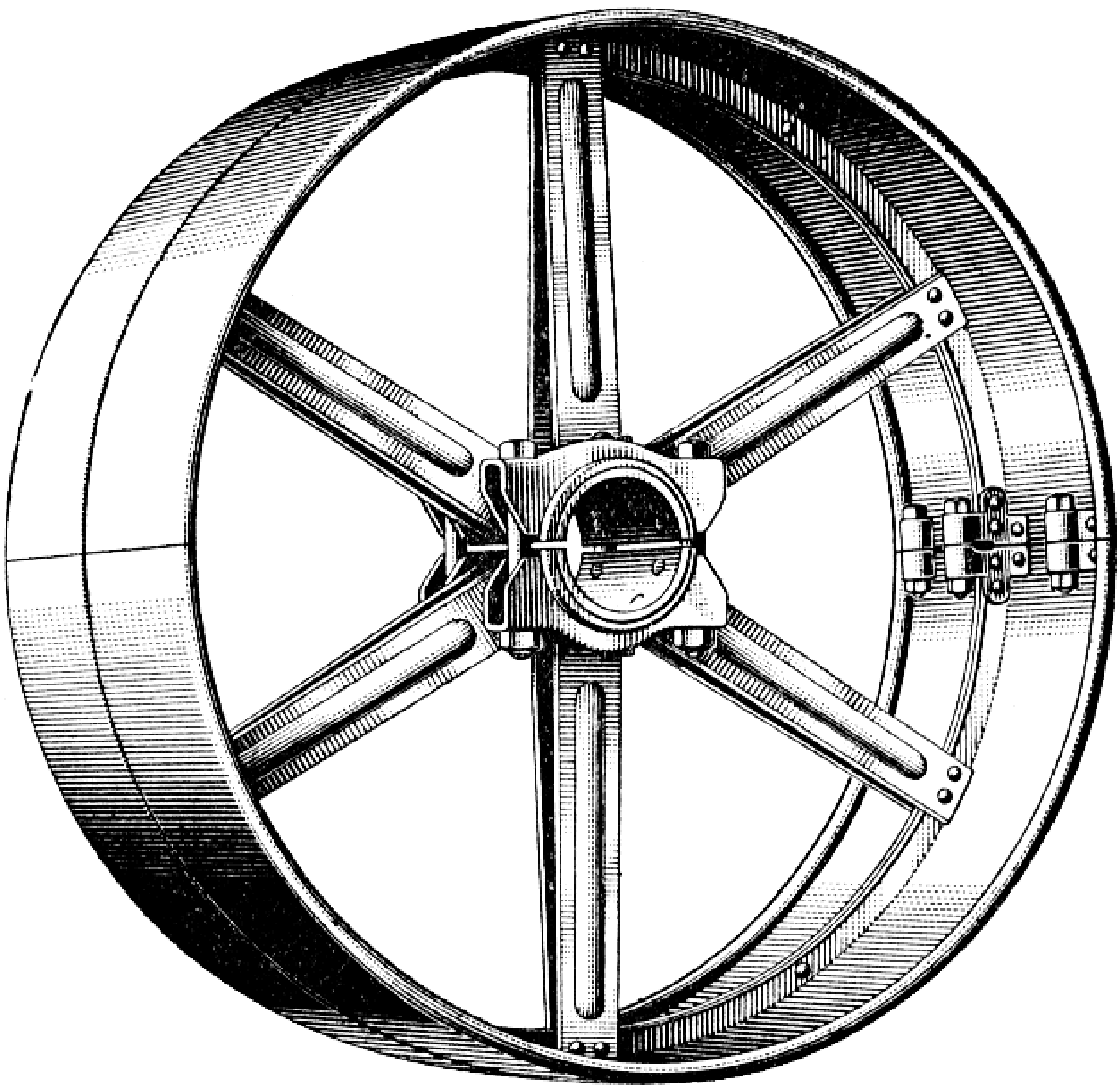

FIG. 16

OpenEars 1.5.2 Manual

Written by Halle Winkler, published by Politepix

Tuesday, September 10, 2013

Table of Contents

Introduction and Installation

Introduction

OpenEars is an shared-source iOS framework for iPhone voice recognition and speech synthesis (TTS). It lets you easily implement round-trip English and Spanish language speech recognition and English text-to-speech on the iPhone, iPod and iPad and uses the open source CMU Pocketsphinx, CMU Flite, and CMUCLMTK libraries, and it is free to use in an iPhone, iPad or iPod app (Spanish text-to-speech is possible on the OpenEars Platform but requires using NeatSpeech since there isn't a Spanish voice for Flite). It is the most popular offline framework for speech recognition and speech synthesis on iOS and has been featured in development books such as O'Reilly's *Basic Sensors in iOS* by Alasdair Allan and *Cocos2d for iPhone 1 Game Development Cookbook* by Nathan Burba.

