sndio – OpenBSD audio & MIDI framework for music and desktop applications

Alexandre Ratchov
alex@caoua.org

AsiaBSDCon 2010

13 march 2010
1 The problem to solve
   - Introduction
   - Purpose of the audio & MIDI subsystem
   - The problem to solve: goal

2 Design choices
   - Kernel vs. user-space implementation
   - Formats & algorithms to support

3 Architecture and implementation
   - Audio server
   - Synchronization & MIDI control

4 Examples

5 Conclusion
Introduction
What is digital audio?

- Sequence of samples at *fixed rate*.
- Played or recorded by the audio interface.
- Full-duplex: $n$-th sample played while $n$-th sample recorded.

**Consequences**

- Clock source (each sample is a clock tick).
- Can be streamed (samples can be buffered).
What is MIDI?

- Slow unidirectional serial link.
- Transmits events to control audio (start, stop, volume).
- Standardized in 1985, used by most professional audio equipment.
- Real-time (i.e., events are processed immediately).

Consequences

- Not a clock source, but can carry clock ticks.
- Usable to control audio (start, stop, volume).
Purpose of the audio & MIDI subsystem

- Fill the gap between the software and the hardware.
  ⇒ Format conversions, resampling.
- Allow multiple programs to use the hardware concurrently.
  ⇒ Mixing, splitting hardware in subdevices.
- Allow multiple programs to cooperate.
  ⇒ Synchronization, communication between programs.
The problem to solve
Conversions & resampling

- Application may not support hardware parameters.
- Two programs may not support common parameters – at least one of them requires conversions.
Splitting hardware in subdevices & mixing

Using a 4-channel card as two independent stereo cards.

**Example**
- Headphones (channels 0,1) for telephony.
- Speakers (channels 2,3) for music.

Remark: *mixing two streams is trivial at this stage.*
To work together, audio programs must:

- Be synchronized.
- Communicate.

⇒ This is the role of audio and MIDI subsystems.
Fault tolerance and correctness

- **Error recovery**
  - Effects of transient errors must be transient
    e.g., a load burst may not cause a program to go out of sync.

- **Isolation**
  - Error in one program should break no other program.

- **Correctness**
  - Complicated architecture leads to bogus implementation.
  - Complicated APIs are misused and lead to broken programs.
  - Complicated tools are misused and lead to broken configurations.
Design considerations
Performance vs. responsiveness

**Performance**

CPU time consumed – e.g., how many CPU cycles to compress a file.

**Responsiveness**

Latency in processing events – e.g., how long it takes to read a block from disk.

For any audio subsystem:

- The CPU usage is negligible.  
  ⇒ Performance is not a concern.
- Delays causes the sound to stutter.  
  ⇒ Responsiveness is of first importance.
User-space vs. kernel: extra latency?

What is latency?

The time between a program produces samples and the time they are played.

- Samples are buffered.
- Buffers are consumed at fixed rate (sample rate).

⇒ The latency is the buffer usage.

No extra latency

Whether buffers are located in kernel or user space doesn’t matter for the latency.
Why do the sound stutter?

The machine is busy, the program (or the audio subsystem) don’t get enough CPU to produce samples to play.

During a load burst:

- User-space components may underrun.
- Kernel components can’t underrun.

⇒ During a load burst, the application underruns, so we’re toast. No matter whether the audio subsystem has user-space components.

Fixing stuttering

- Write programs that don’t block.
- Give enough CPU to programs.
Sample formats choice

- **Integers** (*a.k.a* fixed point numbers) – yes.
  - Any combination of width, signedness, byte order and alignment.
  - Used by most hardware.

- **IEEE floats** used only in the \([-1; 1]\) range – no.
  - Equivalent to integers (fixed range).
  - Trivial to convert from/to integers (FPU required).
  - Mostly used to save development costs.

- **\(\mu\)-law & \(a\)-law** – no.
  - Not usable for audio processing (not linear).
  - Already handled by telephony applications.

- **Encrypted/compressed opaque formats** – no.
  - Not desirable (computers are to process data).
  - Alternatives exist, *i.e.*, the user is not locked.
Architecture and implementation
Overview

- **Audio server**
  - Conversions, resampling, mixing, channel mapping.
  - Subdevices – per-subdevice properties.
  - MIDI controlled per-application volume.
  - MIDI exposed clock source for non-audio applications.
  - MIDI controlled synchronization between applications.

- **MIDI server – software MIDI thru box**
  - “hub” for MIDI data.
  - Software or hardware can be connected to it.

