TECHNISCHE
UNIVERSITÄT
DARMSTADT

ANSIAN - ANDROID SIGNAL ANALYZER

DENNIS MANTZ AND MAX ENGELHARDT

SEEMOO Secure Networking Lab

September 1, 2016

Secure Mobile Networking Lab
Department of Computer Science

SECURE MOBILE NETWORKING

AnSiAn - Android Signal Analyzer
SEEMOO Secure Networking Lab

Submitted by Dennis Mantz and Max Engelhardt
Date of submission: September 1, 2016

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Jiska Classen

ABSTRACT

SDR platforms like the *HackRF* enable their users to explore the electromagnetic spectrum. While a wide range of SDR software toolkits, such as gqrx and SDR#, is available for computers, little such software exists for more mobile devices like smartphones and tablets. AnSiAn is a portable open-source SDR toolkit for Android devices that enables users to inconspicuously monitor the wireless spectrum anywhere. In this lab, AnSiAn's performance was improved and its feature set was extended by adding support for the SDRPlay and rad10 SDR platforms, demodulators for RDS and PSK31 and transmission capability for raw I/Q-Files, Morse and PSK31.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADC	Analog-to-Digital Converter
AGC	Automatic Gain Control
AM	Amplitude Modulation
AnSiAn	Android Signal Analyzer
BPSK	Binary Phase Shift Keying
FFT	Fast Fourier Transform
FM	Frequency Modulation
GC	Garbage Collector
GUI	Graphical User Interface
IF	Intermediate Frequency
MVC	Model–View–Controller
PI	Program Identifier
PLL	Phase Locked Loop
PSK31	Phase Shift Keying, 31 Baud
RDS	Radio Data System
RF	Radio Frequency
SDR	Software-Defined Radio
SSB	Single Side Band
USB	Upper Side Band
VGA	Variable-Gain Amplifier

INTRODUCTION

Android Signal Analyzer (AnSiAn) is an Android application developed by the Secure Mobile Network Lab (SEEMOO) at Technische Universität Darmstadt. It allows for common Software-Defined Radio (SDR) platforms, such as the HackRF and the RTL-SDR, to be used with Android devices. This enables a user to inconspicuously sniff and analyze wireless signals on the go.

The project is based on the open-source app RFAnalyzer by Dennis Mantz [2], which features visual browsing through the frequency domain and demodulation of e.g. Amplitude Modulation (AM) and Frequency Modulation (FM) signals. It has been further developed by Markus Grau and Steffen Kreis into what is now AnSiAn in 2015.

To date, AnSiAn adds to following features on top of RFAnalyzer:

- Waveform graph of received signalsu

- Morse Demodulation and Decoding

- RF Scanner that allows for visually scanning a broad spectrum for signals

- Restructured Graphical User Interface (GUI) with better usability

- Restructured codebase that follows the Model–View–Controller (MVC) pattern

While this makes AnSiAn a powerful tool for mobile signal analysis and processing, further features such as support for additional modulation techniques are desirable. This lab aims to further develop AnSiAn in order to extend its feature set, improve app stability and refine existing features.

A detailed description and explanation of the project goals and their planned schedule is given in Chapter 2. Chapter 3 covers the actual project progress over time as well as encountered problems, delays and unplanned features in each phase of the project. Details on the design and implementation of features and bugfixes are given in Chapter 4. Chapter 5 concludes this documentation.

PROJECT DEFINITION

This chapter defines the mandatory and optional features that were scheduled for implementation throughout the project and divides them into three sprints.

2.1 FEATURES

The new features, that were to be implemented, can be divided into two groups: mandatory features of high priority within this project and optional features, that were to be implemented if time permitted. As can be seen in Section 2.2, the third sprint was reserved either for the implementation of optional features or for completing mandatory features and this documentation.

2.1.1 *Mandatory Features*

The following core features were scheduled for implementation during the first and second sprint:

- Radio Data System (RDS) demodulation
  If the user selects the existing wide-band FM demodulation option, the app shall try to detect and demodulate any existing RDS signal along with the FM audio demodulation. The extracted information shall be displayed on the screen.

- Phase Shift Keying, 31 Baud (PSK31) demodulation
  If the user selects Upper Side Band (USB) demodulation mode, he or she shall have the option to enable PSK31 demodulation along with the audio demodulation. The demodulated text string shall appear and scroll through the analyzer window.

- Extraction of demodulated RDS-, Morse- and PSK31-data to logfiles
  If the user selects to demodulate any digital modulation, the demodulated text shall be written to a log file specified by the user.

- Support for the rad10 badge
  The rad10 badge, which is a modified low-cost replica of the HackRF, shall be supported as a signal source by AnSiAn.

- Transmission support for HackRF and rad10
  If AnSiAn is used with an SDR capable of transmitting signals, it shall offer options to send signals in the following ways:

– Replay I/O samples from a file

– Generate and send Morse code from text

– FM-modulate and send audio from a file

2.1.2 *Optional Features*

Optional features were scheduled for the third and last sprint. However, they were only to be implemented if the last sprint was not needed in order to compensate for delays on the mandatory features. The optional features are listed in order of priority:

- Walkie-Talkie Mode
  The user shall have the possibility to put AnSiAn into a Walkie-Talkie mode. In this mode, the application will demodulate an FM channel and the user can quickly switch between demodulation and transmission of audio recorded from the internal microphone.

- Packet Radio demodulation
  A new mode *Packet Radio* shall be added to AnSiAn. Once selected, it shall allow the user to tune to a Packet Radio channel and display information about demodulated packets on the screen. If time permits, it might even be possible to implement a transmission feature for Packet Radio.

2.2 TIME SCHEDULE

The project had two developers, Dennis Mantz and Max Engelhardt, working in three sprints. There were three milestones corresponding to the sprints, labeled Alpha, Beta and Final Version. They each add an independent and self-contained set of features to the application:

- Software Design (due 12.05.)

- Sprint 1: Alpha Version (due 09.06.)

  – RDS demodulation

  – PSK31 demodulation

  – Extraction of RDS-, Morse- and PSK31-data to logfiles

- Sprint 2: Beta Version (due 21.07.)

  – Support for the rad1o badge

  – Transmission support for HackRF and rad1o

    ∗ Replay I/O samples from a file

    ∗ Generate and send Morse code from text

    ∗ FM-modulate and send audio from a file

- Sprint 3: Final Version (due 25.08.)
    - Complete leftovers from previous sprints
    - Walkie-Talkie Mode (optional)
    - Packet Radio demodulation (optional)

3

PROJECT PROGRESS

3.1 SPRINT 1: ALPHA VERSION

3.2 SPRINT 2: BETA VERSION

3.3 SPRINT 3: FINAL VERSION

DESIGN AND IMPLEMENTATION

This chapter covers the design and implementation of the new features. It also addresses structural changes, that were necessary for these features to be implemented, as well as optimizations and bugfixes in the original codebase.

4.1 MEMORY OPTIMIZATIONS

The original RF Analyzer application's architecture is based on blocking queues that synchronize the various signal processing threads and efficiently manage memory buffers. Unfortunately, this architecture was partly dropped by the developers of AnSiAn when changing to a new architecture based on the EventBus library. As a result, memory allocation management does not work as efficiently with the current version of AnSiAn.

Instead of using cycling buffers for inter-thread-communication, AnSiAn uses EventBus to deliver data. Buffers are always allocated freshly and discarded after use. This results in a high activity of the Garbage Collector (GC) and therefore in a bad overall performance of the app.

Listing 1 shows a logcat output of the app before any optimizations were applied. The GC runs approximately 8 times per second and the slow performance results in stuttering audio demodulation on older hardware.

Listing 1: Logcat output before memory optimizations

```
05-12 17:55:04.060 D/dalvikvm: GC_FOR_ALLOC freed 4347K, 14% free
    54695K/62984K, paused 28ms, total 28ms
05-12 17:55:04.180 D/dalvikvm: GC_FOR_ALLOC freed 4321K, 14% free
    54737K/62984K, paused 26ms, total 26ms
05-12 17:55:04.300 D/dalvikvm: GC_FOR_ALLOC freed 4507K, 14% free
    54705K/62984K, paused 32ms, total 32ms
05-12 17:55:04.420 D/dalvikvm: GC_FOR_ALLOC freed 4454K, 14% free
    54759K/62984K, paused 30ms, total 30ms
```

In order to fix this performance issue, the architecture is reverted to using blocking queues and cycling buffers in places where large memory buffers are passed between threads. EventBus is still used for delivering information which is not tied to large buffers. A schema of the new architecture is depicted in Figure 1.

In this architecture, the buffers cycle between the threads. The reusage of buffers helps to reduce the memory allocation and garbage collection overhead to a minimum. Listing 2 shows the logcat output
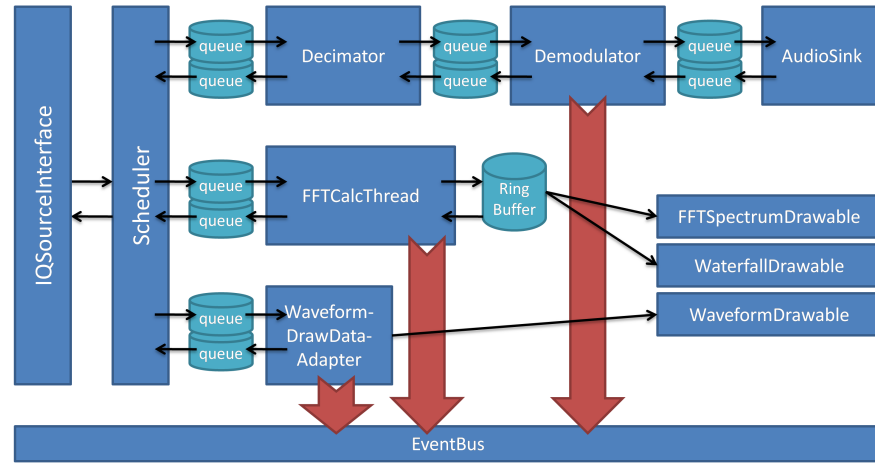
Figure 1: Signal processing architecture with blocking queues

after the architecture changes have been applied. The GC only needs to run every 10 to 20 seconds.

