

BWSip Framework - Dev Guide (iOS)

- Installation
 - For Objective-C projects only
 - For Swift projects only
 - Classes
 - Main
 - Helper
 - Getting Started
 - Basic Steps
 - Tracking Phone Events
 - Authenticating in a Registrar
 - Working with Calls
 - Making a Call
 - Receiving a Call
 - Other Actions
 - Objects Lifecycle
-

Installation

1. Create a new iOS project on Xcode 6 or greater.
2. Unzip the archive file `bwsip-ios-XXXXXXX-XXXXXX.zip` and copy the `BWSip.framework` file to your project.
3. In the project editor, select the target to which you want to add a library or framework; click Build Phases at the top of the project editor; open the Link Binary With Libraries section; click the Add button (+) to add a library or framework.

▼ Link Binary With Libraries (11 items)












Name	Status
 libresolv.dylib	Required ⇅
 BWSip.framework	Required ⇅
 AudioToolbox.framework	Required ⇅
 AVFoundation.framework	Required ⇅
 CFNetwork.framework	Required ⇅
 CoreGraphics.framework	Required ⇅
 CoreLocation.framework	Required ⇅
 CoreTelephony.framework	Required ⇅
 Foundation.framework	Required ⇅
 SystemConfiguration.framework	Required ⇅
 UIKit.framework	Required ⇅
+ — Drag to reorder frameworks	

Figure 1: iOS Project Libraries

4. Add the key `NSLocationWhenInUseUsageDescription` to your project's main Plist:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string></string>
```

5. Add the following libraries to your project:

- libresolv.dylib
- BWSip.framework
- AudioToolbox.framework
- AVFoundation.framework
- CFNetwork.framework
- CoreGraphics.framework
- CoreLocation.framework
- CoreTelephony.framework
- Foundation.framework
- SystemConfiguration.framework
- UIKit.framework

For Objective-C projects only

Since the framework's classes expose the inner C++ objects for lower level access, any source files that use BWSip need to be compiled as Objective-C++. There are two approaches to do that:

- Wrap all framework usage in a single class and give it the *.mm* extension (recommended).
- Set your whole project to be compiled as Objective-C++ by going to the project's Build Settings, looking for *Apple LLVM 6.1 - Language > Compile Sources As* and setting it to *Objective-C++*. Keep in mind that this can have side effects (additional reserved words, etc.).

For Swift projects only

1. Create new header file in your project named `<PROJECT_NAME>-Bridging-Header.h`
2. Edit the new header file and add the following code:

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

3. In your project Build Settings look for *Swift Compiler - Code Generation > Objective-C Bridging Header* and add the path for the new created header file:

```
<PROJECT_NAME>-Bridging-Header.h
```

4. In your project Build Settings look for *Linking > Other Linker Flags* and add the flag: `-lstdc++`

Threading Model

BWSip will use a single internal thread for all interactions with the SIP layer. This is done by internally sending commands to a worker thread and waiting for them to complete. This is necessary to guarantee proper interoperability with the SIP stack. As a client of the library you must know that your calls to BWSip may block if another operation is in progress. You may call BWSip from any thread.

Classes

Main

The *BWSip Framework* has 3 main classes that are used to connect to a registrar and make or receive phone calls. They are:

- **BWPhone:** used to configure the main properties of your phone like the transport used, the STUN servers, set up the SRV resolution, etc.
- **BWAccount:** used to connect to an account in the SIP registrar.
- **BWCall:** used to set and get properties about the calls made/received.

Helper

The *BWSip Framework* has some helper classes that are used to work with codecs, generate tones in the device, get GPS coordinates, etc. They are:

- **BWCredentials:** used to set the user credentials used during the registration. The user can register using the username and password (`initWithUsername: andPassword:`) or the callsign token (`initWithUsername: andToken:`).
- **BWCodec:** used to list available codecs and change the codec priorities.
- **BWTone:** used to play tones in the device.
- **BWGps:** used to get GPS coordinates.
- **BWConsts:** list of constants used by the framework.

Getting Started

Basic Steps

In order to use the *BWSip Framework* to make or receive phone calls you need to instantiate the **BWPhone** class. This is the most important class in the framework. The **BWPhone** class is a singleton because there can be only one instance of this object during the entire framework lifecycle.

To create an instance of the **BWPhone** class you can use the code:

```
// Create an instance of the BWPhone class
BWPhone *phone = [BWPhone sharedInstance];
```

With the **BWPhone** object instantiated, you can now set some important properties, like:

```
// The transport type (UDP [default], TCP or TLS)
phone.transportType = BWTransportTCP;

// The log level (from 0 [no log] to 9 [the most detailed log])
phone.logLevel = 9;
```

You can find more methods and properties available for the **BWPhone** class in the API documentation.

With all properties set in the **BWPhone** object, you must initialize it by calling the method `initialize()`:

