## Contents

Starting an application via SYNC AppLink ..... 3

- 1) Pairing or Connecting your Device ..... 3
  - a) MFD Displays – SYNC AppLink 1.1 ..... 3
  - b) MFD Displays – SYNC AppLink 2.0 ..... 4
  - c) CID ..... 4
  - d) NGN ..... 4
- 2) Start your application on the device ..... 5
- 3) Launching your Application ..... 5
  - Voice ..... 5
  - Haptic ..... 5
  - Successful Launch ..... 6

Getting Started on Android™ ..... 7

- Download Necessary Software ..... 7
- Configuring Eclipse ..... 7
  - For Windows Users ..... 7
  - Creating a New Project ..... 7
- Importing SyncProxy.jar ..... 7
- Modifying your AndroidManifest.xml ..... 8
- Adding AppLink™ Code ..... 9
  - Main Activity ..... 9
  - AppLinkReceiver.java ..... 11
  - AppLinkService.java ..... 13
  - LockScreenActivity.java ..... 20
- Building your Application ..... 20

Getting Started on iOS ..... 22

- Download Necessary Software ..... 22
- Configuring Xcode ..... 22
  - Creating a New Project ..... 22
- Enabling Background Capabilities ..... 23
- Adding the AppLink Framework ..... 24
- Modifying Your Target's Settings ..... 24

Build Phases ........ 25
Targets Info ........ 25
Adding AppLink™ Code ........ 26
Bringing in FMCProxyListener ........ 26
Creating Your SyncProxy ........ 26
Implementing your onProxyOpened() ........ 26
Filling in your onOnHMIStatus() ........ 27
Making a tearDownProxy() Method ........ 28
Setting up the Proxy ........ 28
Building your Application ........ 28

## Starting an application via SYNC AppLink

To start your application for use with SYNC AppLink, there are three major steps that must be taken:

1. Pair and connect your device to the vehicle head unit
2. Start the application on your device
3. Select the application with voice or the screen inside the vehicle

### 1) Pairing or Connecting your Device

If your device isn't already paired and connected to SYNC®, make sure it's Bluetooth® radio is turned on and the screen to select a device is up. Once that is complete, follow the correct steps for your head unit. Note, each head unit is different and may require slightly modified steps.

**If you have an iOS device, you must also connect the device with the system via the Apple provided USB cable. Non-Apple USB cables may not work with AppLink™ and may only allow charging of the device.**

#### a) MFD Displays – SYNC AppLink 1.1

For your MFD displays with AppLink 1.1, follow these steps:

- Press the **Phone** button
- If there are no phones paired, you will see "No phones paired"
    - The prompt will disappear and you will see "Add Bluetooth Device"
    - Press **OK**
    - You will see "Find SYNC" and press **OK** once more
    - You will see "SYNC" followed by a random Pin
- If there are phones paired, SYNC will try to connect to the various phones.
    - You can press **OK** to stop searching
    - Scroll to "Add Bluetooth Device"
    - Press **OK**
    - You will see "Find SYNC" and press **OK** once more
    - You will see "SYNC" followed by a random PIN
    - Enter the PIN if prompted on the device.
- You will receive the following prompts. It is recommended that you select "Yes" to both, as this is a typical user scenario and simplifies future testing:
    - "Set as Primary Phone?"
    - "Download Phonebook?"

## b) MFD Displays – SYNC AppLink 2.0

For your MFD displays with AppLink 2.0, follow these steps:

- Press the **Phone** button
- If there are no phones paired, you will see “No phones paired”
    - The prompt will disappear and you will see “Add Bluetooth Device”
    - Press **OK**
    - You will see “Find SYNC” and press **OK** once more
    - You will see “SYNC” followed by a random Pin
- If there are phones paired, SYNC will try to connect to the various phones.
    - You can press **OK** to stop searching
    - Scroll to “Add Bluetooth Device”
    - Press **OK**
    - You will see “Find SYNC” and press **OK** once more
    - You will see “SYNC” followed by a random PIN
    - Confirm the PIN you see on your device

## c) CID

For your CID display, follow these steps:

- Press the **Phone** button
- If there are no phones paired, you will see “No phones paired”
    - The prompt will disappear and you will see “Add Bluetooth Device”
    - Press **OK**
    - You will see “Find SYNC” and press **OK** once more
    - You will see “SYNC” followed by a random Pin
- If there are phones paired, SYNC will try to connect to the various phones.
    - You can press **OK** to stop searching
    - Scroll to “Add Bluetooth Device”
    - Press **OK**
    - You will see “Find SYNC” and press **OK** once more
    - You will see “SYNC” followed by a random PIN
    - Enter the PIN when prompted on the device

## d) NGN

For your NGN display, follow these steps:

- Press the **Phone** button
- If there are no phones paired, you will see “No phones paired”
    - Press **OK**
- If there are phones paired, SYNC will try to connect to the various phones.
    - Press **OK**

    - You will see “Find SYNC” and press **OK** once more
    - You will see “SYNC” followed by a random PIN
    - Enter the PIN when prompted on the device

## 2) Start your application on the device

For iOS applications, you must start the application on your device so that it connects to the head unit.

After installation, Android applications should be started on your device at least once to ensure the SYNC proxy service is started. After that point in time, it may not be necessary to launch the application again.

## 3) Launching your Application

Once you have an application running on your device, you can launch the application in one of two ways: Voice or Haptic (e.g. through the head unit).

### Voice

On most head units, to launch the application by voice, simply press the Push-To-Talk (PTT) button and after the chime, say “Mobile Applications”, then wait until the next prompt and say the name of the application. On newer head units, the phrase “Mobile Applications” is not needed.

### Haptic

To launch the application from the center stack, follow the correct steps for your Head Unit.

#### *a) MFD Displays – SYNC AppLink 1.1*

For your MFD display with AppLink 1.1, follow these steps:

- Press the **AUX** button
- Press the **MENU** button
- Press the **OK** button for the highlighted item **SYNC-Media**
- Press the **Arrow Down** button until you get to **Mobile Applications** and press the **OK** button.
- Your application should be listed in the menu and press the **OK** button to launch it.

#### *b) MFD Displays – SYNC AppLink 2.0*

For your MFD display with AppLink 2.0, follow these steps:

- Press the **MENU** button
- Press the **Arrow Down** button until you get to **SYNC Applications** and press the **OK** button.
- Press the **Arrow Down** button until you get to **Mobile Apps** and press the **OK** button.
- Your application should be listed in the menu and press the **OK** button to launch it.

### *c) CID*

For your CID display, follow these steps:

- Press the **AUX** button
- Press the **MENU** button
- Scroll until you get to **Mobile Applications** and press the **OK** button.
- Your application should be listed in the menu and press the **OK** button to launch it.

