COM-202: Signal Processing

Chapter 6.c: Real-time audio processing

Summary:

- I/O and DMA
- multiple buffering
- implementation framework
- some guitar effects

Real-time audio processing

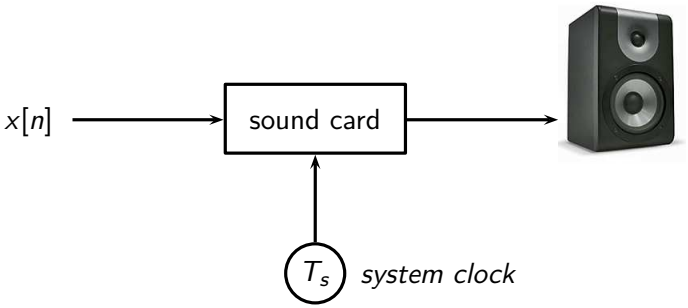
Real-time processing

Everything works in sync with a *system clock* of period T_s:

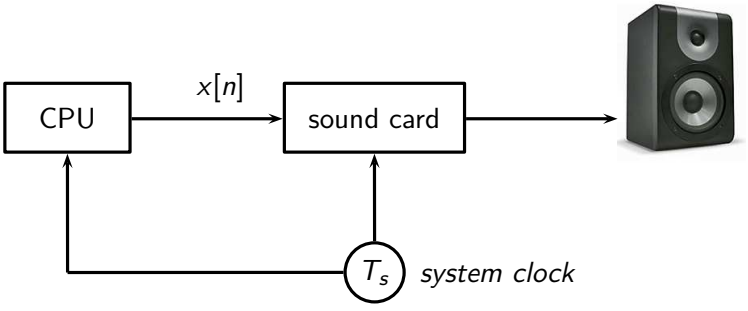
- "record" a value $x_i[n]$
- process the value in a causal filter
- "play" the output $x_o[n]$

everything needs to happen in at most T_s seconds!

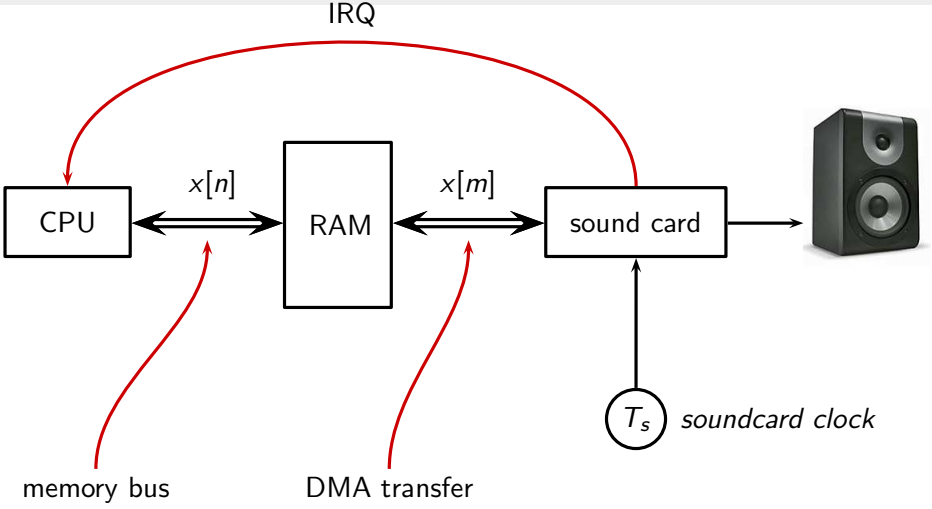
Playing a sound



On dedicated hardware...



On a PC...

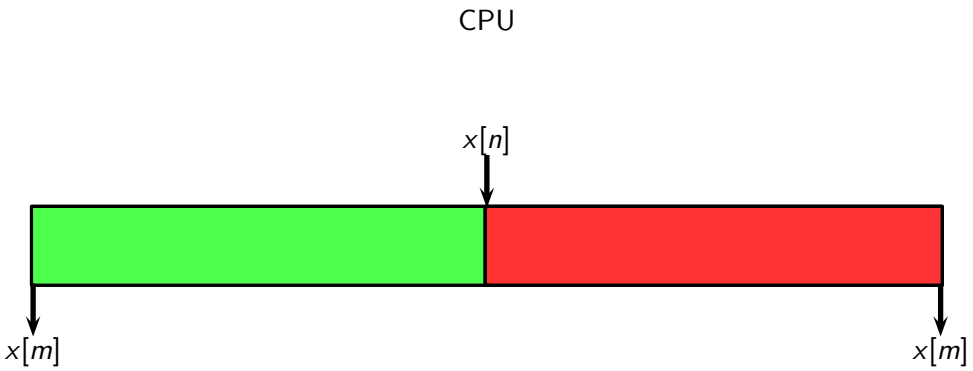

IRQ
CPU
x[n]
RAM
x[m]
sound card
T_s
soundcard clock
memory bus
DMA transfer

Buffering

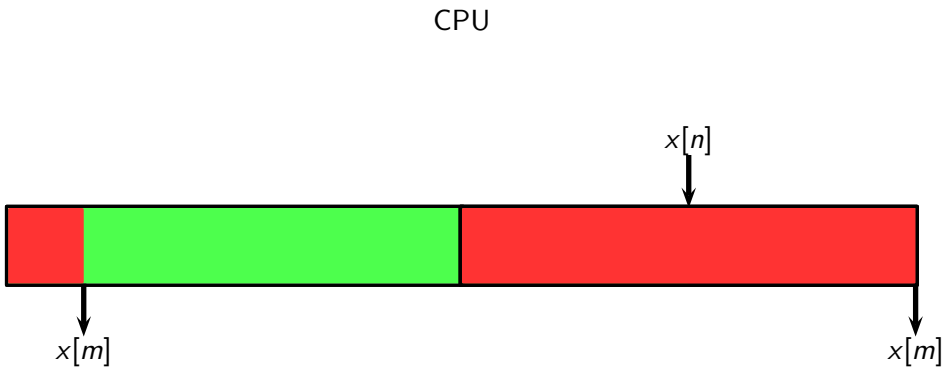
- interrupt for each sample would be too much overhead
- soundcard consumes sample in buffers
- soundcard notifies when buffer used up
- CPU can fill a buffer in less time than soundcard can empty it

buffering introduces delay!

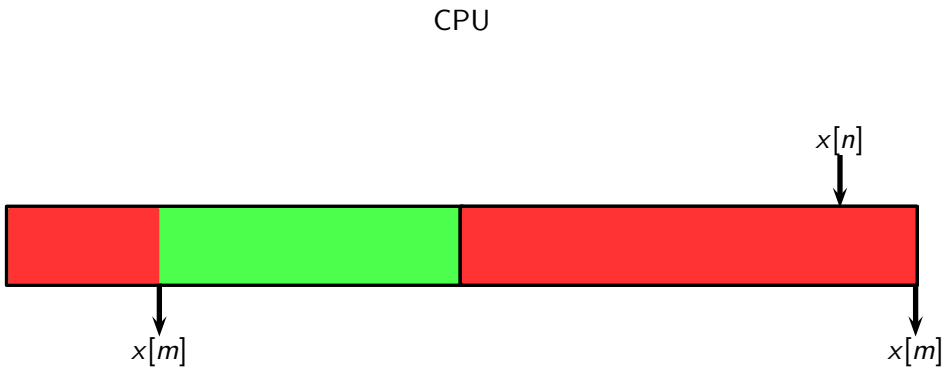
Example: double buffering (output)



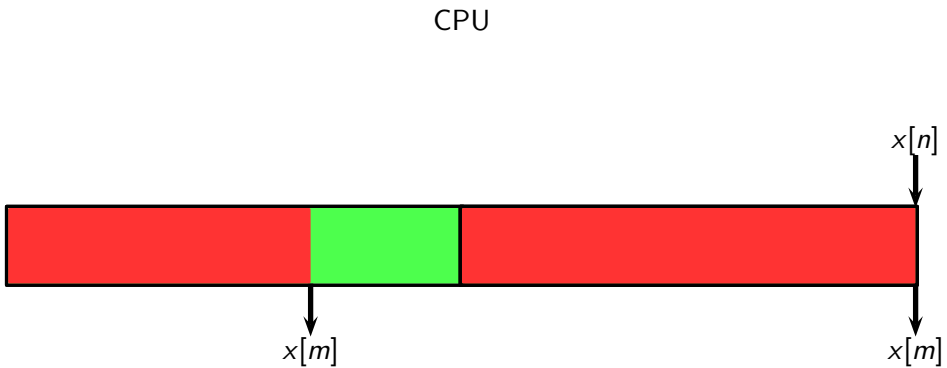
Example: double buffering (output)



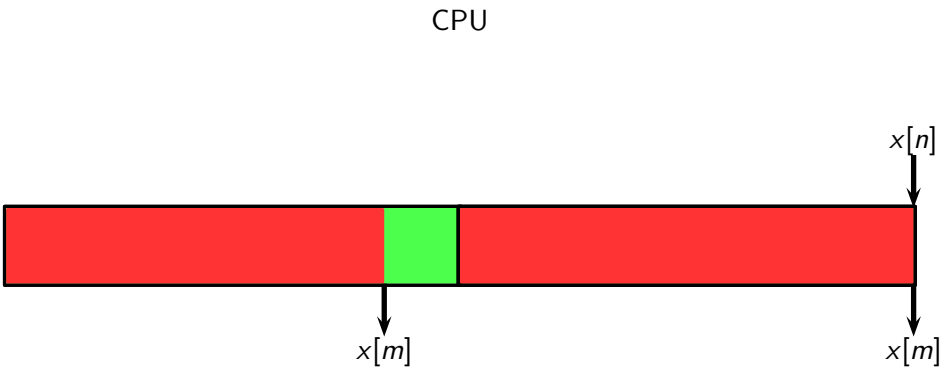
Example: double buffering (output)



