# Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos

as a demonstration of an open sound card system

Smilen Dimitrov
Aalborg University Copenhagen
Lautrupvang 15
DK-2750 Ballerup, Denmark
sd@{imi,create}.aau.dk

Stefania Serafin
Aalborg University Copenhagen
Lautrupvang 15
DK-2750 Ballerup, Denmark
sts@{imi,create}.aau.dk

## ABSTRACT

A contemporary PC user, typically expects a sound card to be a piece of hardware, that: can be manipulated by 'audio' software (most typically exemplified by 'media players'); *and* allows interfacing of the PC to audio reproduction and/or recording equipment. As such, a 'sound card' can be considered to be a *system*, that encompasses design decisions on both hardware and software levels - that also demand a certain understanding of the architecture of the target PC operating system.

This project outlines how an Arduino Duemillanove board (containing a USB interface chip, manufactured by Future Technology Devices International Ltd [FTDI] company) can be demonstrated to behave as a full-duplex, mono, 8-bit 44.1 kHz soundcard, through an implementation of: a PC audio driver for **ALSA** (*Advanced Linux Sound Architecture*); a matching program for the Arduino's ATmega microcontroller - and nothing more than headphones (and a couple of capacitors). The main contribution of this paper is to bring a holistic aspect to the discussion on the topic of implementation of soundcards - also by referring to open-source driver, microcontroller code and test methods; and outline a complete implementation of an open - yet functional - soundcard system.

## Keywords

Sound card, Arduino, audio, driver, ALSA, Linux

## 1. INTRODUCTION

A sound card, being a product originally conceived in industry, can be said to have had a development path, where user demands interacted with industry competition, in order to produce the next generation of soundcard devices. As such, the soundcard has evolved to a product, that most of today's consumer PC users have very specific demands from: they expect to control the soundcard using their favorite 'media player' or 'recorder' audio software from the PC; while the soundcard interfaces with audio equipment like speakers or amplifiers. For professional users, the character of 'audio software' and 'audio equipment' may encompass far more specialized and complex systems – however, the expectations of the users in respect to basic interaction with this part of the system is still the same: high-level, PC software control of the audio reproduced or captured on the hardware.

A development of a soundcard thus requires, to some extent, an interdisciplinary approach - requiring knowledge of both electronics and software engineering, along with operating system architecture. But, even with a more intimate understanding of this architecture, a potential designer of a new soundcard may still experience a 'chicken-and-egg' problem: understanding drivers requires understanding of their target hardware - *and* vice versa. As such, considering this product's origins in industry, it is no wonder that literature discussing implementations of complete 'soundcards' is rare - both hardware and software designs would have to be disclosed, for the discussion to be relevant.

*An open soundcard.* Businesses are, understandably, not likely to disclose hardware designs and driver code publicly; this may explain the difficulty in tracking down prior open devices. It is here that the Arduino [2] platform comes into play. Marketed and sold as an open-source product, it is essentially a board which represents a connection between a USB interface chip, and a microcontroller. As the schematics are available, an Arduino board can, in principle, be assembled by hand - however, a factory production has both a low, popular price; and brings in a level of expected performance, which allows for easier elimination of problems of electrical nature during development. Thus, on one hand, an Arduino board represents *known* hardware - one we could write an **ALSA** driver for; both in principle, and - as this project demonstrates - in reality. On the other hand, the Arduino is typically marketed as supporting communication speeds of up to 115200 bps (an impression also stated in [4]) - which result with data rates, insufficient to demonstrate streaming audio close to the contemporary CD-quality standard (stereo, 16-bit, 44.1 kHz). Yet, the major individual components: FTDI USB interface chip, and ATmega microcontroller - are both individually marketed to support up to 2 Mbps: a data rate that can certainly sustain a CD-quality signal. Thus, in spite of being *known* hardware, the Arduino may have 'officially unsupported' modes of operation, that would allow it to perform as a soundcard - modes that, however, still need to be quantified in the same sense, as if we were starting to design a board from scratch (*with this particular microcontroller, and USB interface chip*).

*Application example.* An open soundcard may bring actual benefits to electronic instrument designers, beyond the opportunity for technical study: consider a system where a vibrating surface (cymbal) is captured using a sensor and Arduino into **PD** software, where it is used to modulate a digital audio signal in realtime. Usual approach would be to read the ARDUINO as a serial port at 115200 bps; this limits the analog bandwidth ($\approx$ 5kHz) and forces the user to code a conversion to **PD**'s audio signal domain; with **AudioArduino** the sensor data could be received directly as a 44.1 kHz audio signal in **PD** - full audio analog bandwidth, no need for signal conversions.

