

# How to Program an Android Application to Access and Manage the Wi-Fi Capabilities of a Smartphone

---

## Application Note

**Matt Gottshall**

**4/1/2011**

This application note is designed to teach the reader how to set up and manage Wi-Fi network connections within an Android application. The first step is getting the development environment ready and given the program the appropriate permissions to change hardware state. Next the application needs to be able to scan for access points within the range of the device and capture the results of that scan. Now the application can “choose” to which network it should connect and to which access point. This chosen network can be access immediately and/or saved in the device’s saved configured networks list and accessed later. This application note will also give tips on useful programming practices related to the shown code and how to program the application to undo any changes it made during its execution if necessary.

## Contents

Key Terms.....	2
Introduction .....	2
Objectives.....	2
Background .....	3
Tools Required to Develop an Android Application.....	3
Android OS Versions .....	3
Getting Started.....	3
Setting the Permissions.....	3
Required and Recommended Libraries.....	4
The Android Wi-Fi Library .....	4
Secondary Required Libraries .....	4
Recommended Libraries .....	4
Turning On/Off the Device’s Wi-Fi.....	4
Scanning for all APs in Range .....	5
Connecting to a Network .....	6
Removing Saved Network Configurations .....	8
Conclusions .....	9
Side Notes and Recommendations .....	9
References .....	10

## Key Terms

API – Application Programming Interface: An interface between one program to another where the first program generally provides tools to assist in the programming of the second.

(W)AP – (Wireless) Access Point: A device that allows the user to connect wirelessly to a network.

App – Short for application.

BSSID – Basic Service Set IDentifier: This is another name for a MAC address.

dBm or dBmW – deciBel milli-Watts: a signal strength rating in decibels as a power rating compared to 1 milli-Watt

IDE – Integrated Development Environment: a program designed to assist a programmer to make programming and executing applications easier.

MAC address – Media Access Address: The physical address of a network device.

SDK – Software Development Kit: a set of tools that assist in the development of an application. Note: This application note is based on the Android ADK revision 10.

SSID – Service Set IDentifier: A name given to a network to describe it.

Wi-Fi – Wireless Fidelity: The protocol most frequently used for wide local area networks.

## Introduction

Smartphones are quickly becoming common personal devices for many people around the world. These devices are able to assist people in many of their everyday tasks including work, gaming, and worldwide internet communication. Since the internet is a part of all of these tasks it is important to be able to develop applications on smartphones that can access the internet for both storage and communication. Before one can consider how to have the application communicate with the internet, he or she must know how to access and use the Wi-Fi capabilities of the phone that will be running the application. Android devices are becoming more and more popular and the development tools for the Android OS are free. As such, Android is a good system for learning how to create smartphone applications as well as for creating applications to distribute. Android also has very good documentation on all of the APIs at <http://developer.android.com/index.html>.

## Objectives

The purpose of this application note is to teach the reader how to develop an Android application that can make use of a smartphone's Wi-Fi capabilities. This includes how to scan for networks and access points, what information can be obtained from these scans, how that information can be used, and how to connect to a wireless network all within an application. All of this will be explained using the Java code examples that have been tested on an Android smartphone.

# Background

## Tools Required to Develop an Android Application

In order to start developing any application for android one must get the Android SDK. This can be downloaded for free from <http://developer.android.com/sdk/index.html> for Windows, Mac, and Linux. This application note is based on the Android SDK revision 10. It is also important to get an IDE to make development easier. The recommended IDE for Android development is the Eclipse IDE which can be downloaded for free from <http://www.eclipse.org/downloads/>. This is the most recommended IDE since the native programming language for Android is Java, and Eclipse is written in Java and designed specifically for Java programming. There is also a required ADT plugin for Eclipse that will incorporate the SDK into the Eclipse IDE. It is best that the ADT and SDK are the same and most updated version. If the ADT is an older version than the SDK then Eclipse will not work for developing Android apps. All the instructions for getting the development environment ready can be found at <http://developer.android.com/sdk/eclipse-adt.html#installing>.

It is important to note that while the SDK can run a virtual Android system simulation to test applications, in order to test any applications that access sensors or scanning hardware it is necessary to have an actual Android device. This means that testing applications scanning and managing Wi-Fi connections must be done on a physical Android device.

