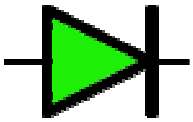
audioGraph

12/6/2011



iOS Audio Processing Graph demonstration

"An **audio processing graph** is a Core Foundation–style opaque type, AUGraph, that you use to construct and manage an audio unit processing chain. A graph can leverage the capabilities of multiple audio units and multiple render callback functions, allowing you to create nearly any audio processing solution you can imagine" *–From Apple's Audio Unit Hosting Guide For iOS*

AudioGraph is a superset of Apple's MixerHost application.

Features include:

- Mono & stereo mic/line input
- Audio effects including:
  - Ring modulator
  - FFT passthrough using Accelerate vDSP framework.
  - Real-time pitch shifting and detection using STFT
  - Simple variable speed delay using a ring buffer
  - Recursive moving average filter with variable number of points
  - Convolution example with variable filter cutoff frequency
- Stereo level meter
- Synthesizer example - sine wave with envelope generator
- iOS 5 features (from Chris Adamson) including:
  - MIDI sampler audio unit
  - file player audio unit
  - audio unit effects
- Runs on iPad, iPhone, and iPod-Touch
- Open source
- Music by Van Lawton
- Everything from MixerHost

Requirements

iPad, iPhone, or iPod-Touch (iOS 5.x)
Headphones.

Instructions

Launch the app and press Play.

Source code, documentation, support

Complete source code in Xcode project format is available from:

http://zerokidz.com/audiograph

Credits

Chris Adamson
Stefan Bernsee
Michael Tyson
Steven W. Smith
Contributors to the Core-Audio mailing list
Contributors to stackoverflow.com
Apple iOS developer program

Thank you.

keywords

core audio, iOS, audio units, audio processing graph, core midi, iPad, iPhone, iPod-Touch, MIDI, Sampler, FFT, Accelerate Framework, DSP, vdsp, objective-C, C, C++, audio, signal processing, digital filters, STFT, audio effects, convolution, open source, iOS 5, callback, streaming, pitch shifting, pitch detection.

Table of Contents

System design and programming

Getting started

I had no intention to write this code. I was working on another project – trying to detect the audio frequency of a car engine. You can program a lot of stuff without knowing anything about Core Audio but eventually you may run into a situation, like I did, where you want to process data at the sample level. If this is your situation you may find this project helpful. At the very least it will give you a working example of the iOS signal processing infrastructure.

There is no definitive set of instructions for Core Audio. Currently the best resource is the work of Chris Adamson, including the draft of his upcoming book “Core Audio”.

The Apple developer resources are great – they lead you to the water’s edge. This project started out as an Apple sample code project called MixerHost. Working code is often more useful than the sort of documentation you’re reading at this very moment.

So if you’re looking for answers, they’re in the code. The source code for this project is available as a free download in Xcode 4.x project format at
https://github.com/tkzic/audiograph

Core Audio programs are typically a hybrid of C, Objective-C and C++. My programming style is “cut and paste”. Some might call it slash and burn. In other words, don’t expect anything you would find in a computer science book. For example, you will see at least three different methods of printing error messages to the console. Why? Code was copied from three different places. The goals here are:

- Make stuff work
- Try to understand why and write it down
- Optimize

You can learn a lot by removing pieces of code and seeing what breaks. My nephew took apart appliances and reassembled them using fewer parts. They would usually work just fine.

Please read the “Audio Unit Hosting Guide For IOS” for the Apple iOS developer library and any other material you can find by Googling “iOS Core Audio. You may not understand it the first time. Unlike human relationships, it will eventually make sense.

Just in case nobody mentioned this: To actually run your Xcode projects on a device you need to become a member of the iOS developer program ($99).

Recommend development setup

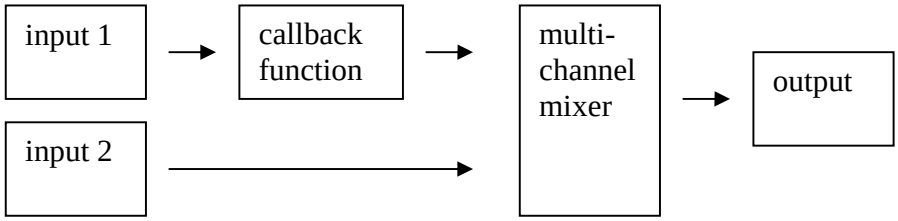
- Device running iOS 5.x
- Mac Computer running OS 10.6 or greater
- Xcode 4.x

The simulator is great but not adequate for testing Core Audio. Try using the newest device you can find, with the most recent operating system. Other useful but non-essential items…

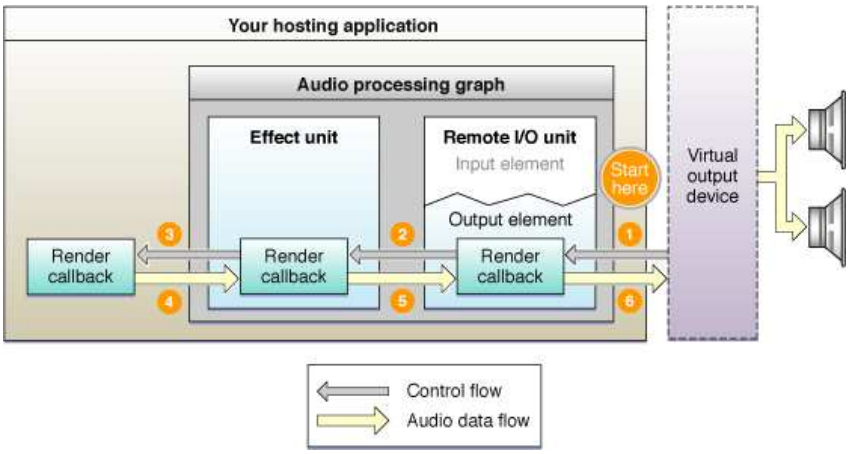
- An iOS MIDI interface (or a WIFI MIDI app like Midi Touch)
- A USB stereo audio interface. (I have tested the Griffin iMic)

Overview

Audiograph is a superset of the MixerHost sample code from Apple. MixerHost provides structure for processing audio at the sample level in callback functions.



This structure is called an audio processing graph. The components are audio units.



Callbacks pull audio from a source, process it, and send it along. Processing typically includes things like:

- filtering
- effects
- analysis
- synthesis

Callbacks can also act as signal generators – synthesizing audio by producing sample data.

Prior to iOS5, audio units were limited to IO and mixing, and format conversion. New audio unit types added in iOS5 include:

- Effects (reverb, filtering, distortion, etc.,)
- MIDI sampler
- file player

In this document I'll focus on specific tasks -things that baffled and confused me - like reading stereo input data and convolution. Hopefully you will benefit from my mistakes.

Source files

In a project with many files it's helpful to know where to look.

Classes

MixerHostAudio.h
MixerHostAudio.m

These files define the MixerHostAudio class and comprise the 'model' for the project. Callbacks, utility functions, instance and class methods. All of the core audio processing happens here.

Classes/MixerHostViewController.h
Classes/MixerHostViewController.m

These files comprise the 'controller' for the project. For simplicity there is only one view controller. All user interface processing happens here.

Views

Resources/en.lproj/MixerHostViewController.xib
Resources/en.lproj/MixerHostViewController~ipad.xib

These files comprise the 'view' for the project. They are the nib files which contain the user interface screens. The iPad and iPhone views share the same controller.

Other Sources

smbPitchShift.m

An adaptation of Stefan Bernsee's STFT pitch shifting functions.

TPCircularBuffer.h
TPCircularBuffer.c

A slightly modified version of Michael Tyson's ring buffer implementation.

SNFCoreAudioUtils.h
SNFCoreAudioUtils.c

Core Audio error printing utility functions from Chris Adamson.

Sound Files

Resources/lead.aupreset
/Sounds/lead.aif

A preset file and it corresponding base sound file for the MIDI sampler.

Sounds/dmxbeat.aiff

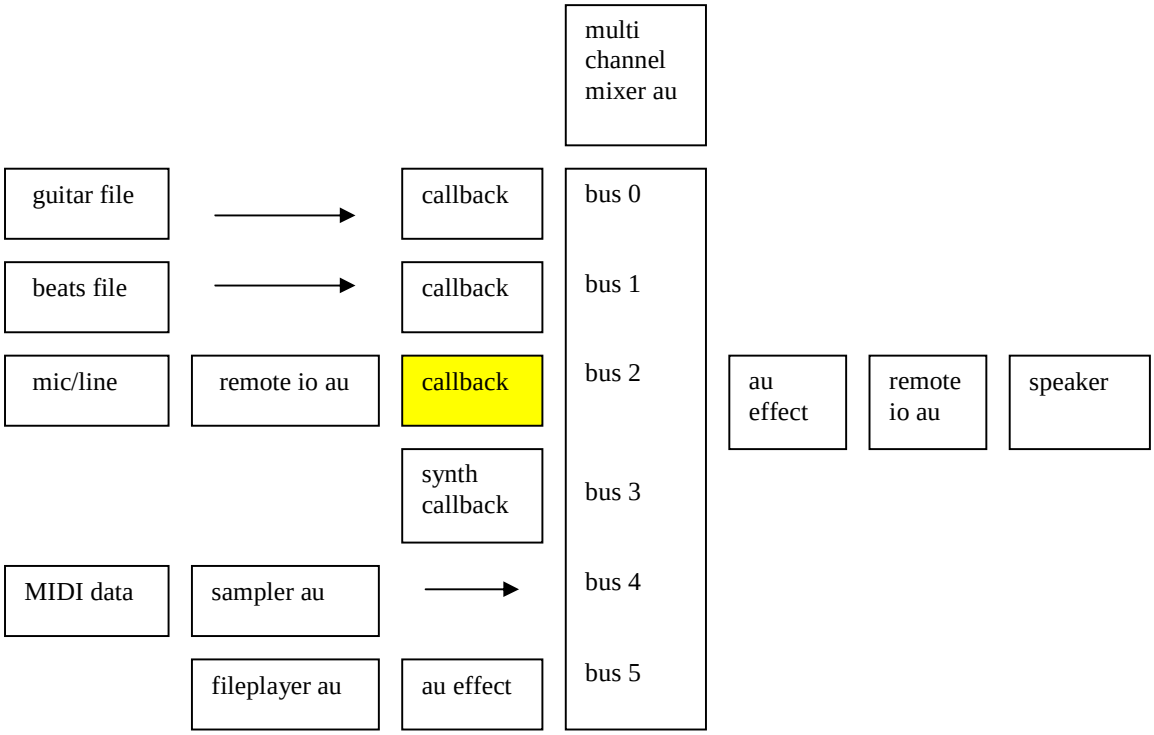
The sound file which is played by the file player audio unit.