Listing 2: Logcat output after memory optimizations

```
05-12 17:27:29.230 D/dalvikvm: GC_FOR_ALLOC freed 3233K, 15% free
    19706K/23000K, paused 32ms, total 33ms
05-12 17:27:40.780 D/dalvikvm: GC_FOR_ALLOC freed 3528K, 16% free
    20235K/23824K, paused 30ms, total 31ms
05-12 17:28:00.110 D/dalvikvm: GC_FOR_ALLOC freed 4130K, 18% free
    20338K/24528K, paused 36ms, total 37ms
05-12 17:28:24.520 D/dalvikvm: GC_FOR_ALLOC freed 4263K, 18% free
    20341K/24664K, paused 49ms, total 49ms
```

Another significant performance improvement can be gained by synchronizing the calculation of the Fast Fourier Transform (FFT) to the refresh cycle of the application GUI. This simple change reduces the number of time the FFT algorithm has to be executed and releases processing resources for other calculations (e.g. in the demodulation path).

Unfortunately, it turned out that the Morse demodulation algorithm of the previous version of AnSiAn was working in the frequency domain and relied on continuous FFT samples without gaps. As result this performance improvement comes at the cost of re-implementing the Morse demodulation algorithm in order to work on time domain samples instead.

4.2   DEMODULATION

This section describes the design and implementation process of the developed demodulation features. Section 4.2.1 covers where and how the new demodulators and the logging functionality were integrated into the existing architecture and which changes were necessary. Due to performance issues with the old frequency-domain-based Morse demodulator, it had to be rewritten. This process is described in . The implementation of the new RDS and PSK31 demodulators is covered in Sections 4.2.3 and 4.2.4 respectively. Section 4.2.5 addresses the implemented logging functionality for demodulator output.

4.2.1   *Design and Structural Changes*

The existing architecture of AnSiAn features individual threads for scheduling, downsampling, demodulation and audio output. The `Demodulator` thread demodulates quadrature samples by calling the `demodulate()` method on an instance of `Demodulation`. `Demodulation` is an abstract class that is implemented by concrete demodulation methods such as `AM`, `FM` and `Morse`.

AnSiAn utilizes the EventBus library in order to pass demodulated Morse text to the GUI. Demodulated audio data is passed to the `AudioSink` thread by enqueuing it into its input queue. This mechanism is explained in more detail in Section 4.1.

In order to extend AnSiAn with demodulation functionality for PSK31 and RDS, the existing architecture needs to be extended by two new subclasses. The extended architecture is depicted in Figure 2 and explained in the following.

Two new classes `PSK31` and `RDS` are integrated into the existing demodulation classes for USB and FM. PSK31 demodulation works on the envelope of the received USB signal. RDS transmits metadata for FM radio channels, which is why the RDS demodulation needs to work on an intermediate result of the wide-band FM demodulation class. Through a new option in the app settings, it will be possible to enable or disable the demodulation of the embedded digital modes when the corresponding analog mode is selected.

Like the existing architecture, the new architecture will use the EventBus library to pass the demodulated text to the GUI. The existing View `MorseReceiveView` is refactored into a universal `DemodulationInfoView` that displays the text output of any selected demodulator. Demodulators pass `DemodInfoEvents` via the EventBus to the `DemodulationInfoView`, which contain demodulated text and further information (e.g. baud rates or raw dits and dahs) and are displayed in separate lines.
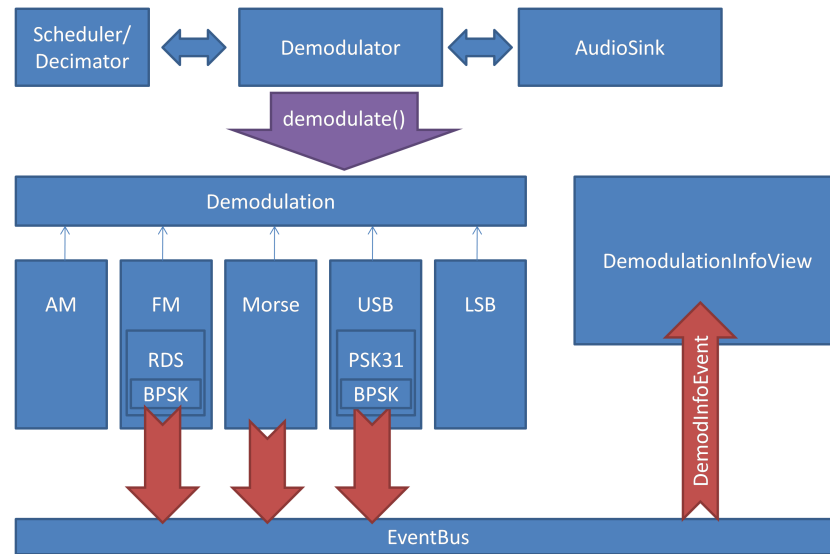
Figure 2: Architecture of the extended demodulation logic and communication with the GUI

The new class `Logger` will also listen for `DemodInfoEvents` and, if logfiles for demodulator output are enabled by the user, persist the demodulated data to these logfiles.

4.2.2 *Re-Implementation of Morse Demodulator*

As explained in Section 4.1, the original version of AnSiAn calculated FFTs excessively often and thus offered bad performance. AnSiAn's original Morse demodulator, which operated in the frequency domain, relied on frequent FFT calculations and did not work after the optimizations described in Section 4.1. In order to still demodulate Morse with the new efficient architecture, the Morse demodulator needed to be re-written to work in the time domain rather than the computationally expensive frequency domain.

4.2.2.1 *Requirements*

Aside from demodulating and decoding Morse transmissions, the old demodulator offered support for the following features, which the new demodulator should implement as well.

- "Manual", "semi-automatic" or "automatic" detection of dit duration and threshold

- Optional automatic re-initialization of detected dit duration and threshold in case of high error rate

Figure 3: States and state transitions of the re-written Morse demodulator
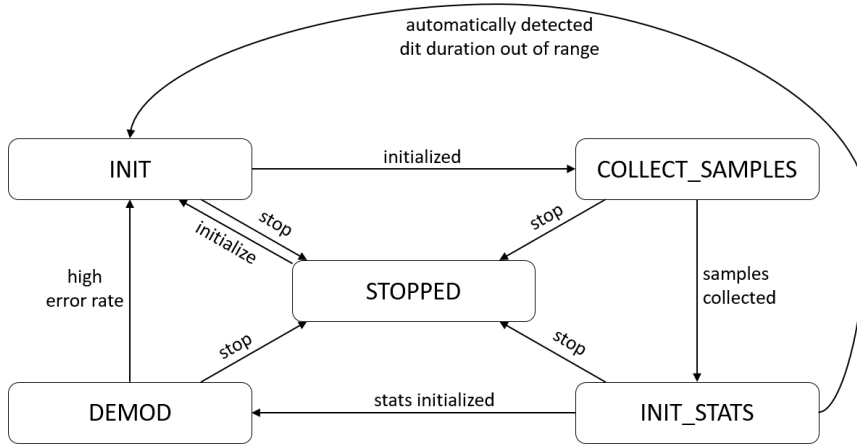
- Optional AM audio demodulation along with Morse demodulation

In automatic mode, the demodulator automatically estimates the high/low threshold and the duration of a dit based on a set of samples, that needs to be collected before the actual demodulation can start. Manual mode requires the user to explicitly specify the dit duration and uses the current squelch as threshold. In semi-automatic mode, the user specifies the dit duration while the threshold is determined automatically.

With the new demodulator getting its samples from the `Decimator` rather than the `FFTCalcThread`, the practice of using the squelch as threshold is no longer applicable, as samples below the squelch level do not get passed to the `Demodulator`. The new implementation of the Morse demodulator thus only supports "automatic" and "manual" mode with respect to dit duration. The threshold is always determined automatically.

4.2.2.2 *Implementation*

The re-written Morse demodulator has five states that are displayed in Figure 3 and explained in the following.

INIT   In this state, the demodulator initializes its internal state. Samples received while the demodulator is in this state, are discarded and not demodulated (in case AM demodulation is activated, audio still gets demodulated).

COLLECT_SAMPLES   The demodulator calculates and stores the envelope of all received samples in a buffer. In case AM demodulation is activated, audio gets demodulated.

INIT_STATS   The collected samples from the previous state are used to initialize the threshold and, optionally, the dit duration. Samples received while the demodulator is in this state, are discarded and not demodulated (in case AM demodulation is activated, audio still gets demodulated).

DEMOD   Received samples are demodulated on-the-fly. If demodulation or decoding error rates are beyond a threshold, the demodulator re-initializes.

STOPPED   The demodulator should not be running; all samples are discarded.

For envelope detection, $\sqrt{I^2 + Q^2}$ is used, with I and Q being the inphase and quadrature components of the sample, respectively. This is computationally more expensive than using the existing AM demodulator to detect the envelope, but produces more exact results and comes without the unwanted automatic gain control of the AM demodulator. For efficient memory management, the envelope of the currently processed sample packet is not always stored in a new array. Instead, a buffer is allocated once and overwritten with the current envelope each time a new sample packet is processed. If AM demodulation is enabled, the detected envelope is copied from the buffer to the output-packet (which is passed to the `AudioSink`).

The threshold, that determines whether a given sample is a "low" or a "high" sample (i.e. if it is part of a dit/dah or a pause), is initialized based on the initialization data collected during the COLLECT_SAMPLES step and continuously updated for every processed sample. It is calculated as $\mathtt{bottom} + \frac{\mathtt{peak} - \mathtt{bottom}}{2}$, with `bottom` being the lowest envelope and `peak` being the highest envelope seen since the demodulator's initialization.

If automatic mode is enabled, the dit duration is determined by counting streaks of consecutive "high" (i.e. bigger than the threshold) samples in the initialization data. The observed streak lengths are then adjusted by adding the number of observed streaks with one sample more or less (i.e. $o_{new}(x) = o(x-1) + o(x) + o(x+1)$ with $o(x)$ being the number of occurrences of a continuous high-streak of x samples). This is done to cope with outliers. The most frequently observed streak length is assumed to be the number of samples per dit. If this number of samples corresponds to less than $7\,ms$ of samples, it is assumed that the initialization data was of poor quality and the demodulator reinitializes itself.