The OpenEars Platform is also a complete development platform for creating your speech recognition and text-to-speech apps including both the free OpenEars SDK documented on this page and a diverse set of plugins that can be added to OpenEars in order to extend and refine its default features: you can read more about the OpenEars platform here. This page is all about the free and shared-source OpenEars SDK, to please read on to learn more about it.

Highly-accurate large-vocabulary recognition (that is, trying to recognize any word the user speaks out of many thousands of known words) is not yet a reality for local in-app processing on a small handheld device given the hardware limitations of the platform; even Siri does its large-vocabulary recognition on the server side. However, Pocketsphinx (the open source voice recognition engine that OpenEars uses) is capable of local recognition of vocabularies with hundreds of words depending on the environment and other factors, and performs very well with command-and-control language models in English and Spanish. The best part is that it uses no network connectivity because all processing occurs locally on the device.

The current version of OpenEars is 1.5.2.Download OpenEars or read its changelog.

Features of OpenEars

OpenEars can:

- Performs speech recognition and language model generation in English and in Spanish
- Performs text-to-speech in English and with the NeatSpeech plugin, can also perform text-to-speech in Spanish
- Listen continuously for speech on a background thread, while suspending or resuming speech processing on demand, all while using less than 4% CPU on average on an iPhone 4 (decoding speech, text-to-speech, updating the UI and other intermittent functions use more CPU),
- Use any of 9 voices for speech, including male and female voices with a range of speed/quality level, and switch between them on the fly,
- Change the pitch, speed and variance of any text-to-speech voice,
- Know whether headphones are plugged in and continue voice recognition during text-to-speech only when they are plugged in,
- Support bluetooth audio devices (experimental),
- Dispatch information to any part of your app about the results of speech recognition and speech, or changes in the state of the audio session (such as an incoming phone call or headphones being plugged in),
- Deliver level metering for both speech input and speech output so you can design visual feedback for both states.
- Support JSGF grammars,
- Dynamically generate new ARPA language models in-app based on input from an NSArray of NSStrings,
- Switch between ARPA language models or JSGF grammars on the fly,
- Get n-best lists with scoring,
- Test existing recordings,
- Be easily interacted with via standard and simple Objective-C methods,
- Control all audio functions with text-to-speech and speech recognition in memory instead of writing audio files to disk and then reading them,
- Drive speech recognition with a low-latency Audio Unit driver for highest responsiveness,
- Be installed in a Cocoa-standard fashion using an easy-peasy already-compiled framework.
- In addition to its various new features and faster recognition/text-to-speech responsiveness, OpenEars now has improved recognition accuracy.
- OpenEars is free to use in an iPhone or iPad app.

Warning

Before using OpenEars, please note it has to use a different audio driver on the Simulator that is less accurate, so it is always necessary to evaluate accuracy on a real device. Please don't submit support requests for accuracy issues with the Simulator.

Installation

To use OpenEars:

- Download the distribution and unpack it.
- Create your own app, and add the iOS frameworks AudioToolbox and AVFoundation to it.
- Inside your downloaded distribution there is a folder called "Frameworks". Drag the "Frameworks" folder into your app project in Xcode.

OK, now that you've finished laying the groundwork, you have to...wait, that's everything. You're ready to start using OpenEars. Give the sample app a spin to try out the features (the sample app uses ARC so you'll need a recent Xcode version) and then visit the Politepix interactive tutorial generator for a customized tutorial showing you exactly what code to add to your app for all of the different functionality of OpenEars.

If the steps on this page didn't work for you, you can get free support at the forums, read the FAQ, brush up on the documentation, or open a private email support incident at the Politepix shop. If you'd like to read the documentation, simply read onward.

Basic concepts

There are a few basic concepts to understand about voice recognition and OpenEars that will make it easiest to create an app.

