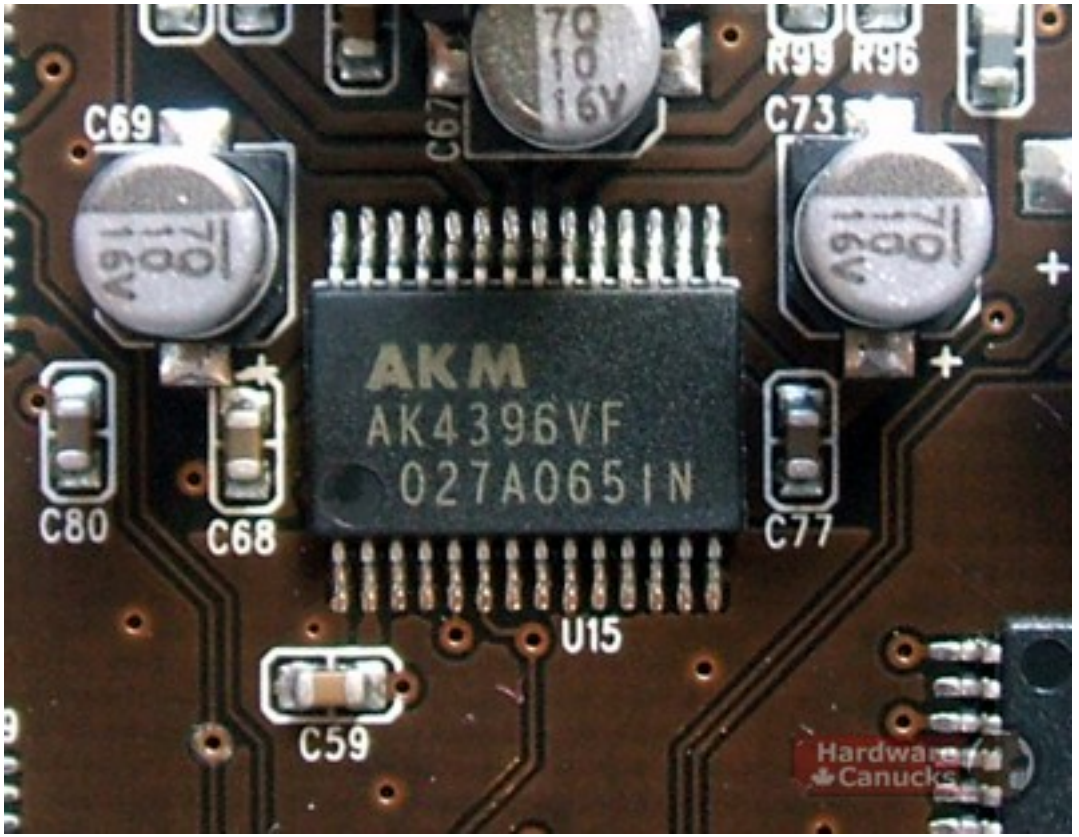
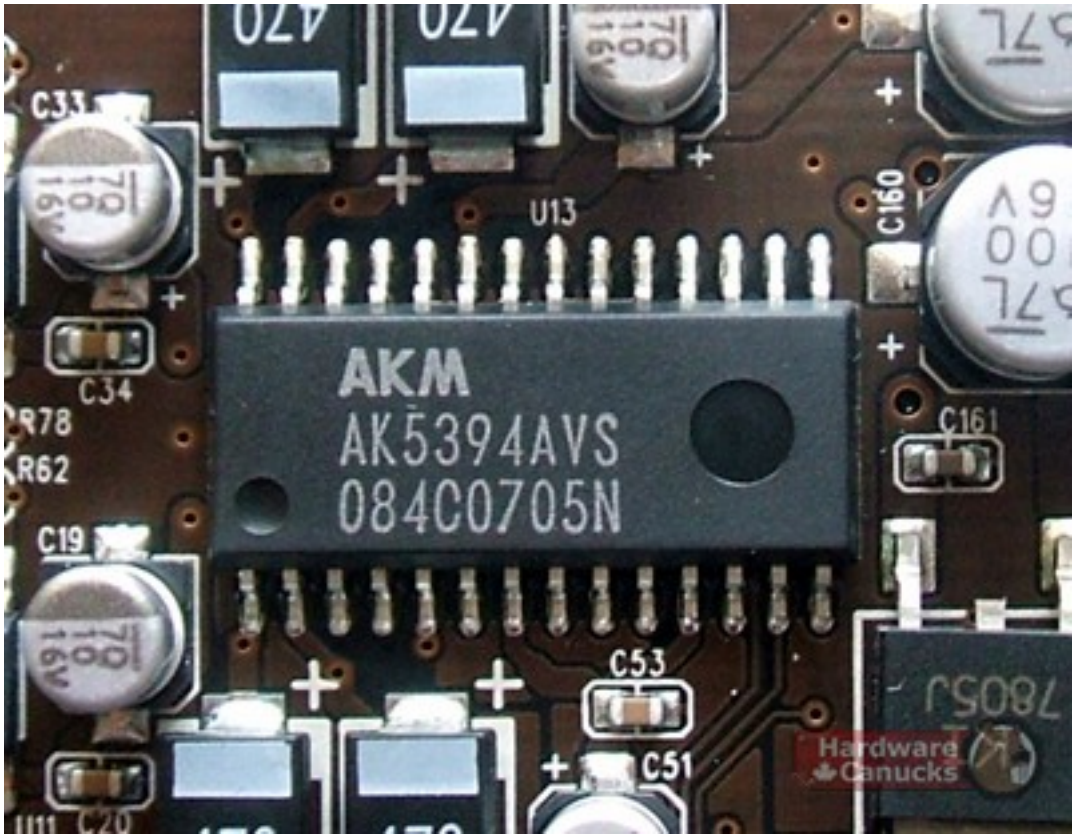
Audio I/O

Getting Audio In, Out and Around
Your (or my) Computer

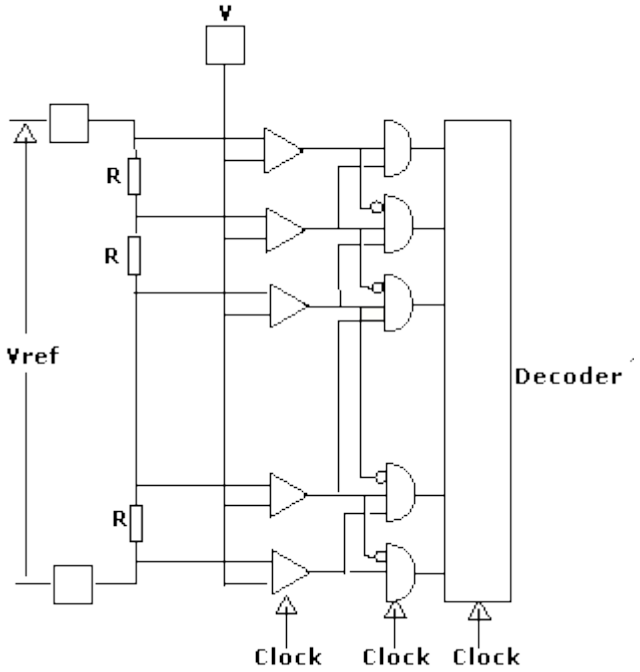
What is this?


