

Week 9 – Audio Concepts, APIs, and Architecture

Roger B. Dannenberg

Professor of Computer Science and Art
Carnegie Mellon University



Introduction

- So far, we've dealt with discrete, symbolic music representations
- “Introduction to Computer Music” covers sampling theory, sound synthesis, audio effects
- This lecture addresses some system and real-time issues of audio processing
- We will not delve into any DSP algorithms for generating/transforming audio samples

Overview

- Audio Concepts
 - Samples
 - Frames
 - Blocks
 - Synchronous processing
- Audio APIs
 - PortAudio
 - Callback models
 - Blocking API models
 - Scheduling
- Architecture
 - Unit generators
 - Fan-In, Fan-Out
 - Plug-in Architectures

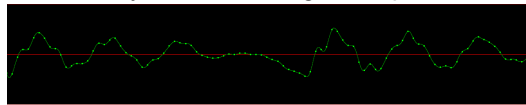
3

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Audio Concepts

- Audio is basically a stream of signal amplitudes



- Typically represented
 - Externally as 16-bit signed integer: +/- 32K
 - Internally as 32-bit float from [-1, +1]
 - Floating point gives >16bit precision
 - And "headroom": samples >1 are no problem as long as later, something (e.g. a volume control) scales them back to [-1, +1]
- Fixed sample rate, e.g. 44100 samples/second (Hz)
- Many variations:
 - Sample rates from 8000 to 96000 (and more)
 - Can represent frequencies from 0 to 1/2 sample rate
 - Sample size from 8bit to 24bit integer, 32-bit float
 - About 6dB/bit signal-to-noise ratio
 - Also 1-bit delta-sigma modulation and compressed formats

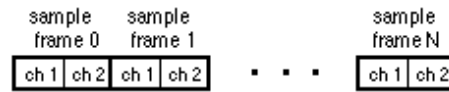
4

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Multi-Channel Audio

- Each channel is an independent audio signal
- Each sample period now has one sample per channel
- Sample period is called an *audio frame*
- Formats:
 - Usually stored as *interleaved* data
 - Usually processed as independent, non-interleaved arrays
 - Exception: Since channels are often correlated, there are special multi-channel compression and encoding techniques, e.g. for surround sound on DVDs.



□ = one sample point

5

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Block Processing Reduces Overhead

- Example task: convert stereo to mono with scale factor
- Naïve organization:


```
read frame into left and right
output = scale * (left + right)
write output
```
- Block processing organization


```
read 64 interleaved frames into data
for (i = 0; i < 64; i++) {
    output[i] = scale * (data[i*2] + data[i*2 + 1]);
}
write 64 output samples
```

System call per frame

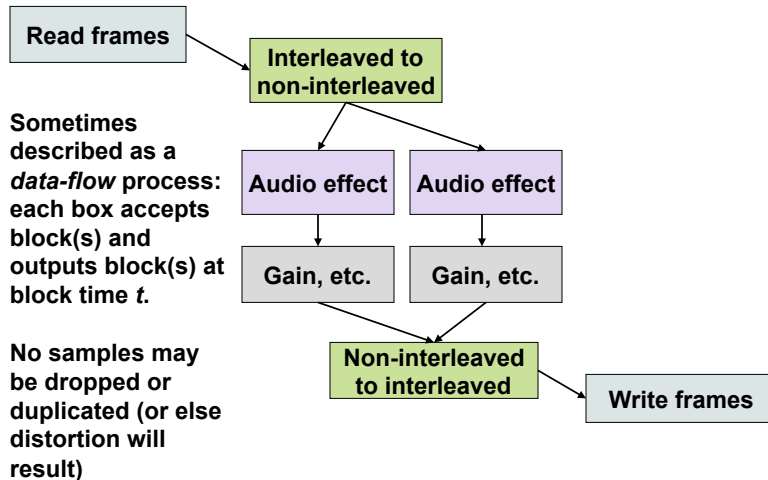
Load scale and locals to registers

6

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Audio is *Always* Processed Synchronously



7

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Audio Latency Is Caused (Mostly) By Sample Buffers

- Samples arrive every 22 μ s or so
- Application cannot wake up and run once for each sample frame (at least not with any efficiency)
- Repeat:
 - Capture incoming samples in input buffer while taking output samples from output buffer
 - Run application: consume some input, produce some output
- Application can't compute too far ahead (output buffer will fill up and block the process).
- But Application *can* fall too far behind (input buffer overflow, output buffer underflow) – bad!