### *d) NGN*

For your NGN display, follow these steps:

- Press the **PHONE** button
- Press the **SYNC Apps** button
- Press the **Mobile Applications** button
- Your application should be listed in the menu and press the **OK** button to launch it.

## Successful Launch

If the application was launched successfully, you should see its persistent template on the display along with any output Show commands it created.

If you followed on of the 'Getting Started' tutorials, you will notice the Show message on the SYNC® display.

Writing to the Display

# Getting Started on Android™

In this tutorial, we exclusively use the Eclipse IDE, but you are able to choose any IDE you wish.

## Download Necessary Software

Your first step in getting started with Android™ development is to download and install the following:

- **Java JDK** - make sure you download and install the whole JDK, not just the Runtime Environment
  - http://www.oracle.com/technetwork/java/javase/downloads/
- **Eclipse IDE for Java Developers**
  - http://www.eclipse.org/downloads/
- **Android™ SDK**
  - http://developer.android.com/sdk/
- **ADT plug-in for Eclipse** - install after Eclipse is installed, and be sure to restart Eclipse after the plug-in installation
  - http://developer.android.com/sdk/eclipse-adt.html#installing
- **AppLink™ SDK** – download the .JAR file to an appropriate location on your file system.
  - [Insert link to beta 2.0 download]

## Configuring Eclipse

Launch the Android™ SDK and AVD Manager from within Eclipse by choosing **Window** > **Android™ SDK and AVD Manager**.
From the menu, choose at least the following:

- SDK Platform Android™ 2.2, API8
- Samples for SDK API8, revision 1
- USB Driver Package

### For Windows Users

Download and install additional USB drivers for the various phones you plan on developing on. Consult the support site of mobile phone manufacturers for these drivers. You can then use Windows Device Manager to browse to the Android™ device and install the driver(s) you just downloaded.

## Creating a New Project

If you already have a project, you can skip to the next section.

Open Eclipse, and choose **File** > **New** > **Android™ Project**. Enter your application's details. You may skip the "New Android™ Test Project" dialog and go directly to **Finish**.

## Importing SyncProxy.jar

Choose **Project** > **Properties** > **Java Build Path** > **Libraries**. Click **Add External Jars…** and then choose *SyncProxyAndroid.jar*.

## Modifying your AndroidManifest.xml

Go to the P**ackage Explorer** (left-hand pane) > **res** > **AndroidManifext.xml** and paste in two new permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.BLUETOOTH"/>
<!-- Required to check if WiFi is enabled -->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

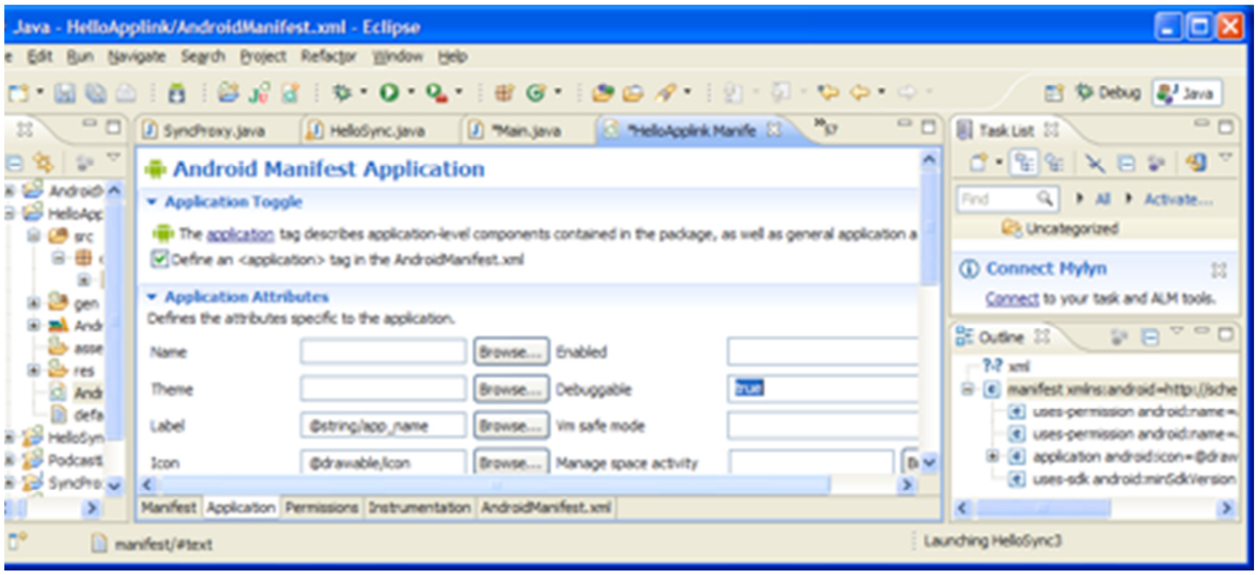
Declare your background service for the SyncProxy:

```
<service android:name=".AppLinkService"></service>
```

Declare the receiver and specify the intents to receive:

```
<receiver android:name=".AppLinkReceiver">
<intent-filter>
    <action android:name="android.bluetooth.adapter.action.STATE_CHANGED" />
    <action android:name="android.bluetooth.device.action.ACL_CONNECTED" />
    <action android:name="android.bluetooth.device.action.ACL_DISCONNECTED"/>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
    <action android:name="android.media.AUDIO_BECOMING_NOISY" />
</intent-filter>
</receiver>
```

Click on the **Application** tab and insure that **Debuggable** is set to **true**.



Android Manifest Application

Application Toggle

The application tag describes application-level components contained in the package, as well as general application a

☑ Define an <application> tag in the AndroidManifest.xml

Application Attributes

Defines the attributes specific to the application.

| | | | |
|---|---|---|---|
| Name | | Enabled | |
| Theme | | Debuggable | true |
| Label | @string/app_name | Vm safe mode | |
| Icon | @drawable/icon | Manage space activity | |

```
?-? xml
manifest xmlns:android=http://sche
    uses-permission android:name=
    uses-permission android:name=
    application android:icon=@draw
    uses-sdk android:minSdkVersion
```

Editing the AndroidManifest.xml

## Adding AppLink™ Code

The key feature of the Android™ implementation is Auto/Remote-start; this enables an application that is NOT running on the phone, i.e. the main activity has not been started by the user, to appear in SYNC®'s 'Mobile Applications' menu. This allows SYNC® to trigger the app to start upon Bluetooth connection. On Android™, this is accomplished by creating a service that instantiates SyncProxy (and re-instantiates as necessary).