## 2. PREVIOUS WORK

Previous attempts to discuss open soundcard implementations couldn't provide a basis for the development here: the Linux kernel contains many open soundcard drivers, but written for commercial (typically undisclosed) hardware. The now defunct german magazine Elrad may have had a series on implementation of a PCI card in 1997, but the remaining reference[1] doesn't contain any useful information. The ARDUINO has previously been used for audio: in [9] as a standalone player; [12] as a standalone DSP - but not specifically as a PC-interfaced soundcard. Thus, this project's basis is mostly in own previous work: [4] demonstrates legacy hardware controlled by PC software; and identifies data throughput control as the main problem in that naïve approach. Modern operating systems address this issue by providing a *driver architecture*; where, in programming a *driver*, the programmer gains a more fine-grained temporal control. In the context of the open **GNU/Linux** operating system(s), acquaintance with its current low-level audio library - **ALSA** - is thus necessary for implementation of soundcard drivers. This project has produced the tutorial driver **minivosc** [7] as an introductory overview of **ALSA** architecture - also used as a starting point of the work in this paper.

## 3. DEGREES OF FREEDOM

It would be interesting to qualify to what extent can **AudioArduino** - a system of ARDUINO Duemillanove, microcontroller code, and matching **ALSA** soundcard driver - be considered to be an 'open' 'soundcard system'. To begin with, hardware production necessarily involves mineral extraction and processing, manufacturing, and distribution - stages that require considerable economic infrastructure; and therefore, there will always be a 'hard' price attributed to it. On the other hand software, in essence, represents the instructions - information - for what we can *do* with this hardware. With the increasing affordability causing mass penetration of computing technology, fewer 'hard' investments need to be made to start with software development; and in principle, the pursuit of software development could thereafter involve only investment of the time of the developer. While developer time also carries inherent cost with it, there are circumstances where sharing the outcome - the source code - becomes preferable, for academic, business or altruistic reasons; especially since, with the expansion of the Internet, the physical cost of sharing information can be considered negligible.

Thus, it is in context of software that the term(s) 'free' or 'open' will be applied in this project (as in FLOSS[2]). To begin with, the driver is developed on **Ubuntu** - a FLOSS **GNU/Linux** operating system; with the main corresponding tool for development, **gcc**, being likewise open. The audio framework for **Linux**, **ALSA**, follows the same license - and the main high-level, user audio programs used, **Audacity** and **arecord**, are likewise open. The ARDUINO as a platform is known to be open, by making the schematic files available, as well as offering an integrated development environment (IDE) for **Linux**, which is also open [2]. The microcontrollers used in the platform are typically ATMEGA's, part of the ATMEL AVR family, which (given the tolerance of Atmel to open source, see Atmel Application note *AVR911*, also [14]) has long had an open toolchain for programming, **avr-gcc**.

At this point, let's note that ARDUINO in 2010 released the ARDUINO UNO board, which is taken to be the 'reference version' for the platform. The reason for this is that the USB interface chip used on the UNO is ATMEGA8U2, and the USB interface functionality is provided by the open-source **LUFA** (Lightweight USB Framework for AVR) firmware. In contrast, earlier versions of USB ARDUINOs, like the DUEMILLANOVE, feature a FTDI FT232RL USB interface chip. FTDI offers two drivers, VCP (Virtual COM Port, offering a standard serial port emulation) and D2XX (direct access) [18, 'Drivers']. Both of these are provided free of charge - however, source code is not available. Also, VCP may offer data transfer rates up to 300 kilobyte/second, while D2XX up to 1 Megabyte/second ([18, 'Products/ICs/FT245R']). Nonetheless, there exists a third-party open-source driver for FTDI in the **Linux** kernel, which corresponds to VCP, named **ftdi-sio** [11] - in fact, **ftdi-sio** forms the basis of the **AudioArduino** driver. With this, the following parts of the **AudioArduino** system can be considered open: *microcontroller code*, and tools to implement/debug it; *audio driver*, and tools to implement/debug it; *operating system*, hosting the development tools, the driver and high-level software; and *high-level audio software*, needed to demonstrate actual functionality – i.e., the bulk of the software domain. The driver was developed on **Ubuntu** 10.04 (Lucid), utilizing the 2.6.32 version of the **Linux** kernel; the code has been released as open source, and it can be found by referring to the home page [3].

