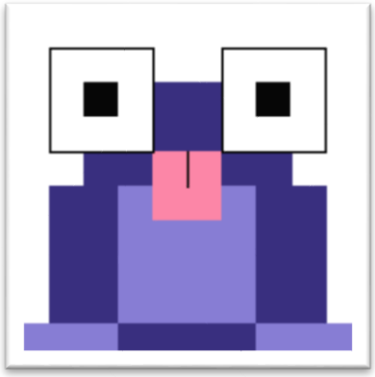
Ribbit

A Cost-Effective iOS Hearing Aid App



Developer's Manual v1.1

Computer Science Department
Texas Christian University
May 2, 2016

Revision Signatures

By signing the following, the team member is stating that he has read the entire document and has verified that the information contained within this document is accurate, relevant to the project, and void of errors.

Name	Signature	Date
Duy Dang		
Robert Kern		
Esteban Kleckner		

Revision History

Version	General Description of Changes	Date Completed
V1.0	Initial Draft	5/1/16
V1.1	Updated screenshot of QR generation	5/2/16

Contents

1 Introduction

1.1 Purpose

This document is intended to provide a detailed developers guide of the Ribbit iOS application, as well as the prescription creation process. Included will be what is required to continue working on the Ribbit iOS Application as well as how to access the QR Code Creation Website.

1.2 Project Overview

The objective of this project is to create an iOS application that functions similarly to a physical hearing aid device, but at a fraction of the cost. The application works within the federal regulations concerning the usage of hearing aids. The aim of the application is to correct the user's perception of sound by changing the sound to fit their inability to hear certain frequencies.

1.3 Overview of Document

- Section 2: In section 2, we give an overview of the different components of our project.
- Section 3: In section 3, we go over the required set up for the project.
- Section 4: In section 4, we give an overview of the main sections and classes of the Ribbit iOS Application.
- Section 5: In section 5, we go over the Accelerate framework in detail and how it is used within the Ribbit iOS Application.
- Section 6: In section 6, we give an overview of the QR Code Creation Website.

2 System Overview

2.1 System Components

Ribbit is composed of two parts: an **iOS Application** and the **QR Code Generation Website.** To access the QR code generation website, open a compatible web browser and navigate to the following URL: http://tcuhearing-ribbitcu.rhcloud.com/.

2.2 Ribbit iOS Application

The main component of the project is the iOS application. This is what the user will interact with on a regular basis. It is divided into four distinct parts: a home page, a table containing all prescriptions that have been read in, a camera view to read in additional QR code prescriptions, and a view to read what the prescription entails. More detail will be given to each part of the application in the following sections.

2.3 QR Code Prescription Generation Website

The QR code generation website is available to anyone through the above URL. The website was designed to allow an Audiologist or trained professional to rapidly generate prescriptions that will be used by the Ribbit iOS application.

3 Development Setup

3.1 iOS Development

In order to develop for iOS using the new standard language Swift 2, a computer running at least OS X 10.11.4 is required. In addition to the hardware requirement, Xcode must be installed on the computer. To install Xcode, if not already installed, visit https://itunes.apple.com/us/app/xcode/id497799835?mt=12. All development takes place within Xcode. In addition to the development computer, an Apple iPhone is also required. Any iPhone 5s or later running at least iOS 7.0. All testing for the application will occur on the iPhone.
In order to put the application on the App Store, an Apple Developer Program is also required. If you are not planning to upload the application to the App Store, an account is not required. What is required regardless is an Apple ID. To create an Apple ID, go to https://appleid.apple.com/account.

Source code for the iOS portion of this project are located within the Source_Code/Ribbit/ folder of the DVD.

3.2 Web Development

In order to develop an application for the internet, any computer with an active internet connection will work. Cloud hosting for the QR Generation Website was done using Red Hat OpenShift, https://www.openshift.com/. Contact Dr. Ma for credentials to access the hosting settings. Required software include: a text editor, a web browser, RHC (Red Hat Cloud client – Instructions: https://developers.openshift.com/managing-your-applications/client-tools.html), Ruby (required to run RHC), and git (used to push code to OpenShift server and deploy the application).

Openshift's getting started instructions for Windows covering Ruby, Red Hat Client (RHC), and git: https://developers.openshift.com/getting-started/windows.html

Source code for the QR Creation Website is located within the Source_Code/QR_Creation/ folder of the DVD.

