THE UNIVERSITY of York

Department of Electronics

Audio Lab

Csound for iOS API

A Beginner's Guide 1.1

17/2/2013

Timothy Neate, Nicholas Arner & Abigail Richardson

Abstract

This tutorial aims to help iOS developers with the implementation of the Mobile Csound Platform for iOS. Developers who are looking to incorporate audio into their apps, but do not want to deal with the complexities of Core Audio, will find this particularly useful.

It provides some background information on the API and outlines how to integrate Csound and iOS, and allow them to communicate. The provided example project is then described - outlining the key features of the API. Some common problems that users are likely to encounter are then discussed to troubleshoot potential issues

Acknowledgements

We would like to thank our supervisor, Dr. Andy Hunt, for his support and guidance while working through the Csound for iOS API, as well as working on this tutorial.

We would also like to thank Dr. Victor Lazzarini and Steven Yi, the authors of the Csound for iOS API. We would especially like to thank Steven for his extremely helpful responses to our questions regarding the API.

Table of Contents

1. Introduction

The traditional way of working with audio on both Apple computers and mobile devices is through the use of Core Audio. Core Audio is a low-level API which Apple provides to developers for writing applications utilizing digital audio. The downside of Core Audio being low-level is that it is often considered to be rather cryptic and difficult to implement, making audio one of the more difficult aspects of writing an iOS app.

In an apparent response to the difficulties of implementing Core Audio, there have been a number of tools released to make audio development on the iOS platform easier to work with. One of these is *libpd*, an open-source library released in 2010. *libpd* allows developers to run Pure Data on both iOS and Android mobile devices. Pure Data is a visual programming language whose primary application is sound processing.

The recent release of the Mobile Csound Platform provides an alternative to the use of PD for mobile audio applications. Csound is a synthesis program which utilizes a toolkit of over 1200 signal processing modules, called opcodes. The release of the Mobile Csound Platform now allows Csound to run on mobile devices, providing new opportunities in audio programming for developers. Developers unfamiliar with Pure Data's visual language paradigm may be more comfortable with Csound's 'C'-programming based environment.

For those who are unfamiliar with Csound, or want to learn more, the FLOSS manuals are an excellent resource, and can be found here:

http://flossmanuals.net/csound/

For more advanced topics in Csound programming, the Csound Book (Boulanger ed., 2000) will provide an in-depth coverage.

In order to make use of the material in this tutorial, the reader is assumed to have basic knowledge of Objective-C and iOS development. Apple's Xcode 4.6.1 IDE (integrated development environment) will be used for the provided example project.

Although the Mobile Csound API is provided with an excellent example project, it was felt that this tutorial will be a helpful supplement in setting up a basic Csound for iOS project for the first time, by including screenshots from the project set-up, and a section on common errors the user may encounter when working with the API.

The example project provided by the authors of the API includes a number of files illustrating various aspects of the API, including audio input/output, recording, interaction with GUI widgets, and multi-touch. More information on the example project can be found in the API manual, which is included in the example projects folder.

1.1. The Csound for iOS API

The Mobile Csound Platform allows programmers to embed the Csound audio engine inside of their iOS project. The API provides methods for sending static program information from iOS to the instance of Csound, as well as sending dynamic value changes based on user interaction with standard UI interface elements, including multi-touch interaction.

1.2. Document Structure

This document begins, in Section 2, by describing the example provided by the authors. Section 2 is divided into two further sections: Section 2.1 which describes the functionality of the example application and Section 2.2 which details line by line through the example code how this application works. Section 3 provides a step by step guide to setting up an Xcode project for use with the Mobile Csound API. This section describes how to download the API and include it into the project (Section 3.1) as well as the necessary components of the view controller (Section 3.2) and Csound file (Section 3.3). Section 4 outlines some common problems, which have been found through the creation of this tutorial, and their solutions. Section 5 is a reference of the methods which are available for use in the Mobile Csound API. This section briefly details the functionality of these methods and their method calls. Section 6 provides the authors' conclusions about this tutorial.

NOTE: This tutorial uses Csound 5, and has not been tested with Csound6.

2 Example Walkthrough

This section discusses why the example was made, and what can be learned from it; giving an overview of its functionality, then going into a more detailed description of its code. A copy of the example project can be found at the following link.

https://sourceforge.net/projects/csoundiosguide/

2.1 Running the Example Project

Run the provided Xcode project, CsoundTutorial.xcodeproj, and the example app should launch (either on a simulator or a hardware device). A screenshot of the app is shown in Figure 2.1 below. The app consists of two sliders, each controlling a parameter of a Csound oscillator. The top slider controls the amplitude, and the bottom slider controls the frequency.

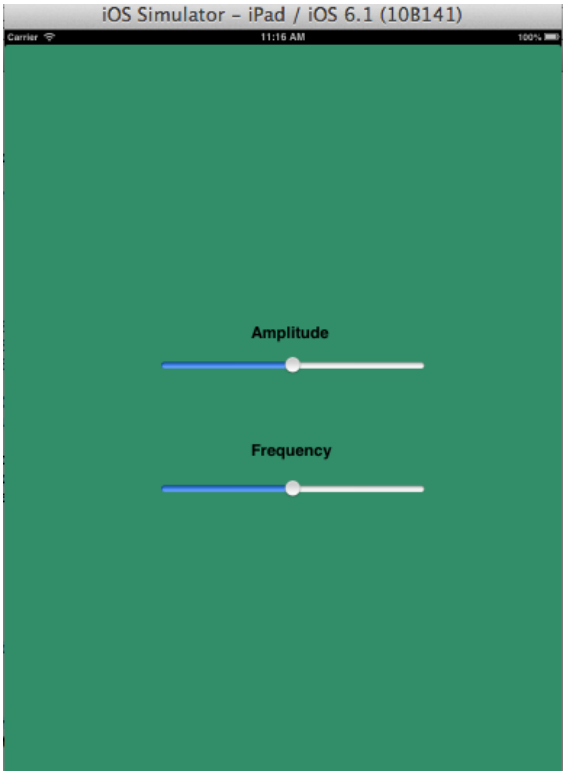

Figure 2.1-App running on iPad simulator

2.2 Oscillator Example Walkthrough

This example outlines how to use the methods in the Csound-iOS API to send values from iOS into Csound. This example was made purposefully simple, with the intent of making its functionality as obvious as possible to the reader. This section begins by giving an overview of both the iOS and Csound implementation, and then describes how this achieved by breaking down the example code. The code to create this oscillator example was done in the *ViewController.h* and the *ViewController.m* files, which are discussed below in sections 2.2.3.1 and 2.2.3.2. The project is split into Objective-C code, Storyboards for the user interface elements, and a Csound file for the audio engine.