## 4. CONCEPT OF AudioArduino

Given that the ATMEGA328 features both ADC, and DAC (in form of PWM), converters - using the ARDUINO as a soundcard hardware is a feasible idea, as long as one trusts that the data transfer between the PC and the ATMEGA328 can occur without errors at audio rates. Developing a USB driver for such data transfer would, essentially, require a good working knowledge of the USB bus and its specifications. However, that is a daunting task for any developer - the USB 2.0 Specification [19] alone is 650 pages long; with actual implementation, in a form of a driver for a given OS, requiring additional effort. Therefore, the starting point of this project is to abstract the USB transport to the greatest extent possible, and avoid dealing with particular details of the USB protocol. This is possible because of the particular architecture of the ARDUINO board, rendered on Fig. 1.

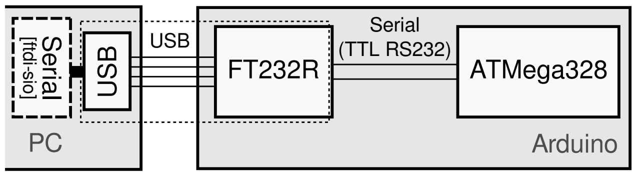

**Figure 1: Simplified context of an ARDUINO, connected to a PC.**

As Fig. 1 shows, the **ftdi-sio** driver makes the FT232 device appear as a 'serial port' in the PC OS, that the user can write arbitrary data to. The driver will format this data as necessary for USB transport, and send it on wire; the FT232 will then accept this data and convert it

[1]http://www.xs4all.nl/~fjkraan/digaud/elrad/pcirec.html
[2]free/libre/open source software

to TTL-level (0-5V) RS-232 signal (and the same happens for the reverse direction, when reading). Given that RS-232 is conceptually much easier to understand (e.g., [5]); we can 'black box' (abstract) the *unknown* (USB) part in the data transfer - and focus on the *known* (RS-232) part.

In order to specify what sampling rates, in terms of digital audio, would this hardware support - the most important factor to consider is the data transfer rate, that can be achieved between the ATMEGA328 and the FT232 over the serial link. As far as this serial link goes, the ATMEGA328 states maximum rate of 2.5 Mbps [16, pg.199]; while the FT232 states up to 3 Mbaud [17, pg.16]. As the **ftdi-sio** driver supports 2 Mbps[3] by default, this is the 'theoretical' speed that should be possible to achieve all the way through to the ATMEGA328. A speed of 2 Mbaud translates to 200000 Bps[3], which would be enough to carry 200000/44100 = 4.5 mono/8-bit/44.1 kHz channels; or two mono/16-bit/44.1 kHz channels; or one CD quality stereo/16-bit/44.1 kHz channel. However, one still needs to determine what *actual* data transfer rates can be achieved, and under which conditions (such as different software). Beyond this, it is the response times of the ATMEGA328 (including DAC and ADC elements), that would limit the use as full-duplex device. The final issue is the analog I/O interface, discussed further in this paper.

*Building and running.* Both the source code, and instructions for building and running, can be found in [3] (and they are similar to those given in [7]). The source code consists of a modified version of [11], **ftdi_sio-audard.c**; the **ALSA**-specific part in **snd_ftdi_audard.h**; associated headers and a **Makefile**; and microcontroller code, **duplexAudard_an8m.pde**. The `.pde` code can be built and uploaded to the ARDUINO using the **Arduino IDE**.

With this in place, high-level audio software (like **Audacity**) will be able to address the ARDUINO, and play back and capture audio data through it. ARDUINO's analog input 0 (`AIN0`) is treated as a soundcard input; sensors (like potentiometers) connected to this input can have their signal captured at 44.1 kHz in audio software. ARDUINO's digital pin 6 (`D6`) is soundcard output; on which, when audio software plays back audio data, (analog) PWM output is generated (audible).

## 5. QUANTIFYING THROUGHPUT RATE - DUPLEX LOOPBACK

As mentioned, one of the biggest issues in estimating if the ARDUINO board can behave as a soundcard, is in measuring the actual data transfer rate that can be achieved. The initial question is what tools can be used for that: the **ftdi-sio** driver will make a connected ARDUINO appear as a special file in the **Linux** system (`/dev/ttyUSB0`), representing a serial port. The serial port settings, such as speed, can be changed by using the **stty** program. Thereafter writing character data to the ARDUINO can be performed by writing to the associated file, say, by using `echo 'some text' > /dev/ttyUSB0` - and reading by, say, `cat /dev/ttyUSB0`.

However, finding the actual data rate in either direction is not the only thing which is interesting; another interesting point is to what extent can the ARDUINO board be considered a *full-duplex* device; i.e., whether the device can both receive and send data *simultaneously* (which, in terms of soundcards, is a standard expected behaviour). To assess both points, we suggest the ATMEGA328 is programmed as a 'digital loopback': to listen for incoming serial data; and send back the received byte through serial, as soon as it has been received. Then for the PC side, we propose a simple threaded program, **writeread.c** [15]: it accepts an input file; initiates write and read operations on a serial port in separate threads, so they can run concurrently; writes the input file, and saves the received data in another; and times these operations, so that the throughput rate can be determined.