4 Accelerate Framework

4.1 What is it

The Accelerate Framework is the API that Apple provides for vector mathematics and digital signal processing, amongst other things. Specifically, for this project, we are using its vDSP portion. vDSP contains methods for vector multiplication and Discrete Fourier Transforms that allow us to take a signal in the time domain and transform it into the frequency domain.

4.2 How do we use it?

Within our application, the vDSP is used for its Fast Fourier Transform (FFT) methods. The FFT, and the Inverse FFT allow us to process input signals and make adjustments to them before sending them back to the user. The FFT takes the sound in the time domain, and transforms it into the frequency domain. Processing sound in the frequency domain is often times more efficient than processing sound in the time domain. Once the signal is in the frequency domain, we apply a Combination Window that filters and amplifies the signal to fit the user's prescription. The signal is then transformed back into the time domain using the inverse FFT. Finally, the signal is stitched together using Partitions of Unity as a method of Transitions Smoothing.

Another application of the vDSP Library within Ribbit is vector multiplication. When we are apply filtering and gain modification to the signal, we create a mathematical window for the signal to be multiplied by. This multiplication is done using SIMD operations, thus requiring the vDSP. All of this multiplication occurs within **FilterWindow.swift** and **SoundEngine.swift**.

5 Ribbit Application

5.1 Class Explanation

The Ribbit iOS Application contains a total of 21 classes that are interconnected. Due to the way that the Swift programming language interprets code, these different files are created purely for readability's sake. When Swift compiles the code, everything is read as one large file.

All of the application's files are listed. More details for each is given in the respective sections.

- **AudioController.swift –** This class contains functions related to sound I/O. Once the sound is received, it is sent to **SoundEngine.swift** where it is processed.
- **ClassExtensions.swift –** This class contains helper functions for the AudioController.
- **FFToolset.swift –** This class contains the FFT functions that are used throughout the project.
- **Globals.swift –** This class contains all of the global variables used throughout the project.
- **HomeViewController.swift –** This is the class loaded by the Home Page ViewController. It contains the functions to authenticate the user via Touch ID and to set the active prescription label.
- **QRScanViewController.swift –** This class contains the code to implement QR Code scanning as well as how to load the information from the QR Codes into both the prescription table and into Core Data.
- **Prescription.swift –** This class contains all information related to the prescriptions, including the RxData struct, our QR Code parser function, and the struct for representing Frequency Bands as thresholds.
- **RxDataViewController.swift –** This is the class loaded by the Rx Data View Controller. It contains the functions to display the prescription as well as those needed to allow the user to set a specific prescription as active.
- **RxTableViewController.swift –** This class contains the functions needed to run the table view used for the prescriptions. It also contains the necessary functions to interface with and load from Core Data.
- **SoundEngine.swift –** This class contains the necessary functions to process the sound, both incoming and outgoing, and to combine both the Filter and Gain windows into one window.
- **WindowGeneration.swift –** This class contains the necessary functions to create the Filter, Gain, and Weight windows.

Each larger part of the application uses these classes, and more, in different ways. These will be discussed in their respective sections.

5.2 Touch ID

When the Ribbit application is launched, it will ask for the user to input their Touch ID information. All functionality is added with the **HomeViewController.swift** within **authenticateUser().** All Touch ID related coding requires the **LocalAuthentication** framework to be imported. If the user does not have Touch ID enabled on their device, the application brings up another window so that they may enter a password. If the user clicks **"Cancel"**, the application sends an **"Authentication was cancelled by the user"** notification and closes the application. This is done in an effort to protect the patient's data from outside access. All Touch ID functionality is done by calling on methods created by Apple and extending them to be used within the confines of the Ribbit application.

```
func authenticateUser() {
    // Get the local authentication context.
    let context = LAContext()

    // Declare a NSError variable.
    var error: NSError?

    // Set the reason string that will appear on the authentication alert.
    let reasonString = "Authentication is needed to access your prescriptions."

    // Check if the device can evaluate the policy.
    if context.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, error: &error) {
        [context .evaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, localizedReason: reasonString, reply: {
            (success: Bool, evalPolicyError: NSError?) -> Void in

            if success {
                print("Authentication succeeded")
            }
            else{
                // If authentication failed then show a message to the console with a short description.
                // In case that the error is a user fallback, then show the password alert view.
                print(evalPolicyError?.localizedDescription)

                switch evalPolicyError!.code {

                case LAError.SystemCancel.rawValue:
                    print("Authentication was cancelled by the system")

                    //handles the case if the user hits cancels touchid authentication
                //displays a alert telling the user that either touchid or a password must be used, then closes the app
                case LAError.UserCancel.rawValue:
                    print("Authentication was cancelled by the user")
                    NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in self.showCancelAlert() })
                    [NSThread .sleepForTimeInterval(5)]
                    exit(0)

                case LAError.UserFallback.rawValue:
                    print("User selected to enter custom password")
                    NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in self.showPasswordAlert() })

                default:
                    print("Authentication failed")
                    NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in self.showPasswordAlert() })
                }
            }

        })]
    }
```

