COM303: Digital Signal Processing

Lecture 14: Real-time signal processing

Summary:

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

Summary:

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

Summary:

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

Summary:

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

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!

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!

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!

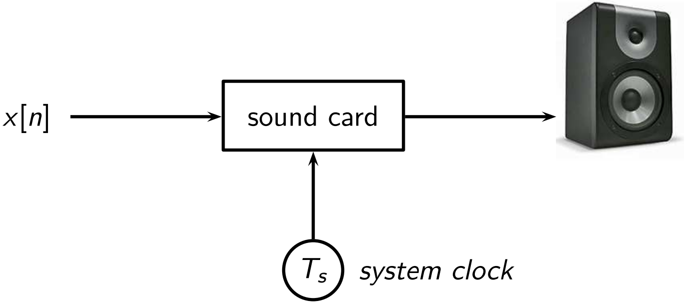
Real-time processing

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

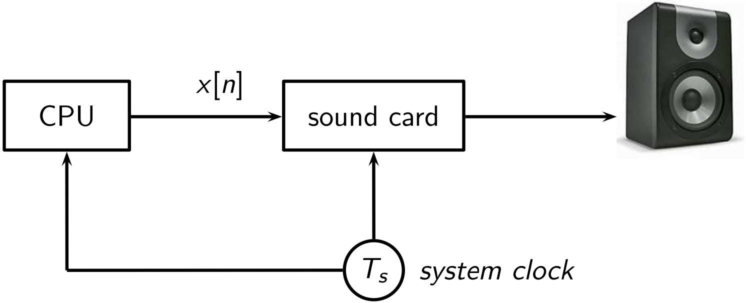
- "record" a value $x_i[n]$
- process the value in a causal filter
- "play" the output $x_o[n]$

everything needs to happen in at most T_s seconds!

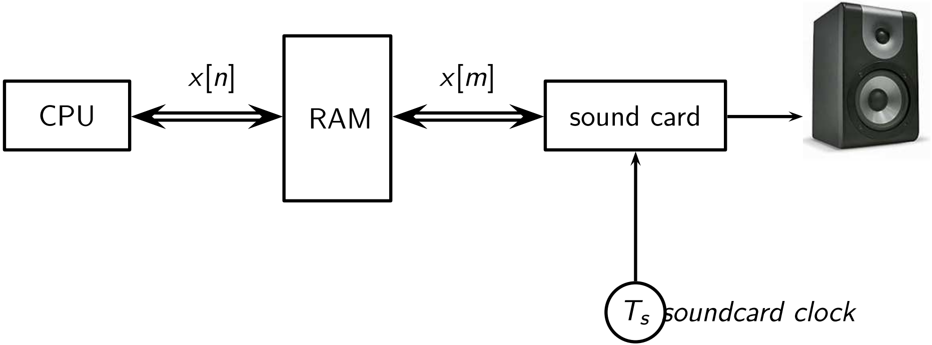
Playing a sound



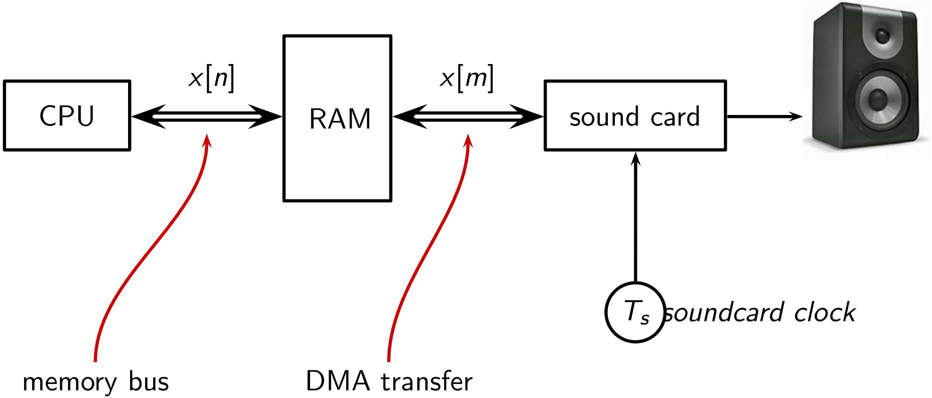
On dedicated hardware...



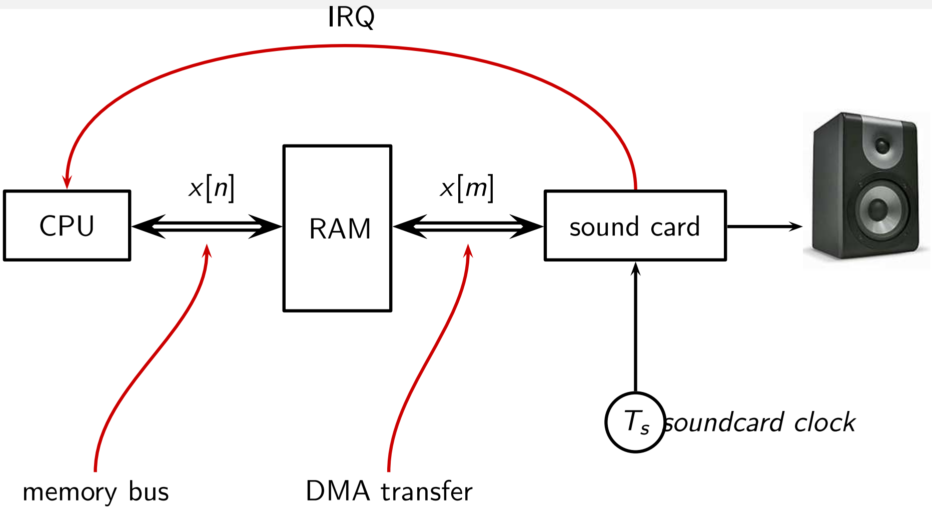
On a PC...



On a PC...


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

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!

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!

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

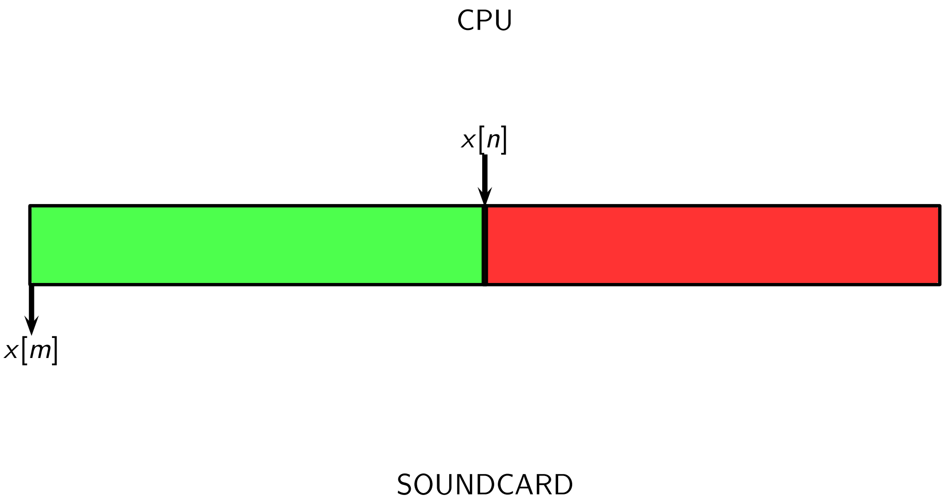
buffering introduces delay!

Buffering

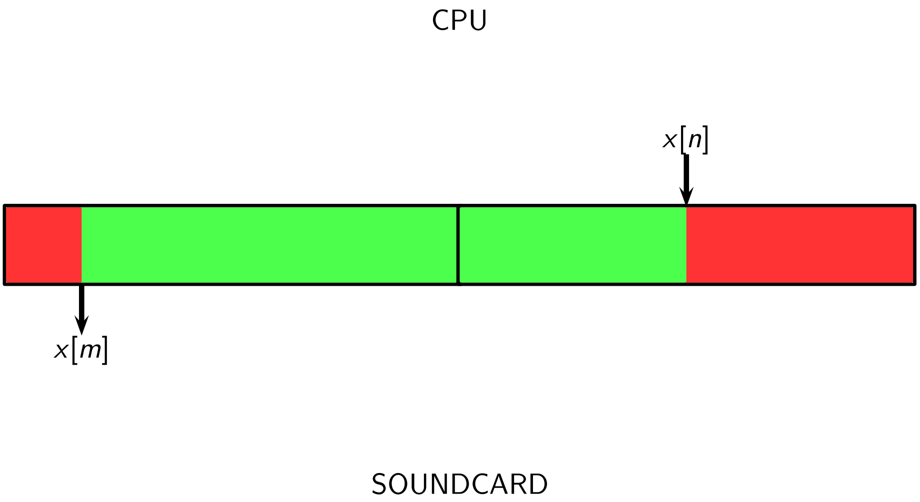
- interrupt for each sample would be too much overhead
- soundcard consumes sample in buffers
- soundcard notifies when buffer used up
- CPU can fill a buffer in less time than soundcard can empty it

buffering introduces delay!