Resources/sounds/Caitlin.caf
Resources/sounds/congaloop.caf

Stereo guitar and mono drum loop files, played via streaming callbacks.

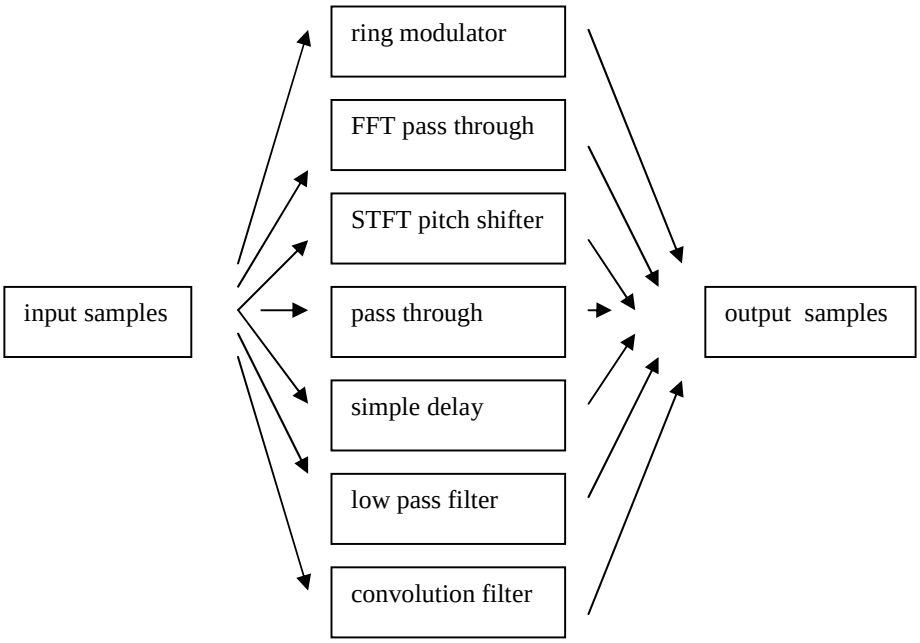
System Design and User Interface

audioGraph Signal Path



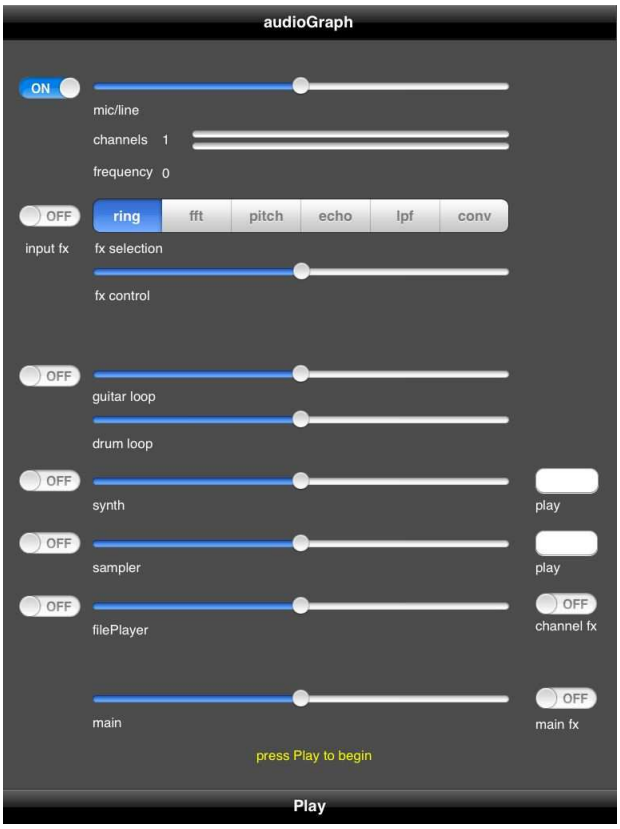
One advantage to audio processing graphs is that the signal path is laid out in an orderly fashion. Connections between audio units can be made directly or through callbacks. The remote IO audio unit serves a dual purpose as an input and output device. It has an input scope and and output scope. In the above diagram it is shown as two separate audio units at the beginning and end of the processing chain. The diagram below show effect processing options for the mic/line input callback function.

Mic/line input callback effect processing



Almost any type of processing can be done in a callback. The only requirement is that data is copied to the output buffer, and that it's done quickly enough to keep up with the sample rate and the frame rate. If these conditions are not met you will hear unpredictable sounds, no sounds, or odd zipper like versions of the sounds you expected.

iPad User Interface



The user interface is arranged to mirror the signal path. Everything is crammed into one screen to minimize UI coding unrelated to audio. The 'Play' button starts and stops the audio processing graph.

Core Audio Setup

The original MixerHost application is a great resource setting up Core Audio programs in iOS. The underlying structure from MixerHost has been left intact in this project.

The key steps are:

1. Setup the audio session
2. Define and configure audio units and stream formats (asbd's)
3. Configure and initialize the audio processing graph

The Apple developer documentation does a great job of explaining how this works. I'll address the confusing aspects of the process.

Setup the Audio Session

The AVAudioSession class configures audio behavior, including sample rate, interruption handling, and input device availability.

This is done in: [setupAudioSession]

Enabling input

By default the audio session is set for output only (playback). If you are using a mic or line in device you will need to set the session to “Play and Record”

```
// Assign the Playback and Record category to the audio session.
    NSError *audioSessionError = nil;
    [mySession setCategory: AVAudioSessionCategoryPlayAndRecord
                     error: &audioSessionError];

    if (audioSessionError != nil) {

        NSLog (@"Error setting audio session category.");

    }
```

Detecting input devices and channels

If you are processing input you will want to know how many channels are available and, in the case of the older iPod-Touch without the built-in mic, whether or not any input device is available.

```
    //
    // check if input is available
    // this only really applies to older ipod touch without builtin mic
    //
    // There seems to be no graceful way to handle this
    //
    // what we do is:
    //
    //  1. set instance var: inputDeviceIsAvailable so app can make decisions
    //     based on input availability
    //  2. give the user a message saying input device is not available
    //  3. set the session for Playback only
    //
    //
    //



    inputDeviceIsAvailable = [mySession inputIsAvailable];


    if(inputDeviceIsAvailable) {
            NSLog(@"input device is available");
    }
    else {
        NSLog(@"input device not available...");
        [mySession setCategory: AVAudioSessionCategoryPlayback
                         error: &audioSessionError];
```

```
    }
```

In the above example, if no input device is detected the session in reset to “playback only” to prevent an error that will occur if the session is started in “playback and record” mode with no input device.

Another idiosyncrasy: The warning message to the user cannot be displayed until the View Controller has completed its loading process. So we set a boolean instance variable: inputDeviceIsAvailable. Then after the loading process is complete the alert message is called in the [viewDidAppear] method of MixerHostViewController.m.

```
- (void) viewDidAppear: (BOOL) animated {

    [super viewDidAppear: animated];
    [[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
    [self becomeFirstResponder];

    // this alert needs to be here, becuase if its in the viewDidLoad sequence you'll
    // get the error: “applications are expected to have a root view controller…”


    if(audioObject.inputDeviceIsAvailable == NO) {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:nil message:@"Mic is not
available. Please terminate the app. Then connect an input device and restart. Or you can
use the app now without a mic."  delegate:self cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [alert show];
        [alert release];

    }

}
```

The only way I have found to tell whether or not an external input device is connected – like a USB audio interface for example – is to call the [currentHardwareInputNumberOfChannels] method, after the audio session is started.

```
// find out how many input channels are available

    NSInteger numberOfChannels = [mySession currentHardwareInputNumberOfChannels];
    NSLog(@"number of channels: %d", numberOfChannels );
    displayNumberOfInputChannels = numberOfChannels;    // set instance variable
```

The displayNumberOfInputChannels instance variable is used throughout the program to determine whether we are dealing with mono or stereo input. That is, one channel means mono – more than one channel means stereo. Of course this is a shortcut and the ideal situation would be to provide a menu to let the user configure mono, stereo, or multichannel input as is done in GarageBand on the iPad.