A **very** important note is that an instance of SyncProxy cannot be reused after it is closed and properly disposed of. Instead, a new instance must be created. Only one instance of SyncProxy should be in use at any given time. For example, reasons to dispose and create a new proxy include: exiting the app on the phone, ignition off, bluetooth toggle, and a bluetooth disconnection from SYNC®. Implementations of all of these will be discussed below.

This auto-start functionality can be seen in the Hello AppLink Android™ sample app. There are four java files to review, plus the AndroidManifest.xml file:

- **MainActivity.java**: starts and stops AppLinkService and LockScreenActivity under certain conditions.
- **AppLinkReceiver.java**: listens for Android™ OS Broadcast intents, start and stops AppLinkService based on intents.
- **AppLinkService.java:** enables auto-start by creating a SyncProxy, which then waits for a connection from SYNC®. This file also sends and receives messages to and from SYNC® after connected.
- **LockScreenActivity.java**: throws up lock screen on phone when app is running through SYNC® while driving.

### Main Activity

#### *Start the SyncProxy*

In onCreate() in your main activity, you need to add a method to start the AppLink™ service so that the app is listening for a SYNC® connection in the case the app is installed while the phone is already connected to SYNC®. (Thus the app will not receive any Bluetooth® events). See the startSyncProxyService() method in MainActivity.java.

#### *Automatic Resets of the SyncProxy*

Reasons to automatically reset the SyncProxy:

1. main activity crashes
2. user exits the app on the phone

In both cases, it is a best practice to remove your app name from SYNC®'s menu by resetting the proxy. To a user, it makes sense that 'shutting down' the app on the phone also 'shuts it down' on SYNC® and removes it from the menu. This will also help to avoid user confusion, such as, the main activity crashing the phone, but the app still working through SYNC® from the background service.

The current proxy must be disposed and recreated in the above scenarios.

By calling proxy.resetProxy(), you will both dispose the current proxy and create a new proxy with one RPC call. It is important to create a new proxy so that if a user chooses to "Find New Apps" on SYNC®, the app will be found again. Thus, within onDestroy(), you should perform a check if the service is currently running (meaning the SyncProxy has been created). If it is running, it should reset.

See the endSyncProxyInstance() method in MainActivity.java.

### *User Reset of the SyncProxy*

You may find it valuable to add a settings option, a button on the lockscreen, or a toggle switch that will 'reset' the proxy; effectively ending the current connection, if one exists, disposing the current proxy, then creating a new one that SYNC® is able to connect to.

Note: Adding these any of these options will add some time to the validation and approval process.

In the Hello AppLink™ app, you can see an example of a 'reset sync connection' settings option. This button press first calls the endSyncProxyInstance() method, as mentioned above, to close and dispose the current proxy and create a new one. However, this method will only be effective if a current background service was already up and running (and had created a proxy). Thus, the button press also calls the startSyncProxyService() method in MainActivity.java to start a SyncProxy background service if needed.

### *Lockscreen*

The MainActivity.java example includes code to manage the lockscreen. It uses a field to do so: actvityOnTop, which keeps track of whether or not the application is the top-most activity on the UI stack on the phone to know when to throw up the lockscreen.

This field is set within the onResume() and onPause() callbacks:

```
protected void onResume() {
    activityOnTop = true;
    ...
}
protected void onPause() {
    activityOnTop = false;
    super.onPause();
}
```

A method for returning the value is needed:

```
public boolean isActivityonTop() {
    return activityOnTop;
}
```

Also, within onResume(), if the service is running and the app is currently connected to SYNC®, the lockscreen should be up on the phone:

```
protected void onResume() {
    activityOnTop = true;

    //check if lockscreen should be up
    AppLinkService serviceInstance = AppLinkService.getInstance();

    if (serviceInstance != null) {
        if (serviceInstance.getLockScreenStatus() == true) {
            if(LockScreenActivity.getInstance() == null) {
                Intent i = new Intent(this, LockScreenActivity.class);
                i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                i.addFlags(Intent.FLAG_ACTIVITY_NO_USER_ACTION);
                startActivity(i);
            }
        }
    }

    super.onResume();
}
```

## AppLinkReceiver.java

The Android™ implementation of the SyncProxy relies heavily on the OS's bluetooth intents. When the phone is connected to SYNC®, the app needs to create a SyncProxy, which publishes an SDP record for SYNC® to connect to. As mentioned previously, a SyncProxy cannot be re-used. When a disconnect between the app and SYNC® occurs, the current proxy must be disposed of and a new one created.

AppLinkReceiver.java extends a Broadcast receiver that listens for the following Android™ OS Bluetooth intents:

1. BluetoothAdapter.ACTION_STATE_CHANGED
2. BluetoothAdapter.ACTION_ACL_CONNECTED
3. BluetoothAdapter.ACTION_ACL_DISCONNECTED

Listening for ACTION_STATE_CHANGE functions as an extra check to stop service and dispose the SyncProxy when BT is turned off on the mobile device:

```
else if (intent.getAction().equals(BluetoothAdapter.ACTION_STATE_CHANGED)) {
    if ((intent.getIntExtra(BluetoothAdapter.EXTRA_STATE, -1) ==
(BluetoothAdapter.STATE_TURNING_OFF))) {
        AppLinkService serviceInstance = AppLinkService.getInstance();

        if (serviceInstance != null) {
            Intent stopIntent = new Intent(context, AppLinkService.class);
            stopIntent.putExtras(intent);
            context.stopService(stopIntent);
        }
    }
}
```

Listening for ACTION_ACL_CONNECTED triggers the app to start the background service and create a SyncProxy when the mobile device connects to SYNC®:

```
if (intent.getAction().compareTo(BluetoothDevice.ACTION_ACL_CONNECTED) == 0) {
    if(bluetoothDevice.getName() != null) {
        if (bluetoothDevice.getName().contains("SYNC")) {
            AppLinkService serviceInstance = AppLinkService.getInstance();

            if (serviceInstance == null) {
                Intent startIntent = new Intent(context, AppLinkService.class);
                startIntent.putExtras(intent);
                context.startService(startIntent);
            }
        }
    }
}
```

Listening for ACTION_ACL_DISCONNECTED triggers the app to stop the background service and dispose the current SyncProxy when the mobile device disconnects from SYNC®:

```
else if (intent.getAction().equals(BluetoothDevice.ACTION_ACL_DISCONNECTED)) {
    if (bluetoothDevice.getName().contains("SYNC")) {
        AppLinkService serviceInstance = AppLinkService.getInstance();

        if (serviceInstance != null) {
            Intent stopIntent = new Intent(context, AppLinkService.class);
            stopIntent.putExtras(intent);
            context.stopService(stopIntent);
        }
    }
}
```

Additionally, AppLinkReceiver.java also listens for the following OS intents:

1. android.media.AudioManager.ACTION_AUDIO_BECOMING_NOISY
2. Intent.ACTION_BOOT_COMPLETED

Listening for ACTION_BOOT_COMPLETED enables the app to start the background service after the phone turns on so the app appears in the menu without having to start it on the phone.