What this experiment shows, is that the usual `C` commands for reading and writing from a serial port (and by extension, user programs like `cat` or `echo`) do not carry the concept of a data rate - they simply try to transfer data as fast as possible; and even for 2 Mbps communication, these commands push data faster than the USB chip can handle, which results with kernel warnings. Therefore, it is up to the program author to implement some sort of buffering, that would provide an effective throughput rate. Yet even with this in place, limiting rate to 2 Mbps within **writeread.c** would *still* cause throttling warnings; but, limiting it to slightly *below* 2 Mbps allows for a error-less demonstration. The reason for this is likely in the asynchronous nature of the serial RS232 protocol: in not sharing a single clock; the PC, the FT232 and the ATMEGA328 each have a slightly different concept of what the basic time unit (clock tick) duration would be - and thus a different concept of what '2 Mbps' is. By lowering the data rate from **writeread.c**, we likely account for these differences, which allows for error-free transmission; and from the PC, we can typically measure around 98% of 2 Mbps achieved for error-free duplex transmission.

Moreover, during this digital loopback experiment, the signals of the `TX` and `RX` connections (between the FT232 and the ATMEGA328) were measured with an AGILENT 54621A[4] oscilloscope; captured with the open-source **agiload** for **Linux**; and analysed using a script produced by this project, written in **python** (utilizing **matplotlib**) that features a serial decoder, called **mwfview-ser.py** [3]. These measurements show that the time for the ATMEGA328 to receive a byte and send it back - the minimal 'quantum' of action, relevant for a 'digital duplex' - is around 6.940 µs (Fig. 2), which is approx. 31% of the 22.6 µs analog sample period (for 44.1 kHz rate); which specifies the latency bottleneck expected from the ARDUINO in 'digital loopback' mode.

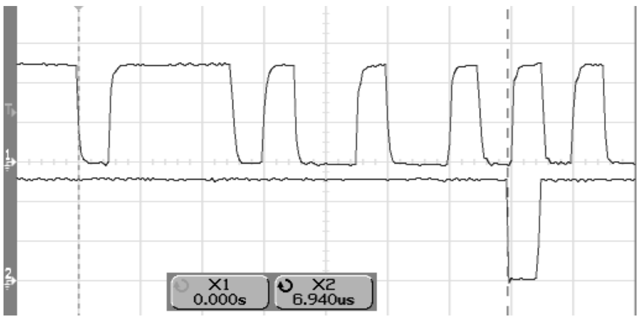

**Figure 2: Oscilloscope capture of RX (top) and TX (bottom) serial lines at the ATMega328, indicating latency between received and sent byte.**

Note that, the ATMEGA328's UART produces a signal

[3]Note that in 8-N-1 RS232 transfer, there are 8 data bits, 1 start and 1 stop bit; so 8-bit data is carried by 10-bit packet. Usually, 'baud' means 'signal transitions per second' and refers to all 10 bits, while 'bps' as 'bits per second' should refer to the 8 data bits only; but they can be often used interchangeably - 'Bps' as 'bytes per second' refers strictly to data payload (see also [15]).

[4]The AGILENT 54621A claims 60 MHz bandwidth, which is sufficient for capture of a 2 Mbps digital signal

with considerably more jitter than the FT232[5]; and there can be gaps in the otherwise sustained rate of serial transmission between the two - but none of this seems to harm error-free transmission at 2 Mbps. Finally, **writeread.c** works both with the 'vanilla' **ftdi-sio** driver, and the **AudioArduino** driver. Also, the same ARDUINO code used to demonstrate digital loopback with **writeread.c**, can be used with the **AudioArduino** driver - allowing for demonstration of a *digital audio loopback*: one can load a file in **Audacity**; play it back through the **AudioArduino** card; and by recording at the same time from the same card, one should capture the very same audio being played back (latency notwithstanding).

## 6. MICROCONTROLLER CODE

There are two distinct versions of microcontroller code for the ATMEGA328 used in this project, both in a form of a `C` language `.pde` file (the default format compilable in the ARDUINO IDE). The first is the mentioned 'digital duplex' code, which simply sends back any byte received through serial, posted in [15]. The main issues here are: the setup of the ATMEGA328's UART to support 2 Mbps (which is not supported in the default ARDUINO API); removing all overhead due to API function calls, by using the function source code directly; and disabling all irrelevant interrupts - before the ARDUINO can start showing 98% of 2 Mbps with **writeread.c**. Beyond this, the code can be implemented either as a single loop, or with interrupts on incoming serial data; with no significant difference in respect to performance. This is the same microcontroller code used as basis for development of the **AudioArduino** driver.