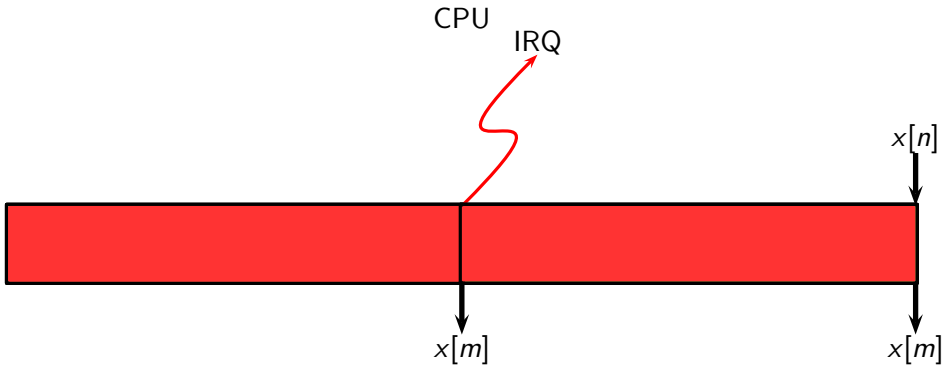
Example: double buffering (output)



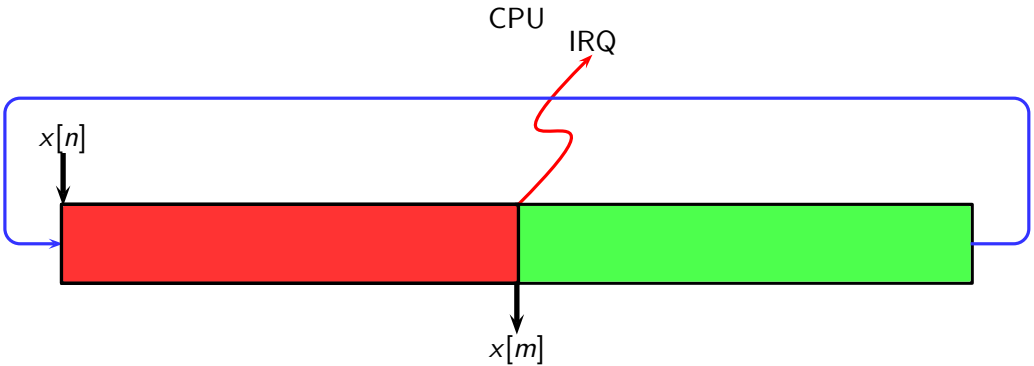
Example: double buffering (output)



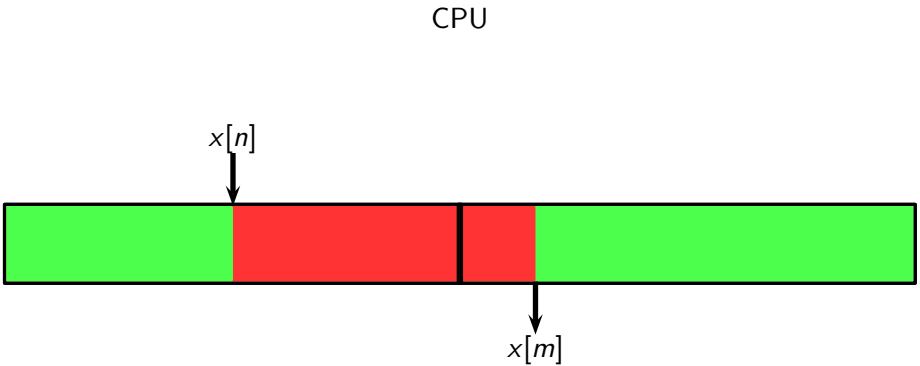
Example: double buffering (output)



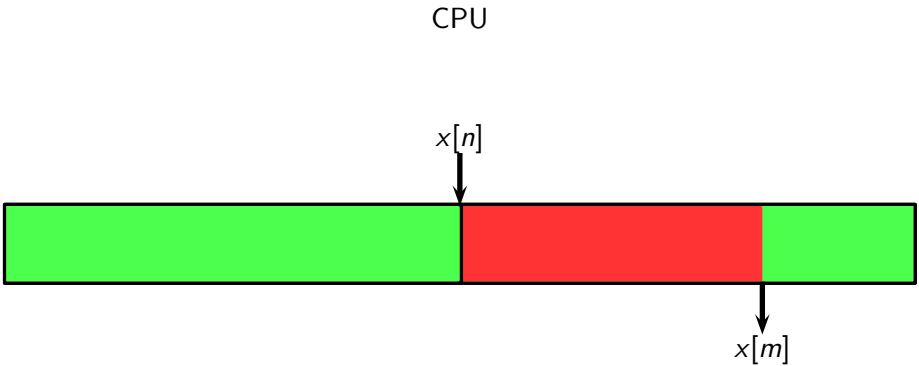
Example: double buffering (output)



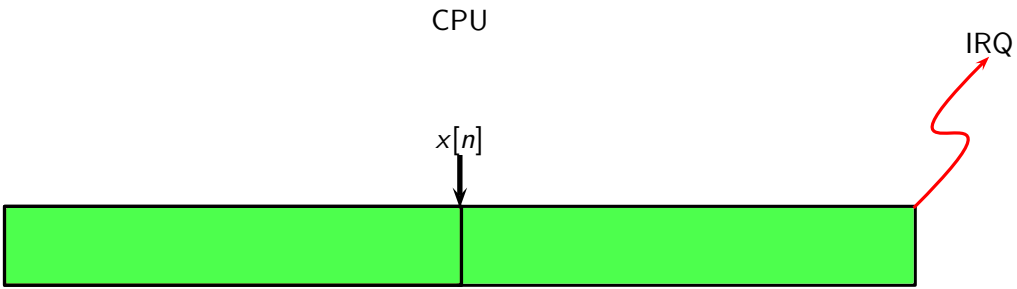
Example: double buffering (output)



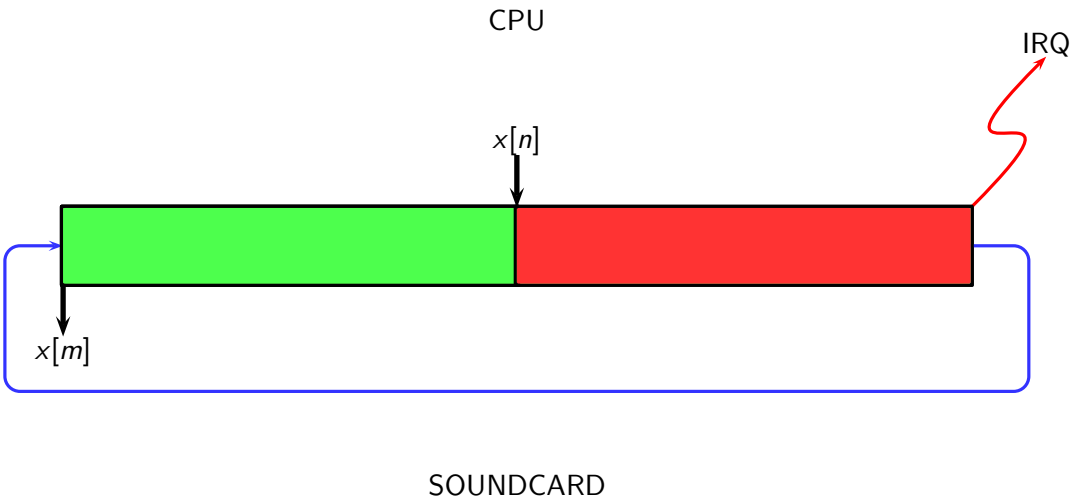
Example: double buffering (output)



Example: double buffering (output)



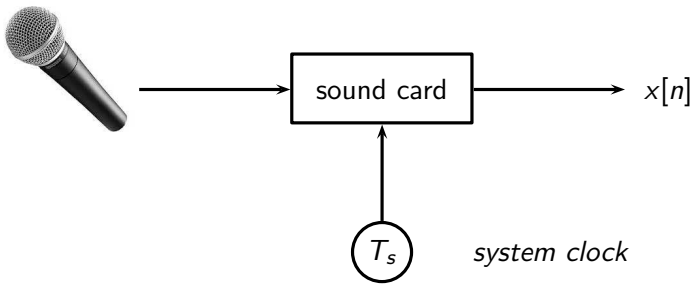
Example: double buffering (output)



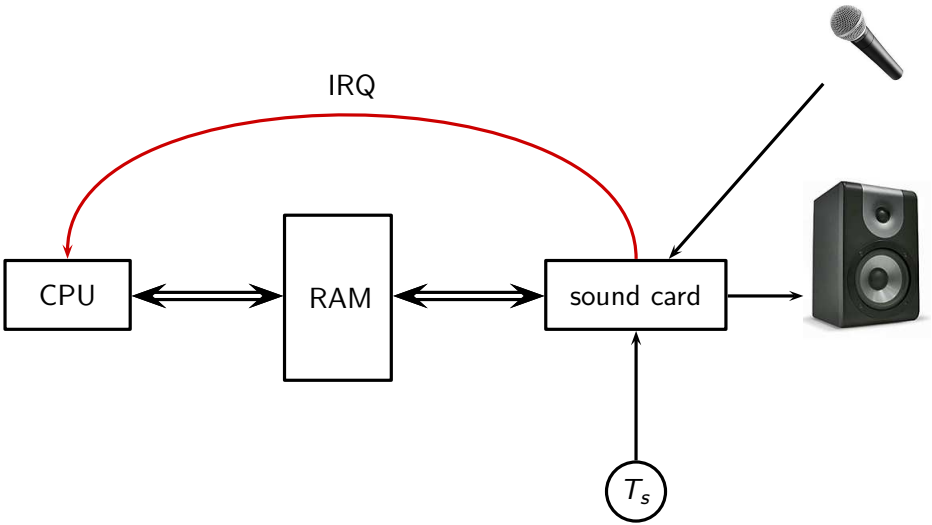
Example: double buffering

- double buffering introduces a delay $d = T_s \times \frac{L}{2}$ seconds
- if CPU doesn't fill the buffer fast enough: **underflow**

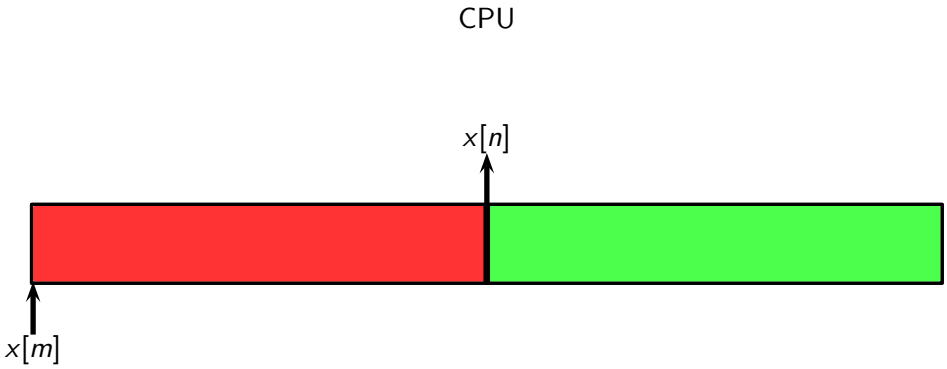
What about the input?



