1. Introduction

Beacodes are inaudible "codes" or messages that are transmitted using acoustic waves (sound). These messages may be received by almost any smartphone including Android, iOS and Windows Phone based devices.

Beacodes define 8 standard audio channels from 18 kHz to 22 kHz. Each channel occupies 500 Hz band. The scanner listens to all channels simultaneously, which means it can receive up to 8 different messages at once. Alternatively, the same message may be transmitted on multiple channels to improve reception quality.

Beacode scanner internally uses few different techniques to improve signal reception. Messages are divided into smaller data frames. Each message is typically transmitted multiple times, so even that some frames get corrupted, the scanner can eventually receive full message after a few retransmissions. This is only necessary when the signal quality is low (low transmitting volume/large distance between transmitter and receiver). The scanner also uses sophisticated error correction algorithms which significantly improve reception quality.

Beacode protocol recognizes popular string payloads and compresses them transparently to the user. For instance transmitting an URL like http://www.example.com encodes only 8 bytes instead of 22 bytes uncompressed. In this case the transmission duration will be about 400 ms instead of 1 second.

2. Getting started

Following instructions and examples were prepared and tested with Android Studio 2.2.3.

2.1. Examples

You will find Android Studio examples located in android/examples directory of this SDK.

2.2. Bootstrapping custom app

1. Copy included "jniLibs" directory to your app/src/main directory. Currently only ARM architecture is supported. Other architectures will be added on request. This will create new files:
   - app/src/main/jniLibs/armeabi/libbeacode.so (scanner)
   - app/src/main/jniLibs/armeabi/libbeacode-siggen.so (signal generator)
   - app/src/main/jniLibs/armeabi-v7a/libbeacode.so (v7A version includes NEON optimizations)
   - app/src/main/jniLibs/armeabi-v7a/libbeacode-siggen.so

If your app only scans or only transmits data then you can remove unused .so libraries.

2. Copy included "com" folder to your app/src/main/java directory. This will create new files:
   - app/src/main/java/com/beacodes/BeacodeScanner.java
   - app/src/main/java/com/beacodes/BeacodeSiggen.java

3. Add RECORD_AUDIO permission to your AndroidManifest.xml file, directly under the root "manifest" element:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

4. Add imports to your main activity java file (typically MainActivity.java):

```
    import com.beacodes.BeacodeScanner;
    import com.beacodes.BeacodeSiggen;
```

5. In your main activity class, declare static variables for handling run-time microphone permission request. If your app is requesting other permissions, you may need to adjust MIC_REQUEST_CODE value.

```
     private static final int MIC_REQUEST_CODE = 1;
     private static boolean isDialogShown = false;
```

6. Define following functions for handling run-time microphone permission request. Change YOUR_APP_NAME accordingly.

```
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
@NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == MIC_REQUEST_CODE) {
        if (grantResults[0] != PackageManager.PERMISSION_GRANTED) {
            // permission denied
            if (!ActivityCompat.shouldShowRequestPermissionRationale(this,
             Manifest.permission.RECORD_AUDIO)) {
                // permission denied with selecting "never ask again"
                AlertDialog alertDialog = new
                    AlertDialog.Builder(MainActivity.this).create();
                alertDialog.setTitle("Microphone required");
                alertDialog.setMessage("Please allow microphone permission in the
Android settings (Settings -> Applications -> YOUR_APP_NAME).");
                alertDialog.setButton(AlertDialog.BUTTON_NEUTRAL, "OK",
                        new DialogInterface.OnClickListener() {
                            public void onClick(DialogInterface dialog, int which) {
                                isDialogShown = false;
                                dialog.dismiss();
                            }
                        });
                alertDialog.show();
                isDialogShown = true;
            }
        }
    }
}

public void checkPermissions() {
    if (ContextCompat.checkSelfPermission(this,
       Manifest.permission.RECORD_AUDIO) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new String[]
             { Manifest.permission.RECORD_AUDIO }, MIC_REQUEST_CODE);
    }
}
```