- Local or offline speech recognition versus server-based or online speech recognition:

most speech recognition on the iPhone, iPod and iPad is done by streaming the speech audio to servers. OpenEars works by doing the recognition inside the device, entirely offline without using the network. This saves bandwidth and results in faster response, but since a server is much more powerful than a phone it means that we have to work with much smaller vocabularies to get accurate recognition.

- Language Models. The language model is the vocabulary that you want OpenEars to understand, in a format that its speech recognition engine can understand. The smaller and better-adapted to your users' real usage cases the language model is, the better the accuracy. An ideal language model for PocketsphinxController has fewer than 200 words.
- The parts of OpenEars. OpenEars has a simple, flexible and very powerful architecture.

PocketsphinxController recognizes speech using a language model that was dynamically created by LanguageModelGenerator. FliteController creates synthesized speech (TTS). And OpenEarsEventsObserver dispatches messages about every feature of OpenEars (what speech was understood by the engine, whether synthesized speech is in progress, if there was an audio interruption) to any part of your app.

AcousticModel Class Reference

Detailed Description

Convenience class for accessing the acoustic model bundles. All this does is allow you to reference your chosen model by including this header in your class and then letting you call [AcousticModel pathToModel:"AcousticModelEnglish"] or [AcousticModel pathToModel:@"AcousticModelSpanish"] in any of the methods which ask for a path to an acoustic model.

Method Documentation

```
+ (NSString *) pathToModel: (NSString *) acousticModelBundleName
```

Reference the path to any acoustic model bundle you've dragged into your project (such as AcousticModelSpanish.bundle or AcousticModelEnglish.bundle) by calling this class method like [AcousticModel pathToModel:"AcousticModelEnglish"] after importing this class.

FliteController Class Reference

Detailed Description

The class that controls speech synthesis (TTS) in OpenEars.

Usage examples

> *Preparing to use the class:*

To use FliteController, you need to have at least one Flite voice added to your project. When you added the "framework" folder of OpenEars to your app, you already imported a voice called Slt, so these instructions will use the Slt voice. You can get eight more free voices in OpenEarsExtras, available at https://bitbucket.org/Politepix/openearsextras

> *What to add to your header:*

Add the following lines to your header (the .h file). Under the imports at the very top:

```
#import <Slt/Slt.h>
#import <OpenEars/FliteController.h>
```

In the middle part where instance variables go:

```
FliteController *fliteController;
Slt *slt;
```

In the bottom part where class properties go:

```
@property (strong, nonatomic) FliteController *fliteController;
@property (strong, nonatomic) Slt *slt;
```

> *What to add to your implementation:*

Add the following to your implementation (the .m file): Under the @implementation keyword at the top:

```
@synthesize fliteController;
@synthesize slt;
```

Among the other methods of the class, add these lazy accessor methods for confident memory management of the object:

```
- (FliteController *)fliteController {
 if (fliteController == nil) {
  fliteController = [[FliteController alloc] init];
 }
 return fliteController;
}

- (Slt *)slt {
 if (slt == nil) {
  slt = [[Slt alloc] init];
 }
 return slt;
}
```

> *How to use the class methods:*

In the method where you want to call speech (to test this out, add it to your viewDidLoad method), add the following method call:

```
[self.fliteController say:@"A short statement" withVoice:self.slt];
```

Warning

There can only be one FliteController instance in your app at any given moment.

Method Documentation

```
- (void) say: (NSString *)    statement
   withVoice: (FliteVoice *)  voiceToUse
```

This takes an NSString which is the word or phrase you want to say, and the FliteVoice to use to say the phrase. Usage Example:

```
[self.fliteController say:@"Say it, don't spray it."
withVoice:self.slt];
```

There are a total of nine FliteVoices available for use with OpenEars. The Slt voice is the most popular one and it ships with OpenEars. The other eight voices can be downloaded as part of the OpenEarsExtras package available at the URL http://bitbucket.org/Politepix/openearsextras. To use them, just drag the desired downloaded voice's framework into your app, import its header at the top of your calling class (e.g. import <Slt/Slt.h> or import <Rms/Rms.h>) and instantiate it as you would any other object, then passing the instantiated voice to this method.

```
- (Float32) fliteOutputLevel
```

A read-only attribute that tells you the volume level of synthesized speech in progress. This is a UI hook. You can't read it on the main thread.