Once the **AudioArduino** driver was confirmed to be working with the 'digital duplex' code - a new, second 'analog I/O' version was written, which also employs the ADC and PWM (as DAC) facilities of the ATMEGA328. This version, as it is supposed to support audio playback and recording, requires deeper involvement with the ATMEGA328 datasheet [16]. In essence, the problem is that **ALSA** will send (mono) data at rate of 44100 Bps, which will appear as chunks of bytes on the 200000 Bps serial ARDUINO line; these bytes need to be stored as soon as possible by the ATMEGA328 in memory (buffer). On the other hand, at a rate of 44100 Hz, the ATMEGA328 should read one byte from the buffer and write it to PWM (the DAC) - and at the same time, read a byte from the ADC, and send it via serial. As we would expect an 8-bit interface (where each byte represents an analog sample) at the driver side, no further digital sample processing needs to be done in either direction. This is solved by code that employs an interrupt on incoming data, where the data is stored in a circular buffer - and a (16-bit) timer interrupt to handle the analog I/O at the 44100 Hz analog rate [3]. Note that this 'analog I/O' version seems to only perform well when implemented with incoming data handled on interrupt; trying to do the same handling in a single loop reveals problems with determining *when* an incoming byte is ready to be read from ATMEGA's UART [3].

## 7. DRIVER ARCHITECTURE

The **AudioArduino** driver is not only based on **ftdi-sio** - **ftdi_sio-audard.c** is a renamed version of [11], with several changes: first, it includes **snd_ftdi_audard.h**, which here is not used in the standard sense of a `C` header, but simply as a container for **ALSA** relevant code (which would, otherwise, have to be written into the already complex [11]). Other changes include calling **ALSA** relevant functions from the default **ftdi-sio** functions: `audard_probe` from `ftdi_sio_probe`; `audard_probe_fpriv` from `ftdi_sio_port_probe`; `audard_remove` from `ftdi_sio_port_remove`; and `audard _xfer_buf` from `ftdi_process_packet` - which connects the soundcard **ALSA** interface to USB events.

Otherwise, the main **ALSA** functionality is contained in **snd_ftdi_audard.h**, whose development is based on **minivosc.c** [7]. Thus, it contains the same type of **ALSA** related structures, but the structure map (shown on Fig. 3) is slightly more complex than in [7]: the main 'device struct', `audard_device`, contains an array holding references to both the playback and the capture substream; the substreams are encapsulated in `snd_audard_pcm` structures, that hold individual buffer position counters. There are separate `snd_pcm_hardware` and `snd_pcm_ops` variables - yet a single `snd_card_audard_pcm_timer_function` - to handle the playback and capture substreams.

In essence, the **AudioArduino** driver leaves, for the most part, the functionality of **ftdi-sio** as is; with several additions. When `ftdi_sio_probe` runs (i.e., when the ARDUINO is connected to PC via USB), the **ALSA** interface is additionally setup, enumerating the ARDUINO as a soundcard. With this in place, on one hand, the driver keeps the serial interface (such as the creation of the `/dev/ttyUSB0`) file. On the other hand, the driver will also react on 'start' or 'stop' commands from high-level audio software as usual: e.g., on 'start' `_trigger` will run, which will start the timer, and thus the periodic calls to `_timer_function`. The `_timer_function`, then, needs to handle the playback direction by copying the respective part of its `dma_area` to USB - which it does by calling `ftdi_write`. For the capture direction, incoming USB data triggers `ftdi_process_packet`, which additionally calls `audard_xfer_buf`; here USB data is copied to a dynamically sized 'intermediate' buffer, `audard_device->IMRX` – and `_timer_function` will thereafter copy the data from the intermediate buffer to the capture substream's `dma_area`, the next time it runs.

The **AudioArduino** driver additionally exposes CD quality, stereo/16-bit/44.1kHz capability - to allow for direct playback interface with **Audacity** (and most media player software). However, since the microcontroller code expects a sequence of 8-bit values, we must convert the stereo 16-bit stream to a mono 8-bit one - this opens a whole new set of problems related to wrapping, which is illustrated on Fig. 4. By declaring the driver capable of 16-bit stereo, we have not changed the number of substreams (which would correspond to connectors on the soundcard); however, Fig. 4 shows that we would have changed the data format carried in the substream's `dma_area` - the stream is now interleaved: consecutive bytes carry a pattern of left channel's 2 bytes, followed by right channel's 2 bytes. Thus an **ALSA** *frame* (size of analog sample in all channels) is now 4 bytes; and the problem becomes how to represent this **ALSA** frame with a single byte. The approach in the **AudioArduino** driver is to simply extract the most significant byte of the left channel, according to the formula (`C` code):

```
(char) (left16bitsample >> 8 & 0b11111111) ^ 0b10000000
```

