

Apple II audio from the ground up

Kris Kennaway
KansasFest 2022

About me

Apple II user since the 1980s

(~2019) became interested in Apple II audio/video

- (KFest 2019) Streaming video + audio over ethernet (]][-Vision)
- (2020) Nox Archaist splash screen (simultaneous audio + animation)
- (KFest 2021) Streaming video + audio from CFFA3000

Never felt I *really* understood how audio worked

(in fact there was a lot I had misunderstood!)

Start at the beginning

APPLE II[®]

REFERENCE MANUAL



 apple computer inc.

THE SPEAKER

Inside the Apple's case, on the left side under the keyboard, is a small 8 ohm speaker. It is connected to the internal electronics of the Apple so that a program can cause it to make various sounds.

The speaker is controlled by a soft switch. The switch can put the paper cone of the speaker in two positions: "in" and "out". This soft switch is not like the soft switches controlling the various video modes, but is instead a *toggle* switch. Each time a program references the memory address associated with the speaker switch, the speaker will change state: change from "in" to "out" or vice-versa. Each time the state is changed, the speaker produces a tiny "click". By referencing the address of the speaker switch frequently and continuously, a program can generate a steady tone from the speaker.

The soft switch for the speaker is associated with memory location number 49200. Any reference to this address (or the equivalent addresses -16336 or hexadecimal `$C030`) will cause the speaker to emit a click.

That's all there is.

Well OK, let's dig deeper

Lies!

A program can “reference” the address of the special location for the speaker by performing a “read” or “write” operation to that address. The data which are read or written are irrelevant, as it is the *address* which throws the switch. Note that a “write” operation on the Apple’s 6502 microprocessor actually performs a “read” before the “write”, so that if you use a “write” operation to flip any soft switch, you will actually throw that switch *twice*. For toggle-type soft switches, such as the speaker switch, this means that a “write” operation to the special location controlling the switch will leave the switch in the same state it was in before the operation was performed.

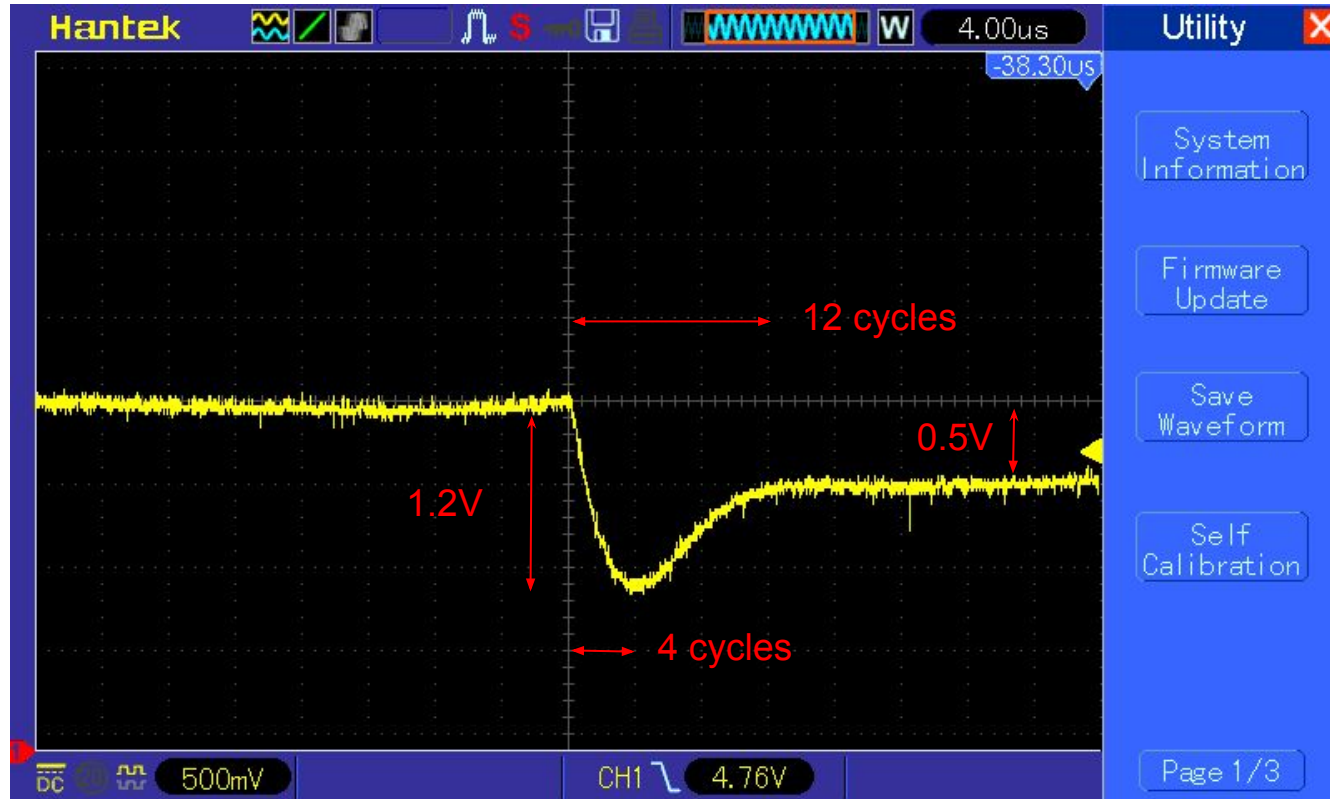
This isn’t true! Only some 65(c)02 opcodes do this

The POKE command in BASIC does have this behaviour...