5.3 Filter Window Generation

The filter window is generated once at application start. It is considered a low pass frequency filter; it has three characteristic parts: low pass, transition, and no pass. The low pass section is characterized by ones, which allows the signal to maintain its original data. The transition section is necessary to avoid any divide by zero errors, when applying the FFT or inverse FFT, it's designed to be infinitely differentiable. The final section is characterized by zeros, where the input signal will be eliminated from the final output signal.

```
    // Filtering
var curveSlice: [Float] = [0.939358, 0.494892, 0.11739]          // infinitely differentiable curve slice
var lowPassSlice = [Float](count: 86, repeatedValue: 1.0)        // part of the window that passes values
var filterWin = [Float](count: 846, repeatedValue: 0.0)          // will become main filter window

// Create the Filter Window
func generateFilterWindow() {

    // construct filter window
    filterWin = lowPassSlice + curveSlice + filterWin
    filterWin = filterWin + curveSlice.reverse() + lowPassSlice

    // Debug Info
    print("Filter Window Generated.")
    print("|filterWin|: \(filterWin.count)")

`
```

5.4 Gain Window Generation

A new pair of gain windows (one for left ear, one for right ear) are generated every time a new prescription is set to be active. The gain window contains values greater than or equal to 1 so that it will increase the power (and ultimately the decibel level) of the frequency domain signal after multiplication.

```
    // Gain
var stepWin = [Float](count: 11, repeatedValue: 1.0)             // infinitely differentiable gain slice
var gainWinL = [Float](count: 1024, repeatedValue: 1.0)          // will become main gain window (Left)
var gainWinR = [Float](count: 1024, repeatedValue: 1.0)          // will become main gain window (Right)
```

However, in order to reduce the Gibbs Effect and artifacts caused by spectral leakages, to increase the power of a frequency bin by a certain ratio, we will need to increase the neighboring bins to a decreasing extents as it is getting farther from the current bin of focus. This is achieved by using a step window which increases gradually from 1 to the desire ratio then decreases back to 1.

```
// Create Step Window
func generateStepWindow() {

    stepWin[0] = 1
    stepWin[1] = 1.0280574
    stepWin[2] = 1.163066
    stepWin[3] = 1.336996
    stepWin[4] = 1.472005
    stepWin[5] = 1.500062
    stepWin[6] = 1.472005
    stepWin[7] = 1.336996
    stepWin[8] = 1.163066
    stepWin[9] = 1.0280574
    stepWin[10] = 1

    // Debug Info
    print("Step Window Generated.")
    print("|stepWin|: \(stepWin.count)")
}
```

The ratio will be determined based on the decibel level that the frequency needs to be increased by. The **dBDiff()** function will take in the current frequency in focus, the hearing data of the user, and the hearing data of a normal hearing person and returns this decibel level.

```
//
func dBDiff(frequency: Float, impaired: [HearingThreshold], normal: [HearingThreshold]) -> Float {

    // local variables
    var impaired_dB: Float = 0
    var normal_dB: Float = 0
    var tempSlope: Float = 0

    // local indices
    var index1 = 0
    var index2 = 0

    // index of frequency bin 250Hz in prescription is 1, index of frequency bin 500Hz in prescription is 2
    if (frequency >= 250 && frequency < 500) {

        index1 = 1
        index2 = 2

    // index of frequency bin 500Hz in prescription is 2, index of frequency bin 1000Hz in prescription is 3
    } else if (frequency >= 500 && frequency < 1000) {

        index1 = 2
        index2 = 3

    // index of frequency bin 1000Hz in prescription is 3, index of frequency bin 2000Hz in prescription is 4
    } else if (frequency >= 1000 && frequency < 2000) {

        index1 = 3
        index2 = 4

    // index of frequency bin 2000Hz in prescription is 4, index of frequency bin 4000Hz in prescription is 5
    } else if (frequency >= 2000 && frequency < 4000) {

        index1 = 4
        index2 = 5
    } else if (frequency >= 4000 && frequency < 8000) {
```

```
    // index of frequency bin 2000Hz in prescription is 4, index of frequency bin 4000Hz in prescription is 5
    } else if (frequency >= 2000 && frequency < 4000) {

        index1 = 4
        index2 = 5
    } else if (frequency >= 4000 && frequency < 8000) {

        index1 = 5
        index2 = 6

    }

    // calculate the slope of the line connecting dB levels of 2 adjacent frequency bins in prescription in order to get the
        corresponding dB level of the given frequency
    tempSlope = (Float(impaired[index2].threshold!) - Float(impaired[index1].threshold!)) / (Float(impaired[index2].band!) -
        Float(impaired[index1].band!))
    impaired_dB = Float(impaired[index1].threshold!) + tempSlope * (frequency - Float(impaired[index1].band!))

    tempSlope = (Float(normal[index2].threshold!) - Float(normal[index1].threshold!)) / (Float(normal[index2].band!) - Float
        (normal[index1].band!))
    normal_dB = Float(normal[index1].threshold!) + tempSlope * (frequency - Float(normal[index1].band!))


    return impaired_dB - normal_dB
}
```