However, as Fig. 4 shows, a bigger problem is that the wrapping boundaries (at the size of the chunk handled at each `_timer_function`, and at the size of `dma_area`) can now occur in the *middle* of a frame (and correspondingly, middle of an 8-bit sample) - which is a situation that doesn't occur for 8-bit streams (where each single byte corresponds to one analog sample). To address this, the **AudioArduino** driver employs yet another intermediate buffer (`audard_device->tempbuf8b`). With this in place, the driver will automat-

[5]A crude measurement of jitter spans around 0.26 µs, which is about 52% of the 0.5 µs period for a bit transition at 2 Mbps, see [15]

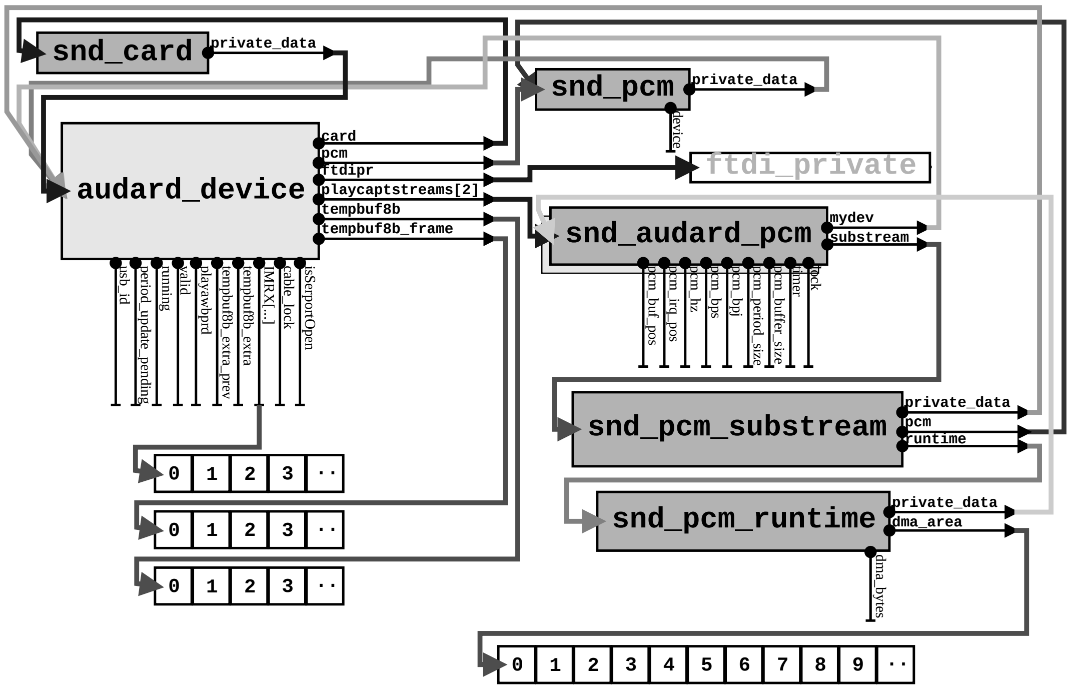

Figure 3: Partial 'structure relationship map' of the AudioArduino driver.

