ExoTracker

Audio structure is mature and stable in my code base, but not fully documented in this doc. More info at exotracker multiple of 1 chip.

Also see sidenotes and discussion (exotracker program structure) and exotracker pattern data type.

Output subsystem

http://www.portaudio.com/docs/latency.html is seriously outdated. The environment variable trick doesn't work.

I'm getting garbled audio on Linux.

- raspberry pi - How to gracefully recover from -EPIPE errors in ALSA? - Stack Overflow
- Garbled audio on Linux (#7) · Issues · nyanpasu64 / exotracker-cpp · GitLab

Synth subsystem

More information in exotracker multiple of 1 chip. I'm not quite sure which one is newer, and how much is duplicated.

Register writes must talk to chips. Sound comes from channels. And each chip may contain multiple channels, and writing to registers of 1 chip may control 1 channel, all channels, or change the number of channels.

- for N163, the entire chip is a monolithic synth, and sound-genning channels independently is dumb.
- for 5b, there are 3 channels, all which can be blended with noise, and the global shared(?) envelope can affect an arbitrary subset of channels.
- for 2a03, there are 2 independent sound synths, which the global doesn't need to tell apart.

The OverallSynth is only concerned about ChipInstance and register writes. It has no concept of channels.

Idea to port ChipInstance to Rust (no base class inheritance):

- ChipInstance = (RegisterScheduler, RegisterHandler)
- RegisterScheduler.process(&mut dyn RegisterHandler)

Class ownership:

- OverallSynth
    - const clocks_per_tick
    - GetDocument get_document;
    - nes_blip: Blip_Buffer

        - Used to convert samples to clocks. Also written to by most chips (though VRC7 can ignore, and FDS can use its own Blip_Buffer).
        - If multi-console support is added, each will have its own event priority queue, with timestamps in CPU clocks (translated by blip_buffer), or in audio samples.
    - pq: EventQueue (https://gist.github.com/nyanpasu64/abbe09a554aaa7bedc50791f1bcbfd04#file-event_queue-cpp)
    - impl GetDocument for History
    - std::vector<std::unique_ptr<ChipInstance>> _chip_instances = {};
        - class ChipInstance:
            - RegisterWriteQueue _register_writes. Reserve 4096 elements per ChannelID. Only pass by reference, never construct.
            - ~~STATIC_DECL(ChipKind chip_kind())~~ (removed, not in use at the moment)
        - class Apu1Instance : public BaseApu1Instance : public ChipInstance
            - ~~STATIC(ChipKind chip_kind(), ChipKind::Apu1)~~
            - using ChannelID = Apu1ChannelID;
            - Apu1Driver _driver;
                - using ChannelID = Apu1ChannelID;
                - _chip_sequencer: ChipSequencer<ChannelID>
                    - Each chip has its own sequencer. See exotracker sequencer design
                    - EnumMap<ChannelID, ChannelSequencer> _channel_sequencers;
                        - ChannelSequencer behaves the same for every sound chip, since we assume every channel processes events identically. But we may need to replace ChannelSequencer for some chips?
                        - In the case of N163, FDS 2op independent mode, or FM 4op independent mode, the number of active channels in a chip is variable!
                - Apu1PulseDriver _pulse1{0};
                - Apu1PulseDriver _pulse2{1};
                - http://forums.nesdev.com/viewtopic.php?t=231 allows updating 2a03 square pitch registers (for vibrato and bends) without resetting phase. Should I use it pervasively?
            - xgm::NES_APU _apu1;
            - MyBlipSynth _apu1_synth;
        - If I ever add sampler or VST synths, I may call it something else? And store it in a different array?
        - If I ever add multiple consoles with different clock rates, I may replace nes_blip with [ConsoleID]Blip_Buffer and chip_synths with [ConsoleID]vec[int]BaseNesSynth, where int is either NesChipID or SnesChipID etc.

Timing and multi-console support

Currently, timing is in multiple of the NES master clock. Both the callback loop, and each chip's synth, are modeled as state machines triggered by clock-timed "interrupts". Global timing flows from **samples** to **nes_blip.count_clocks(nsamp)** to **clocks**.