Property Documentation

```
- (float) duration_stretch
```

duration_stretch changes the speed of the voice. It is on a scale of 0.0-2.0 where 1.0 is the default.

```
- (float) target_mean
```

target_mean changes the pitch of the voice. It is on a scale of 0.0-2.0 where 1.0 is the default.

```
- (float) target_stddev
```

target_stddev changes convolution of the voice. It is on a scale of 0.0-2.0 where 1.0 is the default.

```
- (BOOL) userCanInterruptSpeech
```

Set userCanInterruptSpeech to TRUE in order to let new incoming human speech cut off synthesized speech in progress.

```
- (BOOL) noAudioSessionOverrides
```

Set noAudioSessionOverrides to TRUE in order to run Flite without the OpenEars AudioSessionManager (useful if you aren't using PocketsphinxController and you want to set your own audiosession behavior or have none but the defaults). This can't be used in combination with PocketsphinxController, it is only for apps which used FliteController exclusively. This BOOL is only picked up once per instantiation of FliteController so if you want to switch back and forth it is necessary to release your FliteController and create a new FliteController with the new noAudioSessionOverrides setting.

LanguageModelGenerator Class Reference

Detailed Description

The class that generates the vocabulary the PocketsphinxController is able to understand.

Usage examples

> *What to add to your implementation:*

Add the following to your implementation (the .m file): Under the @implementation keyword at the top:

```
#import <OpenEars/LanguageModelGenerator.h>
```

Wherever you need to instantiate the language model generator, do it as follows:

```
LanguageModelGenerator *lmGenerator = [[LanguageModelGenerator alloc]
init];
```

> *How to use the class methods:*

In offline speech recognition, you define the vocabulary that you want your app to be able to recognize. A good vocabulary size for an offline speech recognition app on the iPhone, iPod or iPad is between 3 and 300 words.

In the method where you want to create your language model (for instance your viewDidLoad method), add the following method call (replacing the placeholders like "WORD" and "A PHRASE" with actual words and phrases you want to be able to recognize):

```
NSArray *words = [NSArray arrayWithObjects:@"WORD", @"STATEMENT",
@"OTHER WORD", @"A PHRASE", nil];
NSString *name = @"NameIWantForMyLanguageModelFiles";
NSError *err = [lmGenerator generateLanguageModelFromArray:words
withFilesNamed:name forAcousticModelAtPath:[AcousticModel
pathToModel:@"AcousticModelEnglish"]]; // Change "AcousticModelEnglish"
to "AcousticModelSpanish" to create a Spanish language model instead of
an English one.


NSDictionary *languageGeneratorResults = nil;

NSString *lmPath = nil;
NSString *dicPath = nil;

if([err code] == noErr) {

 languageGeneratorResults = [err userInfo];

 lmPath = [languageGeneratorResults objectForKey:@"LMPath"];
 dicPath = [languageGeneratorResults objectForKey:@"DictionaryPath"];

} else {
 NSLog(@"Error: %@",[err localizedDescription]);
}
```

If you are using the default English-language or Spanish-language model generation, it is a requirement to enter your words and phrases in all capital letters, since the model is generated against a dictionary in which the entries are capitalized (meaning that if the words in the array aren't capitalized, they will not match the dictionary and you will not have the widest variety of pronunciations understood for the word you are using). If you need to create a fixed language model ahead of time instead of creating it dynamically in your app, just use this method (or generateLanguageModelFromTextFile:withFilesNamed:) to submit your full language model using the Simulator and then use the Simulator documents folder script to get the language model and dictionary file out of the documents folder and add it to your app bundle, referencing it from there.

Method Documentation

```
- (NSError *) generateLanguageModelFromArray: (NSArray *)   languageModelArray
                              withFilesNamed: (NSString *)  fileName
                        forAcousticModelAtPath: (NSString *)  acousticModelPath
```

Generate a language model from an array of NSStrings which are the words and phrases you want PocketsphinxController or PocketsphinxController+RapidEars to understand, using your chosen acoustic model. Putting a phrase in as a string makes it somewhat more probable that the phrase will be recognized as a phrase when spoken. fileName is the way you want the output files to be named, for instance if you enter "MyDynamicLanguageModel" you will receive files output to your Caches directory titled MyDynamicLanguageModel.dic, MyDynamicLanguageModel.arpa, and MyDynamicLanguageModel.DMP. The error that this method returns contains the paths to the files that were created in a successful generation effort in its userInfo when NSError == noErr. The words and phrases in languageModelArray must be written with capital letters exclusively, for instance "word" must appear in the array as "WORD".

```
- (NSError *) generateLanguageModelFromTextFile: (NSString *) pathToTextFile
                                 withFilesNamed: (NSString *) fileName
                          forAcousticModelAtPath: (NSString *) acousticModelPath
```