```
// Initialize the BWPhone object
[phone initialize];
```

Tracking Phone Events

The *BWSip Framework* implements the [Delegation Pattern](#) to track phone events, like successful registration, incoming phone calls, changes in the call state, receiving DTMF tones, etc.

The class that will receive these events (the delegated class) must implement the **BWAccountDelegate** and/or **BWCallDelegate** protocols and pass itself as a parameter to the property **delegate** in the **BWAccount** and/or **BWCall** objects; you also need to override the following methods based on the protocol that you implement:

BWAccountDelegate

- **onRegStateChanged:** it's triggered when the client authenticates in a registrar, but also whenever the registration status changes. For example: registration failed, registration cancelled, registration updated, etc.
- **onIncomingCall:** it's triggered when the client receives a phone call.

BWCallDelegate

- **onCallStateChanged:** it's triggered when the state of a call changes. For example, the call changed from ringing to connected, when the call disconnects, etc.
- **onIncomingDTMF:** it's triggered when the client receives a DTMF tone from the other party.

BWGpsDelegate

- **onCoordinatesUpdated:** it's triggered when the device retrieve new GPS coordinates.

Example:

@implementation ViewController

```
- (void)viewDidLoad
{
    // The ViewController class -self- will now receive events from BWAccount
    BWAccount *account = [[BWAccount alloc] initWithPhone:phone];
    account.delegate = self;

    // The ViewController class -self- will now receive events from BWCall
    BWCall *call = [[BWCall alloc] initWithAccount:account];
    call.delegate = self;

    // The ViewController class -self- will now receive events from BWGps
    BWGps *gps = [[BWGps alloc] init];
    gps.delegate = self;
}

- (void)onRegStateChanged:(BWAccount *)account
{}

- (void)onIncomingCall:(BWCall *)call
{}

- (void)onCallStateChanged:(BWCall *)call
{}

- (void)onIncomingDTMF:(BWCall *)call andDigits:(NSString *)digits
```

```
{}

- (void)onCoordinatesUpdated:(BWGps *)gps
{}

@end
```

More information about each method can be found in the API documentation.

Authenticating in a Registrar

In order to make and receive phone calls you must first authenticate in a SIP registrar. This is done through the **BWAccount** class. To use the **BWAccount** class you must instantiate it passing as a parameter the **BWPhone** object that you created previously:

```
// Creating a BWAccount object
BWAccount *account = [[BWAccount alloc] initWithPhone:phone];
```

With the **BWAccount** object created, you can now set the basic properties needed to connect to a registrar:

```
// The registrar URI
[account setRegistrar:@"registrar.com"];

// Set the user credentials using the username and password
[account setCredentials:[[BWCredentials alloc] initWithUsername:@"username" andPassword:@"password"]];

// Set the user credentials using callsign token
// [account setCredentials:[[BWCredentials alloc] initWithUsername:@"username" andToken:@"token"]];

// Attempt the connect to the registrar using the data set above
[account connect];
```

When the registration completes, either successful or as a failure, it will trigger the event **onRegStateChanged** (see [Tracking Phone Events](#)).

You can find more methods and properties available for the **BWAccount** class in the API documentation.

Working with Calls

In order to make or receive phone calls you must use the **BWCall** class.

Making a Call

To make a call you must instantiate a **BWCall** object passing as a parameter the **BWAccount** object that you are registered:

```
// Creating a BWCall object
BWCall *call = [[BWCall alloc] initWithAccount:account];
```

With the **BWCall** object created, you can now set the basic properties needed to make a call:

```
// Set the remote URI that will receive the call
[call setRemoteUri:@"username@registrar.com"];

// Make the call
[call makeCall];
```

Receiving a Call

The client will receive phone calls through the event **onIncomingCall** in the delegated class that you specified previously (see [Tracking Phone Events](#)). When the **onIncomingCall** method is called you receive a **BWCall** object as a parameter.

With the **BWCall** object that you received, you must first decide if you want to receive the incoming call and send the proper answer to the client calling you. You can send this answer using the piece of code below according to each scenario:

- If you are already on another call and to send the busy tone of the client calling you:

```
// Send the busy tone  
[call answerCall:BWSipReponseBusyHere];
```

- If you are available to receive the call you must let the client calling know that the phone is ringing:

```
// Send the ringtone  
[call answerCall:BWSipReponseRinging];
```

- And finally, to answer the call you send the “OK” signal to establish a connection:

```
// Answer the call  
[call answerCall:BWSipReponseOK];
```

Other Actions

- Inspect the headers on the incoming call INVITE:

```
// returns a NSDictionary<NSString*, NSArray<NSString*>*>* with the headers  
call.inviteHeaders;
```

- Hang-up an active call:

```
// Hang-up the call  
[call hangupCall];
```

- Mute or un-mute the call:

```
// Mute the call (the other party cannot hear you)  
[call setMute:YES];
```

```
// Un-mute the call (the other party can now hear you)  
[call setMute:NO];
```

- Put the call on hold or release the hold:

```
// Put the call on hold (you and the other party cannot hear each other)  
[call setOnHold:YES];
```

```
// Release the hold (you and the other party can now hear each other)  
[call setOnHold:NO];
```

You can find more methods and properties available for the **BWCall** class in the API documentation.

Objects Lifecycle

Keep in mind that the main classes are hierarchically related. In other words, if an object is closed then all its children will be automatically closed as well. For example, closing a **BWAccount** object automatically closes all **BWCall** objects created for that account.

In order to deallocate a BWSip object, you must call the `close()` method:

```
// Close the BWAccount object and any BWCall object that belong to that account  
[account close];  
  
// Close the BWPhone object and any BWAccount, BWCall object  
[phone close];
```