Contents

Contents

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*. Make sure you add the most recent jar file. . . *AppLinkSDKAndroid-2-3.jar.*

If you meet an exception "java.lang.NoClassDefFoundError: com.ford.helloapplink.applink.AppLinkService". Do this in eclipse: go to Project -> Properties -> Java Build Path -> Order and Export, check the box "AppLinkSDKAndroid-2-3.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"
/>
```

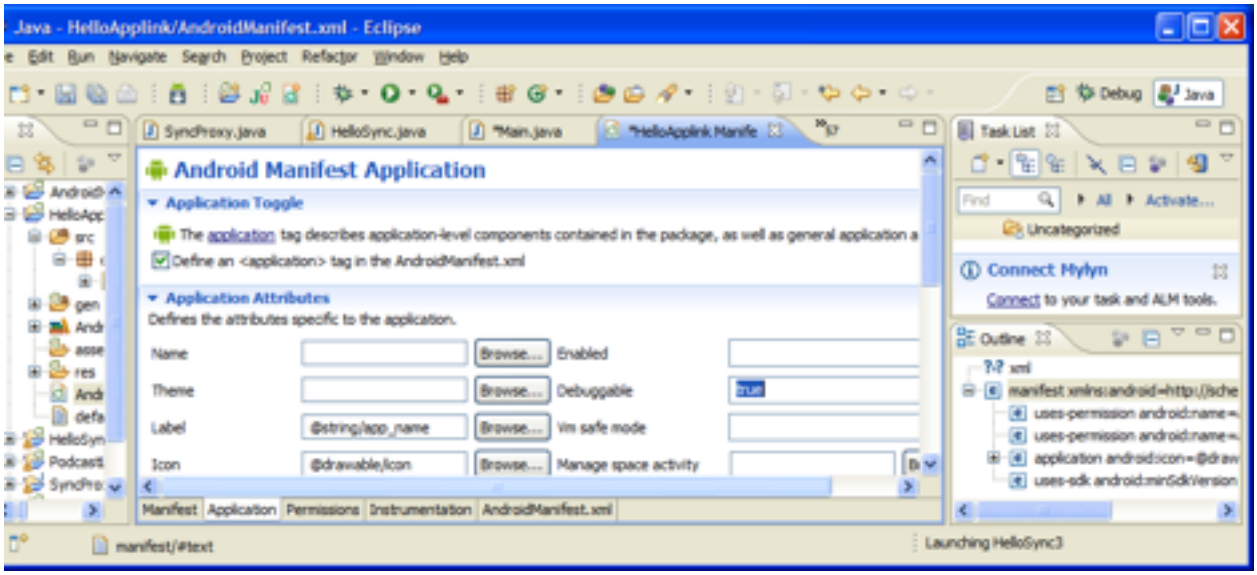
Declare your background service for the SyncProxy:

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

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



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.

Auto Start/Stop Updates for Android

The application is expected to detect a Bluetooth connection to SYNC®, and it should start/stop the AppLink service on Bluetooth connection/disconnection.

Starting with SYNC Gen3 the Bluetooth 'Name Check' for 'SYNC' should no longer be performed when starting the AppLink service. It has been removed to accommodate differences between vehicles. SYNC Gen3 vehicles will have a name that equals the vehicle name. E.g., instead of the name "SYNC", the vehicle may use "Ford Mustang" or "Lincoln MKZ." The name may also vary to accommodate the implementation needs of multiple manufacturers that use the open source Smart Device Link (SDL). Since you cannot predict every possible name, the name check has been removed altogether.

It is not recommended that an application start the AppLink service every time the phone boots up. For this reason the AppLink service will be started on any Bluetooth connection, regardless of the device name. 60 seconds later the service checks to see if the AppLink Proxy actually made a connection to SYNC ® (or SDL). If it has not connected it stops the service.
This saves resources and avoids having the AppLink service running when there is no chance of connecting to anything.

There is a race condition issue in the current version of the AppLink SDK for Android that can occur if the application attempts to shut down the AppLink service on Bluetooth disconnect. The AppLink proxy will also attempt to recycle the connection at the same time. As a temporary solution, when an application detects a Bluetooth disconnect it waits five seconds before trying to shut down the AppLink service. After the five seconds it double checks to make sure the Bluetooth connection has not been reestablished during that delay time. This is to avoid shutting everything down in case of a brief disconnection.