SON 2026-2027 lecture 3

3TC Audio Project @ INSA Lyon

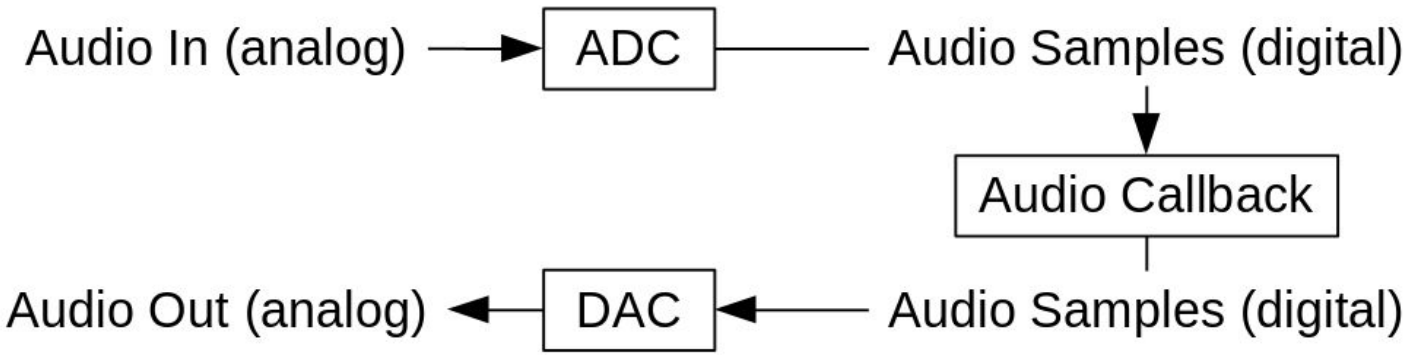
Tanguy Risset

https://inria-emeraude.github.io/son/

Digital Audio Systems Architectures and Audio Callback

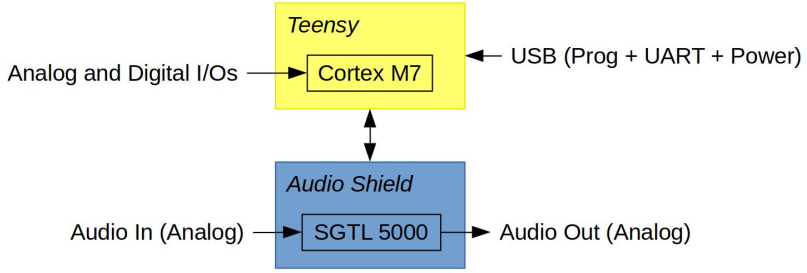
Basic Architecture of a Digital Audio System

- All digital audio systems have an architecture involving:
  - an ADC
  - DAC.
- Audio samples are processed on a computer (i.e., CPU, microcontroller, DSP, etc.) by the *audio callback*
- Audio callback is *trigerred* by DAC/ADC



Architecture of Embedded Audio Systems Such as the Teensy

- The audio codec we use is an SGTL5000 mounted on a board with the same form factor as the Teensy.
- Audio samples are exchanged between the Cortex M7 and the audio codec using the I2S protocol.
- The Cortex M7 has its own analog inputs which is used with sensor datas (e.g., potentiometers, etc.).



Teensy and Audio Shield Overview

Concept of Audio Rate, and Control Rate

- *Audio rate* is 48kHz, audio samples must change at audio rate.
- What happens outside of the `for` loop of the `audioCallback` is called the *control rate*
- The parameters of an audio program (e.g. volume) are typically processed at control rate
- The block size is directly linked to the audio latency of the system by the following formula: $latency = BS/fs$ where $latency$ is in seconds.
- For example, a block size of 256 samples at a sampling rate of 48 kHz will induce a latency of approximately 5ms (10ms between input/output).
- On the Teensy can be as small as 8!