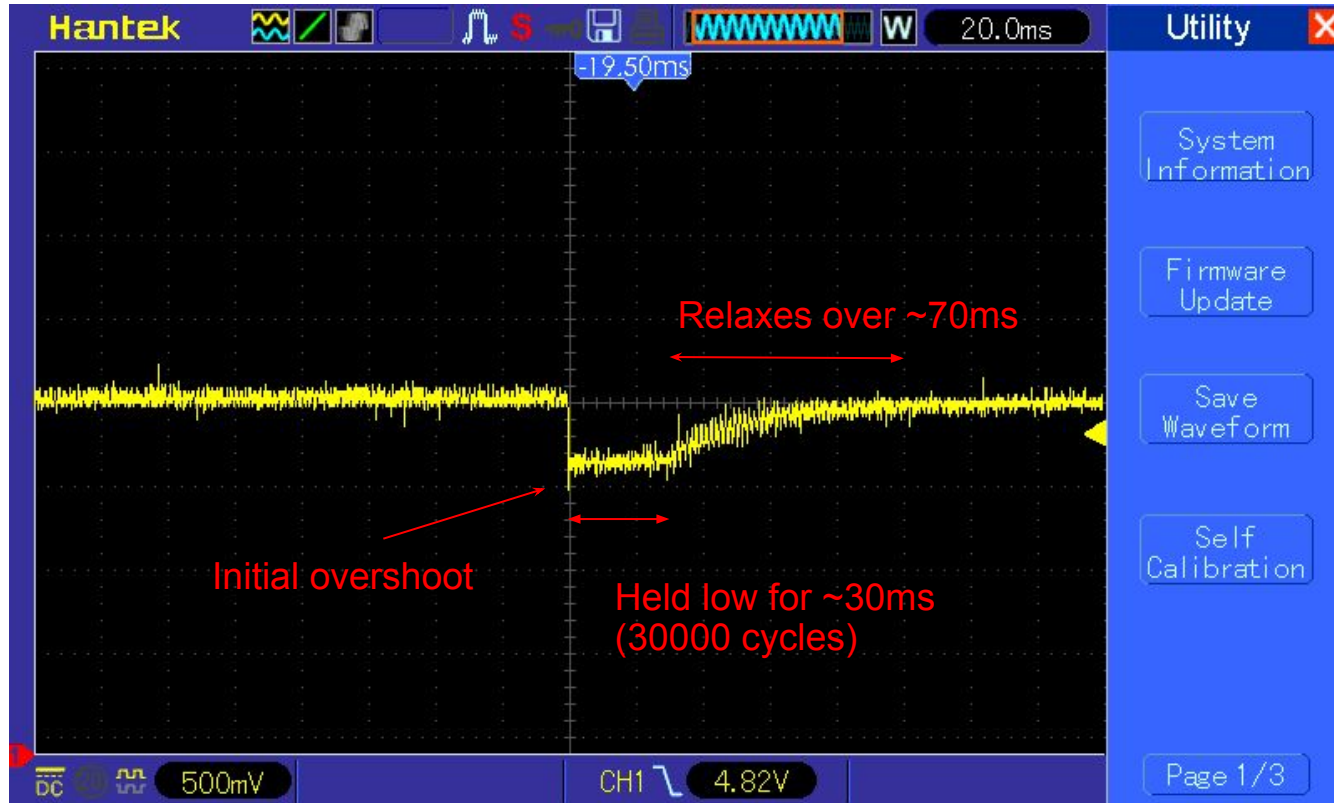
...though only on a 6502 based machine, not 65c02 (e.g. enhanced //e and above)

What happens when we access \$C030?

Voltage changes when toggling speaker



Voltage changes over longer timescale



Don't leave me alone

- If we toggle the speaker low and leave it alone for >30ms, it relaxes back to the high state (after 100ms)
- Next access tries to pull voltage high
- ...but it's already there, so no speaker movement

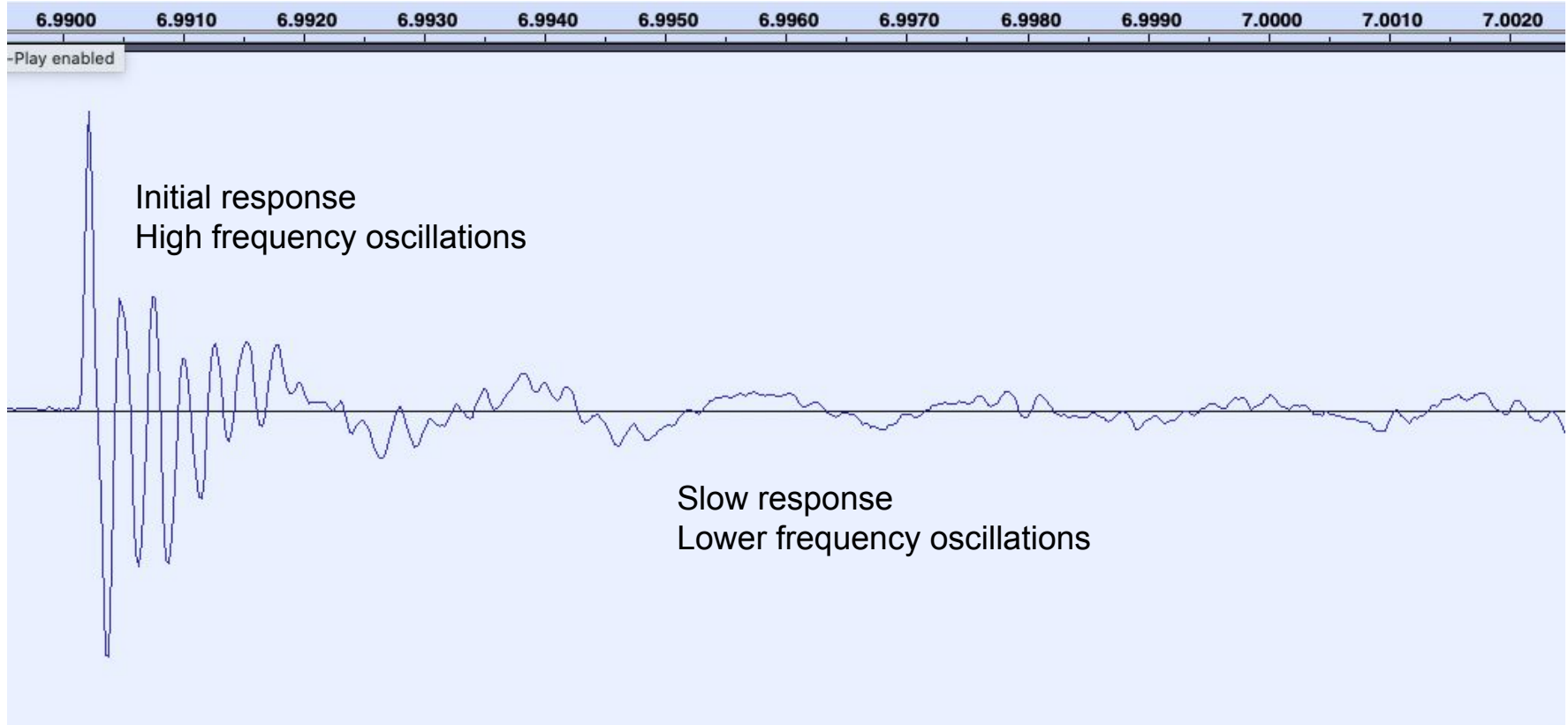
```
]?PEEK(-16336) ← Clicks
55
]?PEEK(-16336) ← Does not click
183
]
```