Setting sample rate and slice buffer size

As you may have already noticed, iOS devices can be stubborn. If you set the sample rate or the currentBufferDuration in the audio session, you aren't guaranteed to get what you asked for. So it's a good idea to actually find out the actual sample rate and buffer size after the audio session is started, using:

[currentHardwareSampleRate] and AudioSessionGetProperty().

```
// Obtain the actual hardware sample rate and store it for later use in the audio
processing graph.
    self.graphSampleRate = [mySession currentHardwareSampleRate];
    NSLog(@"Actual sample rate is: %f", self.graphSampleRate );

    // find out the current buffer duration
    // to calculate duration use: buffersize / sample rate, eg., 512 / 44100 = .012

    // Obtain the actual buffer duration - this may be necessary to get fft stuff working
properly in passthru
    AudioSessionGetProperty(kAudioSessionProperty_CurrentHardwareIOBufferDuration, &sss,
&currentBufferDuration);
    NSLog(@"Actual current hardware io buffer duration: %f ", currentBufferDuration );
```

The above example is a classic example of the unusual mix of C and Objective-C that is Core Audio.

Define Audio Stream Formats (asbd's)

Core Audio uses the asbd (audio stream basic description) structure to set audio formats on the input and output buses of audio units.

The recommended default settings for each type of audio unit can be found in the Apple documentation.

The following methods provide examples of how to set asbd's:

[setupStereoStreamFormat] (fixed point 8.24 (32 bit) non interleaved 2 channel)
[setupMonoStreamFormat] (fixed point 8.24 (32 bit))
[setupSInt16StreamFormat] (Signed 16 bit integer (SInt16))

This is the [setupStereoStreamFormat] method.

```
- (void) setupStereoStreamFormat {

    // The AudioUnitSampleType data type is the recommended type for sample data in audio
    //    units. This obtains the byte size of the type for use in filling in the ASBD.
    size_t bytesPerSample = sizeof (AudioUnitSampleType);


    // Fill the application audio format struct's fields to define a linear PCM,
    //        stereo, noninterleaved stream at the hardware sample rate.
    stereoStreamFormat.mFormatID          = kAudioFormatLinearPCM;
    stereoStreamFormat.mFormatFlags       = kAudioFormatFlagsAudioUnitCanonical;
```

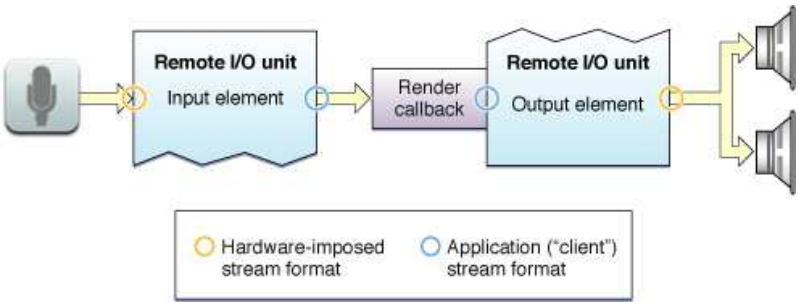
```
    stereoStreamFormat.mBytesPerPacket    = bytesPerSample;
    stereoStreamFormat.mFramesPerPacket   = 1;
    stereoStreamFormat.mBytesPerFrame     = bytesPerSample;
    stereoStreamFormat.mChannelsPerFrame  = 2;                    // 2 indicates stereo
    stereoStreamFormat.mBitsPerChannel    = 8 * bytesPerSample;
    stereoStreamFormat.mSampleRate        = graphSampleRate;


    NSLog (@"The stereo stream format:");
    [self printASBD: stereoStreamFormat];
}
```

In the above example, the `kAudioFormatFlagsAudioUnitCanonical` constant is used to set flag bits indicating

- Fixed point 8.24
- Native endian
- Packed
- Non interleaved

If you look at the flag definitions for this format, it appears at first to be “signed integer” but in fact there is an additional flag bit (unit sample fraction bits) which makes it a fixed point format in iOS. It can be confusing. The [printABSD] method is helpful for examining and debugging stream format issues.

Note: AudioUnitSampleType and AudioSampleType are not the same. In iOS, the former is 8.24 fixed point and the latter is SInt16.



Input device asbd’s

Here are some observations about asbd’s on mic and line inputs.

You can set the remote IO input bus (mic) to SInt16, as long as you are only using one channel. SInt16 format streams will not work with multiple channels, (stereo). Use the canonical AudioUnitSampleType (fixed point 8.24) as shown in the above example. You can always convert samples from one format to another in a callback function. There’s an example of this in micLineInCallback().

I have not been able to get a floating point asbd to work on the input bus of the remote IO unit.

Stick with the canonical AudioUnitSampleType or whatever works.

Audio unit effect asbd's

Now for something completely different. The asbd for audio unit effects is set by *getting* the default asbd setting (which is initialized from the audio unit basic description) and then setting the sample rate. This is done in: [configureAndInitializeAudioProcessingGraph].

```
// get default asbd properties of au effect unit,
// this sets up the auEffectStreamFormat asbd

    UInt32 asbdSize = sizeof (auEffectStreamFormat);
    memset (&auEffectStreamFormat, 0, sizeof (auEffectStreamFormat ));
    CheckError(AudioUnitGetProperty(auEffectUnit, kAudioUnitProperty_StreamFormat,
kAudioUnitScope_Input, 0, &auEffectStreamFormat, &asbdSize),
               "Couldn't get aueffectunit ASBD");

    // debug print to find out what's actually in this asbd

    NSLog (@"The stream format for the effects unit:");
    [self printASBD: auEffectStreamFormat];

    auEffectStreamFormat.mSampleRate = graphSampleRate;   // set sample rate

    // now set this asbd to the effect unit input scope
    // note: if the asbd sample rate is already equal to graphsamplerate
    // then this next statement is not
    // necessary because we derived the asbd from what it was already set to.


    CheckError(AudioUnitSetProperty(auEffectUnit, kAudioUnitProperty_StreamFormat,
kAudioUnitScope_Input, 0, &auEffectStreamFormat, sizeof(auEffectStreamFormat)),
               "Couldn't set ASBD on effect unit input");
```

Note the wrapper function CheckError(). This is a very cool utility by Chris Adamson that simplifies error checking in Core Audio. The Core Audio function is wrapped inside.

By the way in the above example, the effect audio units asbd is set to stereo 32 bit floating point, non interleaved.

asbd ad infinitum

Working with asbd's can be frustrating. With so many parameters it's easy to set something wrong.

In general, the asbd's of connected buses should match. For example, if the output bus of an audio unit effect is connected to the input of a mixer, user the same asbd for both.

The same is true with callbacks. For example, if a callback pulls input data from the remote IO bus and passes it to a mixer input bus, use the same asbd.

The asbd's on the input and output scope of a particular audio unit can be different. For example, various stream formats can be attached to mixer input bus channels even though the mixer output stream format may be completely different. The appropriate conversion is handled automatically within the audio unit.

Helpful Xcode hint: Right-click on any asbd property or constant and select "jump to definition" to go the code where it's defined.

Configure and initialize the audio processing graph

Most of the Core Audio setup code in this project is found in [configureAndInitializeAudioProcessingGraph]. This is where the audio units are defined, configured, and connected.

Here's a rough outline of the steps:

1. Instantiate and open an audio processing graph
2. specify audio unit component descriptions for all units in the graph
3. add nodes to the graph
4. Open graph and get the audio unit nodes for the graph
5. Configure the remote IO unit
6. Configure the Multichannel Mixer unit
    a. Specify the number of input buses, output sample rate, and maximum frames-per-slice
    b. Configure each input channel of mixer
        i. Set callback structs
        ii. Set asbd's
7. Configure any other audio units (fx, sampler, fileplayer)
8. Make processing graph connections
9. Start the processing graph
10. Configure audio unit parameters
11. Setup other post graph parameters (midi and fileplayer)

Enabling Input on the RIO audio unit

By default, input is disabled on the remote IO audio unit. Here's how to make it work.

```
AudioUnitElement ioUnitInputBus = 1;

        // Enable input for the I/O unit, which is disabled by default. (Output is
        //  enabled by default, so there's no need to explicitly enable it.)
        UInt32 enableInput = 1;

        AudioUnitSetProperty (
                          ioUnit,
                          kAudioOutputUnitProperty_EnableIO,
                          kAudioUnitScope_Input,
                          ioUnitInputBus,
```

```
                            &enableInput,
                            sizeof (enableInput)
                            );
```

The remote IO audio unit has a split personality. It's used for connecting to input devices like the microphone (input scope) and output devices like speakers (output scope).

In the above example we're setting the input bus on the input scope of the remote IO audio unit.

Connections, callbacks, and the end of the line

Here are a few important considerations for making connections in an audio processing graph.

Connections or Callbacks, not both: If you connect two audio units using a callback then you should not explicitly connect their nodes using AUGraphConnectNodeInput().

Conversely, you should't define a callback on audio units that will be connected using AUGraphConnectNodeInput().

Pass the class: When setting the *inRefCon struct to pass to a callback function, rather than defining a special struct to pass the data needed by the callback, its much easier to pass a reference to the entire class.

In the following example the inputProcRefCon is set to 'self'

```
UInt16 busNumber = 2;      // mic channel on mixer

        // Setup the structure that contains the input render callback
        AURenderCallbackStruct inputCallbackStruct;

        inputCallbackStruct.inputProc        = micLineInCallback;   // 8.24 version
        inputCallbackStruct.inputProcRefCon  = self;


        NSLog (@"Registering the render callback - mic/lineIn - with mixer unit input bus
%u", busNumber);
        // Set a callback for the specified node's specified input
        result = AUGraphSetNodeInputCallback (
                                              processingGraph,
                                              mixerNode,
                                              busNumber,
                                              &inputCallbackStruct
                                              );

        if (noErr != result) {[self printErrorMessage: @"AUGraphSetNodeInputCallback
mic/lineIn" withStatus: result]; return;}
```

Inside the callback you will have access to all of the instance variables in the MixerHostAudio class. This is a huge convenience.