Listening for ACTION_AUDIO_BECOMING_NOISY enables a media app to stop or pause audio as quickly as possible when the phone is about to disconnect from SYNC®. This intent is received much sooner than ACL_DISCONNECTED, so it helps to avoid the transfer of audio from the vehicle speaker's to the mobile device's speakers.

## AppLinkService.java

### *Bringing in IProxyListenerALM*

Implement IProxyListener in whichever file you want to be able to send RPCs to and from SYNC®. In the HelloAppLink example, this happens in AppLinkService.java:

```
public class AppLinkService extends Service implements IProxyListenerALM {
```

Import the SyncProxyAndroid.jar objects by hovering over IProxyListenerALM and clicking on the following "quick fix":

- import 'IProxyListenerALM' (com.ford.syncv4.proxy)

Add the new unimplemented methods by hovering on AppLinkService and clicking on the following "quick fix":

- Add unimplemented methods

Class variables used in the example:

```
//variable used to increment correlation ID for every request sent to SYNC
public int autoIncCorrId = 0;
//variable to contain the current state of the service
private static AppLinkService instance = null;
//variable to contain the current state of the main UI Activity
private MainActivity currentUIActivity;
//variable to access the BluetoothAdapter
private BluetoothAdapter mBtAdapter;
```

```
//variable to create and call functions of the SyncProxy
private SyncProxyALM proxy = null;
//variable that keeps track of whether SYNC is sending driver distractions
//(older versions of SYNC will not send this notification)
private boolean driverdistrationNotif = false;
//variable to contain the current state of the lockscreen
private boolean lockscreenUP = false;
```

### *Creating your SyncProxy: startProxy()*

In the example implementation, to keep it straightforward, when the service is created, the proxy is started and when the proxy should be disposed of (without re-creation) the service is stopped. If there is an error creating the proxy, the service self-stops.

With this in mind, the SyncProxy is created in the onStartCommand() method of the service. Before creation, it first checks that the bluetooth adapter is available and that it is enabled. It returns START_STICKY since this mode is used when services will be explicitly started and stopped to run for arbitrary periods of time.

```
public int onStartCommand(Intent intent, int flags, int startId) {
    if (intent != null) {
        mBtAdapter = BluetoothAdapter.getDefaultAdapter();

        if (mBtAdapter != null) {
            if (mBtAdapter.isEnabled()) {
                startProxy();
            }
        }
    }

    if (MainActivity.getInstance() != null) {
        setCurrentActivity(MainActivity.getInstance());
    }
    return START_STICKY;
}
```

When you create the proxy, you also specify registration parameters. These parameters include the string SYNC® will use to populate the mobile apps' menu and voice command to start the app on SYNC® = "Hello AppLink" in this example. Also, this app registers as a media app, so that Boolean field is set to "true".

```
public void startProxy() {
    if (proxy == null) {
        try {
            proxy = new SyncProxyALM(this, "Hello AppLink", true);
        } catch (SyncException e) {
            e.printStackTrace();

            //error creating proxy, returned proxy = null
            if (proxy == null) {
                stopSelf();
            }
        }
    }
}
```

### *Disposing your SyncProxy: disposeSyncProxy()*

Disposing the proxy is completed by a simple call to proxy.dispose(). At this time, it is good practice to set the proxy variable to null in case any other methods are attempting to use it, as well as, clearing the lockscreen if it is currently up on the device.

```
public void disposeSyncProxy() {
    if (proxy != null) {
        try {
            proxy.dispose();
        } catch (SyncException e) {
            e.printStackTrace();
        }

        proxy = null;
        clearlockscreen();
    }
}
```

### *onDestroy()*

In case the service crashes or is shutdown by the OS, within onDestroy() of the service, you should dispose the current proxy correctly and clear the lockscreen. Otherwise, SYNC® will issue an error that it lost connection with the app if the connection fails without proper disconnection.

```
public void onDestroy() {
    disposeSyncProxy();
    clearlockscreen();
    instance = null;
    super.onDestroy();
}
```

### *onProxyClosed()*

onProxyClosed() is called whenever the proxy detects some disconnect in the connection, whether initiated by the app, by SYNC®, or by the device's Bluetooth®

connection. As long as the exception does not equal SYNC_PROXY_CYCLED or BLUETOOTH_DISABLED, the proxy would be reset for the exception SYNC_PROXY_DISPOSED.

```
public void onProxyClosed(String info, Exception e) {
    clearlockscreen();

    if((((SyncException) e).getSyncExceptionCause() !=
SyncExceptionCause.SYNC_PROXY_CYCLED)) {
        if (((SyncException) e).getSyncExceptionCause() !=
SyncExceptionCause.BLUETOOTH_DISABLED) {
            reset();
        }
    }
}
```

### *reset()*

This method makes it easier to reset the proxy from another activity or method. If the current proxy is not null, call resetProxy(), otherwise call startProxy(). Regardless, the final outcome of calling this method should be a new proxy, broadcasting an app registration that can be found by SYNC.

```
if (proxy != null) {
    try {
        proxy.resetProxy();
    } catch (SyncException e1) {
        e1.printStackTrace();

        //something goes wrong, & the proxy returns as null, stop the service.
        //do not want a running service with a null proxy
        if (proxy == null){
            stopSelf();
        }
    }
} else {
    startProxy();
}
```

### *onOnHMIStatus()*

This is where you should control your application with SYNC® various HMI Statuses. When you receive the first HMI_FULL, you should initialize your app on SYNC® by subscribing to buttons, registering addcommands, sending an initial show or speak command, etc.