Example: double buffering (output)

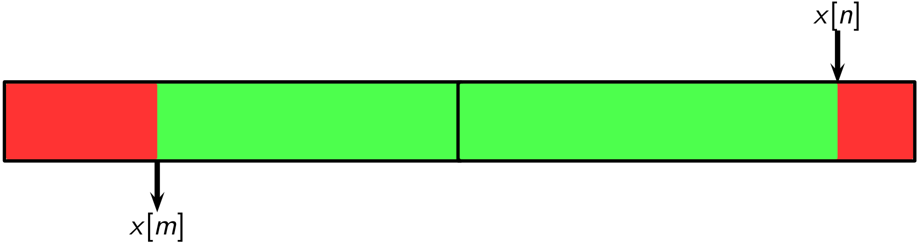


Example: double buffering (output)



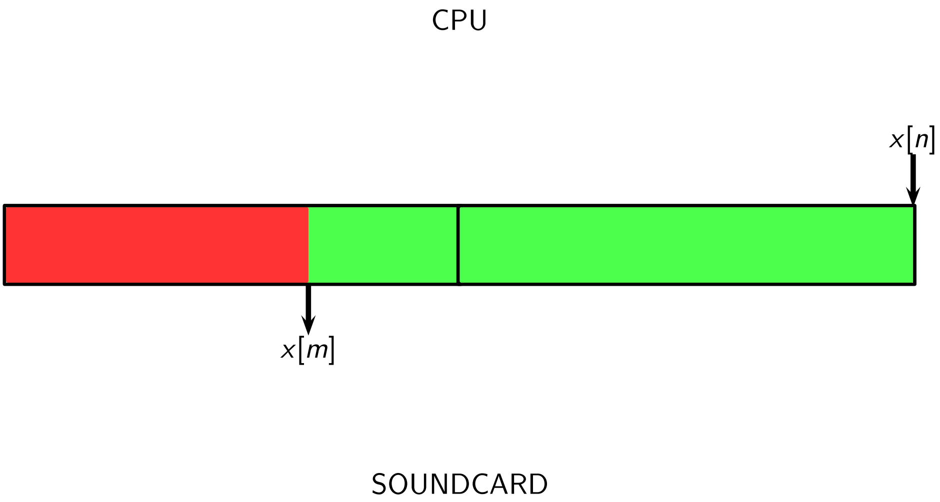
Example: double buffering (output)

CPU

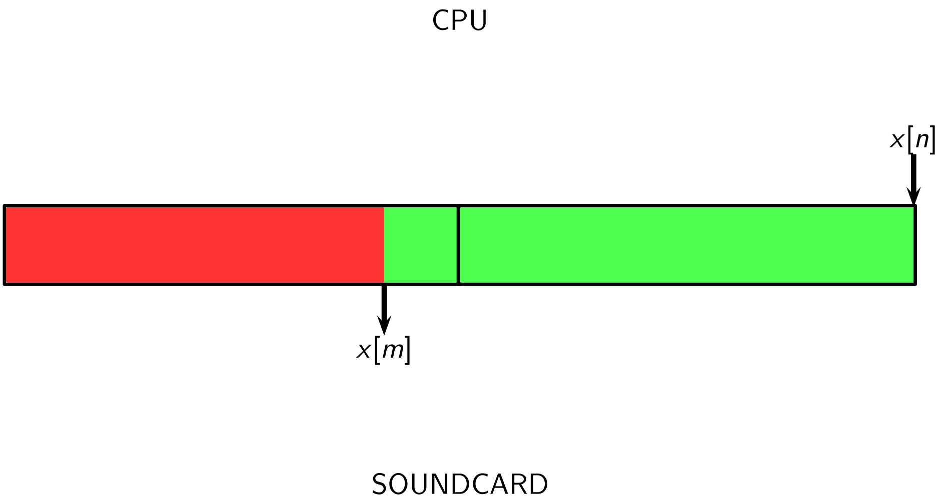


SOUNDCARD

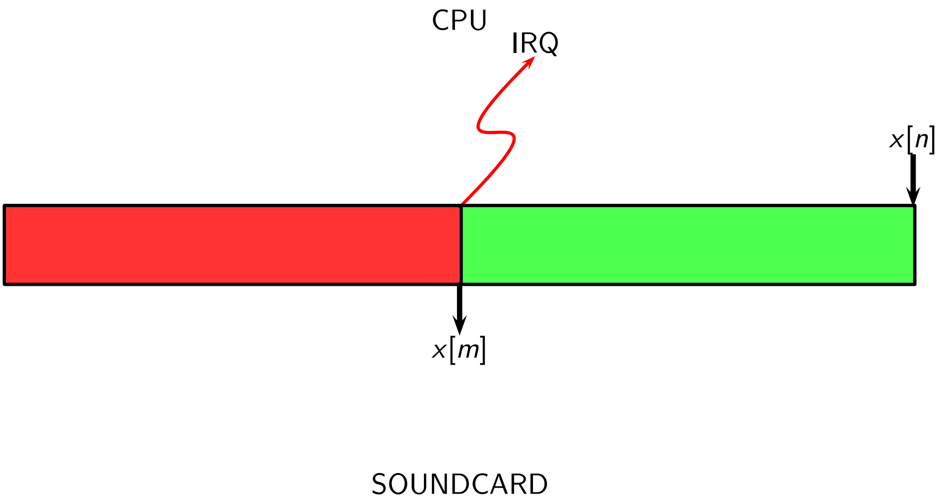
Example: double buffering (output)



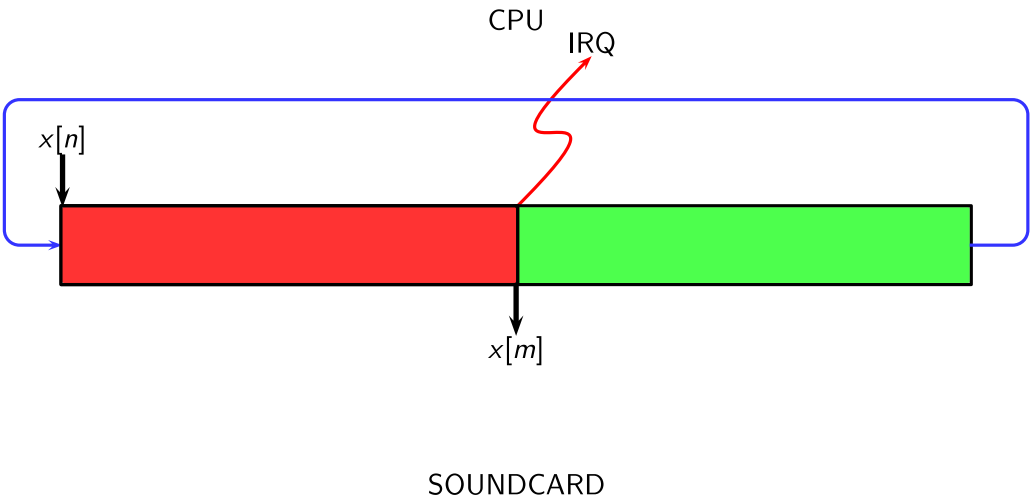
Example: double buffering (output)



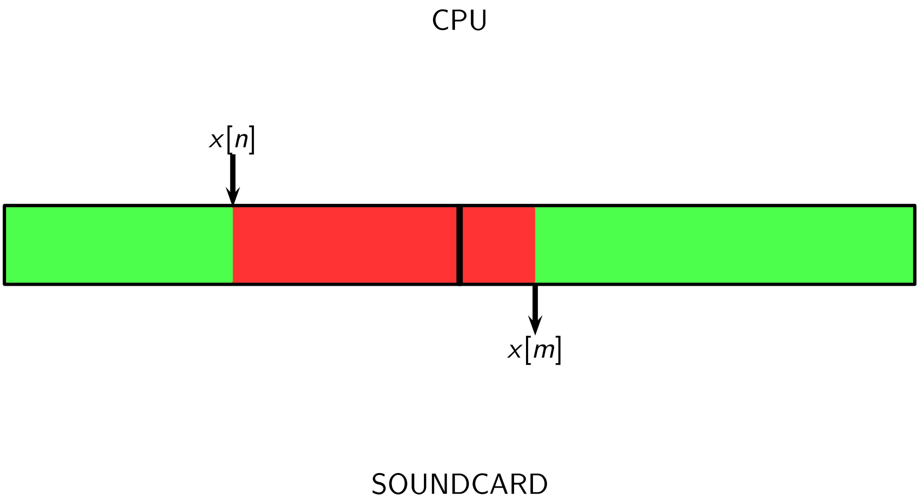
Example: double buffering (output)