C69
C67
R99
R96
C73
AKM
AK4396VF
027A0651N
C80
C68
C77
U15
C59

Or this ....


C33
U13
C160
AKM
AK5394AVS
084C0705N
C34
R78
R62
C161
C19
C53
C51
Hardware Canucks

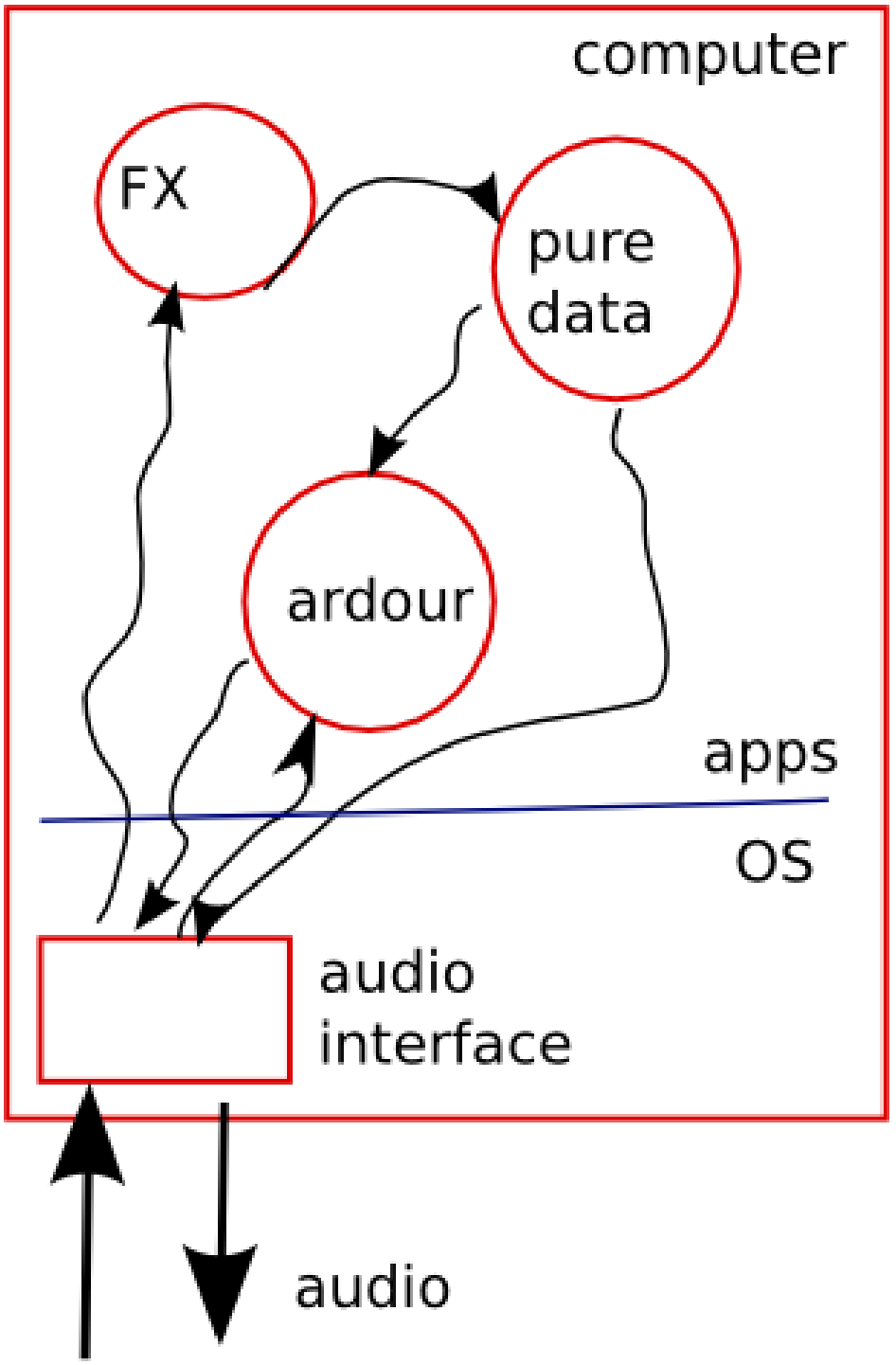
Or even this ...