- **Library-based programming interface**
  - Very simple.
  - Mimics kernel APIs (read, write, ...).
  - No need to handle synchronization and error recovery.
Audio server architecture - processing chain

The audio server framework:

- Elementary data processing units with:
  - conversions, resampling (1 input, 1 output),
  - mixing \((N\) inputs, 1 output),
  - channel extraction (1 input, \(N\) outputs),
  - socket I/O, file I/O (either no inputs or no outputs).

- Processing units are interconnected by FIFOs.
- Event-driven framework for non-blocking I/O (no threads).

Server = network of elementary units

Server behavior is determined by:

- the choice of processing units,
- the way they are interconnected.
Audio processing chain – server example

The diagram illustrates an audio processing chain for a server example. It includes the following components:

- **Input (in)**: The starting point for audio input.
- **Decode**: Processes the incoming audio data.
- **Resamp.**: Performs resampling on the decoded audio.
- **Mix**: Combines audio streams from multiple sources.
- **Playback (play)**: Plays the combined audio.
- **Record (record)**: Records audio.
- **Demux**: Distinguishes between different audio streams.
- **Encode**: Encodes the audio for transmission.
- **Resamp.**: Performs resampling on the encoded audio.
- **Output (out)**: The final output of the audio processing chain.

The diagram also shows two socket connections: **socket1** and **socket2**, facilitating communication between different parts of the system.
file1 and file2 are mixed/merged and the result is stored into file3 and file4 (of different formats).
Latency – flow control

Latency

\[
\text{latency} = \frac{\text{buffered}}{\text{rate}} = \frac{\text{written} - \text{played}}{\text{rate}}
\]

⇒ Control the number of samples written.

Minimum theoretical latency:

- Single application:
  2 blocks (1 for the device + 1 for the application).
- Server with independent applications:
  1 extra block to allow 1 application to underrun without disturbing others.

Current implementation allows 3-block latency (i.e., the minimum).
MIDI controlled per-application volume

Why MIDI control?

- Very simple to use and implement.
- Both software and hardware support MIDI.

Current limitations:

- Which MIDI channel corresponds to which application? Mapping should be exposed through the standard mixer interface.
- No MIDI control utility in OpenBSD yet one must use the bulky MIDI hardware or non-OpenBSD software.

...work in progress!
MIDI controlled synchronization

Start process

- MMC “start” message blocks all streams.
- Once all streams are ready, the server starts.

⇒ No need to modify application code.
Sound card clock exposed through MIDI

Why?

Allows MIDI-aware software (or hardware!) to be synchronized to non-MIDI-aware audio programs.

Typical scenario:

- The user manipulates a MIDI sequencer.
- The sequencer controls audio streams.
- The audio server sends feedback to the sequencer.
- The sequencer stays in sync to audio streams.

⇒ Simple programs work together to achieve a complex task.
Examples
Without the audio server and the MIDI thru box, the same task would require a monolithic application.
Integration in OpenBSD

- Single binary – to start the server on the default device:
  
  $ aucat -l

- No configuration file, options are on the command line:
  E.g., to create “spkr” and “hp” subdevices:

  $ aucat -l -c 0:1 -s spkr -c 2:3 -s hp

- Audio player, recorder and off-line conversion utility:

  $ aucat -i file_to_play.wav

- Devices can be either hardware (character devices) or software subdevices (server connections) new naming scheme is required:

  `<type>:<unit>[:.subdevice]`

  E.g., “aucat:0.hp”, “rmidi:5”, “midithru:0”, …
Playing two files simultaneously.

Controlling volume through MIDI.

Synchronizing a MIDI sequencer to record automation.
Playing two files simultaneously.
Controlling volume through MIDI.
Synchronizing a MIDI sequencer to record automation.
Demo: simple tools working together

- Playing two files simultaneously.
- Controlling volume through MIDI.
- Synchronizing a MIDI sequencer to record automation.
Future work

- Less is more — keep it simple!
- Port more code to `sndio`, improve quality of existing code.
- MIDI mixer — use a single mixer framework.
- Record played streams (e.g., record from softsynths).
- Avoid useless data copying — use shared memory.
Conclusion

- Very simple framework (hopefully!).
  - User-space implementation.
  - Single binary, no configuration file.

- Stable and reliable by design.
  - Fast and structured non-blocking framework.
  - Strict latency control (theoretical minimum reached).
  - Synchronization maintained after underruns.

- Problems of desktop application addressed:
  - Conversions, resampling, mixing, channel mapping, volume control.
  - Subdevices – per-subdevice properties.

- Problems of music applications addressed:
  - MIDI controlled synchronization between applications.
  - Software MIDI ports allowing applications to communicate.