



Course material

Course:

Digital Signal Processing II

Video:

2.2.5.b Real-time processing

Concepts (extracted from automatically generated subtitles):

Sound card. Real time. Signal processing. Output sample. Output process. Write pointer. New buffer. Later use. Data transfer. Form of samples. Zero-filled. Maximum time. Entirety of the first half of the buffer. Buffer. Sound card clock.



[to video sequence search](#)
(within Digital Signal Processing II.)



[to video](#)

Center for Digital Education. More educational support material here:

<https://exoset.epfl.ch/booc/>

Summary:



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

We will first examine how data transfer takes place between your PC and peripherals such as the sound card. We will talk about multiple buffering, which is a technique to ensure smooth data communication between the CPU and the external devices. We will look at the general implementation framework from the point of view of the API. And then we will look at how to implement some guitar effects in real time.

notes

summary

0m 0s



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!

When we talk about real time signal processing, we indicate a situation where the maximum time that we can spend to process one sample is indicated by a system clock of period T_s . We haven't really talked about sampling interpolation. We will do that in the next module.

notes

summary

0m 25s



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!

But you already know that to process a real world signal, we will have to record that in the form of samples. And we will record a sample every T_s seconds. Then we will process this sample in a causal filter because we are in a real time situation.

notes

summary

0m 41s



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!

And the maximum amount of time that we can spend on the processing of one sample is again T_s seconds. And then every T_s seconds we will have to play out the output sample.

notes

summary

0m 54s



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!

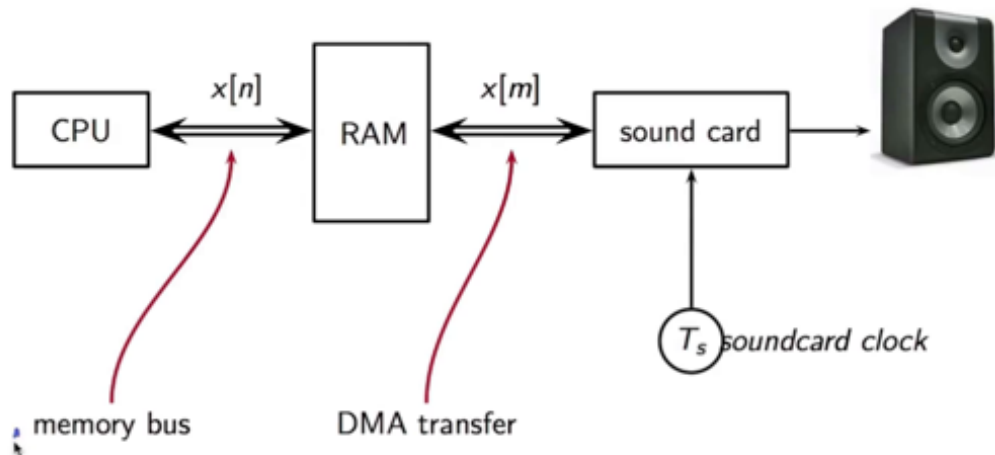
So everything needs to happen in most T_s seconds. Let's look in more detail at the output process.

notes

summary

1m 5s





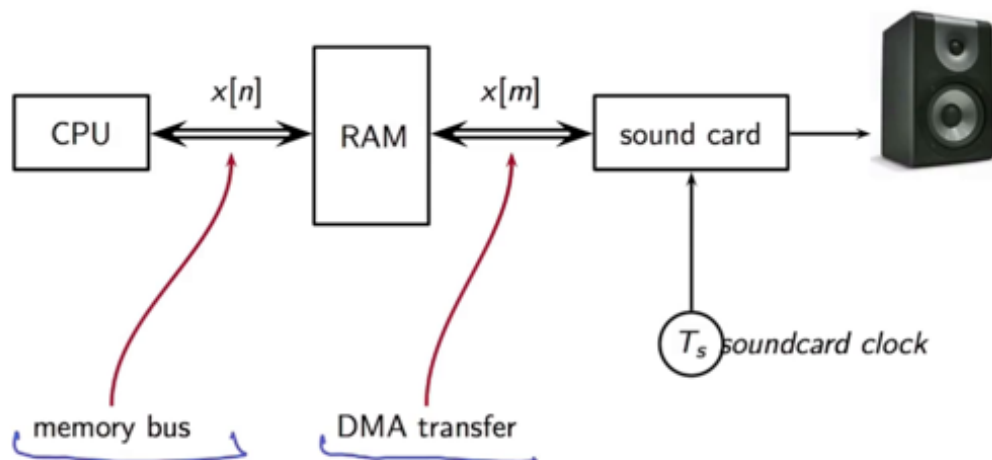
We have a series of samples, X of N . We have a sound card, which is the device that will bridge our digital world to the analog world of the loudspeaker. And the sound card operates with a system clock of T_s . Now let's assume that our sequence X of N is generated by some algorithm by a central processing unit. On a dedicated audio device, we could use the same system clock to drive both the production of sample and the output process. So the system would be rather simple. Now on a PC on the other hand, the CPU works on a clock that is completely independent of the sound card clock. As a matter of fact, it is a clock that is much, much faster than the clock you would find in your sound card. So the situation is the following. The CPU will generate samples at its own pace and put these samples in memory. The sound card, on the other hand, will fetch the samples from memory at its own pace, which is dictated by its clock, and then output them to the loudspeaker. The transfer of data from the CPU to memory happens via a standard memory bus transaction.

notes

summary

1m 14s





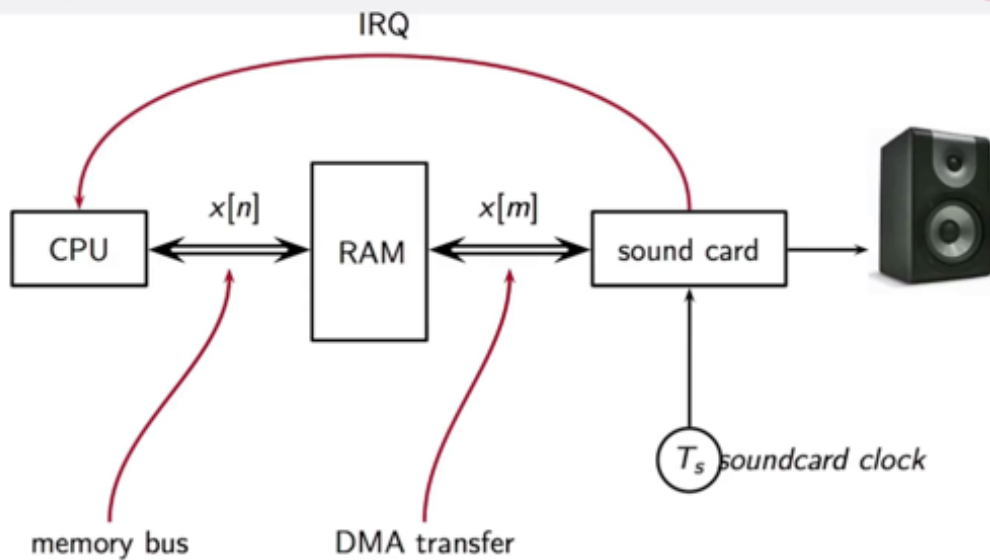
The transfer of data from memory to the sound card takes place in the form of a DMA transfer. DMA stands for direct memory access.

notes

summary

2m 22s





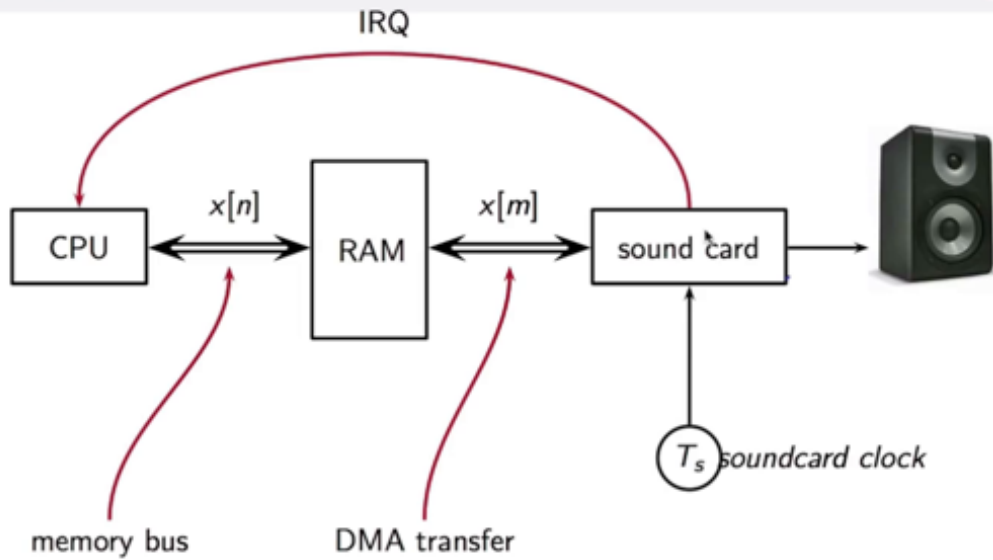
In other words, the sound card does not need to invoke the CPU in order to access the data in memory. And so this transfer and this transfer, they can happen asynchronously. The problem, of course, when you have to a synchronous process is how you have them communicate.

notes

summary

2m 30s





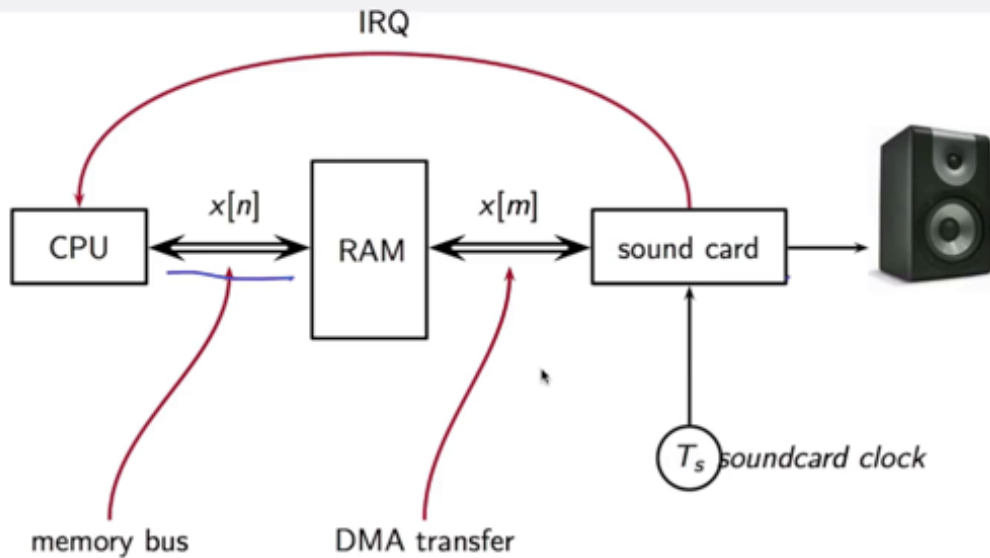
In particular, how does the sound card tell the CPU that more data is needed? This problem is solved by having the CPU issue an IRQ, an interrupt request to the CPU.

notes

summary

2m 49s





And when the CPU receives the signal, it will put another batch of data in memory for the sound card to access. Now, if we were to interrupt the CPU every time we need a new sample, there would be too much overhead.

notes

summary

2m 57s



- ▶ 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!