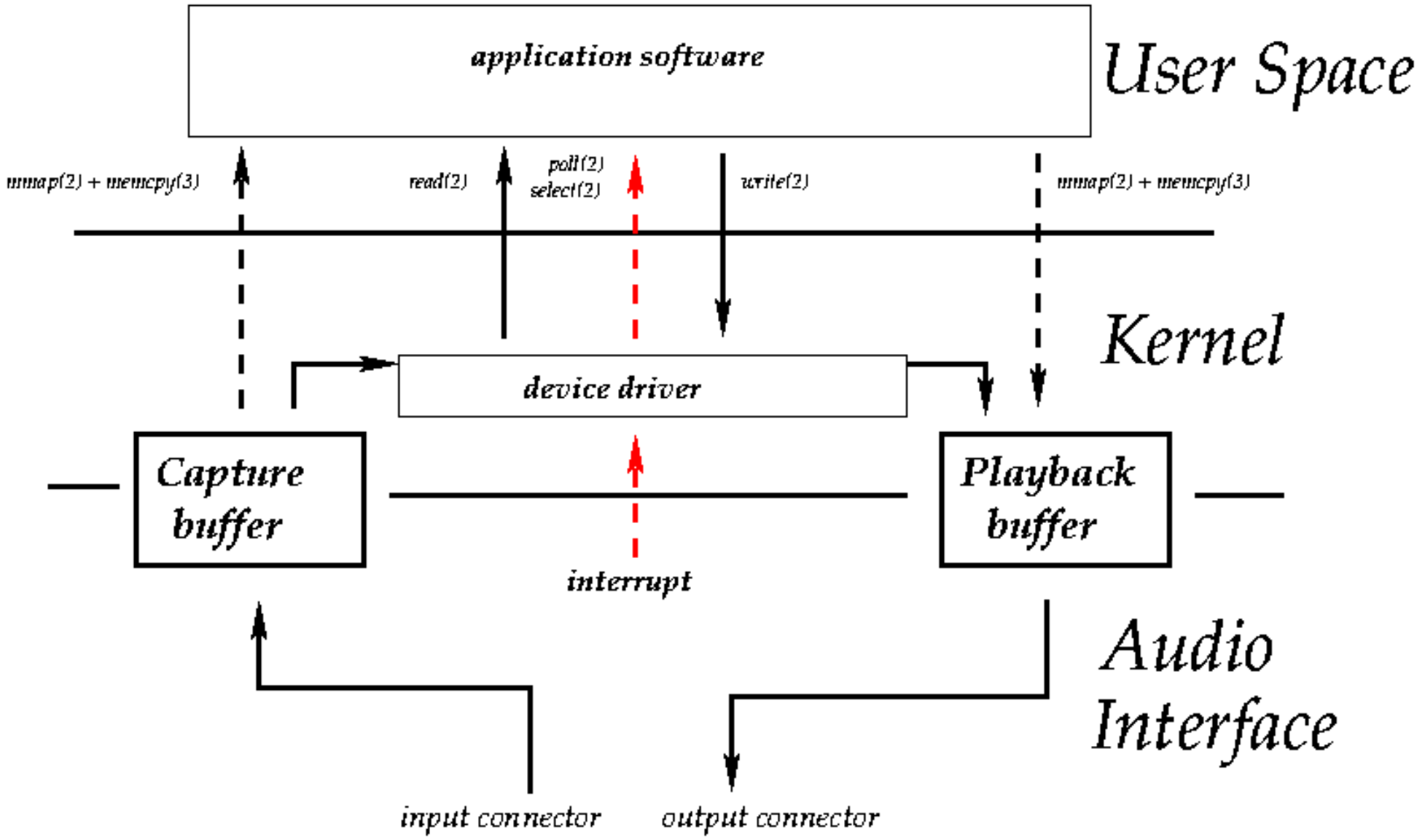


The Good News

You don't need to know
(much)

The Big Picture

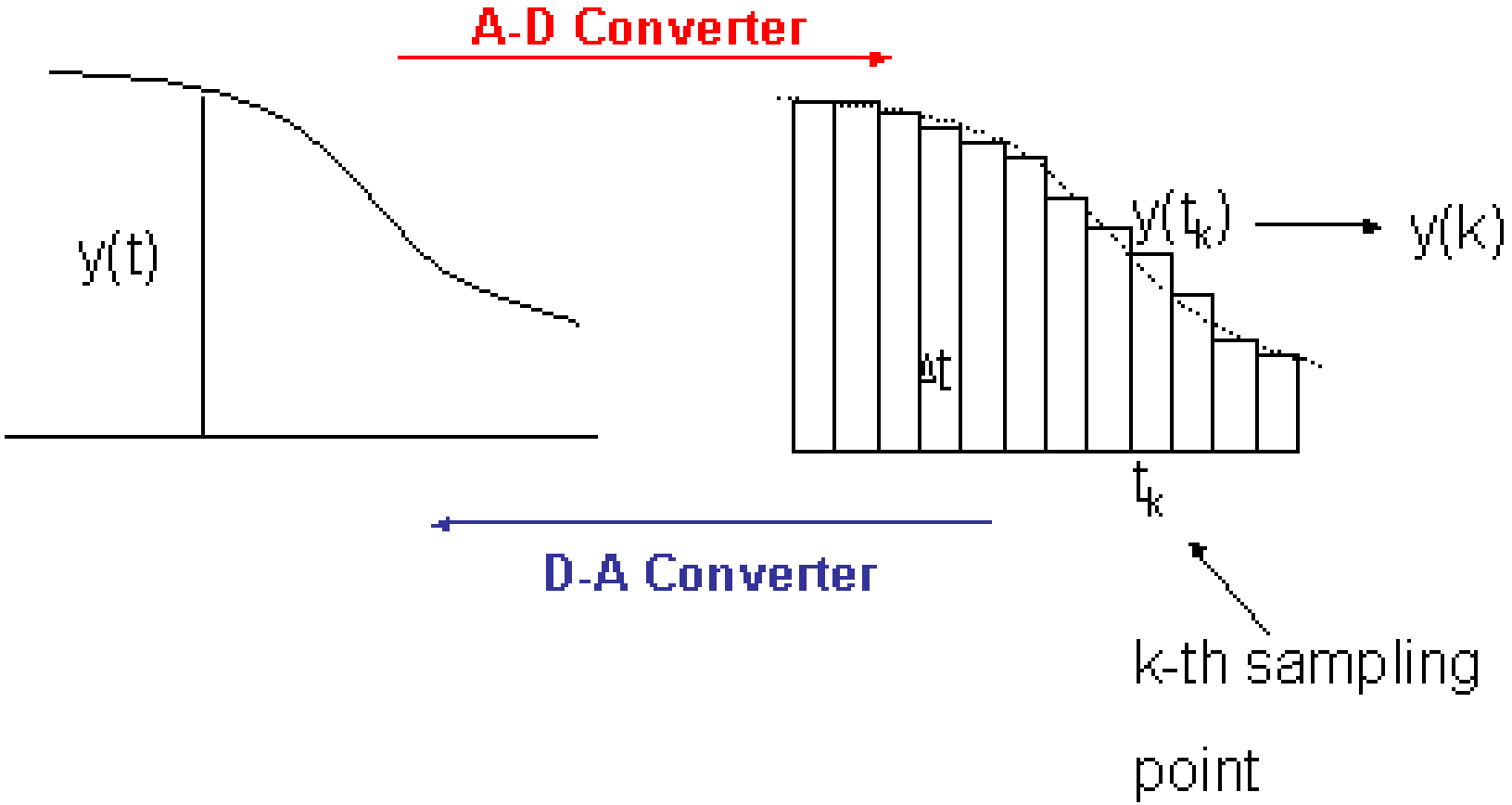



application software
User Space
mmap(2) + memcpy(3)
read(2)
poll(2)
select(2)
write(2)
mmap(2) + memcpy(3)
Kernel
device driver
Capture buffer
Playback buffer
interrupt
Audio Interface
input connector
output connector

