_TCE;
        I2S0_TCR4 = I2S_TCR4_FRSZ(15) | I2S_TCR4_SYWD(0) | I2S_TCR4_MF
                | I2S_TCR4_FSE | I2S_TCR4_FSP;//|I2S_TCR4_FSD;//FSD
        I2S0_TCR5 = I2S_TCR5_WNW(31) | I2S_TCR5_W0W(31) | I2S_TCR5_FBT(31);

        // configure receiver (sync'd to transmitter clocks)
        I2S0_RMR = 0;
        I2S0_RCR1 = I2S_RCR1_RFW(8);
        I2S0_RCR2 = I2S_RCR2_SYNC(1) | I2S_TCR2_BCP ;
        I2S0_RCR3 = I2S_RCR3_RCE;
        I2S0_RCR4 = I2S_RCR4_FRSZ(15) | I2S_RCR4_SYWD(0) | I2S_RCR4_MF
                | I2S_RCR4_FSE | I2S_RCR4_FSP| I2S_RCR4_FSD;
        I2S0_RCR5 = I2S_RCR5_WNW(31) | I2S_RCR5_W0W(31) | I2S_RCR5_FBT(31);

        // configure pin mux for 3 clock signals
        CORE_PIN23_CONFIG = PORT_PCR_MUX(6); // pin 23, PTC2, I2S0_TX_FS (
            LRCLK)
        CORE_PIN9_CONFIG  = PORT_PCR_MUX(6); // pin  9, PTC3, I2S0_TX_BCLK
        CORE_PIN11_CONFIG = PORT_PCR_MUX(6); // pin 11, PTC6, I2S0_MCLK
}

void AudioOutputTDMslave::begin(void)
{
        Serial.print("\n AudioOutputTDMslave:begin ");
        dma.begin(true); // Allocate the DMA channel first

        for (int i=0; i < MAX_CHANNELS; i++) {
                block_input[i] = NULL;
        }

        // TODO: should we set & clear the I2S_TCSR_SR bit here?
        AudioOutputTDMslave::config_tdm();
        CORE_PIN22_CONFIG = PORT_PCR_MUX(6); // pin 22, PTC1, I2S0_TXD0

        dma.TCD->SADDR = tdm_tx_buffer;
```

```
        dma.TCD->SOFF = 4;
        dma.TCD->ATTR = DMA_TCD_ATTR_SSIZE(2) | DMA_TCD_ATTR_DSIZE(2);
        dma.TCD->NBYTES_MLNO = 4;
        dma.TCD->SLAST = -sizeof(tdm_tx_buffer);
        dma.TCD->DADDR = &I2S0_TDR0;
        dma.TCD->DOFF = 0;
        dma.TCD->CITER_ELINKNO = sizeof(tdm_tx_buffer) / 4;
        dma.TCD->DLASTSGA = 0;
        dma.TCD->BITER_ELINKNO = sizeof(tdm_tx_buffer) / 4;
        dma.TCD->CSR = DMA_TCD_CSR_INTHALF | DMA_TCD_CSR_INTMAJOR;
        dma.triggerAtHardwareEvent(DMAMUX_SOURCE_I2S0_TX);
        update_responsibility = update_setup();
        dma.enable();

        I2S0_TCSR = I2S_TCSR_SR;
        I2S0_TCSR = I2S_TCSR_TE | I2S_TCSR_BCE | I2S_TCSR_FRDE;
        dma.attachInterrupt(isr);
}

#endif // KINETISK
```

effect_freeverb.h

```
/* Freeverb for teensy
*
* Copyright (c) 2017, Yasmeen Sultana
*
*
* Permission is hereby granted, free of charge, to any person obtaining a
    copy
* of this software and associated documentation files (the "Software"), to
    deal
* in the Software without restriction, including without limitation the
    rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice, development funding notice, and this permission
* notice shall be included in all copies or substantial portions of the
    Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*/
```

```
#ifndef effect_freeverb_
#define effect_freeverb_

#include "AudioStream.h"

/*All pass filters*/
#define APF_COUNT                                   4

#define APF1_DELAY_LENGTH                           225
#define APF2_DELAY_LENGTH                           556
#define APF3_DELAY_LENGTH                           441
#define APF4_DELAY_LENGTH                           341