So the idea is that the sound card will consume a chunk of samples already in memory called a buffer.

notes

summary

3m 9s



- ▶ 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!

The sound card will notify when the buffer is completely used up, and the CPU will fill a new buffer for the sound card to use.

notes

summary

3m 18s



- ▶ 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!

The key point here, since we're talking about real-time signal processing, is that the time it takes for the CPU to fill up a buffer needs to be strictly less than the time it takes for the sound card to consume that buffer.

notes

summary

3m 27s



- ▶ 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!

Of course, the buffering operation introduces a delay, but this delay is a price we need to pay on a general purpose architecture because we cannot guarantee that the CPU will always be available exactly at the time when we need a new sample. So by processing the data in batches, we insulate the data generation process against concurrent demands on the CPU.

notes

summary

3m 37s



- ▶ 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!

Let's look in more detail how this works. This is an example of double buffering.

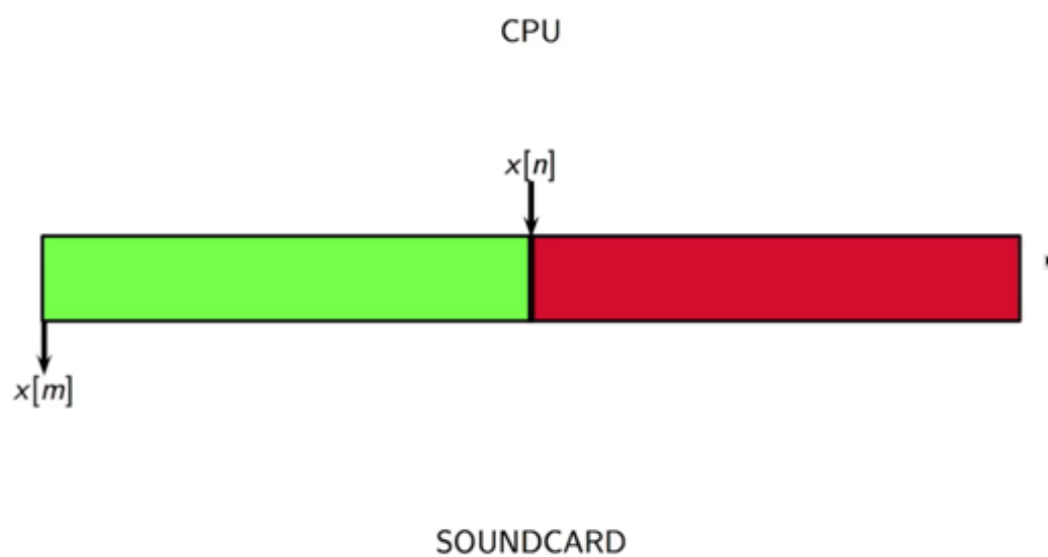
notes

summary

4m 0s



Example: double buffering (output)



The buffer is the strip here, which is a set of contiguous locations in RAM, say L locations for L samples.

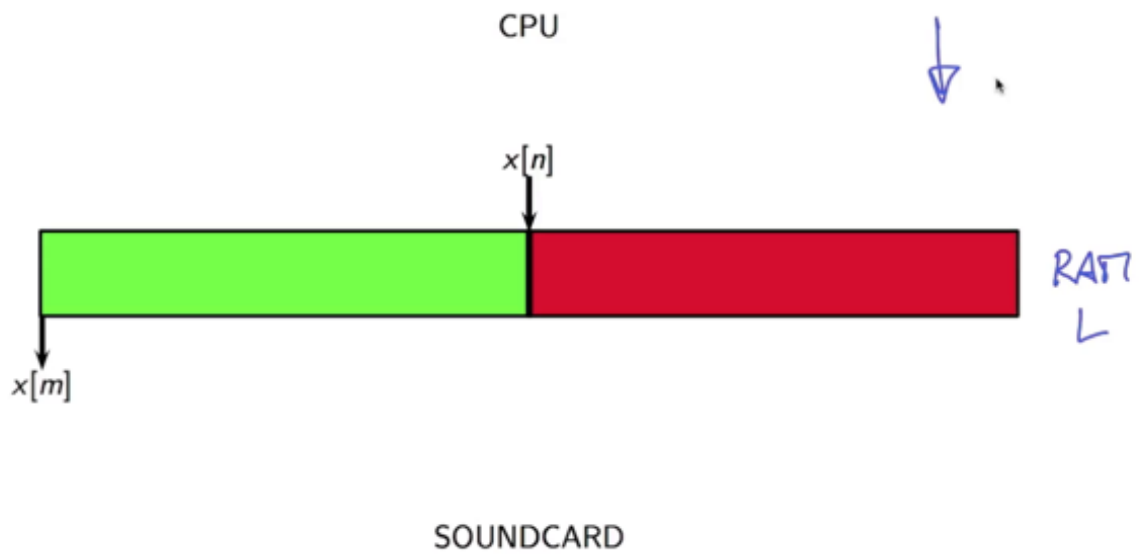
notes

summary

4m 6s



Example: double buffering (output)



The CPU will write the newly computed samples into this buffer, and the sound card will fetch the samples from this buffer for playing.

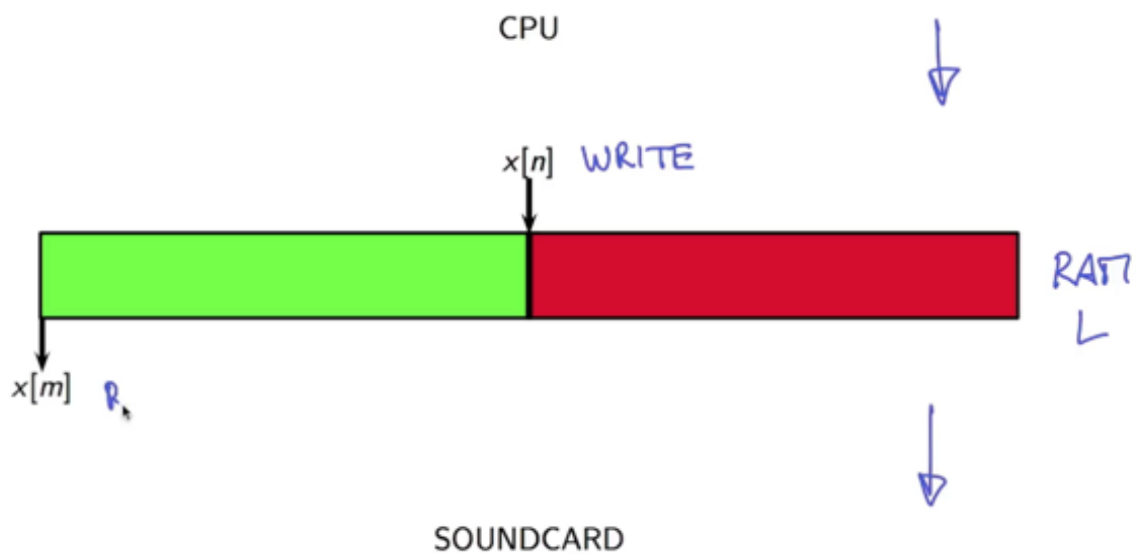
notes

summary

4m 14s



Example: double buffering (output)



There are two pointers into this buffer. There is a write pointer used by the CPU to identify the next memory location into which write a new sample. And there's a read pointer used by the sound card to identify the location from which the next sample has to be fetched.

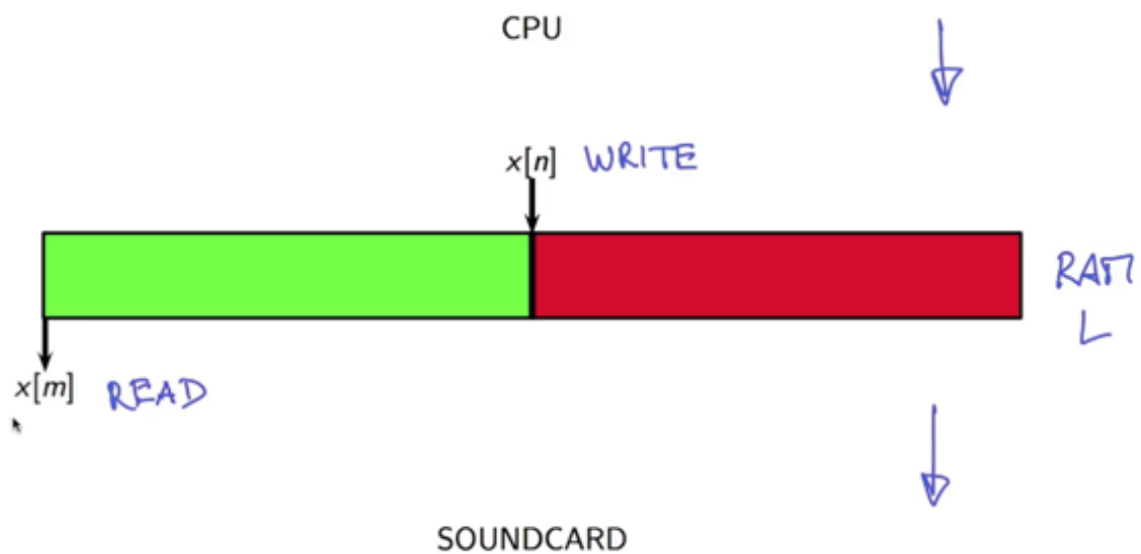
notes

summary

4m 25s



Example: double buffering (output)



The pointers are initialized as such. We assume that the buffer has been filled with zero, say, in the beginning. And we position the read pointer at the beginning of the buffer and the write pointer at the midpoint of the buffer.

