iOS Development Guide for ThinkGear

Features

- Develop iOS applications that utilize ThinkGear technology
- Downloadable ThinkGear-enabled sample iOS project with full sample code
- Uses the iOS External Accessories API (available in iPhone OS 3.0+) or the ThinkGear iOS API

Introduction

Thanks to the availability of the MindWave Mobile, developers can now create iOS applications that can sense users' brainwaves. This application note will walk you through the process of creating a MindWave-capable iOS application.

This application note documents the usage of two different APIs to connect to a MindWave Mobile.

SDK Bugs and Issues

The current iteration of the SDK has the following limitations:

- There are some static analyzer warnings in the FSKLibrary, they are safe to ignore.

Hardware

The ThinkGear iOS API supports the following hardware:

- The MindWave Mobile which connects though an iOS device's built in Bluetooth
- The MindSet Link dongle which plugs into the iOS device's 30-pin connector and pairs wirelessly to a standard MindSet
- Compatible devices which plug into the iOS device's audio jack

Using the ThinkGear iOS API

For most applications, using the ThinkGear iOS API is recommended. It reduces the complexity of managing ThinkGear accessory connections and handles parsing of the data stream from these ThinkGear accessories. To make a brainwave-sensing application, all you need to do is to import a library, add the requisite setup and teardown functions, and assign a delegate object to which accessory event notifications will be dispatched.

Some limitations of the ThinkGear iOS API include:

- Can only communicate with one attached ThinkGear-enabled accessory
- Depending on the value of the user-configured event dispatch interval, some data received from the headset may be discarded

The thinkgear_ios_api_reference contains descriptions of the classes and protocols available in the ThinkGear iOS API.

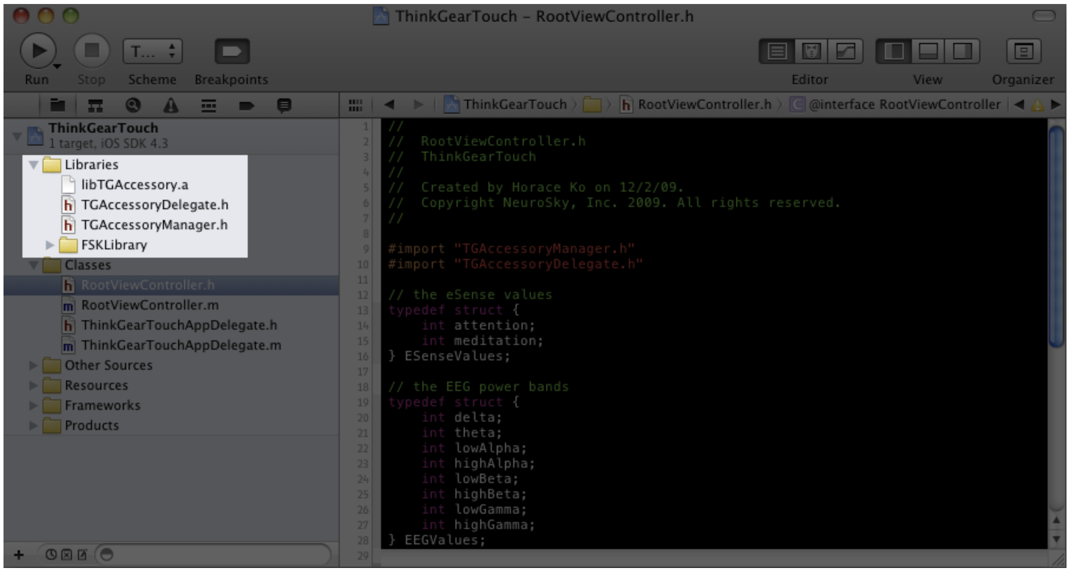
The ThinkGear iOS SDK also includes the `ThinkGearTouch` sample project (contained in `src/`), which is a simple `UITableView`-based iOS application that displays the data coming from a MindSet headset.

Configuring Your Environment

Simply copy the following directories from the `src/lib` directory in the ThinkGear iOS SDK into the `Libraries` group in your project:

- `libTGAccessory.a`
- `TGAccessoryDelegate.h`
- `TGAccessoryManager.h`
- `FSKLibrary/`

Your project window should now look similar to this:



```
//
//  RootViewController.h
//  ThinkGearTouch
//
//  Created by Horace Ko on 12/2/09.
//  Copyright NeuroSky, Inc. 2009. All rights reserved.
//

#import "TGAccessoryManager.h"
#import "TGAccessoryDelegate.h"

// the eSense values
typedef struct {
    int attention;
    int meditation;
} ESenseValues;

// the EEG power bands
typedef struct {
    int delta;
    int theta;
    int lowAlpha;
    int highAlpha;
    int lowBeta;
    int highBeta;
    int lowGamma;
    int highGamma;
} EEGValues;

```

Figure 1: Xcode project window with the ThinkGear iOS library

Next, add the AudioToolbox and the ExternalAccessory frameworks to the project.