Simplifications

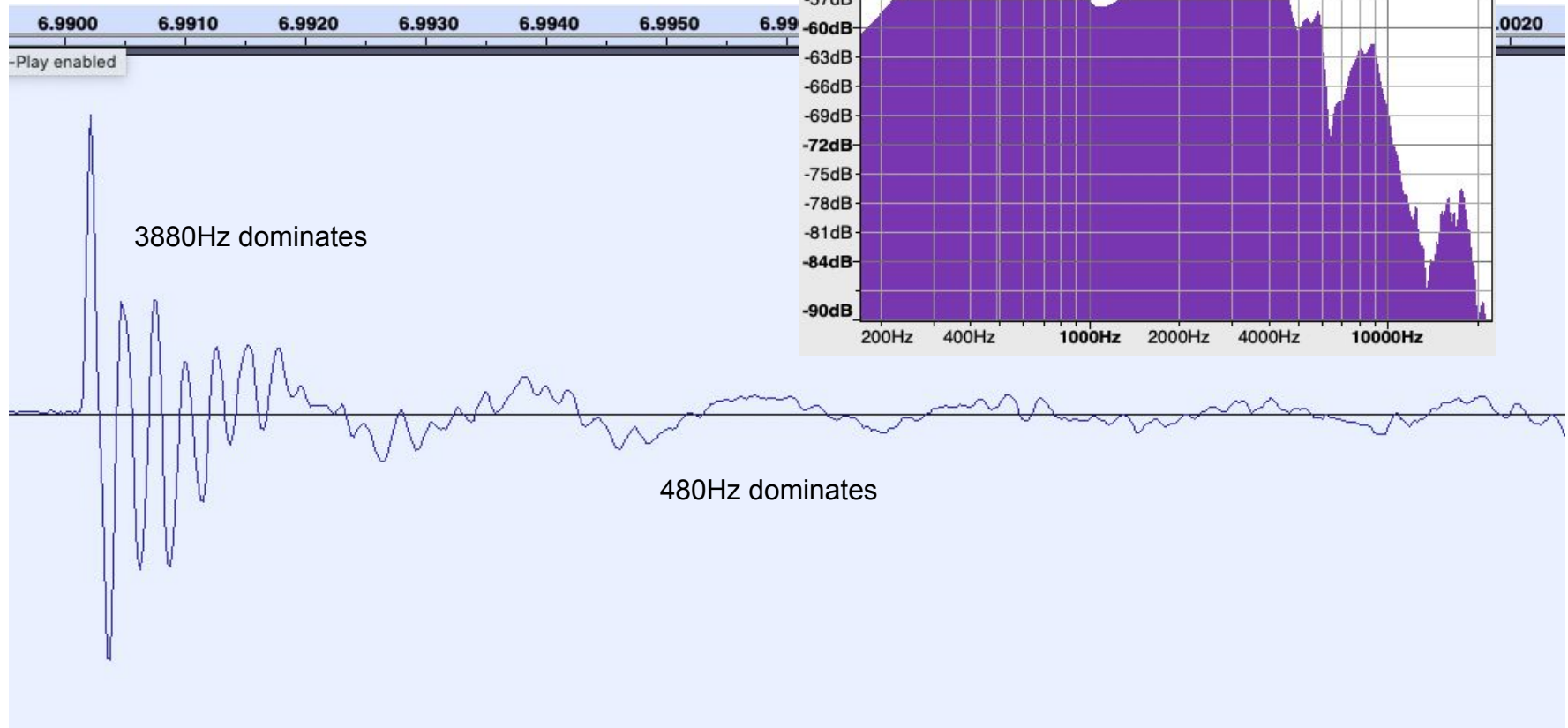
- from now on treat applied speaker voltage as a **square wave**
- ...ignore the initial overshoot
 - may introduce some error, revisit
- ...ignore the long-time decay
 - because we won't leave the speaker alone long enough for this to matter



A speaker click



A speaker click



Speaker response

Applied voltage (~square wave) → speaker response (oscillatory)

Speaker rings like a bell when we kick it

If we can model this impulse response mathematically, we can understand how speaker will behave in response to **arbitrary** stimulus

Simulating the speaker

- interested in speaker response at short time scales
 - ignore 480Hz component
- a damped harmonic oscillator
 - spring subject to damping force
 - RLC circuit
- can simulate in discrete time, e.g. clock cycle by cycle
 - See e.g. *Signal Processing in C* (C. Reid, 1992)
 - complicated maths, simple result
 - should be possible to implement in emulators

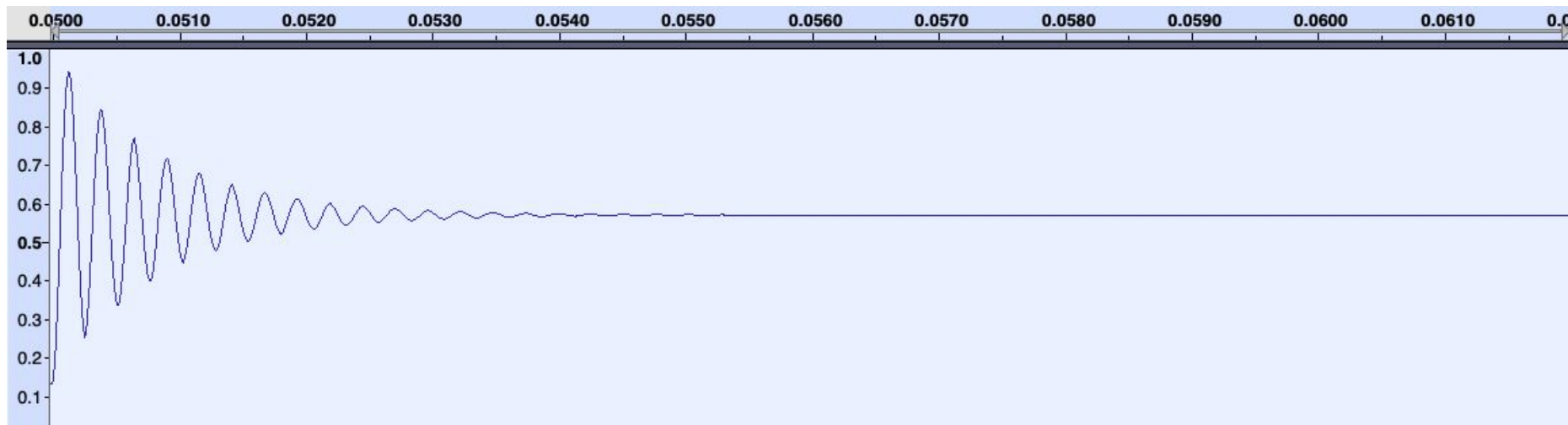
Simulating the speaker

- in order to know the position y_n of the speaker at clock cycle n we just need the applied voltage V_{n-1} and the speaker position at cycles $n-1$, $n-2$

$$y_n = c_1 y_{n-1} - c_2 y_{n-2} + V_{n-1}$$

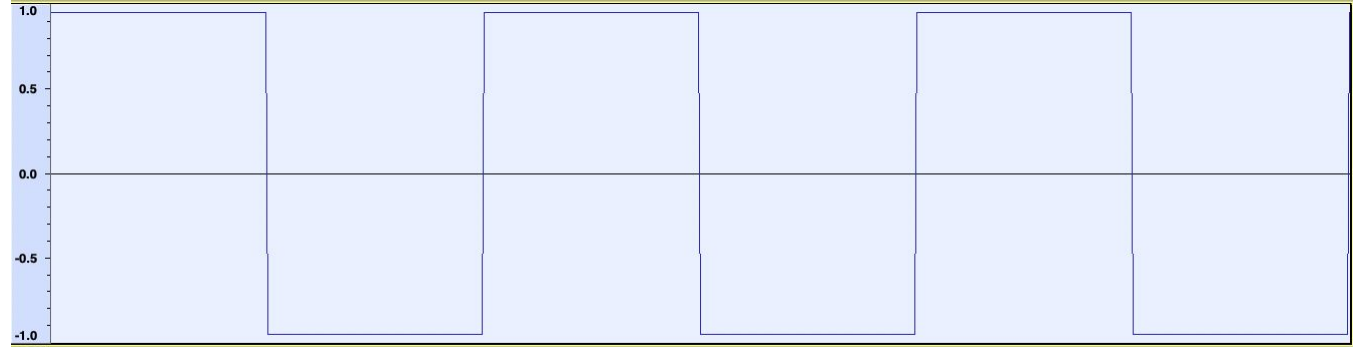
- where c_1 , c_2 are constants that depend on the parameters of the speaker
 - resonant frequency, envelope decay
- we can obtain these by fitting against the recorded click waveform
 - or frequency spectrum