Generate a language model from a text file containing words and phrases you want PocketsphinxController to understand, using your chosen acoustic model. The file should be formatted with every word or contiguous phrase on its own line with a line break afterwards. Putting a phrase in on its own line makes it somewhat more probable that the phrase will be recognized as a phrase when spoken. Give the correct full path to the text file as a string. fileName is the way you want the output files to be named, for instance if you enter "MyDynamicLanguageModel" you will receive files output to your Caches directory titled MyDynamicLanguageModel.dic, MyDynamicLanguageModel.arpa, and MyDynamicLanguageModel.DMP. The error that this method returns contains the paths to the files that were created in a successful generation effort in its userInfo when NSError == noErr. The words and phrases in languageModelArray must be written with capital letters exclusively, for instance "word" must appear in the array as "WORD".

Property Documentation

```
- (BOOL) verboseLanguageModelGenerator
```

Set this to TRUE to get verbose output

```
- (BOOL) useFallbackMethod
```

Advanced: if you are using your own acoustic model or an custom dictionary contained within an acoustic model and these don't use the same phonemes as the English or Spanish acoustic models, you will need to set useFallbackMethod to FALSE so that no attempt is made to use the English or Spanish fallback method for finding pronunciations of words which don't appear in the custom acoustic model's phonetic dictionary.

OpenEarsEventsObserver Class Reference

Detailed Description

OpenEarsEventsObserver provides a large set of delegate methods that allow you to receive information about the events in OpenEars from anywhere in your app. You can create as many OpenEarsEventsObservers as you need and receive information using them simultaneously. All of the documentation for the use of OpenEarsEventsObserver is found in the section OpenEarsEventsObserverDelegate.

Property Documentation

```
- (id< OpenEarsEventsObserverDelegate >) delegate
```

To use the OpenEarsEventsObserverDelegate methods, assign this delegate to the class hosting OpenEarsEventsObserver and then use the delegate methods documented under OpenEarsEventsObserverDelegate. There is a complete example of how to do this explained under the OpenEarsEventsObserverDelegate documentation.

OpenEarsLogging Class Reference

Detailed Description

A singleton which turns logging on or off for the entire framework. The type of logging is related to overall framework functionality such as the audio session and timing operations. Please turn OpenEarsLogging on for any issue you encounter. It will probably show the problem, but if not you can show the log on the forum and get help.

Warning

The individual classes such as PocketsphinxController and LanguageModelGenerator have their own verbose flags which are separate from OpenEarsLogging.

Method Documentation

```
+ (id) startOpenEarsLogging
```

This just turns on logging. If you don't want logging in your session, don't send the startOpenEarsLogging message.

> *Example Usage:*

Before implementation:

```
#import <OpenEars/OpenEarsLogging.h>;
```

In implementation:

```
[OpenEarsLogging startOpenEarsLogging];
```

PocketsphinxController Class Reference

Detailed Description

The class that controls local speech recognition in OpenEars.

Usage examples

> *Preparing to use the class:*

To use PocketsphinxController, you need a language model and a phonetic dictionary for it. These files define which words PocketsphinxController is capable of recognizing. They are created above by using LanguageModelGenerator. You also need an acoustic model. OpenEars ships with an English and a Spanish acoustic model.

> *What to add to your header:*

Add the following lines to your header (the .h file). Under the imports at the very top:

```
#import <OpenEars/PocketsphinxController.h>
#import <OpenEars/AcousticModel.h>
```

In the middle part where instance variables go:

```
PocketsphinxController *pocketsphinxController;
```

In the bottom part where class properties go:

```
@property (strong, nonatomic) PocketsphinxController
*pocketsphinxController;
```

> *What to add to your implementation:*

Add the following to your implementation (the .m file): Under the @implementation keyword at the top:

```
@synthesize pocketsphinxController;
```

Among the other methods of the class, add this lazy accessor method for confident memory management of the object:

```
- (PocketsphinxController *)pocketsphinxController {
 if (pocketsphinxController == nil) {
  pocketsphinxController = [[PocketsphinxController alloc] init];
 }
 return pocketsphinxController;
}
```

> *How to use the class methods:*

In the method where you want to recognize speech (to test this out, add it to your viewDidLoad method), add the following method call:

```
[self.pocketsphinxController
startListeningWithLanguageModelAtPath:lmPath dictionaryAtPath:dicPath
acousticModelAtPath:[AcousticModel pathToModel:@"AcousticModelEnglish"]
languageModelIsJSGF:NO]; // Change "AcousticModelEnglish" to
"AcousticModelSpanish" to perform Spanish recognition instead of
English.
```

Warning

There can only be one PocketsphinxController instance in your app.