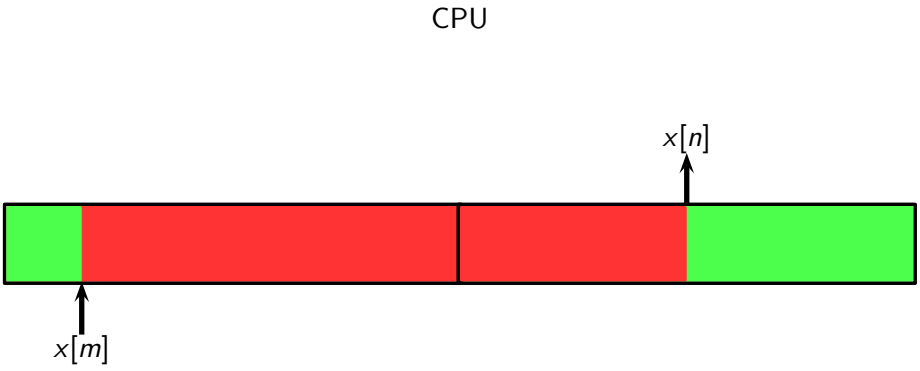
On a PC...



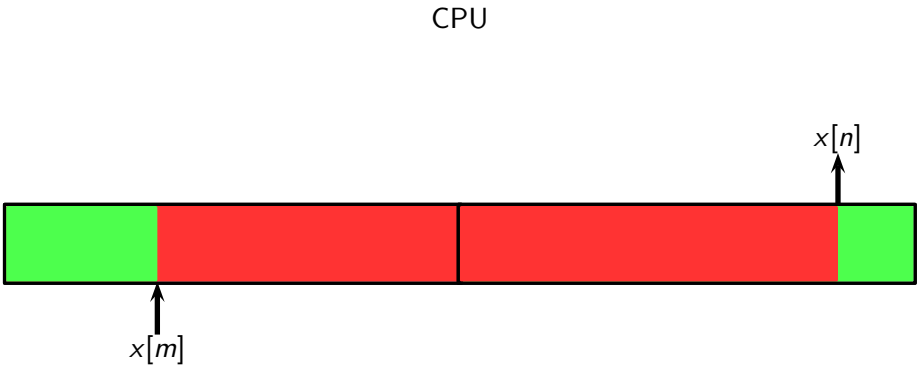
Example: double buffering (input)



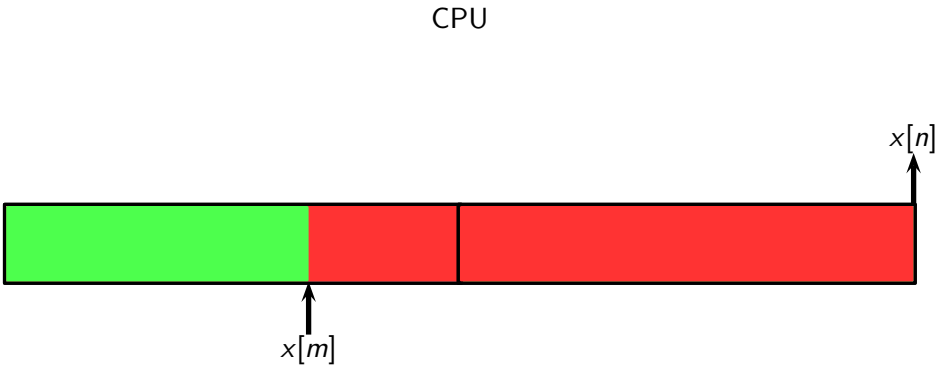
Example: double buffering (input)



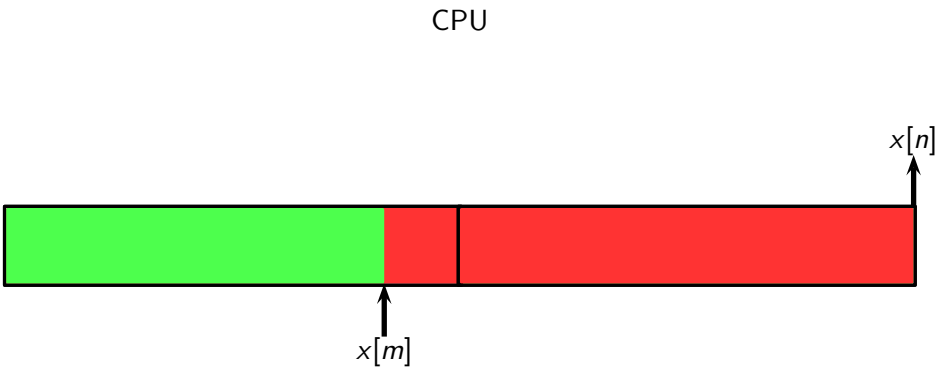
Example: double buffering (input)



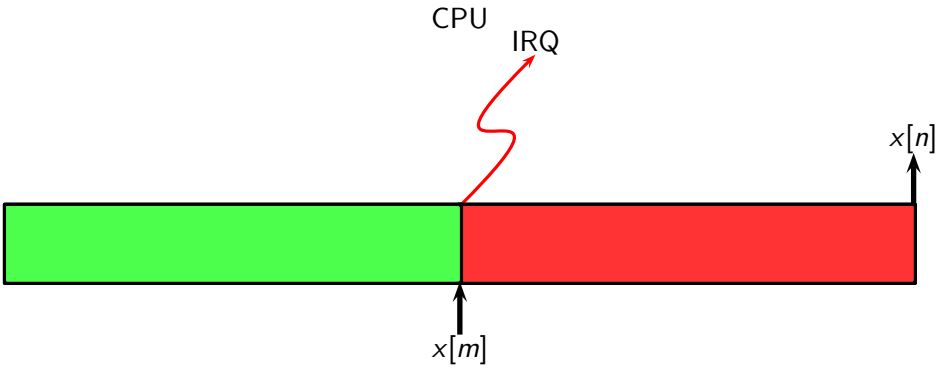
Example: double buffering (input)



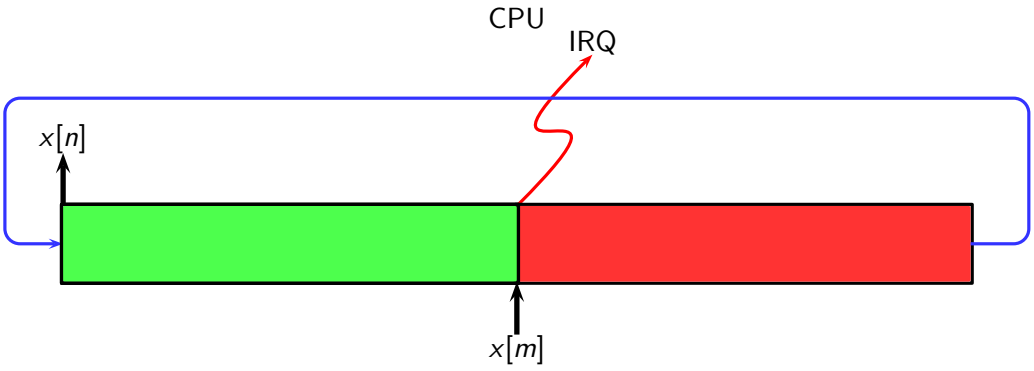
Example: double buffering (input)



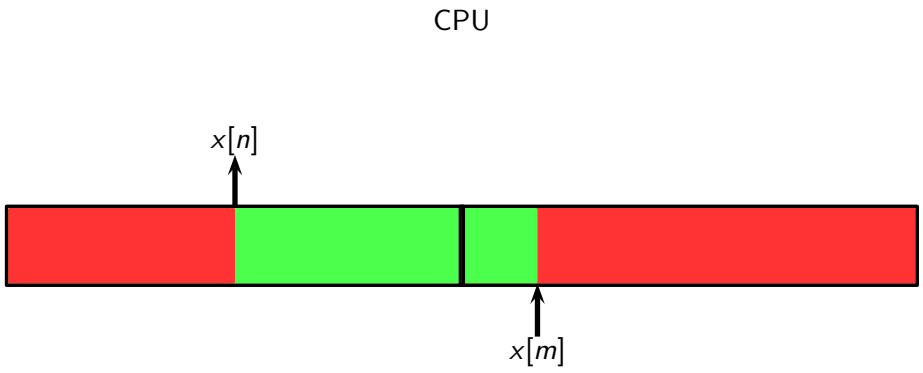
Example: double buffering (input)