Our simulated 3880Hz click

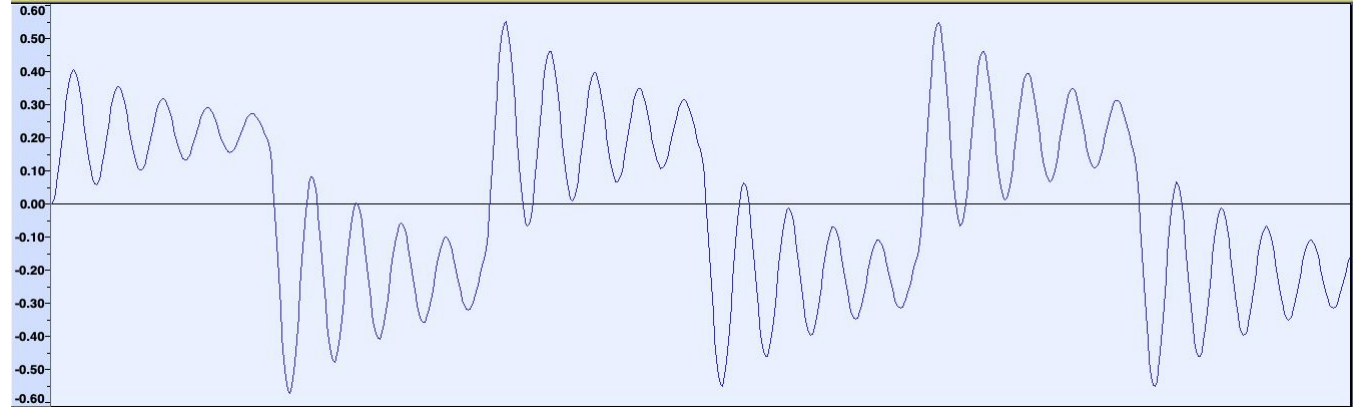


Speaker acts like a filter

Square wave speaker
input



Modulated speaker
output



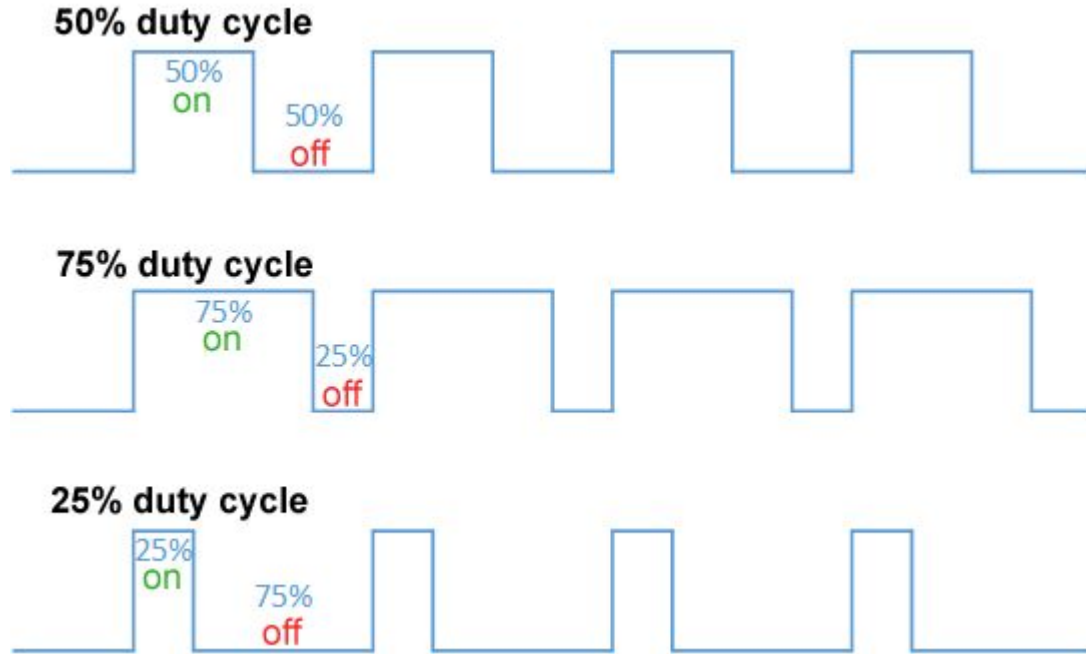
How about arbitrary waveforms?

- we don't just want simple square wave tones
- but how can we do this if we can only control whether the applied speaker voltage is high or low?
- Most existing Apple II audio waveform generation uses Pulse-Width Modulation
 - Dr. Cat, late 80s, unreleased
 - SoftDAC (Scott Alfter, 1990)
 - DAC522, RTSynth, ... (Michael Mahon)
 - J[-Vision (ethernet/CFFA3k video streaming), Nox Archaist splash screen (Kris Kennaway)
 - A2Stream (Oliver Schmidt)
 - ...