In DEMOD state, streaks of continuous high and low values are counted in the currently processed samples packets. The length of a streak and its value (i.e. high or low) are then decoded to find out whether it encodes a dit, dah or a short, medium or long pause. If

a medium pause (encodes the end of character symbol) is detected, the previous dits and dahs are passed to the `Decoder` class, which translates them into a character. The `Decoder` class was already used by the old demodulator, originally taken from the open-source project Morsecoder and could be recycled for the new demodulator.

To implement automatic re-initialization, the demodulator uses `ErrorBitSets`, which are essentially ring buffers for boolean values. Two `ErrorBitSets` are used; one for demodulation (i.e. translating streak lengths into dits, dahs and pauses) and one for decoding (i.e. translating dits and dahs into characters). Every entry represents a successful or unsuccessful decoding/demodulation attempt. The size of the `ErrorBitSet` that tracks demodulation success is 100, the size for the decoding success `ErrorBitSet` is 30. If the success rate drops below 50%, the demodulator re-initializes itself and re-determines the threshold and, if automatic mode is enabled, dit duration.

Listing 3 shows an excerpt of the demodulator's core method `demodulate(SamplePacket input, SamplePacket output)`. The operation of the demodulator, which was described in this section, can be recognized in it.

Listing 3: The demodulate method of the new Morse demodulator (excerpt)

```
switch (this.state) {
    case INIT:
        if (amDemod) {
            // do nothing except AM Demodulation; Demodulator
                might be in inconsistent state
            envelopeToBuffer(input);
            amDemodFromBuffer(output);
        }
        break;
    case COLLECT_SAMPLES:
        // collect samples and write them to initSamples until we
             have enough
        envelopeToBuffer(input);
        if (amDemod)
            amDemodFromBuffer(output);
        collectSamplesFromBuffer();
        if (!(initSamplesCollected < initSamplesRequiredForInit))
            initializeStats();
        break;
    case INIT_STATS:
        if (amDemod) {
            // do nothing except AM Demodulation; Demodulator
                might be in inconsistent state
            envelopeToBuffer(input);
            amDemodFromBuffer(output);
        }
        break;
    case DEMOD:
        // demodulate samples
        envelopeToBuffer(input);
```

```
        if (amDemod)
            amDemodFromBuffer(output);
        updateThresholdFromBuffer();
        binarizeBuffer();
        demodulateBuffer();
        if (automaticReinit && needsReinit())
            init();
        break;
    case STOPPED:
        // discard samples; Demodulator should not be running
}
```

A screenshot of the running Morse demodulator can be seen in Figure 4.
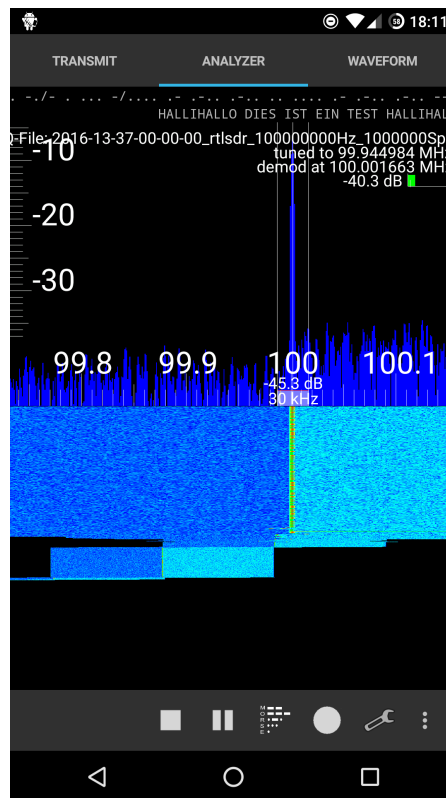
Figure 4: Screenshot of the Morse demodulator on a Nexus 5X

4.2.3 *Radio Data System*

The RDS signal is transmitted along with wide band FM radio signals to provide additional information about the radio station and program.

Demodulation of the RDS signal is first done in Octave in order to evaluate the demodulation algorithm. The octave implementation also helps by providing reference data of the different stages of demodulation.
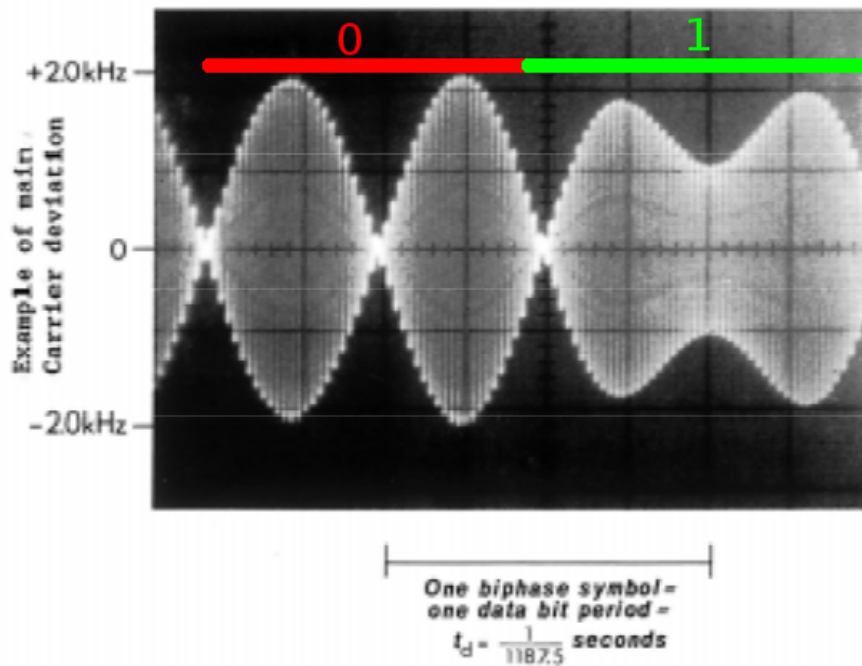
Figure 5: RDS envelope waveform after Frequency demodulation [1]

4.2.3.1   *RDS modulation scheme*

RDS uses Binary Phase Shift Keying (BPSK) with Manchester encoding. The signal is transmitted with an offset of 57 kHz relative to the center frequency of the mono audio signal (baseband). The 19 kHz pilot tone of wideband FM can therefore be used to retrieve the RDS carrier by multiplying it with itself 3 times. The complete FM spectrum can be seen in Figure 8a.

After the RDS baseband signal has been retrieved from the FM signal there are multiple ways of demodulating the BPSK modulation. A sophisticated approach tries to recover the phase synchronized RDS carrier from the signal by using e.g. a form of Phase Locked Loop (PLL) or Costas Loop. The symbols can then be extracted by multiplying the carrier with the modulated signal and apply a threshold operation to get bits.

A much simpler approach is to analyze the envelope of the signal and detect bits based on known shapes of ones and zeros in the waveform. Figure 5 shows the envelope and the pattern of symbols which can be detected.

4.2.3.2   *RDS coding scheme*

RDS frames are called groups and each group consists of 4 blocks called A, B, C and D. One block has a length of 16 bit plus a 10 bit checkword. Block A always contains the Program Identifier (PI)
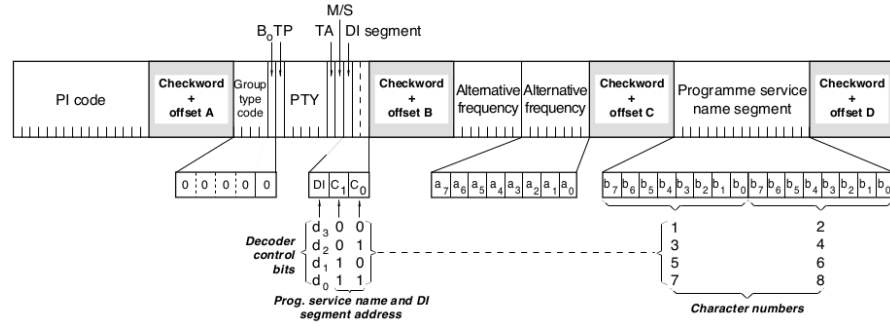
Figure 6: Coding scheme of RDS: group 0A [1]

which identifies the radio station. The content of the other blocks depends on the group type which is located in block B (see Figure 6).

Table 5 lists all group types and their descriptions. The RDS demodulation in AnSiAn only decodes types 0 and 2 because they contain the basic information which is also often displayed on the radio receiver.

4.2.3.3   *Evaluation in Octave*

Developing a signal processing application on Android has many drawbacks. One issue is that it is very hard to debug the actual signal processing components because of the lack of proper tools to visualize and analyze the data that is being processed. It is also not possible to do rapid prototyping without sufficient signal processing libraries available. Therefore, the RDS demodulator was first developed in Octave and afterwards ported to Android.

For development and testing it is better to work on recorded samples instead of live captures. This makes tests reproducible and simplify the development environment. The file was recorded using the record feature of RF Analyzer. It can be imported to Octave by using the `read_cuchar_binary()` script provided by the GNU Radio project. After each step the produced output data can be written back to an *IQ* file in order to use it in the Android application. This way it is possible to develop each component of the demodulation process separately and the output can be visualized on the developing machine.