/*low  pass  feedback combo filters*/

#define LBCF_COUNT              8

#define LBCF1_DELAY_LENGTH                          1557
#define LBCF2_DELAY_LENGTH                          1617
#define LBCF3_DELAY_LENGTH                          1491
#define LBCF4_DELAY_LENGTH                          1422
#define LBCF5_DELAY_LENGTH                          1277
#define LBCF6_DELAY_LENGTH                          1356
#define LBCF7_DELAY_LENGTH                          1188
#define LBCF8_DELAY_LENGTH                          1116

#define RIGHT_STERO_OFFSET                          23

class AudioEffectFreeverb : public AudioStream
{
public:
        AudioEffectFreeverb(void) : AudioStream(1, inputQueueArray)
        {
                /*constructor*/
                init_apf(0.5f,0);
                init_lbcf(0.84f,0.2f,0);

        }
        AudioEffectFreeverb(bool rightChannel) : AudioStream(1,
            inputQueueArray)
        {
                if (rightChannel==true)
                {
                        init_apf(0.5f, RIGHT_STERO_OFFSET);
                        init_lbcf(0.84f, 0.2f, RIGHT_STERO_OFFSET);
                }
                else
                {
                        init_apf(0.5f, 0);
                        init_lbcf(0.84f, 0.2f, 0);
                }
        }
```

```
  AudioEffectFreeverb(float apf_gain, float feedback,float damp,bool
      rightChannel) : AudioStream(1, inputQueueArray)
        {
                if (rightChannel==true)
                {
                        init_apf(apf_gain, RIGHT_STERO_OFFSET);
                        init_lbcf(feedback, damp, RIGHT_STERO_OFFSET);
                }
                else
                {
                        init_apf(apf_gain, 0);
                        init_lbcf(feedback, damp, 0);
                }
        }
        virtual void update(void);

private:
        struct allpass_filter
        {
                int32_t   gain;
    int32_t   gain1;
                int32_t   *pbuffer;
                uint32_t  buf_len;
                uint32_t  delay;
                uint32_t  bufferIndex;
        };
        struct lowpass_comb_filter
        {
                int32_t   damp1; /*d*/
                int32_t   damp2; /*1-d*/
                int32_t   feedback; /*f*/
                int32_t   state_z1; /**/
                int32_t   *pbuffer;
                uint32_t  buf_len;
                uint32_t  delay;
                uint32_t   bufferIndex;
        };

        audio_block_t *inputQueueArray[1];

        struct allpass_filter           apf[APF_COUNT];
        struct lowpass_comb_filter      lbcf[LBCF_COUNT];
    void init_apf(float gain,int32_t offset);
        static void process_apf(struct allpass_filter *apf, int32_t *in_buf,
            int32_t *out_buf);
    void init_lbcf(float feedback, float damp,int32_t offset);
        static void process_lbcf(struct lowpass_comb_filter *lbcf, int32_t *
            in_buf, int32_t *out_buf);

        int32_t apf1_buf[APF1_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t apf2_buf[APF2_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t apf3_buf[APF3_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
```

```
        int32_t apf4_buf[APF4_DELAY_LENGTH+ RIGHT_STERO_OFFSET];

        int32_t lbcf1_buf[LBCF1_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf2_buf[LBCF2_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf3_buf[LBCF3_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf4_buf[LBCF4_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf5_buf[LBCF5_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf6_buf[LBCF6_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf7_buf[LBCF7_DELAY_LENGTH+ RIGHT_STERO_OFFSET];
        int32_t lbcf8_buf[LBCF8_DELAY_LENGTH+ RIGHT_STERO_OFFSET];

  int32_t q31_buf[AUDIO_BLOCK_SAMPLES];
  int32_t sum_buf[AUDIO_BLOCK_SAMPLES];
  int32_t aux_buf[AUDIO_BLOCK_SAMPLES];

};

#endif
```

effect_freeverb.cpp

```
/*
 * Copyright (c) 2017 Yasmeen Sultana
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
    copy
 * of this software and associated documentation files (the "Software"), to
    deal
 * in the Software without restriction, including without limitation the
    rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
     all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
    THE
 * SOFTWARE.
 */

#include "effect_freeverb.h"
#include "utility/dspinst.h"
```

```
#include "math_helper.h"

#if 1
 void
AudioEffectFreeverb::process_apf(struct allpass_filter *apf, int32_t *in_buf,
     int32_t *out_buf)
{
        int32_t bufout;
        int32_t input;
        int32_t z1,w;
        int32_t n;
        int32_t output;

