TECHNISCHE
UNIVERSITÄT
DARMSTADT

ANSIAN - ANDROID SIGNAL ANALYZER

MATTHIAS KANNWISCHER

Secure Mobile Networking Project Documentation

March 16, 2017

Secure Mobile Networking Lab
Department of Computer Science

AnSiAn - Android Signal Analyzer
Secure Mobile Networking Project Documentation

Submitted by Matthias Kannwischer
Date of submission: March 16, 2017

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

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

During the last years multiple low cost software-defined radios (SDRs) capable of transmitting and receiving a very large frequency range have been released. These devices like the HackRF One make signal processing experiments feasible for a large user group. AnSiAn together with a HackRF driver enables signal analysis and synthesis on Android smartphones.

In this project the Android Signal Analyzer (AnSiAn) application is extended, such that it provides more features for amateur radio communications. This includes speech signals using single sideband (SSB) modulation, images using slow-scan television (SSTV) and text using Morse and PSK31. Additionally we implement the modulation of radio data system (RDS) signals.

With the new features added, AnSiAn is a comprehensive amateur radio application that is easy to use, portable and extensible for more complex communication protocols.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

SDR software-defined radio

AnSiAn Android Signal Analyzer

RDS radio data system

BPSK binary phase-shift keying

QAM quadrature amplitude modulation

FM frequency modulation

SSB single sideband

SSTV slow-scan television

PSK phase shift keying

ATV amateur television

FSTV fast-scan television

FFT fast Fourier transformation

1 INTRODUCTION

AnSiAn is an Android application that has been initially developed by Steffen Kreis and Markus Grau at the Secure Mobile Network Lab (SEEMOO) in 2015 and was based on the RFAnalyzer for Android [7] of Dennis Mantz. AnSiAn has been extended and improved by Dennis Mantz and Max Engelhardt in summer 2016. The application is designed to allow the user the explore, capture, demodulate and decode radio frequency signals on a broad range of frequencies. It supports low cost receive-only RTL-SDR dongles, but also more advanced devices like HackRF One, rad1o and SDR Play which are capable of receiving and transmitting signals.

1.1 MOTIVATION

This project further extends and improves the current implementation of AnSiAn. Since some of the supported devices are capable of transmitting signals, the focus of this project is to properly implement the transmission of signals. The most recent version of AnSiAn does implement some prototyped transmission of Morse signals, but there are still some issues that need to be addressed. The implementation of the transmission chain is currently rudimentary and needs to be generalized for the transmission of more complex signals. Additionally, this projects tries to further improve the performance of AnSiAn and to fix existing issues.

1.2 FEATURE PLANNING

The project is implemented by only one developer, and therefore the feature scope needs to be defined accordingly. Since signal processing on Android often causes unforeseen issues, it has been decided to separate required features and optional (stretch) features to be able to plan enough time buffer for unplanned activities without failing to implement crucial functionality.

Note: This planning was created at the beginning of the project. It was decided to adjust the planning after the beta release, and thus features 3 and 5 have not been realized. See Section 2.3 for the adjusted planning.

The following features are currently planned to be implemented during this project:

1. **Implement transmission chain**: Currently AnSiAn only implements a DummyTransmissionChain with very limited function-

ality. The conclusion of the previous AnSiAn documentation stated that it is required to entirely re-implement the transmission chain to enable various use cases and modulations to use it [8].

2. **RDS transmission**: RDS is a specification on how to transmit additional information (like the stations name) alongside conventional FM radio broadcasts. The current version of AnSiAn already supports the demodulation and decoding of RDS signals. This functionality should be extended to being able to transmit own RDS data packets together with an FM modulated audio signal.

3. **BPSK demodulation improvements**: AnSiAn is capable of demodulating binary phase-shift keying (BPSK) signals, which is already used for RDS and PSK31. However, the developers faced some performance issues with the current implementation and suggested that this should be fixed in future releases. The improvement alternatives should be evaluated and implemented as a part of this project.

4. **Walkie-Talkie mode**: AnSiAn should be extended for an Walkie-Talkie functionality. It should be possible to use two smartphones with AnSiAn and a HackRF each as Walkie-Talkies. Therefore, AnSiAn should constantly receive and demodulate on a specified frequency and the user should then be able to quickly switch into transmission mode to send an own audio signal recorded from the included microphone. The time to switch to transmission mode and then back to reception mode should be as short as possible.