This example checks that the lockscreen is up on HMI_FULL, BACKGROUND, and LIMITED per the driver distraction guidelines. However, if SYNC® is sending the onDriverDistraction notification, the lockscreen does not have to be up when the vehicle is moving under 5 mph. Older versions of SYNC® do not provide this notification, so it cannot solely be relied upon.

```
switch(notification.getAudioStreamingState()) {
    case AUDIBLE:
        //play audio if applicable
        break;
    case NOT_AUDIBLE:
        //pause/stop/mute audio if applicable
        break;
    default:
        return;
}

switch(notification.getHmiLevel()) {
    case HMI_FULL:
        if (driverdistrationNotif == false) {
            showLockScreen();
        }
        if(notification.getFirstRun()) {
            //setup app on SYNC
            //send welcome message if applicable
            try {
                proxy.show("this is the first", "show command", TextAlignment.CENTERED,
++autoIncCorrId);
            } catch (SyncException e) {
                DebugTool.logError("Failed to send Show", e);
            }

            //send addcommands

            //subscribe to buttons
            subButtons();

            if (MainActivity.getInstance() != null) {
                setCurrentActivity(MainActivity.getInstance());
            } else {
                try {
                    proxy.show("SyncProxy is", "Alive", TextAlignment.CENTERED,
++autoIncCorrId);
                } catch (SyncException e) {
                    DebugTool.logError("Failed to send Show", e);
                }
            }
        break;
    case HMI_LIMITED:
        if (driverdistrationNotif == false) {
            showLockScreen();
        }
        break;
    case HMI_BACKGROUND:
        if (driverdistrationNotif == false) {
            showLockScreen();
        }
        break;
    case HMI_NONE:
        driverdistrationNotif = false;
        clearlockscreen();
        break;
    default:
        return;
}
```

### *showLockScreen()*

This method is used to ensure the lockscreen is up. The lockscreen activity is only launched if the main app activity is currently the top application to avoid the lockscreen popping-up as the top activity over other applications. It is up to the app developer to decide whether or not to follow this model. An alternative suggestion is to only make the lock screen the top activity when the first HMI_FULL is received.

```
public void showLockScreen() {
    //only throw up lockscreen if main activity is currently on top
    //else, wait until onResume() to throw lockscreen so it doesn't
    //pop-up while a user is using another app on the phone
    if(currentUIActivity != null) {
        if(currentUIActivity.isActivityonTop() == true) {
            if(LockScreenActivity.getInstance() == null) {
                Intent i = new Intent(this, LockScreenActivity.class);
                i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                i.addFlags(Intent.FLAG_ACTIVITY_NO_USER_ACTION);
                startActivity(i);
            }
        }
    }
    lockscreenUP = true;
}
```

### *clearLockScreen()*

Shutdown the lockscreen activity:

```
private void clearlockscreen() {
    if(LockScreenActivity.getInstance() != null) {
        LockScreenActivity.getInstance().exit();
    }

    lockscreenUP = false;
}
```

### *Subscribe to Buttons*

You must subscribe to buttons in order to receive notifications that one was pressed:

```
public void subButtons() {
    try {
        proxy.subscribeButton(ButtonName.OK, autoIncCorrId++);
        proxy.subscribeButton(ButtonName.SEEKLEFT, autoIncCorrId++);
        proxy.subscribeButton(ButtonName.SEEKRIGHT, autoIncCorrId++);
        proxy.subscribeButton(ButtonName.TUNEUP, autoIncCorrId++);
        proxy.subscribeButton(ButtonName.TUNEDOWN, autoIncCorrId++);
        proxy.subscribeButton(ButtonName.PRESET_1, autoIncCorrId++);
        ...etc...
    } catch (SyncException e) {
    }
}
```

### *onOnDriverDistraction*

This notification is not available in older versions of SYNC®, so it cannot be solely relied upon. However, if an app desires to clear the lockscreen when the vehicle is moving under 5 mph (such as when parked), it can respond to this notification.

```
public void onOnDriverDistraction(OnDriverDistraction notification) {
    driverdistrationNotif = true;

    if (notification.getState() == DriverDistractionState.DD_OFF) {
        Log.i(TAG,"clear lock, DD_OFF");
        clearlockscreen();
    } else {
        Log.i(TAG,"show lockscreen, DD_ON");
        showLockScreen();
    }
}
```

### *onBind()*

Finally, since this service is not bound to any activity, return null in the required onBind() method.

```
@Override
public IBinder onBind(Intent intent) {
    return null;
}
```

### LockScreenActivity.java

If you include button such as "Reset SYNC® Connection" or "Disconnect," this file offers an example.

When the button is pressed, you should revert back to the main activity, reset the SyncProxy, and exit the lockscreen activity to clear it.

The example sends an Intent to start the Main activity:

```
if(MainActivity.getInstance() == null) {
    Intent i = new Intent(getBaseContext(), MainActivity.class);
    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    getApplication().startActivity(i);
}
```

Then it requests a proxy reset:

```
AppLinkService serviceInstance = AppLinkService.getInstance();
if (serviceInstance != null) {
    SyncProxyALM proxyInstance = serviceInstance.getProxy();
    if(proxyInstance != null) {
        serviceInstance.reset();
    } else {
        serviceInstance.startProxy();
    }
}
```

## Building your Application

You will need to launch your application through Eclipse by selecting **Run** > **Debug** > **Android™ Application**.

Once the application is running on the device, you will want to launch the application on SYNC®. Follow the Launching an AppLink™ Application tutorial if you are unfamiliar with SYNC® HMI.

# Getting Started on iOS

## Download Necessary Software

Your first step is to download and install the following:

- **Xcode** – make sure you download and install the latest update
  - https://developer.apple.com/xcode/index.php
- **AppLink™ 2.0 SDK** – download the Framework to an appropriate location on your file system.
  - [insert link to beta 2.0 download]

## Configuring Xcode

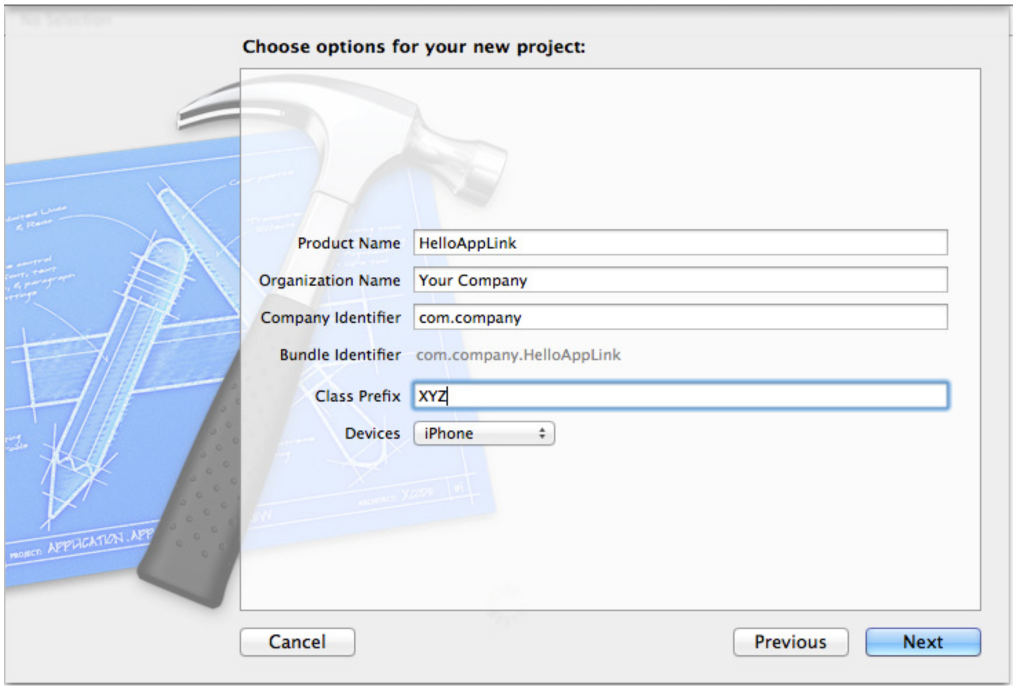
To run an application on an iOS device, you'll have to first set up your certificates, devices, provisioning profiles, etc.