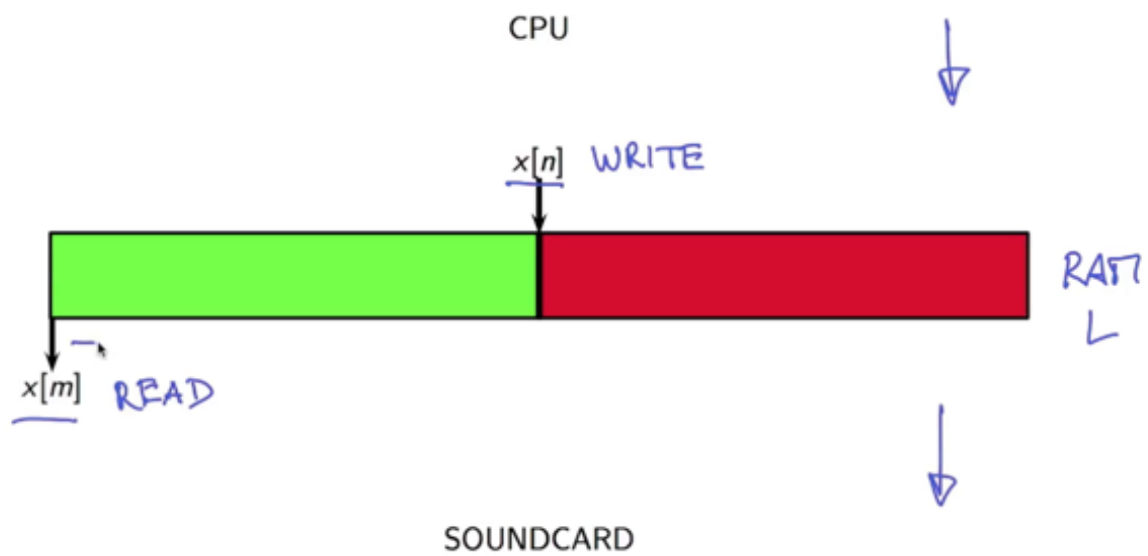
notes

summary

4m 43s



Example: double buffering (output)



And now the dance begins. The sound card is started, and the sound card will start fetching samples from the read buffer and move the read pointer in this direction.

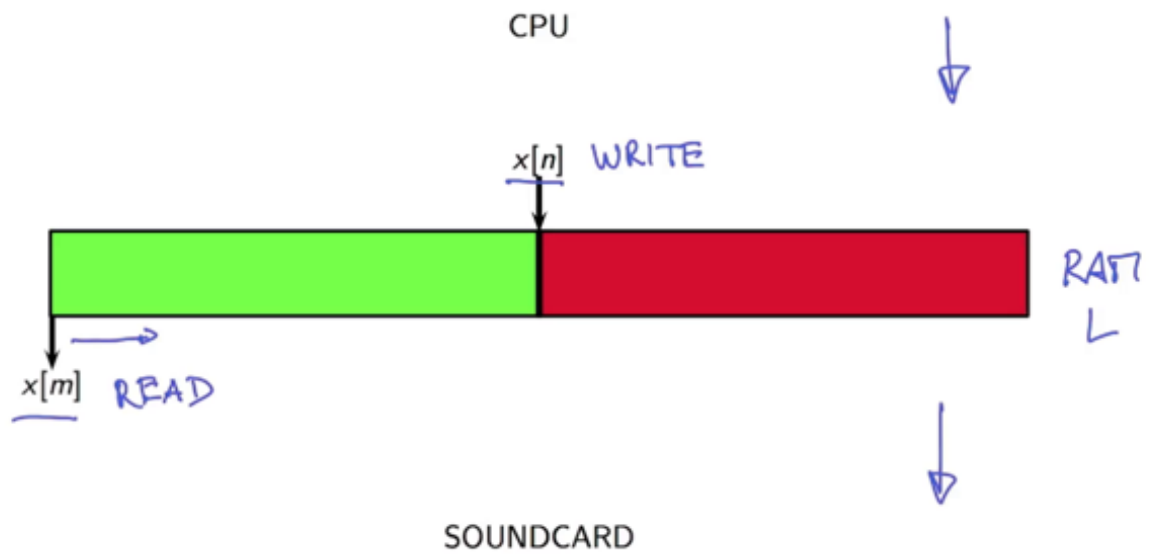
notes

summary

4m 57s



Example: double buffering (output)



At the same time, the CPU will start writing samples into the buffer, and it will move the write pointer in this direction.

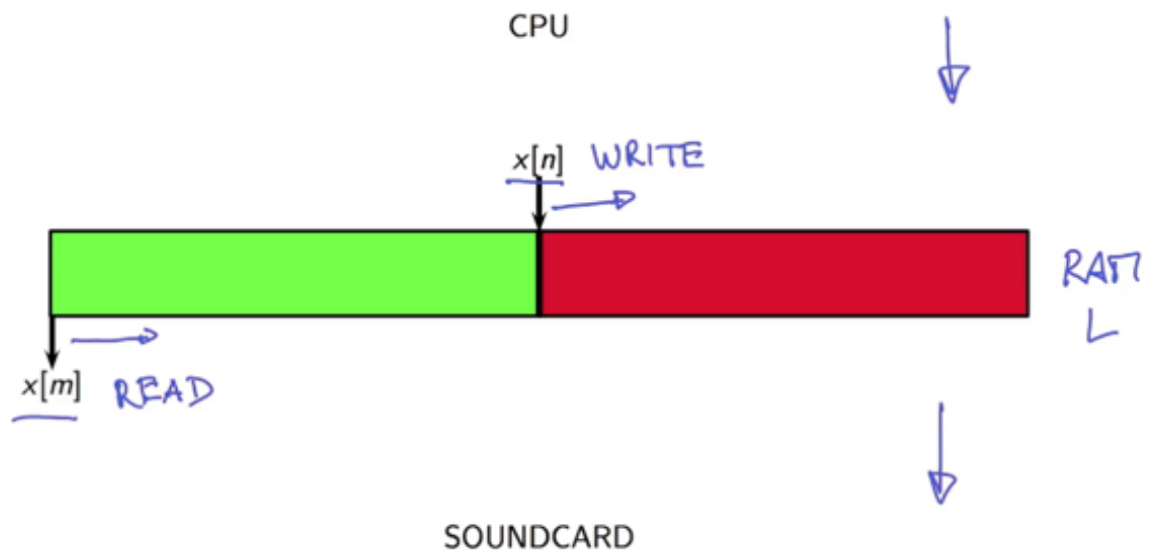
notes

summary

5m 10s



Example: double buffering (output)



Remember, the key is that the CPU writes sample faster than the sound card reads them.

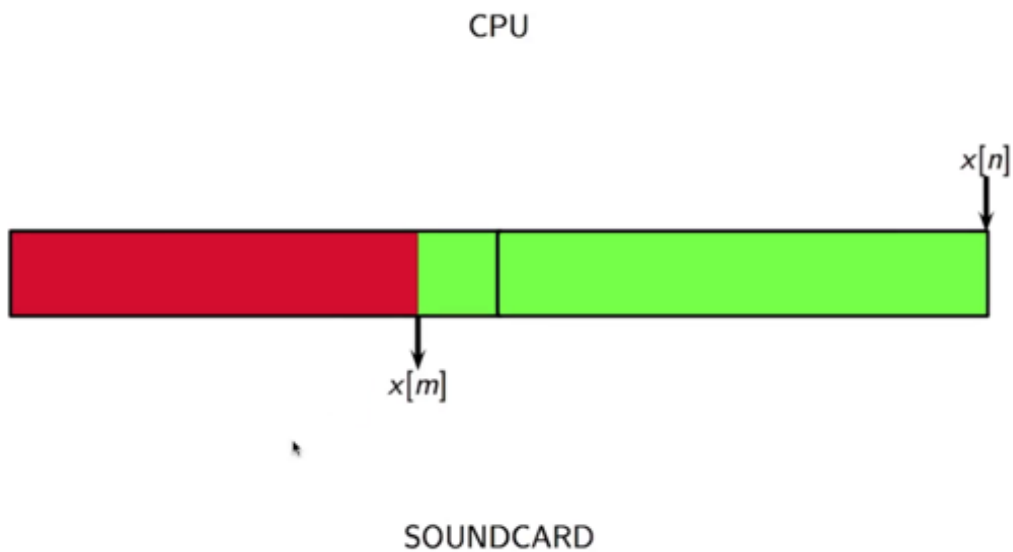
notes

summary

5m 13s



Example: double buffering (output)



So as time progresses, you can see that the read pointer advances, but the write pointer advances faster. And after a few milliseconds, the CPU will have filled the second half of the buffer, and it will stop. At the same time, the sound card will continue reading, and when it reads the entirety of the first half of the buffer,

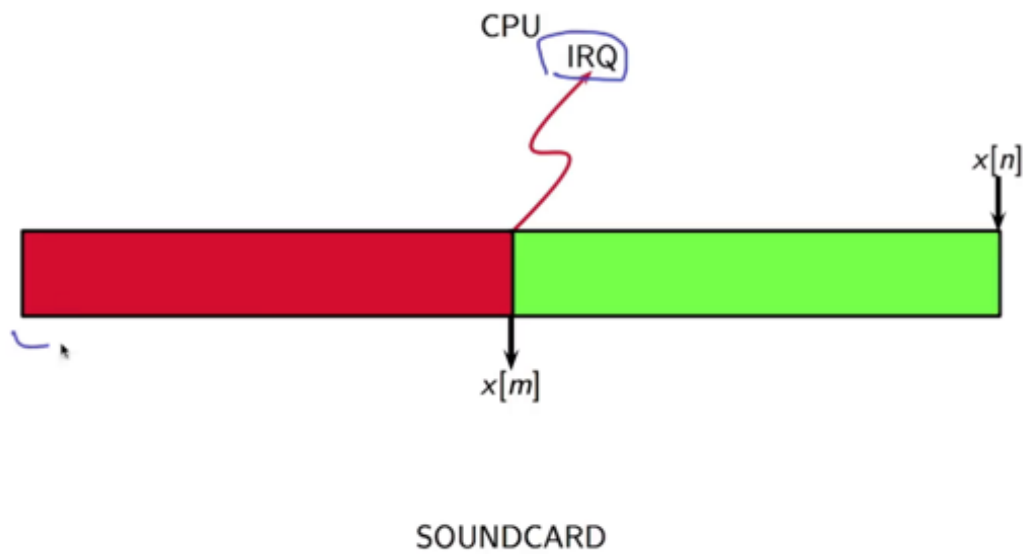
notes

summary

5m 25s



Example: double buffering (output)



it will signal to the CPU that this part of the buffer is now depleted before moving on to reading samples in the second half of the buffer.