7. Implement BeacodeScanner.Listener interface in your activity class:

```
    public class YourActivity extends AppCompatActivity implements
            BeacodeScanner.Listener
    {
        …
        @Override
        public void onStateChange(BeacodeScanner.State oldState,
        BeacodeScanner.State newState) {
        }

        @Override
        public void onPartialMessage(int messageId, int percent) {
        }

        @Override
        public void onPartialMessageCancelled(int messageId) {
        }

        @Override
        public void onMessage(int messageId, BeacodeScanner.Message
        message) {
        }
    }
```

8. Setup the scanner in onCreate():

```
    BeacodeScanner.setListener(this);
    BeacodeScanner.setProfile(1);
    // listen on all standard channels
    long[] channelSpecs = new long[8];
    for (int i = 0; i < 8; ++i)
        channelSpecs[i] = BeacodeScanner.getStandardChannelSpec(i);
    BeacodeScanner.setChannelSpecs(channelSpecs);
```

9. Start scanning in onResume(), include microphone permission handling code:

```
    @Override
    protected void onResume() {
        super.onResume();
        // wait for a full stop to avoid calling start in
        // the RUNNING state because otherwise it would not
        // fire the state change event
        while (BeacodeScanner.getState() != BeacodeScanner.State.STOPPED)
            SystemClock.sleep(10);
        if (!isDialogShown) {
            checkPermissions();
            BeacodeScanner.start();
        }
    }
```

10. Stop scanning in onPause():

```
    @Override
    protected void onPause() {
        super.onPause();
        BeacodeScanner.stop();
    }
```

11. Build and run your app. Examine Android Monitor window, BeacodeScanner will log each received message there. Try some demonstration signals from “demo” directory in this SDK.

12. Write your own message handling code in onMessage() callback. Optionally use other callbacks for tracking message scanning progress.

3. BeacodeScanner reference

```
public static native void start();
```

Starts the receiver, typically called in Activity's onResume()

```
public static native void stop();
```

Stops the receiver, typically called in Actitivy's onPause()

```
public static native boolean setProfile(int profile);
```

Used to set processing profile, currently there are only two profiles

- 0 – baseline profile
- 1 – „better” profile, this one does additional processing, which improves signal reception (recommended profile, unless target device is very low-end and slow)

Returns true if the passed profile is valid (currently 0 or 1), false otherwise

```
public static native boolean setChannelSpecs(long[] channelSpecs);
```

Sets listening channel specifications. Scanner supports maximum 32 channels.

```
public static native long getStandardChannelSpec(int channel);
```

Returns standard channel specification. There are 8 standard channels (0 - 7).

```
public static void setListener(Listener listener);
```

Used to set the listener for Beacode callbacks

```
public static Listener getListener();
```

Returns current listener

```
public static State getState();
```

Returns current state

```
public interface Listener {
    void onScannerStateChange(State oldState, State newState);
```

Called when receiver state changes, the states are

- STOPPPED – initial state
- MIC_IN_USE – when other app is currently using the microphone
- RUNNING – when the receiver is listening for the Beacodes

BeacodeScanner will automatically switch between MIC_IN_USE and RUNNING states, f.i. when microphone is released by another app, the scanner will automatically switch to RUNNING state (see attached diagram below).

```
    void onPartialMessage(int messageId, int percent);
```

Beacodes are composed of smaller frames that build entire message. Single frames may become corrupt, depending on the signal quality. The scanner will listen for the retransmission and will build up the message from tiny pieces. This means that repeating the same message many times will significantly improve the chance of receiving full message. This callback is called on each partial message progress. For example it may be called on 30%, then on 60 % and so on. It won't be called on full message though. Full messages are always signaled by onMessage callback.

`int messageId` – this is unique id per message, as there may be many messages transmitted concurrently, typically on separate channels
`int percent` – current progress of the message [1..99]

```
    void onPartialMessageCancelled(int messageId);
```

When single message is not completed for some time, it is cancelled. Currently the time limit is fixed to 60 seconds. It may become configurable in future releases.

`int messageId` – unique id of the message previously signaled by onPartialMessage.

```
    void onMessage(int messageId, Message message);
```

This is called when full message is received. Please see Message documentation below.