After that, this decibel level will be put in the formula: **ratio = 10 ^ (decibel difference / 20)** to get the ratio. This ratio will be used to build a step window with the top value equal to the ratio. This step window will then be put into the right section of the gain window. The process of creating new step window to be put into the gain window will be done in a loop to cover all frequency bins in the frequency domain. Notice that this loop will be done twice, each for one ear (left and right).

```
// Create Gain Window
func generateGainWindow() {


    //Populate gainWindow
    var frequency: Float = 0
    var gainNeeded: Float = 0
    var magRatio: Float = 0

    // Process Gain Window (Left Ear)
    for var i = 5; i < 124 - 5; i+=5 {

        frequency = Float(binWidth * (i + 1))


        gainNeeded = dBDiff(frequency, impaired: rxActive!.leftEar!, normal: rxGain.leftEar!)

        magRatio = pow(10, gainNeeded / 20)

        for j in 0 ..< 11 {

            gainWinL[i-5+j] = gainWinL[i-5+j] + ( (stepWin[j] - 1) * (magRatio - 1) / 0.5 )

            if (1024 - ( i - 5 + j) < 1024) {
                gainWinL[1024 - ( i - 5 + j )] = gainWinL[1024 - ( i - 5 + j )] + ( (stepWin[j] - 1) * (magRatio - 1) / 0.5 )
            }
        }

    }

    // Process Gain Window (Right Ear)
    for var i = 5; i < 124 - 5; i+=5 {

        frequency = Float(binWidth * (i + 1))

        gainNeeded = dBDiff(frequency, impaired: rxActive!.rightEar!, normal: rxGain.rightEar!)

        magRatio = pow(10, gainNeeded / 20)


        for j in 0 ..< 11 {

            gainWinR[i-5+j] = gainWinR[i-5+j] + ( (stepWin[j] - 1) * (magRatio - 1) / 0.5 )

            if (1024 - ( i - 5 + j) < 1024) {
                gainWinR[1024 - ( i - 5 + j )] = gainWinR[1024 - ( i - 5 + j )] + ( (stepWin[j] - 1) * (magRatio - 1) / 0.5 )
            }
        }

    }

    // Debug Info
    print("Gain Windows Generated.")
    print("|gainWinL|: \(gainWinL.count)")
    print("|gainWinR|: \(gainWinR.count)")
}
```

5.5 Combination Window Generation

```
// Global Windows
    // Combined Window
var combinedWinL = [Float](count: 1024, repeatedValue: 1.0)        // combined Filter and Gain window
var combinedWinR = [Float](count: 1024, repeatedValue: 1.0)        // combined Filter and Gain window
```

Every time a new hearing prescription (QR code) is set active, the application will call to **generateGainWindow()** to generate a new pair of gain windows (for left and right ear) that reflects the hearing data in the new prescription. After that, the Combination window for the left ear (**combineWinL**) will be set to as the product of the Filter Window and the Gain Window for the left ear. Similarly, the combination window for the right ear (**combineWinR**) will be set as the product of the Filter Window and the new Gain Window for the right ear. The two combination windows are now ready to be applied to the signal in the frequency domain.

5.6 Partitions of Unity

Partitions of Unity is the technique used to smooth out the mismatch in the transitions between processed frames, which is caused by modifying discrete signal in the frequency domain using FFT and IFFT. Our implementation of Partitions of Unity will modify the signal in not only adjacent signal frames in the frequency domain but also the overlapping frames (50% overlapping before and after the current frame in focus). Then, the two signals in the overlapping sections will be blended into each other by gradually decreasing the weight of one while increasing that of the other (the total weight in any point must be equal to 1).