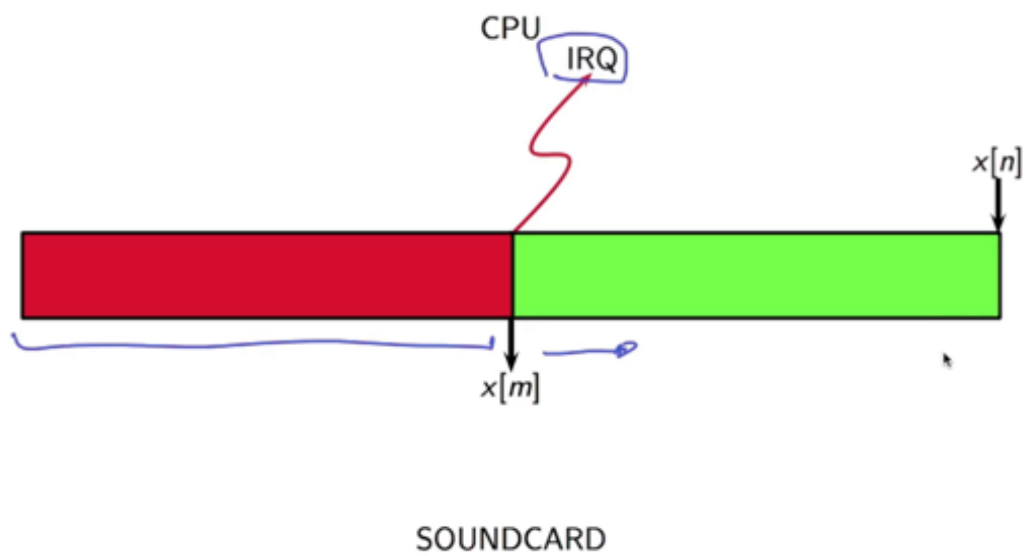
notes

summary

5m 46s



Example: double buffering (output)



When the CPU receives the IRQ, it knows that it will have to start filling the first half of the buffer. This is a circular buffer, as we have seen in the previous module, so the pointer will be rolled around, and the CPU will start writing samples here.

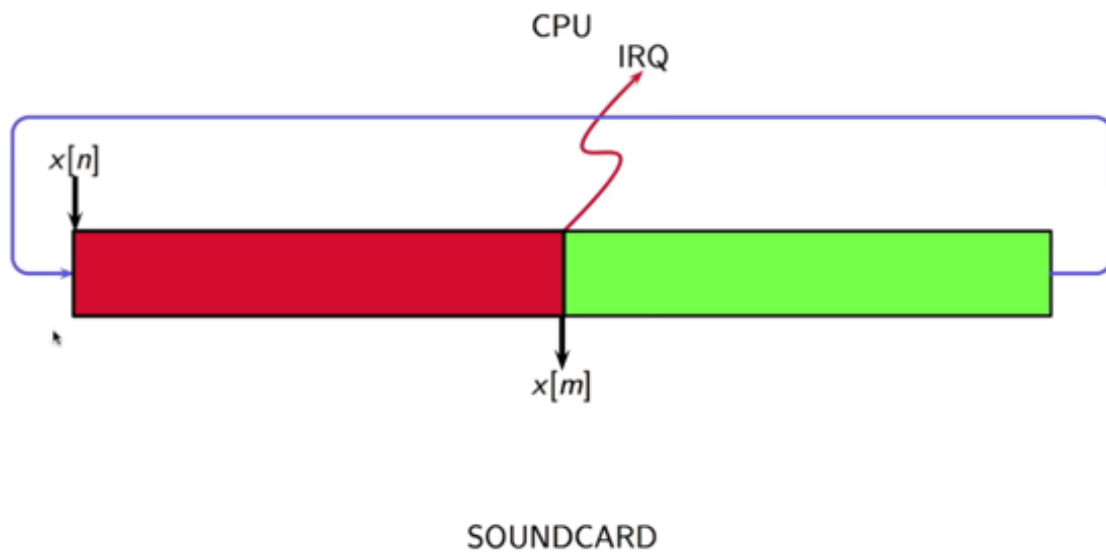
notes

summary

5m 55s



Example: double buffering (output)



So the reading process continues at the same pace as before, and now the filling process continues on the first half of the buffer at a faster pace than the reading process.

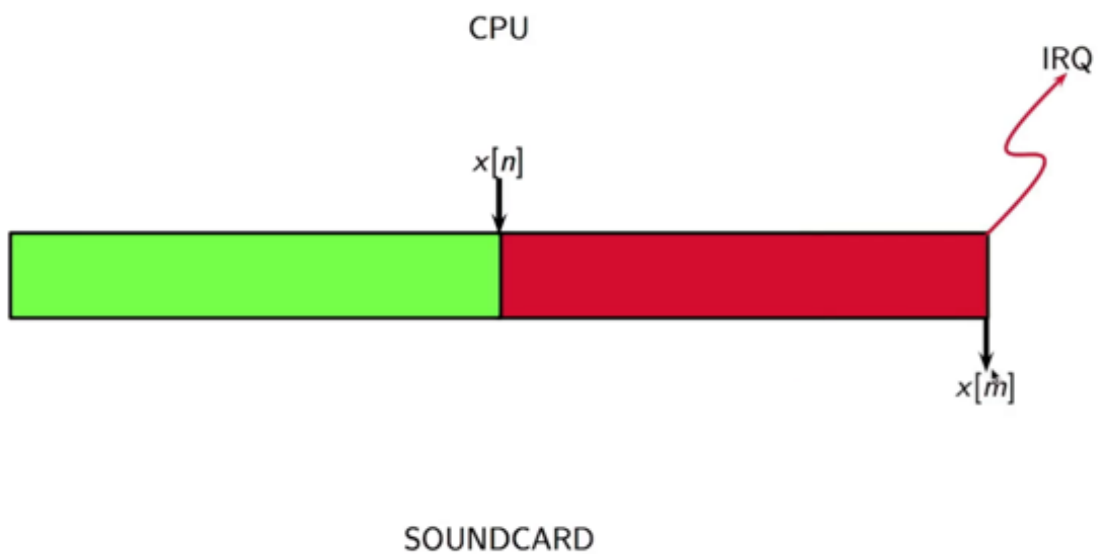
notes

summary

6m 11s



Example: double buffering (output)



And so again, the CPU will have filled the first half of the buffer while the sound card is still reading the second half. When the second half is depleted, a new IRQ will signal to the CPU that the second half of the buffer needs to be filled again, and so on and so forth.

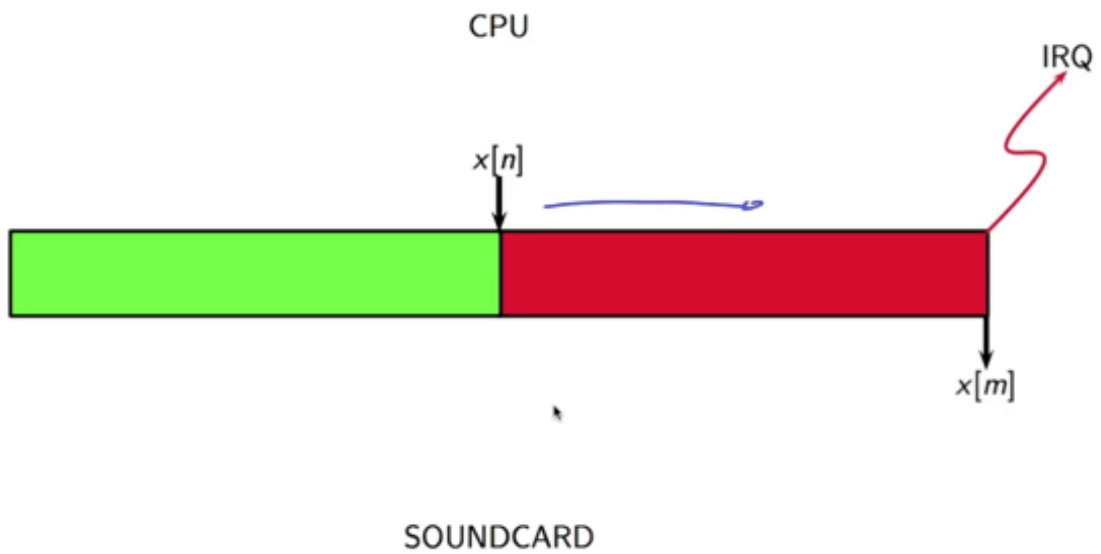
notes

summary

6m 15s



Example: double buffering (output)



DoubleBuffer introduces a delay that is equal to the system clock times half the length of the buffer.

notes

summary

6m 34s



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

If the CPU is not fast enough, the read pointer of the sound card will trail over memory cells that contain invalid samples, old samples, and we will have a situation of underflow.

notes

summary

6m 40s





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

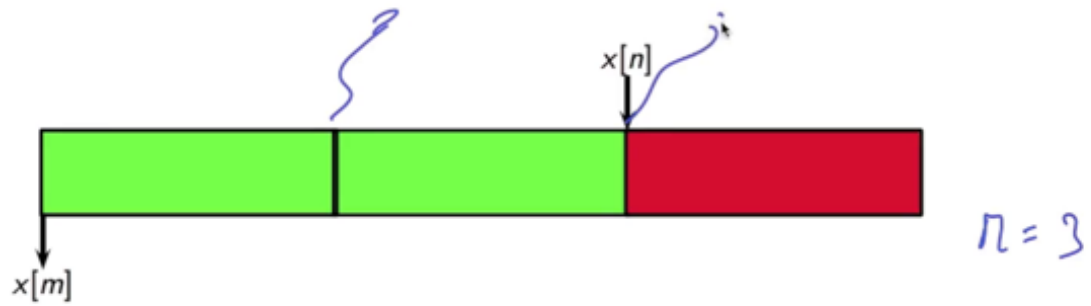
In practical application, it's common to use multiple bufferings instead of simple double buffering. In this case, the buffer is divided into sub-buffers, and an IRQ will be emitted every time the read pointer trails one of the boundaries.