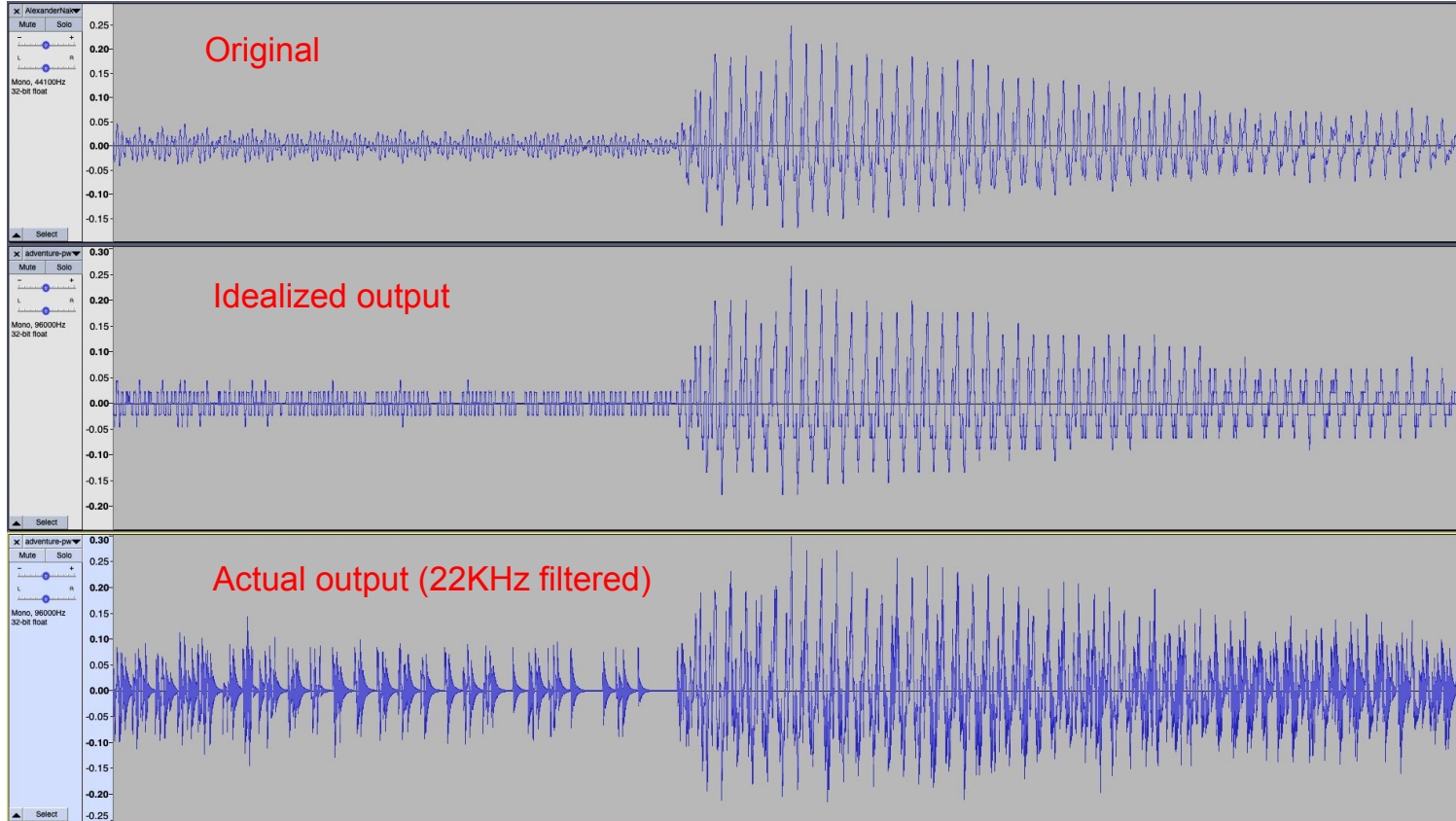
Pulse Width Modulation

- we can't generate a **constant** voltage other than high or low
- but we can generate an **average** voltage over some interval
- ...by holding the speaker high for some fraction **a** of the interval and then low for the remainder **b**
- i.e. we are using the varying **width** of this speaker **pulse** to **modulate** the audio
- speaker will oscillate up and down, but on average its position is given by **$a/(a+b)$**
- This fraction is called the duty cycle

Pulse width modulation - duty cycle





Pulse width modulation - ideal vs actual response



Pulse Width Modulation - limitations

- effectively turns 1-bit speaker control (2 positions) @ 1Mhz into ~5 bits of control (~40 positions) @ 22Khz
- Can produce quite good audio quality. But...
 - spectrum is dominated by 22KHz “carrier wave” - chosen to be inaudible
 - oscillation of speaker around average position → **audio distortion**
 - we can only produce a small number of average speaker positions → **limited audio detail**
- Louder waveforms are usually not too bad
 - higher signal:noise
- Limitations are especially noticeable for quiet audio
 - waveforms close to 0 are dominated by noise

Pulse Width Modulation in action

- Let's hear this in action
- We can use our speaker model to simulate the sound of PWM audio
- Picked a quiet track that shows the limitations
- Keep in mind this is showing the worst case behaviour for PWM
 - also this audio file is much louder than the sound produced by Apple II speaker
- Original audio 
- (Simulated) PWM 

Music: Adventure by [Alexander Nakarada](#)

Licensed under [Creative Commons BY Attribution 4.0 License](#)

Can we do better?

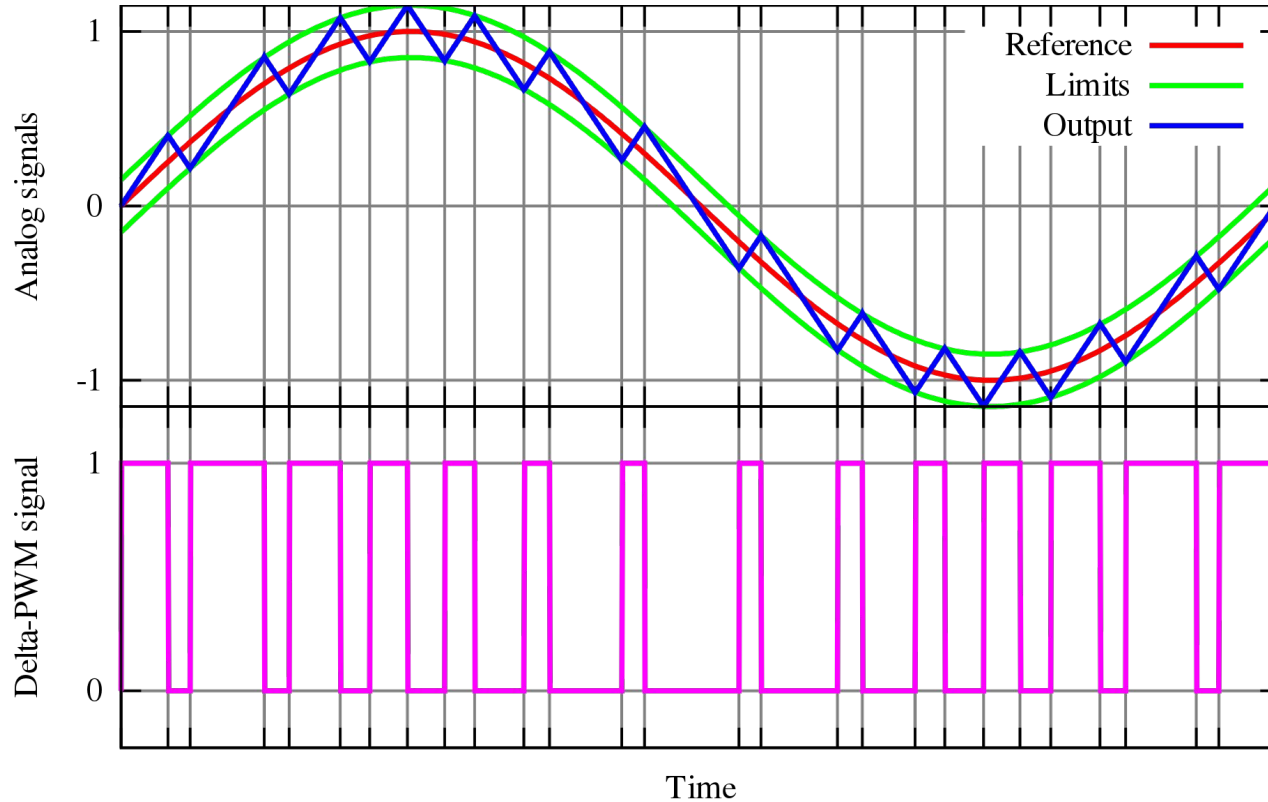
Applying our speaker model

- We know from our model how the speaker will respond when we kick it (invert applied voltage)
- If we have fine enough timing control, we can time the kicks to direct the speaker where we want it to go
- i.e. to cause it to trace out an arbitrary waveform **precisely**
 - (within limits of our model)