The demodulation is done in the following steps:

1. Downmixing the radio signal to baseband and filter it (see Figure 7).

2. FM demodulation (see Figure 8a).

3. Downmixing the RDS signal to baseband and filter it (see Figure 8 b and c).
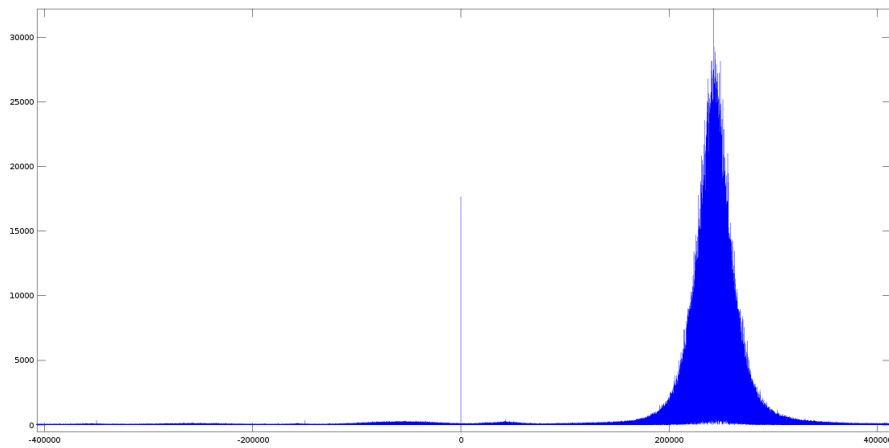
(a) Spectrum of the captured signal
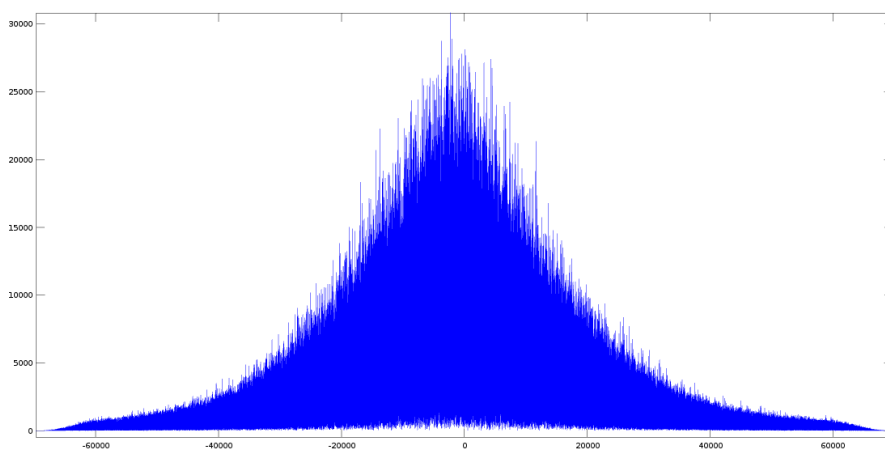
(b) Spectrum after downmixing and filtering

Figure 7: FM Modulated Signal

4. Take the absolute value of the signal to get the envelope that was shown above (see Figure 9).

5. Find the beginning of a symbol by searching for a minimum in the waveform. From there find the end of the symbol with the same strategy. Now determine whether the symbol is a one or a zero according to the value of the minimum found in the middle of the sample compared to its peaks.

The octave code used to execute the steps mentioned above is shown in the listing below:

Listing 4: Octave implementation of the RDS demodulator

```
signal = read_cuchar_binary ("rtlsdr_100550000Hz_1000000Sps.iq");
t = linspace(0, length(signal)/1000000, length(signal))';
carrier = e.^(2*pi*-245000*t*i);
down = carrier .* signal;
fl = fir1(300, 100000/1000000*2);
```
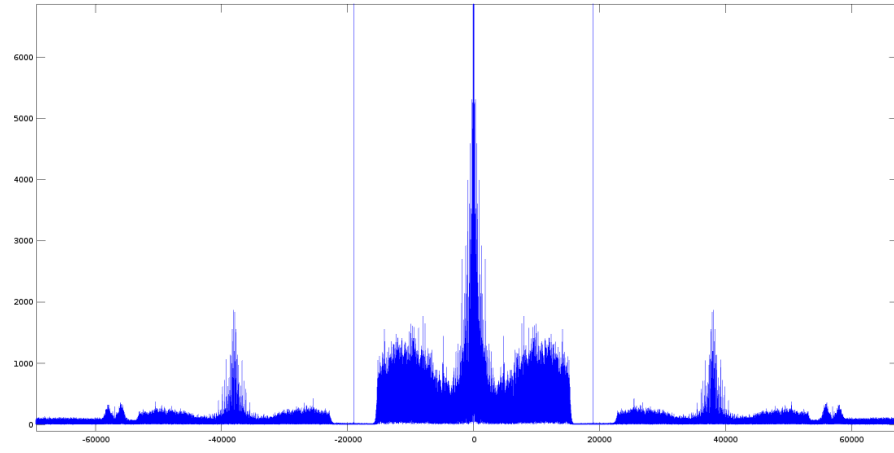
(a) Signal spectrum after FM demodulation
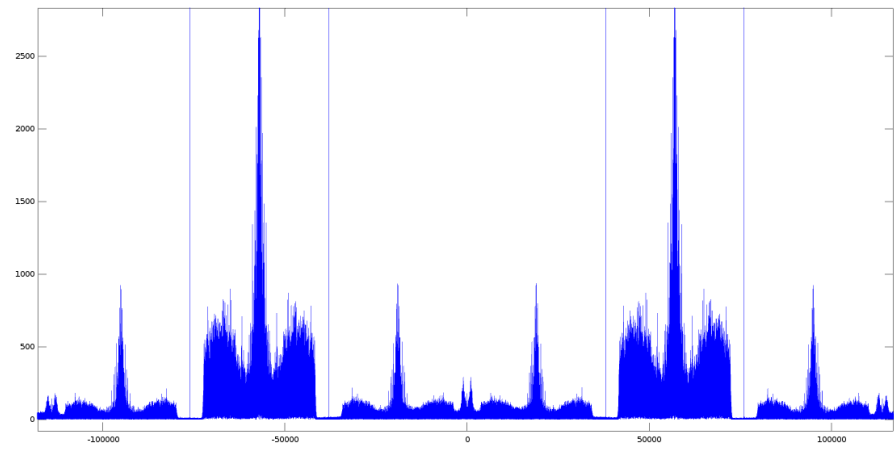
(b) RDS baseband spectrum after downmixing
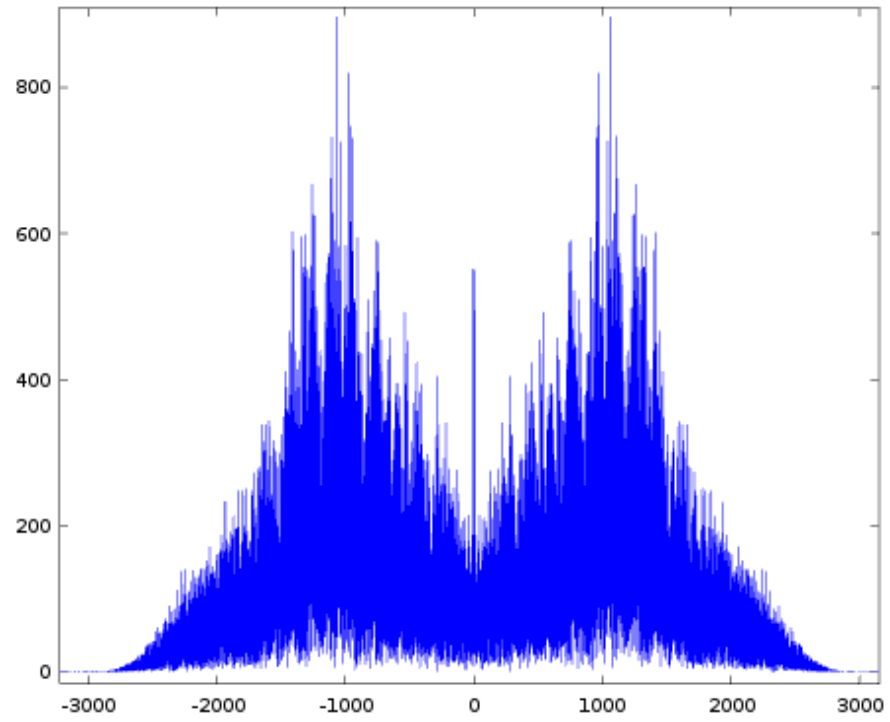
(c) RDS baseband spectrum after filtering

Figure 8: Extracting the RDS signal from the FM signal

Figure 9: RDS waveform after take the absolute values

```
filtered = filter(fl, 1, down);
demod = quad_demod(filtered, 1);
t2 = linspace(0, length(demod)/1000000, length(demod));
rdscarrier = cos(2*pi*-57000*t2)';
rdsbase = demod(1:length(rdscarrier)) .* rdscarrier ;
frds = fir1(300, 2400/1000000*2);
rdsbase_filtered = filter(frds,1,rdsbase);
downsampled = decimate(rdsbase_filtered, 16).*80;
write_cuchar_binary (downsampled, "rds_baseband_62500sps.iq");
bits = rds_bpsk_demodulate(downsampled, 62500);
rds_decode(bits)
```
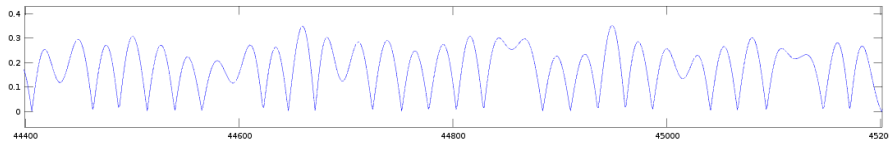
The `quad_demod()` function does the quadrature demodulation (FM demodulation). The `rds_bpsk_demodulate()` function is shown in the following listing:

Listing 5: Octave implementation of the BPSK demodulation

```
function demod = rds_bpsk_demodulate(signal, fs)
  samples_per_symbol = fs/1187.5
  samples_per_symbol = ceil(samples_per_symbol)
  envelope = abs(signal);

  % Find the first minimum
  [minimum, idx1] = min(envelope(1:samples_per_symbol))

  bits = [];
  while (idx1 + samples_per_symbol*2 < length(envelope))
    % find end of symbol idx2 (minimum near idx1 +
        samples_per_symbol)
    from = round(idx1+samples_per_symbol*0.75);
    to   = round(idx1+samples_per_symbol*1.25);
    [minimum, idx2] = min(envelope(from:to));
    idx2 = idx2 + from;

    % calc mean of all samples between idx1 and idx2 and calc
        threshold = mean/2
    m = mean(envelope(idx1:idx2));
    threshold = m/2;

    % get minimum sample in the middle between idx1 and idx2 ...
    span = idx2 - idx1;
    from = round((idx1+idx2)*0.5 - 0.25*span);
    to   = round((idx1+idx2)*0.5 + 0.25*span);
    [minimum, idxmiddle] = min(envelope(from:to));
    idxmiddle = idxmiddle + from;
```

```
% Check whether we have the correct timing. It might be, that
    idx2 is
% actually in the middle of a symbol than at its end.
if (envelope(idx2) > threshold)
  % In this case we find the minimum between idx1 and idx2
      and set it
  % as idx1 for the next round:
  %printf("WARNING: Wrong timing. thres=%f < envelope(idx2=%d
      )=%f\n",threshold,idx2,envelope(idx2));
  idx1 = idxmiddle;
  continue;
endif

% ... and check it against the threshold
s = envelope(idxmiddle);
if (s > threshold)
  bits = [bits 1];
else
  bits = [bits 0];
endif

% idx1 = idx2 and continue with the next symbol..
idx1 = idx2;

endwhile
demod = bits;
```

4.2.3.4 *Android Implementation*

For the Android implementation two classes are added to the AnSiAn codebase:

- BPSK: This class handles the BPSK demodulation and can be reused by other demodulators using the BPSK modulation scheme (e.g. PSK31).