notes

summary

6m 50s





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

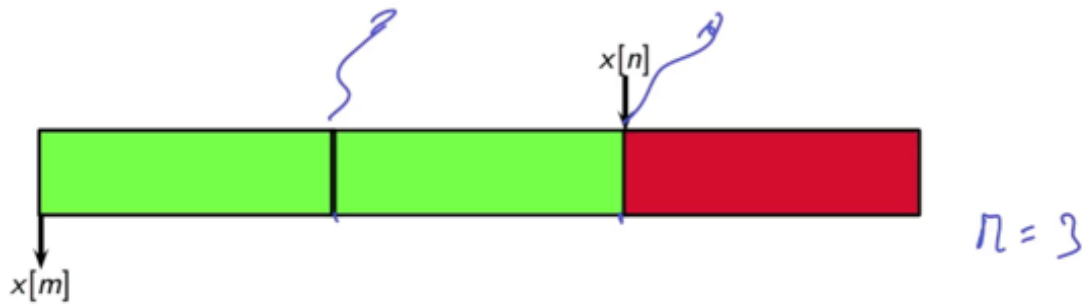
The advantage is that the CPU will be called more often, and so the load on the CPU will be better distributed in time, while still affording a reasonable underflow protection.

notes

summary

7m 10s





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

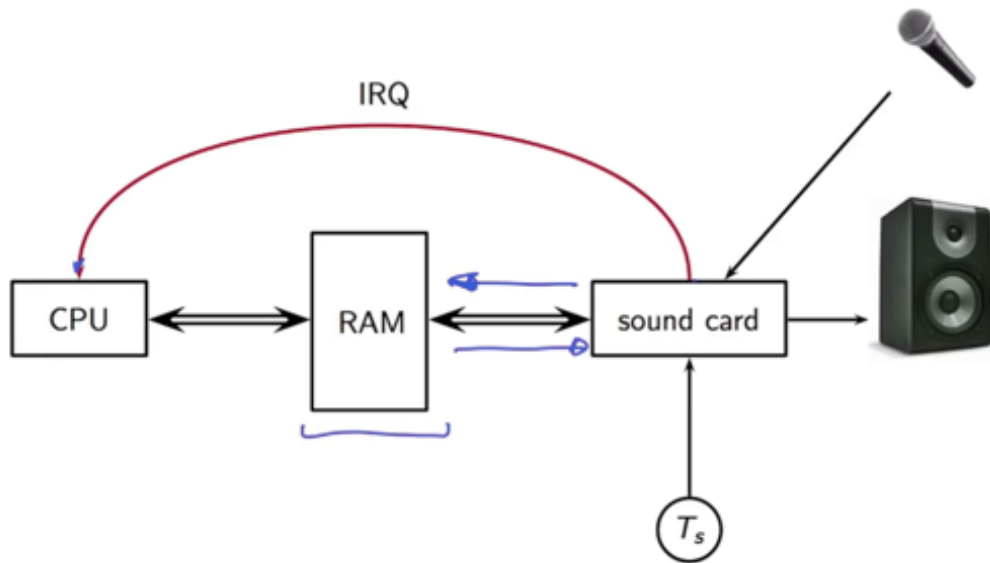
So, so far we looked at the output process of the sound card. What about the input?

notes

summary

7m 14s





The input is really symmetrical. Instead of having the sound card fetch sample from memory, we will have the sound card put samples in memory and notify the CPU that a new buffer is available for processing. The CPU will work at its own pace. It will share the RAM with the sound card. The sound card will both place input sample in memory and fetch output samples to play, and will signal to the CPU when either of these buffers are empty. Very quickly, the roles are reversed. If this is the buffer and RAM, we now have that the write pointer belongs to the sound card, and the read pointer belongs to the CPU.

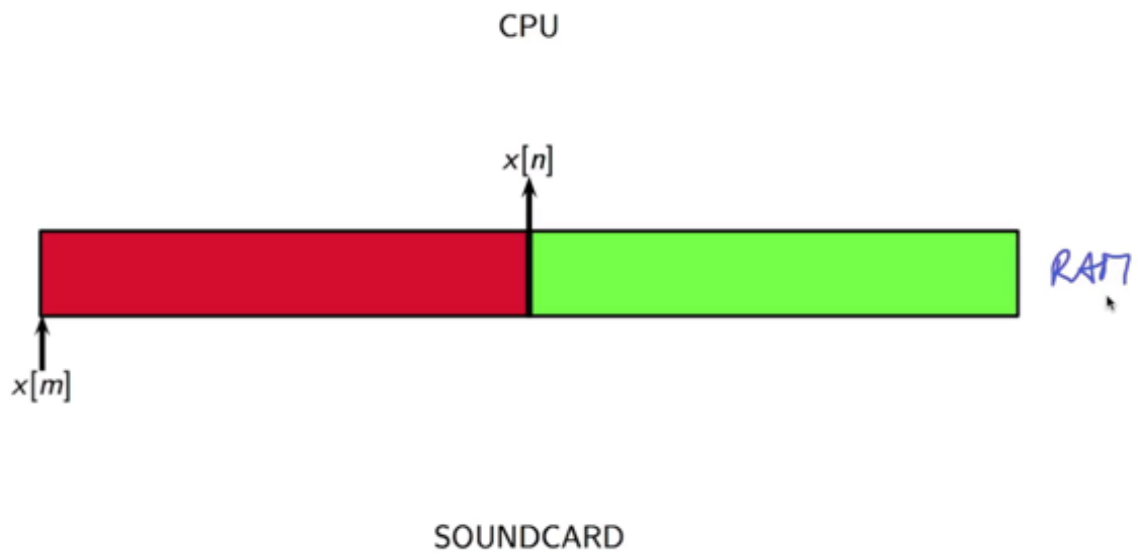
notes

summary

7m 28s



Example: double buffering (input)



We assume, as per usual, that the initial buffer is filled with zero, and we start the system.

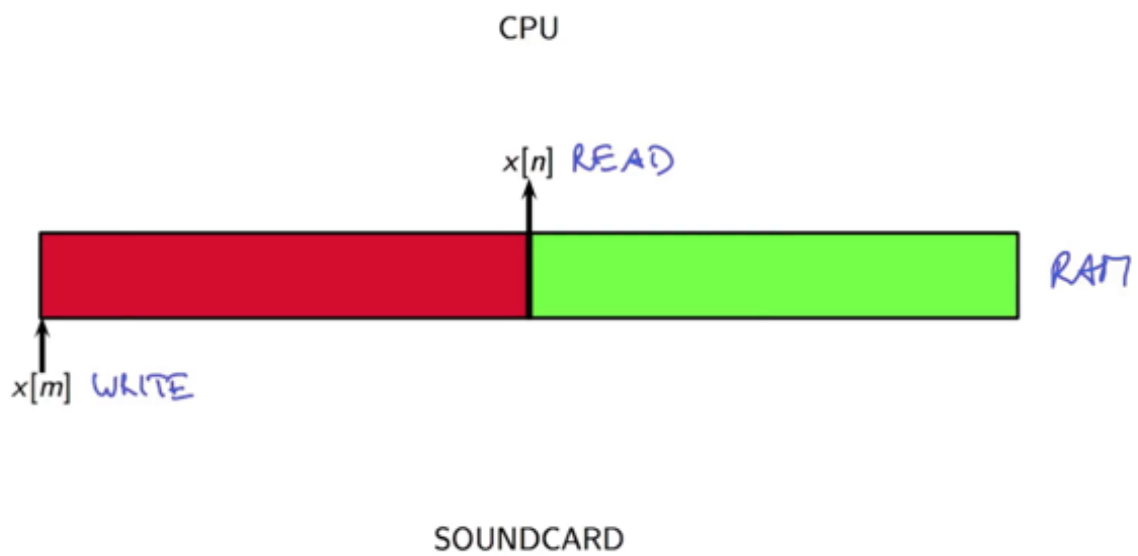
notes

summary

8m 6s



Example: double buffering (input)



The CPU will start getting samples from the buffer, and the sound card will start writing samples to the buffer. The CPU will be faster, and so it will deplete the buffer before the sound card has finished filling the other half of the buffer.

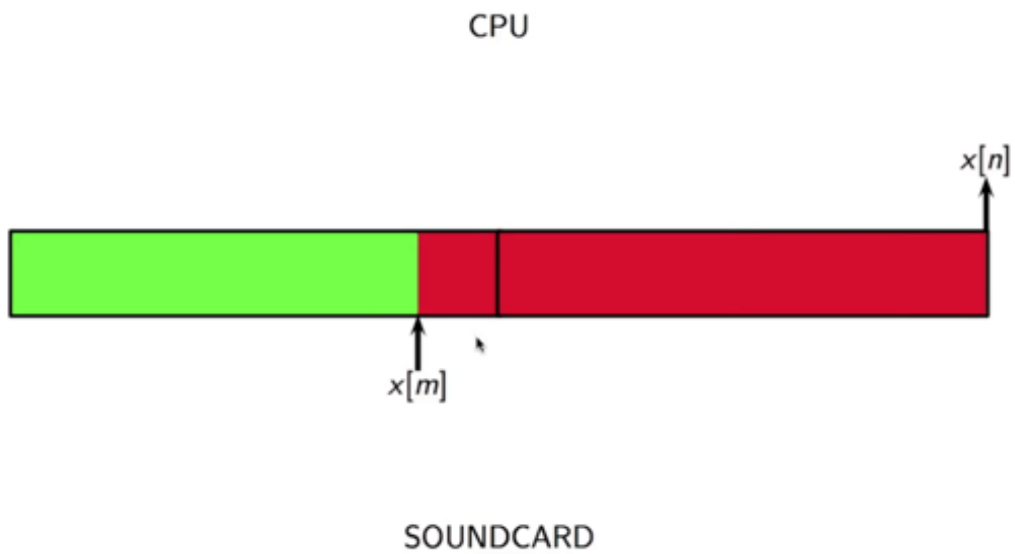
notes

summary

8m 13s



Example: double buffering (input)



When the sound card has finished filling the first half, it will trigger an IRQ, and the CPU will know that this buffer is now ready for consumption, so it will roll the pointer and start fetching the data and process it, and so on and so forth.