8

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency/Buffers Are Not Completely Bad

- Of course, there's no reason to increase buffer sizes just to add delay (latency) to audio!
- What about reducing buffer sizes?
 - Very small buffers (or none) means we cannot benefit from block processing: more CPU load
 - Small buffers (~1ms) lead to underflow if OS does not run our application immediately after samples become available.
- Blocks and buffers are a “necessary evil”

9

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

There Are Many Audio APIs

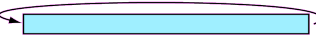

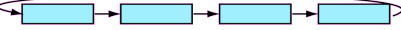
- Every OS has one or more APIs:
 - **Windows**: WinMM, DirectX, ASIO, Kernel Streaming
 - **Mac OS X**: Core Audio
 - **Linux**: ALSA, Jack
- APIs exist at different levels
 - Device driver – interface between OS and hardware
 - System/Kernel – manage audio streams, conversion, format
 - User space – provide higher-level services or abstractions through a user-level library or server process

10

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Buffering Schemes

- Hardware buffering schemes include:
 - Circular Buffer 
 - Double Buffer 
 - Buffer Queues 
- these may be reflected in the user level API
- Poll for buffer position, or get interrupt or callback when buffers complete
 - What's a callback?
- Typically audio code generates blocks and you care about adapting block-based processing to buffer-based input/output. (It may or may not be 1:1)

11

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency in Detail



- Audio input/output is strictly synchronous and precise (to $< 1\text{ns}$)
- Therefore, we need input/output buffers
- Assume audio block size = b samples
- Computation time r sample times
- Assume *pauses* up to c sample periods
- Worst case:
 - Wait for b samples – inserts a delay of b
 - Process b samples in r sample periods – delay of r
 - Pause for c sample periods – delay of c
 - Total delay is $b + r + c$ sample periods

12

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency In Detail: Circular Buffers

- Assumes sample-by-sample processing
 - Audio latency is $b + r + c$ sample periods
 - In reality, there are going to be a few samples of buffering or latency in the transfer from input hardware to application memory and from application memory to output hardware.
 - But this number is probably small compared to c
 - Normal buffer state is: input empty, output full
- 
- Worst case: output buffer almost empty
- 
- Oversampling A/D and D/A converters can add 0.2 to 1.5ms (each)

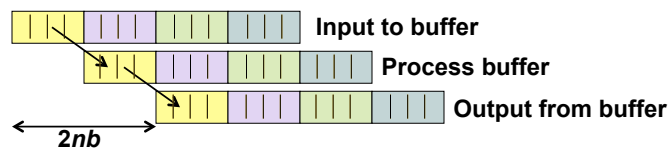
13

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency In Detail: Double Buffer

- Assumes block-by-block processing
- Assume buffer size is nb , a multiple of block size
- Audio latency is $2nb$ sample periods



- How long to process one buffer (worst case)?
- How long do we have?

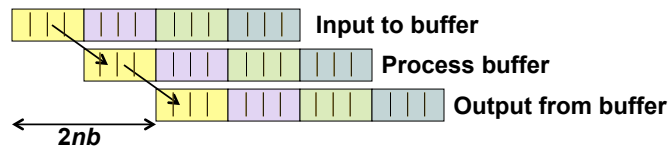
14

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency In Detail: Double Buffer

- Assumes block-by-block processing
- Assume buffer size is nb , a multiple of block size
- Audio latency is $2nb$ sample periods



- How long to process one buffer (worst case)? $nr + c$
- How long do we have? nb
- $n \geq c / (b - r)$

15

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency In Detail: Double Buffer (2)

- $n \geq c / (b - r)$
- Example 1:
 - $b = 64$
 - $r = 48$
 - $c = 128$
 - $\therefore n = 8$
 - Audio latency = $2nb = 1024$ sample periods
- Example 2:
 - $b = 64$
 - $r = 48$
 - $c = 16$
 - $\therefore n = 1$
 - Audio latency = $2nb = 128$ sample periods

How does this compare to circular buffer?