As a result, for every frame that we output we will need to apply the combination window three times (one to the current signal of focus, two to the overlapping frames). However, after optimization, our implementation will only need to do this twice for every frame interval but the mechanism and effect remain intact. To make several applications of combination window easier, we create the function **applyCombinedWindow()** that can put the time domain signal into frequency domain, then modify by multiplying the combination window to the frequency domain signal, then transform this processed frequency domain signal back to the time domain.

```
func applyCombinedWindow( real: UnsafeMutablePointer<Float>, imag: UnsafeMutablePointer<Float>, channel: String ) {

    // Call Function to perform FFT, and return a list of (DSPSplitComplex, and FFTSetup) tuples
    let setup = FFT(real, imagp: imag, length: fftLength)

    if channel == "left" {
        // Multiply signal by Combined Window
        vDSP_vmul(setup.SC.realp, 1, &combinedWinL, 1, setup.SC.realp, 1, length)
        vDSP_vmul(setup.SC.imagp, 1, &combinedWinL, 1, setup.SC.imagp, 1, length)
    } else {
        // Multiply signal by Combined Window
        vDSP_vmul(setup.SC.realp, 1, &combinedWinR, 1, setup.SC.realp, 1, length)
        vDSP_vmul(setup.SC.imagp, 1, &combinedWinR, 1, setup.SC.imagp, 1, length)
    }

    // Invert the FFTs
    invFFT(setup.FFTSetup, merged: setup.SC, length: fftLength, size: frameSize)


    // Destroy FFT Setup
    vDSP_destroy_fftsetup(setup.FFTSetup)

}
```

After we have got the processed time domain signals (one current frame and two overlapping frames) we will be able to apply the weight window and blend them together to create a time domain signal that will smoothly transition into the next frame.

```
//
//  SoundEngine.swift
//  Ribbit
//
//  Copyright © 2016 Texas Christian University. All rights reserved.
//

import Foundation
import Accelerate

func processSignal( buffer: [Float], channel: String ) -> [Float] {


    // Transition Smoothing local variable
    var real = [Float](count: 1024, repeatedValue: 0.0)
    var imag = [Float](count: 1024, repeatedValue: 0.0)
    var overlapWin = [Float](count: 1024, repeatedValue: 0.0)
    var overlapImag = [Float](count: 1024, repeatedValue: 0.0)

    if channel == "left" {
        // Local Variables
        real = prevWinL
        prevWinL = buffer

        // Assuming the "now" real buffer (used to be previous before the swap) has already been applyGainFilter
        for i in 0 ..< 512 {

            overlapWin[i] = real[i+512]         // From real
            overlapWin[i+512] = prevWinL[i]     // From previousBuffer (storing current buffer)

        }
    } else {
        // Local Variables
        real = prevWinR
        prevWinR = buffer

        // Assuming the "now" real buffer (used to be previous before the swap) has already been applyGainFilter
        for i in 0 ..< 512 {

            overlapWin[i] = real[i+512]         // From real
            overlapWin[i+512] = prevWinR[i]     // From previousBuffer (storing current buffer)

        }
    }

    // applyCombinedWindow to the real
    applyCombinedWindow(&real, imag: &imag, channel: channel)

    // Multiply real by weightWindow
    vDSP_vmul(&real, 1, &weightWindow, 1, &real, 1, vDSP_Length(1024))

    // applyGainFilter to overlapWin  (note: overlapBuffer1 has already been applyGainFiltered)
    applyCombinedWindow(&overlapWin, imag: &overlapImag, channel: channel)

    // Multiply overlapWin by weightWindow
    vDSP_vmul(&overlapWin, 1, &weightWindow, 1, &overlapWin, 1, vDSP_Length(1024))

    // Add weighted right half of overlapBuffer1 to weighted left half of real
    // Add weighted left half of overlapBuffer2 to weighted right half of real
    // Put content of overlapBuffer2 into overlapBuffer1
    if channel == "left" {
        for i in 512 ..< 1024 {

    // Add weighted right half of overlapBuffer1 to weighted left half of real
    // Add weighted left half of overlapBuffer2 to weighted right half of real
    // Put content of overlapBuffer2 into overlapBuffer1
    if channel == "left" {
        for i in 512 ..< 1024 {

            real[i - 512] += overPrevWinL[i]
            real[i] += overlapWin[i - 512]
            overPrevWinL[i] = overlapWin[i]

        }
    } else {
        for i in 512 ..< 1024 {

            real[i - 512] += overPrevWinR[i]
            real[i] += overlapWin[i - 512]
            overPrevWinR[i] = overlapWin[i]

        }
    }


    return real

}
```

5.7 Core Data

Core Data is Apple's form of data persistence within their products. It is effectively a SQLite database saved within the application's file system. Core Data is intended to reduce the amount of code that is used to save data, such as the prescriptions for Ribbit. In order to work, Core Data needs access to the CoreData framework as well as a .xcdatamodelID file that includes the schema for the database that the information to be saved in. For more detailed information on how to set up Core Data, go to https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/index.html#//apple_ref/doc/uid/TP40001075-CH2-SW1.