        //bufout = buffer[bufidx];
        //buffer[bufidx] = input + (bufout * feedback);
        //output = -buffer[bufidx] * feedback + bufout;

for (n = 0; n < AUDIO_BLOCK_SAMPLES; n++)
        {
                bufout = apf->pbuffer[apf->bufferIndex];
                input = in_buf[n];
                z1 = multiply_32x32_rshift32_rounded( bufout, apf->gain);
                input += (z1 << 2);
                w= multiply_32x32_rshift32_rounded(-input, apf->gain);
                output =bufout +(w << 2);
                out_buf[n] = output;
                apf->pbuffer[apf->bufferIndex] = input;
                apf->bufferIndex++;
                if (apf->bufferIndex >= apf->delay)
                {
                        apf->bufferIndex = 0;
                }
        }
}
#endif

#if 0
void
AudioEffectFreeverb::process_apf(struct allpass_filter *apf, int32_t *in_buf,
     int32_t *out_buf)
{
        int32_t bufout;
        int32_t input;
        int32_t z1,w;
        int32_t n;
        int32_t output;
  int32_t bufGain;

        //bufout = buffer[bufidx];
        //buffer[bufidx] = input + (bufout * feedback);
        //output = -buffer[bufidx] * feedback + bufout;
```

```
        for (n = 0; n < AUDIO_BLOCK_SAMPLES; n++)
        {
                bufout = apf->pbuffer[apf->bufferIndex];
                input = in_buf[n];
                z1 = multiply_32x32_rshift32_rounded( bufout, apf->gain);
                //input += (z1 << 2);
    bufGain = (input +(z1 << 2));
                w= multiply_32x32_rshift32_rounded(-bufGain, apf->gain1);
                output = bufout +(w << 2);
                out_buf[n] = output;
                apf->pbuffer[apf->bufferIndex] = bufGain;
                apf->bufferIndex++;
                if (apf->bufferIndex >= apf->delay)
                {
                        apf->bufferIndex = 0;
                }
        }
}
#endif
void
AudioEffectFreeverb::process_lbcf(struct lowpass_comb_filter *lbcf, int32_t *
    in_buf, int32_t *out_buf)
{
        int32_t bufout;
        int32_t input;
        int32_t n;
        int32_t sum=0;
        int32_t sum2 = 0;
        /*
        output = buffer[bufidx];
        filterstore = (output * damp2) + (filterstore * damp1);
        buffer[bufidx] = input + (filterstore * feedback);
        */
        for (n = 0; n < AUDIO_BLOCK_SAMPLES; n++)
        {
                bufout = lbcf->pbuffer[lbcf->bufferIndex];
                input = in_buf[n];

                sum = multiply_32x32_rshift32_rounded( bufout, lbcf->damp2);
                sum2= multiply_32x32_rshift32_rounded(lbcf->state_z1, lbcf->
                    damp1);
                lbcf->state_z1 = ((sum + sum2) << 2);
                sum = multiply_32x32_rshift32_rounded(lbcf->state_z1, lbcf->
                    feedback);
                sum2 = input + (sum << 2);
                lbcf->pbuffer[lbcf->bufferIndex] = sum2;
                out_buf[n] = bufout;
                lbcf->bufferIndex++;
                if (lbcf->bufferIndex >= lbcf->delay)
                {
                        lbcf->bufferIndex = 0;
                }
```

```
        }
}
void AudioEffectFreeverb::init_apf(float gain,int32_t offset)
{
        uint8_t n = 0;

        for (n = 0; n < APF_COUNT; n++)
        {
                apf[n].gain = (int32_t)(gain*1073741824.0); /*2.30 format*/
    apf[n].gain1 = (int32_t)((1+gain)*1073741824.0); /*2.30 format*/

    apf[n].bufferIndex = 0;
        }

        memset(&apf1_buf[0], 0, sizeof(apf1_buf));
        memset(&apf2_buf[0], 0, sizeof(apf2_buf));
        memset(&apf3_buf[0], 0, sizeof(apf3_buf));
        memset(&apf4_buf[0], 0, sizeof(apf4_buf));

        apf[0].pbuffer  = &apf1_buf[0];
        apf[0].delay    = APF1_DELAY_LENGTH + offset;
        apf[0].buf_len = APF1_DELAY_LENGTH + offset;

        apf[1].pbuffer  = &apf2_buf[0];
        apf[1].delay    = APF2_DELAY_LENGTH + offset;
        apf[1].buf_len  = APF2_DELAY_LENGTH + offset;

        apf[2].pbuffer  = &apf3_buf[0];
        apf[2].delay    = APF3_DELAY_LENGTH + offset;
        apf[2].buf_len  = APF3_DELAY_LENGTH + offset;

        apf[3].pbuffer  = &apf4_buf[0];
        apf[3].delay    = APF4_DELAY_LENGTH + offset;
        apf[3].buf_len = APF4_DELAY_LENGTH + offset;