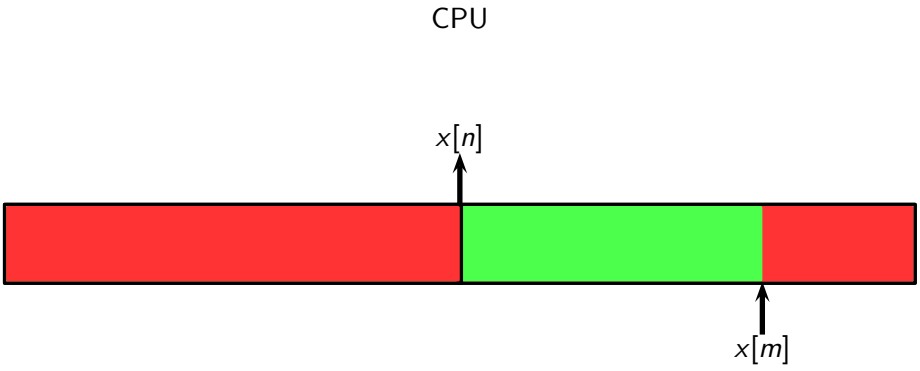
Example: double buffering (input)



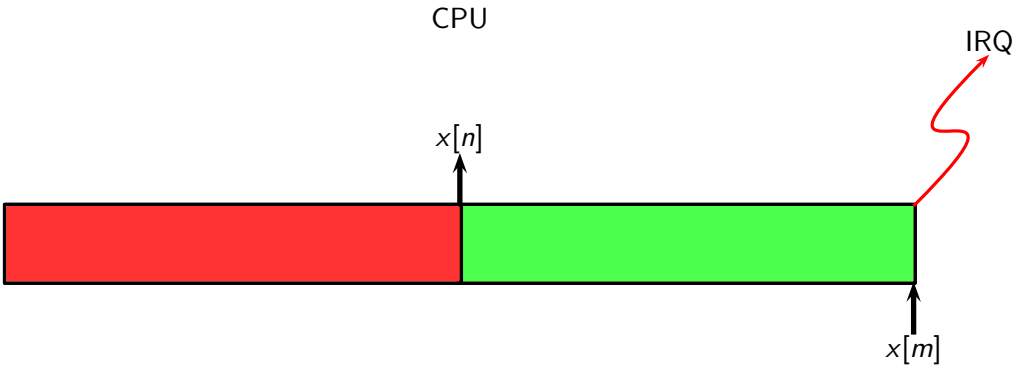
Example: double buffering (input)



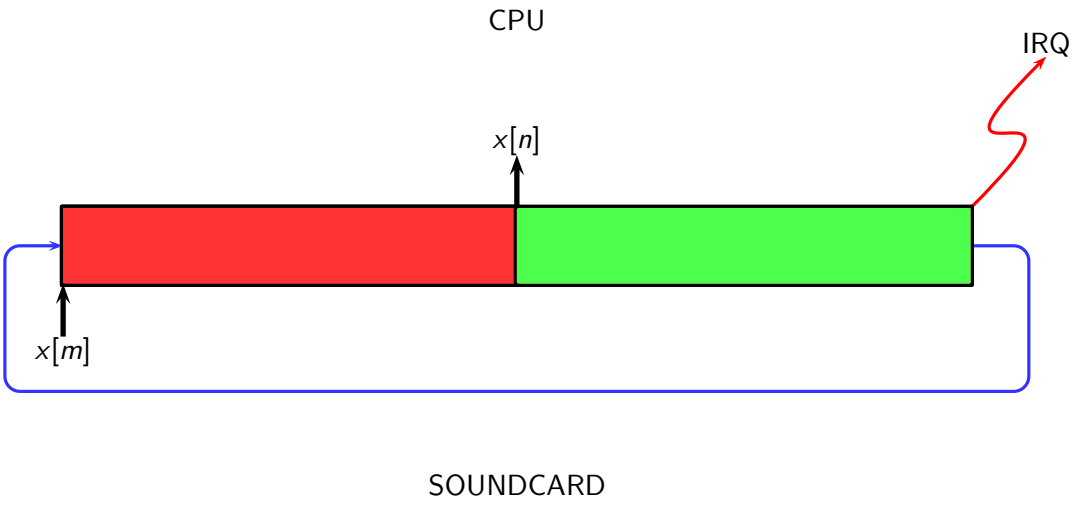
Example: double buffering (input)



Example: double buffering (input)



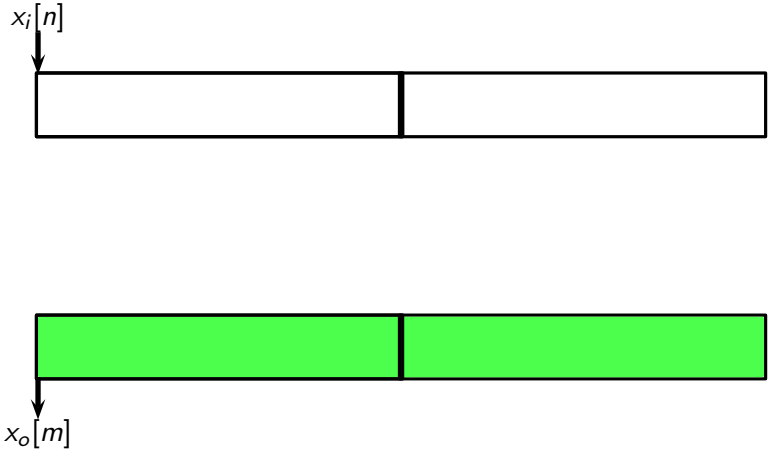
Example: double buffering (input)



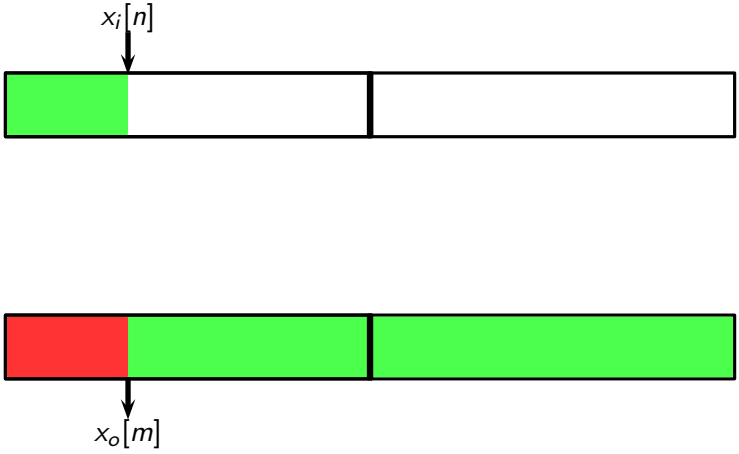
Putting it all together

- multiple input buffers and output buffers
- equal chunk sizes
- input IRQ drives processing

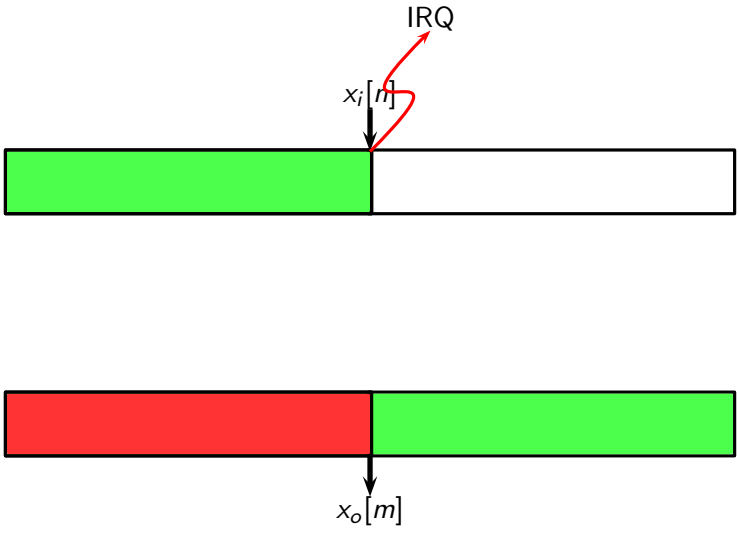
Real-time I/O processing with double buffering



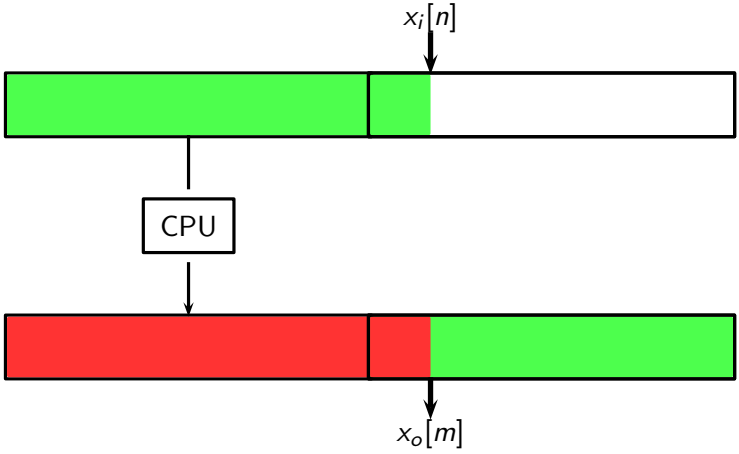
Real-time I/O processing with double buffering