2.2.1 iOS Example Outline

In the Xcode project user interface sliders are used to allow a user to control the Csound audio engine through iOS. Communication begins with iOS requesting some memory within Csound; setting a pointer to this location. It updates this pointer with values from the user interface sliders. Csound references the same memory location by naming it with a string, this named communication link is called a channel. When sending this information, iOS uses methods within the iOS-Csound API to setup this channel name, and update it dependant on the control rate.

2.2.2. Csound Example Outline

In this example, Csound is not aware of iOS. All it knows is that there is a piece of memory assigned for it, and it must retrieve information from here dependent on its control rate. Csound uses the *chnget* opcode to do this. *chnget* searches for some channel with a specific name and retrieves values from it.

2.2.3. The iOS File

This example is implemented across two main files:

The **.h file** is used to include all the necessary classes, declare properties, and allow for user interaction by connecting the interface to the implementation.
The **.m file** is used to implement communication between the interface methods declared in the .h file, and the Csound file. These will now be discussed in more depth, with code examples.

2.2.3.1. The .h File

The imports (discussed in detail in section 3.2.1) are declared:

```
#import <UIKit/UIKit.h>
#import "CsoundObj.h"
#import "CsoundValueCacheable.h"
```

Apart from the standard UIKit.h (which gives access to iOS interface widgets) these ensure that the code written can access the information in the other files in the Csound API.
Next comes the class definition:

```
@interface ViewController  : UIViewController
<CsoundObjCompletionListener, CsoundValueCacheable>
```

Every iOS class definition begins with the **@interface** keyword, followed by the name of the class. So our class is called *ViewController*, and the colon indicates that our class inherits all the functionality of the *UIViewController*.

Following this are two Protocol definitions, which are listed between the triangular brackets < >. In Objective-C a Protocol is a list of **required** functionality (i.e., methods) that a class needs to implement. In this case there are two Protocols that are defined by the Csound API, that we want our class to conform to: *CsoundObjCompletionListener* and *CsoundValueCacheable*. This will allow us to send data between iOS and Csound, and so is essential for what we are about to do. The required functions that we have to implement are described in the section following this one (2.2.3.2).

The Csound object needs to be declared as a property in the .h file, which is what this next line of code does:

```
//Declare a Csound object
@property (nonatomic, retain) CsoundObj* csound;
```

The next section of code allows for the interface objects (sliders) to communicate with the .m file:

```
- (IBAction)amplitudeSlider:(UISlider *)sender;
- (IBAction)frequencySlider:(UISlider *)sender;
```

Just to the left of each of these IBAction methods, you should see a little circle. If the storyboard is open (MainStoryboard.storyboard) you will see the appropriate slider being highlighted if you hover over one of the little circles.

2.2.3.2. The .m File

The .m file imports the .h file so that it can access the information within it, and the information that it accesses.

At the beginning of the implementation of the ViewController, the *csound* variable which was declared in the .h file is instantiated with *@synthesize* thus:

```
@implementation ViewController

@synthesize csound = mCsound;
```

Note that the Csound object must be released later in the *dealloc* method as shown below:

```
- (void)dealloc
{
    [mCsound release];
    [super dealloc];
}
```

For each parameter you have in iOS that you wish to send to Csound, you need to do the things outlined in this tutorial. In our simple example we have an iOS slider for Frequency, and one for Amplitude, both of which are values we want to send to Csound.

Some global variables are then declared, as shown in Table 2.1, a holder for each iOS parameter's current value, and a pointer for each which is going to point to a memory location within Csound.

Variable	Description
`float myFrequency;`	This value comes from the frequency slider in the interface. It is a float, as the value to send from iOS to Csound needs to be a floating point number. Its range is 0 – 500.
`float myAmplitude;`	This value comes from the amplitude slider in the interface. Its range is 0 – 1 because of the way the gain is controlled in the .csd file.
`float* freqChannelPtr;`	These variables are used in conjunction with the method *getInputChannelPtr* (described towards the end of this section) to send frequency and amplitude values to Csound.
`float* ampChannelPtr;`	

Table 2.1-Variables for the .m File

The next significant part of the .m file is the *viewDidAppear* method. When the view loads, and appears in iOS, this iOS SDK method is called. In the example, the following code is used to locate the Csound file:

```
NSString *tempFile = [[NSBundle mainBundle] pathForResource:@"aSimpleOscillator" ofType:@"csd"];
NSLog(@"FILE PATH: %@", tempFile);
```

This code searches the main bundle for a file called *aSimpleOscillator* of the type *csd* (which you will be able to see in Xcode's left-hand File List, under the folder Supporting Files). It then assigns it to an *NSString* named *tempFile*. The name of the string *tempFile* is then printed out to confirm which file is running.

The methods shown in Table 2.2 are then called:

Method Call	Description
`self.csound = [[CsoundObj alloc] init];`	This instantiates the csound object, which will be our main contact between iOS and Csound. It allocates and initialises some memory to make an instance of the CsoundObj class.
`[self.csound addCompletionListener:self];`	Sets our code (self – i.e. ViewController) to be a listener for the Csound object.
`[self.csound addValueCacheable:self];`	Sets our code (self) to be able to send real-time values to the Csound object.
`[self.csound startCsound:tempFile];`	The Csound object uses the method *startCsound* to run the file at the string *tempFile*. Remember how *tempFile* was set up to point to the Csound csd file (in our case *aSimpleOscillator.csd*). So, in other words, this line launches Csound with the csd file you have provided.