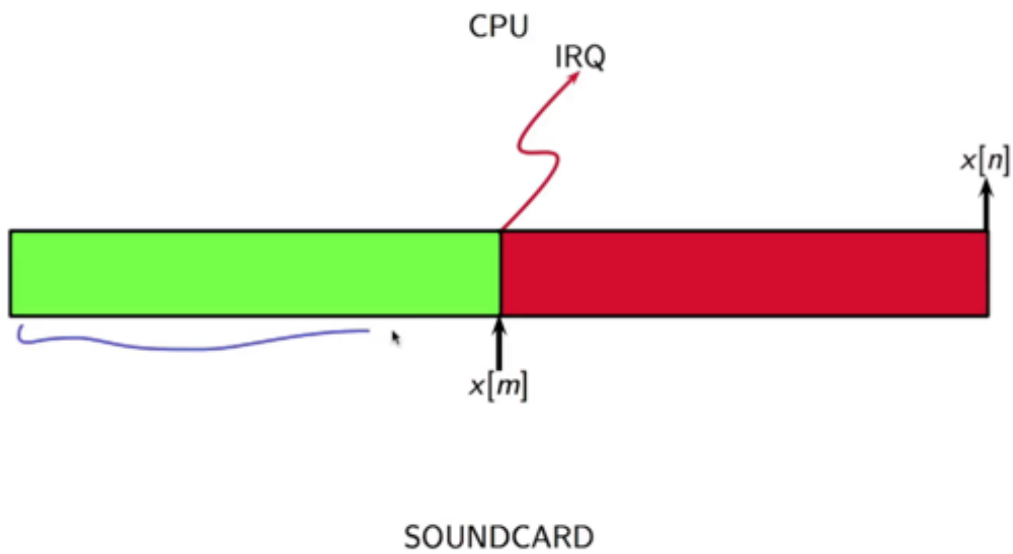
notes

summary

8m 34s



Example: double buffering (input)



We now need to put it all together. So we will have an input buffer and an output buffer. We will use a multiple buffer strategy, say with three sub-buffers.

notes

summary

8m 46s



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

We will select the buffer sizes and the number of sub-buffers to be the same, both for input and output, and we will use the input IRQ to drive the processing.

notes

summary

8m 53s



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

This is the input buffer, and so the sound card will place samples into this buffer as they come from, say, the microphone.

notes

summary

9m 10s





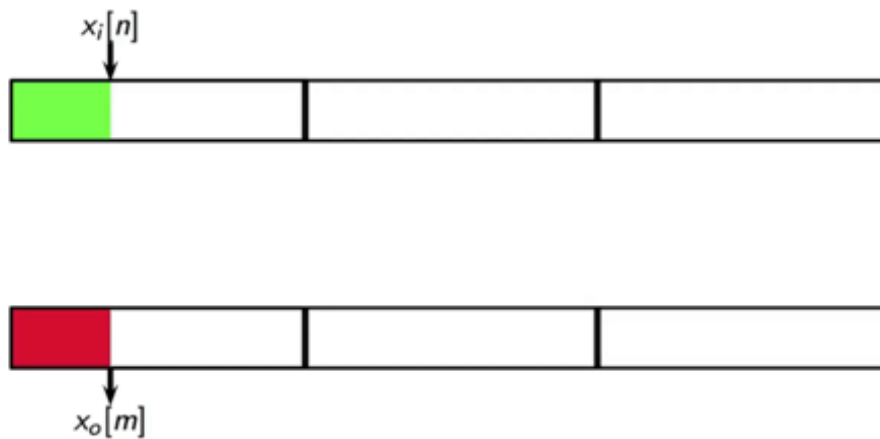
This is the output buffer, and so the sound card will fetch samples from this buffer and play them through the loudspeaker.

notes

summary

9m 21s





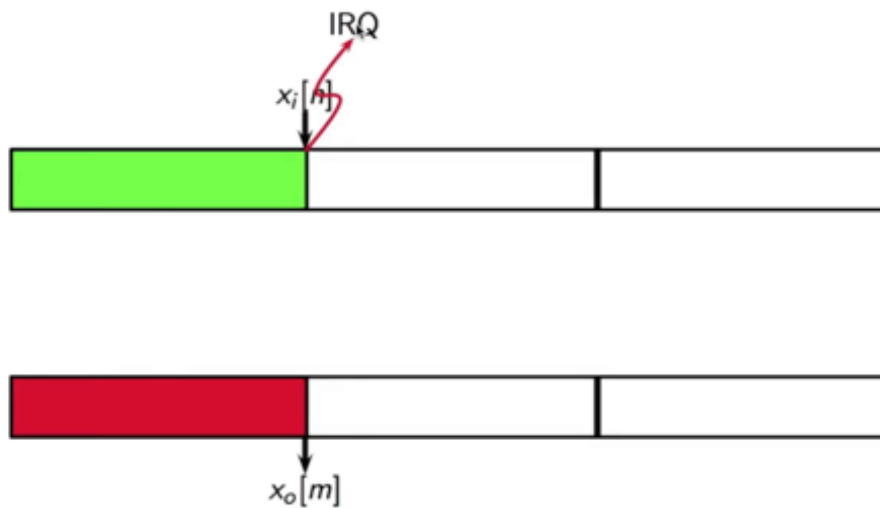
Assume that the buffer is a zero-filled. Since the clock for the sound card is the same both for input and output, the input buffer will be filled at the same rate as the output buffer is emptied. After a few milliseconds, the first chunk in the buffer will be filled, and conversely, the output buffer will be emptied.

notes

summary

9m 31s





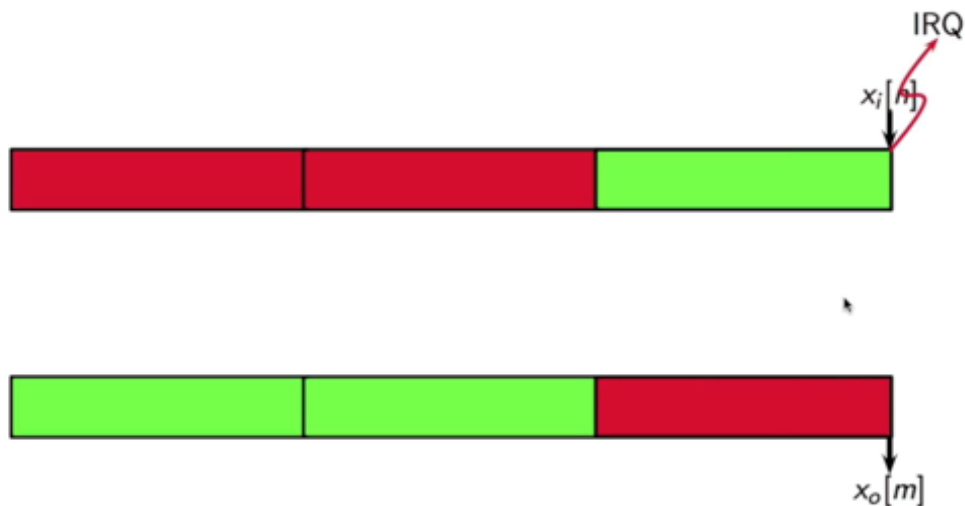
An IRQ will be launched, at which point the CPU will intervene and start processing the input samples, say filtering them, and placing them in the corresponding chunk of the output buffer for later use.

notes

summary

9m 48s





Again, the CPU will have to perform this operation faster than the time it takes for the sound card to fill a chunk of the buffer. So here, for instance, when the sound card has filled, say, a third of the next buffer, the CPU is done with the processing. It will mark the input sub-buffer as used up and the output sub-buffer as ready for playing. The process continues. The sound card is still filling up the second chunk of the buffer, and when it's done, it will launch an IRQ, and the CPU will start processing this input chunk into this output chunk. The processing requires less time than the sound card needs to fill the third part of the buffer, and when the third part of the buffer is filled, an IRQ will be triggered, pointers will be rolled around, and the sound card will start outputting the first chunk of the buffer that was processed earlier,

notes

summary

10m 0s





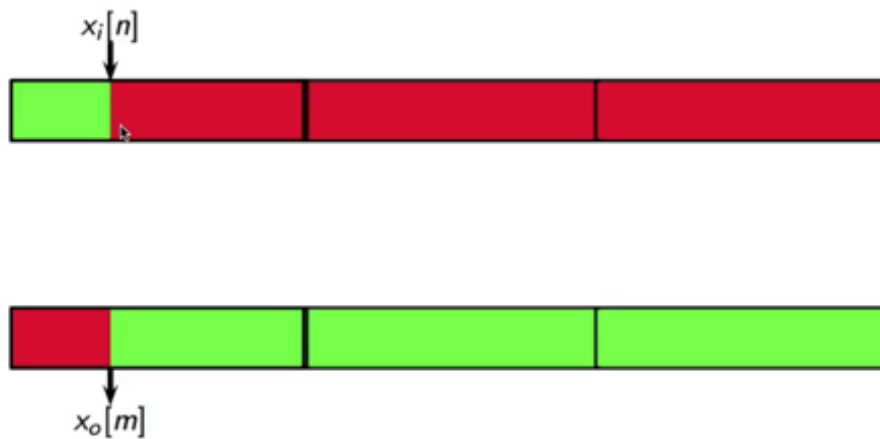
while the corresponding chunk of the input buffer will be filled with new input samples. And the process repeats.

notes

summary

10m 59s





The total delay of the system is equivalent to TS times the number of samples in the buffer, because of how we position the read and write pointers.

notes

summary

11m 4s



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

We usually start the output process first, because it doesn't matter if the input trails a little bit, and buffers can be collapsed.

notes

summary

11m 13s



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

In other words, the computation can be done in place.

notes

summary

11m 25s



- ▶ low level:

- ▶ high level:

Okay, so how do we do this in practice? We get our PC, and one strategy is the low-level strategy. We study the sound card data sheet. Each chip is different. We write code to program the sound card directly.

notes

summary

11m 29s



- _____

This will require us to write some particular values to some particular IOPorts. We need to write an interrupt handler, and we need to write the code that handles the data.

notes

summary

11m 47s



- ▶ 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
 - write a callback function to handle the data

It's certainly interesting, but very time-consuming, and not very portable. The high-level strategy is to choose a good API that allows us to attack the functionality of the sound card without worrying about the details of the implementation, and just write a callback function that handles the data when we are notified that a new buffer is available. Let's look at an example of a typical callback prototype. We will use C in our next examples.

notes

summary

11m 51s



```
int Callback( const void *input, 4  
              void *output, 4  
              unsigned long samples, 4  
              ...);
```

This callback will take simply three mandatory arguments. A pointer to the input buffer, a pointer to the output buffer, and the length of said buffers.

notes

summary

12m 27s



```
int Callback( const void *input,
              void *output,
              unsigned long samples)
{
    float* pIn  = (float *)input;
    float* pOut = (float *)output;

    for (int n = 0; n < samples; n++)
        *pOut++ = Process(*pIn++);
}
```

For example, we could write a callback that does simply a cast to convert the generic pointers to the buffers into the right data type.

notes

summary

12m 45s



Callback example



```
int Callback( const void *input,
              void *output,
              unsigned long samples)
{
    float* pIn  = (float *)input;
    float* pOut = (float *)output;

    for (int n = 0; n < samples; n++)
        *pOut++ = Process(*pIn++);
}
```

Handwritten blue annotations: A bracket under the for loop, a bracket under the Process call, and an arrow pointing to the Process function with a question mark.