5. **Optional: QAM demodulation**: AnSiAn already implements BPSK, which is a special case of quadrature amplitude modulation (QAM). If there is time left at the end of this project this implementation could be extended to support 4-QAM, 16-QAM, 64-QAM and 256-QAM. This could for example be used to implement 802.11 in the future.

1.3 SPRINT PLANNING

The project will be implemented by a single developer. Therefore, the time available is 270 hours distributed over the entire winter term. We tried to estimate the required time for all features by splitting each feature up into consecutive tasks:

- **Project initiation phase [20h]**
- **Meetings with supervisor, final presentation [10h]**

- **Feature 1: Implement transmission chain [50h]**
  - Task 1: Investigation of existing code [5h]
  - Task 2: Rewrite modulator [10h]
  - Task 3: Implementation interpolator [10h]
  - Task 4: Refactor existing code and finalize [5h]
  - Task 5: Integration with already implemented transmission modes [10h]
  - Task 6: Regression testing [10h]
- **Feature 2: RDS transmission [35h]**
  - Task 7: Investigation of existing code [5h]
  - Task 8: Implementation in MATLAB [5h]
  - Task 9: Portation to Java [15h]
  - Task 10: Testing and bugfixing [10h]
- **Feature 3: BPSK demodulation improvements [55h]**
  - Task 11: Investigate existing code [5h]
  - Task 12: Research improvement alternatives [5h]
  - Task 13: Implement improvement [15h]
  - Task 14: Analyze performance increase [15h]
  - Task 15: Testing and bug fixing [10h]
  - Task 16: Regression testing [5h]
- **Feature 4: Walkie-Talkie mode [60h]**
  - Task 17: Design and implement UI [15h]
  - Task 18: Implement AM [15h]
  - Task 19: Implement FM [10h]
  - Task 20: Implement SSB [10h]
  - Task 21: Testing and bug fixing [10h]
- **Optional: Feature 5: QAM demodulation [ca. 50h]**
- **Documentation [25h]**
- **Presentation preparation [15h]**

Excluding the optional feature this sums up to 270 hours. To distribute the work equally over the entire winter term, we assigned each feature to one of the predefined submission dates:

- **Alpha Release - 22.12.2016**
  - Feature 1

  - Feature 2

- **Beta Release - 02.02.2017**
  - Feature 3
  - Feature 4

- **Final Release - 09.03.2017**
  - Documentation
  - Extensive Testing
  - Bug Fixes
  - Optional: Feature 5

1.4 DETAILED PROJECT PLAN

In this project we try to apply agile software development methods to reduce organizational and planning overhead. Therefore, we decompose each submission into 3 sprints of 2 to 3 weeks. After each sprint the achieved progress should be reviewed and the the plans for the next sprints should be adjusted.

We tried to assign each task to one of the sprints. However, this is not a static assignment and it must be reviewed and adjusted regularly. The preliminary planning currently is:
Sprint 1 04.11.2016-20.11.2016 [2,5 weeks] - Task 1,2,3,4 [30h]
Sprint 2 21.11.2016-04.12.2016 [2 weeks] - Task 5,6,7 [25h]
Sprint 3 05.12.2016-25.12.2016 [3 weeks] - Task 8,9,10 [30h]
////////// ALPHA RELEASE
Sprint 4 26.12.2016-08.01.2017 [2 weeks] - Task 11,12,13,14,15 [50h]
Sprint 5 09.01.2017-22.01.2017 [2 weeks] - Task 16,17,18 [35h]
Sprint 6 23.01.2017-05.02.2017 [2 weeks] - Task 19,20,21 [30h]
////////// BETA RELEASE
Sprint 7 06.02.2017-19.02.2017 [2 weeks] - Doc and Bug Fixes
Sprint 8 20.02.2017-05.03.2017 [2 weeks] - Doc and Bug Fixes
Sprint 9 06.03.2017-19.03.2017 [2 weeks] - Presentation
////////// FINAL RELEASE

To track the time spend on each task we use an online tool called Agilefant, which can be used to generate charts and to compare the expected and the actual time required to implement a feature.

1.5 SOURCECODE

For development we use a git repository on [3] which has been forked from the last version of Dennis Mantz and Max Engelhardt. All changes (including the documentation) will be available there.

1.6 TESTING

Testing is done on a Samsung Galaxy S6 and a Samsung Galaxy S6 Edge with Android 6.0.1. Additionally, **SEEMOO!** (**SEEMOO!**) provided a Nexus 6p with Android 7.1.1. For transmission and reception we use two HackRF Ones connected to either one of the smartphones or a linux computer with gqrx. For regression tests we will also use a RTL-SDR (RTL2832U).