}

void AudioEffectFreeverb::init_lbcf(float feedback, float damp,int32_t offset
    )
{
        uint8_t n = 0;

        for (n = 0; n < LBCF_COUNT; n++)
        {
                lbcf[n].feedback        = (int32_t)(feedback*1073741824.0);
                    /*2.30 format*/
                lbcf[n].damp1           = (int32_t)(damp*1073741824.0);
                            /*2.30 format*/
                lbcf[n].damp2           = (int32_t)((1-damp)*1073741824.0);
                          /*2.30 format*/
        }
```

```
        memset(&lbcf1_buf[0], 0, sizeof(lbcf1_buf));
        memset(&lbcf2_buf[0], 0, sizeof(lbcf2_buf));
        memset(&lbcf3_buf[0], 0, sizeof(lbcf3_buf));
        memset(&lbcf4_buf[0], 0, sizeof(lbcf4_buf));
        memset(&lbcf5_buf[0], 0, sizeof(lbcf5_buf));
        memset(&lbcf6_buf[0], 0, sizeof(lbcf6_buf));
        memset(&lbcf7_buf[0], 0, sizeof(lbcf7_buf));
        memset(&lbcf8_buf[0], 0, sizeof(lbcf8_buf));

        lbcf[0].pbuffer = &lbcf1_buf[0];
        lbcf[0].delay   = LBCF1_DELAY_LENGTH+ offset;
        lbcf[0].buf_len = LBCF1_DELAY_LENGTH+ offset;

        lbcf[1].pbuffer = &lbcf2_buf[0];
        lbcf[1].delay   = LBCF2_DELAY_LENGTH + offset;
        lbcf[1].buf_len = LBCF2_DELAY_LENGTH + offset;

        lbcf[2].pbuffer = &lbcf3_buf[0];
        lbcf[2].delay   = LBCF3_DELAY_LENGTH + offset;
        lbcf[2].buf_len = LBCF3_DELAY_LENGTH + offset;

        lbcf[3].pbuffer = &lbcf4_buf[0];
        lbcf[3].delay   = LBCF4_DELAY_LENGTH + offset;
        lbcf[3].buf_len = LBCF4_DELAY_LENGTH + offset;

        lbcf[4].pbuffer = &lbcf5_buf[0];
        lbcf[4].delay   = LBCF5_DELAY_LENGTH + offset;
        lbcf[4].buf_len = LBCF5_DELAY_LENGTH + offset;

        lbcf[5].pbuffer = &lbcf6_buf[0];
        lbcf[5].delay   = LBCF6_DELAY_LENGTH + offset;
        lbcf[5].buf_len = LBCF6_DELAY_LENGTH + offset;

        lbcf[6].pbuffer = &lbcf7_buf[0];
        lbcf[6].delay   = LBCF7_DELAY_LENGTH + offset;
        lbcf[6].buf_len = LBCF7_DELAY_LENGTH + offset;

        lbcf[7].pbuffer = &lbcf8_buf[0];
        lbcf[7].delay   = LBCF8_DELAY_LENGTH + offset;
        lbcf[7].buf_len = LBCF8_DELAY_LENGTH + offset;

}
void
AudioEffectFreeverb::update(void)
{

  audio_block_t *block;

  if (!(block = receiveWritable()))
    return;

  if (!block->data)
```

```
    return;

  arm_q15_to_q31(block->data, q31_buf, AUDIO_BLOCK_SAMPLES);
  arm_shift_q31(q31_buf, -3, q31_buf, AUDIO_BLOCK_SAMPLES);
  process_lbcf(&lbcf[0], q31_buf, sum_buf);
  process_lbcf(&lbcf[1], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_lbcf(&lbcf[2], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_lbcf(&lbcf[3], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_lbcf(&lbcf[4], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_lbcf(&lbcf[5], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_lbcf(&lbcf[6], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_lbcf(&lbcf[7], q31_buf, aux_buf);
  arm_add_q31(sum_buf, aux_buf, sum_buf, AUDIO_BLOCK_SAMPLES);

  process_apf(&apf[0], sum_buf, q31_buf);
  process_apf(&apf[1], q31_buf, q31_buf);
  process_apf(&apf[2], q31_buf, q31_buf);
  process_apf(&apf[3], q31_buf, q31_buf);
  arm_q31_to_q15(q31_buf, block->data, AUDIO_BLOCK_SAMPLES);

  transmit(block, 0);

  release(block);
}
```

ctag_face24_reverb.ino

```
#include <Audio.h>
#include <Wire.h>
#include <SPI.h>

#include"control_ad1938.h"