Delta modulation

- This is called delta modulation
- We monitor the audio signal produced by the filter (simulated speaker), and when it drifts too far from the desired value we kick it in the other direction
- Speaker constantly jiggled back and forth to stay centered on the desired waveform position

Delta modulation



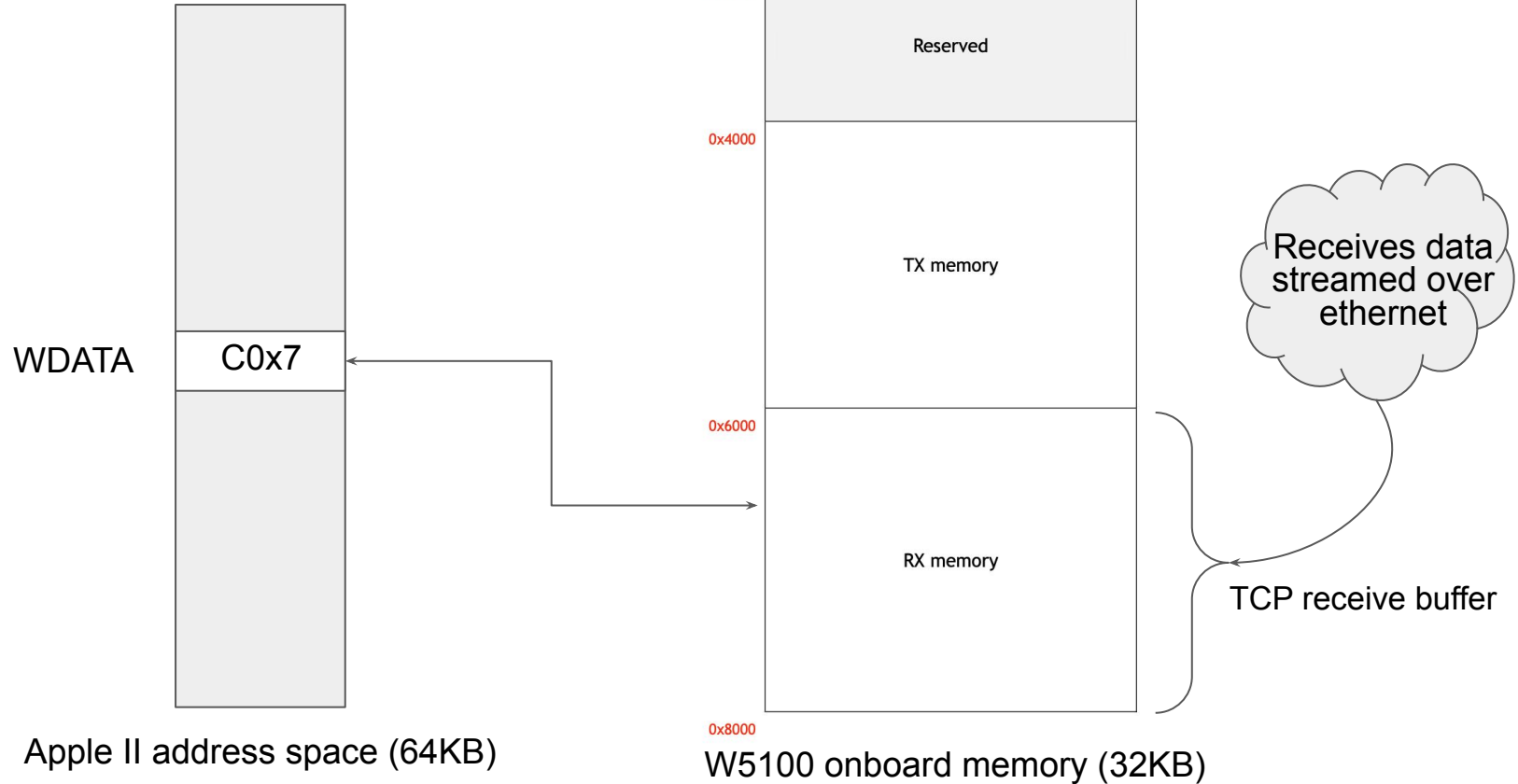
Delta modulation vs standard PWM

- Delta modulation is a form of PWM
 - we're still modulating the pulse width of a square wave signal
- Differences from standard PWM:
 - doesn't modulate at a fixed frequency
 - needs high modulation frequency to control white noise ("quantization error") due to tracking back and forth either side of target waveform
 - not "fire and forget" - needs understanding of speaker output to determine when to modulate PWM signal

