Analyzing Active Scans

The Goals

1. Find how higher level scan functions call the 802.11 scanning functions.
2. Find how android decides which channels to scan and in what order.

The Call Graph

Tracing a Periodic Scan

The start of this trace occurs at `startPeriodicScan` in this file. For this trace, we'll skip some of the details pertaining to the timing of scan requests and instead focus on tracing these calls to the 802.11 functions as well as investigating any channel related behavior.

The only line of importance is:

```
private void startPeriodicSingleScan() {

  ...

  boolean isFullBandScan = true;

  ...

}
```

So each time we do a periodic scan, we scan the full band. `startPeriodicScan` then calls `startSingleScan`, which in turn calls `startForcedSingleScan`. Within `startForcedSingleScan`, we check whether we passed in true for `isFullBandScan` or if we should be looking for a list of channels to scan somewhere else. Since we always pass in true for this, we don't worry about the other case.

The important code in `startForcedSingleScan` is highlighted below:

```
private void startForcedSingleScan(boolean isFullBandScan, WorkSource workSource) {
  ...

  settings.band = getScanBand(isFullBandScan);

  // some stuff related to scanning 6GHz, left out for now

  ...
}
```

`getScanBand` will return one of the following options:

1. WifiScanner.WIFI_BAND_24_5_WITH_DFS_6_GHZ;

2. WifiScanner.WIFI_BAND_BOTH_WITH_DFS;
3. WifiScanner.WIFI_BAND_ALL;
4. WifiScanner.WIFI_BAND_UNSPECIFIED;

We can safely rule out (4) since it depends on `isFullBandScan` being false which is not possible. Deciding between the first three is rather interesting. The code follows the following structure:

```
if SDK > 31 && canScan6GHz { return WifiScanner.WIFI_BAND_24_5_WITH_DFS_6_GHZ; }

if SDK > 31 { return WifiScanner.WIFI_BAND_BOTH_WITH_DFS; }

return WifiScanner.WIFI_BAND_ALL;
```

Now if we take a look at how these values are defined, we get:

```
... int WIFI_BAND_ALL = (1 << WIFI_BAND_COUNT) - 1;

... int WIFI_BAND_24_5_WITH_DFS_6_GHZ = WIFI_BAND_BOTH_WITH_DFS | WIFI_BAND_6_GHZ;

... int WIFI_BAND_BOTH_WITH_DFS = WIFI_BAND_24_GHZ | WIFI_BAND_5_GHZ | WIFI_BAND_5_GHZ_DFS_
```

All of the constants on the right hand side are represented as bitmaps with the following values:

```
... int WIFI_BAND_24_GHZ = 1 << WIFI_BAND_INDEX_24_GHZ; // = 1 << 0 = 0

... int WIFI_BAND_5_GHZ = 1 << WIFI_BAND_INDEX_5_GHZ; // 1 << 1 = 2

... int WIFI_BAND_5_GHZ_DFS_ONLY  = 1 << WIFI_BAND_INDEX_5_GHZ_DFS_ONLY; // 1 << 2 = 4

... int WIFI_BAND_6_GHZ = 1 << WIFI_BAND_INDEX_6_GHZ; // 1 << 3 = 8
```

Thus, going back to figuring out what we scan, we actually scan every channel that we can if the phones SDK is less than 32 (which is interesting) Going back to `startForcedSingleScan` we call `WiFiScanner.startScan()` which is located here This functions packages up everything we've found so far and sends an asynchronous message containing the constant `CMD_START_SINGLE_SCAN`. That is the end of this chain, but we can see how this message gets handled by searching for this string. Doing so yields two functions:

```
// FUNCTION 1
handleScanStartMessage()


// Function 2
mSingleScanStateMachine.sendMessage()
```

I felt that `handleScanStartMessage` seemed more like what we were looking for, so I stuck with that one. I did place inspecting `mSingleScanStateMachine.sendMessage()`

to a list of things to try if our results are not favorable. Anyways, `handleScanStartMessage` is located here. The important chunk of code in this function happens near the end:

```
if (getCurrentState() == mScanningState) {

    // (1) currently scanning but the current scan can satisfy the new request
    if (activeScanSatisfies(scanSettings)) {
        mActiveScans.addRequest(ci, handler, workSource, scanSettings);
    }

    // (2) currently scanning but the current scan cannot satisfy the new request
    else {
        mPendingScans.addRequest(ci, handler, workSource, scanSettings);
    }
}

// (3) currently idle, just start the scan
else if (getCurrentState() == mIdleState) {
    mPendingScans.addRequest(ci, handler, workSource, scanSettings);
    tryToStartNewScan();
}

// (4) only reach here if its an emergency scan
else if (getCurrentState() == mDefaultState) {
    mPendingScans.addRequest(ci, handler, workSource, scanSettings);
```