Method Documentation

```
- (void) startListeningWithLanguageModelAtPath: (NSString *) languageModelPath
                              dictionaryAtPath: (NSString *) dictionaryPath
                           acousticModelAtPath: (NSString *) acousticModelPath
                           languageModelIsJSGF: (BOOL)       languageModelIsJSGF
```

Start the speech recognition engine up. You provide the full paths to a language model and a dictionary file which are created using LanguageModelGenerator and the acoustic model you want to use (for instance [AcousticModel pathToModel:"AcousticModelEnglish"]).

```
- (void) stopListening
```

Shut down the engine. You must do this before releasing a parent view controller that contains PocketsphinxController.

```
- (void) suspendRecognition
```

Keep the engine going but stop listening to speech until resumeRecognition is called. Takes effect instantly.

```
- (void) resumeRecognition
```

Resume listening for speech after suspendRecognition has been called.

```
- (void) changeLanguageModelToFile: (NSString *)  languageModelPathAsString
                    withDictionary: (NSString *)  dictionaryPathAsString
```

Change from one language model to another. This lets you change which words you are listening for depending on the context in your app. If you have already started the recognition loop and you want to switch to a different language model, you can use this and the model will be changed at the earliest opportunity. Will not have any effect unless recognition is already in progress. It isn't possible to change acoustic models in the middle of an already-started listening loop, just language model and dictionary.

```
- (Float32) pocketsphinxInputLevel
```

Gives the volume of the incoming speech. This is a UI hook. You can't read it on the main thread or it will block.

```
- (void) runRecognitionOnWavFileAtPath: (NSString *)  wavPath
             usingLanguageModelAtPath: (NSString *)  languageModelPath
                     dictionaryAtPath: (NSString *)  dictionaryPath
                  acousticModelAtPath: (NSString *)  acousticModelPath
                  languageModelIsJSGF: (BOOL)        languageModelIsJSGF
```

You can use this to run recognition on an already-recorded WAV file for testing. The WAV file has to be 16-bit and 16000 samples per second.

Property Documentation

```
- (float) secondsOfSilenceToDetect
```

This is how long PocketsphinxController should wait after speech ends to attempt to recognize speech. This defaults to .7 seconds.

```
- (BOOL) returnNbest
```

Advanced: set this to TRUE to receive n-best results.

```
- (int) nBestNumber
```

Advanced: the number of n-best results to return. This is a maximum number to return – if there are null hypotheses fewer than this number will be returned.

```
- (int) calibrationTime
```

How long to calibrate for. This can only be one of the values '1', '2', or '3'. Defaults to 1.

```
- (BOOL) verbosePocketSphinx
```

Turn on verbose output. Do this any time you encounter an issue and any time you need to report an issue on the forums.

```
- (BOOL) returnNullHypotheses
```

By default, PocketsphinxController won't return a hypothesis if for some reason the hypothesis is null (this can happen if the perceived sound was just noise). If you need even empty hypotheses to be returned, you can set this to TRUE before starting PocketsphinxController.

```
- (NSString *) pathToTestFile
```