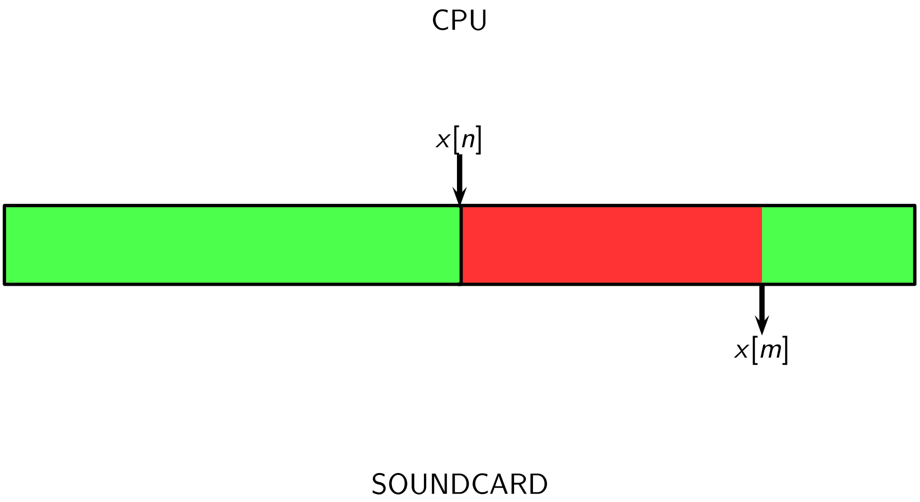
Example: double buffering (output)



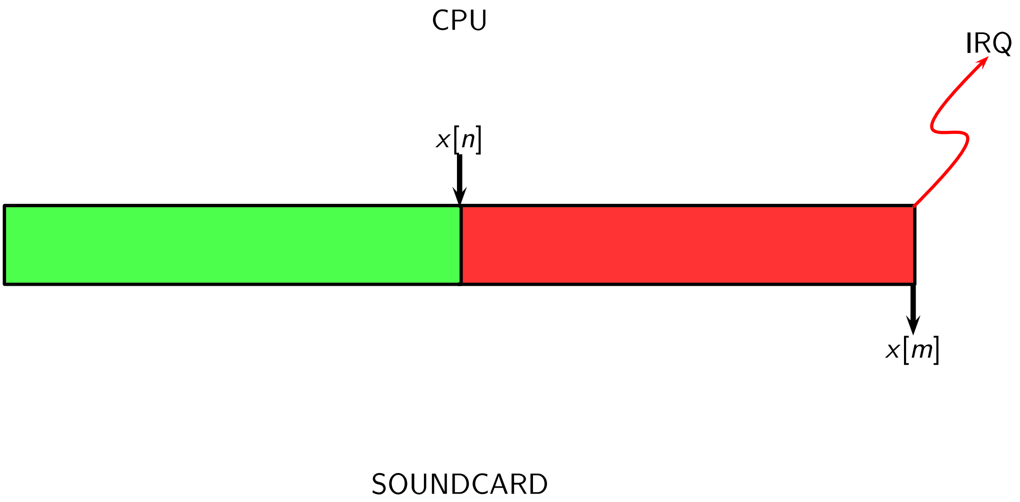
Example: double buffering (output)



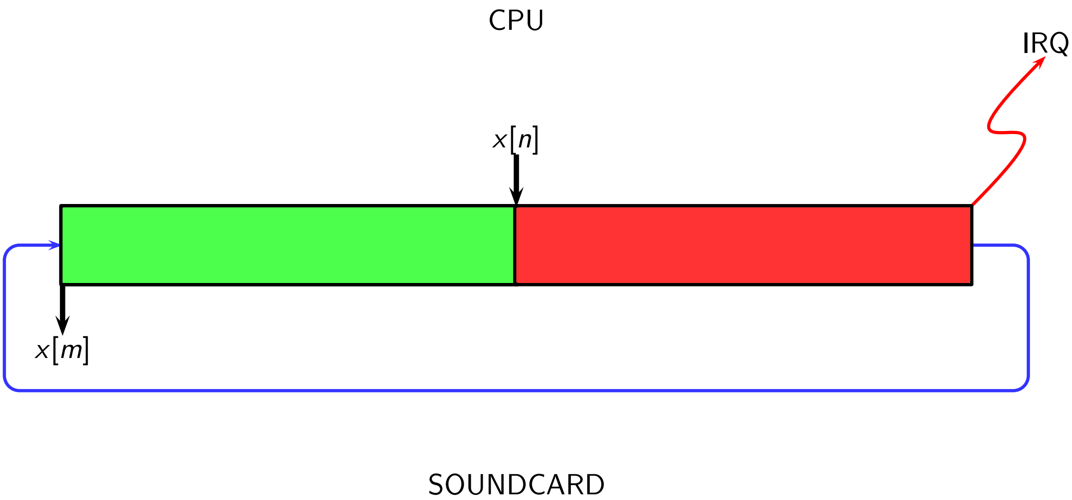
Example: double buffering (output)



Example: double buffering (output)



Example: double buffering (output)



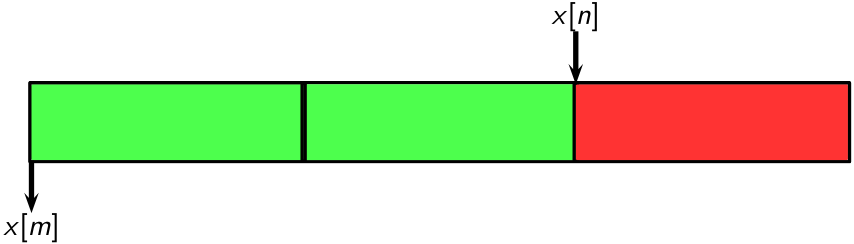
Example: double buffering

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

Example: double buffering

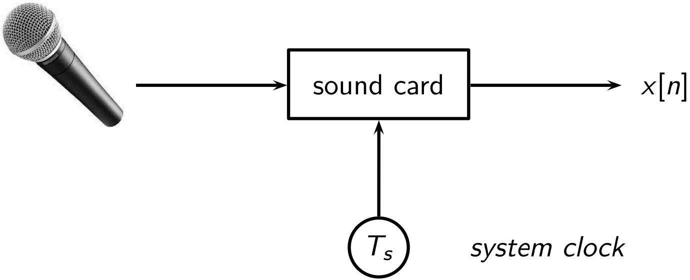
- double buffering introduces a delay $d = T_s \times \frac{L}{2}$ seconds
- if CPU doesn't fill the buffer fast enough: **underflow**

Multiple buffering

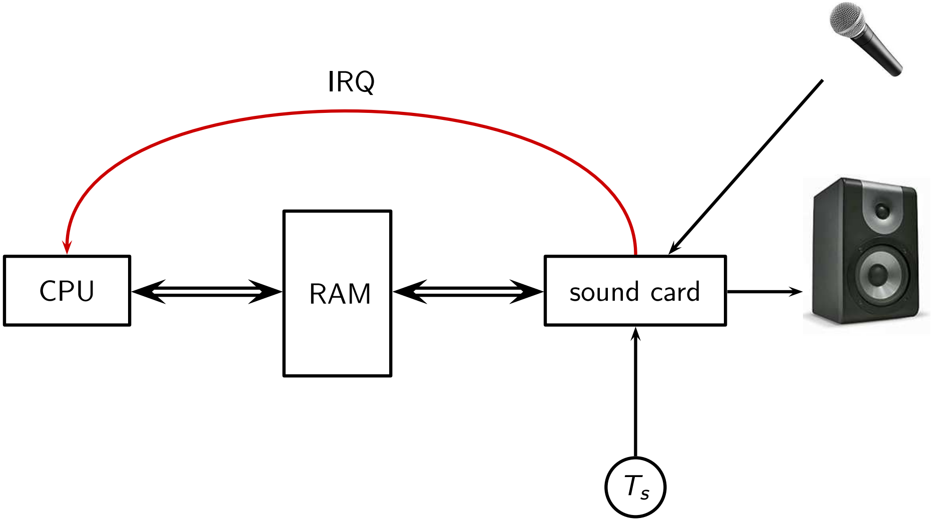


- call the CPU more often (balance load)
- keep reasonable underflow protection

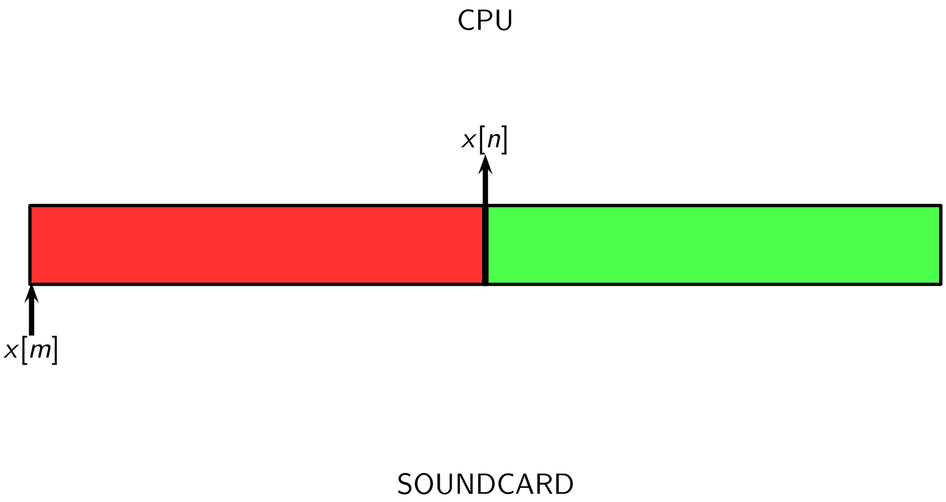
What about the input?