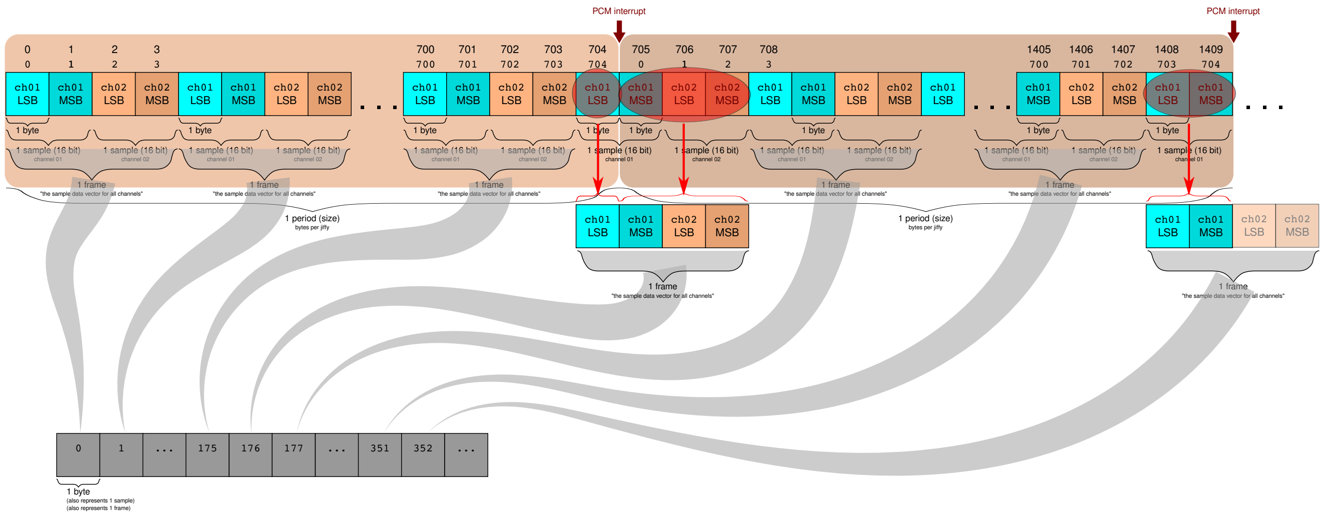

Figure 4: Visualisation of driver's playback buffer boundaries, and CD to mono/8-bit conversion.

ically convert a 16-bit stereo stream from **Audacity** to an 8-bit one, preserving the 44100 Bps rate, before it sends it to USB - and thus, an audio 'digital loopback' can be demonstrated on this driver directly from **Audacity**.

Finally, note that 'DMA' in '**dma_area**' stands for 'Direct Memory Access', which "*allows devices, with the help of the Northbridge, to store and receive data in RAM directly without the intervention of the CPU (and its inherent performance cost)* [8]". Interestingly, in this case: while the transfer of incoming USB data to PC memory (as part of **ftdi-sio**); as well as the transfer of data from **dma_area** to user memory of high-level audio software (as part of the **ALSA** 'middle layer'); likely involves DMA – the transfer of memory that is performed as part of **AudioArduino**'s `_timer_function` definitely *doesn't*; as we use the `memcpy` command to transfer data (which does involve the CPU).

## 8. ANALOG I/O

The **ALSA** driver can be developed in its entirety with the 'digital duplex' ARDUINO code; if thereafter the 'analog I/O' microcontroller code is 'burned' on the ARDUINO - the driver will, effectively, utilize analog input pin 0 as analog input connector, and digital pin 6 as analog output connector. However, both the analog input range, and the output PWM signal, span the voltage range from 0 to 5V - while a typical off-the shelf soundcard typically contains 'line' input and output connectors, as well as 'mic in' and 'speaker out' connectors, which follow a different analog standard. These topics are discussed in more detail in an associated paper, [6].

The use of analog pins on the ARDUINO to read sensors is standard practice, and plenty of examples can be found on the web [2]; thus an arbitrary sensor signal can be captured through high-level audio software at 8-bit, 44.1kHz quality (in the same spirit of [4]). Note that the analog input voltage range, 0-5V, will be represented with the span of 8-bit values from 0 to 255 - which within **Audacity** may be treated as floating point values -1 and 1, respectively.

The use of PWM to deliver an analog audio signal is based on the premise that the highest PWM frequency obtainable from the ARDUINO, 62500 Hz [6], will be sufficient to reproduce a 44100 Hz digital (22.05 kHz analog) audio signal. To a novice, used to analog voltage waveforms, this can be problematic to assess - as the binary nature of PWM makes it seem inherently 'distorted' in the time domain. However, industry insiders are well aware of the practice of using PWM for audio, e.g., in the mobile or automotive industry [10], and often to drive speakers directly [1]. This project demonstrates that as well: upon playback of audio from high-level software, one can simply connect the output pin 6 to a channel on headphone jack, and connect the ground of the headphone jack to ARDUINO's ground - and audible sound would be perceived from the headphones' speaker (but use of a capacitor will result with a louder, clearer sound [3]). Note that there are inherent jitter problems in reproducing HF tones with this technique, while mid-range music can be reproduced with acceptable quality [3, 6].

## 9. CONCLUSIONS

As this paper outlines, development of a soundcard can be a complex and involved issue. The particular approach used here, avoids many electronic engineering issues by choosing the ARDUINO DUEMILLANOVE as soundcard hardware; and avoids deeper involvement with the USB protocol by the specific use of the **ftdi-sio** driver as a basis. In doing that, the overview of the **ALSA** architecture, started in [7], is finalized - as **ALSA** is discussed in its full intended scope: in relation to a given soundcard hardware, and given interface bus. This allows for focus on issues in soundcard implementation that are close to 'first principles', and as such could serve in educational context, as a basic introduction to newcomers to the field - which is the main contribution of this paper and source code.