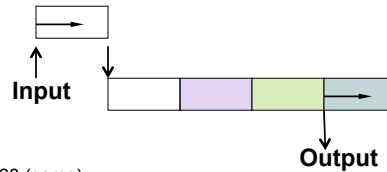
16

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Latency In Detail: Buffer Queues

- Assume queue of buffers with b sample each (buffer size = block size)
- Queues of length n on both input and output
- In the limit, this is same as circular buffers
- In other words, circular buffer of n blocks
- If we are keeping up with audio, state is:
- Audio latency = $(n - 1)b$
- Need: $(n - 2)b > r + c$
- $\therefore n \geq (r + c) / b + 2$
- Example 1: latency = 256 vs 1024, Ex 2: 128 (same)



17

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Synchronous/blocking vs Asynchronous/callback APIs

- Blocking APIs**
 - Typically provide primitives like `read()` and `write()`
 - Can be used with `select()` to interleave with other operations
 - Users manage their own threads for concurrency (consider Python, Ruby, SmallTalk, ...)
 - Great if your OS threading services can provide real-time guarantees (e.g. some embedded computers, Linux)
- Callback APIs**
 - User provides a function pointer to be called when samples are available/needed
 - Concurrency is implicit, user must be careful with locks or blocking calls
 - You can assume the API is doing its best to be real-time

18

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

PortAudio: An Abstraction of Audio APIs

- PortAudio wraps multiple Host APIs providing a unified and portable interface for writing real-time audio applications
- Main entities:
 - **Host API** – a particular user-space audio API (ie JACK, DirectSound, ASIO, ALSA, WMME, CoreAudio, etc.)
 - PaHostApiInfo, Pa_GetHostApiCount(), Pa_GetHostApiInfo()
 - **Device** – a particular device, usually maps directly to a host API device. Can be full or half duplex depending on the host
 - PaDeviceInfo, Pa_GetDeviceCount(), Pa_GetDeviceInfo()
 - **Stream** – an interface for sending and/or receiving samples to an opened Device
 - PaStream, Pa_OpenStream(), Pa_StartStream()
- See <http://www.portaudio.com>

19

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

PortAudio Example: Generating a Sine Wave

```
struct TestData {
    float sine[TABLE_SIZE];
    int phase;
};

static int TestCallback( const void *inputBuffer,
    void *outputBuffer, unsigned long framesPerBuffer,
    const PaStreamCallbackTimeInfo* timeInfo,
    PaStreamCallbackFlags statusFlags, void *userData ) {
    TestData *data = (TestData*) userData;
    float *out = (float*) outputBuffer;

    for (int i=0; i<framesPerBuffer; i++) {
        float sample = data->sine[ data->phase++ ];
        *out++ = sample; /* left */
        *out++ = sample; /* right */
        if (data->phase >= TABLE_SIZE)
            data->phase -= TABLE_SIZE;
    }
    return paContinue;
}
```

20

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

PortAudio Example: Running a Stream (1)

```
int main(void)
{
    TestData data;
    for (int i=0; i < TABLE_SIZE; ++i)
        data.sine[i] = sin(M_PI * 2 *
                           ((double)i/(double)TABLE_SIZE));

    data.phase = 0;

    Pa_Initialize();

    PaStreamParameters outputParameters;
    outputParameters.device = Pa_GetDefaultOutputDevice();
    outputParameters.channelCount = 2;
    outputParameters.sampleFormat = paFloat32;
    outputParameters.suggestedLatency =
        Pa_GetDeviceInfo(outputParameters.device)->
        defaultLowOutputLatency;
    outputParameters.hostApiSpecificStreamInfo = NULL;

    ...
}
```