Table 2.2-Csound API Methods

The methods that allow the value of the slider to be assigned to a variable are then implemented. This is done with both frequency, and amplitude. As shown below for the amplitude slider:

```
- (IBAction)amplitudeSlider:(UISlider *)sender
{
        UISlider *ampSlider = (UISlider *)sender;
        myAmplitude = ampSlider.value;
}
```

This method is called by iOS every time the slider is moved (because it is denoted as an *IBAction*, i.e. an Interface Builder Action call). The code shows that the *ampSlider* variable is of type *UISlider*, and because of that the current (new) value of the slider is held in ampSlider.value. This is allocated to the variable *myAmplitude*. Similar code exists for the frequency slider.

The protocol methods are then implemented. The previous section showed how we set up our class (*ViewController*) to conform to two Protocols that the Csound API provides: *CsoundObjCompletionListener* and *CsoundValueCacheable*.

Take a look at the place where these Protocols are defined, because a Protocol definition lists clearly what methods are required to be implemented to use their functionality.

For *CsoundValueCacheable* you need to look in the file *CsoundValueCacheable.h* (in the folder *valueCacheable*). In that file it's possible to see the protocol definition, as shown below, and its four required methods.

```
#import <Foundation/Foundation.h>

@class CsoundObj;

@protocol CsoundValueCacheable <NSObject>

-(void)setup:(CsoundObj*)csoundObj;
-(void)updateValuesToCsound;
-(void)updateValuesFromCsound;
-(void)cleanup;

@end
```

Every method needs at least an empty function shell. Some methods, such as *updateValuesFromCsound* are left empty, because – for the tutorial example – there is no need to get values from Csound. Other protocol methods have functionality added. These are discussed below.

The *setup* method is used to prepare the *updateValuesToCsound* method for communication with Csound:

```
-(void)setup:(CsoundObj* )csoundObj

{
    NSString *freqString = @"freqVal";
    freqChannelPtr = [csoundObj getInputChannelPtr:freqString];

    NSString *ampString = @"ampVal";
    ampChannelPtr = [csoundObj getInputChannelPtr:ampString];
}
```

The first line of the method body creates a string; *freqString*, to name the communication channel that Csound will be sending values to. The next line uses the *getInputChannelPtr* method to create the channel pointer for Csound to transfer information to. Effectively, iOS has sent a message to Csound, asking it to open a communication channel with the name "*freqVal*". The Csound object allocates memory that iOS can write to, and returns a pointer to that memory address. From this point onwards iOS could send data values to this address, and Csound can retrieve that data by quoting the channel name "*freqVal*". This is described in more detail in the next section (2.2.4).

The next two lines of the code do the same thing, for amplitude parameter. This process creates two named channels for Csound to communicate through.

The protocol method *updateValuesToCsound* uses variables in the .m file and assigns them to the newly allocated memory address used for communication. This ensures that when Csound looks at this specific memory location, it will find the most up to date value of the variable. This is shown below:

```
-(void)updateValuesToCsound
{
    *freqChannelPtr = myFrequency;
    *ampChannelPtr = myAmplitude;
}
```

The first line assigns the variable *myFrequency* (the value coming from the iOS slider for Frequency) to the channel *freqChannelPtr* which, as discussed earlier, is of type float*. The second line does a similar thing, but for amplitude.

For the other Protocol CsoundObjCompletionListener it is possible to look for the file CsoundObj.h (which is found in Xcode's left-hand file list, in the folder called classes). In there is definition of the protocol.

```
@protocol CsoundObjCompletionListener

-(void)csoundObjDidStart:(CsoundObj*)csoundObj;
-(void)csoundObjComplete:(CsoundObj*)csoundObj;
```

In this example there is nothing special that needs to be done when Csound starts running, or when it completes, so the two methods (csoundObjDidStart: and csoundObjComplete:) are left as empty function shells. In the example, the protocol is left included, along with the empty methods, in case you wish to use them in your App.

2.2.4 The Csound File

This Csound file contains all the code to allow iOS to control its values and output a sinusoid at some frequency and amplitude taken from the on-screen sliders. There are three main sections: The Options, the Instruments, and the Score. These are all discussed in more detail in section 4. Each of these constituent parts of the .csd file will now be broken down to determine how iOS and Csound work together.

2.2.4.1 The Options

There's only one feature in the options section of the .csd that needs to be considered here; the flags. Each flag and its properties are summarised in Table 2.3.

Flag	Description
-o dac	Enables audio output to default device
-+rtmidi=null	Disables real-time MIDI Control
-d	Suppress all displays

Table 2.3-Csound Flags

2.2.4.2 The Instrument

The first lines of code in the instrument set up some important values for the .csd to use when processing audio. These are described in Table 2.4, and are discussed in more detail in the Reference section of the Csound Manual

Line	Description
sr = 44100	This sets the sample rate of Csound to 44100 Hz. It is imperative that the sample rate of the Csound file corresponds with the sample rate of the sound card the code is running on.
ksmps = 64	This defines the control rate. In the example this will determine the speed that the variables in Csound are read. **ksmps** is actually the number of audio samples that are processed before another control update occurs. The actual control rate equates to sample rate / ksmps (i.e. 44100 / 64 = 689.0625 Hz).
nchnls = 2	This is the number of audio channels. 2 = standard stereo.
0dbfs = 1	This is used to ensure that audio samples are within the apropriate range, between zero and one. Anything greater than one will induce clipping to the waveform.

Table 2.4-Csound .csd Options

The instrument then takes values from Csound using the *chnget* opcode:

```
kfreq chnget "freqVal"
kamp chnget "ampVal"
```

Here, the *chnget* command uses the *"freqVal"* and "*ampVal"* channels previously created in iOS to assign a new control variable. The variables *kfreq* and *kamp* are control-rate variables because they begin with the letter 'k'. They will be updated 689.0625 times per second. This may be faster or slower than iOS updates the agreed memory addresses, but it doesn't matter. Csound will just take the value that is there when it accesses the address via the named channel.