No asbd at the end of the line: Technically the final audio unit in the processing graph is the output scope of the remote IO unit. But in pragmatic terms consider what comes just before that. In our case it's an audio unit effect tacked on to the mixer output as a master effect. In any case, the only parameter that needs to be set is the sample rate. The processing graph will handle everything else.

```
NSLog (@"Setting sample rate for au effect unit output scope");
    // Set the mixer unit's output sample rate format. This is the only aspect of the
output stream
    //    format that must be explicitly set.
    result = AudioUnitSetProperty (
                                   auEffectUnit,
                                   kAudioUnitProperty_SampleRate,
                                   kAudioUnitScope_Output,
                                   0,
                                   &graphSampleRate,
                                   sizeof (graphSampleRate)
                                   );

    if (noErr != result) {[self printErrorMessage: @"AudioUnitSetProperty (set au effect
unit output stream format)" withStatus: result]; return;}
```

Communicating with the user interface

If you grew up in the old country you're probably not thrilled with the Model-View-Controller paradigm. But here we are.

Methods in the MixerAudioHost class should not directly control the user interface. This is the realm of the view controller. But the view controller has access to methods and properties of MixerAudioHost.

Look at the [viewDidLoad] method in MixerHostViewController.m

```
- (void) viewDidLoad {

    [super viewDidLoad];

    MixerHostAudio *newAudioObject = [[MixerHostAudio alloc] init];
    self.audioObject = newAudioObject;
    [newAudioObject release];


    [self registerForAudioObjectNotifications];
    [self initializeMixerSettingsToUI];


}
```

This is the code that gives the view controller to access the MixerHostAudio class using the reference: audioObject. For example,

[audioObject playSynthNote]; invokes the [playSynthNote] method from within the view controller.

From the perspective of the MixerHostAudio class - if methods in MixerHostAudio need to communicate with the view controller, they should set instance variables which the view controller can discover.

Timer loop

Here's an example to illustrate the behavior described above. In the micLineInCallback() The level of the input signal is measured for each slice of sample data. Then it's saved in the MixerHostAudio instance variables: displayInputLevelLeft and displayInputLevelRight.

```
    // get average input volume level for meter display
    //
    // (note: there's a vdsp function to do this but it works on float samples



   THIS.displayInputLevelLeft = getMeanVolumeSint16( sampleBufferLeft, inNumberFrames);
// assign to instance variable for display
    if(isStereo) {
        THIS.displayInputLevelRight = getMeanVolumeSint16(sampleBufferRight,
inNumberFrames); // assign to instance variable for display
      }
```

Now back to the view controller, in [initializeMixerSettingsToUI], we start up a timer.

```
[NSTimer scheduledTimerWithTimeInterval:0.1
                                  target:self
                                selector:@selector(myMethod:)
                                userInfo:audioObject
                                 repeats: YES];
```

The timer invokes a callback called [myMethod] every 100 milliseconds.

Inside [myMethod] the view controller discovers current values of the MixerHostAudio instance variables and displays them as progress indicators on the screen. They actually look like level meters if you're sleepy.

```
// This is the timer callback method
// sorry about the name
//
// it checks the value of instance variables in MixerHostAudio
// and displays them at regular intervals

// in the crazy convoluted world of objective-c
// userInfo conveniently points to AudioObject
//
- (void) myMethod: (NSTimer *) timer {

//  float z = [[timer userInfo] frequency];
//    UInt32 y = [[timer userInfo] micLevel];
    int numChannels;

    micFreqDisplay.text = [NSString stringWithFormat:@"%d",
                           [[timer userInfo] displayInputFrequency]];
```

```
    numChannels = [[timer userInfo] displayNumberOfInputChannels];
    numberOfInputChannelsDisplay.text = [NSString stringWithFormat:@"%d",numChannels];



    inputLevelDisplayLeft.progress =  [[timer userInfo] displayInputLevelLeft];

    if(numChannels == 1) {      // if mono duplicate left channel meter
        inputLevelDisplayRight.progress = [[timer userInfo] displayInputLevelLeft];
    }
    else if(numChannels == 2) {                     // otherwise use separate data for
right channel
        inputLevelDisplayRight.progress = [[timer userInfo] displayInputLevelRight];
    }

    // note: we're asuuming that we'll only have 1 or 2 channels (mono or stereo)
    // If zero input channels the program would have exited from AVSession init
    // if more than two, we actually need to do some work to find out what they are and
    // provide a UI to configure them



}
```

This form of communication is like leaving notes for family members rather than talking to them directly.

Handling events and interruptions

What happens when you unplug a headset? The audioRouteChangeListenerCallback() handles this type of event. This code from the MixerHost application has been left intact.

Notes on Audio Units

Effects Units

Audio unit effects are new in iOS 5.

Audio unit effects are probably the easiest type of audio unit to configure.

1. Specify the audio unit component description
2. Add the unit to the processing graph
3. Set the asbd
4. Connect the processing graph node
5. After starting the processing graph, set effect parameters

As of iOS 5.0.1 there is a bug in the bypass parameter for some of the audio unit effects. After the first time the effect is bypassed it doesn’t start back up again when it is un-bypassed. This happens with the distortion and reverb effects. The lowpass and highpass filter effects, used in this project, are working properly.

MIDI Sampler Unit

(under construction)

The MIDI Sampler audio unit is new to iOS 5. The sampler plays AUpreset files. To make an AUpreset file, use the AU Lab application located in:

/Developer/Application/Audio/AU Lab

The MIDI Sampler code in this project was adapted from the Apple LoadPresetDemo and Chris Adamson's VTMAUGraphDemo.

MIDI device connections

I have not tested MIDI hardware interfaces with iOS but I have successfully used an app called touchMIDI. touchMIDI running in the background creates a MIDI via WIFI connection to the iPad.

This, for example, allows you to play the MIDI sampler in this program, from a MIDI keyboard connected to a Macbook – or from any applications running in Mac OS which generates MIDI data, like Garageband, Max/MSP, or Ableton Live.

Fileplayer Unit

(under construction)

The Fileplayer audio unit is new to iOS 5.

The Fileplayer code in this project was adapted from Chris Adamson's VTMAUGraphDemo.

Audio callbacks

Audio callbacks are where the magic happens. An audio callback pulls sample data into the input bus of an audio unit. It all happens very fast. For example, at a sample rate of 44.1 KHz and a slice size of 1024 frames, the callback function runs 43 times per second. The prime directive of an audio callback function is to fill its output buffers with sample data.

What can you accomplish in 23 milliseconds? As it turns out, quite a lot. For example, effects processing, analysis, pitch shifting, and filtering. All these can be accomplished in real time on an iOS device.

We will look at two callbacks:

1. the mic/line input callback which pulls data from the output bus of the input scope of the remote IO audio unit into an input bus of the multichannel mixer.
2. the synth callback which generates sample data that gets pulled into an input bus of the multichannel mixer

Things to avoid in audio callbacks:

- Anything that takes a lot of time
  - Allocating buffers
  - Objective-C
  - User interface processing
- Blocking processes
  - User input
  - Reading files

Mic/Line input callback

The mic/line input callback gets data by “rendering” incoming audio from the remote IO audio unit – which is directly connected to a microphone or input device.

*inRefCon

The *inRefCon argument to the callback function is a pointer to a scope or context which may contain data that is needed by the callback. For convenience this is set to the MixerHostAudio class – making available all the instance variables of the class.

```
// scope reference that allows access to everything in MixerHostAudio class

    MixerHostAudio *THIS = (MixerHostAudio *)inRefCon;
```

For example, in the callback, THIS.isStereo would refer to the MixerHostAudio instance variable isStereo.

Audio rendering

Here is an example of rendering data from the remote IO audio unit inside the callback function. It uses AudioUnitRender().

```
// copy all the input samples to the callback buffer - after this point we could bail and
have a pass through
```

```
    renderErr = AudioUnitRender(rioUnit, ioActionFlags,
                                inTimeStamp, bus1, inNumberFrames, ioData);
    if (renderErr < 0) {
        return renderErr;
    }
```

At this point, if you wanted to pass audio through, without additional processing, the callback could return, having completed its mission.

Sample type conversion

As mentioned in the section on asbd's, the default sample format for iOS input devices is 8.24 fixed point. There are only a few kids in the neighborhood who understand fixed point math. A typical signal processing textbook will present examples in floating point or integer. There are examples of both in this project.

So after rendering the sample data the next priority is to convert the data. We are converting to SInt16 (Signed 16 bit integer) format – as a first step and then later converting to floating point if necessary.