Beyond (hopefully) furthering the discussion on DIY implementations of PC interfaced digital audio hardware, this project may have a practical impact as well - as there are research projects in the computer audio community and related fields (such as haptics [13]), which use the ARDUINO to capture sensor data; and as such, could benefit from the audio-rate capture quality, and the possibility to leverage the real-time performance of applicable high-level audio software, such as **PureData**.

## 10. FUTURE WORK

The current **AudioArduino** code could, in principle, easily be modified to demonstrate stereo 8-bit performance, or even 16-bit mono (say, by using separate PWM for LSB and MSB, and mixing them in the analog domain). A more involved work would be to port the concept to the reference ARDUINO UNO - as that will require work on the **LUFA** firmware, which doesn't currently support 2 Mbps[15]; on the other hand, the **LUFA** could allow the ARDUINO to be recognized as a 'USB audio' class device, instead of a 'USB serial' one. Finally, as in [7], it would be interesting to see to what degree could **AudioArduino** be ported to the major proprietary PC operating systems.

## 11. ACKNOWLEDGMENTS

The authors would like to thank the Medialogy department at Aalborg University in Copenhagen, for the support of this work as a part of a currently ongoing PhD project.

## 12. REFERENCES

[1] F. T. Agerkvist and L. M. Fenger. Subjective test of class d amplifiers without output filter. In *117th Audio Engineering Society Convention*, 2004.

[2] arduino.cc. Arduino homepage. http://arduino.cc/.

[3] S. Dimitrov. AudioArduino homepage. WWW: http://imi.aau.dk/~sd/phd/index.php?title=AudioArduino.

[4] S. Dimitrov. Extending the soundcard for use with generic DC sensors. In *NIME++ 2010: Proceedings of the International Conference on New Instruments for Musical Expression*, pages 303–308, 2010.

[5] S. Dimitrov and S. Serafin. A simple practical approach to a wireless data acquisition board. In *Proceedings of the 2006 conference on New interfaces for musical expression*, pages 184–187. IRCAM-Centre Pompidou, 2006.

[6] S. Dimitrov and S. Serafin. An analog I/O interface board for Audio Arduino open soundcard system. In *Proceedings of the 2011 Sound and Music Computing Conference*, 2011.

[7] S. Dimitrov and S. Serafin. Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture). In *Not published*, 2011.

[8] U. Drepper. What every programmer should know about memory. 2007. http://people.redhat.com/drepper/cpumemory.pdf.

[9] L. Fried. ladyada.net Wave Shield - Audio Shield for Arduino. WWW: http://www.ladyada.net/make/waveshield, Accessed: 29 Dec, 2010.

[10] M. C. W. Høyerby, M. A. E. Andersen, D. R. Andersen, and L. Petersen. High bandwidth automotive power supply for low-cost pwm audio amplifiers. In *NORPIE2004*, Trondheim, 2004.

[11] G. Kroah-Hartman, B. Ryder, and K. Ober. drivers/usb/serial/ftdi_sio.c. WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob;f=drivers/usb/serial/ftdi_sio.c, Accessed: 29 Dec, 2010.

[12] M. Nawrath. Arduino Realtime Audio Processing. WWW: http://interface.khm.de/index.php/lab/experiments/arduino-realtime-audio-processing/, Accessed: 29 Dec, 2010.

[13] L. Turchet, R. Nordahl, S. Serafin, A. Berrezag, S. Dimitrov, and V. Hayward. *Audio-haptic physically based simulation of walking sounds*, pages 269–273. IEEE Press, 2010.

[14] S. Wilson, M. Gurevich, B. Verplank, and P. Stang. Microcontrollers in music HCI instruction: reflections on our switch to the Atmel AVR platform. In *Proceedings of the 2003 conference on New interfaces for musical expression*, pages 24–29. Citeseer, 2003.

[15] www.arduino.cc. Arduino Forum - Measuring Arduino's FT232 throughput rate ? WWW: http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1281611592/0, Accessed: 29 Dec, 2010.

[16] www.atmel.com. Atmel ATmega48A/48PA/88A/88PA/168A/168PA/328/328P datasheet. WWW: http://www.atmel.com/dyn/resources/prod_documents/doc8271.pdf, Accessed: 29 Dec, 2010.

[17] www.ftdichip.com. FT232R USB UART IC Datasheet Version 2.07. WWW: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf, Accessed: 29 Dec, 2010.

[18] www.ftdichip.com. FTDI Homepage. WWW: http://www.ftdichip.com/, Accessed: 29 Dec, 2010.

[19] www.usb.org. USB.org - Documents [Specifications home]. WWW: http://www.usb.org/developers/docs/, Accessed: 29 Dec, 2010.