These control-rate variables are used to control the amplitude and frequency fields of the opcode *oscil*; the Csound opcode for generating sinusoidal waves. This is then output in stereo using the next line.

```
asig poscil kamp,kfreq,1
outs asig,asig
endin
```

The third parameter of the *oscil* opcode in this case is 1. This means 'use f-table 1'. Section 3.3 explains f-tables in more depth.

2.2.4.3 The Score

The score is used to store the f-tables the instrument is using to generate sounds, and it allows for the playing of an instrument. This instrument is then played, as shown below:

```
i1 0 10000
```

This line plays instrument 1 from 0 seconds, to 10000 seconds. This means that the instrument continues to play until it is stopped, or a great amount of time passes.

It is possible to send score events from iOS using the method *sendScore*. This is discussed in more depth in section. 6.1

3 Using the Mobile Csound API in an Xcode Project

Section 3 provides an overview of how to set up your Xcode project to utilize the Mobile Csound API, as well as how to download the API and include it into your project.

3.1 Setting up an Xcode Project with the Mobile Csound API

This section describes the steps required to set up an Xcode project for use with the Mobile Csound API. Explanations include where to find the Mobile Csound API, how to include it into an Xcode project and what settings are needed.

3.1.2 Creating an Xcode Project

This section briefly describes the settings which are needed to set up an Xcode project for use with the Mobile Csound API. Choose the appropriate template to suit the needs of the project being created. When choosing the options for the project, it is important that *Use Automatic Reference Counting* is not checked (Figure. 3.1). It is also unnecessary to include unit tests.

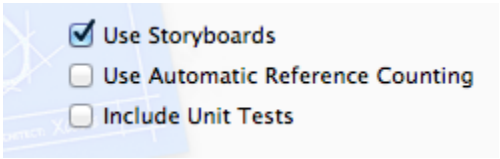

Figure 3.1-Project Set Up

Note: When including this API into a pre-existing project, it is possible to turn off ARC on specific files by entering the compiler sources, and changing the compiler flag to: '-fno-objc-arc'

3.1.3 Adding the Mobile Csound API to an Xcode Project

Once an Xcode project has been created, the API needs to be added to the Xcode project. To add the Mobile Csound API to the project, right click on the Xcode project and select *Add files to <myProject>*. This will bring up a navigation window to search for the files to be added to the project. Navigate to the *Csound-iOS* folder, which is located as shown in Figure 3.2 below.

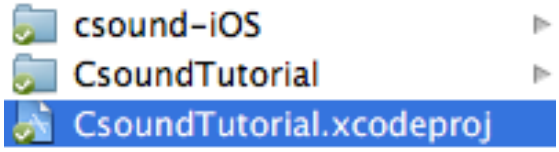

Figure 3.2-Navigating to the API Folder

Select the whole folder as shown and click *add*. Once this has been done, Xcode will provide an options box as shown in Figure 3.3. Check *Copy items into destination group's folder (if needed)*.

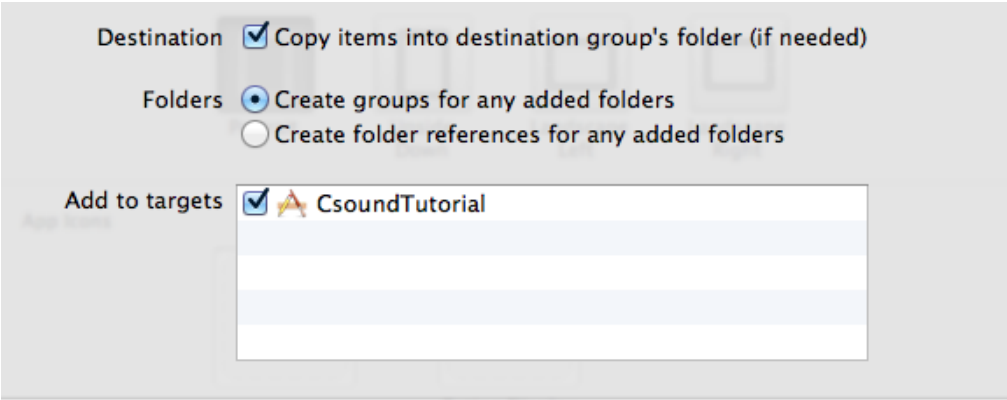

Figure 3.3-Adding the API Folder

The options in Figure 3.3 are selected so that the files which are necessary to run the project are copied into the project folder. This is done to make sure that there are no problems when the project folder is moved to another location - ensuring all the file-paths for the project files remain the same.

Once this addition from this section has been made, the project structure displayed in Xcode should look similar to that in Figure 3.4.

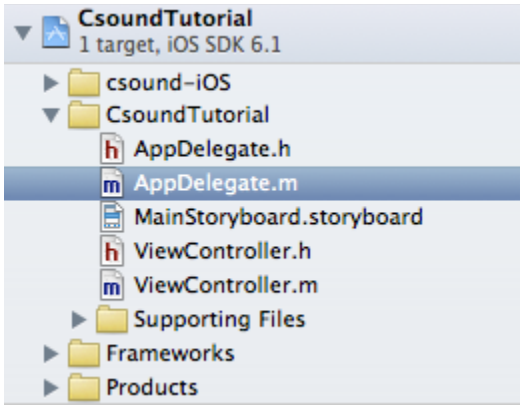

Figure 3.4 - The Main Bundle for the Project

3.1.4 Compiling Sources

A list of compile sources is found by clicking on the blue project file in Xcode, navigating to the *Build Phases tab* and opening *Compile Sources*. Check that the required sources for the project are present in the *Compile Sources* in Xcode. All the files displayed in Figure 3.5 should be present, but not necessarily in the same order as shown.