21

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

PortAudio Example: Running a Stream (2)

```
...

PaStream *stream;
Pa_OpenStream(&stream, NULL /* no input */,
              &outputParameters,
              SAMPLE_RATE, FRAMES_PER_BUFFER, paClipOff /*flags*/,
              TestCallback, &data);

Pa_StartStream(stream);

printf("Play for %d seconds.\n", NUM_SECONDS);
sleep(NUM_SECONDS);

Pa_StopStream(stream);
Pa_CloseStream(stream);
Pa_Terminate();
}
```

22

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Modular Audio Processing

- Unit generators
- Graph evaluation
- Evaluation mechanisms
- Block-based processing
- Vector allocation strategies
- Variations

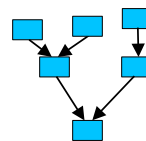
23

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Unit Generators

- A sample generating or processing function, and its accompanying state. e.g. Oscillators, filters, etc.
 - A functional view:
 - $f(\text{state}, \text{inputs}) \rightarrow (\text{state}, \text{outputs})$
 - An OOP view:
 - `Class Ugen{ virtual Update(float*[] ins, float *[] outs); }`
- In a dynamic system, the flow between units is explicitly represented by a “synchronous dataflow graph”



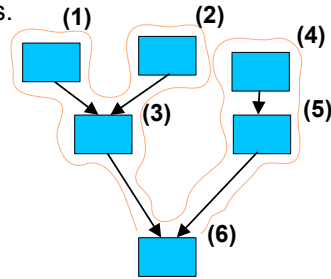
24

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Graph Evaluation

- Generators which produce signals must be evaluated before the generators which consume those signals*, therefore: execute in a depth-first order starting from sinks.
- Note: depth-first implies sinks are the *last* to evaluate in any graph traversal.

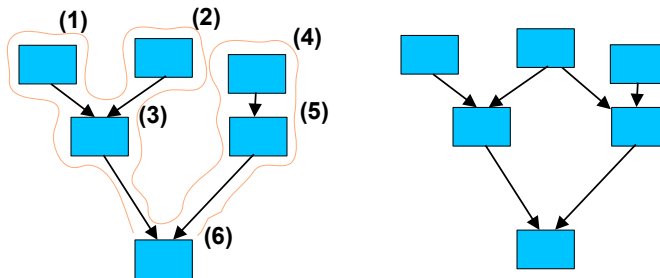


*Why?

*Or else, outputs from generator will not be considered until the next "pass", introducing a one-block delay, or even worse, if outputs go to reusable memory buffers, output could be overwritten.

Evaluation Mechanisms

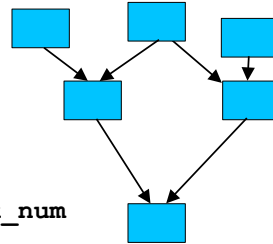
- Direct graph traversal (using topological sort algorithm)
 - Simple, dynamic
 - Can't modify the graph while evaluating



Topological Sort

```
class Ugen
  var block_num
  var inputs

  def update(new_block_num)
    if new_block_num > block_num
      for input in inputs
        input.update(new_block_num)
      really_update() // virtual method
      block_num = block_num + 1
```



Question: Why not just ask each block to update/compute its ancestors before running its own update/compute method instead of messing with block numbers and “if” tests?

27

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Evaluation Mechanisms (2)

- Execution sequence (list of function pointers, polymorphic object pointers, bytecodes)
 - Possibly more efficient, harder to modify
 - Decouples evaluation from traversal. Graph can be modified during traversal; later sequence/program must be computed again.
 - Essentially the same topological sort algorithm is used, but traversal order is stored as a sequence or program.

28

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Block-Based Processing

- Process arrays of input samples and produces arrays of output samples
- Pros: more efficient (common subexpressions, register loads, indexing, cache line prefetching, loop unrolling, SIMD etc)
- Cons: latency, feedback loops incur blocksize delay
- Vector size:
 - fixed (c.f. Csound k-rate, Aura)
 - Variable with upper bound

29

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Variable Block Size

- Rarely used, but this is a good topic to test your understanding of unit generator implementation
- Imagine fixed block size of N and every UG has an inner sample computation loop that runs N times; samples are written to output arrays that hold N samples.
- Now imagine that N is a variable. If the next “event” – some parameter update – is scheduled 5 samples after the start time of the next block, we set N to 5 and all the UGs compute 5 samples. (Remember that all computation is synchronous, so *all* UGs have the same number of input and output samples.)
- After running all the UGs, we get 5 samples of output, do the event/update, and then compute the next value of N .
- We limit N to an upper bound to avoid reallocating buffers of memory that hold samples. These stay at some fixed size N_MAX .
- Main drawback: closely spaced events/updates impact efficiency, so performance is less predictable.