In this case, we choose to use floats. And then a simple loop that, for each sample in the input buffer, calls a processing function and stores the result into the output buffer. So now the question is, what do we put into the processing function? So let's expand that, and let's define a generic processing gateway.

notes

summary

12m 54s



```
// 10 sec @ 24KHz
enum {BUF_LEN = 0x10000};
enum {BUF_MASK = BUF_LEN - 1};

float m_pY[BUF_LEN];
float m_pX[BUF_LEN];
int m_Ix;
int m_Iy;
```

```
float Process(float Sample)
{
    m_pX[m_Ix] = Sample;
    float y = Effect();
    m_pY[m_Iy] = y;
    m_Ix = (m_Ix + 1) & BUF_MASK;
    m_Iy = (m_Iy + 1) & BUF_MASK;
    return y;
}
```

If we're going to use filters, we will need to put in place buffers to store past samples of the input and of the output. So we do this here on this left part of the panel. As we said before, we decide to use floats, so we define a buffer of a certain length, and we choose a power of 2 for convenience so that the circular buffer can be rolled around with a simple masking operation. So we define two buffers of equal length and two indices into the buffer. The processing function will take the current sample, it will store the sample into the input memory, it will call a function that we call effect because we're interested in implementing guitar effects, that will compute the current output, it will store the output in the memory for the output samples, and it will update the indices into the buffers with the usual circular strategy. So we increment the pointer by one, and then we roll around the pointer by binary masking. And the processing function will return the current output sample that, as you remember, will be put in the output buffer of the sound card. So let's look at the simple effects that we can generate with this programming paradigm.

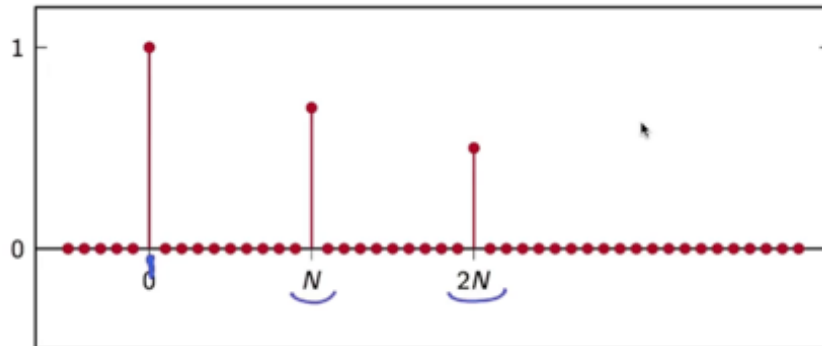
notes

summary

13m 14s



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



An echo is a situation where you have a sound that bounces back and forth between two reflecting surfaces. And a simple simulation is given by the following impulse response. You have the original sound, and then you have a first reflection after N samples, and a second reflection after two N samples. You have equally spaced replicas of the signal, which we scale by different weights in order to simulate a decay in time. So we have the original signal scaled by A , A is usually one, a first reflection scaled by B , and a second reflection scaled by C .

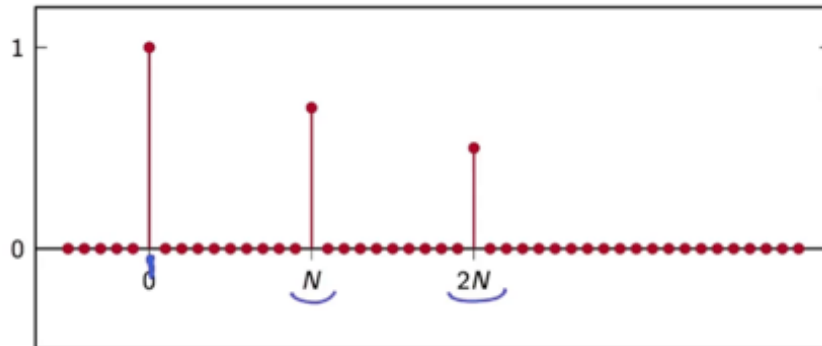
notes

summary

14m 28s



$$y[n] = \frac{ax[n] + bx[n-N] + cx[n-2N]}{a+b+c}$$



And then we normalize the sum of reflection by the sum of the coefficients in order to have unit gain.

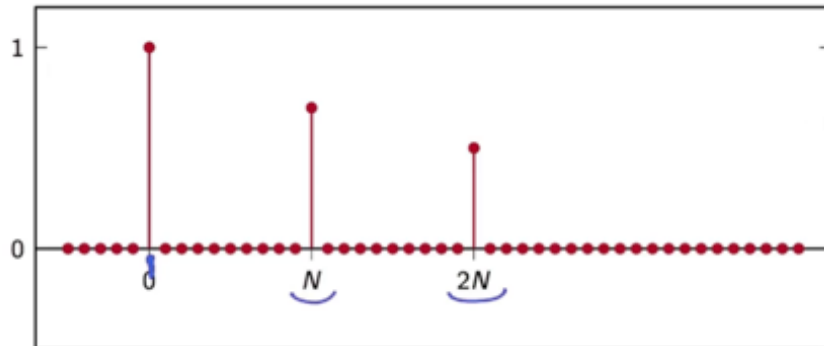
notes

summary

15m 0s



$$y[n] = \frac{ax[n] + bx[n - N] + cx[n - 2N]}{(a + b + c)}$$



To implement this in our framework, all we need is a straightforward translation of the transfer function into C code.

notes

summary

15m 7s



```
float Echo()
{
    static float a = 1;
    static float b = 0.7f;
    static float c = 0.5f;
    static float norm = 1.0f / (a+b+c);
    static int N = (int)(0.3 * m_SR);

    return norm * (    a * m_pX[m_Ix]
                    + b * m_pX[(m_Ix + BUF_LEN - N) & BUF_MASK]
                    + c * m_pX[(m_Ix + BUF_LEN - 2*N) & BUF_MASK]);
}
```

So here we define the three coefficients that we use for the three replicas of the signal, the normalizing coefficient, and the delay between replicas, which is here expressed as a function, of course, of the sampling frequency.

notes

summary

15m 13s



```

float Echo()
{
    static float a = 1;
    static float b = 0.7f;
    static float c = 0.5f;
    static float norm = 1.0f / (a+b+c);
    static int N = (int)(0.3 * m_SR);

    return norm * (    a * m_pX[m_Ix]
                    + b * m_pX[(m_Ix + BUF_LEN - N) & BUF_MASK]
                    + c * m_pX[(m_Ix + BUF_LEN - 2*N) & BUF_MASK]);
}

```

The output value is simply the weight sum of three delayed replicas of the input. Acoustically, a simple echo sounds like this. First you will hear the original sound, and then you will hear the same sound processed by the simple echo. The simple echo has two drawbacks with respect to a natural echo. The first one is that it has only a finite number of repetition, whereas in a natural situation it would have a theoretically infinite number of back and forth reflections. The second drawback is that each repetition is just an exact replica of the original sound, simply scaled in amplitude, whereas the reflection process in a natural echo would introduce a low pass characteristic in each reflection. To obtain a better echo, we can go back to our old friend, the Karpo-Strong algorithm.

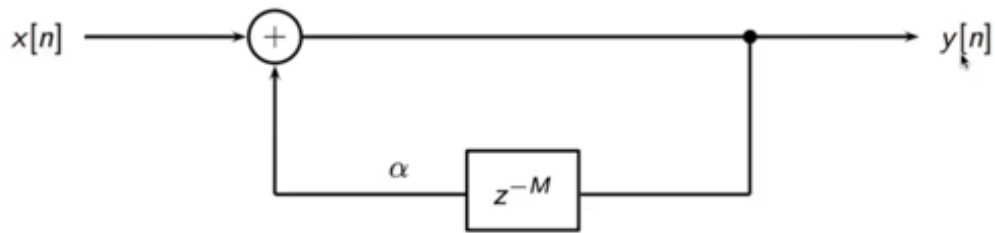
notes

summary

15m 25s



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



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

If you remember, the Karpo-Strong works by building equally spaced replicas of the input signal to produce the output. The fact that the input signal is finite support prevents overlap between the output copies,

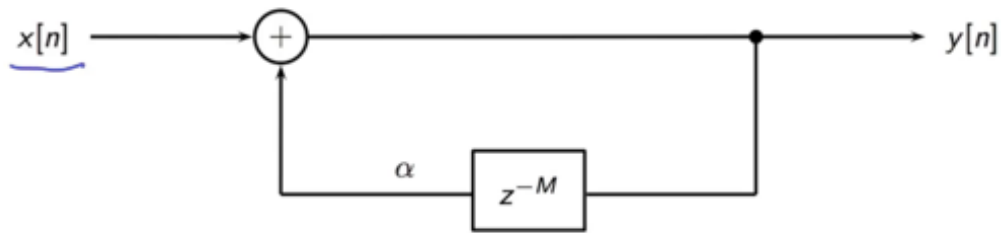
notes

summary

16m 26s



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



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

but if we input a standard sequence to the feedback loop, we will have overlapping replicas in time. So the feedback loop already takes care of the fact that we want an infinite number of repetitions in the echo. In order to introduce the low pass characteristic of a natural echo, we simply introduce a low pass filter in the feedback path,