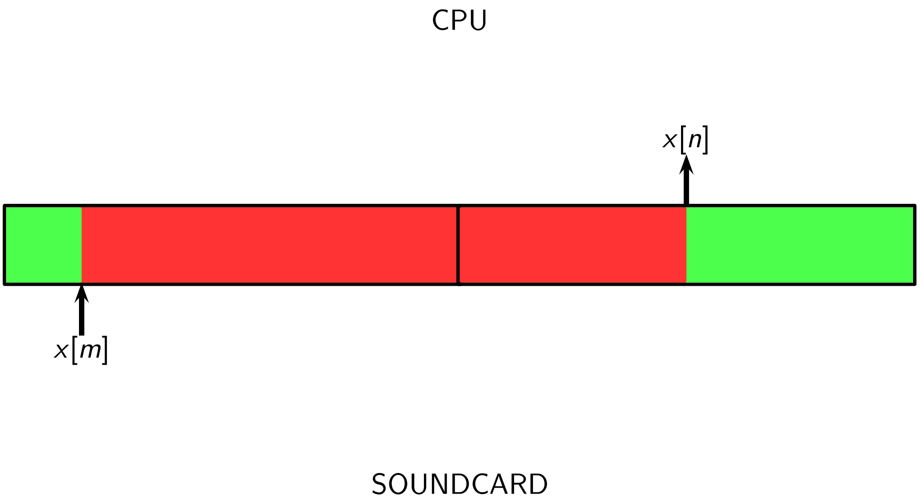
On a PC...



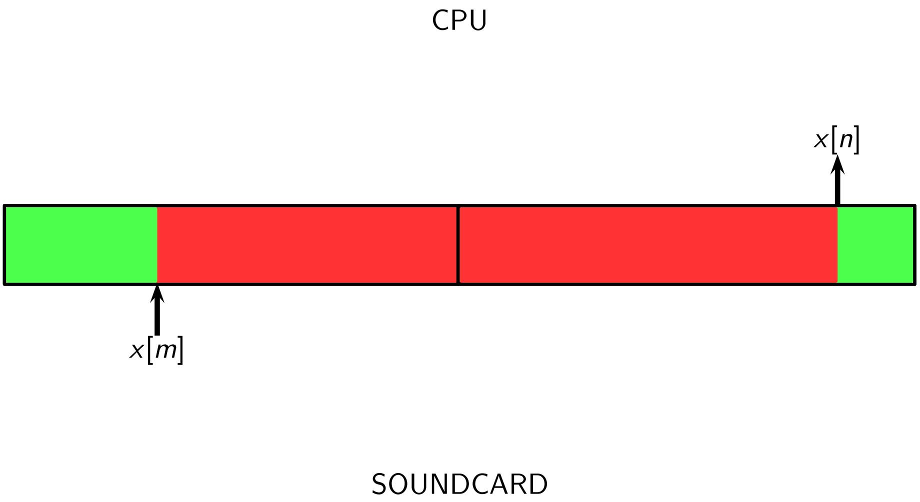
Example: double buffering (input)



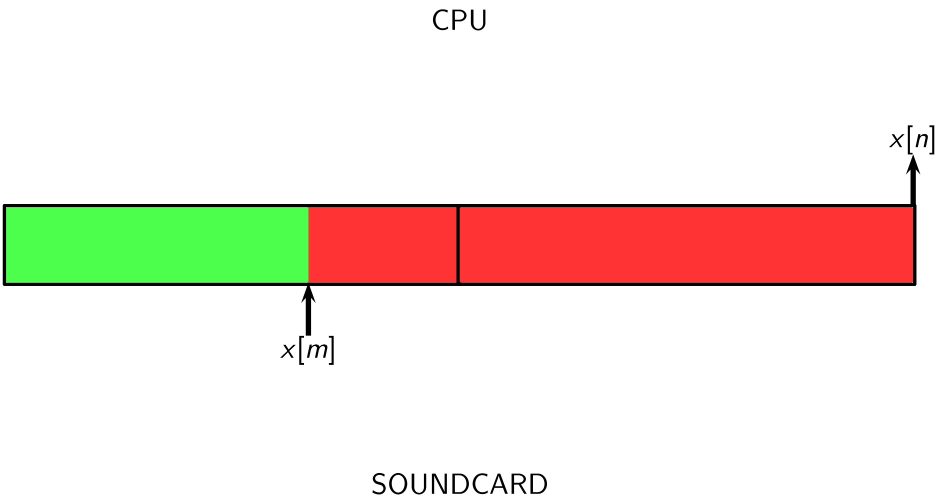
Example: double buffering (input)



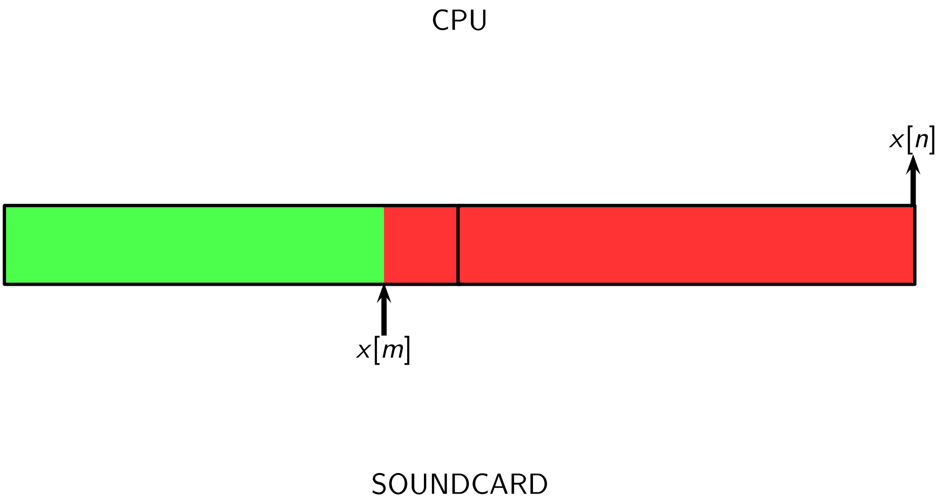
Example: double buffering (input)



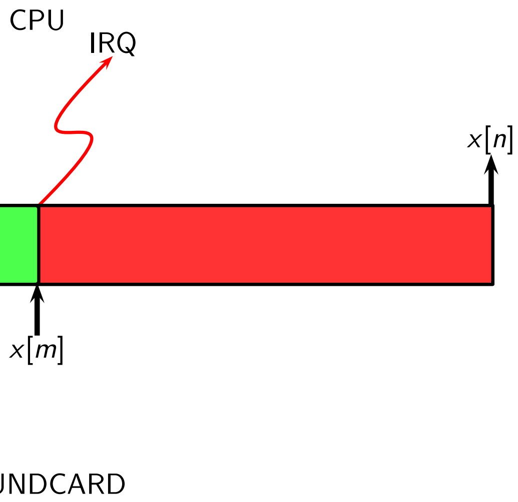
Example: double buffering (input)



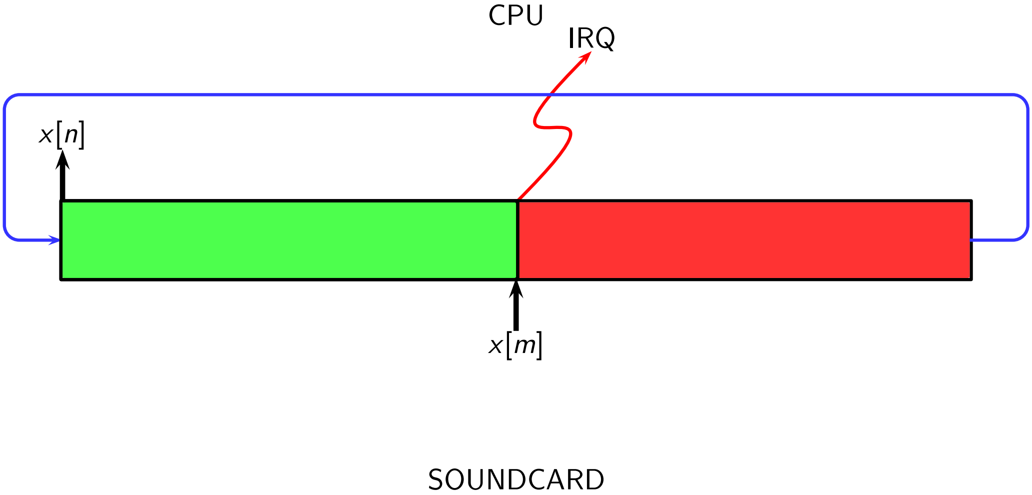
Example: double buffering (input)