Here are the functions to convert fixed point 8.24 to SInt16 and back again.

```
//////////////////////////////////////////////////////
// convert sample vector from fixed point 8.24 to SInt16
void fixedPointToSInt16( SInt32 * source, SInt16 * target, int length ) {

    int i;

    for(i = 0;i < length; i++ ) {
        target[i] =  (SInt16) (source[i] >> 9);

    }

}

//////////////////////////////////////////////////////
// convert sample vector from SInt16 to fixed point 8.24
void SInt16ToFixedPoint( SInt16 * source, SInt32 * target, int length ) {

    int i;

    for(i = 0;i < length; i++ ) {
        target[i] =  (SInt32) (source[i] << 9);
        if(source[i] < 0) {
            target[i] |= 0xFF000000;
        }
        else {
            target[i] &= 0x00FFFFFF;
        }

    }

}
```

To convert fixed point to integer, shift the data 9 bits to the right. To go from integer to fixed point, shift the data 9 bits to the left and extend the sign bit by masking. Some resolution is lost going from 24 to 16 bits but it's not critical for a typical iOS application.

Stereo data and Interleaving

The default for sample data in iOS is non-interleaved. That is, each channel is stored in a contiguous block. Here is an example of how to access left and right channel data. In fact it's the fixed point to SInt16 conversion described above.

```
// convert to SInt16


    inSamplesLeft = (AudioUnitSampleType *) ioData->mBuffers[0].mData; // left channel
    fixedPointToSInt16(inSamplesLeft, sampleBufferLeft, inNumberFrames);

    if(isStereo) {
        inSamplesRight = (AudioUnitSampleType *) ioData->mBuffers[1].mData; // right
channel
        fixedPointToSInt16(inSamplesRight, sampleBufferRight, inNumberFrames);
    }
```

After rendering, sample data is stored in a buffer list. In this case ioData->mBuffers[0].mdata is the left channel. For example, ioData->mBuffers[0].mdata[3] would be the fourth sample in the left channel buffer. Got that?

Stereo effects processing

Without optimization some of the effects work ok in stereo on a classic iPod-Touch, but most exhibited disturbing zippering sounds as the callback churned and labored to meet its prime directive. For now stereo channel data is summed and processed in a single channel.

Signal processing in audio callbacks

This section describes the signal processing examples which are attached to the mic/line input callback. To me this is the most exciting part of the project. It was around 4 in the morning when I finally debugged the pitch-shifting code and heard a Darth Vader version of my voice in the earbuds. Sadly the people trying to sleep, in other rooms of the house, were only hearing an insomniac talking to his iPod.

Ring Modulator

A ring modulator multiplies two input signals in the time domain. The result is a signal which contains the sum and difference of the input signals. In this example the mic/line input is multiplied by a sine wave. The frequency of the sine wave is set by the fx control slider and varies from .00001 Hz to 4000 Hz.

Here is the ringMod() function. It is called from the mic/line callback function.

```
//////////////////////////////////////////
// ring modulator effect - for SInt16 samples
//
// called from callback function that passes in a slice of frames
//
void ringMod(
             void *inRefCon,               // scope (MixerHostAudio)

             UInt32 inNumberFrames,        // number of frames in this slice
             SInt16 *sampleBuffer) {      // frames (sample data)

    // scope reference that allows access to everything in MixerHostAudio class

    MixerHostAudio* THIS = (MixerHostAudio *)inRefCon;

    UInt32 frameNumber;     // current frame number for looping
    float theta;            // for frequency calculation
    static float phase = 0; // for frequency calculation
    float freq;             // etc.,
    AudioSampleType *outSamples;    // convenience pointer to result samples

    outSamples  = (AudioSampleType *) sampleBuffer; // pointer to samples

    freq = (THIS.micFxControl * 4000) + .00001; // get freq from fx control slider
    // .00001 prevents divide by 0

    // loop through the samples

    for (frameNumber = 0; frameNumber < inNumberFrames; ++frameNumber) {

        theta = phase * M_PI * 2;   // convert to radians
        outSamples[frameNumber] = (AudioSampleType) (sin(theta) *
outSamples[frameNumber]);

        phase += 1.0 / (THIS.graphSampleRate / freq);   // increment phase
        if (phase > 1.0) {                              // phase goes from 0 -> 1
            phase -= 1.0;
        }

    }



}
```

A ring modulator is a simple effect to program. It illustrates the basic processing loop which iterates through a slice of frames generating output samples from input samples.

When generating signals one sample at a time the sample rate is critical. For example, if the sample rate is 44.1 KHz, each iteration of the processing loop will generate 1/44100th of the cycle.

In this example, and in the synthesizer callback, we calculate the sample value from the instantaneous phase value. In this example, phase runs from 0->1 in 1/44100 increments and is converted to radians for the sin() function.

Keep in mind that we're working with Integer samples. The results of any calculations should be cast to SInt16 format. Fun fact: In iOS (AudioSampleType) is SInt16. But (AudioUnitSampleType) is fixed point 8.24.

One final observation - after each invocation of ringMod() the phase value needs to be preserved for the next slice of sample data. This is handled by a static local variable.

Simple delay

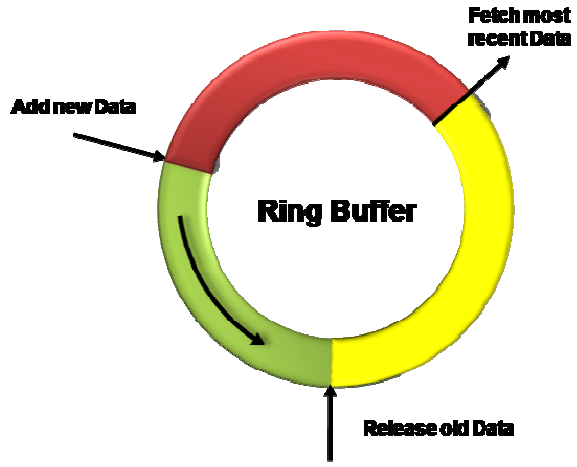
The simple delay (echo) example mixes the current input signal with the input signal from some point in the past. The fx control slider determines the length of the delay.

There are two extra steps required prior to the processing loop. Current sample data needs to be saved. And the delayed data needs to be retrieved. You'll need to save enough sample data to equal the amount of the delay. For example, if the delay is one second at a sample rate of 44.1 KHz, the delay buffer should hold at least 44100 samples. This delay line is implemented with a ring buffer.

We're using an adaptation of the TPCircularBuffer class written by Michael Tyson
https://github.com/michaeltyson/TPCircularBuffer

A ring buffer is typically separates the process of writing and reading data. The write pointer or "head" is always kept ahead of the read point ("tail") by a sufficient amount to allow for timing differences between reading and writing.

Without a ring buffer you would need a buffer size equal to the maximum possible amount of data that would be processed. In the case of real time audio processing, the buffer would need to be infinitely large. So in a sense ring buffers are a computer science version of recycling.



With a delay line the tail position is determined by subtracting the delay length from the head. Also. locking isn't needed since the write and read processes happen in the same thread.

The amount of delay is determined by the distance of the tail from the head. Our adaptation of TPCircularBuffer allows re-calculation of the tail based on the amount of delay.

Here is the code that writes data into the ring buffer.

```
    int32_t tail;           // tail of ring buffer (read pointer)

    SInt16 *targetBuffer, *sourceBuffer;   // convenience pointers to sample data


    SInt16 *buffer;         //
    int sampleCount = 0;                    // number of samples processed in ring buffer
    int samplesToCopy = inNumberFrames;     // total number of samples to process
    int32_t length;                         // length of ring buffer
    int32_t delayLength;                    // size of delay in samples
    int delaySlices;    // number of slices to delay by


    // Put audio into circular delay buffer

    // write incoming samples into the ring at the current head position
    // head is incremented by inNumberFrames


    // The logic is a bit different than usual circular buffer because we don't care
    // whether the head catches up to the tail - because we're going to manually
    // set the tail position based on the delay length each time this function gets
    // called.

    samplesToCopy = inNumberFrames;

    sourceBuffer = sampleBuffer;
    length = TPCircularBufferLength(&delayBufferRecord);
    // printf("length: %d\n", length );


    while(samplesToCopy > 0) {
        sampleCount =  MIN(samplesToCopy, length -
TPCircularBufferHead(&delayBufferRecord));
        if(sampleCount == 0) {
            break;
        }
        buffer = delayBuffer + TPCircularBufferHead(&delayBufferRecord);
        memcpy( buffer, sourceBuffer, sampleCount*sizeof(SInt16)); // actual copy
        sourceBuffer += sampleCount;
        samplesToCopy -= sampleCount;
        TPCircularBufferProduceAnywhere(&delayBufferRecord, sampleCount);  // this
increments head
    }
```

It looks more complicated that it is. With ring buffers you can’t just copy in a chunk of data. If the head is closer to the physical end of the buffer than the length of the incoming data, then the copy must be done in 2 steps. The remaining data gets copied to the physical start of the buffer. If this is confusing, try sketching some diagrams of various data sizes and head positions. The difficulty lies in imposing a circular construct on a finite linear buffer.