Step #1: Getting audio TO the computer

- Analog signal: continuously variable information (e.g . acoustic pressure,electrical voltage)
- Digital signal: discretely valued data (e.g. 8 integer values from 0 to 7)
- Computers are digital, they don't manipulate analog signals
- Audio (electrical or acoustic versions) are traditionally analog signals
- So, we have to convert

Analog-Digital Digital-Analog Converter



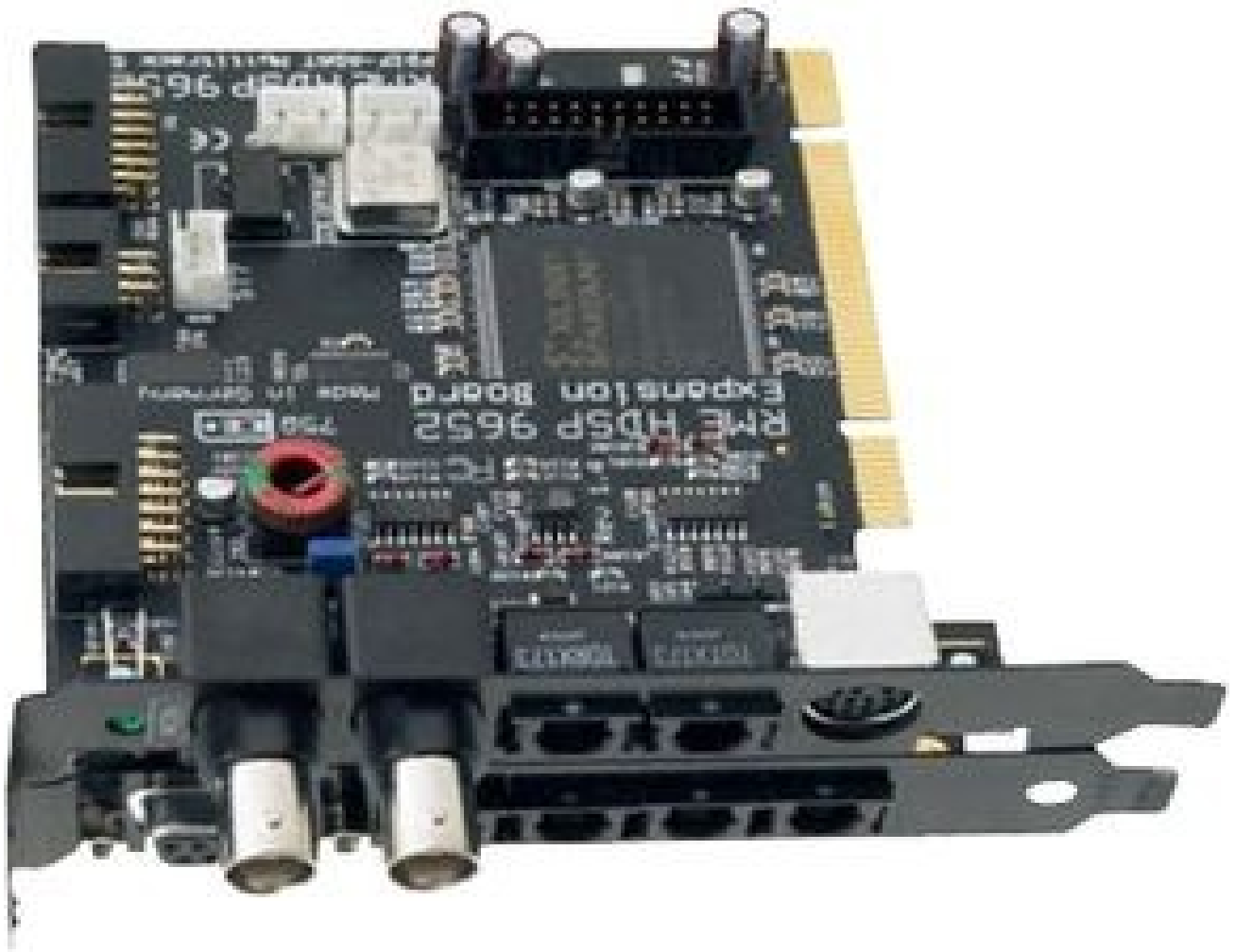
Converter Basics

- More bits per sample is good
  - 24 is enough
- More samples per second is good
  - 48KHz is probably enough
  - 96KHz definitely is
- Sampling: taking the value of an analog signal at regular time intervals
- The clock is the most overlooked component (by users)