When you create a new iOS project that includes Core Data, Xcode automatically creates the Core Data stack that will be loaded every time that the application is loaded. This includes creating the PersistentStoreController, the part that acts a scratch board for Core Data before things are saved, the managedObjectModel, the model that includes the data currently saved within Core Data as well as information waiting to be saved, and sets the saving URL within the application's file system.

```
// MARK: - Core Data stack

lazy var applicationDocumentsDirectory: NSURL = {
    // The directory the application uses to store the Core Data store file. This code uses a directory named "TCU.Ribbit" in the
        application's documents Application Support directory.
    let urls = NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // The managed object model for the application. This property is not optional. It is a fatal error for the application not to
        be able to find and load its model.
    let modelURL = NSBundle.mainBundle().URLForResource("Ribbit", withExtension: "momd")!
    return NSManagedObjectModel(contentsOfURL: modelURL)!
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This implementation creates and returns a coordinator, having added
        the store for the application to it. This property is optional since there are legitimate error conditions that could
        cause the creation of the store to fail.
    // Create the coordinator and store
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
    let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading the application's saved data."
    do {
        try coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: url, options: nil)
    } catch {
        // Report any error we got.
        var dict = [String: AnyObject]()
        dict[NSLocalizedDescriptionKey] = "Failed to initialize the application's saved data"
        dict[NSLocalizedFailureReasonErrorKey] = failureReason

        dict[NSUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999, userInfo: dict)
        // Replace this with code to handle the error appropriately.
        // abort() causes the application to generate a crash log and terminate. You should not use this function in a shipping
            application, although it may be useful during development.
        NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
        abort()
    }

    return coordinator
}()
```

To incorporate the QR Reader into Core Data, you must set the incoming data as a savable format. This means set the incoming metadata steams as an entity for the Core Data database. Once you have it in the correct format, you can add it to the Prescription Table, and then save it into the Core Data database.

```
// create entity object
let entity = NSEntityDescription.entityForName("AudioGram", inManagedObjectContext: managedContext)

// create prescription object
let prescription = NSManagedObject(entity: entity!, insertIntoManagedObjectContext: managedContext)

// set prescription value, set only input string for later parsing
prescription.setValue(qrData!, forKey: "prescription")

// save the context
do {

    audiograms.append(prescription)
    try managedContext.save()

} catch let err as NSError {
    print("Could not save!: \(err) \n \(err.userInfo)\n")
}
```

Once the new prescription data has been read into the Core Data database and has been loaded onto the Prescription Table, you have to tell the Prescription Table to refresh so to get the new information. This is done within the **viewWillAppear()** function in the **RxDataViewController.swift.** This function is called every time that the ViewController is loaded, or is force loaded by the refreshing the table. When the table refreshes, this function tells the table to query the Core Data database and pull any new prescriptions that are not already stored in the Prescription Table.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    // set up fetchRequest
    let fetchRequest = NSFetchRequest(entityName: "AudioGram")

    do {
        let results = try managedContext.executeFetchRequest(fetchRequest)

        print(" \n\n|audiograms|: \(audiograms.count)")
        audiograms = results as! [NSManagedObject]
        print("|audiograms'|: \(audiograms.count)\n\n")

        let numberAudioGrams = audiograms.count

        if numberAudioGrams != 0 {

            rxTable.removeAll()

            for i in 0 ..< numberAudioGrams {

                let audiogram = audiograms[i]
                let agString = audiogram.valueForKey("prescription") as? String
                rxTable.append(qrParse(agString!))