- RDS: This class integrates in the existing FM class for frequency demodulation. It handles the decoding and processing of RDS groups.

A screenshot of the application demodulating the RDS signal of the *Antenne Frankfurt* station is shown in Figure 10.

4.2.4 *PSK31*

PSK31 refers to phase shift keying modulation using a baud rate of 31.25 Hz. It was developed by Peter Martinez in 1998 to introduce a narrow bandwidth digital mode for live chatting. Because of the simple and efficient modulation and coding scheme (e.g. it does not
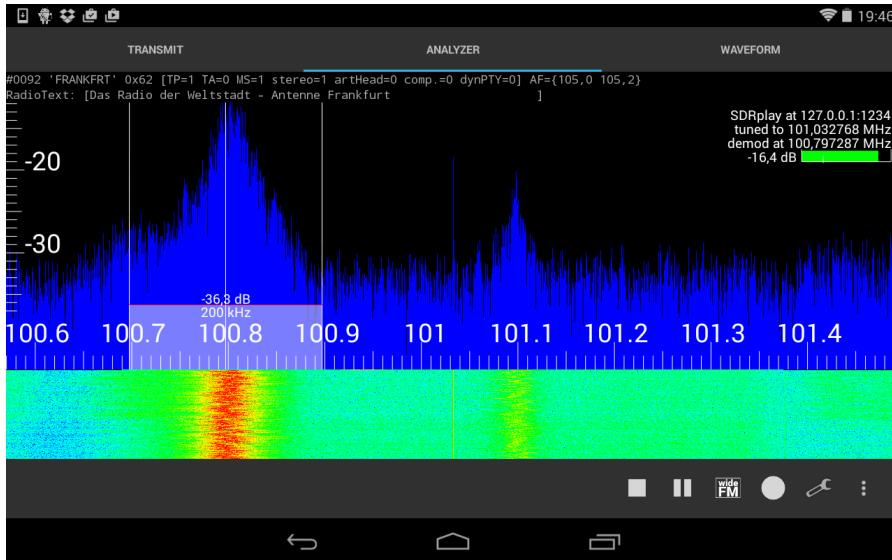
Figure 10: Screenshot of the RDS demodulator on a Nexus 7

have an error correction mechanism) it is very widespread amongst amateur radio operators.

A typical setup for PSK31 operation is a Single Side Band (SSB) transceiver connected to the sound card of a computer. The audio channel that is fed into the computer can contain multiple parallel PSK31 transmissions, which are demodulated using a special software.

PSK31 is usually operated in USB mode and shall therefore be available in AnSiAn when the USB demodulator is active. As with the RDS demodulator the algorithm to demodulate and decode PSK31 is first evaluated in Octave and later ported to the AnSiAn application.

4.2.4.1 *PSK31 modulation scheme*

PSK31 in its basic form uses BPSK to transmit binary information by sending a single side band signal with either a 180 degree phase shift (digital '1') or without a phase shift (digital '0'). Additionally, a root raised cosine filter is used in order to smooth the phase shift and therefore keep the bandwidth narrow. Each symbol contains information about one bit and is always 32 ms long.

Figure 11 shows the modulated PSK31 signal. As already explained in Section 4.2.3.1, the signal may be demodulated by using a second order loop (e.g. a Costas loop) in order to recover the phase information and correct for small frequency variations. However, the simple approach using envelope detection is also possible. The BPSK code used for RDS demodulation can be partly reused and only needs some modifications as PSK31 does not use Manchester Encoding.
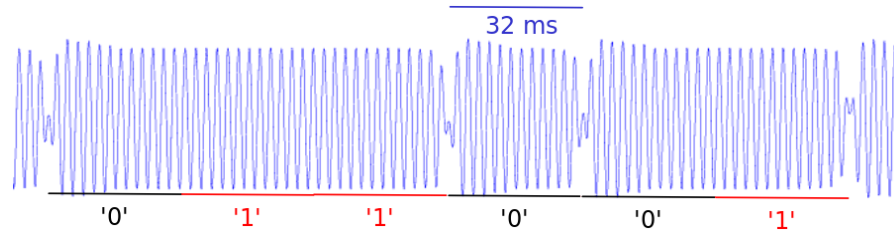
Figure 11: PSK31 modulation scheme: Each symbol is 32ms long. A 180°
phase shift indicates a '0', no phase shift indicates a '1'

4.2.4.2 *PSK31 coding scheme*

PSK31 uses a variable length encoding called *Varicode* which assigns
frequently used characters a shorter code similar to Morse. Characters
are separated by two consecutive zeros. Table 4 in the appendix lists
all characters and their varicode encodings.

4.2.4.3 *Evaluation in Octave*

The demodulation algorithm in octave comprises the following steps:

1. Downmixing the USB signal to baseband.

2. USB demodulation (complex bandpass filter to get only the up-
   per side band). See Figure 12 b.

3. Downsampling in order to reduce the workload for the PSK31
   demodulation.

4. Envelope detection. See Figure 13.

5. Find the beginning of a '0' symbol by searching for a minimum
   in the envelope of the signal. Determine whether the next sym-
   bol is a '0' (another minimum is found around 32ms from the
   beginning of the current symbol) or a '1' (no minimum at the
   beginning of the next symbol). If another '0' was detected, the
   timing can be corrected by centering the minimum at the begin-
   ning of the signal and search on from there.

The octave code used to execute the steps mentioned above is shown
in the listing below:

Listing 6: Octave implementation of the PSK31 demodulator

```
% load signal
signal = read_cuchar_binary ("rtlsdr_434570917Hz_1000000Sps.iq");
fs = 1000000;

% downmixing (signal is at ~ 55 Khz)
fc = 55000;
t = linspace(0, length(signal)/fs, length(signal))';
```
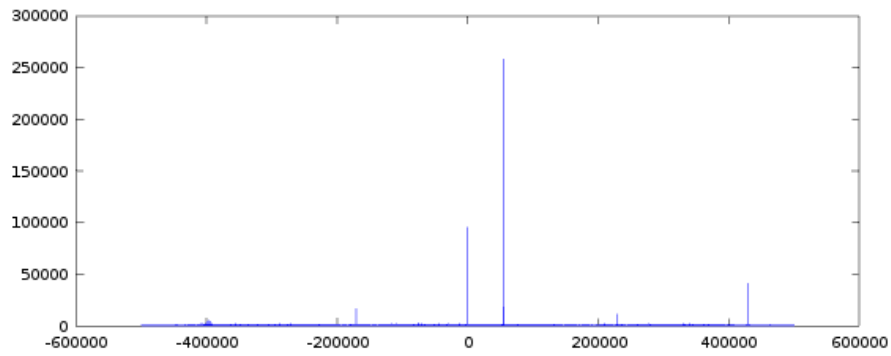
(a) Raw spectrum. The PSK31 signal is at around 55 KHz.
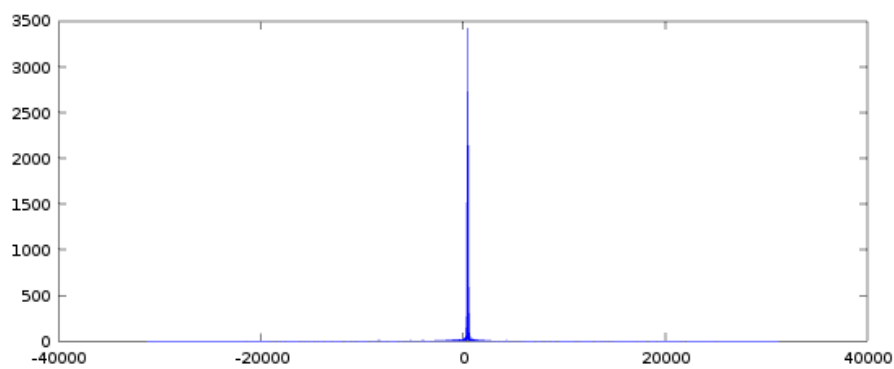
(b) Spectrum after downmixing and filtering (USB demodulation)

Figure 12: PSK31 signal in the frequency domain (spectrum)
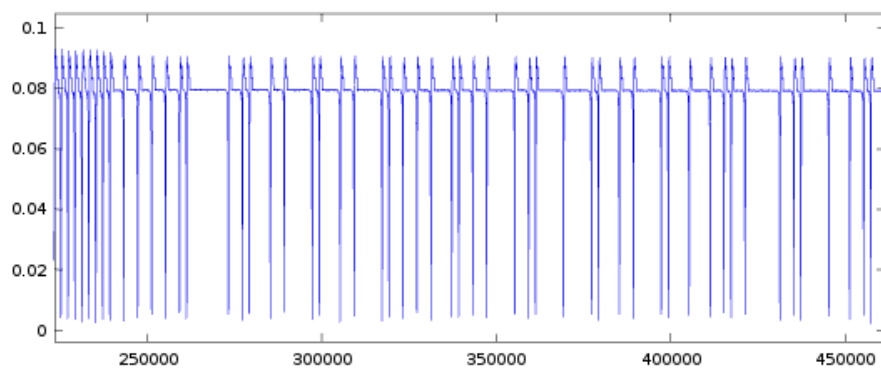
Figure 13: Envelope of the downsampled, USB demodulated signal.

```octave
carrier = e.^(2*pi*-fc*t*i);
down = carrier .* signal;

% USB demod (bandpass filter [200Hz-1000Hz])
bpfilter = firls(300, [0 200 200 1000 1000 fs/2]./(fs/2), [0 0 0
    1 0 0]);
filtered = filter(bpfilter, 1, down);

% downsampling
downsampled = decimate(filtered, 16);

% BPSK demodulation
bits = psk31_bpsk_demodulate(downsampled, fs/16);

% Varicode decoding
psk31_decode(bits)
```