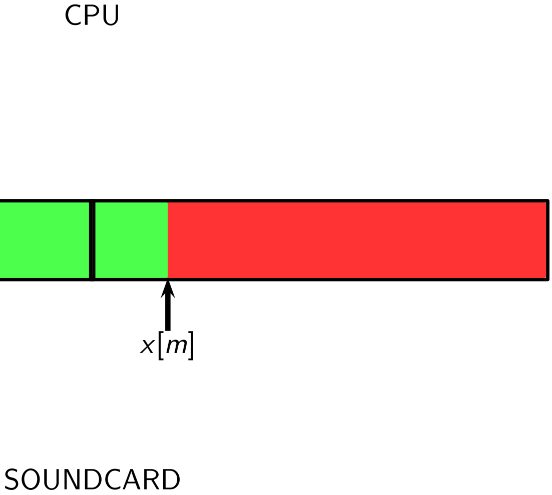
Example: double buffering (input)



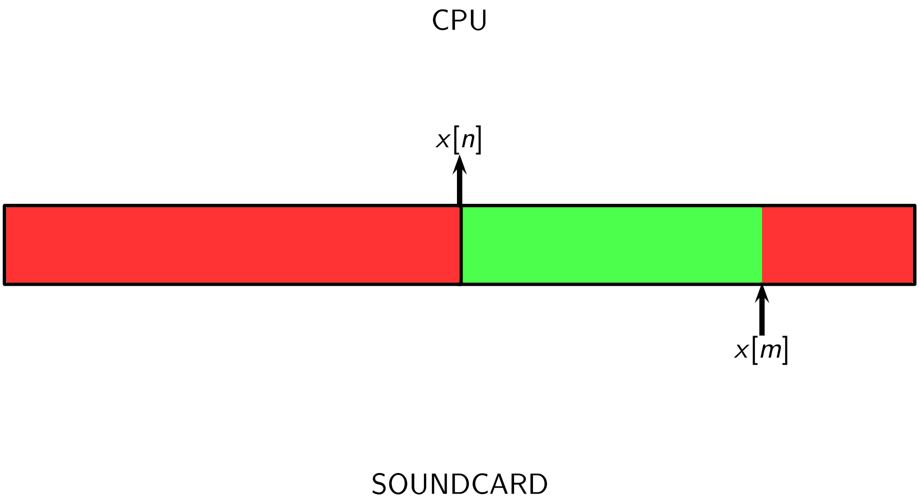
Example: double buffering (input)



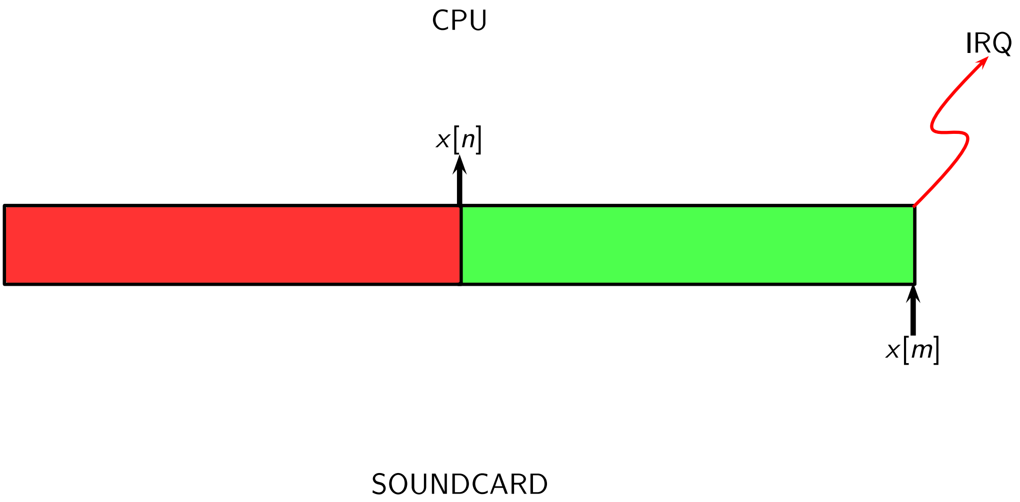
Example: double buffering (input)



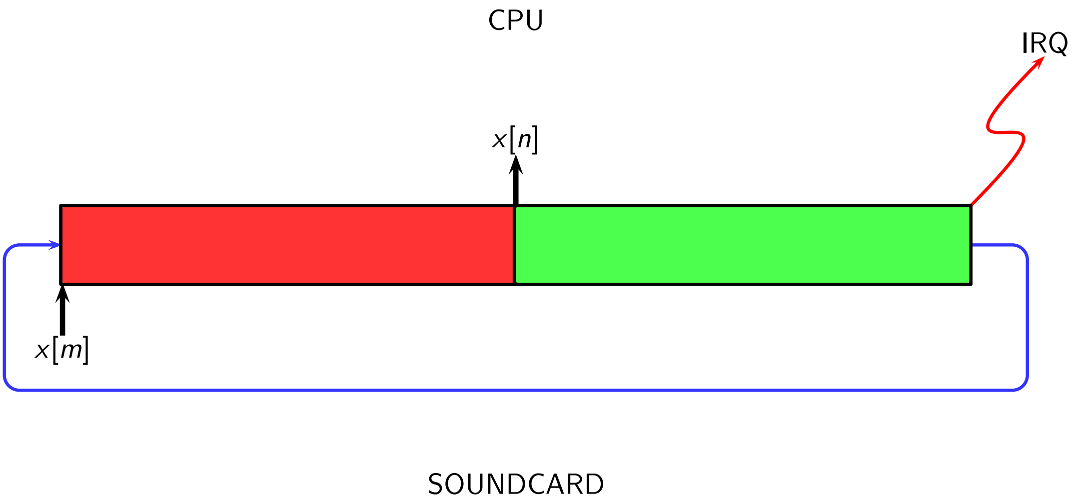
Example: double buffering (input)



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

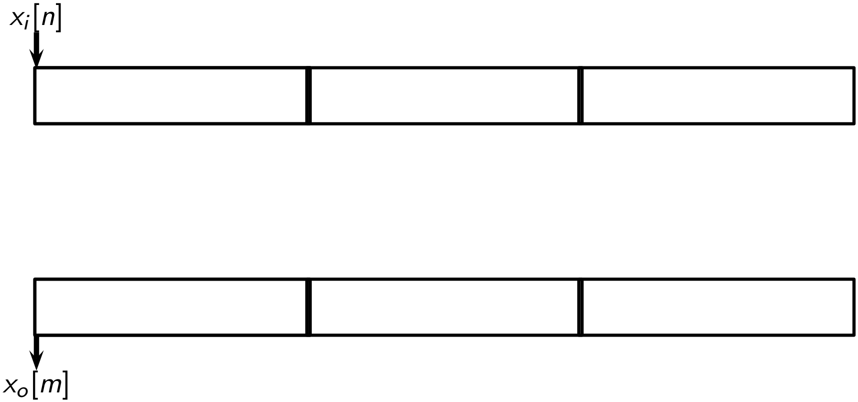
Putting it all together

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

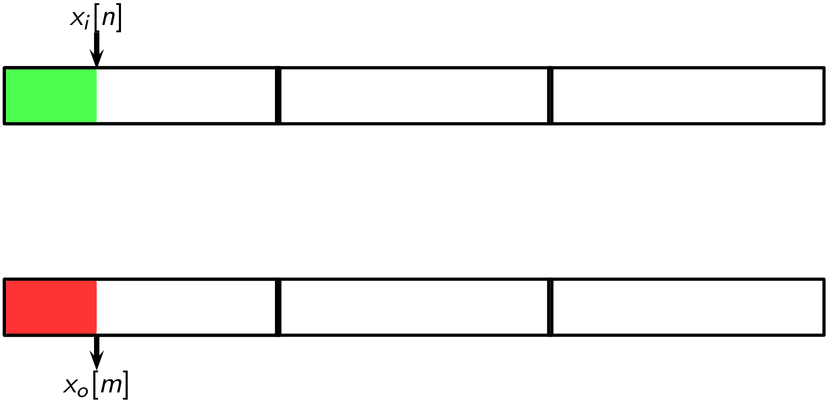
Putting it all together

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

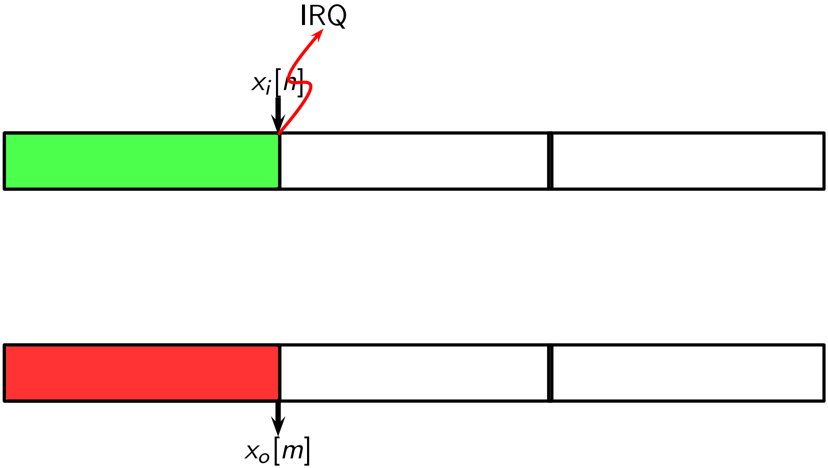
Real-time I/O processing with multiple buffering