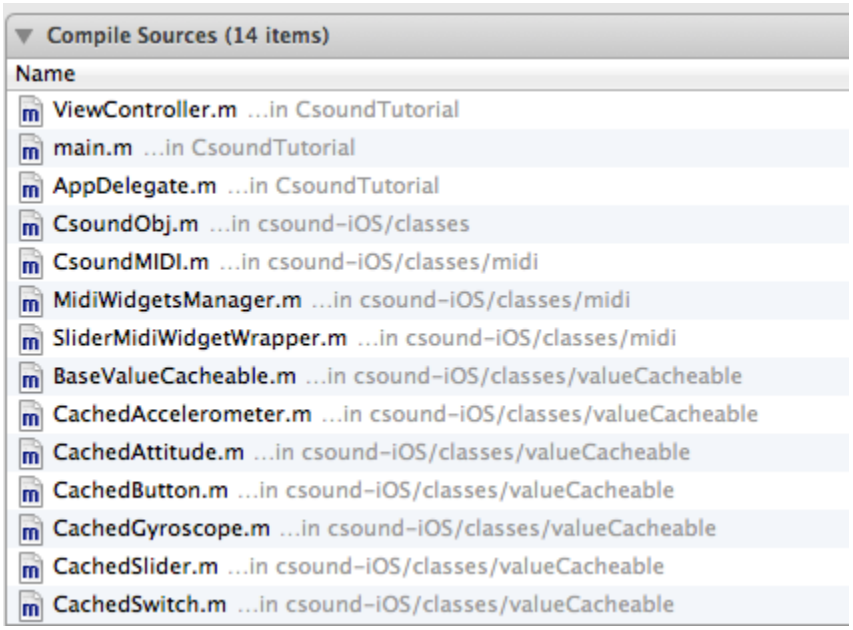

Figure 3.5-View of 'Compile Sources' Window

3.1.5 Including the Necessary Frameworks

There are some additional frameworks which are required to allow the project to run. These frameworks are:

- AudioToolbox.framework
- CoreGraphics.framework
- CoreMotion.framework
- CoreMIDI.framework

To add these frameworks to the project, navigate to the 'Link Binary With Libraries' section of Xcode. This is found by clicking on the blue project folder and navigating to the 'Build Phases' tab, followed by opening 'Link Binary With Libraries'. To add a framework, click on the plus sign and search for the framework required. Once all the necessary frameworks are added, the 'Link Binary With Libraries' should look similar to Figure 3.6 below.

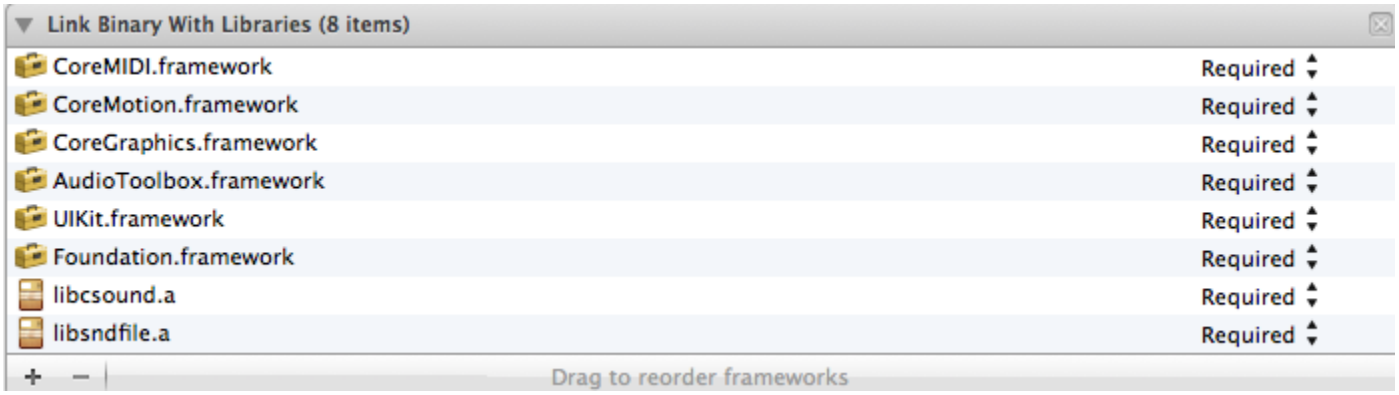

Figure 3.6-Adding Necessary Frameworks

3.1.6 The .csd File

The project is now set up for use with the Mobile Csound API. The final file which will be required by the project is a .csd file which will describe the Csound instruments to be used by the application. A description of what the .csd file is and how to include one into the project is found in *Section 3.3*. This file will additionally need to be referenced appropriately in the Xcode project. A description of where and how this reference is made is available in *Section 2.2.3.2*.

3.2 Setting up the View Controller

This section describes how the *ViewController.h* and the *ViewController.m* should be set up to ensure that they are able to use the API. It will discuss what imports are needed; conforming to the protocols defined by the API; giving a brief overview. This section can be viewed in conjunction with the example project provided.

3.2.1 Importing

So that the code is able to access other code in the API, it is important to include the following imports, along with imports for any additional files required. The three imports shown in Table 3.1 are used in the header file of the view controller to access the necessary files to get Csound-iOS working:

Import	Description
#import "CsoundObj.h"	This is used so that the code is able to access all the key methods of the API.
#import "CsoundValueCacheable.h"	This must be used to access the methods 'updateValuesFromCsound' and 'updateValuesToCsound'. These methods are used to communicate between Csound and iOS.