The `psk31_bpsk_demodulate()` function is shown in the following listing:

Listing 7: Octave implementation of the BPSK demodulation

```octave
function demod = psk31_bpsk_demodulate(signal, fs)
  % PSK31 baud rate is 31.25 Hz
  samples_per_symbol = fs/31.25
  samples_per_symbol = ceil(samples_per_symbol)
  envelope = abs(signal);

  % Find the first minimum
  [minimum, idx1] = min(envelope(1:samples_per_symbol))

  bits = [];
  while (idx1 + samples_per_symbol*2 < length(envelope))
    % search for a minimum at the position of the next sample
    % (minimum near idx1 + samples_per_symbol)
    from = round(idx1+samples_per_symbol*0.5);
    to   = round(idx1+samples_per_symbol*1.5);
    [minimum, idx2] = min(envelope(from:to));
    idx2 = idx2 + from;

    % calc mean of all samples between idx1 and idx2 and calc
        threshold = mean/2
    m = mean(envelope(idx1:idx2));
    threshold = m/2;

    % Check whether we have a minimum (->0) or not (->1).
    if (envelope(idx2) > threshold)
      % In this case we have a bit 1.
      bits = [bits 1];
      idx1 = idx1 + samples_per_symbol;
    else
      % In this case we have a bit 0.
      bits = [bits 0];
```
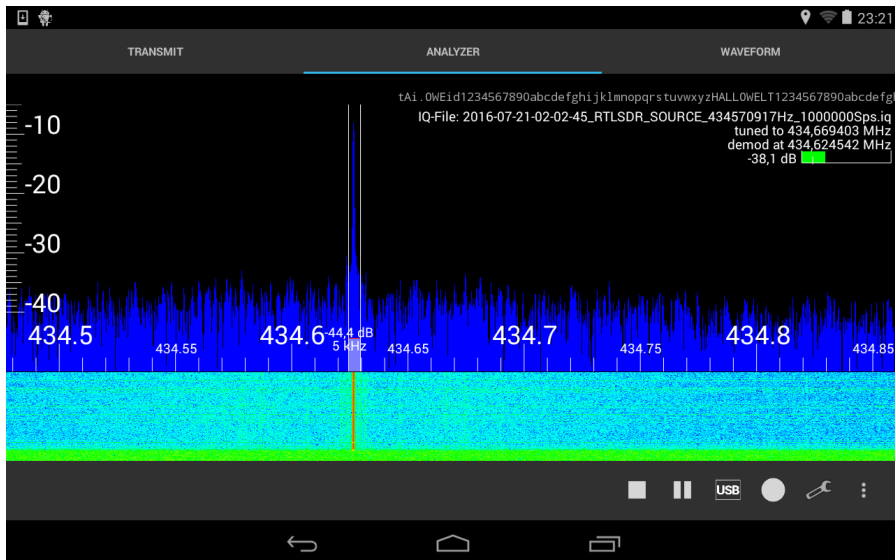
Figure 14: Screenshot of the PSK31 demodulator on a Nexus 7

```
    idx1 = idx2;
  endif
endwhile
demod = bits;
```

4.2.4.4 *Android Implementation*

For the Android implementation the BPSK class is modified to handle both BPSK variations needed in AnSiAn:

- RDS: BPSK with Manchester Encoding

- PSK31: BPSK without Manchester Encoding

Additionally, the class PSK31 is added which is a subclass of Demodulation. It integrates with the existing USB class to provide PSK31 demodulation when USB demodulation is active.

A screenshot of the application demodulating a PSK31 signal is shown in Figure 14.

4.2.5 *Logging*

As explained in Section 4.2.1, digital demodulators use the EventBus library to pass demodulated text and other information to the user interface. In order to not only display but also persist this data, the class Logger listens to DemodInfoEvents from the modulators and, if logging is enabled in the preferences by the user, writes them to logfiles.

DemodInfoEvent has an attribute position, which can be either TOP or BOTTOM, depending on whether the contained text is to be displayed

on the upper or the lower line of `DemodulationInfoView`. Accordingly, the `Logger` maintains one logfile for the upper and the lower line. Every `DemodInfoEvent` is logged to one of these files depending on their `position` attribute. The paths of the logfiles can be defined in the Demod preferences.

4.3   SUPPORT FOR NEW SDR PLATFORMS

AnSiAn supports different SDR hardware such as:

- RTL-SDR dongle

- HackRF

The architecture of the app allows to easily add support for additional hardware, as long as it provides IQ data samples and an Android driver is available.

4.3.1   *rad1o*

The *rad1o* is a SDR platform with the same design as the *HackRF* that was used as badge for the *Chaos Communication Camp* 2015. Some modifications to the design were made in order to reduce costs and therefore, minor differences exist compared to a HackRF (e.g. the option to activate the antenna power is missing for the *rad1o*).

The *rad1o* offers the same USB interface as the *HackRF* and can be used with the same driver, which is *hackrf_android*. The driver only needs two minor modifications:

1. If the driver detects a rad1o, the option to enable the antenna power is disabled.

2. Because the *rad1o* has a different USB product ID, the driver has to be extended in order to also detect a *rad1o* as valid *HackRF*.

Both changes have been applied to the *hackrf_android* driver. Unfortunately, the creators of AnSiAn have replaced the driver library with a plain copy of the driver source code, which makes it hard to update it to a newer version. Therefore, the driver code was removed from AnSiAn and replaced again with the current version of the driver library.

This is the only change necessary to add support for the new SDR hardware. The *rad1o* can now be used by selecting the *HackRF* source in AnSiAn.

4.3.2   *SDRPlay*

The *SDRplay* is a low cost SDR with a frequency range between 100 kHz and 2 GHz. It has a 12-bit Analog-to-Digital Converter (ADC)

Command name	Code	Description
SET_FREQ	0x01	Change the tuning (center) frequency of the dongle
SET_SAMPLE_RATE	0x02	Change the sample rate of the dongle
SET_GAIN_MODE	0x03	Change gain mode (auto or manual)
SET_GAIN	0x04	Change gain value (if gain mode is manual)
SET_FREQ_CORRECTION	0x05	Change frequency correction value (in ppm)
SET_IF_TUNER_GAIN	0x06	Change Intermediate Frequency (IF) gain
SET_TEST_MODE	0x07	Turn on test mode
SET_AGC_MODE	0x08	Activate or deactivate Automatic Gain Control (AGC)
SET_DIRECT_SAMPLING	0x09	Activate or deactivate direct sampling mode
SET_OFFSET_TUNING	0x0a	Activate or deactivate offset tuning
EXIT	0x7e	Cause the driver to turn off itself
GAIN_BY_PERCENTAGE	0x7f	Change gain value as percentage
ENABLE_16_BIT_SIGNED	0x80	Enable 16 bit unsigned sample size (SDRplay only)

Table 1: Commands for the RTL-SDR / SDRplay driver by Martin Marinov

and therefore suffers less from quantification noise than the *HackRF* and the *RTL-SDR* (both having an 8-bit ADC). Additionally, it provides passive Radio Frequency (RF) filters on the front end to prevent strong out-of-band noise from distorting the input signal.

Martin Marinov developed an Android driver for the *SDRplay* dongle, which is currently in beta state. The API is similar to the *RTL-SDR* driver which is developed by the same person.

4.3.2.1 *Implementation*

The *SDRplay* driver is started by an Intend with an URL of the format `'iqsrc://...'`. It takes an IP address and TCP port as arguments and will open a socket respectively. The host app can the connect to the socket and will receive the samples via TCP. Control commands such as changing the frequency, sample rate or other parameters can be sent through the TCP socket to the driver. A list of supported commands can be found on the GitHub page of the driver. They are listed in Table 1.

4.3.2.2 *Open Issues*

As the *SDRplay* Android driver is still in beta stage, the support in AnSiAn has also still beta status. 12-bit samples are currently not working correctly and therefore AnSiAn uses the RTL-SDR compatibility mode of the driver which delivers stripped 8-bit samples. However,

the 12-bit converter code is ready to be used and might be enabled in future versions of AnSiAn.

4.4 TRANSMISSION

RF Analyzer as well as AnSiAn both have a receive-only signal processing chain, even though they support SDR devices which are capable of transmitting signals, such as the *HackRF*. This section covers the design and (partial) implementation of a transmission chain to make use of supported devices' transmission capabilities.

4.4.1 *Design*

The implementation of a complete transmission chain is split into multiple steps to go along with the Agile approach (see Figure 15).

In the first step the transmission chain will only include the `IQSink` which is able to replay a recorded IQ file. The file has to have the correct format according to the SDR hardware and the sample rate cannot be adjusted.

The second step will provide `Modulators` for some digital and analog modes along with a dummy transmission chain which will use the modulator to create an IQ file and feed it to the `IQSink` directly.

In the final step the dummy chain is replaced by the complete transmission chain which includes:

- the `Modulators` from step two

- the `Interpolator` which is able to adjust the sample rate at the output of the `Modulators` to a value that is supported by the `IQSink`

- a `float-to-binary` converter which will output the samples in the correct binary format according to the SDR hardware

After the final step the transmission chain will be able to modulate and transmit signals in real-time, assuming the Android device is capable of the necessary calculations.

Step one and two were implemented during the third sprint and provide a proof-of-concept implementation to demonstrate the transmission capabilities of the app. Step three is left for future work on the application as well as the addition of more modulators.