1. Navigate to your project settings
2. Select your target
3. Select Build Phases
4. Expand **Link Binary With Libraries**

5. Click on `+` and select `AudioToolbox.framework` and `ExternalAccessory.framework` and click **Add**

Your project window should now look similar to this:

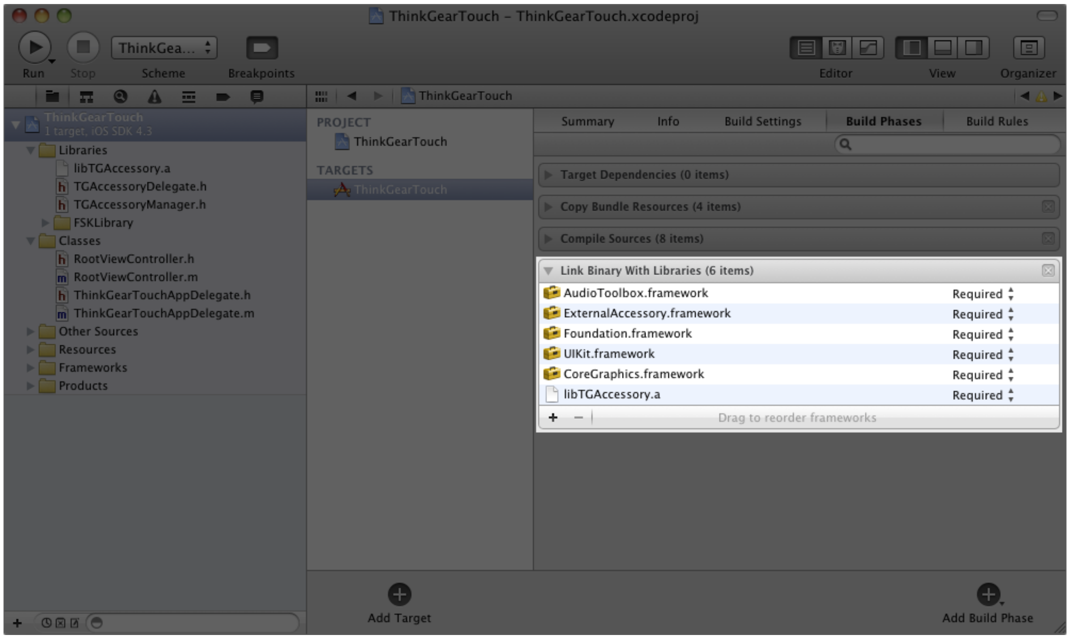


Figure 2: Add frameworks to project

Then, import the appropriate header files (`TGAccessoryManager.h` and `TGAccessoryDelegate.h`) into the requisite classes.

Setting Up the TGAccessoryManager

Setting up the `TGAccessoryManager` should be performed as early as necessary. Typically, this would be in the `applicationDidFinishLaunching:` method in the application delegate class. Simply add the following two lines to that method:

```
[[TGAccessoryManager sharedTGAccessoryManager] setupManagerWithInterval:0.05];
[[TGAccessoryManager sharedTGAccessoryManager] setDelegate:(RootViewController
*)[[navigationController viewControllers] objectAtIndex:0]];
```

This sets up the `TGAccessoryManager` instance to dispatch `dataReceived:` notifications every 0.05s, or roughly 20 times per second. The delegate can be set to any class that implements the `TGAccessoryDelegate` protocol — in this case, it's an instance of `RootViewController`.

Before the application quits, teardown of the `TGAccessoryManager` instance should be performed. This should be performed as late as necessary, typically in the `applicationWillTerminate:` method in the application delegate class. The following code should be added to that method:

```
[[TGAccessoryManager sharedTGAccessoryManager] teardownManager];
```

Handling Data Receipt

Since the delegate object was set to be a `RootViewController` instance, we have to edit its class definition to indicate support of the `TGAccessoryDelegate` protocol. In the sample project file, the class definition in `RootViewController.h` looks similar to the following:

```
@interface RootViewController : UITableViewController
```

Simply modify the definition in the following way:

```
@interface RootViewController : UITableViewController <TGAccessoryDelegate>
```