By setting pathToTestFile to point to a recorded audio file you can run the main Pocketsphinx listening loop (not runRecognitionOnWavFileAtPath but the main loop invoked by using startListeningWithLanguageModelAtPath:) over a pre-recorded audio file instead of using it with live input. In contrast with using the method runRecognitionOnWavFileAtPath to receive a single recognition from a file, with this approach the audio file will have its buffers injected directly into the audio driver circular buffer for maximum fidelity to the goal of testing the entire codebase that is in use when doing a live recognition, including the whole driver, the calibration code, and the listening loop including all of its features. This is for creating tests for yourself and for sharing automatically replicable issue reports with Politepix. To use this, make an audio recording on the same device (i.e., if you are testing PocketsphinxController on an iPhone 5 with the internal microphone, make a recording on an iPhone 5 with the internal microphone, for instance using Apple's Voice Memos app) and then convert the resulting file to a 16-bit, 16000 sample rate, mono WAV file. You can do this with the output of Apple's Voice Memos app by taking the .m4a file that Voice Memos outputs and run it through this command in Terminal.app: "afconvert -f WAVE -d LEI16@16000 -c 1 ~/Desktop/Memo.m4a ~/Desktop/Memo.wav" Then add the WAV file to your app, and right before sending the call to startListeningWithLanguageModelAtPath, set this property pathToTestFile to the path to your audio file in your app as an NSString (e.g. [[NSBundle mainBundle] pathForResource:"Memo" ofType:@"wav"]). Note: when you record the audio file you will be using to test with, **always** make sure to have 5 seconds of silence at the beginning so there is enough time for calibration to be performed on your recording environment, since calibration is also part of the test. SmartCMN is disabled during testing so that the test gets the same results when run for different people. Please keep in mind that there are some settings in Pocketsphinx which may prevent a deterministic outcome from a recognition, meaning that you should expect a **similar** score over multiple runs of a test but you may not always see the **identical** score. For this reason and the fact that PocketsphinxController is asynchronous and results in real practice are delivered via uncoupled callback it has not been designed as a purely automated test, but as an observed practical test. If it were designed as a purely automated test it would be testing something other than the way PocketsphinxController/OpenEarsEventsObserver works in an app, which is designed for good speech implementations rather than tests.

```
- (BOOL) audioSessionMixing
```

Set this to true in order to allow audio session mixing

```
- (NSString *) audioMode
```

If you are using 5.0 or greater, you can set audio modes for the audio session manager to use. This can be set to the following:

"Default" to use kAudioSessionMode_Default = â€˜dfltâ€™, @"VoiceChat" to use kAudioSessionMode_VoiceChat = â€˜vcctâ€™, @"VideoRecording" kAudioSessionMode_VideoRecording = â€˜vrcdâ€™, @"Measurement" kAudioSessionMode_Measurement = â€˜msmtâ€™

If you don't set it to anything, default will automatically be used.

<OpenEarsEventsObserverDelegate> Protocol Reference

Detailed Description

OpenEarsEventsObserver provides a large set of delegate methods that allow you to receive information about the events in OpenEars from anywhere in your app. You can create as many OpenEarsEventsObservers as you need and receive information using them simultaneously.

Usage examples

> *What to add to your header:*

Add the following lines to your header (the .h file). Under the imports at the very top:

```
#import <OpenEars/OpenEarsEventsObserver.h>
```

at the @interface declaration, add the OpenEarsEventsObserverDelegate inheritance. An example of this for a view controller called ViewController would look like this:

```
@interface ViewController : UIViewController
<OpenEarsEventsObserverDelegate> {
```

In the middle part where instance variables go:

```
OpenEarsEventsObserver *openEarsEventsObserver;
```

In the bottom part where class properties go:

```
@property (strong, nonatomic) OpenEarsEventsObserver
*openEarsEventsObserver;
```

> *What to add to your implementation:*

Add the following to your implementation (the .m file): Under the @implementation keyword at the top:

```
@synthesize openEarsEventsObserver;
```

Among the other methods of the class, add this lazy accessor method for confident memory management of the object:

```
- (OpenEarsEventsObserver *)openEarsEventsObserver {
 if (openEarsEventsObserver == nil) {
  openEarsEventsObserver = [[OpenEarsEventsObserver alloc] init];
 }
 return openEarsEventsObserver;
}
```

and then right before you start your first OpenEars functionality (for instance, right before your first self.fliteController say:withVoice: message or right before your first self.pocketsphinxController startListeningWithLanguageModelAtPath:dictionaryAtPath:languageModelIsJSGF: message) send this message:

```
[self.openEarsEventsObserver setDelegate:self];
```