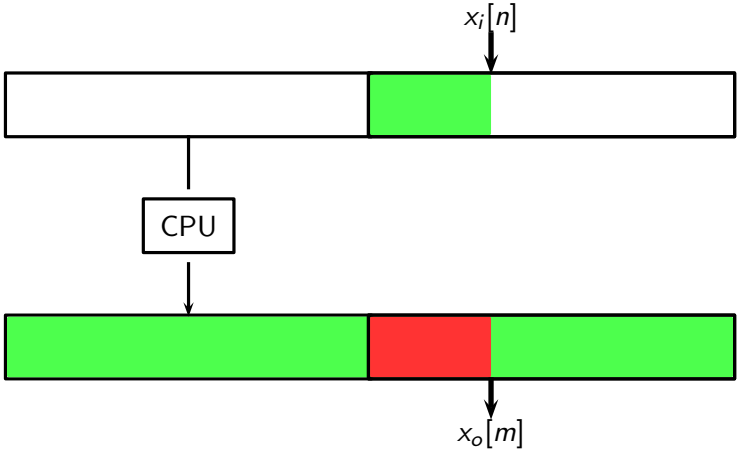
Real-time I/O processing with double buffering



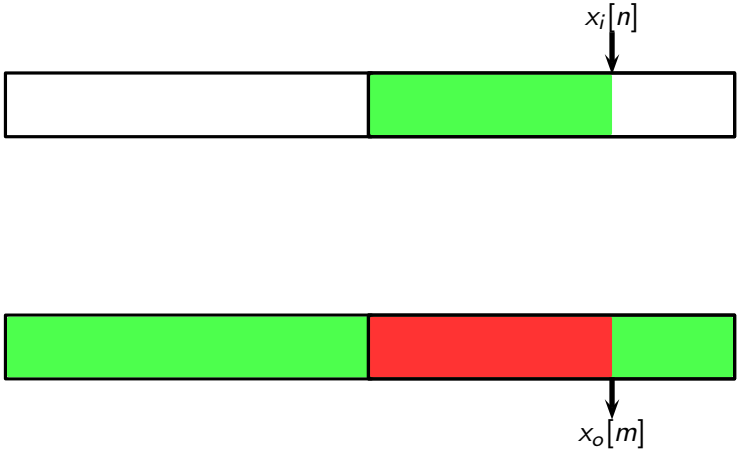
Real-time I/O processing with double buffering



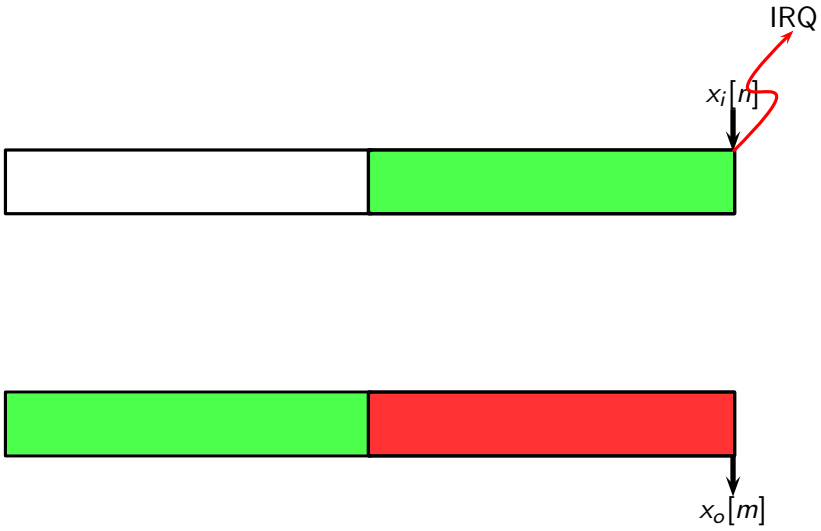
Real-time I/O processing with double buffering



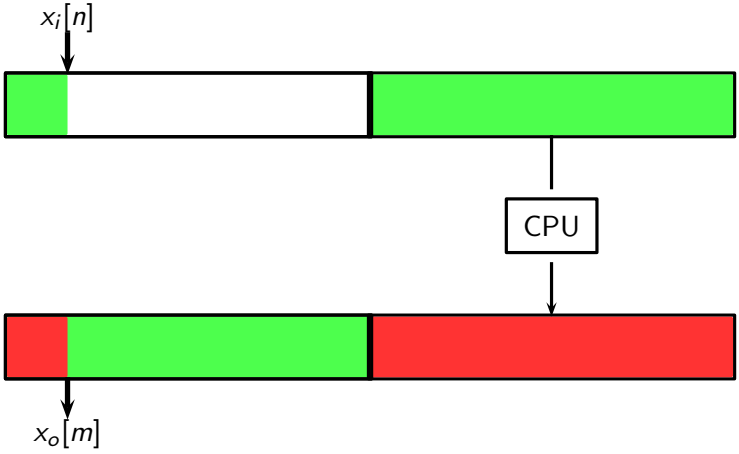
Real-time I/O processing with double buffering



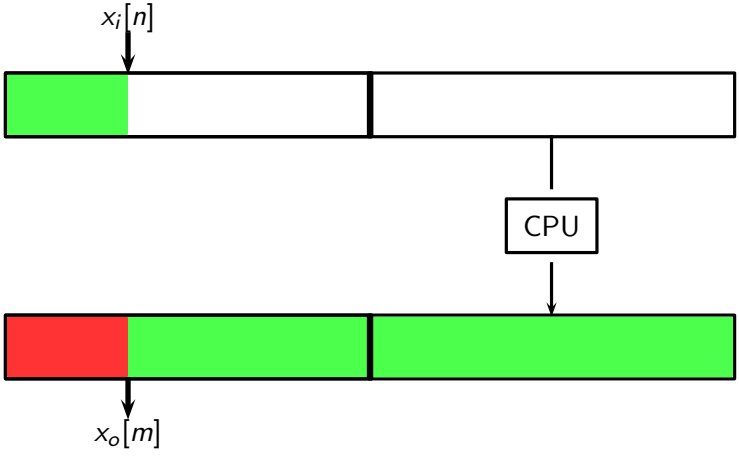
Real-time I/O processing with double buffering



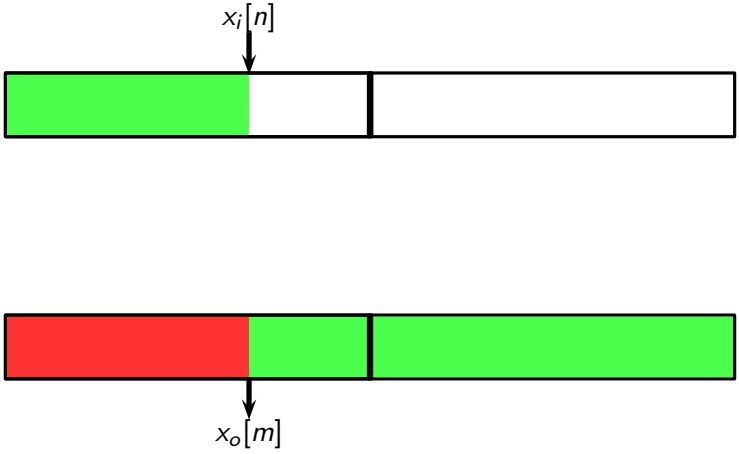
Real-time I/O processing with double buffering



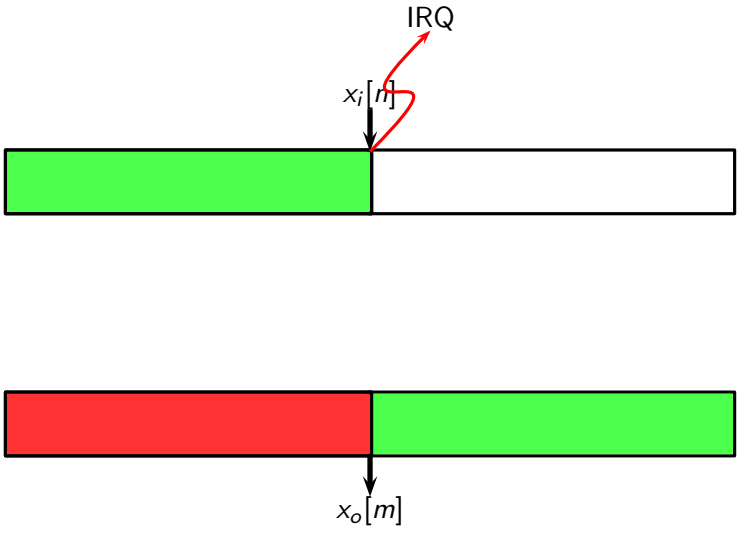
Real-time I/O processing with double buffering



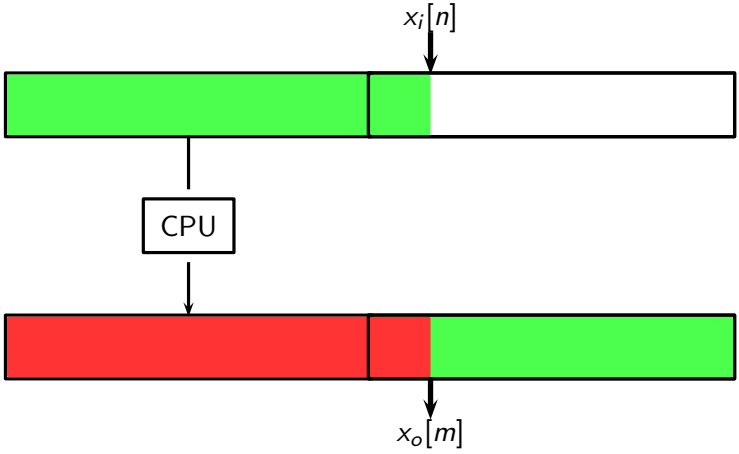
Real-time I/O processing with double buffering