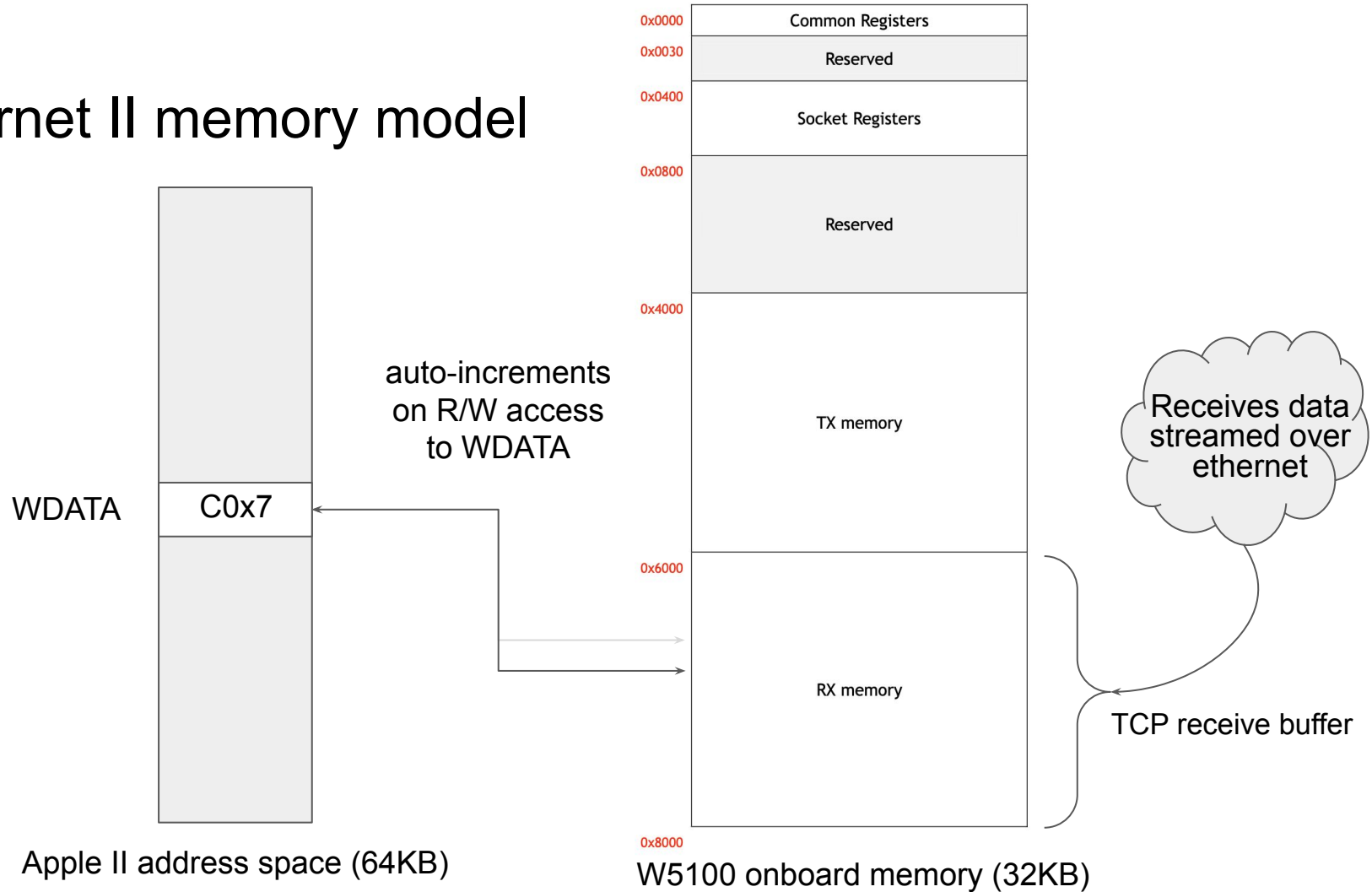
Delta modulation on the Apple II

- What is possible to implement on the Apple II?
 - Previous work: “BTC player” (Oliver Schmidt, 2018)
 - in-memory playback, encoder uses a simpler speaker model
- Know from past work ([II-Vision](#)) that streaming audio using Uthernet II has higher performance than playback from memory
- Let’s look at the core audio loop - what’s the tightest we can get?

Uthernet II memory model



Uthernet II memory model



First approach

audio_operation:

STA \$C030 ; toggle speaker [4 cycles]

; maybe some NOPs to generate different cycle timings

LDA WDATA ; read low byte of next address to jump to [4 cycles]

STA @0+1 ; self-modify to jump to it [4 cycles]

@0:

JMP \$20xx ; jump to next operation [3 cycles]

; 4 + 4 + 4 + 3 = 15 cycles, toggles speaker at 68KHz

Not bad, but can we do better?

- Deeper look at Uthernet II memory map
- Onboard hardware (W5100) exposed to Apple II via 4 soft switches

C0x4 (x = slot# + 8)	W5100 mode register	Normally #\$03
C0x5	W5100 address high	Pointer into onboard W5100 memory address space
C0x6	W5100 address low	
C0x7	Data port (WDATA)	read/write results in R/W to address specified by \$(C0x5). Normally auto-increments C0x5/6 (in mode #\$03)

A trick!

- The Uthernet II hardware doesn't decode all of the address lines
- soft switches at \$C0x4...\$C0x7 are also mapped at \$C0x8...\$C0xB
- Read as a 16-bit address, the high byte of \$C0x7 (WDATA) is given by \$C0x8 (WMODE)
- ...which is set to #\$03
- JMP (WDATA) will jump to an address in page 3
- ...with page offset chosen by the next byte in the W5100's onboard memory
- we can chain player operations directly from the TCP socket buffer!
 - no need for self modifying code

Improved approach

audio_operation:

```
    STA $C030; toggle speaker
```

```
    JMP (WDATA) ; jump to $03xx with low byte taken from socket buffer
```

; 4 + 6 = 10 cycles on 65c02 (9 on 6502)

; toggles at 102Khz (113KHz on 6502)

- audio playback is actually **10% faster on 6502** than 65c02!
 - 65c02 fixed a 6502 bug when JMP (indirect) crosses a page boundary, at the cost of an extra cycle (6 vs 5)
 - possibly the most anyone's ever used JMP (indirect)

A 16-byte audio player

; even cycle intervals

tick_00:

 NOP ; 2 cycles

tick_01:

 NOP ; 2 cycles

tick_02:

 STA \$C030 ; 4 cycles

tick_05:

 JMP (WDATA) ; 6 cycles

; odd cycle intervals

tick_08:

 NOP ; 2 cycles

tick_09:

 NOP ; 2 cycles

tick_0a:

 ; When X=#\$31, accesses

 ; \$C030 on cycle 5

 STA \$BFFF,X ; 5 cycles

 JMP (WDATA) ; 6 cycles

Chaining operations

By chaining together these operations we can toggle the speaker at any cycle interval ≥ 10 . For example:

```
tick_02: STA $C030 ; toggle on cycle 4
```

```
tick_05: JMP (WDATA) ; WDATA = #$05 → tick_05 [6 cycles]
```

```
tick_05: JMP (WDATA) ; WDATA = #$01 → tick_01 [6 cycles]
```

```
tick_01: NOP ; 2 cycles
```

```
    STA $C030 ; toggles on cycle 4
```

```
; 6 + 6 + 2 + 4 = 18 cycles between toggles
```


Core playback loop

- Easily fits in page 3 (16 bytes)
- Play audio by placing page 3 offsets in TCP stream to chain together 7 audio operations
- Can toggle speaker with 1MHz precision
 - pick exactly which CPU cycle to toggle on
- ...at anything up to 100Khz frequency
 - toggle no more often than every 10 cycles
- Not the whole story
 - we still have to work out what happens when we finish stepping through the socket buffer memory
- But let's look at how we can use this audio player for delta modulation