Table 3.1-Header File Imports

In our example you can see these at the top of *ViewController.h*

3.2.2 Conforming to Protocols

It is imperative that the view controller conforms to the protocols outlined the CsoundObj.h file; the file in the API that allows for communication between iOS and Csound. This must then be declared in the ViewController.h file:

```
@interface ViewController  : UIViewController
<CsoundObjCompletionListener, CsoundValueCacheable>
```

The API authors chose to use protocols so that there is a defined set of methods that must be used in the code. This ensures that a consistent design is adhered to. They are defined in the *CsoundValueCacheable.h* file thus:

```
@class CsoundObj;

@protocol CsoundValueCacheable <NSObject>

-(void)setup:(CsoundObj*)csoundObj;
-(void)updateValuesToCsound;
-(void)updateValuesFromCsound;
-(void)cleanup;
```

Each of these must then be implemented in the *ViewController.m* file. If it is unnecessary to implement one of these methods, it still *must* appear but the method body can be left blank, thus:

```
-(void)updateValuesFromCsound
{

    //No values coming from Csound to iOS

}
```

3.2.3 Overview of Protocols

When writing the code which allows us to send values from iOS to Csound, it is important that the code conforms to the following protocol methods (Table 3.2):

Protocol methods	Action
-(void)setup:(CsoundObj*)CsoundObj	Set up the necessary channels and pointers to communicate with Csound.
-(void)updateValuesToCsound	Update the values being sent from iOS to Csound.
-(void)updateValuesFromCsound	Collect any values from Csound.
-(void)cleanup	Reset any values used in communication and de-allocate any memory used.
-(void)csoundObjDidStart:(CsoundObj*)csoundObj	This method is called when a Csound object is created. This allows developers to notify the user that Csound is running on iOS.
-(void)csoundObjComplete:(CsoundObj*)csoundObj	Much like the way the 'csoundObjDidStart'method works, this allows developers to notify the user that Csound has stopped running in iOS.

Table 3.2-Protocol methods which must be implemented in your *ViewController*.

3.3 Looking at the Csound '.csd' File

The following section provides an overview of the Csound editing environment, the structure of the .csd file, and how to include the .csd file into your Xcode project.

3.3.1 Downloading Csound

A Csound front-end editor, CsoundQt, can be used for editing the .csd file in the provided example project. It is advised to use CsoundQt with iOS because it is an ideal environment for developing and testing the Csound audio engine – error reports for debugging, the ability to run the Csound audio code on its own, and listen to its results. However, using CsoundQt is not essential to use Csound as an audio engine as Csound is a standalone language. CsoundQt is included in the Csound package download.

In order to use Csound in iOS, the latest version of Csound *(Version 5.19)* will need to be installed.

Csound 5.19 can be downloaded from the following link:

http://sourceforge.net/projects/Csound/files/Csound5/Csound5.19/

In order for Xcode to see the .csd file, it must be imported it into the Xcode project. This is done by right-clicking on the 'Supporting Files' folder in the project, and clicking on 'Add files to *(project name)*' (Figure 3.7).

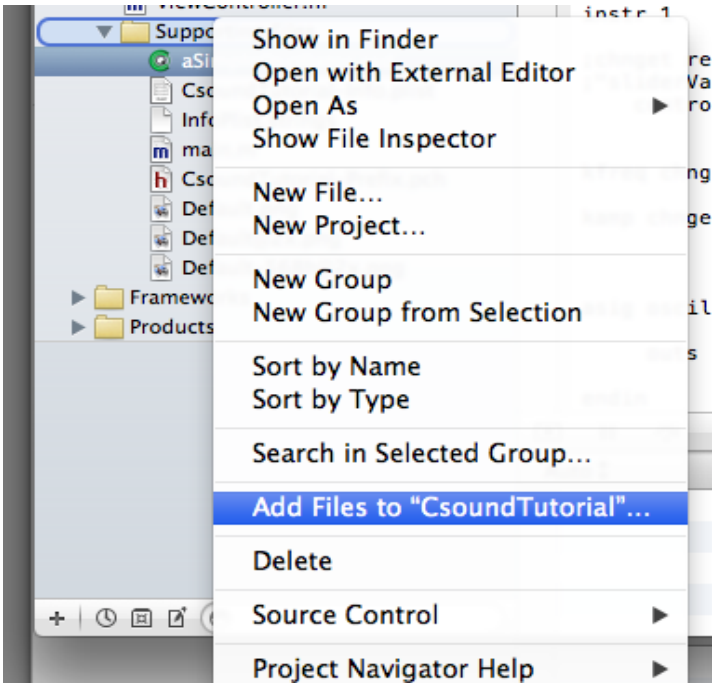

Figure 3.7-Adding the .csd to iOS Project

It is possible to edit the .csd file while also working in Xcode. This is done by right-clicking on the .csd file in Xcode, and clicking on 'Open With External Editor' (Figure 3.8).

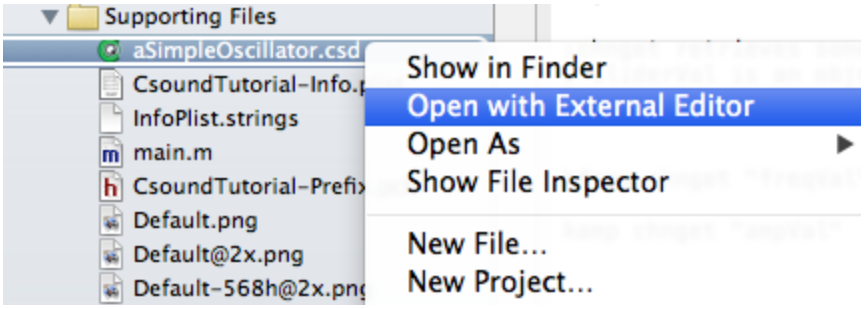

Figure 3.8-Opening the .csd file with an external editor

However, it is important to remember to save any changes to the .csd file before the Xcode project is recompiled.

3.3.2 The .csd File

When setting up a Csound project, it is important that various audio and performance settings configured correctly in the header section of the .csd file. These settings are described in Table 3.3, and are discussed in more detail in the Csound Manual.

Setting	Description
`sr`	Sample rate
`kr`	Control rate
`ksmps`	Number of samples in control period (sr/kr)
`nchnls`	Number of channels of audio output
`0dbfs`	Sets value of 0 decibels using full scale amplitude

Table 3.3-Csound .csd Settings

It is important that the sample rate for the Csound project be set to the same sample rate as the hardware it will be run on. For this project, make sure the sample rate set to 44100, as depicted in Figure 3.9. This is done by opening the Audio MIDI Setup, which is easily found on all *Mac* computers by searching in *Spotlight*.