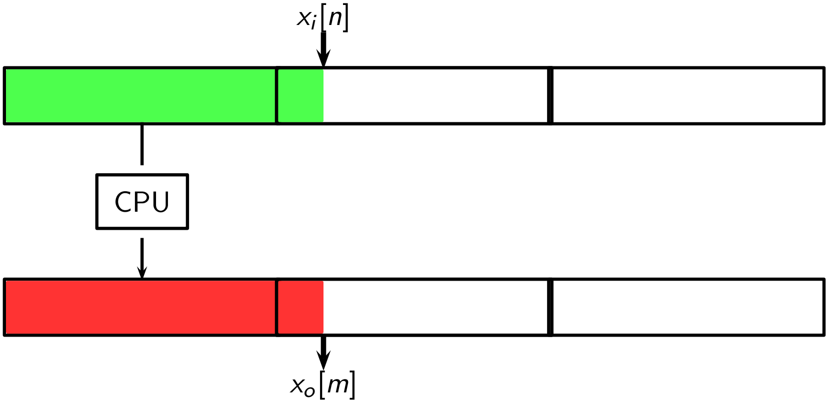
Real-time I/O processing with multiple buffering



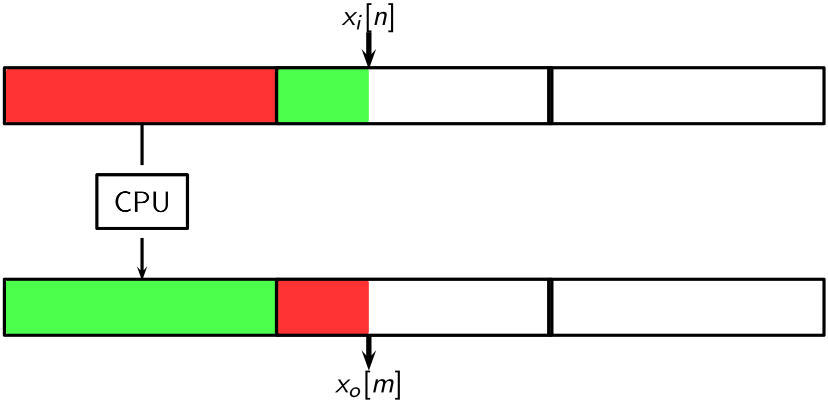
Real-time I/O processing with multiple buffering



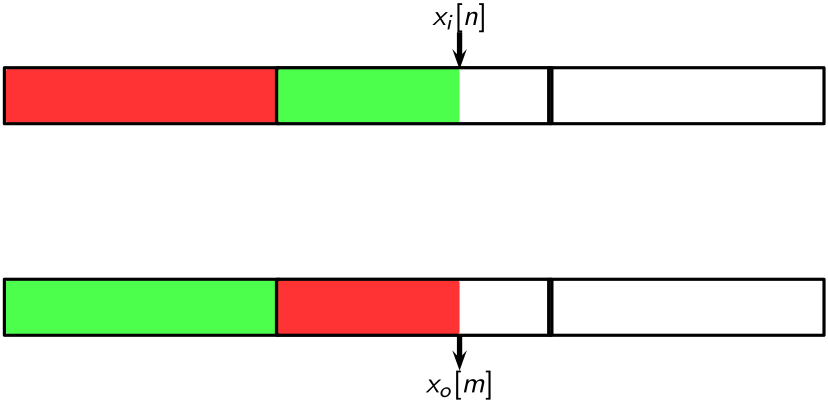
Real-time I/O processing with multiple buffering



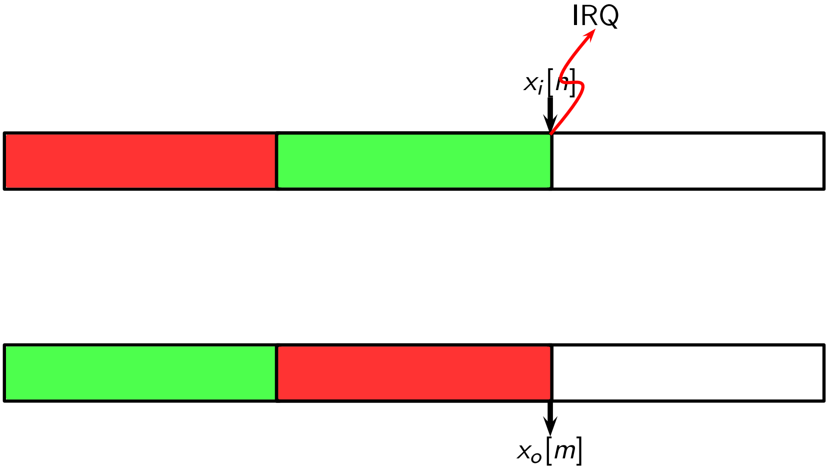
Real-time I/O processing with multiple buffering



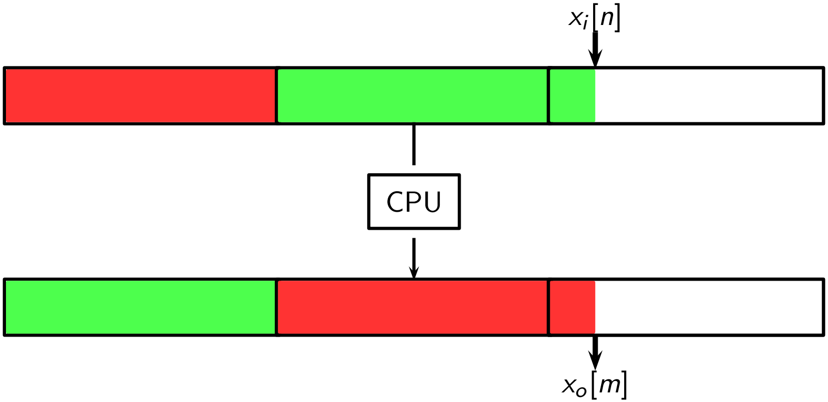
Real-time I/O processing with multiple buffering



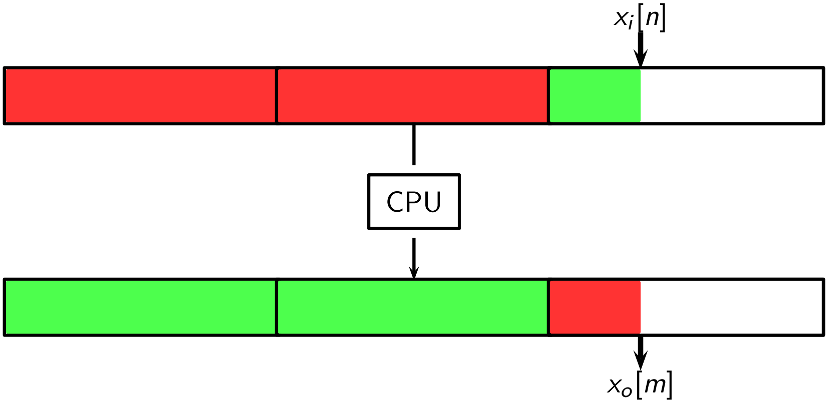
Real-time I/O processing with multiple buffering



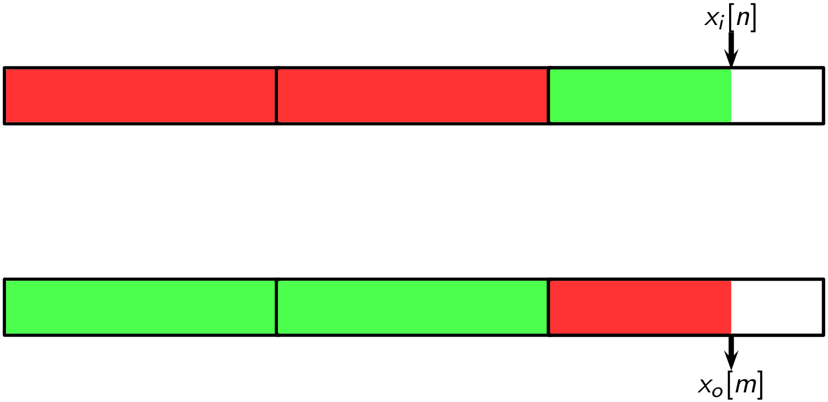
Real-time I/O processing with multiple buffering



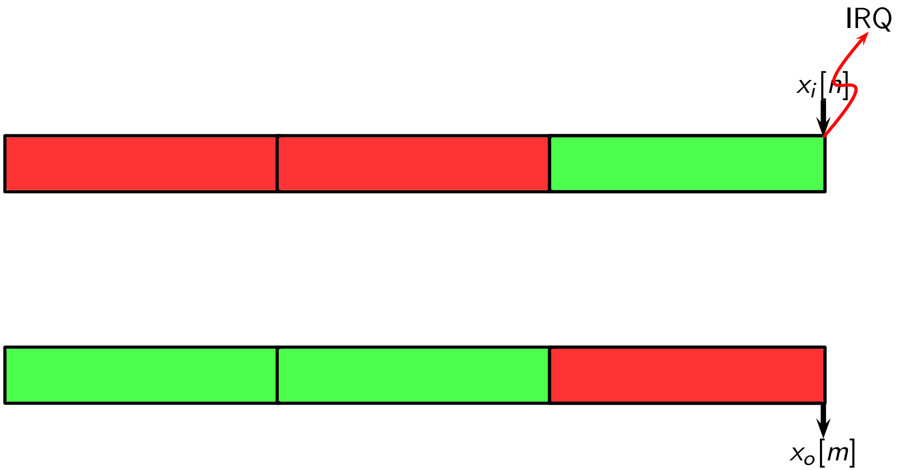
Real-time I/O processing with multiple buffering