/*
AD1938 pins  Teensy gpio pins
Chip select - 7
MOSI        - 11
MISO         -12
SCK          -14
RESET        -17
```

```
DAC/ADCBLK      -9
DAC/ADCLRCLK    -23

DSDATA1 TX    -22
ASDARA1 RX    -13
*/

/*
The daisy chain connection of two ad1938 and Teensy are as follows
AD1938 (slave)->AD1938(master)->Teensy(slave)
*/

int clatch_slave =6;//ad1938 slave spi latch
int clatch =7;       //ad1938 master spi latch
int cout=12;         //spi miso
int cin =11;         //spi  mosi
int cclk =14;        //spi clock
int reset_pin_slave =16; //ad1938 slave reset pin
int reset_pin =17;        //ad1938 master reset pin

AudioControlAD1938       ad1938master;
AudioControlAD1938       ad1938slave;

AudioMixer4              input_gain;
AudioMixer4              left_mix;
AudioMixer4              right_mix;
/*default values 0.5f,0.84f,0.2*/
AudioEffectFreeverb       reverbL(0.5f,0.84f,0.2f,false);/*allpass gain,
    feedback,damping, stereo pad*/
AudioEffectFreeverb       reverbR(0.5f,0.84f,0.2f,true);/*adding stereo
    padding for right channel*/

#if 1
//only for slave ++
//AudioInputTDM            i2s_in;
//AudioOutputTDM           i2s_out;
//only for slave --

AudioSynthWaveformSine   sine1;

//only for ad138 master  and teensy slave++
AudioInputTDMslave   i2s_in;
AudioOutputTDMslave  i2s_out;
//only for master --

/*ADC0 to DAC0*/
//AudioConnection          pc0(i2s_in, 0, i2s_out, 0);  // ADC1 L -> DAC1 L
//AudioConnection          pc2(i2s_in, 2, i2s_out, 2);  // ADC1 R -> DAC1 R
/*freeverb */
/*mix left and right channels*/
AudioConnection          patchCord1(i2s_in, 0, input_gain, 0);
AudioConnection          patchCord2(i2s_in, 2, input_gain, 1);
```

```
/*give the combined signal to freeverb class*/
AudioConnection          patchCord3(input_gain, reverbL);
AudioConnection          patchCord4(input_gain, reverbR);

/*Mix the left channel*/
AudioConnection          patchCord5(reverbL, 0, left_mix, 0);
AudioConnection          patchCord6(reverbR, 0, left_mix, 1);
AudioConnection          patchCord7(i2s_in, 0, left_mix, 2);

/*Mix the right channel*/
AudioConnection          patchCord8(reverbR, 0, right_mix, 0);
AudioConnection          patchCord9(reverbL, 0, right_mix, 1);
AudioConnection          patchCord10(i2s_in, 1, right_mix, 2);

AudioConnection          patchCord11(left_mix, 0, i2s_out, 0);
AudioConnection          patchCord12(right_mix, 0, i2s_out, 2);

/*ADC1 to DAC1*/
AudioConnection          pc4(i2s_in, 4, i2s_out, 4);  // ADC2 L -> DAC2 L
AudioConnection          pc6(i2s_in, 6, i2s_out, 6);  // ADC2 R -> DAC2 R

/*ADC2 to DAC2*/
AudioConnection          pc8(i2s_in, 8, i2s_out, 12);    // ADC3 L -> DAC3 L
AudioConnection          pc10(i2s_in,10, i2s_out, 14);  // ADC3 R -> DAC3 R