            }
        }

    } catch let error as NSError {
        print("Could not fetch!: \(error) \n \(error.userInfo)")
    }
}

// Reloads tableView from external class
func refreshList(notification: NSNotification){
    tableView.reloadData()
}
```

5.8 QR Reader

In order to easily and securely read in prescriptions, we have gone with a QR Reader approach. In order to utilize a QR code reader in Swift, you have to import the **AVFoundation** framework, which stands for Audio Video Foundation. This allows this class to have access to both the microphone and the cameras on the phone. As this section of the application only uses the rear camera in order to successfully read in the QR Code.

The code below shows how to initialize the rear camera and to start up the camera capture when the ViewController is first loaded. The incoming information from the camera is read to a AVCaptureMetadataOutput() method, as the incoming data is seen as metadata before it is given context. This context is given when the call "**captureMetadataOutput.metadataObjectTypes = [AVMetadataObjectTypeQRCode]"** is made. From that point on, the application is looking for anything that looks like a QR Code.

After the camera is initialized and given context, you have to start up a videoPreviewLayer, which allows the user to see what the camera sees so they know where to aim the camera. After the videoPreviewLayer is created and added to the view layer, you start the capture session. Then you had a QR Code frame that appears when the camera finds a QR Code within its view area.

```
do {
    input = try AVCaptureDeviceInput(device: captureDevice)
} catch let error as NSError {
    print("Capture Device Error!: \(error) \n \(error.userInfo)")
}

// Initialize the captureSession object.
captureSession = AVCaptureSession()
// Set the input device on the capture session.
captureSession?.addInput(input as! AVCaptureInput)

// Initialize a AVCaptureMetadataOutput object and set it as the output device to the capture session.
let captureMetadataOutput = AVCaptureMetadataOutput()
captureSession?.addOutput(captureMetadataOutput)

// Set delegate and use the default dispatch queue to execute the call back
captureMetadataOutput.setMetadataObjectsDelegate(self, queue: dispatch_get_main_queue())
captureMetadataOutput.metadataObjectTypes = [AVMetadataObjectTypeQRCode]

// Initialize the video preview layer and add it as a sublayer to the viewPreview view's layer.
videoPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
videoPreviewLayer?.videoGravity = AVLayerVideoGravityResizeAspectFill
videoPreviewLayer?.frame = view.layer.bounds
view.layer.addSublayer(videoPreviewLayer!)

captureSession?.startRunning()

// Initialize QR Code Frame to highlight the QR code
qrCodeFrameView = UIView()
qrCodeFrameView?.layer.borderColor = UIColor.greenColor().CGColor
qrCodeFrameView?.layer.borderWidth = 2
view.addSubview(qrCodeFrameView!)
view.bringSubviewToFront(qrCodeFrameView!)
```

If a QR is found, the camera processes the metadata into a string so that it can be worked with. The QRController makes sure that it contained the phrase "rx info", parses the QR Code, displays the

prescription information inside of the QR code so the user can verify the information. After it is displayed, the QRController checks to see if that Prescription is already in the table. If the prescription is not there, the Prescription is loaded into the prescription table and saved as a new Prescription in Core Data.

```
if metadataObj.stringValue != nil
{

    let qrData = metadataObj.stringValue

    if qrData.containsString("rxinfo") {

        // Parse QR Code and add to table
        rxDat = qrParse(qrData)

        // black magic compares if the previous object has been added before
        // equality is based on a keystring = <rxName + patient + prescriber + date + debug>
        if !rxTable.contains( { $0.equals(rxDat!)! }) && !rxDat!.equals(rxGain)! {

            // Generate Popup Message for Imported Prescription
            var messageString = "Prescription \"" + (rxDat?.rxName!)! + "\" Imported!\n" + (rxDat?.date!)!

            if (rxDat?.debug!)! == true {
                messageString += "\nBase Gain Adjusted!"
            }

            let messageAlert = UIAlertController(title: "QR Code Info",message: messageString, preferredStyle:
                UIAlertControllerStyle.Alert)
            messageAlert.addAction(UIAlertAction(title: "Dismiss", style: UIAlertActionStyle.Default, handler: nil))
            self.presentViewController(messageAlert, animated: true, completion: nil)

            // if debug == true; change gain basis if prescription is set to Debug
            if (rxDat?.debug!)! == true {
                print("\n\nrxDebug was: \(rxGain)")
                rxGain = rxDat!
                print("\nrxDebug is: \(rxGain)\n")

            // otherwise, input prescription into table
            } else {
                rxTable.append((rxDat!))
                print("|rxTable|: \(rxTable.count)")

                // add new prescriptions to the persistent table

                // create entity object
                let entity = NSEntityDescription.entityForName("AudioGram", inManagedObjectContext: managedContext)

                // create prescription object
                let prescription = NSManagedObject(entity: entity!, insertIntoManagedObjectContext: managedContext)