As a requisite of supporting the `TGAccessoryDelegate` protocol, the `dataRecieved:` method must be implemented. In the header (`.h`) file, add the following method definition:

```
- (void)dataReceived:(NSDictionary *)data;
```

And in the implementation (`.m`) file, implement the method. A few `NSLog` calls are provided as a trivial example of accessing the `data` parameter. Check the thinkgear_ios_api_reference for a full list of the supported keys.

```
- (void)dataReceived:(NSDictionary *)data {
    NSLog(@"Data received!");
    NSLog(@"Raw: %d", [[data valueForKey:@"raw"] intValue]);
    NSLog(@"Attention: %d", [[data valueForKey:@"eSenseAttention"] intValue]);
}
```

Handling Accesssory Connection and Disconnection

The `TGAccessoryDelegate` protocol also specifies two optional methods for the delegate object to handle accessory connection and disconnection — `accessoryDidConnect:` and `accessoryDidDisconnect`. Add the following method definitions to the header file:

```
- (void)accessoryDidConnect:(EAAccessory *)accessory;
- (void)accessoryDidDisconnect;
```

In the implementation file, implement these methods:

```
- (void)accessoryDidConnect:(EAAccessory *)accessory {
    NSLog(@"%@ was connected to this device.", [accessory name]);
}

- (void)accessoryDidDisconnect {
    NSLog(@"An accessory was disconnected.");
}
```

Starting the Data Stream

When your application is ready to receive the headset data, call the `startStream` method in `TGAccessoryManager`. In the sample project, this is done in the `viewWillAppear:` method. It is advisable to check whether an accessory was found by the `TGAccessoryManager` before starting the data stream:

```
if([[TGAccessoryManager sharedTGAccessoryManager] accessory] != nil)
    [[TGAccessoryManager sharedTGAccessoryManager] startStream];
```

You will also need a matching call to `stopStream` in the `viewWillDisappear:` method. Again, it is advisable to make sure that a data stream is connected and active before closing it:

```
if([[TGAccessoryManager sharedTGAccessoryManager] connected])
    [[TGAccessoryManager sharedTGAccessoryManager] startStream];
```

Application lifecycle

On devices that support multitasking, your application should expect the following behavior:

- Upon entering the background, `accessoryDidDisconnect:` will be called.
- Upon returning from the background, `accessoryDidConnect:` will be called.

Your application **must** restart the data stream when resuming from the background. For example:

```
- (void)accessoryDidConnect:(EAAccessory *)accessory {
    [[TGAccessoryManager sharedTGAccessoryManager] startStream];
}
```

Before your application is terminated, you **must** stop the manager if you have not done so already.

```
- (void)applicationWillTerminate:(UIApplication *)application {
    [[TGAccessoryManager sharedTGAccessoryManager] teardownManager];
}
```

Log messages

The TGAccessory library will emit some debug messages through NSLog() to help you develop and debug your application. These messages will be prefixed with "TGAccessory:" .

Further Considerations

- The application should not expect there to be a ThinkGear accessory attached to the iOS-based device on startup. As such, it should handle that case accordingly (e.g. by displaying a static splash screen prompting the user to connect a ThinkGear accessory).

Using Apple's External Accessories APIs

If you need finer-grained control over external accessory management or data stream handling, you can use Apple's External Accessories API.

Configuring Your Environment

You'll first need to add the External Accessory framework to your Xcode project. See the previous section about configuring your environment on how to add the ExternalAccessory.framework.

Enumerating Connected Accessories

Retrieving a list of the accessories currently connected to the device simply requires the following code:

```
NSArray * accessories = [[EAAccessoryManager sharedAccessoryManager] connectedAccessories];
```

You can then iterate over the resulting `NSArray` and process the accessories; to look for accessories supporting a particular protocol string, for example, you would do the following:

```
for(EAAccessory * accessory in accessories){
    if([[accessory protocolStrings] containsObject:@"com.neurosky.thinkgear"]){
        // do some stuff here
    }
}
```

Connecting to an Accessory

Once you've decided on an accessory to connect to, you must create an `EASession` instance and set up the `NSInputStream` and `NSOutputStream` instances. This requires knowledge of the specific protocol with which you use to communicate with the accessory. For example, using the `com.neurosky.thinkgear` protocol:

```
EASession * session = [[EASession alloc] initWithAccessory:accessory
                                               forProtocol:@"com.neurosky.thinkgear"];

if(session){
    // data stream from the accessory
    [[session inputStream] setDelegate:self];
    [[session inputStream] scheduleInRunLoop:[NSRunLoop currentRunLoop]
                                     forMode:NSDefaultRunLoopMode];
    [[session inputStream] open];

    // data stream to the accessory
    [[session outputStream] setDelegate:self];
    [[session outputStream] scheduleInRunLoop:[NSRunLoop currentRunLoop]
                                      forMode:NSDefaultRunLoopMode];
    [[session outputStream] open];
}
```