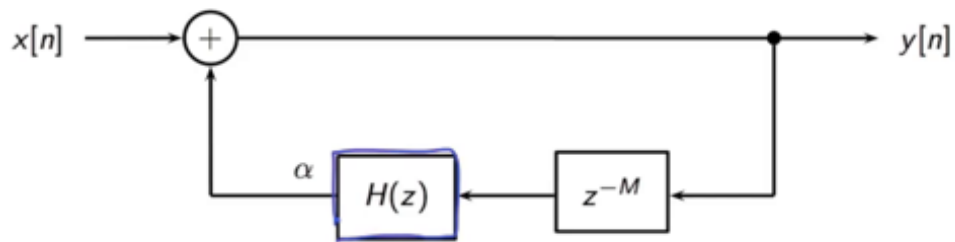
notes

summary

16m 43s



a natural echo has a lowpass characteristic



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

and the output at this point can be expressed as the usual decay factor times the convolution of the output and the impulse response of the low pass, delayed by capital M, plus the input. If we choose a simple low pass such as the Leakey integrator,

notes

summary

17m 4s

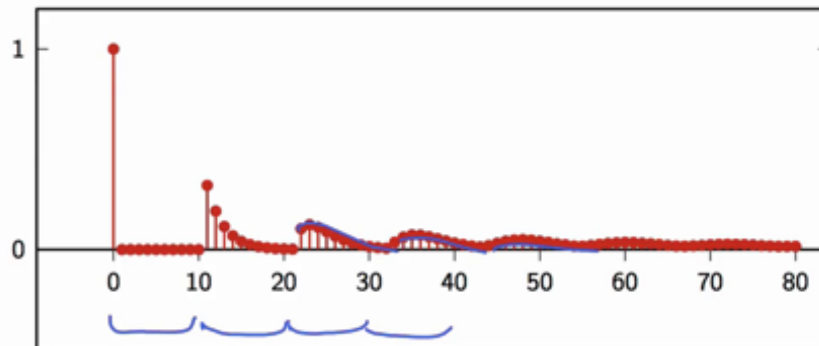


A better echo



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

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



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

we can actually write the constant coefficient difference equation, which turns out to be like that. And if we compute the impulse response numerically, we get something that each repetition of the period of the echo gets smoother and smoother, which is really what you obtain if you apply a low pass filter repeatedly. The implementation in our processing framework is once again very straightforward.

notes

summary

17m 22s



```
float NaturalEcho()
{
    static float a = 0.7; )
    static float L = 0.6; )
    static float norm = 1.0f / (1+a); )
    static int N = (int)(0.3 * m_SR); )

    return norm * (
        m_pX[m_Ix]
        - L *      m_pX[(m_Ix + BUF_LEN - 1) & BUF_MASK]
        + L *      m_pY[(m_Iy + BUF_LEN - 1) & BUF_MASK]
        + a*(1-L) * m_pY[(m_Iy + BUF_LEN - N) & BUF_MASK]);
}
```

We just need to convert the constant coefficient difference equation to C statements. We define the decay factor, the lambda factor for the Leakey integrator. We define the normalizing factor that will preserve unit gain, and again the echo delay as a function of the sampling frequency. And then we return the output as a linear combination of inputs and output samples. If we now play the same clip as before through this natural echo, it will hopefully sound better and more realistic. So here is a comparison between the two.

notes

summary

17m 46s



- ▶ 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

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

The next linear time invariant effect that we will consider is reverb. Reverb is really the superposition of very, very many echo-like reflections that come from different directions and therefore have different delays and different type of attenuations associated to them. Because of the complexity of the impulse response associated to reverb,

notes

summary

18m 39s



- ▶ 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

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

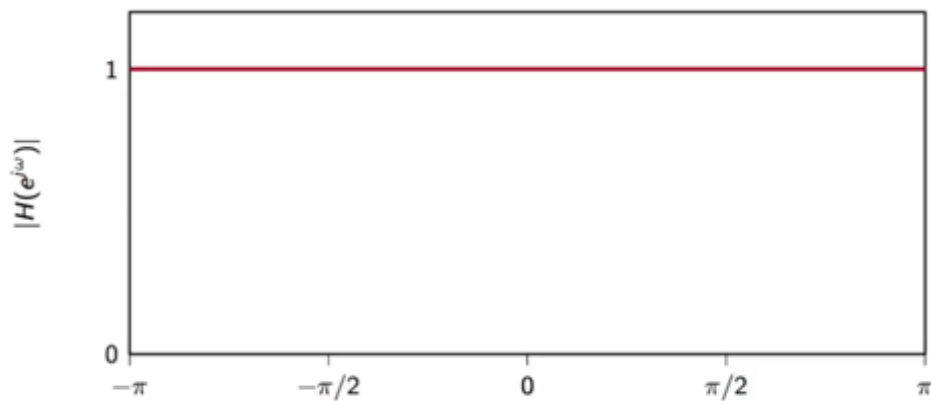
we can get away by using a simple all-pass filter as shown here. Remember, the all-pass filter is a filter whose magnitude response is unit over the entire minus pi to pi frequency axis.

notes

summary

19m 6s





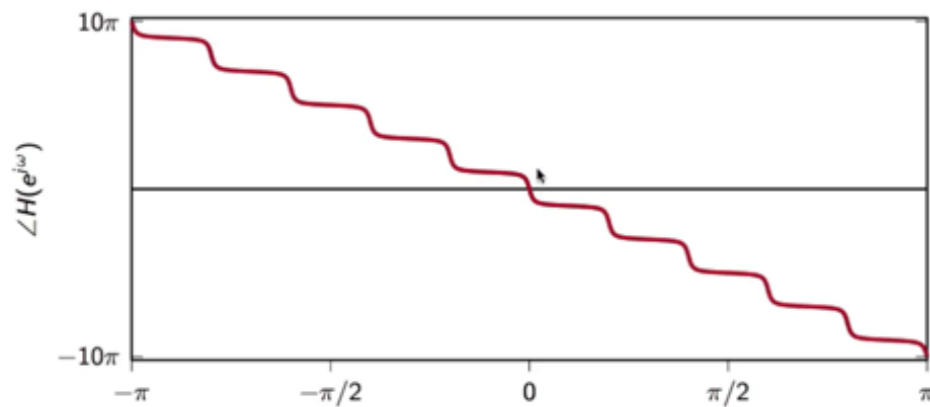
By appropriately choosing the parameters α and capital N in the all-pass,

notes

summary

19m 20s





we can obtain a phase response with the staircase-like characteristic shown here. This phase response will introduce different delays to different frequency regions,

notes

summary

19m 25s



```
float Reverb()
{
    static float a = 0.8; )
    static int N = (int)(0.02 * m_SR); )

    return ( - a * m_pX[m_Ix]
            +   m_pX[(m_Ix + BUF_LEN - N) & BUF_MASK]
            + a * m_pY[(m_Iy + BUF_LEN - N) & BUF_MASK]);
}
```

thereby simulating a multifaceted reflective surface. The implementation of the all-pass filter is again extremely straightforward. We only have two parameters to choose, and then we just implement the custom coefficient difference equation in C. We can now try to apply the reverb to a little guitar snippet. We'll play the original and then the processed signal. Let's now look at some very common guitar effects that do not belong to the class of linear time invariant filters. This is to give you an idea that there is a whole world out there beyond linear processing that is very, very interesting.

notes

summary

19m 37s



```
float Reverb()
{
    static float a = 0.8; )
    static int N = (int)(0.02 * m_SR); )

    return ( - a * m_pX[m_Ix]
            +   m_pX[(m_Ix + BUF_LEN - N) & BUF_MASK]
            + a * m_pY[(m_Iy + BUF_LEN - N) & BUF_MASK]);
}
```

The first type of effect that we will consider is the distortion.

notes

summary

20m 12s



- ▶ distortion: 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

$$y[n] = x[n] + x[n - \lfloor d(1 + \cos(\omega_0 n)) \rfloor]$$

Simple distortion can be achieved by truncating a scaled version of the signal. Tremolo is an effect achieved by multiplying the input signal by a slowly varying sinusoidal component.

notes

summary

20m 13s



- ▶ distortion: 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

$$y[n] = x[n] + x[n - \lfloor d(1 + \cos(\omega_0 n)) \rfloor]$$

This will change the volume periodically in time. Flanger is obtained by summing a signal to a delayed version of itself,

notes

summary

20m 26s



- $$y[n] = x[n] + x[n - \lfloor d(1 + \cos(\omega_0 n)) \rfloor]$$



- ▶ distortion: 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

$$y[n] = x[n] + x[n - \underbrace{[d(1 + \cos(\omega_0 n))]}_{\text{delay}}]$$

When you sum a signal to a delayed version of itself, you obtain some spectral nulls in frequency. By changing the delay, the spectral nulls will change their position in time, and this will give rise to a variety of effects from, say, a robotic sound to an artificial chorus. From the point of view of implementation, the fact that these effects are nonlinear doesn't really complexify the associated code,

notes

summary

20m 55s



```
float Fuzz()
{
    static float limit = 0.005;
    static float G = 5;

    float y = m_pX[m_Ix];
    if (y > limit)
        y = limit;
    if (y < -limit)
        y = -limit;
    return G*y;
}
```

and this is really one of the strong points of digital signal processing. We can easily move beyond the standard filtering paradigm without incurring much complexity. So distortion simply takes the input signal, operates a hard limit, and rescales the output. And it sounds like this. Tremolo requires you to define the speed of the modulating sinusoid,

notes

summary

21m 15s



```
float Tremolo()
{
    static double phi = 5 * 2*PI / m_SR; // 5Hz LFO
    static double omega = 0;

    omega = omega + phi;
    return (1 + cos(omega)/4) * m_pX[m_Ix];
}
```

and then the output is obtained simply by multiplying the input times an appropriately scaled sinusoid.

notes

summary

21m 48s



```
float Tremolo()
{
    static double phi = 5 * 2*PI / m_SR; // 5Hz LFO
    static double omega = 0;

    omega = omega + phi;
    return (1 + cos(omega)/4) * m_pX[m_Ix];
}
```

Tremolo sounds like this.

notes

summary

21m 52s



```
float Flanger()
{
    static int N = (int)(0.002 * m_SR);    // max delay
    static double phi = 0.1 * 2*PI / m_SR; // 0.1Hz LFO
    static double omega = 0;

    int d = (int)(N * (1 + cos(omega)));
    omega = omega + phi;
    return 0.5f + (m_pX[m_Ix] + m_pX[(m_Ix + BUF_LEN - d) & BUF_MASK]);
}
```

Finally, with Flanger, we need to define the maximum delay, the frequency at which this delay will oscillate around its mean value, and then we compute the instantaneous delay by converting to an integer the product of the maximum delay times the oscillating component, and with some of the two delayed versions of the input.

notes

summary

22m 10s