I found a good article by byuu on "clock cycle" schedulers: https://byuu.net/design/schedulers . I think it's a good read. Note that Higan's scheduler is tied into Higan's "thread" concept (thread is a C++ wrapper for libco cooperative threads with switchable stacks. byuu calls them cooperative threads, not coroutines.).

~~Idea: Higan (absolute scheduler) uses 64-bit timestamps to ensure all chips operate on a single unified timebase. I don't know what MAME does. I could someday port my code to run blip_buffer with 64-bit absolute timestamps (not in units of clock cycles). 64-bit timestamps may not be possible, or may require code modifications, not sure yet.~~

Master-slave synchronization

in my tracker, if i add multi console support, maybe the other systems will be slaves to the primary system's clock, and the primary system clock will determine when the secondary systems get tracker ticks. the secondary system clock will only be used for audio generation timing.

This resembles MIDI sync. The primary console sends ticks, and the secondary consoles receive ticks.

this will dodge the task of synchronising each console's tick rate, and the issue where ticks could round to one callback on console 1, and the next callback on console 2. it'll be easier to hot-patch into my code later on.

issue is, it's inaccurate and hacky, doesn't match how hardware behaves

but i think it's the simplest solution with the least user-facing issues

Audio callback function (write to registers)

blip_buffer API usage:

- synth.update(0..t)
- blip.mix_samples(in *Amplitude, int nsamp)
- blip.end_frame(t)
- blip.read_samples(out *Amplitude, max_samples) → nsamp_returned

Call graph:

- OverallSynth.synthesize_overall(nsamp, output_buffer) → [nchan][nsamp]Amplitude
    - let document = GetDocument()
    - assert(clk_to_run <= clk_before_tick);
    - EventQueue<SynthEvent> self._timing;
    - while (true) {
        - auto [event_id, prev_to_next] = _timing.next_event();
        - if (prev_to_next > 0)

                - for (auto & chip : _chip_instances)
                    - chip->run_chip_for(prev_to_tick, prev_to_next, _nes_blip, _temp_buffer)
                - _nes_blip.end_frame((blip_nclock_t) prev_to_next);
                - _nes_blip.read_samples(whatever's left of output_buffer)… read the code for details
            - switch (event_id)
                - case SynthEvent::EndOfCallback: return
                - case SynthEvent::Tick:
                    - for chip_index, chip in enumerate(_chip_instances):
                        - chip._register_writes.clear()
                        - chip.driver_tick(document, chip_index)… see [exotracker multiple of 1 chip # Flow of data]
- ChipInstance.run_chip_for(clk_before_tick, clk_to_run, chip_id) { **(outdated, read the code, tbh I don't understand it anymore)**
    - auto & register_writes = self.chip_register_writes[chip_id];
    - auto & chip = *self.chip_synths[chip_id];

    - EventQueue<RunChipBeforeTick> timing;
    - timing.set_timeout(RegWrite, register_writes.peek_mut());
    - timing.set_timeout(PauseCallback, clk_to_run);
    - timing.set_timeout(EndOfTick, clk_before_tick);

    - span uwu = self.temp_buffer;
    - while (true)
        - auto relative = timing.next_event();
        -
        - RunChipBeforeTick id = relative.event_id;
        - switch (id)
            - case RegWrite: {
                - chip.write_reg(register_writes.next());
                - auto nsamp_returned = ->synthesize_chip_clocks(nclk, out uwu);
                    - *If a chip writes to the out buffer, it takes the role of blip_synth, so should have treble filtering but not bass filtering.*
                    - *FDS is clock-locked. It will use an internal blip_buffer (with bass filtering disabled), write to temp_buffer, then low-pass the results. [See sidenotes document.]*
                    - *0CC's VRC7/OPLL emulator only exposes an API where you can request 1 sample at a time. Unlike blip_buffer, you cannot send clock-timed register writes and pull sample-timed audio independently.*
                    - *[Discussed in my extra document. Turns out 0CC's VRC7 emulator is bad and outdated.]*

                - uwu = subslice(uwu, nsamp_returned);
                - break
            - case
        - nsamp_written = uwu.begin() - self.temp_buffer;
    - if nsamp_written
        - nes_blip.mix_samples(self.temp_buffer,)

Chip synth callbacks (global blip buffer, plus per-chip) (outdated)