30

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Buffer Allocation Strategies

- 1) One buffer/vector per generated signal, i.e. for every Unit Generator output.
- 2) Reuse buffers once all sinks have consumed them (c.f. Graph coloring register allocation)
- Dannenberg's measurements indicate this is wasted effort
 - Buffers are relatively small
 - Cache is relatively big
 - DSP is relatively expensive compared to (relatively few) cache faults
 - So speedup from buffer reuse (2) is insignificant

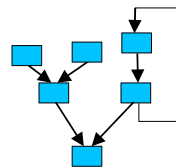
31

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Feedback

- Don't visit a node more than once during graph traversal
- Save output from previous evaluation pass so it can be consumed during next evaluation



32

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Variations on Block-Based Processing

- Hierarchical block sizes e.g. process subgraphs with smaller blocks to reduce feedback delay
- Synchronous multi-rate: separate evaluation phases using the same or different graphs (e.g. Csound krate/arate passes).
- Or support signals with one sample per block time: “Block-rate” UGs have no inner loop and support a sample rate of
$$\text{BLOCK_SR} = \text{AUDIO_SR} / \text{BLOCKSIZE}.$$
- Combine synchronous dataflow graph for audio with asynchronous message processing for control (e.g. Max/MSP)

33

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Audio Plug-Ins

- A *plug-in* is a software object that can extend the functionality of an audio application, e.g. an editor, player, or software synthesizer.
- Effectively a *plug-in* is a unit generator:
 - audio inputs
 - audio outputs
 - parametric controls
- Plug-ins are
 - dynamically loadable and
 - self-describing

34

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

VST Plug-Ins

- Proprietary spec: Steinberg
- Commonly used and widely supported
- Multiplatform:
 - Windows (a multithreaded DLL)
 - Mac OS-X (a bundle)
 - Linux (sort-of)
 - Uses WINE (Windows emulation)
 - Kjetil Matheussen's original [vstserver](#),
 - The [fst](#) project from Paul Davis and Torben Hohn,
 - Chris Cannam's [dssi-vst](#) wrapper plugin

35

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Example VST GUI



jack_fst running the Oberon VSTi synth

36

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

VST Conventions

- *Host* calls plug-in, sets up input buffers and controls buffer size and when processing is performed
- *process()*: must be implemented, output is *added* to the output buffer
- *processReplacing()*: optional, output overwrites data in output buffer
- Parameters range: 0.0 to 1.0 (32-bit float)
- Audio samples: -1.0 to +1.0 (32-bit float)

37

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Example Code

```
AGain::AGain(audioMasterCallback audioMaster)
    : AudioEffectX(audioMaster, 1, 1) // 1 program, 1 parameter only
{ fGain = 1.; // default to 0 dB
  setNumInputs(2); // stereo in
  setNumOutputs(2); // stereo out
  setUniqueID('Gain'); // identify
  canMono(); // makes sense to feed both inputs the same signal
  canProcessReplacing(); // supports both accumulating and replacing
  strcpy(programName, "Default"); // default program name
}

AGain::~AGain() { } // nothing to do here

void AGain::setProgramName(char *name)
{ strcpy(programName, name);
}

void AGain::getProgramName(char *name)
{ strcpy(name, programName);
}
```

38

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Example Code (2)

```
void AGain::setParameter(long index, float value)
{ fGain = value;
}

float AGain::getParameter(long index)
{ return fGain;
}

void AGain::getParameterName(long index, char *label)
{ strcpy(label, "Gain"); // default max string length is 24 (!)
}

void AGain::getParameterDisplay(long index, char *text)
{ dB2string(fGain, text);
}

void AGain::getParameterLabel(long index, char *label)
{ strcpy(label, "dB");
}
```