Converter Placement

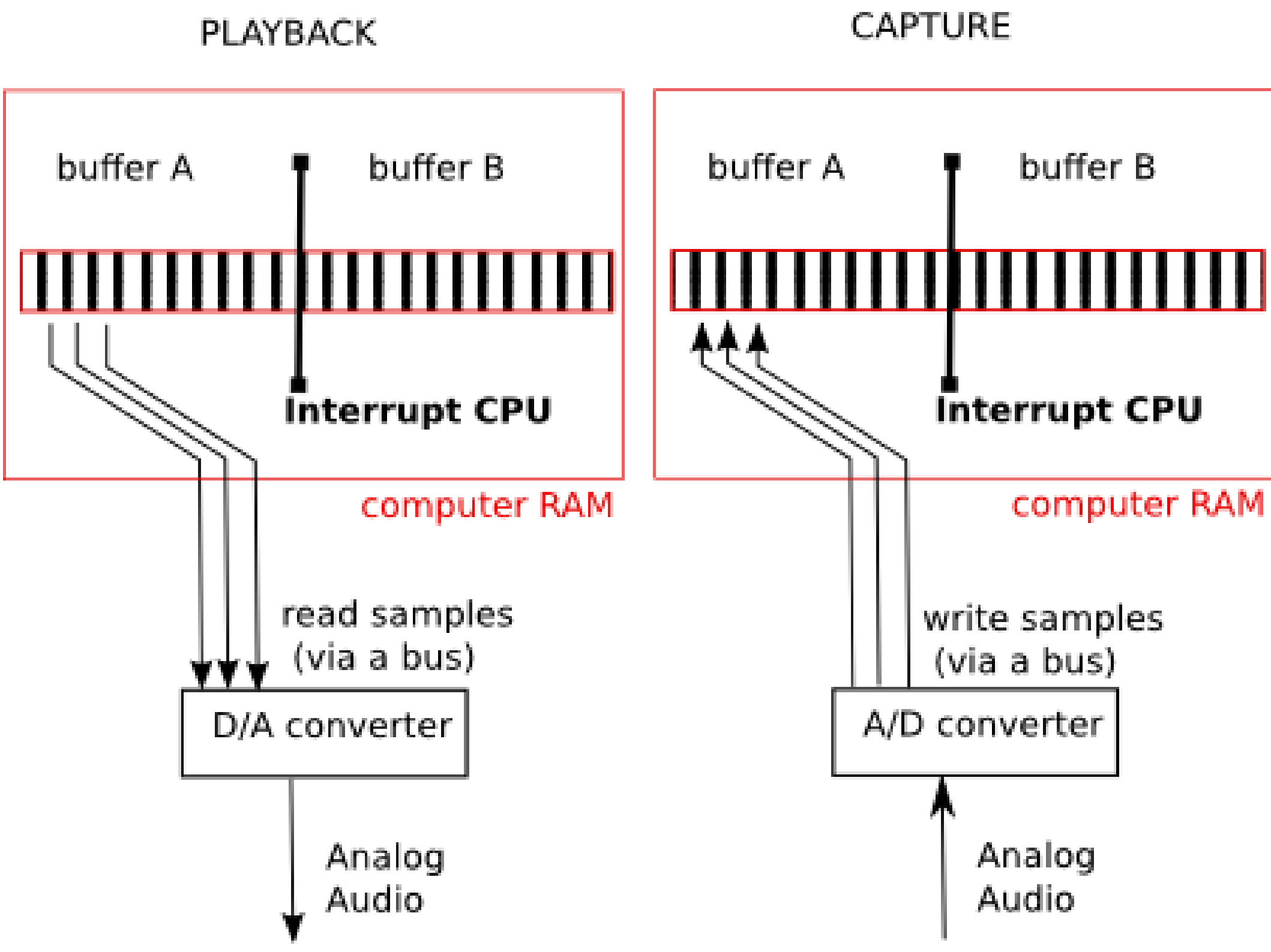
- Convert “outside” the computer (less RF noise, more modular, better physical cable connections)
- Convert “inside” the computer (cheaper, sometimes more convenient)
- Analog to Digital and Digital to Analog converters have to be somewhere
- We can deliver digital data or analog
- Now we're inside the box

It Starts Here



Why Hardware Matters

- Assumptions: (1) low latency is an important goal (2) realtime audio (not offline rendering) is the operating scenario
- Hardware imposes demands on software
- Possible to ignore hardware characteristics by using lots of buffering
- Buffering == latency
- If we can't buffer, then we have to deal with the way the hardware actually works


PLAYBACK
buffer A
buffer B
Interrupt CPU
computer RAM
read samples
(via a bus)
D/A converter
Analog
Audio
CAPTURE
buffer A
buffer B
Interrupt CPU
computer RAM
write samples
(via a bus)
A/D converter
Analog
Audio

Important Things

- The audio I/O hardware does not read or write single samples
- The number of samples it reads/writes (a “block”) is the lower limit to the latency
- The blocksize is limited by the bus used to move the data to/from memory
  - PCI: 64 bytes lower limit
- 32 bit floats == 2 samples, or 1 stereo frame

Less Important Things

- Bus differences
- USB and Firewire are isochronous busses
- Driven by a clock that signals when to transfer information
- Not the sample clock!
- PCI: interrupts after N samples
- Firewire: 1 packet every 125usecs
- However .. USB & Firewire are PCI-based too

Less Important Things

- Bus differences
- USB and Firewire are isochronous busses
- Driven by a clock that signals when to transfer information
- Not the sample clock!
- PCI: interrupts after N samples
- Firewire: 1 packet every 125usecs
- However .. USB & Firewire are PCI-based too
- So it doesn't really matter

Less Important Things

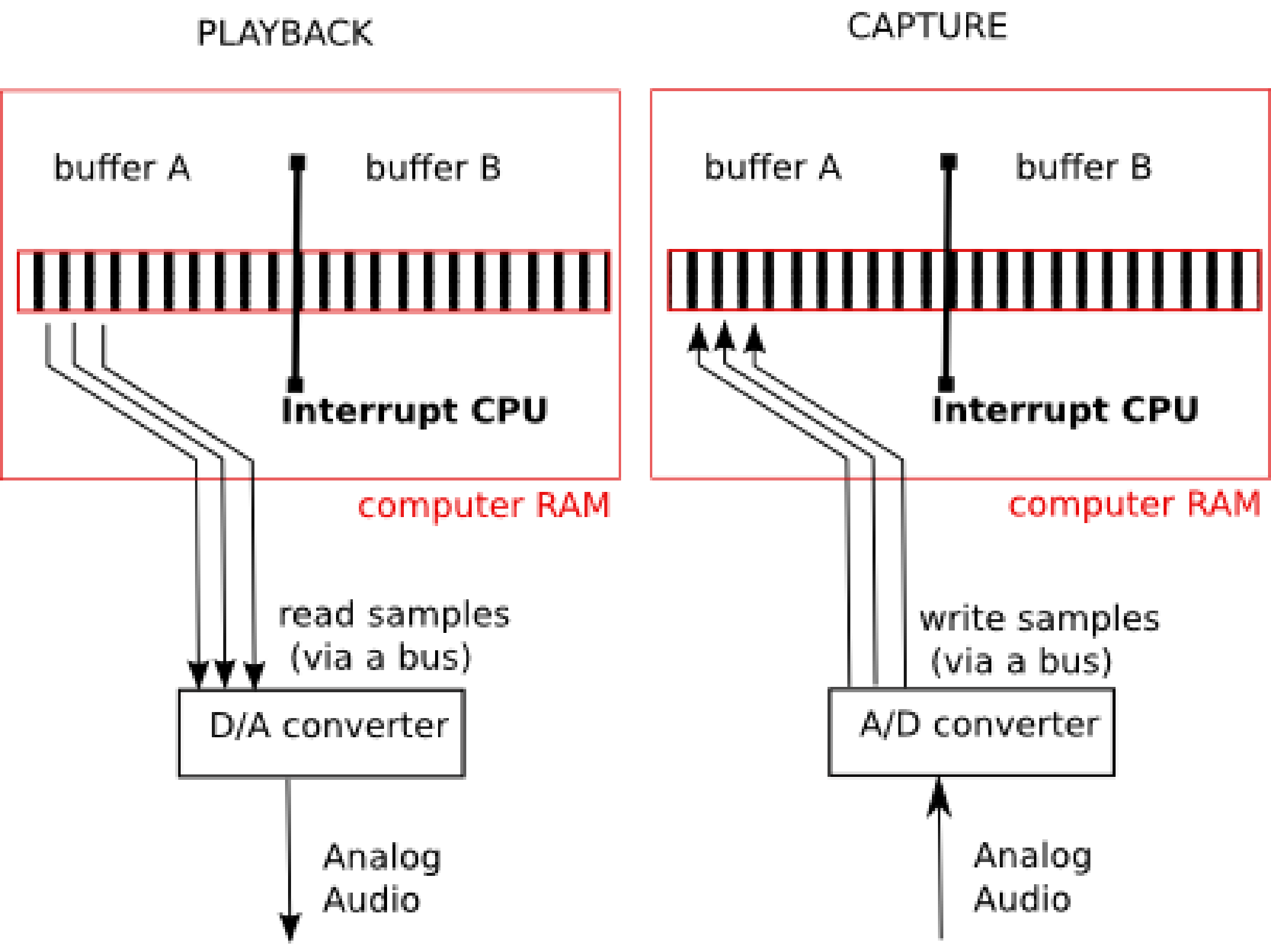
- Bus differences
- USB and Firewire are isochronous busses
- Driven by a clock that signals when to transfer information
- Not the sample clock!
- PCI: interrupts after N samples
- Firewire: 1 packet every 125usecs
- However .. USB & Firewire are PCI-based too
- So it doesn't really matter
- Except for USB