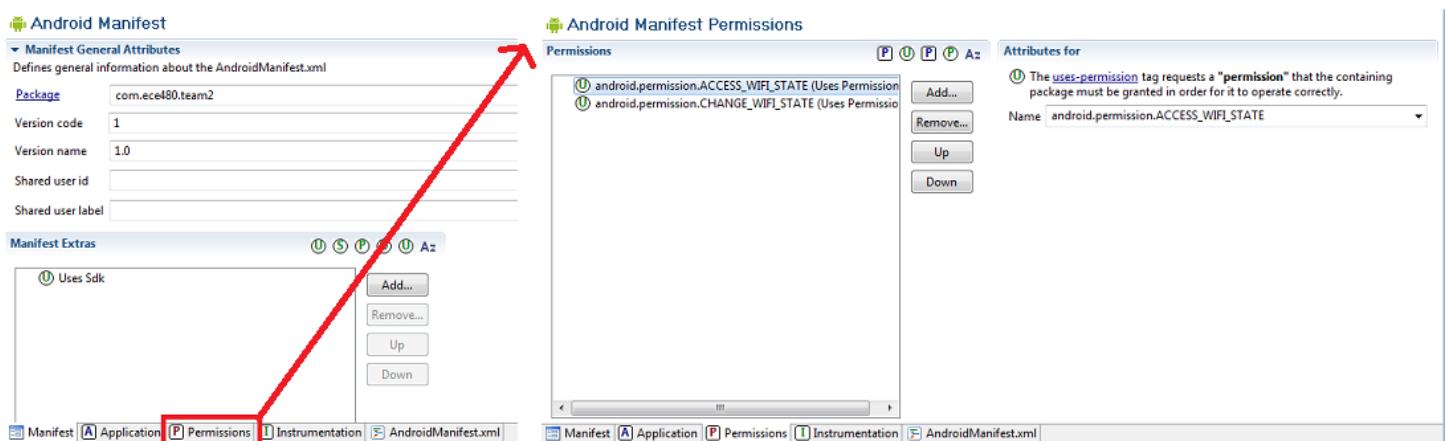
## Android OS Versions

There are several versions of the Android out today. The newest is version 3.0, which is specifically designed for tablet devices. Most phones are on the previous version: 2.3. These newer versions add new functionality to control new hardware devices like gyroscopes. This application note uses the API targeting Android version 2.1-update 1. This is an older version of Android, but the necessary Wi-Fi management code is the same for the newer versions, so this should not be an issue.

## Getting Started

### Setting the Permissions

The first thing that needs to be done in order to program an app to use a device's Wi-Fi hardware is to give that application permission to do so. Open the AndroidManifest.xml file in the main directory of the project and select the Permissions tab.



In the permissions window press the Add button. Choose “Uses Permission” from the list and press next. There will now be a drop box to select the specific permission desired. The four most important ones for managing the Wi-Fi are android.permission.ACCESS\_WIFI\_STATE, android.permission.CHANGE\_WIFI\_STATE,

android.permission.ACCESS\_NETWORK\_STATE, and android.permission.CHANGE\_NETWORK\_STATE. Select one of those four and then repeat the process to add the rest.

## Required and Recommended Libraries

### The Android Wi-Fi Library

Next the correct libraries need to be imported into the program. The main library that is required for Wi-Fi management is the wifi library. Add `import android.net.wifi.*;` to the top of the java file of the class that will be doing the Wi-Fi management. This will import the classes WifiManager, ScanResult, WifiConfiguration, and WifiInfo. If any of these classes are not needed for the app then just the specific classes that are needed can be imported individually. This is done by replacing the \* with the specific class desired. The WifiManager class is the main API for managing Wi-Fi connections. The ScanResult class holds information on a scanned access point like MAC address, SSID, and signal strength. WifiConfiguration holds information about a Wi-Fi network that has been configured for the device including MAC address, SSID, and any security information needed to access that network. WifiInfo holds information on the network to which the device is currently connected, or the network to which it is connecting.

Full documentation for this library can be found at <http://developer.android.com/reference/android/net/wifi/package-summary.html>.

### Secondary Required Libraries

In order to get an instance of the device's WifiManager the program will need to import the Context class with line `import android.content.Context;`.

Scanned APs in range are returned to the programmer in a List so the List class will need to be imported using `import java.util.List;`.

### Recommended Libraries

The TimerTask class is very useful because it allows the programmer to run a task or tasks periodically at a set period. This is imported using `import java.util.TimerTask;`. For Wi-Fi management this could mean checking for AP scan results at a programmer designed interval. The TimerTask class itself is abstract and must be extended by a new class that the programmer creates. To use a TimerTask the programmer must also include the Timer class using `import java.util.Timer;`. The TimerTask will be passed to the Timer to be run at whatever interval is desired. The Timer is executed in a new thread which means that the task can be run without affecting the main program or display. Documentation for the TimerTask class can be found at <http://developer.android.com/reference/java/util/TimerTask.html> which also directs the reader to the documentation for a Timer.