For detailed information on doing this, visit the following Apple documentation: https://developer.apple.com/Library/ios/referencelibrary/GettingStarted/RoadMapiOS/chapters/RM_DevelopingForAppStore/DevelopingForAppStore/DevelopingForAppStore.html

## Creating a New Project

If you already have a project, you can skip to the next section.

Open Xcode, and choose the **Single View Application** template. Enter your application's details. Change the device family to **iPhone**. Click **Next**, and then **Create**.



Choose options for your new project:

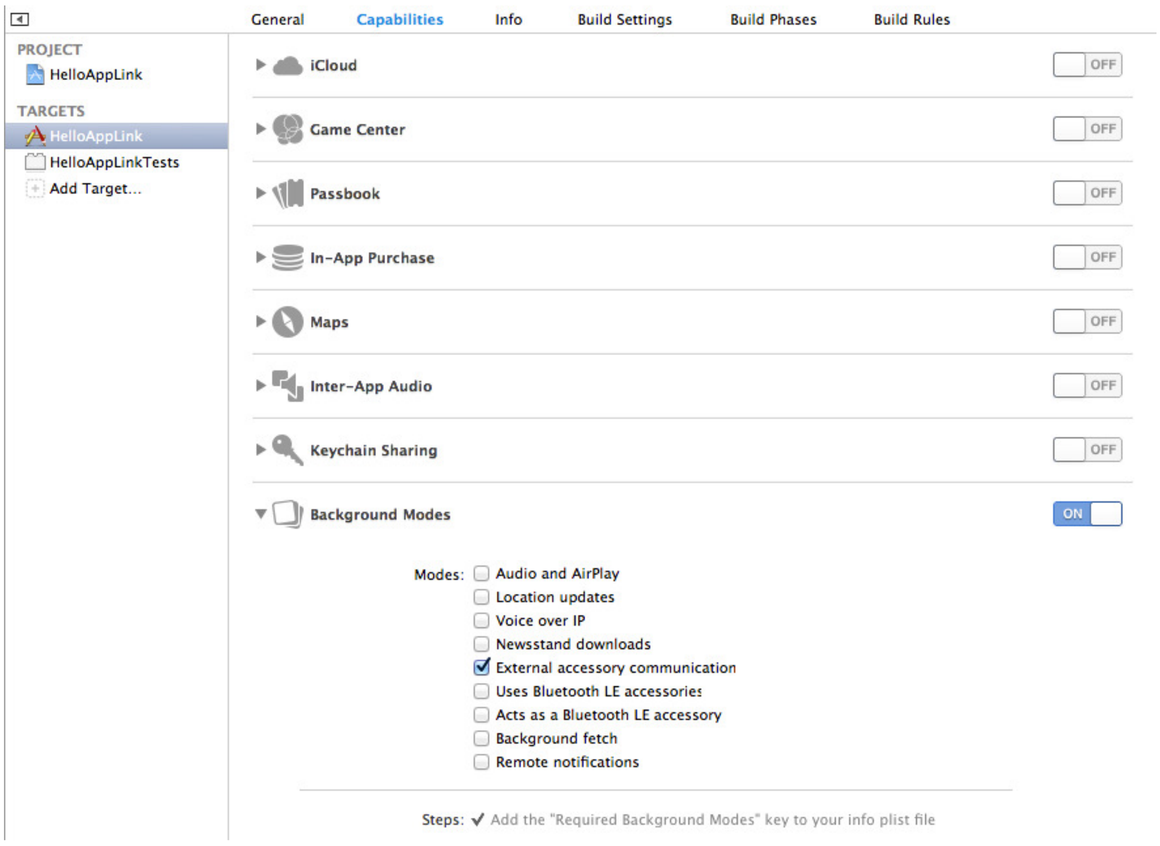
Product Name: HelloAppLink
Organization Name: Your Company
Company Identifier: com.company
Bundle Identifier: com.company.HelloAppLink
Class Prefix: XYZ
Devices: iPhone

## Enabling Background Capabilities

iOS 5 introduced the capability for an iOS application to maintain a connection to an external accessory while the application is in the background. This must be enabled manually and will only affect phones running iOS5+.

This can be enabled using one of two methods:

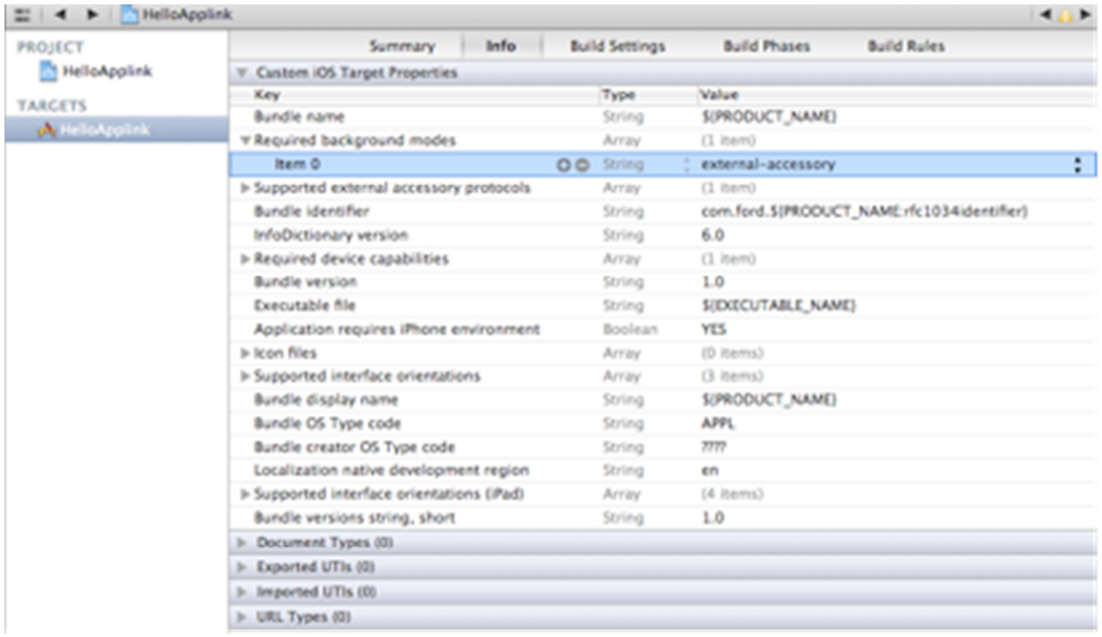
Method 1: Setting app Capabilities in build target settings
Click on your project in the Project Navigator, then select your app's build target. Click on the Capabilities tab and enable "Background Modes" and "External accessory communication":