                // set prescription value, set only input string for later parsing
                prescription.setValue(qrData!, forKey: "prescription")

                // save the context
                do {

                    audiograms.append(prescription)
                    try managedContext.save()

                } catch let err as NSError {
                    print("Could not save!: \(err) \n \(err.userInfo)\n")
                }

                // send notification to reload tableView
                NSNotificationCenter.defaultCenter().postNotificationName("refreshMyTableView", object: nil)
            }
        } else {
            // Generate Popup Message for Imported Prescription
            let messageString = "QR Code has already been scanned!"

            let messageAlert = UIAlertController(title: "QR Code Info",message: messageString, preferredStyle:
                UIAlertControllerStyle.Alert)
            messageAlert.addAction(UIAlertAction(title: "Dismiss", style: UIAlertActionStyle.Default, handler: nil))
            self.presentViewController(messageAlert, animated: true, completion: nil)
        }
```

If the QR code contains any other information of than our prescription format, i.e. a hyperlink, the captureSession displays a dialog saying “Invalid QR Code.”

```
} else {

    rxDat = nil

    let messageAlert = UIAlertController(title: "QR Code Info",message: "Invalid QR Code",preferredStyle:
        UIAlertControllerStyle.Alert)
    messageAlert.addAction(UIAlertAction(title: "Dismiss", style: UIAlertActionStyle.Default, handler: nil))
    self.presentViewController(messageAlert, animated: true, completion: nil)
}
```

6 QR Code Generation Website

6.1 Red Hat OpenShift

For the creation of the prescription QR codes, we chose to use a web hosting service in Red Hat OpenShift hosting, which allows us to host on a Red Hat Enterprise Linux server. For access to the online console, go to https://www.openshift.com/, click on My Account, choose OpenShift Web Console, and enter the credentials that Dr. Ma has given you. The only domain there should be “tcuhearing”. This is the domain that hosts our QR Code generation website.

6.2 QR Generation

Our QR Code generation site is running HTML5. All of the heavy lifting is done in **Node.js 0.10.** Node.js is a scalable derivative of Javascript that allows multiple network connections at a given time. It has the same syntax of Javascript and follows the same scripting formatting.

```
<script type="text/javascript">
            function genQRcode() {

                var rxString = "";
                rxString = rxString + "Rxtype: " + $("#info1").val() + "\n";
                rxString = rxString + "Patient: " + $("#info2").val() + "\n";
                rxString = rxString + "Presriber: " + $("#info3").val() + "\n";
                rxString = rxString + "Date: " + $("#info4").val() + "\n";
                rxString = rxString + "Debug: " + $("#info5").val() + "\n";

                rxString = rxString + "leftear {\n";
                rxString = rxString + "(125): " + $("#info6").val() + "\n";
                rxString = rxString + "(250): " + $("#info7").val() + "\n";
                rxString = rxString + "(500): " + $("#info8").val() + "\n";
                rxString = rxString + "(1000): " + $("#info9").val() + "\n";
                rxString = rxString + "(2000): " + $("#info10").val() + "\n";
                rxString = rxString + "(4000): " + $("#info11").val() + "\n";
                rxString = rxString + "(8000): " + $("#info12").val() + "\n";
                rxString = rxString + "}\n";

                rxString = rxString + "rightear {\n";
                rxString = rxString + "(125): " + $("#info13").val() + "\n";
                rxString = rxString + "(250): " + $("#info14").val() + "\n";
                rxString = rxString + "(500): " + $("#info15").val() + "\n";
                rxString = rxString + "(1000): " + $("#info16").val() + "\n";
                rxString = rxString + "(2000): " + $("#info17").val() + "\n";
                rxString = rxString + "(4000): " + $("#info18").val() + "\n";
                rxString = rxString + "(8000): " + $("#info19").val() + "\n";
                rxString = rxString + "}";

                $("#QR_frame").val("");

                $("#QR_frame").qrcode({
                    "render": "div",
                    "size": 250,
                    "color": "#3a3",
                    "text": rxString
                });
            };
</script>
```

All computation for the QR Code is done on the user’s browser so that no medical information is sent of the internet. This is done for medical information security.

Glossary of Terms

Gibb's Effect: the effects related to signal mismatch happening when using FFT and IFFT in discrete signal processing

Javascript: A scripting programming language used to make webpages dynamic.

QR Code: A form of bar code used to link different websites or to send information

Spectral Leakage: Occurs when an incoming frequency does not match a frequency bin in the FFT

Touch ID: A security mechanism used to maintain exclusive access based on a user's fingerprint