/*ADC3 to DAC3*/
AudioConnection          pc12(i2s_in, 12, i2s_out, 8); // ADC4 L -> DAC4 L
AudioConnection          pc14(i2s_in, 14, i2s_out, 10); // ADC4 R -> DAC4 R
#if 1//TMD16
/*sine to DAC4*/
AudioConnection          pc16(i2s_in, 0, i2s_out, 16); // ADC1 L -> DAC5 L
AudioConnection          pc18(i2s_in, 2, i2s_out, 18); // ADC1 R -> DAC5 R
/*sine to DAC5*/
AudioConnection          pc20(i2s_in, 4, i2s_out, 20); // ADC2 L -> DAC6 L
AudioConnection          pc22(i2s_in, 6, i2s_out, 22); // ADC2 R -> DAC6 R

/*sine to DAC6*/
AudioConnection          pc24(i2s_in, 0, i2s_out, 24); // sine -> DAC7 L
AudioConnection          pc26(i2s_in , 2 , i2s_out, 26); // sine -> DAC7 R
/*sine to DAC 7*/
AudioConnection          pc28(sine1, 0, i2s_out, 28); // sine -> DAC8 L
AudioConnection          pc30(sine1, 0, i2s_out, 30); // sine -> DAC8 R

#endif

#endif

void setup() {
  float wet, wet1, wet2,dry,dry1;

 const float scaleWet = 3;
 const float scaleDry = 2;
```

```
 float effectMix = 0.5;// only wet
 float width = 0.5;// complete seperation (1 no effect from the other channel
     )

#if 1

 wet1 = scaleWet * effectMix;
 dry1 = scaleDry * (1.0-effectMix);

wet =wet1/(wet1+dry1); //scaling
dry =dry1/(wet1+dry1);

wet1 = wet * (width/2.0 + 0.5);
wet2 = wet * (1.0 - width)/2.0;

input_gain.gain(0,0.5);
input_gain.gain(1,0.5);

/*
wet1 =1.0; wet2 =0; dry=1.0;
*/
/*left channel matrix*/
left_mix.gain(0,wet1);
left_mix.gain(1,wet2);
left_mix.gain(2,dry);

/*rigt channel matrix gain*/
right_mix.gain(0,wet1);
right_mix.gain(1,wet2);
right_mix.gain(2,dry);
#endif
  // put your setup code here, to run once:
 delay(2000);
 Serial.begin(115200);
 sine1.frequency(440);
 sine1.amplitude(0.4);

 #if 1
  ad1938slave.spiInit( clatch_slave, reset_pin_slave, cout, cin, cclk);
  delay(200);
  //ad1938slave.config(FS_48000,BITS_24,I2S_TDM_8CH,AD1938_I2S_SLAVE);
  ad1938slave.config(FS_48000,BITS_16,I2S_TDM_16CH,AD1938_I2S_SLAVE);
  ad1938slave.volume(1);

delay(200);

#endif

#if 1
  /*configure AD1938 (slave) wit */
  ad1938master.spiInit( clatch, reset_pin, cout, cin, cclk);
  delay(200);
```

```
  //ad1938master.config(FS_48000,BITS_24,I2S_TDM_8CH,AD1938_I2S_MASTER);
  ad1938master.config(FS_48000,BITS_16,I2S_TDM_16CH,AD1938_I2S_MASTER);
  ad1938master.volume(1);
  ad1938master.enable();
  ad1938slave.enable();

 #endif
  AudioMemory(256);
  AudioInterrupts();

 AudioProcessorUsageMaxReset();
 AudioMemoryUsageMaxReset();
  reverbL.processorUsageMaxReset();
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print("\n");
  Serial.print("CPU: ");
  Serial.print("freeverb left =");
  Serial.print(reverbL.processorUsage());
  Serial.print(",");
  Serial.print(reverbL.processorUsageMax());
  Serial.print("  ");
  Serial.print("Freeverb right=");
  Serial.print(reverbR.processorUsage());
  Serial.print(",");
  Serial.print(reverbR.processorUsageMax());
  Serial.print("  ");
  Serial.print("I2S =");
  Serial.print(i2s_in.processorUsage());
  Serial.print(",");
  Serial.print(i2s_in.processorUsageMax());
  Serial.print("  ");
  Serial.print("all=");
  Serial.print(AudioProcessorUsage());
  Serial.print(",");
  Serial.print(AudioProcessorUsageMax());
  Serial.print("    ");
  Serial.print("Memory: ");
  Serial.print(AudioMemoryUsage());
  Serial.print(",");
  Serial.print(AudioMemoryUsageMax());
  Serial.print("    ");

  Serial.println();
  delay(1000);
}
```