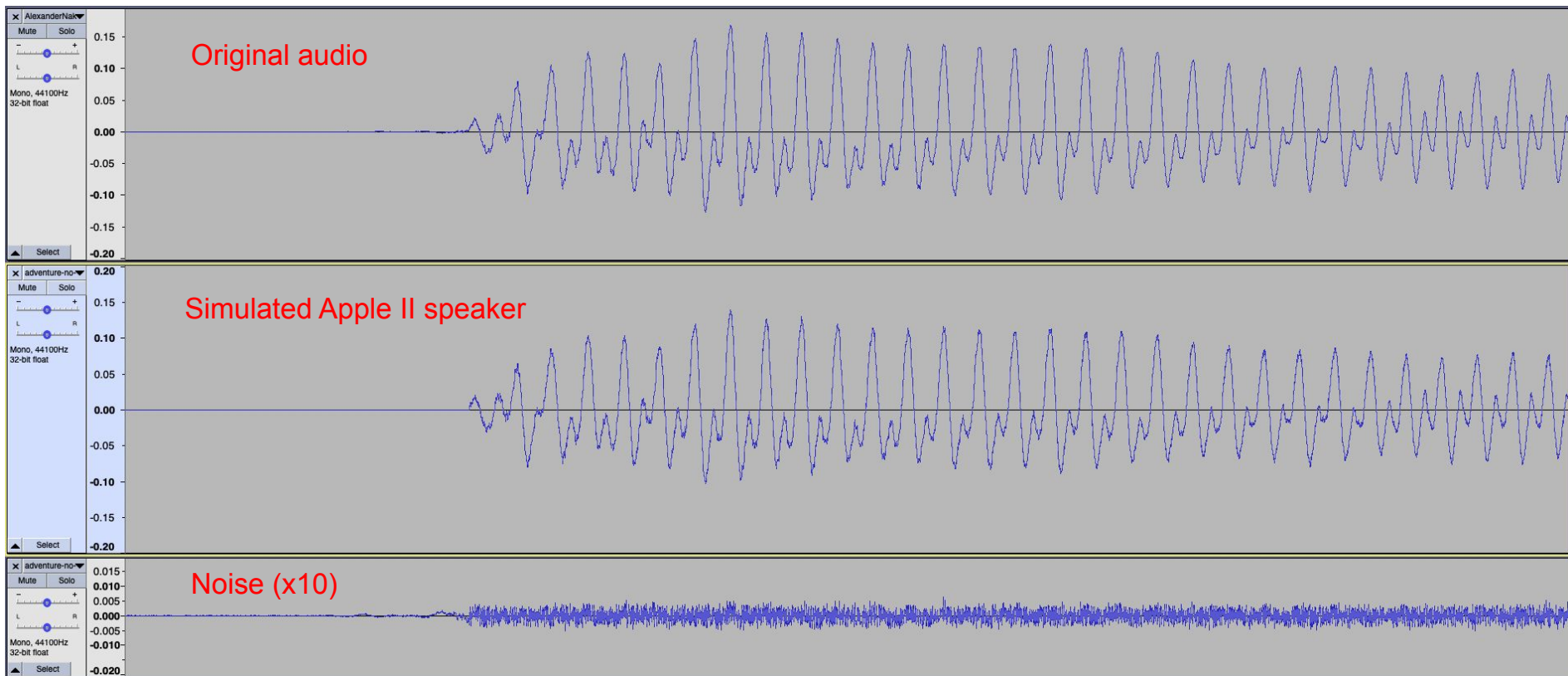
Encoding the audio

- Given audio input file (e.g. .wav, .mp3)
- Upscale it from input sample rate (usually 44100Hz) to ~1MHz CPU clock rate
 - actually 1020484Hz on NTSC, 1015657Hz on PAL
- Each 1MHz audio sample gives the desired speaker position at successive clock cycles
- At each step, we have a choice of 7 possible player operations we could do
- Simulate them all and pick the best one
 - i.e. the one that tracks the target waveform most closely
- Record it and play the simulated audio forward to the end of that operation

Encoding audio

- Can do better: look several steps into the future (e.g. 30 cycles) and simulate all possible combinations
 - allows adapting better to upcoming waveform changes
- Typically toggles the speaker at around 100Khz
- Here's what it sounds like (simulated) 

Delta modulated waveforms (simulated)



OK so we're done, right?

- Not so fast, we only have 8KB of data to work with in the ethernet socket buffer
- ~ 8ms of playback @ 100Khz
- We need to periodically drop out of continuous audio playback to tell the W5100 to manage the socket buffer
- We know what the speaker does when left to its own devices: it clicks
- So we have to keep modulating the speaker while doing this
- Insert a special operation every 2KB in the audio stream that causes us to jump to a management path


TCP buffer management

- buffer management takes 70 cycles, ignoring the speaker
- we can't control speaker closely during this period - too many combinations
- do the best we can
- repeating duty cycle (a, b) but with variable modulation frequency
 - $\geq 22\text{KHz}$, so not audible
- one variant for all possible speaker duty cycles (a,b), $a+b \leq 46$
 - subject to some additional constraints
- a lot of tedious code to write by hand
- wrote a program to generate it

TCP buffer management

- end up with 209 variants for end-of-frame processing
 - fairly complicated 3-stage jump process, 19KB of (generated) assembly
 - effectively gives ~ 7.7 bits of speaker control, though only a fraction are commonly used
- let audio optimizer pick the best one each time
- 0.7% of playback time is spent in end-of-frame (~ 140 cycles out of ~ 20480)
- some quality degradation but minimal
- manifests as a $\sim 50\text{Hz}$ background crackle from slightly losing control of the speaker tracking during end of frame processing

Putting it all together

- Review:
 - we worked out how to **simulate** the Apple II speaker on a cycle by cycle cadence
 - we built an ethernet audio player that can **control** the speaker with cycle level precision at up to 100Khz
 - i.e. toggling the speaker every ≥ 10 cycles
 - we built an audio encoder that works out how to drive the player to make the (simulated) speaker precisely **trace out** an arbitrary audio waveform
- Let's hear it in action
- This is an audio recording of the built-in speaker on an Apple IIe 
 - unprocessed except to normalize volume

What's next?

- <https://github.com/KrisKennaway/ii-sound/>
- not 100% happy with end of frame processing
 - ~50Hz crackle due to tracking errors during end of frame
 - should be possible to tune further
- improve speaker modeling
 - applied voltage is also a damped harmonic oscillator
 - speaker response has two coupled oscillators (~3880Hz, ~480Hz)
- scale down to in-memory playback
 - quality and playback duration will be lower
 - but maybe still competitive with PWM?
- implement speaker modeling in emulators?

Questions?

Bonus material

Delta modulation with an RC circuit

Delta modulation with an RC circuit

- [Prior work](#) in the Apple II community (Oliver Schmidt, 2018) implementing “Binary Time Constant” audio player
- really just delta modulation but with a simpler model for speaker response
- models speaker as an RC circuit (first order DE) instead of RLC (second order DE)
- response to applied voltage is exponential, not oscillatory
- I started the [work](#) this talk is based on back in 2020 using this method, but couldn't get quality high enough
- gives good results for continuous sample playback, but plagued by clicks during end of frame processing

What's going on?

- it “works” because as long as we’re toggling the speaker often enough (~ 10 cycles), RLC speaker response (oscillation) is approximately the same as RC speaker response (exponential)
- approximation breaks down over longer time scales, e.g. during EOF processing (~ 140 cycles)
- because we’re using the “wrong” model for predicting the speaker response, we get audio clicks when playing on the physical speaker

W5100 buffer management

W5100 buffer management

- Minimal end-of-frame buffer management requires about 70 cycles
 - tell the card to ACK the 2KB we've just read from the socket
 - behind the scenes the W5100 will fill up another 2KB ready for us to read later
 - double check there is at least another 2KB ready to go
- How can we control the speaker during this work?
 - can't just pick a fixed cadence, e.g. every 10 cycles cycles
 - we know from PWM this will cause speaker to drop to the midpoint (0-position)
 - might be far from desired waveform - introduces audio errors
 - worse, we also know what happens if we have a sudden sustained voltage change: the speaker clicks

PWM with a twist

- Best we can do is generate lots of variations of the same basic buffer management but with different (a, b) duty cycles
- PWM with a variable duty period, still using delta modulation to select
- Limit $a+b \leq 46$ to keep carrier frequency $\geq 22\text{KHz}$
- Let the audio encoder pick the best one
- e.g. (4, 21) \rightarrow 25 cycle period, 40.8KHz
- 209 variations, effectively gives ~ 7.7 bits of speaker control
- ~ 140 cycles out of ~ 20480 spent in buffer management = 0.7% of playback time
- minimal quality degradation

- other big constraint: we have to start the process by jumping to a location in page 3
- but we can't directly enumerate all 208 buffer management entry points in page 3 (6 bytes each)
- ended up being a fairly complex 3-stage jump process to vector from page 3 to the actual buffer management variants
- would be a nightmare to code this by hand; wrote code to generate it
- 13K lines of assembly (19KB) vs 16 bytes for core audio loop!