First Audio Program on the Teensy: `crazy-sine`

- Crazy sine: an example containing a program synthesizing a sine wave on the Teensy and controlling its frequency: crazy-sine
- Examinez la méthode `update()` de la classe `MyDsp`

```
#define MULT_16 32767

void MyDsp::update(void) {
  audio_block_t* outBlock[AUDIO_OUTPUTS];
  for (int channel = 0; channel < AUDIO_OUTPUTS; channel++) {
    outBlock[channel] = allocate();
    if (outBlock[channel]) {
      for (int i = 0; i < AUDIO_BLOCK_SAMPLES; i++) {
        float currentSample = echo.tick(sine.tick())*0.5;
        currentSample = max(-1,min(1,currentSample));
        int16_t val = currentSample*MULT_16;
        outBlock[channel]->data[i] = val;
      }
      transmit(outBlock[channel], channel);
      release(outBlock[channel]);
    }
  }
}
```

Analysis of the structure of `crazy-sine` application

- 3 files:
  - `crazy-sine.ino` (The sketch, same name as the directory)
  - `MyDsp.cpp` and `MyDsp.h` defining the class MyDsp realizing the audio
- `crazy-sin.ino` :

```
MyDsp myDsp;
AudioOutputI2S out;
AudioControlSGTL5000 audioShield;
AudioConnection patchCord0(myDsp,0,out,0);
AudioConnection patchCord1(myDsp,0,out,1);
```

  - Defines the *patch*, here: no inputs, connects output of `MyDsp` to audio output (`out`)
  - instantiates a `MyDsp` object and control its parameter in the `loop()` function
- `MyDsp.h` indicates the components used by `MyDsp` : a sine wave (class `Sine`) and an echo (class `Echo`)
- `MyDsp.cpp` defines the function `MyDsp::update()` and `MyDsp::SetFreq(float freq)`
- each sample is computed by the code: `currentSample = echo.tick(sine.tick())*0.5`

Analysis of `crazy-sine`

- The `update` method *is the audio callback*
- it is called every time a new audio buffer is needed by the system.
- A new audio buffer `outBlock` containing `AUDIO_OUTPUTS` channels is first created.
- For every audio channel, memory is allocated and a full block of samples is computed.
- Each sample is computed in `currentSample` (`float`) using the `sine` and `echo` components.
- `currentSample` range is {-1;1}.
- `max(-1,min(1,currentSample));` ensures it doesn't exceed this range.

```
#define MULT_16 32767

void MyDsp::update(void) {
  audio_block_t* outBlock[AUDIO_OUTPUTS];
  for (int channel = 0; channel < AUDIO_OUTPUTS; channel++) {
    outBlock[channel] = allocate();
    if (outBlock[channel]) {
      for (int i = 0; i < AUDIO_BLOCK_SAMPLES; i++) {
        float currentSample = echo.tick(sine.tick())*0.5;
        currentSample = max(-1,min(1,currentSample));
        int16_t val = currentSample*MULT_16;
        outBlock[channel]->data[i] = val;
      }
      transmit(outBlock[channel], channel);
      release(outBlock[channel]);
    }
  }
}
```

Analysis of `crazy-sine` (2)

- The values contained in `currentSample` (between -1 and 1) must be converted to 16 bits signed integers
- For that, we just have to multiply `currentSample` by 2^{16-1}.
- `currentSample` is multiplied by 0.5 to prevent potential saturation).
- The output buffer is transmitted to codec (`transmit` function).
- The memory allocated is freed using the (`release` function).
- The `update` method is called over and over until the Teensy is powered out.

```
#define MULT_16 32767

void MyDsp::update(void) {
  audio_block_t* outBlock[AUDIO_OUTPUTS];
  for (int channel = 0; channel < AUDIO_OUTPUTS; channel++) {
    outBlock[channel] = allocate();
    if (outBlock[channel]) {
      for (int i = 0; i < AUDIO_BLOCK_SAMPLES; i++) {
        float currentSample = echo.tick(sine.tick())*0.5;
        currentSample = max(-1,min(1,currentSample));
        int16_t val = currentSample*MULT_16;
        outBlock[channel]->data[i] = val;
      }
      transmit(outBlock[channel], channel);
      release(outBlock[channel]);
    }
  }
}
```

Exercice: have crazy sine running on your Teensy

- Open the `crazy sine` project in arduino (<../examples/teensy/projects/crazy-sine/crazy-sine.ino>)
- Compile and Run
- Plug the headphone (in the jack socket of the audio shield)
- hear the music...