Here's the code to retrieve the data. The position of the tail (read pointer) is determined by the delay length which the user controls with the fx slider. Copying data out of the ring buffer involves the same two step process as described above.

```
// Now we need to calculate where to put the tail - note this will probably blow
    // up if you don't make the circular buffer big enough for the delay

    delaySlices = (int) (THIS.micFxControl * 80);

    delayLength = delaySlices * inNumberFrames;      // number of slices do delay by
    // printf("delayLength: %d\n", delayLength);
    tail = TPCircularBufferHead(&delayBufferRecord) - delayLength;
    if(tail < 0) {
        tail = length + tail;
    }



    TPCircularBufferSetTailAnywhere(&delayBufferRecord, tail);


    targetBuffer = tempDelayBuffer; // tail data will get copied into temporary buffer
    samplesToCopy = inNumberFrames;



    // Pull audio from playthrough buffer, in contiguous chunks



    // this is the tricky part of the ring buffer where we need to break the circular
    // illusion and do linear housekeeping. If we're within 1024 of the physical
    // end of buffer, then copy out the samples in 2 steps.

    while ( samplesToCopy > 0 ) {
        sampleCount = MIN(samplesToCopy, length -
TPCircularBufferTail(&delayBufferRecord));
        if ( sampleCount == 0 ) {
            break;
        }
        // set pointer based on location of the tail

        buffer = delayBuffer + TPCircularBufferTail(&delayBufferRecord);



        memcpy(targetBuffer, buffer, sampleCount*sizeof(SInt16)); // actual copy

        targetBuffer += sampleCount;    // move up target pointer
        samplesToCopy -= sampleCount;   // keep track of what's already written
        TPCircularBufferConsumeAnywhere(&delayBufferRecord, sampleCount);  // this
increments tail
    }
```

At this point we have a slice of delayed samples and a slice of current samples. They are mixed together by scaling and adding.

```
// convenience pointers for looping

    AudioSampleType *outSamples;
    outSamples = (AudioSampleType *) sampleBuffer;
```

```
    // mix the delay buffer with the input buffer

    // so here the ratio is .4 * input signal
    // and .6 * delayed signal

    for ( i = 0; i < inNumberFrames ; i++ ) {
        outSamples[i] = (.4 * outSamples[i]) + (.6 * tempDelayBuffer[i]);
    }
```

There are simpler ways to implement a delay line. But you will probably find the ring buffer a useful tool for other audio processing tasks. We will use the same algorithm in the next two sections to implement digital filters

Recursive moving average (lowpass) filter

This example illustrates a simple moving average filter with a variable number of points (3->101) determined by the fx control slider. Increasing the number of points is roughly equivalent to lowering the cutoff frequency.

$$y[80] = \frac{x[78] + x[79] + x[80] + x[81] + x[82]}{5}$$

The filter algorithm is adapted from Steven W. Smith's book "The Scientist and Engineers Guide to Digital Signal Processing" available free, online at http://www.dspguide.com

The algorithm is found in table 15-2.

A recursive filter runs faster because it uses the results of previously calculated samples to calculate new ones.

This algorithm requires a buffer of input samples equal to the size of the input signal plus the size of the filter minus 1, which, by the way, is the requirement for any digital filter based on convolution.

So if our callback function needs 1024 samples and the filter size is 101 we'll need a buffer with at least 1034 samples. To get the extra samples we'll use a ring buffer as described above in the Simple Delay.

One other housekeeping task: We need to convert sample data from SInt16 to floating point. The Accelerate vDSP framework provides functions to do this:

```
//    ConvertInt16ToFloat
```

```
    vDSP_vflt16((SInt16 *) sampleBuffer, stride, (float *) analysisBuffer, stride,
bufferCapacity );
```

The stride factor relates to interleaving. In this case the data is non-interleaved so the stride = 1. There is an excellent discussion of stride factors in the Apple iOS developer library: “vDSP programming guide”.

The size of the filter (and the position of the tail in the ring buffer) is determined by the position of the fx control slider.

Here is the code to run the filter.

```
// ok now we have enough samples in the temp delay buffer to actually run the
    // filter. For example, if slice size is 1024 and filterLength is 101 - then we
    // should have 1124 samples in the tempDelayBuffer


    signalBuffer = tempCircularFilterBuffer;
    resultBuffer = THIS.outputBuffer;



    acc = 0;  // accumulator - find y[50] by averaging points x[0] to x[100]

    for(i = 0; i < filterLength; i++ ) {
        acc += signalBuffer[i];
    }


    resultBuffer[0] = (float) acc / filterLength;

    // recursive moving average filter

    middle = (filterLength - 1) / 2;


    for ( i = middle + 1; i < (inNumberFrames + middle) ; i++ ) {
        acc = acc + signalBuffer[i + middle] - signalBuffer[i - (middle + 1)];
        resultBuffer[i - middle] = (float) acc / filterLength;
    }
```

All that remains is to convert the data back to SInt16 format – which is done using another vDSP function.

```
// now convert from float to Sint16

    vDSP_vfixr16((float *) resultBuffer, stride, (SInt16 *) sampleBuffer, stride,
bufferCapacity );
```

Low Pass convolution filter

This example is a low pass windowed-sinc filter with a variable cutoff frequency implemented by convolution.

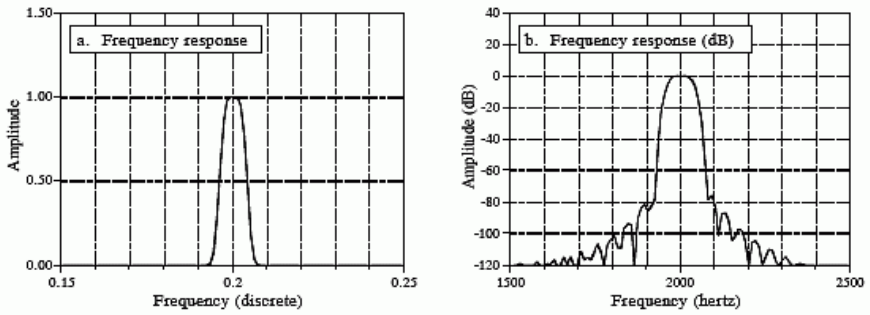


FIGURE 16-6
Example of a windowed-sinc band-pass filter. This filter was designed for a sampling rate of 10 kHz. When referenced to the analog signal, the center frequency of the passband is at 2 kHz, the passband is 80 hertz, and the transition bands are 50 hertz. The windowed-sinc uses 801 points in the filter kernel to achieve this roll-off, and a Blackman window for good stopband attenuation. Figure (a) shows the resulting frequency response on a linear scale, while (b) shows it in decibels. The frequency axis in (a) is expressed as a fraction of the sampling frequency, while (b) is expressed in terms of the analog signal before digitization.

This algorithm is adapted from Steven W. Smith's book "The Scientist and Engineers Guide to Digital Signal Processing", table 16-1, available at http://www.dspguide.com

The filter is calculated in real time based on the cutoff frequency set by fx control slider. Here is the algorithm. It uses a Hamming window which is calculated at the same time as the filter.

```
//////////////////////////////////////////////////////////////////////////
//
// 101 point windowed sinc lowpass filter from http://www.dspguide.com/
// table 16-1
//
void  lowPassWindowedSincFilter( float *buf , float fc ) {

    // re-calculate 101 point lowpass filter kernel

    int i;
    int m = 100;
    float sum = 0;


    for( i = 0; i < 101 ; i++ ) {
        if((i - m / 2) == 0 ) {
            buf[i] = 2 * M_PI * fc;
        }
        else {
            buf[i] = sin(2 * M_PI * fc * (i - m / 2)) / (i - m / 2);
        }
        buf[i] = buf[i] * (.54 - .46 * cos(2 * M_PI * i / m ));
    }

    // normalize for unity gain at dc


    for ( i = 0 ; i < 101 ; i++ ) {
        sum = sum + buf[i];
    }

    for ( i = 0 ; i < 101 ; i++ ) {
        buf[i] = buf[i] / sum;
    }

}
```

Here is the implementation in convolutionFilter():

```
// get mix fx control for cutoff freq (fc)

    fc = (THIS.micFxControl * .18) + .001;

    // make filter with this fc

    lowPassWindowedSincFilter( filterBuffer, fc);
```

The cutoff frequency .18 is expressed as a fraction of the Nyquist frequency (sample rate / 2). At 44.1Khz the slider range is 22 Hz to 3970 Hz.

Like the moving average filter, the convolution filter also requires floating point conversion and a ring buffer. The tail of the ring buffer is positioned 1124 samples behind the head to accommodate a 1024 sample slice of signal data and a 101 point filter size.

The Accelerate vDSP framework provides a convenient function, vDSP_conv(), to perform convolution and correlation. Here is the code to run the filter:

```
  // ok now we have enough samples in the temp delay buffer to actually run the
    // filter. For example, if slice size is 1024 and filterLength is 101 - then we
    // should have 1124 samples in the tempDelayBuffer


    // do convolution

    filterStride = -1;      // convolution
    vDSP_conv( signalBuffer, stride, filterBuffer + filterLength - 1, filterStride,
resultBuffer, stride,  resultLength, filterLength );
```

The final step is to convert the sample vector from floating point to SInt16 format using the same method as with the moving average filter.

This concludes our excursion through the time domain…

FFT pass through

The fast Fourier transform (fft) converts a time domain signal into the frequency domain. This example illustrates a forward and inverse fft using the Accelerate vDSP framework.

The “vDSP programming guide” from the iOS developer library is an excellent resource. Sample code is included in the guide:

The code for this example is in fftPassThrough().

Here is an outline of the steps involved:

Prior to the running the AU graph:

Allocate buffers and run vDSP_create_fftsetup()

Inside the callback function:

- Convert SInt16 sample vector to floating point using vDSP_vflt16()
- Transform real vector into a split complex array using vDSP_ctoz()
- Run the forward fft using vDSP_fft_zrip()
- Convert the split complex vector to a complex interleaved vector for analysis using vDSP_ztoc()
- Perform analysis using a vector processing loop
- Run the inverse fft using vDSP_fft_zrip()
- Scale the results using vDSP_vsmul()
- Convert the split complex vector back to a complex interleaved vector using vDSP_ztoc()
- Convert floating point format samples to SInt16 using vDSP_vfixr16()

Before your mind glazes over, realize that most of the processing involves format conversion of one sort or another. Let's break it down.