Modes:
- [ ] Audio and AirPlay
- [ ] Location updates
- [ ] Voice over IP
- [ ] Newsstand downloads
- [✓] External accessory communication
- [ ] Uses Bluetooth LE accessories
- [ ] Acts as a Bluetooth LE accessory
- [ ] Background fetch
- [ ] Remote notifications

Steps: ✓ Add the "Required Background Modes" key to your info plist file

Method 2: Directly modifying the .plist file
Navigate to your target's Info (or the .plist file) and add a row to the Custom iOS Target Properties section titled "Required background modes". Add an item to this array with the value "external-accessory":



Custom iOS Target Properties

| Key | Type | Value |
|---|---|---|
| Bundle name | String | $(PRODUCT_NAME) |
| Required background modes | Array | (1 item) |
| Item 0 | String | external-accessory |
| Supported external accessory protocols | Array | (1 item) |
| Bundle identifier | String | com.ford.$(PRODUCT_NAME:rfc1034identifier) |
| InfoDictionary version | String | 6.0 |
| Required device capabilities | Array | (1 item) |
| Bundle version | String | 1.0 |
| Executable file | String | $(EXECUTABLE_NAME) |
| Application requires iPhone environment | Boolean | YES |
| Icon files | Array | (0 items) |
| Supported interface orientations | Array | (3 items) |
| Bundle display name | String | $(PRODUCT_NAME) |
| Bundle OS Type code | String | APPL |
| Bundle creator OS Type code | String | ???? |
| Localization native development region | String | en |
| Supported interface orientations (iPad) | Array | (4 items) |
| Bundle versions string, short | String | 1.0 |

Document Types (0)
Exported UTIs (0)
Imported UTIs (0)
URL Types (0)

Adding iOS 5+ Capabilities

## Adding the AppLink Framework

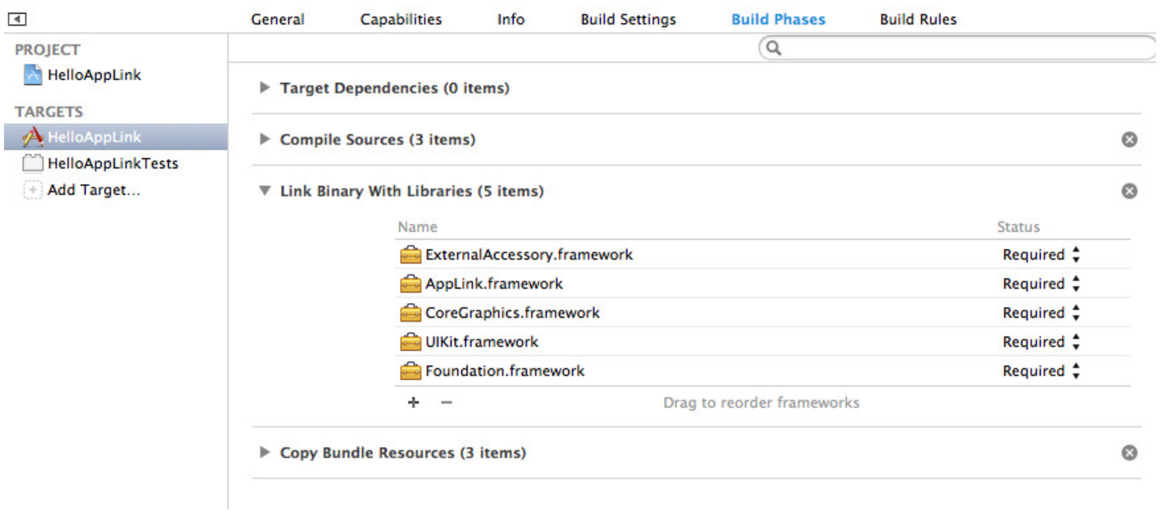
Navigate to your project folder in Finder. Copy the AppLink Framework to this location, or create a symbolic link from the framework location to your project folder.

Add the AppLink framework to the Linked Frameworks and Libraries section of the target by clicking the Add (+) button, then the “Add Other…” button and then select “Applink.framework”

## Modifying Your Target's Settings

## Build Phases

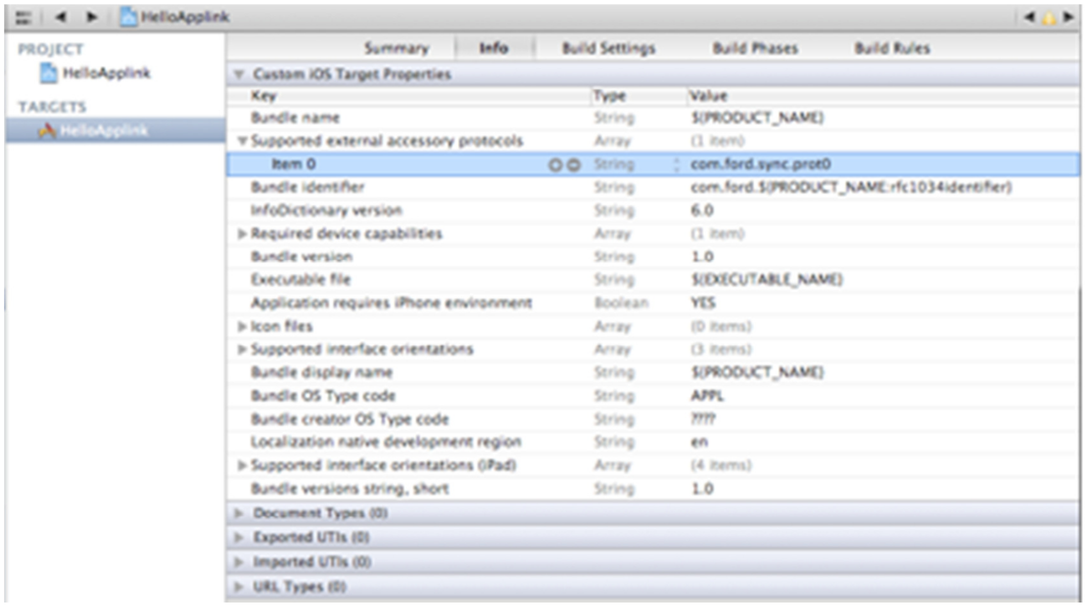
Add "ExternalAccessory.framework" under the "Linked Frameworks and Libraries" section:


General
Capabilities
Info
Build Settings
Build Phases
Build Rules
PROJECT
HelloAppLink
TARGETS
HelloAppLink
HelloAppLinkTests
Add Target...
Target Dependencies (0 items)
Compile Sources (3 items)
Link Binary With Libraries (5 items)
Drag to reorder frameworks
Copy Bundle Resources (3 items)

| Name | Status |
|---|---|
| ExternalAccessory.framework | Required |
| AppLink.framework | Required |
| CoreGraphics.framework | Required |
| UIKit.framework | Required |
| Foundation.framework | Required |

## Targets Info

The SYNC® Protocol String is required in an AppLink™-enabled application's .plist file. Add a row titled "Supported external accessory protocols". Add an item to this array with the value "com.ford.sync.prot0":



| Key | Type | Value |
|---|---|---|
| Bundle name | String | $(PRODUCT_NAME) |
| Supported external accessory protocols | Array | (1 item) |
| Item 0 | String | com.ford.sync.prot0 |
| Bundle identifier | String | com.ford.$(PRODUCT_NAME:rfc1034identifier) |
| InfoDictionary version | String | 6.0 |
| Required device capabilities | Array | (1 item) |
| Bundle version | String | 1.0 |
| Executable file | String | $(EXECUTABLE_NAME) |
| Application requires iPhone environment | Boolean | YES |
| Icon files | Array | (0 items) |
| Supported interface orientations | Array | (3 items) |
| Bundle display name | String | $(PRODUCT_NAME) |
| Bundle OS Type code | String | APPL |
| Bundle creator OS Type code | String | ???? |
| Localization native development region | String | en |
| Supported interface orientations (iPad) | Array | (4 items) |
| Bundle versions string, short | String | 1.0 |

Adding external accessory protocol

## Adding AppLink™ Code

Ideally, all of the AppLink related code would be placed in its own class. However, for the sake of simplicity in sample code it may be located in a ViewController.

### Bringing in FMCProxyListener

Add the following import where you will create the connection to AppLink:

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

Add the class variables to store the Proxy such as:

```
@property (strong, nonatomic) FMCSyncProxy* proxy;
@property (assign, nonatomic) int autoIncCorrID;
@property (assign, nonatomic) BOOL syncInitialized;
```

Add the stubs for the required FMCProxyListener's methods:

```
-(void) onOnHMIStatus:(FMOnHMIStatus*) notification {
}
-(void) onOnDriverDistraction:(FMOnDriverDistraction*)notification {
}
-(void) onProxyClosed {
}
-(void) onProxyOpened {
}
```

### Creating Your SyncProxy

Next, you will create a function setupProxy():

```
-(void) setupProxy {
        self.proxy = [FMCSyncProxyFactory buildSyncProxyWithListener: self];
        self.autoIncCorrID = 101;
        [FMCDebugTool logInfo:@"Created SyncProxy instance"];
}
```

### Implementing your onProxyOpened()

Implement the FMCProxyListener callbacks OnProxyOpened(), onError(), and onProxyClosed():

```
-(void) onProxyOpened {
	FMCRegisterAppInterface* regRequest = [FMCRPCRequestFactory
buildRegisterAppInterfaceWithAppName:@"Hello iPhone"
languageDesired:[FMCLanguage EN_US] appID:@"1234"];
	regRequest.isMediaApplication = [NSNumber numberWithBool:YES];
	regRequest.ngnMediaScreenAppName = nil;
	regRequest.vrSynonyms = nil;
	[consoleController appendMessage:regRequest];
	[self.proxy sendRPCRequest:regRequest];
}
-(void) onError:(NSException*) e {
	[FMCDebugTool logInfo:@"proxy error occurred: %@", e];
}
-(void) onProxyClosed {
	[FMCDebugTool logInfo:@"onProxyClosed"];
	[self tearDownProxy];
	[self setupProxy];
}
```

AppID is assigned to you through this developer site, and if you have not requested one yet, do so now through the following link:
[Insert Link]

Your apps will not work on Sync with out a valid AppName and AppID

## Filling in your onOnHMIStatus()

By demonstrating a response to the onOnHMIStatus() message, we'll have shown that the application is correctly attached to AppLink and then you can begin to add real functionality.

```
-(void) onOnHMIStatus:(FMCOnHMIStatus*) notification {
	if (notification.hmiLevel == FMCHMILevel.HMI_NONE ) {
		// TODO:
	} else if (notification.hmiLevel == FMCHMILevel.HMI_FULL ) {
		[FMCDebugTool logInfo:@"HMI_FULL"];
		if (self.syncInitialized)
			return;
		self.syncInitialized = YES;
		FMCShow* msg = [FMCRPCRequestFactory
buildShowWithMainField1:@"Hi" mainField2:@"Mom!" alignment:[FMCTextAlignment
LEFT_ALIGNED] correlationID:[NSNumber numberWithInt:self.autoIncCorrID++]];
		[self.proxy sendRPCRequest:msg];
	} else if (notification.hmiLevel == FMCHMILevel.HMI_BACKGROUND ) {
        // TODO:
    } else if (notification.hmiLevel == FMCHMILevel.HMI_LIMITED ) {
        // TODO:
    }
}
```

## Making a tearDownProxy() Method

You'll also want a corresponding teardown function:

```
-(void) tearDownProxy {
    [FMCDebugTool logInfo:@"Disposing proxy"];
    [self.proxy dispose]; // required for both ARC and non-ARC
    //[self.proxy release]; // for non-ARC
    self.proxy = nil;
}
```

## Setting up the Proxy

In your view controller or AppDelegate, create your proxy controller and call the setupProxy method:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    if (!self.proxyController)
    {
        self.proxyController = [[XYZProxyController alloc] init];
    }
    [self.proxyController setupProxy];
}
```

Note: if you did not create a separate class for the proxy connection and put everything in a ViewController, the code above should instead contain:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self setupProxy];
}
```

## Building your Application

Make sure that your device is set to an iOS device and not simulator.

Launch the application through XCode by clicking the "Play" button.

Once the application is running on the device, you will want to launch the application on SYNC®. Follow the Launching an AppLink™ Application tutorial if you are unfamiliar with SYNC® HMI.

## Don’t forget to follow iOS developer guidelines

As an iOS developer you are also required to comply with all iOS developer program guidelines before submitting your application to Apple for review. There are iOS stipulations that you have to follow and it is up you as a developer to check the current Apple documentation and verify what guidelines are required. Understand that you are connecting to an external accessory over a USB connection and you need to follow iOS guidelines for connecting to an external device or accessory. Please review the iOS Developer Library to ensure you are meeting these requirements.