Parenthesis: From C to C++

What you need to know to read and modify the Teensy examples

Why C++ here?

- The Teensy (Arduino) is programmed in **C++**, not C
- Good news: this C++ stays very close to C
  - no STL, no lambdas, no smart pointers
  - just **a handful of object-oriented concepts** on top of what you already know
- Goal of this session: be able to read and modify
  - `MyDsp.h` / `MyDsp.cpp` in each project
  - the small DSP classes in `libraries/mydsp/src` (`Sine`, `Echo`, `Phasor`, …)

`class`: a `struct` that also bundles functions

- In C: data (`struct`) and functions are separate.
- In C++: a **class** bundles data (called *members*, or *fields*) **and** functions (called *methods*):

```
class Sine {
  public:
    Sine(int SR);
    void setFrequency(float f);
    void setGain(float g);
    float tick();
  private:
    SineTable sineTable;
    Phasor phasor;
    float gain;
    int samplingRate;
};
```

- `public` : what the rest of the program is allowed to call
- `private` : internal details, hidden from the outside (**encapsulation**)

Calling a method: `object.method()`

- No explicit first struct parameter like in C: the object a method is called on is **implicit**.

```
// inside MyDsp::update()
float currentSample = echo.tick(sine.tick()) * 0.5;
```

- roughly equivalent, in spirit, to this C code:

```
float currentSample = echo_tick(&echo, sine_tick(&sine)) * 0.5;
```

- `sine.tick()` already "knows" it's operating on `sine`.

Constructor / destructor

Replace the `xxx_init()` / `xxx_free()` functions from C: they're called **automatically** at creation/destruction.

```
class Echo {
  public:
    Echo(int SR, int maxDel);   // constructor
    ~Echo();                    // destructor
  private:
    float* delBuffer;
    ...
};


Echo::Echo(int SR, int maxDel) : ... {
  delBuffer = new float[maxDel];   // allocated on creation
  ...
}
Echo::~Echo() {
  delete[] delBuffer;              // freed automatically
}
```

`new / delete`: C++'s `malloc / free`

- `new T` / `new T[n]` → allocates and **calls the constructor**
- `delete p` / `delete[] p` → **calls the destructor**, then frees

```
delBuffer = new float[maxDel];   // instead of malloc(maxDel * sizeof(float))
...
delete[] delBuffer;              // instead of free(delBuffer)
```

⚠️ Never mix `new` / `delete` with `malloc` / `free` on the same pointer.

The initializer list (: after the constructor)

The syntax that surprises C programmers the most:

```
Sine::Sine(int SR) :
  sineTable(SINE_TABLE_SIZE),
  phasor(SR),
  gain(1.0),
  samplingRate(SR)
{
  //code of Sine Initialization
}
```

- Each member is initialized **before** the `{}` body runs
- For member objects (`sineTable` , `phasor`), this is what calls **their own constructor**
- Rough C equivalent: calling `sine_table_init(&sineTable, SINE_TABLE_SIZE)` as the first line of the constructor

Composition: an object that contains other objects

```
class Sine {
  private:
    SineTable sineTable;   // an object, not just plain data
    Phasor phasor;         // with its own constructor/destructor
    ...
};
```

- In `CrazySin`, the `MyDsp` class contains a `Sine` **and** an `Echo`
- Each sub-object manages its own construction/destruction: `MyDsp` doesn't need to do anything special — it happens automatically

`.h` declares, `.cpp` defines: the `::` operator

Just like in C you split prototype (`.h`) and implementation (`.c`), but here you also specify **which class the function belongs to**:

```
// Sine.h
class Sine { public: void setFrequency(float f); ... };

// Sine.cpp
void Sine::setFrequency(float f) {   // "Sine's setFrequency method"
  phasor.setFrequency(f);
}
```