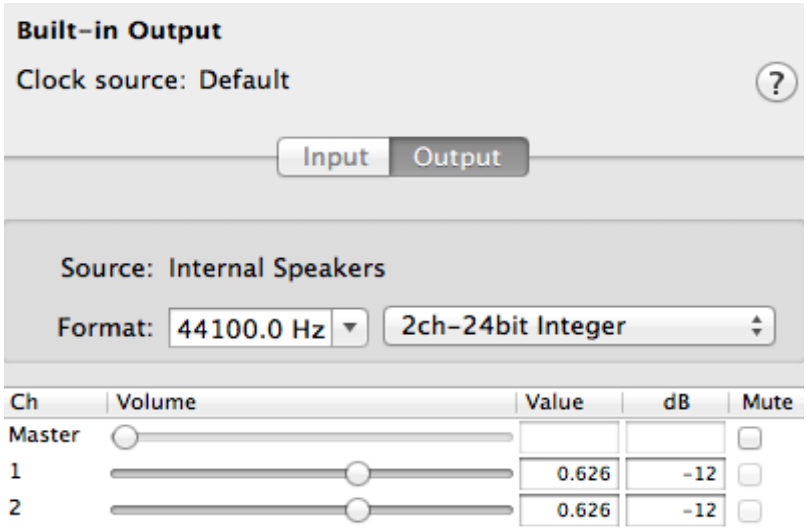

Figure 3.9-Configuring Audio Hardware Settings

3.3.3 Instruments

As mentioned previously, Csound instruments are defined in the orchestra section of the .csd file. The example project provided by the authors uses a simple oscillator that has two parameters: amplitude and frequency, both of which are controlled by UI sliders.

Figure 3.10 on the following page shows a block diagram of the synthesizer we are using in the example project.

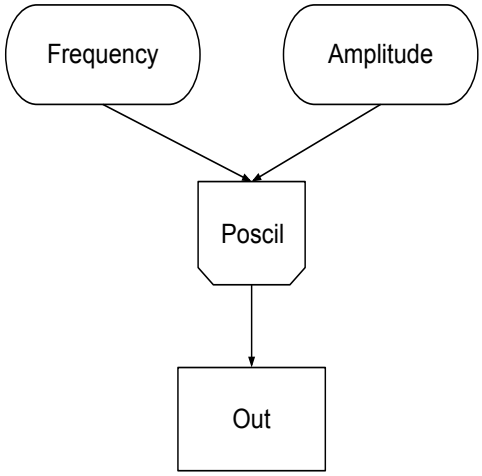

Figure 3.10-Block Diagram for .csd Instrument

3.3.4 Score

The score is the section of the .csd file which provides instruments with control instruction, for example pitch, volume, and duration. However, as the goal here is for users to be able to interact with the Csound audio engine in real-time, developers will most likely opt instead to send score information to Csound that is generated by UI elements in the Xcode project. Details of the instrument and score can be found in the comments of the *aSimpleOscillator.csd* file

Csound uses GEN (f-table generator) routines for a variety of functions. This project uses GEN10, which create composite waveforms by adding partials. At the start of the score section, a GEN routine is specified by function statements (also known as *f-statements*). The parameters are shown below in Table 3.4:

Parameter	Description
`f1`	Unique f-table identification number
`0`	f-statement initialization time expressed in score beats
`16384`	f-table size
`10`	GEN routine called to create the f-table
`1`	strength of ascending partials

Table 3.4-Csound .csd F-Table Parameters

In a Csound score, the first three parameter fields (also known as p-fields) are reserved for the instrument number, the start time, and duration amount. P-fields 4 and 5 are conventionally reserved for amplitude and frequency, however, P-fields beyond 3 can be programmed as desired.

The p-fields used in the example project are shown in Table 3.5.

p-field	1	2	3	4	5
Parameter	Instrument Number	Start	Duration	Amplitude	Frequency

Table 3.5-Csound .csd P-field Parameters

In this project, the first three p-fields are used: the instrument number (i1), the start time (time = 0 seconds), and the duration (time = 1000 seconds). Amplitude and frequency are controlled by UI sliders in iOS.

4 Common Problems

This section is designed to document some common problems faced during the creation of this tutorial. It is hoped that by outlining these common errors, readers can debug some common errors they are likely to come across when creating applications using this API. It discusses each error, describes the cause and outlines a possible solution.

4.1 UIKnob.h is Not Found

This is a problem related to the API. The older versions of the API import a file in the examples that sketches a UIKnob in Core Graphics. This is not a part of the API, and should not be included in the project.

The file in question is a part of the examples library provided with the SDK. It is used in the file 'AudioIn test' and is used to sketch a radial knob on the screen. It gives a good insight into how the user can generate an interface to interact with the API.

Solution: Comment the line out, or download the latest version of the API.

4.2 Feedback from Microphone

This is generally caused by the sample rate of a .csd file being wrong.

Solution: Ensure that the system's sample rate is the same as in the .csd file. Going to the audio and MIDI set-up and checking the current output can find the computer's sample rate. See section 3.3.2 for more information.

4.3 Crackling Audio

There are numerous possible issues here, but the main cause of this happening is a CPU overload.

Solution: The best way to debug this problem is to look through the code and ensure that there are no memory intensive processes, especially in code that is getting used a lot. Problem areas include fast iterations (loops), and code where Csound is calling a variable. Functions such as *updateValuesToCsound* and *updateValuesFromCsound* are examples of frequently called functions.

An example: an NSLog in the *updateValuesToCsound* method can cause a problem. Say, the *ksmps* in the .csd is set to 64. This means that the Csound is calling for iOS to run the method *updateValuesToCsound* every 64 samples. Assuming the sample rate is 44.1k this means that this CPU intensive NSLog is being called ~689 times a second; very computationally expensive.

4.4 Crackling from amplitude slider

When manipulating the amplitude slider in iOS, a small amount of clicking is noticeable. This is due to the fact that there is no envelope-smoothing function applied to the amplitude changes. While this would be an improvement on the current implementation, however; it was felt that the current implementation would be more conducive to learning for the novice Csound user. This would be implemented by using a *port* opcode.