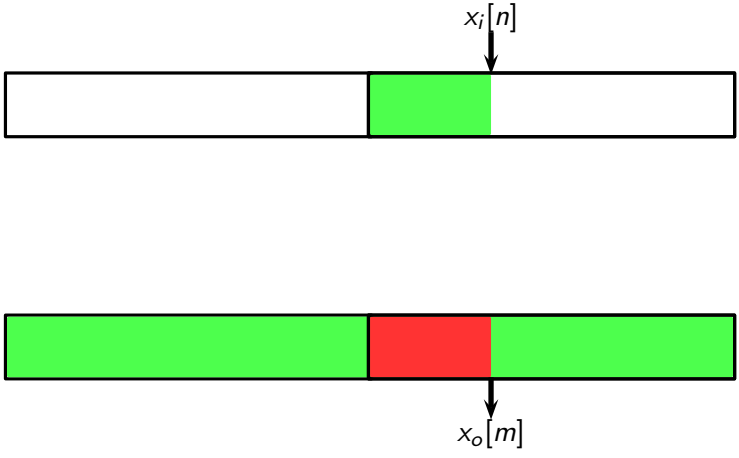
Real-time I/O processing with double buffering



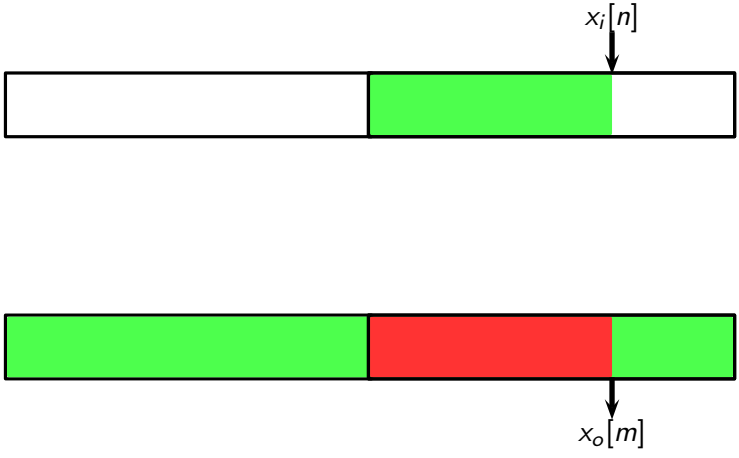
Real-time I/O processing with double buffering



Real-time I/O processing with double buffering



Real-time I/O processing with double buffering



Real-time I/O processing with double buffering

- total delay $d = T_s \times L$ seconds
- usually start output process first
- buffers can be collapsed (use same memory space)

Implementation

- low level:
  - study soundcard data sheet (each one is different)
  - write code to program soundcard via writes to IO ports
  - write an interrupt handler
  - write the code to handle the data
- high level:
  - choose a good API (eg. PortAudio)
  - write a callback function to handle the data

real-time audio processing with PyAudio

PyAudio

- simple Python wrapper for PortAudio
- cross-platform real-time audio playback and/or capture
- relies on the underlying OS audio API $\Rightarrow$ can't control min latency
- not really suitable for real-time processing (delay too big) but good proof of concept

The callback prototype

```
def callback(in_data, frame_count, time_info, status):
    audio_in = np.array(np.frombuffer(in_data, dtype=np.int16))
    audio_out = np.int16(processor.process(audio_in))
    return audio_out, pyaudio.paContinue
```

Processing gateway

```
class RTProcessor:
    def __init__(self, rate, channels=1, max_delay=1):
        self.SF = rate
        self.x = CircularBuffer(max_delay)
        self.y = CircularBuffer(max_delay)

    def process(self, samples):
        for n, x in enumerate(samples):
            y = self._process(x)
            self.x.push(x)
            self.y.push(y)
            samples[n] = y
        return samples
```

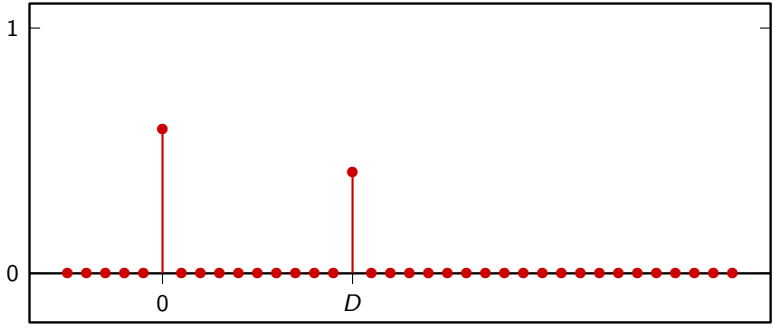
Circular Buffer

```
class CircularBuffer(object):
    def __init__(self, length):
        self.length = length
        self.buf = np.zeros(self.length).astype(float)
        self.ix = 0

    def push(self, x):
        self.buf[self.ix] = x
        self.ix = np.mod(self.ix + 1, self.length)

    def get(self, n):
        assert n > 0, 'can only access past values'
        return self.buf[np.mod(self.ix + self.length - n, self.length)]
```

guitar effects

Simple Echo

$$y[n] = \frac{x[n] + \alpha\, x[n - D]}{1 + \alpha}$$



Simple Echo

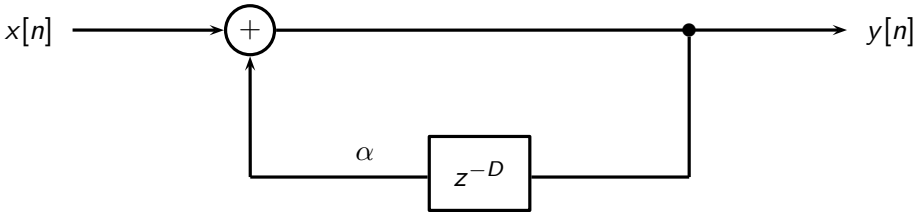
```
class Echo(RTProcessor):
    def __init__(self, rate, channels):
        # 1 replicas 1/3 of a sec apart -> 1 sec buffering
        super().__init__(rate, channels, max_delay=rate)

        self.alpha = 0.7
        self.norm = 1.0 / (1 + self.alpha)
        self.D = int(0.3 * self.SF)

    def _process(self, x):
        return self.norm * (x + self.alpha * self.x.get(self.D))
```

Slapback echo

- echo sounds like a distinct echo only if delay greater than 100ms
- slapback echo uses a delay at the threshold value ($\approx$ 100 ms) and $\alpha \approx 0.4$
- at the border between echo and reverb
- used often for vocals until the 1970s (Elvis, Lennon) and for rockabilly guitar

A recursive echo

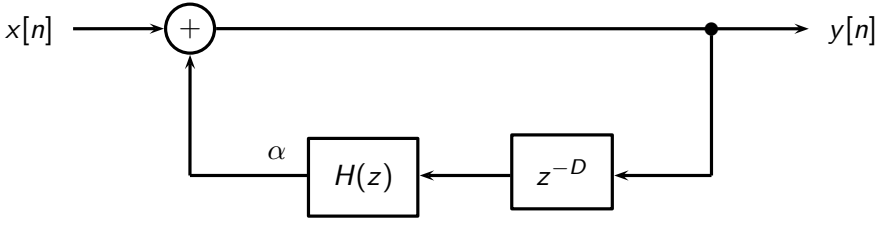
simple feedback loop to simulate back and forth reflections



$$y[n] = \alpha\, y[n - D] + x[n]$$

Adding the effect of materials

reflections may have a non-uniform response (e.g. lowpass)

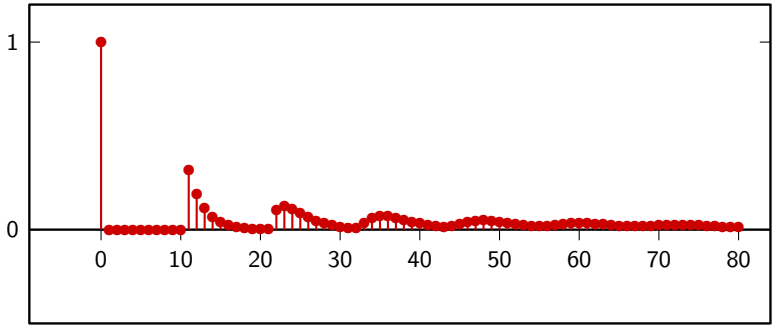


$$y[n] = \alpha(h * y)[n - D] + x[n]$$

A more realistic echo

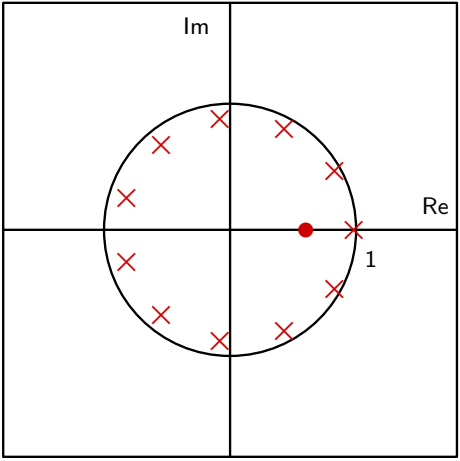
Choose for instance $H(z) = (1 - \lambda)(1 - \lambda z^{-1})$, a leaky integrator:

$$y[n] = x[n] - \lambda x[n-1] + \lambda y[n-1] + \alpha(1-\lambda)y[n-D]$$

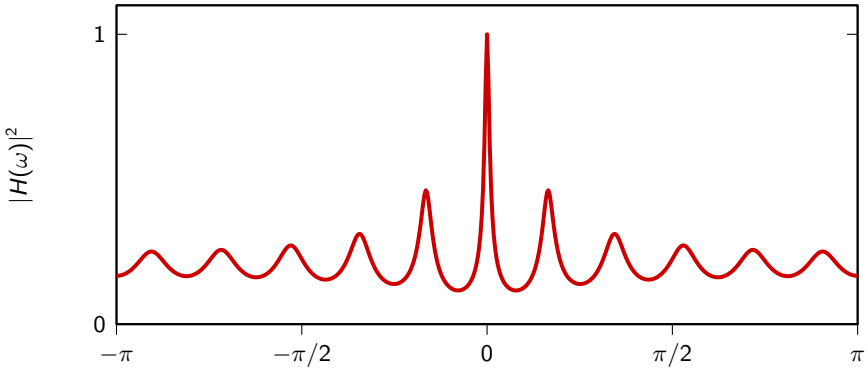


$D = 11, \lambda = 0.6, \alpha = 0.8$

A more realistic echo



A more realistic echo



A more realistic echo

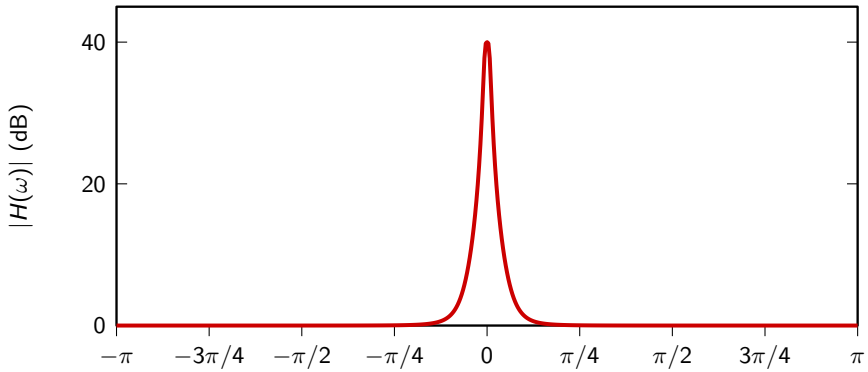
```
class Natural_Echo(RTProcessor):
    def __init__(self, rate, channels):
        super().__init__(rate, channels, max_delay=rate)

        self.a = 0.8
        self.l = 0.7
        self.D = int(0.3 * self.SF)

    def _process(self):
        return x - self.l * self.x.get(1) + \
          self.l * self.y.get(1) + self.a * (1-self.l) * self.y.get(self.D)
```

Shelving filters

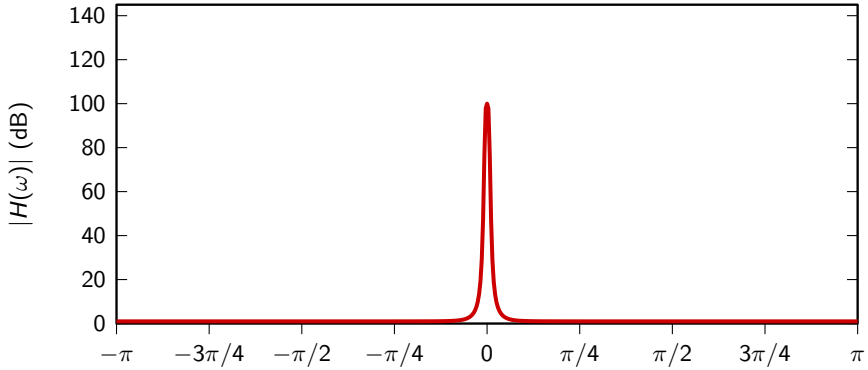
Shelving filters boost a signal's low end or high end

- used in consumer audio appliances (the "Bass" and "Treble" tone knobs)
- high gain at low (or high) frequencies, unit gain elsewhere
- can be implemented with a second-order section
- design parameters:
  - shelf gain G (dB)
  - shelf midpoint frequency f_0 (Hz)

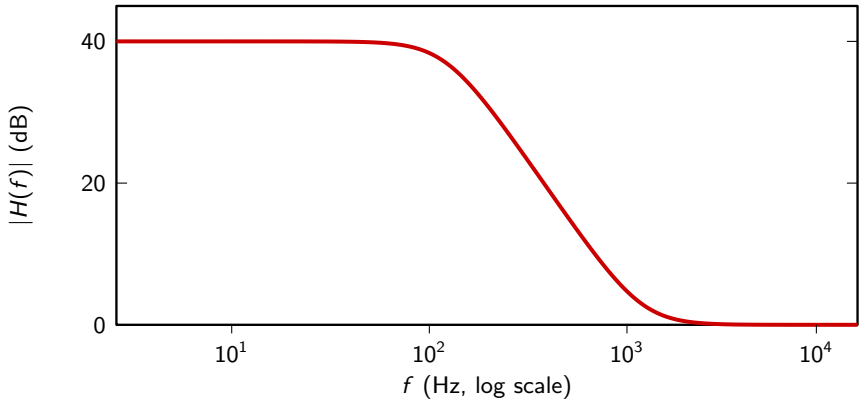
Low shelf



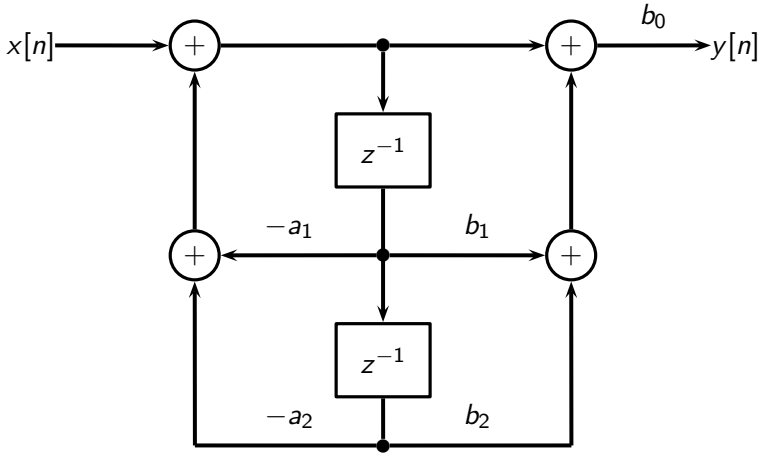
Low shelf, gain in dB



Low shelf, log-log scale, $F_s = 16,000$ Hz



Shelving filter structure



Recipe for low shelf coefficients

$$\omega_0 = 2\pi(f_0/F_s)$$

$$A = 10^{G_{\text{dB}}/40}$$

$$C = (A+1) + (A-1)\cos\omega_0 + \sqrt{2A}\,\sin\omega_0$$

$$a_1 = -2\Big((A-1) + (A+1)\cos\omega_0\Big)/C$$

$$a_2 = \Big((A+1) + (A-1)\cos\omega_0 - \sqrt{2A}\,\sin\omega_0\Big)/C$$

$$b_0 = A\left((A+1) - (A-1)\cos\omega_0 + \sqrt{2A}\,\sin\omega_0\right)/C$$

$$b_1 = 2A\Big((A-1) - (A+1)\cos\omega_0\Big)/C$$

$$b_2 = A\left((A+1) - (A-1)\cos\omega_0 - \sqrt{2A}\,\sin\omega_0\right)/C$$

Low shelf

```
def _process(self, x):
    return self.norm * \
        (self.b0 * x + self.b1 * self.x.get(1) + self.b2 * self.x.get(2)) \
         - self.a1 * self.y.get(1) - self.a2 * self.y.get(2)
```

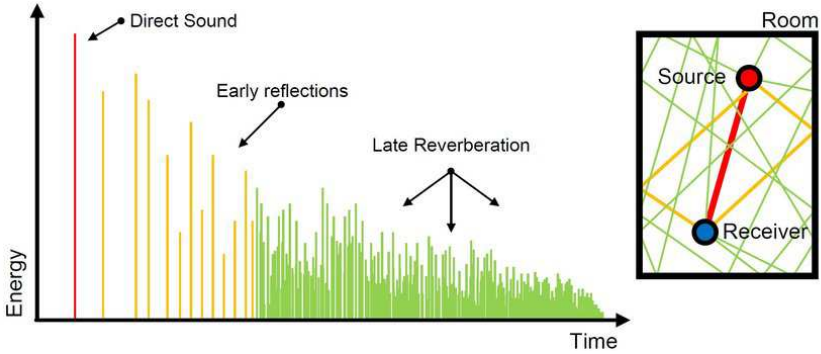
Low shelf

```
def __init__(self, rate, channels, cutoff=300, gain=15):
    super().__init__(rate, channels, max_delay=2)
    Q = 1 / np.sqrt(2)
    w = 2 * np.pi * cutoff / rate
    A = 10 ** (gain / 40)
    alpha = np.sin(w) / (2 * Q)
    c = np.cos(w)
    a0 = (A + 1) + (A - 1) * c + 2 * np.sqrt(A) * alpha
    self.a1, self.a2 = \
        (-2 * ((A - 1) + (A + 1) * c)) / a0, \
        ((A + 1) + (A - 1) * c - 2 * np.sqrt(A) * alpha) / a0
    self.b0, self.b1, self.b2 = \
        (A * ((A + 1) - (A - 1) * c + 2 * np.sqrt(A) * alpha)) / a0,\
        (2 * A * ((A - 1) - (A + 1) * c)) / a0, \
        (A * ((A + 1) - (A - 1) * c - 2 * np.sqrt(A) * alpha)) / a0
    self.norm = .5
```

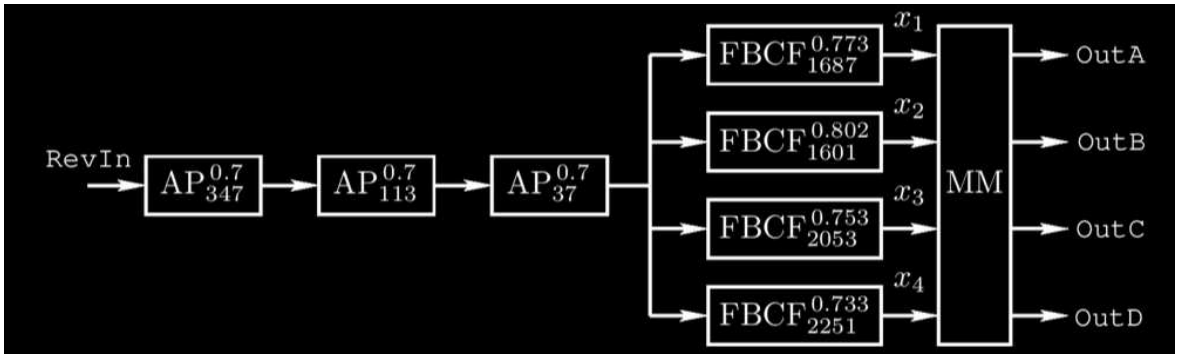
Reverb

- reverb is the cumulative effect of all the multiple reflections of a sound wave in a physical space
- every room has an "impulse response" (RIR) that characterizes the reverberation
- RIRs can be very long (order of seconds in large halls)
- although RIRs can be measured precisely, it's very costly to use RIRs in real time
- synthetic reverberation algorithms exist
- most famous is Schoeder's reverb (1962)