A common useful class is the KeyEvent class imported with `import android.view.KeyEvent;`. This class is sent as an argument to an app's onKeyDown method that is used to handle user input to the device via key presses. It contains the information associated with the event. The keys included in this type of event are keyboard keys as well as the standard Menu, Home, and Back keys.

## Turning On/Off the Device's Wi-Fi

An app taking advantage of a device's Wi-Fi capabilities may need to turn on the device's Wi-Fi as not all users have it activated constantly to conserve battery power. It is also good a programming practice to have the app return the device to whatever state it was in before it ran. This means turning Wi-Fi off as the app exits if it turned Wi-Fi on during its running. These tasks are accomplished using the WifiManager class. Here is sample code to get a WifiManager for the device and activate the device's Wi-Fi:

```

WifiManager myWifiManager = (WifiManager) getSystemService(Context.WIFI_SERVICE);
boolean wasEnabled = myWifiManager.isWifiEnabled();
myWifiManager.setWifiEnabled(true);
while(!myWifiManager.isWifiEnabled()){

```

This code will most likely go in the necessary class' onCreate method or constructor. The first line "asks" the system for an instance of the WifiManager for the device. The purpose of saving the initial state of the Wi-Fi is so that the app can later return the Wi-Fi settings to their previous state. The wasEnabled and myWifiManager variables will most likely need to be member variables since the Wi-Fi state will likely need to be restored in another method later in the code. Lastly, it may take several seconds to turn on the system's Wi-Fi so the programmer must be sure that it is activated before trying to perform any actions on or with it. It is a good idea to do something worthwhile during this time, but this sample code just has an empty while loop.

Turning off the Wi-Fi will most likely need to be done when the app closes. Generally for Android apps the back button ends an app's execution. Sample code for handling how to end when the back button is pressed is provided. If the Wi-Fi needs to be deactivated at another time, most of the code will be the same.

```

public boolean onKeyDown(int keyCode, KeyEvent event){
    if(keyCode == KeyEvent.KEYCODE_BACK)
    {
        myWifiManager.setWifiEnabled(wasEnabled);
        this.finish();
    }

    return super.onKeyDown(keyCode, event);
}

```

This is simple code that checks if the key pressed was the back key and if it was then return the Wi-Fi state to its prior state and finish the app.

## Scanning for all APs in Range

Scanning for APs is done periodically by the OS, but it is also possible to initiate a scan manually.

```

if(myWifiManager.isWifiEnabled()){
    if(myWifiManager.startScan()){
        // List available APs
        List<ScanResult> scans = myWifiManager.getScanResults();
        if(scans != null && !scans.isEmpty()){
            for (ScanResult scan : scans) {
                int level = WifiManager.calculateSignalLevel(scan.level, 20);
                //Other code
            }
        }
    }
}

```

Unless the programmer is absolutely sure that Wi-Fi will be enabled when the code is run it is important to wrap the code with that first if statement to make sure it is enabled. Next a scan is initiated. If the scan was successfully started then the startScan method will return true. It may take some time before the list of APs is ready so the getScanResults method may return null and will need to be called again later for the results. If this scanning is done on a timer then it doesn't matter too much if the results are not ready yet, they will just be picked up on the next cycle. If scanning is not being done on a timer, or if the results should be processed as soon as they are ready then an event is

posted when the results are ready. Example code for this method was taken from <http://stackoverflow.com/questions/2981751/android-scan-for-wifi-networks> and edited for this application note.

In the onCreate method or constructor for the class running the Wi-Fi management code register a receiver for the event sent when the results are ready. A few new classes will need to be imported to get this to work:

```
import android.content.BroadcastReceiver;
import android.content.IntentFilter;
import android.content.Intent;

IntentFilter i = new IntentFilter();
i.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context c, Intent i) {
        // Code to execute when SCAN_RESULTS_AVAILABLE_ACTION event occurs
        myWifiManager = (WifiManager) c.getSystemService(Context.WIFI_SERVICE);
        List<ScanResult> wireless = myWifiManager.getScanResults(); // Returns a <list> of
scanResults
    }
}, i);
```

This code creates a new BroadcastReceiver and registers that receiver with the message that is posted by the WifiManager. This means that when the WifiManager posts that event this receiver will capture that event and run the code given. Using this code reading and managing the scan results list will be done in the receiver code.