`int messageId` – unique id of the message
`Message message` – received message, see below

```
}

public static class Message {
    public int getId();
```

Returns message's unique id, the same as in the callbacks.

```
    public boolean isUidIncluded();
```

Beacode message may optionally contain unique 32 bit id, for use with geo tagging beacons. This function returns true if message contains such UID.

```
    public int getUid();
```

Returns unique 32 bit id attached to this message. See isUidIncluded() above.

```
    public Tag getTag();
```

Returns message type tag. Currently there are only two types, RAW and TEXT.

```
    public Object getPayload();
```

Returns message payload (actual transmitted data). It will be byte[] if getTag() returns RAW and String if getTag() returns TEXT.

```
    public byte[] getRawPayload();
```

Convenience function to get raw payload without casting.

```
    public String getStringPayload();
```

Convenience function to get string (TEXT) payload without casting.

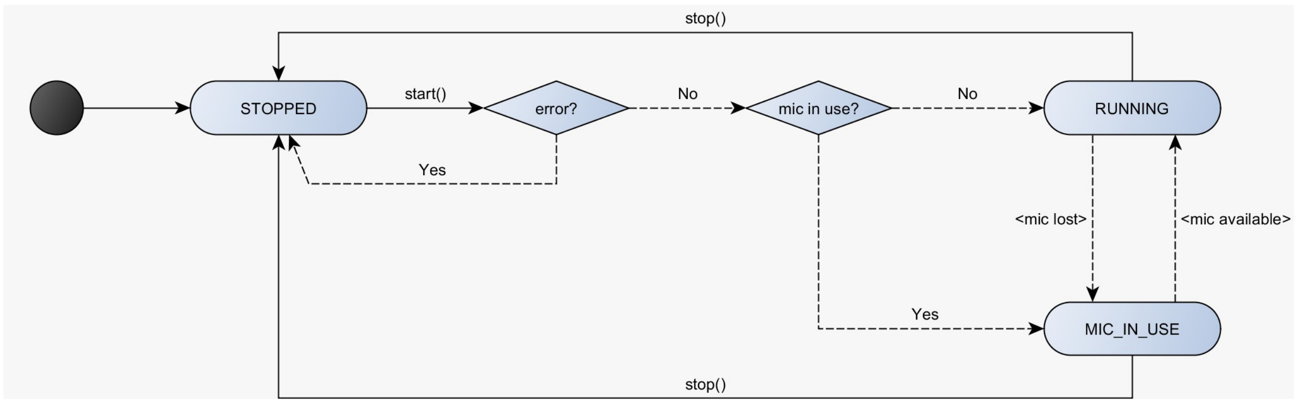
```
    public String toString();
```

Returns textual representation of the message data. It will be actual string data for TEXT payloads and hexadecimal representation of the RAW payloads.

```
}
```

4. BeacodeScanner state machine

State MIC_IN_USE applies only to Android. Other OSes allow microphone sharing between apps.

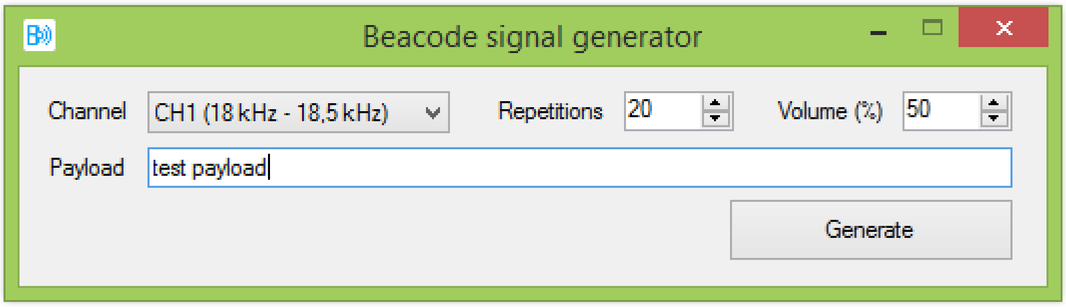


5. Beacode signal generator app (full SDK only)

This application require .NET Framework 4.0 or newer.