Less Important Things

- Bus differences
- USB and Firewire are isochronous busses
- Driven by a clock that signals when to transfer information
- Not the sample clock!
- PCI: interrupts after N samples
- Firewire: 1 packet every 125usecs
- However .. USB & Firewire are PCI-based too
- So it doesn't really matter
- Except for USB
- Which you should avoid using for audio

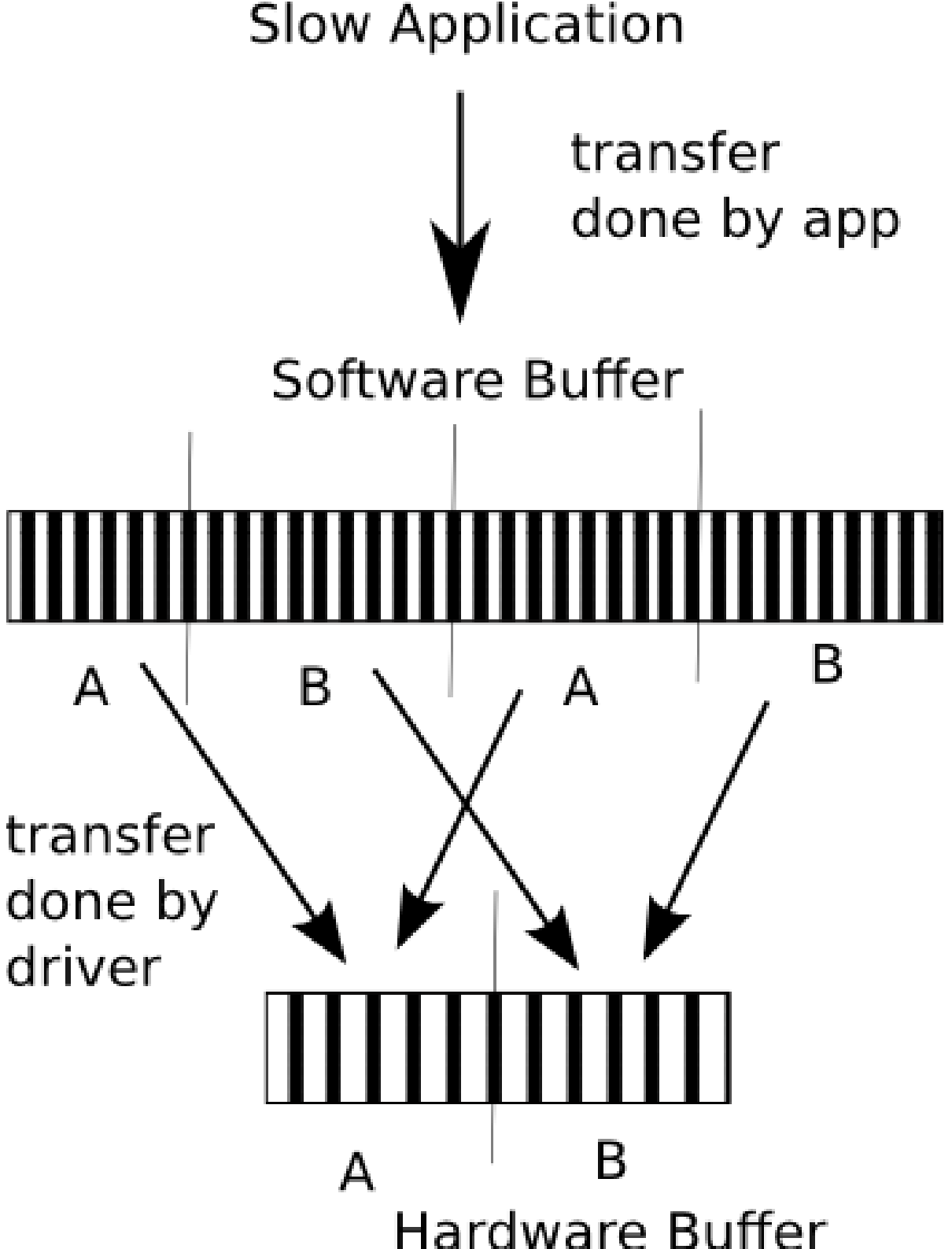
Important Things 2

- After the hardware has interrupted the CPU, there is “one buffer” of time left until we run out of data/space
- This is why audio needs “real time” handling by the operating system (and applications)
- For playback: miss the deadline, the hardware plays old samples
- For capture: the hardware overwrites samples that the CPU (and application hasn't read yet)


PLAYBACK
buffer A
buffer B
Interrupt CPU
computer RAM
read samples
(via a bus)
D/A converter
Analog
Audio
CAPTURE
buffer A
buffer B
Interrupt CPU
computer RAM
write samples
(via a bus)
A/D converter
Analog
Audio

Making Life Easier

- Make the hardware buffer bigger
  - (more samples = more time)
- Use a bigger buffer in software somewhere
  - Do a better job writing the OS


Slow Application
transfer
done by app
Software Buffer
A
B
A
B
transfer
done by
driver
A
B
Hardware Buffer