5 Csound Library Methods

This section will present and briefly describe the methods which are available in the M

Name	Method Call	Description
startCsound	`-(void) startCsound: (NSString*)csdFilePath;`	Provides the location of the .csd file which is to be used with the Csound object.
	`-(void)startCsound: (NSString *)csdFilePath recordToURL:(NSURL *)outputURL;`	Provides the location of the .csd file which is to be used with the Csound object and specifies a URL to which it will record.
startCsoundToDisk	`-(void)startCsoundToDisk: (NSString*)csdFilePath outputFile: (NSString*)outputFile;`	Provides the location of the .csd file which is to be used with the Csound object and specifies a file to which it will record. This does not occur in realtime, but as fast as possible to the disk. This method is useful for batch rendering.
stopCsound	`-(void)stopCsound;`	This uses the Csound object's method 'stopCsound' to stop the instance of CsoundObj that it is called on.
muteCsound	`-(void)muteCsound;`	Mutes all instances of Csound
unmuteCsound	`-(void)unmuteCsound;`	Unmutes all instances of Csound
recordToURL	`-(void)recordToURL: (NSURL *)outputURL;`	Begins recording to a specified URL. This can be defined at a later point in the code, even after Csound has been started.
stopRecording	`-(void)stopRecording;`	Stops recording to URL

Table 5.1-Basic API Methods

5.2 UI and Hardware Methods

Name	Method Call	Description
addSwitch	`(id<CsoundValueCacheable>) addSwitch: (UISwitch*)uiSwitch forChannelName: (NSString*)channelName;`	Adds a switch to the Csound object. The method requires a switch which already exists as part of the user interface and a name for the channel which will provide information about this switch to the .csd file. For more information about channels of information between Xcode and Csound see section 5.
addSlider	`(id<CsoundValueCacheable>) addSlider: (UISlider*)uiSlider forChannelName:(NSString*) channelName;`	Adds a slider to the Csound Object. The method requires a slider and a channel name.
addButton	`(id<CsoundValueCacheable>) addButton: (UIButton*)uiButton forChannelName:(NSString*) channelName;`	Adds a button to the Csound Object. The method requires a button and a channel name.
enableAccelerometer	`(id<CsoundValueCacheable>) enableAccelerometer;`	Enables the accelerometer for use with the Csound object.
enableGyroscope	`(id<CsoundValueCacheable>) enableGyroscope;`	Enables the gyroscope for use with the Csound object.
enableAttitude	`(id<CsoundValueCacheable>) enableAttitude;`	Enables attitude to allow device motion to be usable with the Csound object.

Table 5.2-UI and Hardware Methods

5.3 Communicating between Xcode and Csound

Name	Method Call	Description
addValueCacheable	`-(void)addValueCacheable:(id<CsoundValueCacheable>)valueCacheable;`	Adds to a list of watched objects so that they can update every cycle of ksmps.
removeValueCacheable	`-(void)removeValueCaheable:(id<CsoundValueCacheable>)valueCacheable;`	Removes a cacheable value from the Csound Object.

sendScore	`-(void)sendScore:(NSString*)score;`
	
	Eg:
	`[self.csound sendScore:[NSString stringWithFormat:@"i1 0 10 0.5 %d", myPitch,]];`
	
	(sends a score to instrument 1 that begins at 0 seconds, stops at 10 seconds, with amplitude 0.5 and a pitch of the objective-C variable 'myPitch').
		Sends a score as a string to the .csd file. See section 4 for formatting a Csound score line.

addCompletionListener	`-(void)addCompletionListener:(id<CsoundObjCompletionListener>)listener;`	Adds a listener for the Csound Object which waits for an action to be performed that the Csound object needs to react to.

Table 5.3-API Communication Methods

5.4 Retrieve Csound-iOS Information

Name	Method Call	Description
getCsound	`-(CSOUND*)getCsound;`	Returns the C structure that that the CsoundObj uses. This allows developers to use the Csound C API in conjunction with the Objective-C CsoundObj API.
getInputChannelPtr	`(float*)getInputChannelPtr: (NSString*)channelName;`	Returns the float of an input channel pointer.
getOutputChannelPtr	`(float*)getOutputChannelPtr: (NSString*)channelName;`	Returns the float of an output channel pointer.
getOutSamples	`-(NSData*)getOutSamples;`	Gets audio samples from Csound.
getNumChannels	`-(int)getNumChannels;`	Returns the number of channels in operation.
getKsmps	`-(int)getKsmps;`	Returns ksmps as defined in the .csd file.
setMessageCallback	`-(void)setMessageCallback: (SEL)method withListener:(id)listener;`	Sets up a method to be the callback method and a listener id.
performMessageCallback	`(void) performMessageCallback: (NSValue *)infoObj;`	Performs the message callback.

Table 5.4-Retrieve Csound-iOS Information Methods

6 Conclusions

This tutorial provided an overview of the Csound-iOS API, outlining its benefits, and describing its functionality by means of an example project. It provided the basic tools for using the API, equipping iOS developers to explore the potential of this API in their own time.

APIs such as this one, as well as others including *libpd* and *The Amazing Audio Engine* provide developers with the ability to integrate interactive audio into their apps, without having to deal with the low-level complexities of Core Audio.

6.1 Additional Resources

Upon completion of this tutorial, the authors suggest that the reader look at the original Csound for iOS example project, written by Steven Yi and Victor Lazzarini.

This is available for download from:

http://sourceforge.net/projects/csound/files/csound5/iOS/

About the Authors

The authors are Masters students at the University of York Audio Lab. Each one is working on a separate interactive audio app for the iPad, and has each been incorporating the Mobile Csound API for that purpose. They came together to write this tutorial to make other developers aware of the Mobile Csound API, and how to utilize it.

The motivation behind this tutorial was to create a step by step guide to using the Mobile Csound API. When the authors originally started to develop with the API, they found it difficult to emulate the results of the examples that were provided with the API download. As a result, the authors created a simple example using the API, and wanted others to learn from our methods and mistakes. The authors hope that this tutorial provides clarity in the use of the Mobile Csound API.