> *How to use the class methods:*

Add these delegate methods of OpenEarsEventsObserver to your class:

```
- (void) pocketsphinxDidReceiveHypothesis:(NSString *)hypothesis
recognitionScore:(NSString *)recognitionScore utteranceID:(NSString
*)utteranceID {
 NSLog(@"The received hypothesis is %@ with a score of %@ and an ID of
%@", hypothesis, recognitionScore, utteranceID);
}

- (void) pocketsphinxDidStartCalibration {
 NSLog(@"Pocketsphinx calibration has started.");
}

- (void) pocketsphinxDidCompleteCalibration {
 NSLog(@"Pocketsphinx calibration is complete.");
}

- (void) pocketsphinxDidStartListening {
 NSLog(@"Pocketsphinx is now listening.");
}
```

```
- (void) pocketsphinxDidDetectSpeech {
 NSLog(@"Pocketsphinx has detected speech.");
}

- (void) pocketsphinxDidDetectFinishedSpeech {
 NSLog(@"Pocketsphinx has detected a period of silence, concluding an
utterance.");
}

- (void) pocketsphinxDidStopListening {
 NSLog(@"Pocketsphinx has stopped listening.");
}

- (void) pocketsphinxDidSuspendRecognition {
 NSLog(@"Pocketsphinx has suspended recognition.");
}

- (void) pocketsphinxDidResumeRecognition {
 NSLog(@"Pocketsphinx has resumed recognition.");
}

- (void) pocketsphinxDidChangeLanguageModelToFile:(NSString
*)newLanguageModelPathAsString andDictionary:(NSString
*)newDictionaryPathAsString {
 NSLog(@"Pocketsphinx is now using the following language model: \n%@
and the following dictionary:
%@",newLanguageModelPathAsString,newDictionaryPathAsString);
}

- (void) pocketSphinxContinuousSetupDidFail { // This can let you know
that something went wrong with the recognition loop startup. Turn on
OPENEARSLOGGING to learn why.
 NSLog(@"Setting up the continuous recognition loop has failed for some
reason, please turn on OpenEarsLogging to learn more.");
}
- (void) testRecognitionCompleted {
 NSLog(@"A test file that was submitted for recognition is now
complete.");
}
```

Method Documentation

`- (void) audioSessionInterruptionDidBegin`

There was an interruption.

`- (void) audioSessionInterruptionDidEnd`

The interruption ended.

`- (void) audioInputDidBecomeUnavailable`

The input became unavailable.

`- (void) audioInputDidBecomeAvailable`

The input became available again.

`- (void) audioRouteDidChangeToRoute: (NSString *)` *`newRoute`*

The audio route changed.

`- (void) pocketsphinxDidStartCalibration`

Pocketsphinx isn't listening yet but it started calibration.

`- (void) pocketsphinxDidCompleteCalibration`

Pocketsphinx isn't listening yet but calibration completed.

`- (void) pocketsphinxRecognitionLoopDidStart`

Pocketsphinx isn't listening yet but it has entered the main recognition loop.

`- (void) pocketsphinxDidStartListening`

Pocketsphinx is now listening.

`- (void) pocketsphinxDidDetectSpeech`

Pocketsphinx heard speech and is about to process it.

```
- (void) pocketsphinxDidDetectFinishedSpeech
```

Pocketsphinx detected a second of silence indicating the end of an utterance

```
- (void) pocketsphinxDidReceiveHypothesis: (NSString *)  hypothesis
                         recognitionScore: (NSString *)  recognitionScore
                              utteranceID: (NSString *)  utteranceID
```

Pocketsphinx has a hypothesis.

```
- (void) pocketsphinxDidReceiveNBestHypothesisArray: (NSArray *)  hypothesisArray
```

Pocketsphinx has an n-best hypothesis dictionary.

```
- (void) pocketsphinxDidStopListening
```

Pocketsphinx has exited the continuous listening loop.

```
- (void) pocketsphinxDidSuspendRecognition
```

Pocketsphinx has not exited the continuous listening loop but it will not attempt recognition.

```
- (void) pocketsphinxDidResumeRecognition
```

Pocketsphinx has not existed the continuous listening loop and it will now start attempting recognition again.

```
- (void) pocketsphinxDidChangeLanguageModelToFile: (NSString *)  newLanguageModelPathAsString
                                    andDictionary: (NSString *)  newDictionaryPathAsString
```

Pocketsphinx switched language models inline.

```
- (void) pocketSphinxContinuousSetupDidFail
```

Some aspect of setting up the continuous loop failed, turn on OpenEarsLogging for more info.

```
- (void) testRecognitionCompleted
```

Your test recognition run has completed.

```
- (void) fliteDidStartSpeaking
```

Flite started speaking. You probably don't have to do anything about this.

```
 - (void) fliteDidFinishSpeaking
```

Flite finished speaking. You probably don't have to do anything about this.