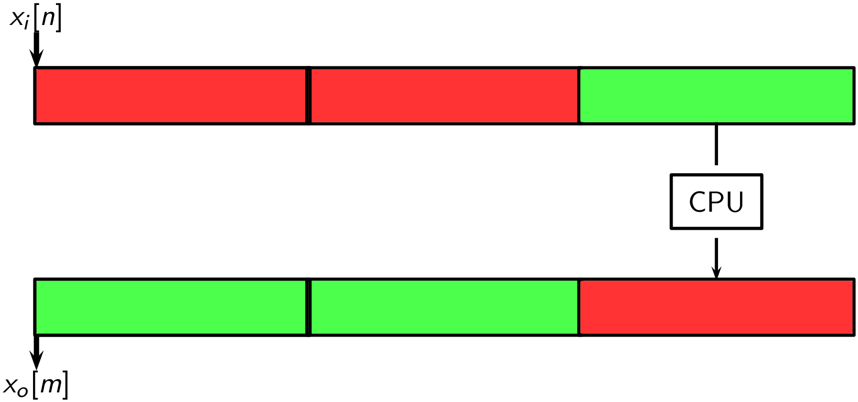
Real-time I/O processing with multiple buffering



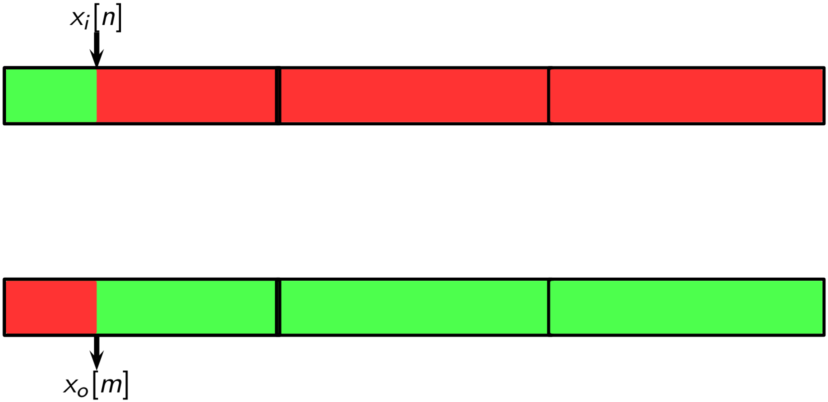
Real-time I/O processing with multiple buffering



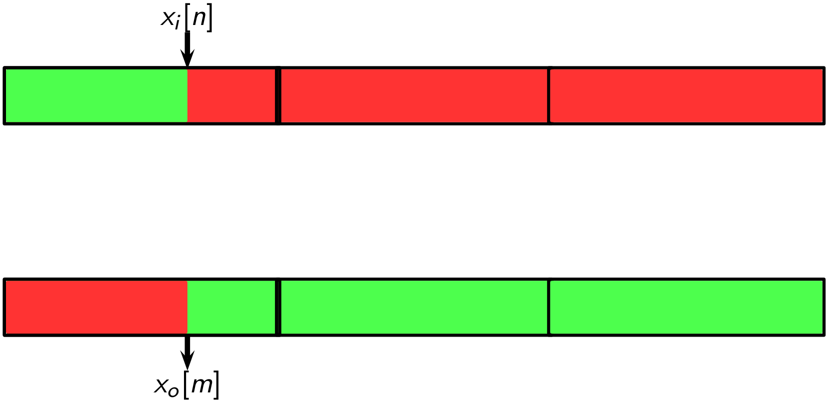
Real-time I/O processing with multiple buffering



Real-time I/O processing with multiple buffering



Real-time I/O processing with multiple buffering



Real-time I/O processing with multiple buffering

- total delay $d = T_s \times L$ seconds
- usually start output process first
- buffers can be collapsed

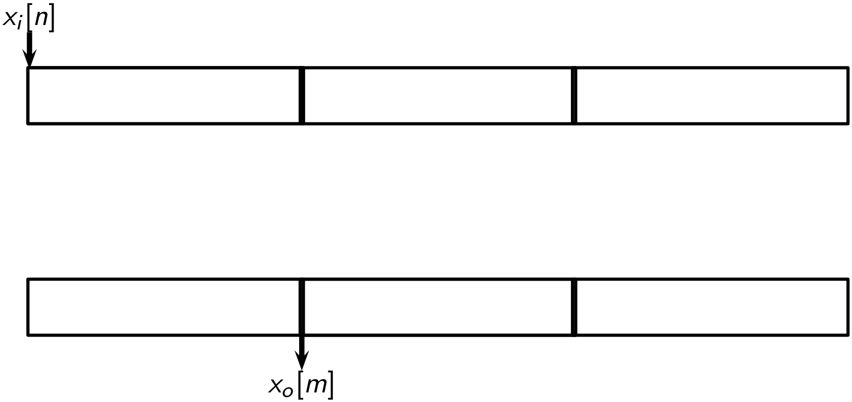
Real-time I/O processing with multiple buffering

- total delay $d = T_s \times L$ seconds
- usually start output process first
- buffers can be collapsed

Real-time I/O processing with multiple buffering

- total delay $d = T_s \times L$ seconds
- usually start output process first
- buffers can be collapsed

Less delay, more risk



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

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

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

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

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

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

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

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

Callback prototype for PortAudio

```
def callback(in_data, ...):
    audio_data = np.fromstring(in_data, dtype=np.int32)
    for n in range(0, len(audio_data)):
        audio_data[n] = np.int32(processor.process(audio_data[n]))
    return audio_data
```

Processing gateway

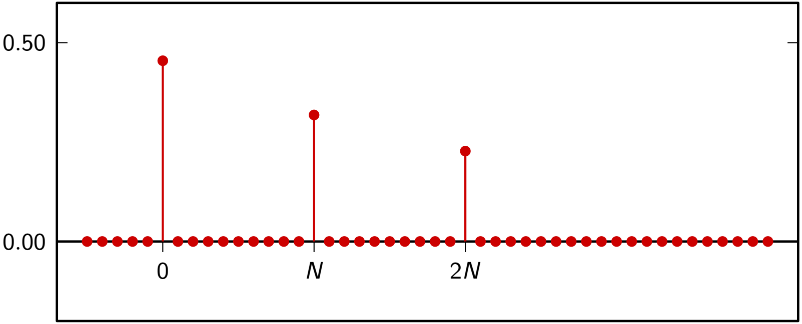
```
class RTProcessor(object):
    def __init__(self, rate, channels=1, max_delay=1):
        self.SF = rate
        self.x = CircularBuffer(max_delay)
        self.y = CircularBuffer(max_delay)

    def process(self, sample):
        self.x.push(sample)
        y = self._process()
        self.y.push(y)
        return y
```

Circular Buffer

```
class CircularBuffer(object):
    def __init__(self, length):
        self.length = length + 1
        self.buf = np.zeros(self.length)
        self.ix = self.length - 1

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

    def get(self, n):
        return self.buf[np.mod(self.ix + self.length - n, self.length)]
```

Simple Echo

$$y[n] = \frac{a\,x[n] + b\,x[n-N] + c\,x[n-2N]}{a+b+c}$$

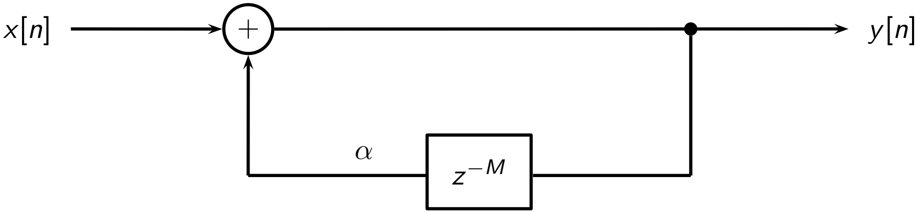


Simple Echo

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

        self.a = 1
        self.b = 0.7
        self.c = 0.5
        self.norm = 1.0 / (self.a + self.b + self.c)
        self.N = int(0.3 * self.SF)