The initial setup is done prior to launching the audio processing graph in [fftSetup]

```
//////////////////////////////////////////////////
// Setup FFT - structures needed by vdsp functions
//
- (void) FFTSetup {

    // I'm going to just convert everything to 1024


    // on the simulator the callback gets 512 frames even if you set the buffer to 1024,
so this is a temp workaround in our efforts
    // to make the fft buffer = the callback buffer,


    // for smb it doesn't matter if frame size is bigger than callback buffer

    UInt32 maxFrames = 1024;    // fft size
```

```
    // setup input and output buffers to equal max frame size

    dataBuffer = (void*)malloc(maxFrames * sizeof(SInt16));
    outputBuffer = (float*)malloc(maxFrames *sizeof(float));
    analysisBuffer = (float*)malloc(maxFrames *sizeof(float));

    // set the init stuff for fft based on number of frames

    fftLog2n = log2f(maxFrames);        // log base2 of max number of frames, eg., 10 for
1024
    fftN = 1 << fftLog2n;                   // actual max number of frames, eg., 1024 -
what a silly way to compute it


    fftNOver2 = maxFrames/2;                // half fft size
    fftBufferCapacity = maxFrames;          // yet another way of expressing fft size
    fftIndex = 0;                           // index for reading frame data in callback
```

```
    // split complex number buffer
    fftA.realp = (float *)malloc(fftNOver2 * sizeof(float));        //
    fftA.imagp = (float *)malloc(fftNOver2 * sizeof(float));        //


    // zero return indicates an error setting up internal buffers

    fftSetup = vDSP_create_fftsetup(fftLog2n, FFT_RADIX2);
    if( fftSetup == (FFTSetup) 0) {
        NSLog(@"Error - unable to allocate FFT setup buffers" );
    }

}
```

Note that fftSetup is an instance variable which will be passed as an argument to the fft function. fftSetup is a pointer to predefined weights arrays (twiddle factors) which boost performance of the fft at runtime.

The important considerations in the setup are

1. Set the fft size (N) to the largest size fft that you plan to use
2. Preallocate any buffers that will be used inside the callback

Inside the callback, the fft function needs a sample vector in a split complex format. Please refer the vDSP programming guide (mentioned above) for details. Suffice to say there is a standard procedure for converting vectors in and out of this format. Here is the code from fftPassThrough() to do format conversion and run the forward fft:

```
// *************** FFT ***************
        // convert Sint16 to floating point

        vDSP_vflt16((SInt16 *) dataBuffer, stride, (float *) outputBuffer, stride,
bufferCapacity );


        //
        // Look at the real signal as an interleaved complex vector by casting it.
        // Then call the transformation function vDSP_ctoz to get a split complex
        // vector, which for a real signal, divides into an even-odd configuration.
        //

        vDSP_ctoz((COMPLEX*)outputBuffer, 2, &A, 1, nOver2);


        // Carry out a Forward FFT transform.

        vDSP_fft_zrip(fftSetup, &A, stride, log2n, FFT_FORWARD);
```

At this point the frequency domain data is stored in the split complex vector: A. In this example we'll find the frequency of the input signal by looking for the bin with the greatest amplitude. The first step is to convert the split complex vector back to an interleaved complex vector. Then we can loop through the vector to analyze the frequency domain samples.

```
        // The output signal is now in a split complex form. Use the vDSP_ztoc to get
        // an interleaved complex vector.
```

```
        vDSP_ztoc(&A, 1, (COMPLEX *)analysisBuffer, 2, nOver2);

        // for display purposes...
        //
        // Determine the dominant frequency by taking the magnitude squared and
        // saving the bin which it resides in. This isn't precise and doesn't
        // necessary get the "fundamental" frequency, but its quick and sort of works...

        // note there are vdsp functions to do the amplitude calcs

        float dominantFrequency = 0;
        int bin = -1;
        for (int i=0; i<n; i+=2) {
            float curFreq = MagnitudeSquared(analysisBuffer[i], analysisBuffer[i+1]);
            if (curFreq > dominantFrequency) {
                dominantFrequency = curFreq;
                bin = (i+1)/2;
            }
        }

        dominantFrequency = bin*(THIS.graphSampleRate/bufferCapacity);

        // printf("Dominant frequency: %f   \n" , dominantFrequency);
        THIS.displayInputFrequency = (int) dominantFrequency;   // set instance variable
with detected frequency
```

The interleaved complex vector is in the format:

real = buffer[i]
imaginary = buffer[i + 1]

from 0 to N/2

MagnitudeSquared() calculates the square of the magnitude ((re * re) + (im * im)). To get the actual magnitude you would take the square root of the result. But we're in kind of a hurry. So the assumption is that the bin with the greatest "magnitude squared" value probably represents the fundamental frequency. Anyway, it is the "dominant" frequency.

The result of the calculation is stored in the instance variable: displayInputFrequency which will be discovered by the view controller.

In the next section we'll explore a more accurate (and costlier) method of pitch detection.

The final step here is to reverse the process. Let's backtrack to the point of the forward fft. The code to undo the fft is almost a mirror image of the steps leading up to the forward transform, except that after the inverse transform, the results need to be scaled back to the original level. Here's the code:

```
// Carry out an inverse FFT transform.

        vDSP_fft_zrip(fftSetup, &A, stride, log2n, FFT_INVERSE );

        // scale it
```

```
        float scale = (float) 1.0 / (2 * n);
        vDSP_vsmul(A.realp, 1, &scale, A.realp, 1, nOver2 );
        vDSP_vsmul(A.imagp, 1, &scale, A.imagp, 1, nOver2 );


        // convert from split complex to complex interleaved

        vDSP_ztoc(&A, 1, (COMPLEX *) outputBuffer, 2, nOver2);

        // now convert from float to Sint16

        vDSP_vfixr16((float *) outputBuffer, stride, (SInt16 *) sampleBuffer, stride,
bufferCapacity );
```

Had we wanted to modify the samples while in the frequency domain – for example, inside the processing loop where we calculated frequency, we would have used vDSP_ctoz() to convert back to a split Complex vector before performing the inverse fft.

We'll see an example of that in the next section.

If all this seems overwhelming, try using the code as a cookbook. When you cook enough pots of Chili it suddenly makes sense.

STFT, pitch shifting and detection

The short time Fourier transform combined with phase analysis gives a more precise picture of frequency, and can be used to perform real-time pitch shifting.

The code in this example is adapted from an excellent article and sample code by Stefan Bernsee from DSP dimension. Please read the article if you want to understand how this code works.

http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/

The adaptation of the code involved:

- Formatting signal vectors (as described in the fftPassThrough example above)
- Replacing fft functions with vDSP fft functions
- Saving frequency analysis data in a MixerHostAudio instance variable for display

First lets look at the wrapper for the STFT code in fftPitchShift().

```
// ConvertInt16ToFloat

    vDSP_vflt16((SInt16 *) sampleBuffer, stride, (float *) analysisBuffer, stride,
bufferCapacity );
```

```
    // run the pitch shift

    // scale the fx control 0->1 to range of pitchShift .5->2.0

    pitchShift = (THIS.micFxControl * 1.5) + .5;

    // osamp should be at least 4, but at this time my ipod touch gets very unhappy with
    // anything greater than 2

    osamp = 4;
    fftSize = 1024;     // this seems to work in real time since we are actually doing
the fft on smaller windows

    smb2PitchShift( pitchShift , (long) inNumberFrames,
                   fftSize,  osamp, (float) THIS.graphSampleRate,
                   (float *) analysisBuffer , (float *) outputBuffer,
                   fftSetup, &frequency);


    // display detected pitch


    THIS.displayInputFrequency = (int) frequency;


    // very very cool effect but lets skip it temporarily
    //    THIS.sinFreq = THIS.frequency;   // set synth frequency to the pitch detected
by microphone



    // now convert from float to Sint16

    vDSP_vfixr16((float *) outputBuffer, stride, (SInt16 *) sampleBuffer, stride,
bufferCapacity );
```

The sample vector is converted from SInt16 to floating point. The pitch shift factor is set by the fx control slider. Then everything gets passed into smb2PitchShift().

The osamp variable specifies the overlap or oversampling factor. This value should be a power of 2 and should be at least four to produce reasonably accurate pitch shifting. Higher numbers work better but require more processing time. There will be a point of diminishing returns. There will also be a point where the callback can't keep up.

Here are the original comments in the source code from Stefan Bernsee.

```
/****************************************************************************
*
* NAME: smbPitchShift.cpp
* VERSION: 1.2
* HOME URL: http://www.dspdimension.com
* KNOWN BUGS: none
*
* SYNOPSIS: Routine for doing pitch shifting while maintaining
* duration using the Short Time Fourier Transform.
*
* DESCRIPTION: The routine takes a pitchShift factor value which is between 0.5
* (one octave down) and 2. (one octave up). A value of exactly 1 does not change
```

```
* the pitch. numSampsToProcess tells the routine how many samples in indata[0...
* numSampsToProcess-1] should be pitch shifted and moved to outdata[0 ...
* numSampsToProcess-1]. The two buffers can be identical (ie. it can process the
* data in-place). fftFrameSize defines the FFT frame size used for the
* processing. Typical values are 1024, 2048 and 4096. It may be any value <=
* MAX_FRAME_LENGTH but it MUST be a power of 2. osamp is the STFT
* oversampling factor which also determines the overlap between adjacent STFT
* frames. It should at least be 4 for moderate scaling ratios. A value of 32 is
* recommended for best quality. sampleRate takes the sample rate for the signal
* in unit Hz, ie. 44100 for 44.1 kHz audio. The data passed to the routine in
* indata[] should be in the range [-1.0, 1.0), which is also the output range
* for the data, make sure you scale the data accordingly (for 16bit signed integers
* you would have to divide (and multiply) by 32768).
```

It may be instructive to compare the original code with the modified version in smb2PitchShift.m to see how the Accelerate vDSP functions were incorporated. There is really not much difference in the use of the fft functions between this example and the fftPassThrough code. But the code here is more complicated and again I would recommend reading the article for a full understanding of STFT and pitch detection using phase change.