Reverberation



Schroeder's Reverb

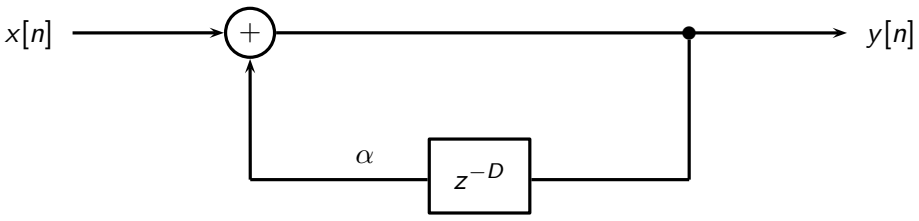


Schroeder's Reverb

- three allpass filters in cascade
- four comb filters in parallel
- output is sum of comb filters
- filter delays are chosen to be coprime

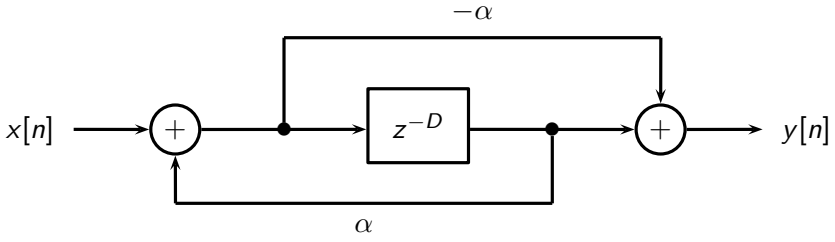
Comb filters

simple recursive echo filters



Allpass filters

$$H(z) = \frac{-\alpha + z^{-D}}{1 - \alpha z^{-D}}$$

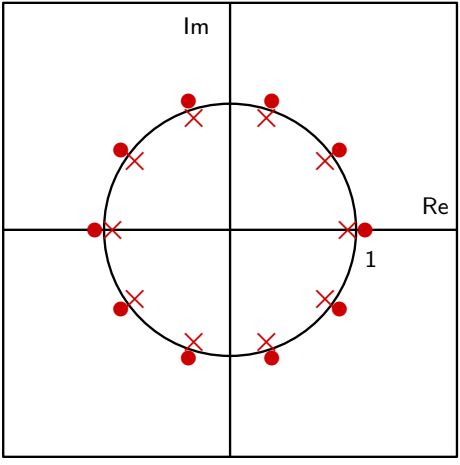


Allpass filters

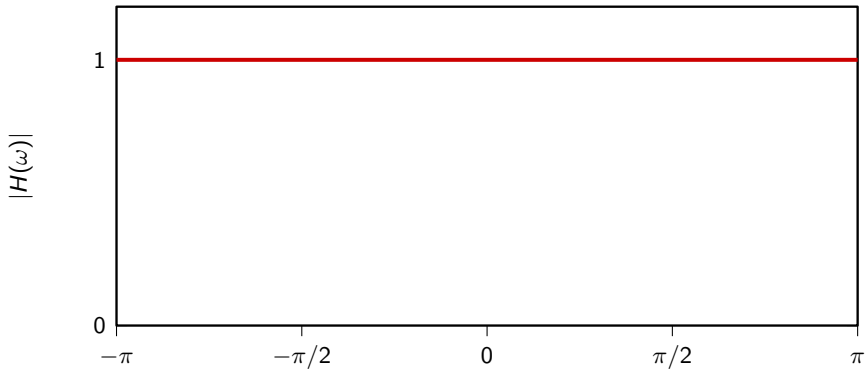
- unit magnitude response: $|H(\omega)| = 1$
- nonlinear phase response
- used to "spread out" short amplitude transients
- impulse response is an exponentially-decaying sequence

$$h[n] = \begin{cases} 0 & n < 0 \\ -\alpha & n = 0 \\ \alpha^k(1-\alpha^2) & n = kD \\ 0 & \text{otherwise} \end{cases}$$

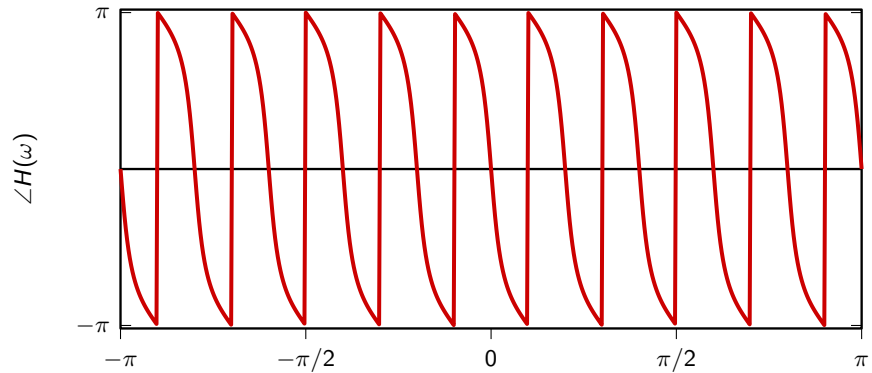
Allpass, poles and zeros ($\alpha = 0.5, N = 10$)



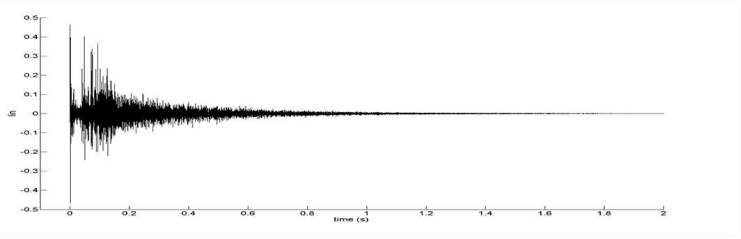
Allpass, magnitude response

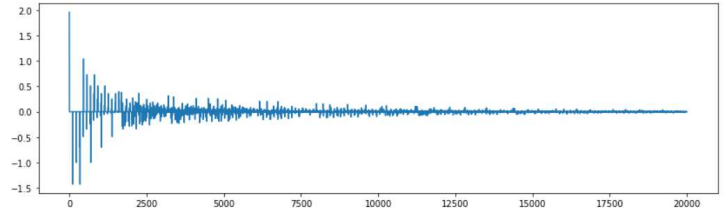


Allpass, phase response



Measured and simulated RIRs





Reverb

```
class Reverb(RTProcessor):
    def __init__(self, rate, channels):
        super().__init__(rate, channels, max_delay=rate)

        self.a = 0.8
        self.norm = 0.5
        self.N = int(0.02 * self.SF)

    def _process(self):
        return self.norm *
          (-self.x.get(0) + self.x.get(self.N) + self.a * self.y.get(self.N))
```

Some non-LTI effects

- distortion (fuzz): clip the signal

$$y[n] = \text{trunc}(ax[n])/a$$

- tremolo: sinusoidal amplitude modulation

$$y[n] = (1 + \cos(\omega_0 n)/G)x[n]$$

- flanger: sinusoidal delay

$$y[n] = (x[n] + x[n - d(n)])/2$$
$$d(n) = \text{round}(M\,(1 - \cos(\omega_0 n)))$$

- wah-wah: time-varying bandpass filter

$$H(z, n) = \frac{(1 - z(n)z^{-1})(1 - z^*(n)z^{-1})}{(1 - p(n)z^{-1})(1 - p^*(n)z^{-1})}$$
$$p(n) = \rho(1 + (\cos\omega_0 n))\, e^{j\theta(1+\cos\omega_1 n)}$$

Fuzz

```
class Fuzz(RTProcessor):
    def __init__(self, rate, channels):
        # memoryless
        super().__init__(rate, channels)

        self.limit = 32767 * 0.01
        self.gain = 10

    def _process(self, x):
        return self.gain * max(min(x, self.limit), -self.limit)
```

Tremolo

```
class Tremolo(RTProcessor):
    def __init__(self, rate, channels):
        super().__init__(rate, channels, max_delay=1)

        self.depth = 0.9
        self.phi = 5 * 2*np.pi / self.SF
        self.omega = 0

    def _process(self, x):
        self.omega += self.phi
        return ((1 - self.depth) + self.depth * 0.5 * (1 + np.cos(self.omega))) * x
```

Flanger

```
class Flanger(RTProcessor):
    def __init__(self, rate, channels):
        super().__init__(rate, channels, max_delay=rate)

        self.maxd = 0.015 * self.SF
        self.phi = 0.2 * 2*np.pi / self.SF
        self.omega = 0
        self.a = 0.6

    def _process(self, x):
        self.omega += self.phi;
        d = int(self.maxd * (1.0 - np.cos(self.omega)))
        return x if d == 0 else self.a * x + (1.0 - self.a) * self.x.get(d)
```

Wah

```
def _process(self, x):
    """ Wah-wah autopedal. A slow oscillator moves the positions of
    the poles in a second-order filter around their nominal value
    The result is a time-varying bandpass filter
    """
    # current angle of the pole
    d = self.pole_delta * (1.0 + np.cos(self.omega)) / 2.0
    self.omega += self.phi

    # recompute the filter's coefficients
    self.b1 = -2.0 * self.zero_mag * np.cos(self.zero_phase + d)
    self.a1 = -2.0 * self.pole_mag * np.cos(self.pole_phase + d)

    return 0.3 * (x + self.b1 * self.x.get(1) + self.b2 * self.x.get(2)) - \
                  self.a1 * self.y.get(1) - self.a2 * self.y.get(2)
```