To generate a signal, select a channel (CH1-CH8), specify payload text and click Generate button. You will be prompted for output file name (*.wav). Number of repetitions means number of message copies, it is recommended that there are at least 10 repetitions. This will allow to assemble the message from its frames when some of them become corrupt during transmission. In addition, frames are shuffled between repetitions to increase reception probability (it helps when part of the channel's frequency band is noisy or there is some interference in the channel). Volume is the maximum sound level of the output file, it is recommended to use more than 50% volume only when transmitting on a single channel. Playing multiple channels simultaneously (multiple messages on separate channels) with high volume may produce clicking in the speakers.

Output file may be played continuously in a loop.



Channel: CH1 (18 kHz - 18,5 kHz)
Repetitions: 20
Volume (%): 50
Payload: test payload

6. BeacodeSiggen interface (full SDK only)

```
public static native void start();
```

Starts the signal generator, typically called in Activity's onResume()

```
public static native void stop();
```

Stops the signal generator, typically called in Actitivy's onPause()

```
public static native void resetPayload();
```

Resets the payload to empty one as if the signal generator was initialized for the first time. This may be called in RUNNING state.

```
public static native boolean setTextPayload(String payload);
```

Sets the text payload for transmission. Returns true on success, and false if payload is too long or empty.

```
public static native boolean setRawPayload(byte[] payload);
```

Sets raw byte[] payload for transmission. Returns true on success, and false if payload is too long or empty.

```
public static native boolean setStandardChannel(int channel);
```

Sets standard transmission channel (0 – 7). Returns true on success, and false when channel is out of range.

```
public static native boolean setCustomChannel(long customChannel);
```

Sets custom transmission channel. Returns true on success, and false when customChannel is invalid.

```
public static native boolean setSignalVolume(int volume);
```

Sets the signal volume (0 – 100). This is independend of the speaker volume. Please make sure that speaker volume is also raised.
Recommended values are 90-100 for signal volume, and 50-70 for speaker volume. Setting higher values may cause clicking in the speakers.

```
public static void setListener(Listener listener);
```

Used to set the listener for Beacode callbacks

```
public static Listener getListener();
```

Returns current listener

```
public static State getState();
```

Returns current state

```
public interface Listener {
    void onSiggenStateChange(State oldState, State newState);
```

Called when signal generator state changes, the states are

- STOPPPED – initial state
- RUNNING – when the signal generator is transmitting Beacodes

Signal generator will switch to RUNNING state even if the payload is not set (or reset). To transmit Beacodes you need to also setup the payload (setXXXPayload functions).

```
}
```

7. Examples

Please open the examples in Android Studio and examine them and their source code. You can use example code as a base for your application.

7.1. HelloBeacodes

This is simple example of how to use BeacodeScanner interface. It displays the log of all received full and partial messages.

7.2. SiggenDemo (full SDK only)

This is the signal generator example.

Please run this example on the first device, and Beacode Scanner app on the second device. Press START button to enable signal transmission. You can change transmission channel, signal and speaker volume and text message on the fly. Beacode Scanner should receive the messages you are transmitting.

7.3. ChallengeResponseDemo (full SDK only)

This demo shows how to use both BeacodeScanner and BeacodeSiggen in one application to securely open gateway doors.

Please run this example on two devices. On the first device, please select GATEWAY option, and on the second device please select KEY FOB option. If you place these two devices in proximity, the challenge response protocol will authorize the user and will "open the gate".

As a gateway you can add, edit and remove users.
As a key fob, you can edit your ID number and password.

The gateway constantly transmits a nonce (that is number used once). Nonces must be unpredictable, we recommend that they are generated by cryptographically secure random number generator. When the key fob device receives the nonce, it will compute the SHA-256 hash based on the user password and the nonce. The hash is then truncated to 11 bytes and transmitted with user's ID to the gateway. When the gateway receives the ID and hash, it recomputes the hash based on the password stored in the database and compares it to the received hash. If the hashes match, the user is considered as authorized and the gate may be opened.

Both gateway and key fob transmit on separate channels to avoid mutual packet collisions.