Synthesizer Callback

The synthesizer callback generates a sine wave with amplitude controlled by an envelope generator. The envelope is triggered by pressing a button in the user interface.

The code for generating a sine wave is also discussed in the Ring Modulator section.

The synthesizer callback differs from the other callbacks in this project, in that it does not process input samples from a source. It generates samples.

The asbd for the callback is SInt16 format – eliminating the need to convert from fixed point 8.24 and back again.

The callback function is synthRenderCallback()

```
//////////////////////////////////////////////
//  synth callback - generates a sine wave with
//
//  freq = MixerHost.sinFreq
//  phase = MixerHost.sinPhase
//  note on = MixerHost.synthNoteOn
//
//  its a simple example of a synthesizer sound generator
//

static OSStatus synthRenderCallback (
                                     void *                          inRefCon,
                                     AudioUnitRenderActionFlags * ioActionFlags,
                                     const AudioTimeStamp *          inTimeStamp,
                                     UInt32                          inBusNumber,
                                     UInt32                          inNumberFrames,
                                     AudioBufferList *               ioData) {
```

```
    MixerHostAudio* THIS = (MixerHostAudio *)inRefCon;  // scope reference that allows
access to everything in MixerHostAudio class



    float freq = THIS.sinFreq;      // get frequency data from instance variables
    float phase = THIS.sinPhase;


    float sinSignal;                //
    float envelope;                 // scaling factor from envelope generator 0->1


    //  NSLog(@"inside callback - freq: %f phase: %f", freq, phase );


    double phaseIncrement = 2 * M_PI * freq / THIS.graphSampleRate; // phase change per
sample


    AudioSampleType *outSamples;
    outSamples = (AudioSampleType *) ioData->mBuffers[0].mData;



// if a note isn't being triggered just fill the frames with zeroes and bail.
// interesting note: when we didn't zero out the buffer, the microphone was
// somehow activated on the synth channel... weird???
//
// synth note triggering is handled by envelope generator now but I left above comment -
to illustrate
// what can happen if your callback doesn't fill its output data buffers

/*
    if( noteOn == NO ) {
        memset(outSamples, 0, inNumberFrames * sizeof(SInt16));
        return noErr;
    }

*/

// build a sine wave (not a teddy bear)

    for (UInt32 frameNumber = 0; frameNumber < inNumberFrames; ++frameNumber) {


        sinSignal = sin(phase); // if we were using float samples this would be the value


        // scale to half of maximum volume level for integer samples
        // and use envelope value to determine instantaneous level


        // envelope = 1.0;
        envelope = getSynthEnvelope( inRefCon );  // envelope ranges from 0->1


        outSamples[frameNumber] =  (SInt16) (((sinSignal * 32767.0f) / 2) * envelope);
        phase = phase + phaseIncrement; // increment phase

        if(phase >= (2 * M_PI * freq)) {         // phase wraps around every cycle
            phase = phase - (2 * M_PI * freq);
        }

    }



    THIS.sinPhase = phase;      // save for next time this callback is invoked

    return noErr;
```

```
}
```

Frequency, phase, and noteOn data is acquired via instance variables.

The sine wave is generated in steps based on the sample rate. At 44.1 KHz each sample represents 1/44100th of a cycle. Each sample value is calculated by the sin function operating on a phase value which iterates through a cycle from 0->1 (or 0->2 PI radians) split into 44100 steps. The phase value is saved in an instance variable after processing each slice. It could also have been preserved in a static variable, as was done in the ring modulator.

Envelope Generator

The envelope returns a scaling factor from 0->1. It is calculated for each sample and multiplied by the sine wave data. Here is the code for the envelope generator. It is an AR (attack release) generator implemented as a finite state machine.

```
// simple AR envelope generator for synth note
//
// for now, attack and release value params hardcoded in this function
//

#define ENV_OFF 0
#define ENV_ATTACK 1
#define ENV_RELEASE 2

float getSynthEnvelope( void * inRefCon ) {

    MixerHostAudio* THIS = (MixerHostAudio *)inRefCon;  // access to mixerHostAudio scope

    static int state = ENV_OFF;           // current state
    static int keyPressed = 0;            // current(previous) state of key
    static float envelope = 0.0;          // current envelope value 0->1

    float attack = 1000.0;                    // attack time in samples
    float release = 40000.0;                  // release time in samples

    float attackStep;                       // amount to increment each sample during
attack phase
    float releaseStep;                      // amount to decrement each sample during
release phase

    int newKeyState;                       // new on/off state of key

    // start

    attackStep = 1.0 / attack;              // calculate attack and release steps
    releaseStep = 1.0 / release;

    newKeyState = THIS.synthNoteOn == YES ? 1 : 0;

    // printf("envelope: %f, state: %d, keyPressed: %d, newKeyState: %d\n", envelope,
state, keyPressed, newKeyState);

    if(keyPressed == 0) {       // key has been up
        if(newKeyState == 0) {      //  if key is still up
            switch(state)
            {
                case ENV_RELEASE:
```

```
                    // printf("dec: env: %f, rs: %f\n", envelope, releaseStep );
                    envelope -= releaseStep;
                    if(envelope <= 0.) {
                        envelope = 0.0;
                        state = ENV_OFF;
                    }
                    break;
                default:
                    state = ENV_OFF;    // this should already be the case
                    envelope = 0.0;
                    break;
            }
        }
        else {  // key was just pressed
            keyPressed = 1;                 // save new key state
            state = ENV_ATTACK;             // change state to attack

        }
    }
    else {  // key has been down

        if(newKeyState == 0) {      // if key was just released
            keyPressed = 0;         // save new key state
            state = ENV_RELEASE;

        }
        else {                  // key is still down
            switch(state)
            {

                case ENV_ATTACK:
                    // printf("inc: env: %f, as: %f\n", envelope, attackStep );
                    envelope += attackStep;
                    if (envelope >= 1.0) {
                        envelope = 1.0;
                    }
                    break;

                default:
                    state = ENV_ATTACK;         // this should already be the case
                    break;
            }
        }

    }


    return (envelope);
}
```

Attack and release values are hard coded and represented as a “number of samples” – but could easily be converted to milliseconds and controlled in the user interface.

The attack state begins when a key (button) is pressed and rises to its full value during the duration of the attack. It then holds its maximum value. The release state begins when a key is released and continues for the duration of the release – at which point the value of the envelope is 0 – indicating the note is off.

To implement a real synthesizer you would probably add functions to produce wave forms, modulators, filters, and the ability to configure the signal and control path.

The core midi framework could also be used to provide a midi interface for your synthesizer in the same manner as it's used in the midi sampler example.

Development

The source code for this project is in a github repository at:
https://github.com/tkzic/audiograph

I would be very interested in your thoughts and ideas. Please contact me at
audiograph@zerokidz.com

References

Wondering why some of the link fonts (below) are so small? It's a workaround. My ancient word processer has issues converting links spanning more than one line when creating .pdf files.

"Core Audio" by Chris Adamson

This yet to be published book is available in draft form from Safaribooksonline. It's an excellent resource for learning Core Audio on Mac OSx – with sample code.

http://my.safaribooksonline.com/book/audio/9780321636973

iOS Developer Library – AudioUnit Hosting Guide for iOS

This is the document you must pass through on the way to enlightenment.

http://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/Introduction/Introduction.html

iOS Developer Library – Audio Mixer (MixerHost)

MixerHost is the parent. audioGraph is the child.

http://developer.apple.com/library/ios/#samplecode/MixerHost/Introduction/Intro.html

iOS Developer Library – vDSP Programming Guide

An interesting example of technical writing and an amazing signal processing library.

http://developer.apple.com/library/ios/#documentation/Performance/Conceptual/vDSP_Programming_Guide/Introduction/Introduction.html

VTMAUGraphDemo (by Chris Adamson)

Sample code that demonstrates new core audio features in iOS 5

http://www.subfurther.com/blog/2011/11/16/what-you-missed-at-voices-that-matter-ios-fall-2011/

[Time code]; A digital Media development Blog (by Chris Adamson)

The university of Core Audio.

http://www.subfurther.com/blog/

A Tasty Pixel (by Michael Tyson)

The blog is well written and contains excellent examples of Core Audio iOS programming.

http://atastypixel.com/

Pitch Shifting Using The Fourier Transform (by Stefan Bernsee)

This article and accompanying source code unveil the mystery of programming in the frequency domain.

http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/

The Scientist and Engineers Guide to Digital Signal Processing (by Stephen W. Smith)

It's available free online but I actually bought the book.

http://www.dspguide.com/pdfbook.htm

Stackoverflow.com (Core Audio)

Have a question about anything? Someone on stackoverflow has the answer.

http://stackoverflow.com/questions/tagged/core-audio

Apple Core Audio Mailing List Archive

The Jedi Knights of Core Audio.

http://lists.apple.com/archives/coreaudio-api