    def _process(self):
         return self.norm * (
             self.a * self.x.get(0) +
             self.b * self.x.get(self.N) +
             self.c * self.x.get(2 * self.N))
```

A better echo

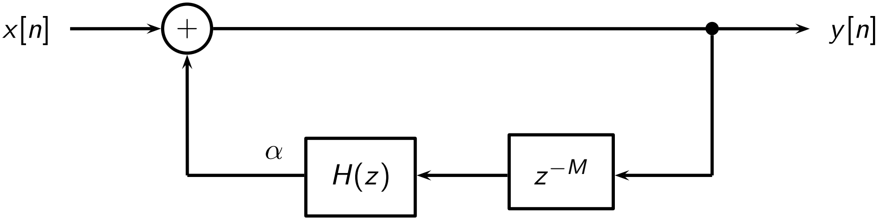
remember the KS algorithm? it's a sort of IIR echo



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

A better echo

a natural echo has a lowpass characteristic

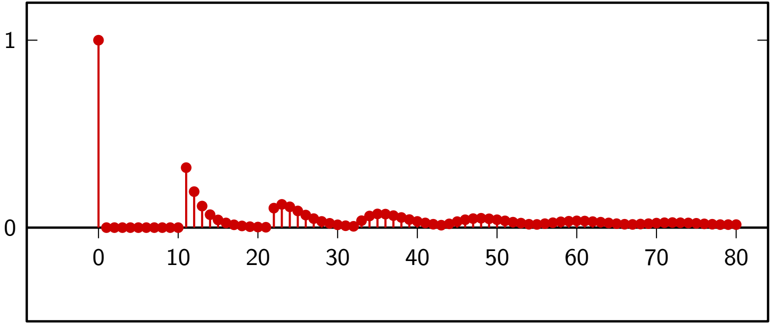


$$y[n] = \alpha(h * y)[n - M] + x[n]$$

A better echo

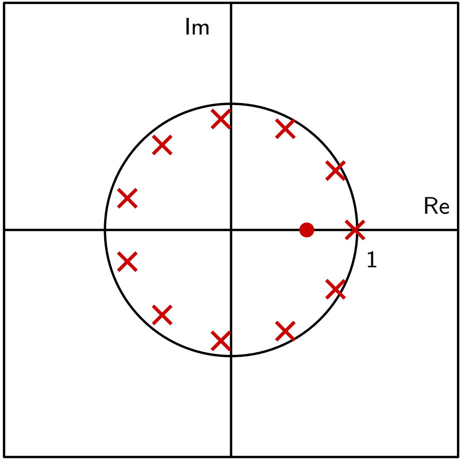
Choose for instance $H(z) =$ leaky integrator:

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

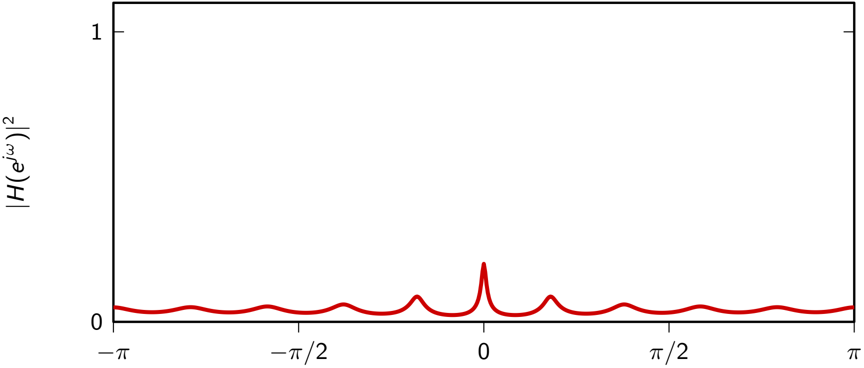


$N = 10, \lambda = 0.6, \alpha = 0.8$

A better echo



A better echo



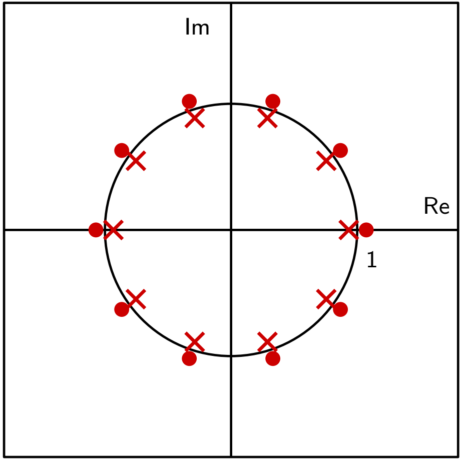
“Natural” Echo

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

        self.a = 0.9
        self.l = 0.8
        self.N = int(0.3 * self.SF)

    def _process(self):
        return self.x.get(0) - self.l * self.x.get(1) + \
          self.l * self.y.get(1) + self.a * (1-self.l) * self.y.get(self.N)
```

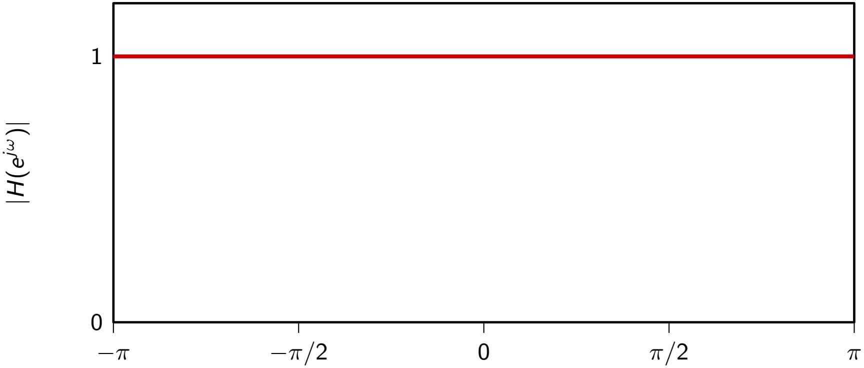
Reverb

- reverb is given by the superposition of many many echos with different delays and magnitudes
- many ways to simulate, always rather costly
- a cheap alternative is to use an allpass filter
$$H(z) = \frac{-\alpha + z^{-N}}{1 - \alpha z^{-N}}$$

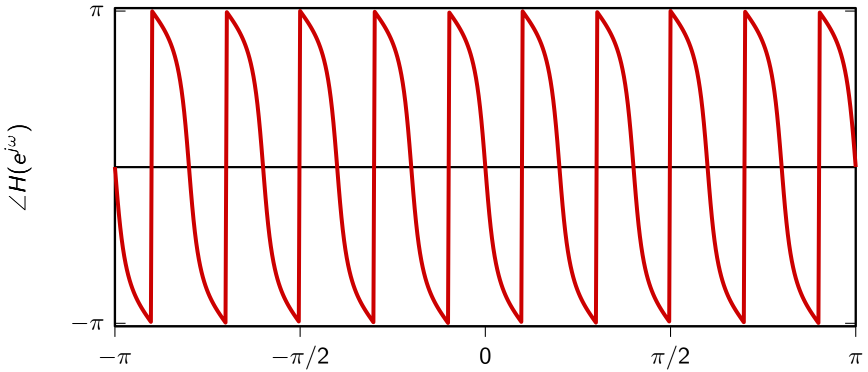
Reverb, poles and zeros ($\alpha = 0.5$, $N = 10$)



Reverb, magnitude response



Reverb, phase response



Reverb

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

$$H(z, n) = \frac{(1 - z(n)z^{-1})(1 - z^*(n)z^{-1})}{(1 - p(n)z^{-1})(1 - p^*(n)z^{-1})}$$

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

Some non-LTI effects

- distortion (fuzz): clip the signal

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

- tremolo: sinusoidal amplitude modulation

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

- flanger: sinusoidal delay

$$y[n] = (x[n] + x[n - d(n)])/2$$

$$d(n) = \mathrm{round}(M(1 - \cos(\omega_0 n)))$$

- wah-wah: time-varying bandpass filter

$$H(z, n) = \frac{(1 - z(n)z^{-1})(1 - z^*(n)z^{-1})}{(1 - p(n)z^{-1})(1 - p^*(n)z^{-1})}$$

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

Some non-LTI effects

- distortion (fuzz): clip the signal

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

- tremolo: sinusoidal amplitude modulation

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

- flanger: sinusoidal delay

$$y[n] = (x[n] + x[n - d(n)])/2$$

$$d(n) = \mathrm{round}(M(1 - \cos(\omega_0 n)))$$

- wah-wah: time-varying bandpass filter

$$H(z, n) = \frac{(1 - z(n)z^{-1})(1 - z^*(n)z^{-1})}{(1 - p(n)z^{-1})(1 - p^*(n)z^{-1})}$$

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

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(Fuzz, self).__init__(rate, channels)

        self.T = 0.005
        self.G = 5
        self.limit = 0x7FFFFFFF * self.T

    def _process(self):
        y = self.x.get(0)
        if (y > self.limit):
            y = self.limit
        if (y < -self.limit):
            y = -self.limit
        return self.G * y
```

Tremolo

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

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

    def _process(self):
        self.omega += self.phi;
        return ((1.0 - self.depth) +
          self.depth * 0.5 * (1 + np.cos(self.omega))) * self.x.get(0)
```

Flanger

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

        self.maxd = 0.008 * self.SF
        self.phi = 0.2 * 2*np.pi / self.SF
        self.omega = 0
        self.a = 0.6

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

Wah

```
def _process(self):
    """ 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 *
      (self.x.get(0) + self.b1 * self.x.get(1) + self.b2 * self.x.get(2) - \
       self.a1 * self.y.get(1) - self.a2 * self.y.get(2))
```