We disregard (4), since it is not relevant to periodic scanning (as far as I know of). Paths (2) and (3) are the ones we are interested in, and they are both handled by the same function located here : `tryToStartNewScan()`. This function contains a lot of information on how the collection of channels is created so we'll start by breaking those down. First we call `mChannelHelper.updateChannels()` located in this file. `updateChannels()` calls `mWifiNative.getChannelForBands()` which uses the integer values for the bands we looked at earlier (i.e WIFI_BAND_24_GHZ). `getChannelForBands()` in turn calls `mWifiCondManager.getChannelsMhzForBand()` located here. Since this function doesn't have an implementation in Java (that I could see), it likely queries the interface for which channels it can scan. The list that is returned likely abides by country regulations since the drivers know the country code of the device. Even though we can't see the implementation, we know it returns an array of integers. It is likely that these are channel numbers: `(1, 6, 11, ...)` or a list of frequencies in Megahertz: `(2312, 2437, ...)`.

Back in `tryToStartNewScan` we've got our list of channels and now this loop:

```
for (RequestInfo<ScanSettings> entry : mPendingScans) {

    ...
    // (1) adding channels based on the scan request
    channels.addChannels(entry.settings);

    ...
}
```

Since we're passing in a value other than `WIFI_BAND_UNSPECIFIED`, `addChannels` calls `addBand()` which is defined here. `addBand()` in turn calls `getAvailableScanChannels(band)`. This function just checks which of the channels we can scan given a value like `WIFI_BAND_24_5_WITH_DFS_6_GHZ` and returns a list of channels that we want to scan.

Once we have our array of channels to scan, we fill the buckets that we will pass to the next scanning function by calling `fillBucketSettings` located here.

```
public void fillBucketSettings(WifiNative.BucketSettings bucketSettings, int maxChannels) {

  // (1) we want to scan all of the bands that we can
  if ((mChannels.size() > maxChannels || mAllBands == mExactBands) && mAllBands != 0) {
      bucketSettings.band = mAllBands;
      bucketSettings.num_channels = 0;
      bucketSettings.channels = null;
  }

  // (2) we want to scan a particular set of channels
  else {
      bucketSettings.band = WIFI_BAND_UNSPECIFIED;
      bucketSettings.num_channels = mChannels.size();
      bucketSettings.channels = new WifiNative.ChannelSettings[mChannels.size()];
      for (int i = 0; i < mChannels.size(); ++i) {
          WifiNative.ChannelSettings channelSettings = new WifiNative.ChannelSettings();
          channelSettings.frequency = mChannels.valueAt(i);
          bucketSettings.channels[i] = channelSettings;
      }
  }
}
```

Thus, this may be a good point to look at in the future if we want to determine what the bucket looks like before it gets passed to a scan function. For now we will continue our trace. Now that we have our buckets ready, we call `mScannerImplsTracker.startSingleScan()` which is located here. This function tries the scan on any available interface by calling `impl.startSingleScan()`.

This function is defined in 2 places: 1 and 2. I focused more on 1 since it looked more promising.

(1) is implemented in WificondScannerImpl, where the following happens:

```
...

ChannelCollection allFreqs = mChannelHelper.createChannelCollection();

for (int i = 0; i < settings.num_buckets; ++i) {

    ...


    allFreqs.addChannels(bucketSettings)

}

...

freqs = allFreqs.getScanFreqs();

success = mWifiNative.scan(getIfaceName(), settings.scanType, freqs,
      hiddenNetworkSSIDSet,settings.enable6GhzRnr);
```

`freqs` is just a set of frequencies in MHz (i.e 2312, . . .)

Looking into `mWifiNative.scan()`(located here), we see that it calls `mWifiCondManager.startScan()`, which in turn does some conversions with its parameters and passes them to `WifiNl80211Manager.startScan()`(located here ,which in turn converts the parameters to the required type and calls `scannerImpl.scan()` (defined here.

Since this function does not have a definition, it is likely that this is the last message (or one of the last) messages before sending the request to the device. Although this is the path I traced, there may be some older devices still using `startBackgroundScan()` (located here which eventually leads to `BackgroundScanScheduler` (located here. This class contains a lot of relevant functions, but since background scans are depreciated, I chose to ignore them.

Other Things to Try

1. See what `mSingleScanStateMachine.sendMessage()` does.