`Sine::` = scope resolution operator

Inheritance: `MyDsp` is an `AudioStream`

```
class MyDsp : public AudioStream {
  public:
    MyDsp();
    ~MyDsp();
    virtual void update(void);   // see next slide
    void setFreq(float freq);
  private:
    Sine sine;
    Echo echo;
};
```

- `MyDsp` inherits everything `AudioStream` already knows how to do (managing audio blocks, `transmit()`, `allocate()`, `release()` ...)
- `MyDsp` only adds what's specific to it: the DSP processing

`virtual`: overriding an inherited method

- `AudioStream` declares `update()` as `virtual`
- `MyDsp::update()` **overrides** `AudioStream::update()` it with its own behavior
- The audio library calls `update()` automatically, at regular intervals → **your** version is the one that runs

```
virtual void MyDsp::update(void) {
  audio_block_t* outBlock[AUDIO_OUTPUTS];
  for (int channel = 0; channel < AUDIO_OUTPUTS; channel++) {
    outBlock[channel] = allocate();
    for (int i = 0; i < AUDIO_BLOCK_SAMPLES; i++) {
      float s = echo.tick(sine.tick()) * 0.5;
      outBlock[channel]->data[i] = s * MULT_16;
    }
    transmit(outBlock[channel], channel);
    release(outBlock[channel]);
  }
}
```

This is the mechanism that drives the whole audio processing loop.

Summary: your own C++ coding

- When you create a new arduino application, e.g. a variation of `CrazySine` :
  - create a directory named "my_audio_project" for instance.
  - copy the `CrazySine` directory from `examples/teensy/project` to `my_audio_project`
  - rename the directory and the sketch (`.ino`) accordingly, `My_CrazySine` for instance
  - Use the same structure:
    - `MyDsp` object perform the audio effet
    - `My_CrazySine` controls the `MyDsp` object
  - **Do not modify** the files in `examples/teensy/project` or in `$$ARDUINO/libraries$`

Sine Wave Oscillator

- Sine wave are at the basis of many algorithms in the field of audio.
- The sound of a sine wave is what we call a "pure tone" since it only has a single harmonic.
- All sounds can be synthesized using a combination of sine waves (Fourier transform).

a sine oscillator corresponds to this equation:

$$x(t) = Asin(\omega t + \phi)$$

with:

- A: the peak amplitude
- $\omega = 2\pi f$: the radian frequency (rad/sec)
- f: the frequency in Hz
- t: the time seconds
- ϕ: the initial phase (radians)

Sine Wave Oscillator in C++

- $x(t)$ could be translated to C++ by writing something like (ϕ is ignored here):

```
float currentSample = A*std::sin(2*PI*f*t);
```

- The would be very computationally expensive (call the `std::sin` function $48\,000$ times per second)
- In practice: pre-compute the sine wave and store it in a wave table before computation starts.
- That kind of algorithm is then called a "wave table oscillator."

Sine.cpp, which is used in uses SineTable.cpp which pre-computes a sine table:

```
table = new float[size];
for(int i=0; i<size; i++){
  table[i] = std::sin(i*2.0*PI/size);
}
```

and then makes it accessible through the `tick` (compute) method:

```
float SineTable::tick(int index){
  return table[index%tableSize];
}
```