Push and Pull

- The hardware is going to carry on regardless of whether the software reads/writes samples in time
- “push” samples to the hardware based on the applications timing
- “pull” samples from the application based on hardware timing

Push Model

- Nice for application authors
- “Do what I want, when I want to”
- Load a soundfile into memory, write it to the audio interface. Done!
- Fits very well with the Unix open/close/read/write model for doing I/O
- Terrible for latency – some data is written seconds before its played back
- Absolute limits on what be written at once: h/w
- Real limits on what can be written: API & OS

Pull Model

- Tough on stupid application authors
- “Do what we tell you, when we tell you”
- Deliver data in small chunks, based on h/w and API internals
- Great for latency – we can write data very shortly before it is played
- If the chunks are small, the timing is very critical – hard on the application and the OS

Push-style APIs

- ALSA, Windows MME, other *nix-based APIs
- Basic functions
  - Open device
  - Configure device (sample rate, sample format, buffer size)
  - Read/write data
  - Close device

Pull-style APIs

- ASIO, CoreAudio, JACK
- Open device
- Possibly configure device, or query what the settings are
- Register a callback
- Wait for callbacks
- Close device

We're missing something

- There's a lot of music synthesis software
- There's a lot of FX software
- Audio I/O is not just about hardware anymore
- How do we move audio around between programs?

Answer #0: loopback cable

This is NOT an answer!!

Answer #1: don't

- This the official Apple and Windows answer
- Next contestant please.

Answer #2: Plugins

- Don't move audio around between programs
- Do everything inside one program
- Load plugins and allow them access to the audio data already flowing through the program
- Not bad!
- Its not for everyone
- Pressures standalone apps to provide plugin versions (eg. melodyne)

Answer #3: Rewire

- *"Think of Rewire as an invisible cable that streams audio from one application to another"*
- Limited number of channels
- Proprietary (and **very** proprietary)
- I can't tell you how it really works
- No Linux (sob, sob)
- Relatively widely used but not useful for free or open source applications
- There is also DirectConnect (Digidesign), ReaRoute (Cockos/Reaper) which are very similar designs

Answer #4


YOU DON'T KNOW
JACK
Volume
YOU DON'T KNOW
JACK
YOU DON'T KNOW
JACK
"The Lost Gold"
The Original Irreverent Quiz Show Party Game

No, the other one

JACK Basics

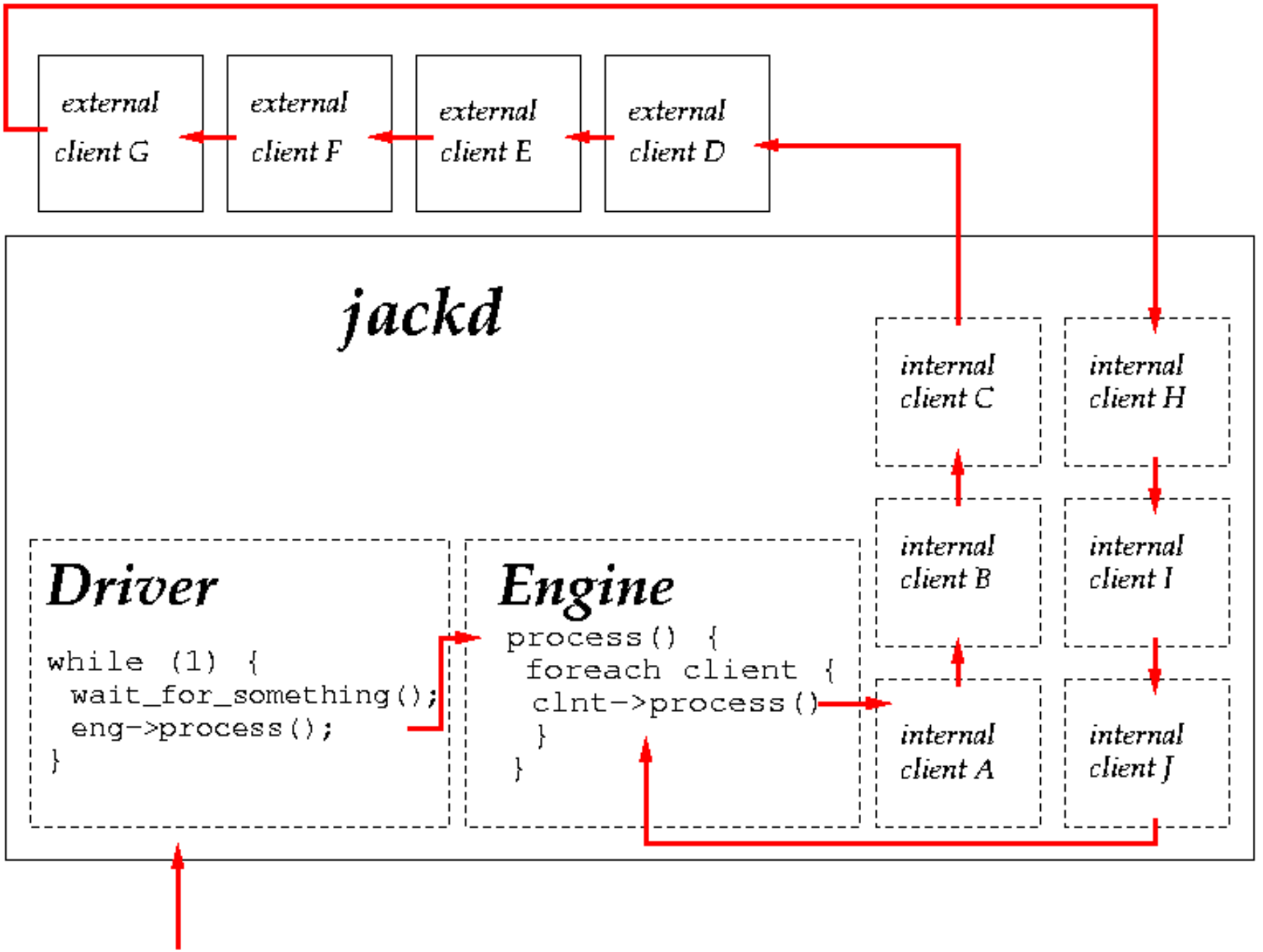
- Target: pro-audio and musical applications doing realtime, low latency audio I/O
  - Shared access to hardware
  - Audio routing between different software
  - Identical API regardless of whether audio is routed to hardware or other software
- Sample synchronous execution: in a given time period, all clients operate on audio that will be played (or was captured) at the same time