Note: The Android OS will automatically initiate scans periodically. startScan is only necessary when additional scans are required, or a programmer specified frequency is required.

The calculateSignalLevel method takes the given level which is the AP's signal strength in dBm and returns a value from 1 to the second parameter (in this case 20) as a rating for that AP's signal strength. The stronger the signal the higher the level returned will be.

## Connecting to a Network

There are a lot of settings when connecting to a network that are network specific. The code sample on the next page has all options activated to show what is available. Only the settings that match the desired network are necessary to activate. The lines that are highlighted are settings that work for a network secured with a WPA/WPA2 key. There are a couple of important syntax rules that are crucial. First, the SSID and key must be stored surrounded by quotes. That is why these elements are surrounded by “\” and “\”. Without these quotes the name and key will not be matched. If the MAC address for the desired AP is desired it does not need to be surrounded by quotes. Most likely this will not need to be specified since normally only the network needs to be stored (via SSID) and not a specific AP so that if the user leaves the range of that AP the device can connect to another AP with the same SSID.

The sample code selects the desired AP based on its BSSID, but this is only one way of selecting the network. One could choose a specific SSID instead, or check the levels of each AP and connect to the stronger signal, or some combination. A useful method would be `myWifiManager.compareSignalLevel(scan.level, scan2.level)`; which returns <0 if the first signal is weaker than the second, 0 if they are equal and >0 if the first signal is stronger.

Since the OS scans for APs periodically if this code is placed inside the previous code that handles the `SCAN_RESULTS_AVAILABLE_ACTION` then the programmer needs to check if the desired network is already configured or else the app will add an instance of the network into the configuration list whenever the event is captured. The line `myWifiManager.enableNetwork(id, true)`; could be placed outside of the if-statement so that the device stays

connected to the network. This line connects the device to the network with the ID=id and disconnects from all other networks if the second parameter is true.

```
List<ScanResult> wireless = myWifiManager.getScanResults(); // Returns a <list> of
scanResults

for(ScanResult scan : wireless)
{
    if(scan.BSSID.equals(desiredMACAddress))
    {
        boolean cont = true;
        for(WifiConfiguration w: myWifiManager.getConfiguredNetworks())
        {
            String s = "\"" + scan.SSID + "\"";
            String bs = scan.BSSID;
            if((w.SSID != null && w.SSID.equals(s)) || (w.BSSID != null && w.BSSID.equals(bs)))
            {
                cont = false;
                break;
            }
        }
        if(cont)
        {
            WifiConfiguration config = new WifiConfiguration();
            config.SSID = "\"" + scan.SSID + "\"";
            config.BSSID = scan.BSSID;
            config.priority = 1;
            config.preSharedKey = "\"" + "PASSWORD" + "\"";

            config.status = WifiConfiguration.Status.DISABLED;
            config.status = WifiConfiguration.Status.CURRENT;
            config.status = WifiConfiguration.Status.ENABLED;

            config.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.IEEE8021X);
            config.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_PSK);
            config.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.NONE);
            config.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_EAP);

            config.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.TKIP);
            config.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.CCMP);
            config.allowedPairwiseCiphers.set(WifiConfiguration.PairwiseCipher.NONE);

            config.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.CCMP);
            config.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.TKIP);
            config.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.WEP104);
            config.allowedGroupCiphers.set(WifiConfiguration.GroupCipher.WEP40);

            config.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.OPEN);
            config.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.SHARED);
            config.allowedAuthAlgorithms.set(WifiConfiguration.AuthAlgorithm.LEAP);

            config.allowedProtocols.set(WifiConfiguration.Protocol.WPA);
            config.allowedProtocols.set(WifiConfiguration.Protocol.RSN);

            id=myWifiManager.addNetwork(config);
            myWifiManager.enableNetwork(id, true);
            myWifiManager.saveConfiguration();
        }
    }
}
```



The WifiConfiguration class has eleven main fields that hold the state of the network configuration. Their values can be found at <http://developer.android.com/reference/android/net/wifi/WifiConfiguration.html>.

SSID stores the name of the network.