Example Code (3)

```
bool AGain::getEffectName(char* name)
{ strcpy(name, "Gain");
  return true;
}

bool AGain::getProductString(char* text)
{ strcpy(text, "Gain");
  return true;
}

bool AGain::getVendorString(char* text)
{ strcpy(text, "Steinberg Media Technologies");
  return true;
}
```

Example Code (4)

```
void AGain::process(float **inputs, float **outputs,
                  long sampleFrames)
{
    float *in1 = inputs[0];
    float *in2 = inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];

    while (--sampleFrames >= 0)
    {
        (*out1++) += (*in1++) * fGain;    // accumulating
        (*out2++) += (*in2++) * fGain;
    }
}
```

41

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Example Code (5)

```
void AGain::processReplacing(float **inputs, float **outputs,
                             long sampleFrames)
{
    float *in1 = inputs[0];
    float *in2 = inputs[1];
    float *out1 = outputs[0];
    float *out2 = outputs[1];

    while (--sampleFrames >= 0)
    {
        (*out1++) = (*in1++) * fGain;    // replacing
        (*out2++) = (*in2++) * fGain;
    }
}
```

42

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

VST on the Host Side

```
typedef AEffect *(*mainCall)(audioMasterCallback cb);
audioMasterCallback audioMaster;
void instantiatePlug(mainCall plugsMain)
{ AEffect *ce = plugsMain (&audioMaster);
  if (ce && ce->magic == AEffectMagic) { ... }
}
----- the main() routine in the plugin (DLL): -----
AEffect *main(audioMasterCallback audioMaster)
{ // check for the correct version of VST
  if (!audioMaster(0,audioMasterVersion,0,0,0,0)) return 0;
  ADelay* effect = new ADelay(audioMaster); // Create the AudioEffect
  if (!effect) return 0;
  if (oome) { // Check if no problem in constructor of AGain
    delete effect;
    return 0;
  }
  return effect->getAeffect(); // return C interface of our plug-in
}
```

Assume host loaded plugin and has its main

43

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

More VST

- Program = full set of parameters
- Bank = set of programs (user can call up preset)
- Interactive Interfaces
 - Host can construct editor based on text:
 - Parameter name, display, label – “Gain: -6 dB”
 - Plug-In can open a window and make a GUI
 - Plug-In can use VSTGUI library to make a cross-platform GUI
- VSTi – plug-in instruments (synthesizers)
 - Plug-In has API for receiving MIDI events

44

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

LADSPA – Linux Audio Developers’ Simple Plugin Architecture

- the plugin library is loaded (using a system-specific method like `dlopen` or for `glib`, `gtk+` users, `g_module_open`).
- the plugin descriptor is obtained using the plugin library's `ladspa_descriptor` function, which may allocate memory.
- the host uses the plugin's `instantiate` function to allocate a new (or several new) sample-processing instances.
- the host must connect buffers to every one of the plugin's ports. It must also call `activate` before running samples through the plugin.
- the host processes sample data with the plugin by filling the input buffers it connected, then calling either `run` or `run_adding`. The host may reconnect ports with `connect_port` as it sees fit.
- the host deactivates the plugin handle. It may opt to activate and reuse the handle, or it may destroy the handle.
- the handle is destroyed using the `cleanup` function.
- the plugin is closed. Its `_fini` function is responsible for deallocating memory.

45

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Summary

- Audio samples, frames, blocks
- Synchronous processing:
 - Never skip or duplicate samples
 - Buffers are essential
 - Latency comes (mostly) from buffer length
- PortAudio
 - Host API
 - Device
 - Stream

46

Carnegie Mellon University

© 2019 by Roger B. Dannenberg

Summary (2)

- Modular Audio Processing
 - Unit Generator
 - Networks of Unit Generators
 - Synchronous Dataflow
- Plug-ins
 - VST example
 - Unit Generator that is...
 - Dynamically loadable
 - Self-describing
 - May have its own graphical interface