The delegate object for the `inputStream` and `outputStream` instances must then implement the delegate method:

```
// Handle communications from the streams.
- (void)stream:(NSStream*)theStream handleEvent:(NSStreamEvent)streamEvent {
    switch(streamEvent){
        case NSStreamHasBytesAvailable:
            // Process the incoming stream data.
            break;

        case NSStreamEventHasSpaceAvailable:
            // Send the next queued command.
            break;

        default:
            break;
    }
}
```

Register for Accessory Notifications

To receive notifications that an accessory has connected or disconnected, you must register for accessory notifications. Generally, it is recommended to do this as early in the application lifetime as possible, so this is typically done in the `applicationDidFinishLaunching:` method in the application delegate class:

```
[[EAAccessoryManager sharedAccessoryManager] registerForLocalNotifications];
```

It is also prudent to unregister for accessory notifications before the application quits. This should be done as late in the application lifetime as possible, ideally in the `applicationWillTerminate:` method.

```
[[EAAccessoryManager sharedAccessoryManager] unregisterForLocalNotifications];
```

Once you've let the system know you're interested in accessory notifications, you must write code to handle them.

Handling Connection Events

To handle events that are fired when an accessory is connected, you must register a method to handle the `EAAccessoryDidConnectNotification` event.

```
[[NSNotificationCenter defaultCenter] addObserver: self
                                         selector: @selector(accessoryConnected:)
                                             name: EAAccessoryDidConnectNotification
                                           object: nil];
```

Note that the observer of the event is a method called `accessoryConnected:` on the `self` instance; we must now create this method. This method can also include code that only handles accessories that support the `com.neurosky.thinkgear` protocol:

```
- (void)accessoryConnected:(NSNotification *)notification {
    // the accessory can be retrieved by using the EAAccessoryKey key in the userInfo dictionary
    EAAccessory * accessory = [[notification userInfo] objectForKey:@"EAAccessoryKey"];

    if([[accessory protocolStrings] containsObject:@"com.neurosky.thinkgear"]){
       // do some stuff to handle a headset connection event here
    }
}
```

Handling Disconnection Events

You must first specify a delegate class to handle the disconnection events. The delegate class must implement the `EAAccessoryDelegate` protocol, which entails adding the `accessoryDidDisconnect:` method in your delegate class:

```
- (void)accessoryDidDisconnect:(EAAccessory *)accessory {
    // do some stuff to handle the accessory disconnection event...
}
```

Disconnection event delegates are specified for each individual accessory, rather than via a global handler. We can conveniently expand the `accessoryConnected:` code written in the previous section to also assign a delegate for the accessory:

```
- (void)accessoryConnected:(NSNotification *)notification {
    // the accessory can be retrieved by using the EAAccessoryKey key in the userInfo dictionary
    EAAccessory * accessory = [[notification userInfo] objectForKey:@"EAAccessoryKey"];
    if([[accessory protocolStrings] containsObject:@"com.neurosky.thinkgear"]){
        // do some stuff to handle the headset connection event here...

        // now assign a delegate to the accessory to handle disconnection events
        [accessory setDelegate:self];
    }
}
```

This will call the `accessoryDidDisconnect:` method when that particular accessory disconnects.

References

- Communicating with External Accessories (Apple documentation)
- EAAccessoryManager Class Reference
- EAAccessory Class Reference
- EASession Class Reference

Corporate Address
NeuroSky, Inc.
125 S. Market St., Ste. 900
San Jose, CA 95113
United States
(408) 600-0129

Questions/Support: http://support.neurosky.com
or email: support@neurosky.com

Community Forum: http://developer.neurosky.com/forum

Disclaimer: The information in this document is provided in connection with NeuroSky products. No license, express or implied, by estoppels or otherwise, to any intellectual property rights is granted by this document or in connection with the sale of NeuroSky products. NeuroSky assumes no liability whatsoever and disclaims any express, implied or statutory warranty relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or non-infringement. In no even shall NeuroSky be liable for any direct, indirect, consequential, punitive, special or incidental damages (including, without limitation, damages for loss of profits, business interruption, or loss of information) arising out of the use of inability to use this document, even if NeuroSky has been advised of the possibility of such damages. NeuroSky makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. NeuroSky does not make any commitment to update the information contained herein. NeuroSky's products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

NeuroSky
Brain-Computer Interface Technologies