BSSID stores a specific AP's MAC address.

preSharedKey stores the WPA key for the network.

wepKeys stores up to four WEP keys for the network.

wepTxKeyIndex stores the index of the default WEP key.

status stores the current status of the network: ENABLED means the network is usable. DISABLED means it cannot be used. CURRENT means that the network is the network currently being used.

allowedKeyManagement stores all of the possible allowable key management schemes: IEEE8021X uses EAP and optionally dynamically created WEP keys. NONE means that WPA is not used, but static WEP can be used. WPA\_EAP is for WPA using EAP authentication. WPA\_PSK is for using a pre-shared WPA password key.

allowedPairwiseCiphers stores the allowable pairwise ciphers for WPA. CCMP is for AES in counter mode with CBC-MAC. NONE means only group keys are used. TKIP is for Temporal Key Integrity Protocol.

allowedGroupCiphers stores the recognized group ciphers. CCMP and TKIP are the same as above. WEP104 is for 104-bit WEP keys. WEP40 is for 40-bit WEP keys.

allowedAuthAlgorithms stores the recognized IEEE 802.11 authentication algorithms for the network. LEAP is for a LEAP network. OPEN is for an open system and is used with WPA/WPA2. SHARED is used with a WEP secured network.

allowedProtocols stores the recognized security protocols for the network. RSN is used with WPA2 security. WPA is used with WPA security.

All these settings are configuration settings of the AP or network to which the device is connecting. If these settings are unknown they will need to be obtained from the network administrator, or a different network will need to be chosen.

## Removing Saved Network Configurations

If the app is only meant to use a certain network temporarily and the user is not supposed to have continued access, or does not need continued access, to that network it is very simple to remove the saved configuration settings from the device. The method `myWifiManager.removeNetwork(id)`; will remove the network with the passed id from the list of configured networks, forgetting the settings for that network. This network id is returned by the method `myWifiManager.addNetwork(config)`; when the connection is saved, but can also be obtained from the WifiConfiguration object associated with that network. This means that it is possible to remove a network whose network id was not saved, or a network that was not added during the running of the current app.

```

for (WifiConfiguration config : myWifiManager.getConfiguredNetworks())
{
    if (config.SSID == "\"" + "DESIRED SSID" + "\")
    {
        myWifiManager.removeNetwork(config.networkId);
        break;
    }
}

```

Again, the desired network can be found by any of the saved information in the WifiConfiguration.

## Conclusions

While there is a lot of good documentation giving brief explanations on each method and class values, there isn't much on how to put all the Wi-Fi management code together to make it work. The purpose of this application note is to help remedy that issue and give a good starting point for using the WifiManager class to accomplish meaningful tasks within an Android application.

## Side Notes and Recommendations

Notes on the WifiManager class: There are two methods that can be run without an instantiation of the WifiManager class. These are calculateSignalLevel() and compareSignalLevel(). These are static methods that don't depend on any information within a WifiManager and just convert or compare two values. This means both of these methods can be invoked with WifiManager.calculateSignalLevel() and WifiManager.compareSignalLevel().

There are also a couple classes within the WifiManager: MulticastLock and WifiLock. Multicast lock is used to allow the app to receive multicast packets.

```

WifiManager.MulticastLock myMCLock = myWifiManager.createMulticastLock(Tag);
myMCLock.acquire();

```

This allows the app to receive multicast packets until myMCLock.release(); is called. If reference counting is activating with myMCLock.setReferenceCounted(true); then acquire method is invoked multiple times, the MulticastLock class will keep a counter of how many times it has been acquired. Each invocation of release() will then decrement that count and once it reaches zero the app will stop receiving multicast packets. If the reference counted field is set to false then all calls to acquire after the first are ignored and the first call to release() will close the lock. The documentation for this class is located at <http://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.html>.

The WifiLock class is used in a similar fashion, but its function is to prevent the device from shutting off the Wi-Fi while it is idle. This is done by default to conserve the battery. The calls are the same as above and this lock also can be reference counted. The documentation for this class can be found at <http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html>.

It is always good programming practice to wrap any code using a WifiManager with

```

if (myWifiManager != null && myWifiManager.isWifiEnabled())
{
    //The code
}

```

To prevent attempting to access a null reference or perform actions on a Wi-Fi system that is not enabled.

It is possible that the user may want to keep the Wi-Fi activated after the app runs. If this is the case then the programmer could have a dialog box come up asking for user input instead of automatically turning the Wi-Fi back off.

If a TimerTask is used it is very important to know how to end the task properly. Doing this only takes a couple of lines of code:

```
task.cancel();//cancel all scheduled instances of this task  
timer.cancel();//cancel all scheduled tasks associated with this timer  
timer.purge();//remove all canceled tasks from the task queue
```

## References

*Android Developers*. (2011). Retrieved from <http://developer.android.com/index.html>

Nils. (2010, June 5). *Android Scan for Wifi networks*. Retrieved March 30, 2011, from Stack Overflow:  
<http://stackoverflow.com/questions/2981751/android-scan-for-wifi-networks>