4.4.2 *Transmission of Raw I/Q Files*

RF Analyzer and AnSiAn both use the open-source *HackRF* driver from Dennis Mantz[3]. This driver was released with a simple demo application that allowed the user to record signals to files and retransmit them. The logic of the `IQSink` is based on code from the demo
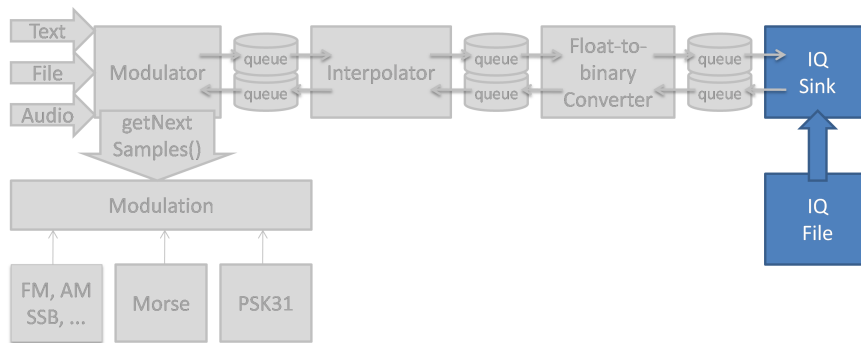
(a) Step 1: Raw IQ File Transmission
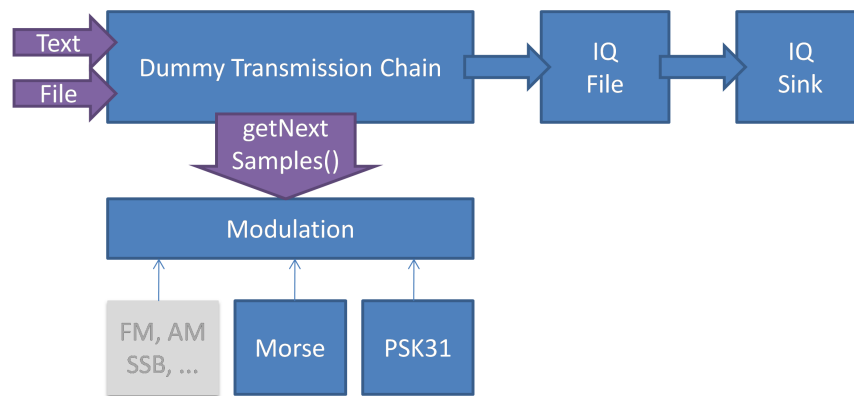
(b) Step 2: Modulated data is written to an IQ file and send afterwards
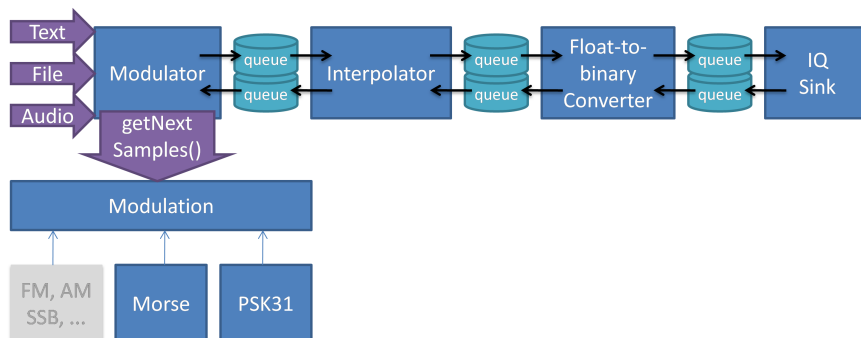
(c) Step 3: Real-time modulation, interpolation, conversion and transmission

Figure 15: Incremental implementation steps of the transmission chain

application that was adapted to AnSiAn's Architecture and extended to work e.g. with the EventBus library.

For efficiency, the *HackRF* driver uses asynchronous calls and cycling buffers. The calling program gets assigned free buffers from the driver's queue as they become available and fills them with the data that is to be transmitted. Unfortunately, the driver does not offer any possibility to register a callback function that is called when the driver has finished transmitting all queued data.

A drawback of the current implementation of `IQSink` is that it transmits noise for several hundred milliseconds after the driver has emptied its transmit buffer. This is because the driver apparently takes some time to realize that it sent all data and can stop transmitting. An alternative would be for the program to explicitly tell the driver to stop sending. As the program has no way of knowing when the driver finishes sending, this would either mean stopping the transmission early or having to estimate the remaining transmission time based on the sample rate and queue length. Experiments with the latter did, however, not yield satisfying results.

4.4.3   *Modulators*

Each `Modulator` will provide an interface to retrieve the next packet of modulated samples. The payload data (e.g. audio or text) might be given at instantiation time or through a queue (in order to enable real-time transmission).

As mentioned above, the step two implementation will include a dummy transmission chain that reads the output of the `Modulator`, converts it to binary IQ data and writes it to a temporary file which will then be transmitted by the `Transmitter` implemented in the previous section.

4.4.3.1   *Morse Modulator*

The Morse modulator takes a string as input payload and produces a baseband signal with dits and dahs respectively. The Morse encoder from the previous AnSiAn version can be reused to transform the payload text into a sequence of dits ('.'), dahs ('-'), breaks ('␣') and word boundaries ('/'). Each character in this sequence corresponds to a predefined packet of IQ samples (see Table 2).

Because the dit and dahs each also contains silence of the length of one dit at the beginning, the letter separator ('␣') and the word separator ('/') need to be one dit length shorter than they actually are.

To increase the modulation performance, the complex sinusoid is generated ahead of time and the character translation only consists of a copy operation.

Morse element	Corresponding IQ sample structure
.	one dit length of silence + one dit length of tone
-	one dit length of silence + three dit lengths of tone
␣	two dit lengths of silence
/	six dit lengths of silence

Table 2: Translation from Morse elements to IQ samples

Previous state (phase)	Current bit	BPSK output
0°	0	$\cos(\frac{\pi \cdot 31.25\text{Hz} \cdot t}{f_s})$
180°	0	$-\cos(\frac{\pi \cdot 31.25\text{Hz} \cdot t}{f_s})$
0°	1	1
180°	1	-1

Table 3: Output of the BPSK modulation based on its previous state

4.4.3.2 *PSK31 Modulator*

The `PSK31` modulator also takes a string as input payload, encodes it with Varicode and modulates the bits with BPSK. The Varicode dictionary of the PSK31 demodulator can be reused to perform the first step of creating the bitstream from the payload text. The BPSK modulation simply outputs the samples based on the current state (phase) of the modulator (see Table 3).

4.5 GUI

In order to expose the newly implemented features and their parameters to the user, changes to the user interface of AnSiAn were necessary. Section 4.5.1 describes the changes made in the preferences menu and Section 4.5.2 covers the GUI of the new Transmit Tab. addresses a change in the frequency selection dialog that was made to avoid unexpected behavior.

4.5.1 *Preferences*

AnSiAn features a preference menu with four tabs that can be seen in Figure 16. In the original AnSiAn, one of these tabs is the Morse tab that holds options for Morse demodulation and audio generation. As the new version of AnSiAn replaces Morse audio generation with actual modulation and transmit capability and supports RDS and PSK31 as additional digital demodulation schemes, it was decided to refactor the Morse preference tab into a general Demod

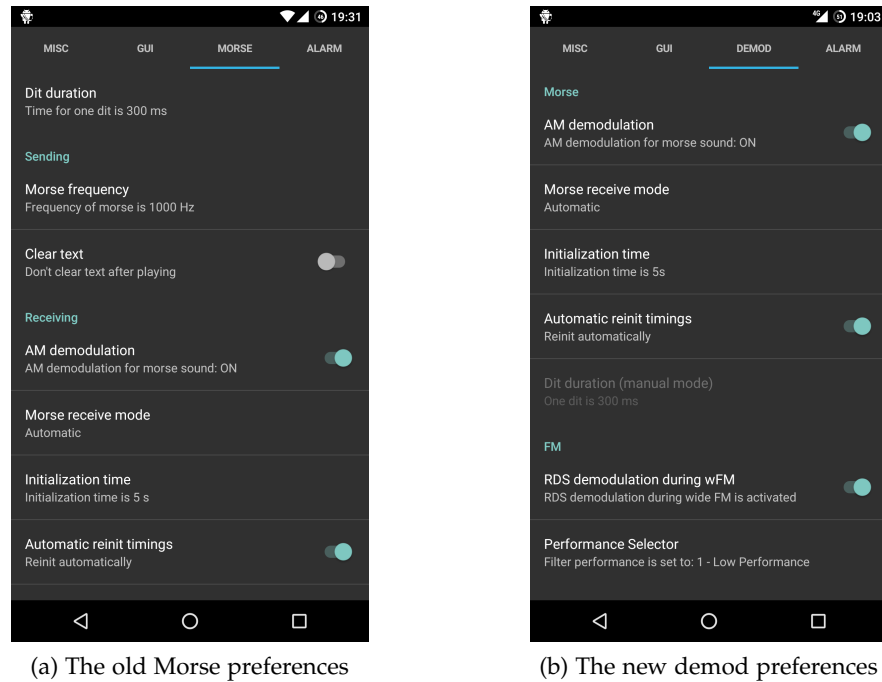(a) The old Morse preferences   (b) The new demod preferences

Figure 16: Screenshot of the preference menu on a Nexus 5X

preference tab while bundling transmission preferences for all available transmission modes in the Transmit Tab on the main screen (see Section 4.5.2).

As can be seen in Figure 16, the Demod preference tab holds preferences for various demodulation modes. It also offers a switch to turn on PSK31 demodulation on using the {acUSB demodulator as well as options to turn on logging of demodulated data and to specify the respective logfiles (both not in the screenshot).

4.5.2 *Transmit Tab*

The original AnSiAn had a Morse tab that offered information about the state of the Morse demodulator, the possibility to manually reinitialize the demodulator and enabled the user to generate and play Morse audio from text. It can be seen in .