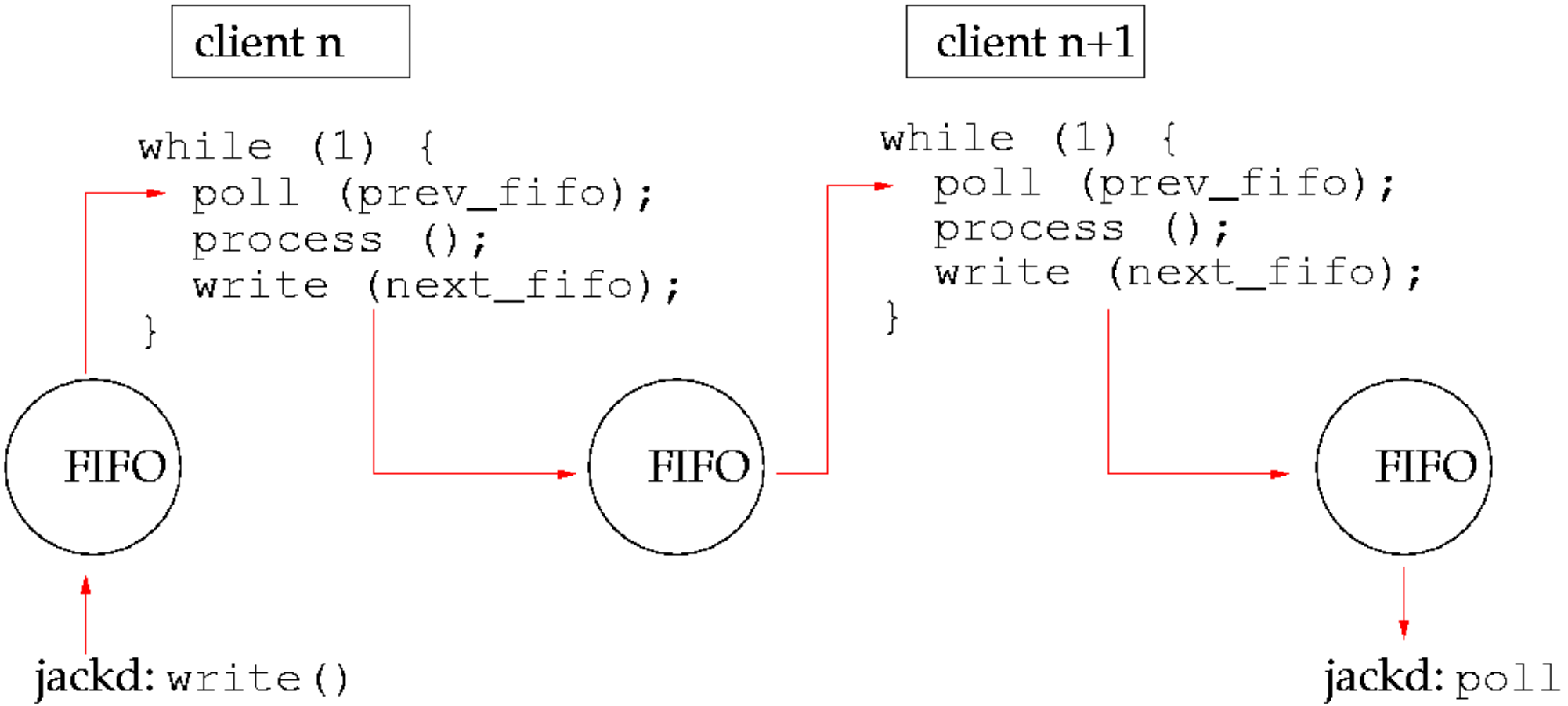
JACK Abstractions

- Server: “jackd” - runs the show
- Backend: loaded by server, interacts with hardware, drives timing, does “real” audio i/o
- Client: talks to server and other clients
- Port: a place to read/write data
- Basic callback: `int process (nframes_t)`
- Many other callbacks, none required
- Libjack: linked into client applications, creates and manages thread(s) used to make JACK work

JACK Properties

- Any number of channels from any number of clients
- Arbitrary routing (including feedback loops)
- Clients can connect, disconnect from the server at will
- Arbitrary data types (currently just audio & MIDI, though video patches exist)
- Ports can be connected and disconnected at will
- No data copying is involved when routing audio from one client to another (its all in shared memory)
- Shared transport control


external client G
external client F
external client E
external client D
jackd
internal client C
internal client H
internal client B
internal client I
internal client A
internal client J
Driver
while (1) {
wait_for_something();
eng->process();
}
Engine
process() {
foreach client {
clnt->process()
}
}
interrupt from audio interface


client n
client n+1
while (1) {
poll (prev_fifo);
process ();
write (next_fifo);
}
while (1) {
poll (prev_fifo);
process ();
write (next_fifo);
}
FIFO
FIFO
FIFO
jackd: write()
jackd: poll

JACK Coolness

- Add a new client with a new property – all JACK clients gain this
  - e.g network audio, timecode input/output
- JACK runs every WFS installation in the world
- JACK in use on Harrison consoles (ethernet audio I/O)

As an API JACK provides many benefits because of its high-level of abstraction:

* no hardware configuration
* no software (driver) configuration
* no format negotiation (all channels are mono, 32 bit floating point)
* no main loop (JACK-managed thread handles it)

** on the other hand:*

- requires multi-threaded (lock-free) programming skills
- involves use of a client/server system
- "pull" model is harder for some kinds of apps

The Future of JACK

- Ported to Windows (done)
- JACK 1.0 release (expected in November)
- Jackdmp formally adopted as codebase for JACK 2.0
- JACK server control API finished
- MIDI integration on OS X
- The API is stable – old applications do not even need to be recompiled

Giving Credit Where Its Due

- I was the original author of JACK
- Many others have made major contributions
  - Jack O'Quinn
  - Stephane Letz
  - Pieter Pallmers
  - Rui Nuno Capela
- Andy Wingo, Kai Vehmanen, Fernando Pablo Lopez-Lezcano, Jeremy
  - Hall, Steve Harris, Martin Boer, Taybin Rutkin, Melanie Thielker,
  - Jussi Laako, Tilman Linneweh, Johnny Petrantoni, Karsten Wiese,
  - Lee Revell, Ian Esten, Frank van der Pol, Dmitry Baikov, Pieter
  - Palmers, Nedko Arnaudov, Jacob Meuser, Marc-Olivier Barre
  - And others

Next Week

- Threads & Parallelism #1
- As usual: http://tu.linuxaudiosystems.com/
- Please check the website starting tomorrow for some reading materials