- nes_blip
- audio callback::synthesize_all_for {
    - should we write audio on every event, or at end of callback?
- }
- SynthesizeAudio/AudioChip/BaseNesSynth {
    - synthesize_chip_clocks(&mut self, nclk, gsl::span<Amplitude> out write_buffer) -> {bool wrote_audio, nsamp_returned} {
        - either use nes_blip or local_blip
        - blip_synth holds a Rust-illegal reference to blip_buffer. don't pass it a blip_buffer, unless I redesign blip_synth to not hold on.
    - }
    - // gets all audio synthesized since last get_audio call
    - get_audio(&mut self) -> Option<audio buffer to add>
- }
- ISSUE! blip buffer will produce different amount of audio compared to FM or sampled.
    - In j0CC, VRC7 does not run in sync with the system clock. Instead, 0CC first asks nes_blip for how many audio samples we have, then runs the VRC7 for that many samples.
    - The VRC7 either runs at the audio sampling rate, ~~or at a sampling rate proportional to the system clock, using linear interpolation to return 1 audio sample at a time~~ (only if quality != 0, but quality is set to 0 because OPLL_new calloc zero-fills the struct).
    - This is definitely not hardware-accurate at all, but "good enough" I guess.
    - blip_buf's output is around 12-ish samples behind the input, since the linear-phase impulses are not causal. 0CC outputs whatever's available after each frame, so the VRC7 will run 12-ish samples (/48000 = 0.25ms) ahead of the other chips.

Sound chip emulation on j0CC

- How does j0CC/0CC make sound chips pausable?
- Does it loop once per clock cycle, or use a scheduler to only run events when needed?
- Does it increment the N163 counter once per clock cycle?
    - (I'm sure it doesn't emulate how volume is probably generated using 15-step PWM tied to the master clock.)

j0CC emulates sound chips in **m_pAPU->Process();** It counts clocks within that function. I very much prefer my design. 0CC is cursed and has 5 separate clock counters.

- Turns out 0CC isn't cursed, it just emulates obscure NES features I've never heard of, without explaining what it's doing. https://wiki.nesdev.com/w/index.php/APU_Frame_Counter The 2a03 has both hardware envelopes and a hardware sequencer.
    - "Some Nintendo arcade boards, even those not directly based on the NES, use the 2A03 CPU as a sound processor. Examples include *Punch-Out!!* and *Donkey Kong 3*. This IRQ allows the CPU to keep time even if no PPU is connected to the bus."
    - "The name "frame counter" might be slightly misleading because the clocks have nothing to do with the video signal." it's very misleading.

CAPU::Process() is kinda event-driven. But it doesn't need to be.

- 2a03 sequencer runs at 240Hz. I may not use it.
- m_iFrameClock and EndFrame() should be handled by the top-level audio callback's event loop (EventID::Tick), not the sound generator.

The individual sound chips it calls ((Chip: CSoundChip*)->virtual Process(Time)):

- 2a03:
    - APU1: (min of 2 periods).clamp([7..]).clamp([..time])
    - APU2: (min of tri/noise/dpcm periods).clamp([7..]).clamp([..time])
    - Why do these clamps exist? Replacing 7 with 10'000 ~~doesn't change the audible output~~ (both bass=$7FF and B8=$00D play fine).
        - I ran wave export and they're very slightly different. I get much bigger deviations when the dpcm kicks in, so **I'm pretty sure "run APU for only 1 period" affects nonlinear mixing…**
        - **Maybe nonlinear mixing is only recomputed once per single-channel Process() call.**
- n163 "new mixer" is an ad-hoc and unclear priority-queue state machine.
- vrc7 does not run on-the-fly.

j0CC audio processing loop

- foreach channel:
    - m_pChannels[i]->virtual RefreshChannel(); *(writes to chip registers)*
    - m_pChannels[i]->static FinishTick(); *(nearly useless)*
        - CChannelHandler::RefreshChannel()
    - if channel's owner chip enabled:
        - SoundGen.add_cycles(i32 count)
            - called from player thread
            - let count = min(count, SoundGen.m_iUpdateCycles - SoundGen.m_iConsumedCycles)
            - SoundGen.m_iConsumedCycles += count

                - APU.add_time(count)
            - }
        - APU.process()
- APU.add_time(SoundGen.m_iUpdateCycles - SoundGen.m_iConsumedCycles)
- APU.process()