As the newly implemented transmission and modulation feature described in Section 4.4 needed a graphical interface, it was decided to refactor the old Morse tab into a Transmit tab that gives the user access to the transmission features. It can be seen in Figure 17b and provides options for the HackRF's features, such as amplification and Variable-Gain Amplifier (VGA) gain. The spinner on the top lets the user select a transmit mode. Available modes are Raw IQ File, Morse and PSK31. Fields are hidden, shown or disabled based on the selected mode, i.e. the Morse-specific options for dit duration and Morse frequency are hidden on selecting PSK31.
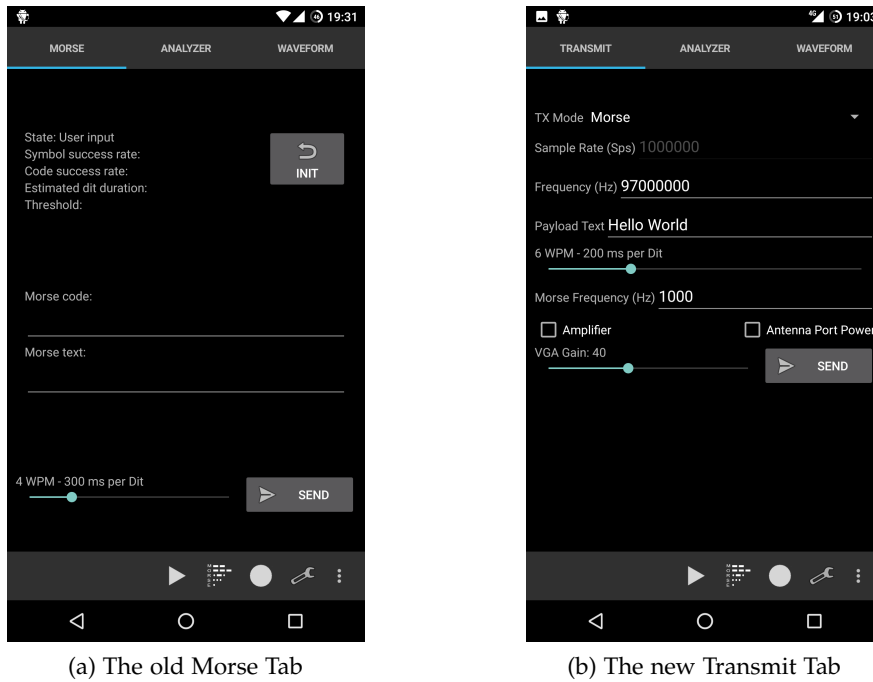
| (a) The old Morse Tab | (b) The new Transmit Tab |

Figure 17: Screenshot of the Morse/Transmit Tab on a Nexus 5X
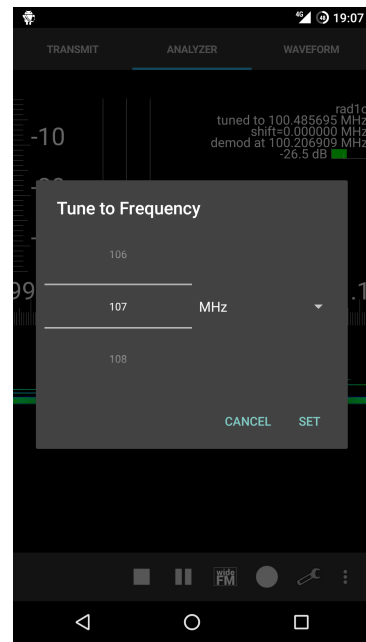
4.5.3 *Frequency Selection Dialog*

By clicking the menu on the bottom right and then "Set Frequency", AnSiAn enables the user to quickly tune to a desired frequency without having to scroll through the spectrum. In the original version, this dialog also offers the user to optionally set the sampling rate (called bandwidth in the dialog) to an arbitrary value. By default, this option is enabled and the corresponding field is filled with a value that is not the current sampling rate. A user who only wants to jump to another frequency might therefore inadvertently change the sampling rate.

This has serious consequences, as the demodulation chain can currently only process sampling rates of $1\,Msamples/s$. For other sampling rates, the `Decimator` stops working and does not provide the `Demodulator` with samples any more. The user does not get any visual feedback on this and can only observe that audio and text demodulation stop working for no apparent reason. As patching the decimator to efficiently support other sampling rates would be a major task, it was decided remove the possibility to change the sample rate in order to prevent users from inadvertently breaking demodulation.

The old and the new frequency selection dialogs can be seen in Figure 18.
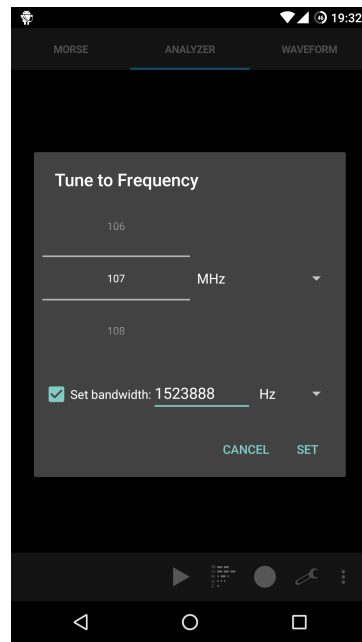
(a) The old frequency selection dialog

(b) The new frequency selection dialog

Figure 18: Screenshot of the frequency selection dialog on a Nexus 5X

5

CONCLUSION

AnSiAn as fork of RF Analyzer has made huge structural changes to the application architecture in order to implement a clean MVC pattern. Although this has improved the code quality and maintainability it also introduced issues as described in Section 4.1. These issues and their impact on other parts of the app (see Section 4.2.2) had to be addressed before implementing the new features listed in Section 2.1. There were also additional features such as support for the SDRplay hardware or PSK31 transmission, that were added after the initial feature list was created and therefore influenced the time schedule.

Nevertheless, all mandatory features apart from some minor exceptions were successfully implemented. This includes new digital demodulation modes (PSK31 and RDS), logging of demodulator outputs, support for the rad10 badge and transmission functionality along with some basic transmission modes (Morse and PSK31) as well as plain IQ file replay. Not implemented was the FM-modulated transmission of audio files and the optional features: Walky-Talky-Mode and packet radio demodulation.

Some open tasks and improvements remain to be done in future work on the application. These include the implementation of the complete transmission chain as outlined in Section 4.4, additional modulation modes and improvements on the BPSK demodulation algorithm by switching to a PLL-based approach.

APPENDIX

[h] Character	Encoding	Character	Encoding
NUL	1010101011	!	111111111
SOH	1011011011	"	101011111
STX	1011101101	#	111110101
ETX	1101110111	$	111011011
EOT	1011101011	%	1011010101
ENQ	1101011111	&	1010111011
ACK	1011101111	'	101111111
BEL	1011111101	(	11111011
BS	1011111111	)	11110111
HT	11101111	*	101101111
LF	11101	+	111011111
VT	1101101111	,	1110101
FF	1011011101	-	110101
CR	11111	.	1010111
SO	1101110101	/	110101111
SI	1110101011	0	10110111
DLE	1011110111	1	10111101
DC1	1011110101	2	11101101
DC2	1110101101	3	11111111
DC3	1110101111	4	101110111
DC4	1101011011	5	101011011
NAK	1101101011	6	101101011
SYN	1101101101	7	110101101
ETB	1101010111	8	110101011
CAN	1101111011	9	110110111
EM	1101111101	:	11110101
SUB	1110110111	;	110111101
ESC	1101010101	<	111101101
FS	1101011101	=	1010101
GS	1110111011	>	111010111
RS	1011111011	?	1010101111
US	1101111111	@	1010111101

SP	1		A	1111101
B	11101011		c	101111
C	10101101		d	101101
D	10110101		e	11
E	1110111		f	111101
F	11011011		g	1011011
G	11111101		h	101011
H	101010101		i	1101
I	1111111		j	111101011
J	111111101		k	10111111
K	101111101		l	11011
L	11010111		m	111011
M	10111011		n	1111
N	11011101		o	111
O	10101011		p	111111
P	11010101		q	110111111
Q	111011101		r	10101
R	10101111		s	10111
S	1101111		t	101
T	1101101		u	110111
U	101010111		v	1111011
V	110110101		w	1101011
W	101011101		x	11011111
X	101110101		y	1011101
Y	101111011		z	111010101
Z	1010101101		{	1010110111
[	111110111		|	110111011
\	111101111		}	1010110101
]	111111011		~	1011010111
^	1010111111		DEL	1110110101
_	101101101		a	1011
'	1011011111		b	1011111

Table 4: Varicode Table

Group Type	Group Version	Description
0	A	Basic tuning and switching information only
0	B	Basic tuning and switching information only
1	A	Programme Item Number and slow labelling codes
1	B	Programme Item Number
2	A	RadioText only
2	B	RadioText only
3	A	Applications Identification for ODA only
3	B	Open Data Applications
4	A	Clock-time and date only
4	B	Open Data Applications
5	A	Transparent Data Channels
5	B	Transparent Data Channels
6	A	In House applications or ODA
6	B	In House applications or ODA
7	A	Radio Paging or ODA
7	B	Open Data Applications
8	A	Traffic Message Channel or ODA
8	B	Open Data Applications
9	A	Emergency Warning System or ODA
9	B	Open Data Applications
10	A	Programme Type Name
10	B	Open Data Applications
11	A	Open Data Applications
11	B	Open Data Applications
12	A	Open Data Applications
12	B	Open Data Applications
13	A	Enhanced Radio Paging or ODA
13	B	Open Data Applications
14	A	Enhanced Other Networks information only
14	B	Enhanced Other Networks information only
15	A	Defined in RBDS [15] only
15	B	Fast switching information only

Table 5: RDS Group Types

BIBLIOGRAPHY

[1] RDS Forum Office. "The new RDS IEC 62106:1999 standard." In: (1999).

[2] *demantz/RFAnalyzer: Spectrum Analyzer for Android using the HackRF.* https://github.com/demantz/RFAnalyzer.

[3] *demantz/hackrf_android: A ported version of Michael Ossmann's libhackrf library to use with Android 3.1+.* https://github.com/demantz/hackrf_android.