Sine Oscillator precision

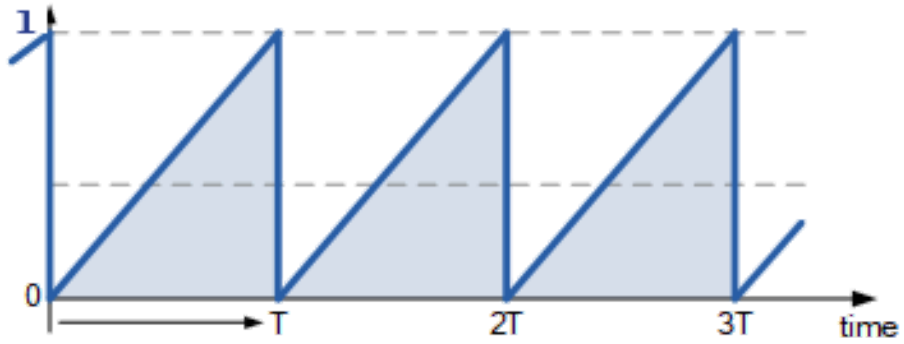
- The greater the size, the more accurate/pure the sine wave. A low resolution sine wave will produce more distortion.
- In `Sine.cpp`, the sine wave table has a size of 2^{14} ($\simeq$ 64 KB) which presents a good compromise between sound quality and memory.
- It is important to keep in mind that when working with embedded systems memory is also an important factor to take into account.
- Teensy has approximaely 1MB of memory in RAM

Phasor

- The sine table is then read with a "phasor."
- A phasor produces a ramp signal which is reset at a certain frequency.
- It can also be seen as a sawtooth wave. Phasor.cpp is used for that purpose and its `tick` method is defined as:

```
void Phasor::setFrequency(float f){
  phasorDelta = f/samplingRate;
}

float Phasor::tick(){
  float currentSample = phasor;
  phasor += phasorDelta;
  phasor = phasor - std::floor(phasor);
  return currentSample;
}
```



Phasor at frequency 1/T

- In one second, f_s samples are produced
- We want the phasor to have f cycle in one second
- One cycle adds `PhasorDelta` up to 1
- Hence we want the Phasor to add up to f in one seconds

`PhasorDelta` = (quantity\ added)/(number\ of\ adds) = f/f_s$$

Using phasor and SineTable

It hence ramps from 0 to 1 at a given frequency. The `phasor` object in `Sine.cpp` is used to read the sine table by adjusting the range of its output:

```
float Sine::tick(){
  int index = phasor.tick()*SINE_TABLE_SIZE;
  return sineTable.tick(index)*gain;
}
```

Exercices:

- Looping Through a Small Tune
- Basic Additive Synthesis
- Stereo Echo

Looping Through a Small Tune

- Musical notes are usually represented by MIDI numbers.
- In MIDI, each pitch of the chromatic scale has a number between 0 and 127 associated to it: https://djip.co/blog/logic-studio-9-midi-note-numbers
- Middle C (Do) corresponds to number 72.
- MIDI note numbers can be converted to a frequency using the following formula:

$$f = 2^{(d-69)/12}.440$$

- where d is the MIDI number.
- Write a small tune/song looping through at least 5 notes and play it with the `crazy-sine` program on your Teensy.

Basic Additive Synthesis

- *Additive synthesis* consists of adding multiple sine wave oscillators together to "sculpt" the timbre of a sound.
- A simple additive synthesizer could be implemented from the `crazy-sine` example by declaring multiple instances of `sine`. E.g.:

```
float currentSample = echo.tick(sine0.tick()*gain0 + sine1.tick()*gain1);
```

- But this would mean that memory will be allocated twice for the `sineTable` array.
- Try to call the `sineTable` a second time after `float currentSample = sineTable[index]*gain;` to add a second oscillator to the generated sample without dupplicating the sineTable. Try for instance:

$$x(t) = sin(2\pi ft) + sin(2\pi(1.5f)t)$$

- **Warning**, you should create a copy of the `Sine` class (e.g. `Sine2`)

Hint: Beware of clipping!

Stereo Echo

- Create a second instance of `Echo` (connected to the same instance of `sine`) *with different parameters from the first one* that will be connected to the second channel of the output.
- The first instance of `Echo` should be connected to the left channel and the second one to the right channel

```
float sineSample = sine.tick();
float currentSampleL = echo0.tick(sineSample)*0.5;
float currentSampleR = echo1.tick(sineSample)*0.5;
```

Hint: Beware of memory allocation again! Make sure that the maxim delay of your echo (on the 2 parameters of the class constructor) doesn't exceed 10000 for now for both instances of the echo.

End of Lecture 3