2 PROJECT PROGRESS

At the very beginning of this project a detailed project plan has been defined. We divided each feature into a list of subsequent tasks and estimated the work required to complete them.

This chapter reviews the progress throughout the project. At the each release we analyze how much time has been spent on the single tasks and if our initial planning needs to be adapted.

2.1 ALPHA RELEASE: 22.12.2016

The goal was to implement feature 1 (Transmission Chain) and feature 2 (RDS Transmission). This goal has been achieved. Additionally we started to document the implementation, which was initially planned to be done at the end of the project.

Table 1: Expected and Actual Workload for Features in Alpha Release

	expected	actual
Transmission Chain	50h	24h
RDS Transmission	35h	55h
Documentation	0h	7h
Total	85h	86h

Table 1 shows the total workloads that were required to implement the features. We can see that we overestimated the transmission chain feature and underestimated the RDS feature. In total the required time is roughly as expected.

Table 2: Expected and Actual Workload for Feature 1: Transmission Chain

	expected	actual
Task 1: Investigation of Existing Code	5h	3h
Task 2: Rewrite Modulator	10h	9h
Task 3: Implementation Interpolator	10h	0h
Task 4: Refactor Existing Code and Finalize	5h	2h
Task 5: Integration	10h	4h
Task 6: Regression Testing	10h	6h
Total	50h	24h

Table 2 lists the single tasks for feature 1. We decided to leave out the interpolator at first (see Section 3.1). All other tasks were much easier as expected. Even the integration and testing with the already existing transmission modes went very well.

Table 3: Expected and Actual Workload for Feature 2: RDS Transmission

	expected	actual
Task 7: Investigation of Existing Code	5h	5h
Task 8: Implementation in MATLAB	5h	6h
Task 9: Portation to Java	15h	17h
Task 10: Testing and Bug Fixing	10h	26h
Total	35h	55h

Table 3 evaluates the required time for the tasks in feature 2. Tasks 7, 8 and 9 were nearly completed within the expected time, while the testing and debugging took much longer than expected. We highly underestimated the complexity of finding bugs in signal processing code on Android. Since we are dependent on the HackRF hardware and driver, most of the testing needs to be done on the actual smartphone. This means that it requires a lot of manual work - although we tried to implement some automated testing (see 3.2).

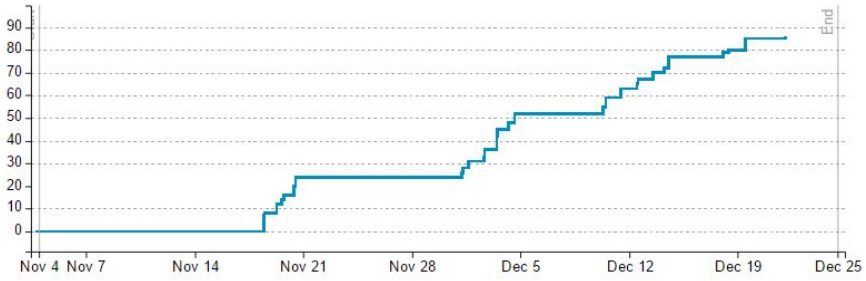

Figure 1: Workload Distribution for Alpha Release (04.11.2016-22.12.2016)

Figure 1 gives the distribution of the workload as exported from our tracking tool Agilefant. The workload was not well distributed at the beginning, mainly focused on single weekends with long breaks in between. During the last weeks before the deadline this became slightly more distributed.

2.2 BETA RELEASE: 02.02.2017

The (ambitious) goal of this release was to finish all remaining features. This goal was not achieved, due to lack of time. However, we already finished a good portion of the documentation, which was initially planned for the final release.

Table 4: Expected and Actual Workload for Features in Beta Release

	expected	actual
Walkie Talkie	60h	65h
BPSK Demodulation Improvement	55h	0h
Documentation	0h	13h
Bug Fixing of previous features	0h	10h
Total	115h	88h

Table 4 shows the expected and actual time spent implementing the features. The Walkie Talkie was expected to take 60 hours and was completed in 65 hours. The BPSK feature was expected to take 55 hours, but it was decided to exclude this feature, because the benefit of an improved BPSK demodulation implementation is not high enough. Additionally, we spent 13 hours creating this documentation and needed 10 unplanned hours to fix problems in the transmission chain and RDS implementation. In total we have spent 88 hours which is way below the planned value of 115 hours.

Table 5: Expected and Actual Workload for Feature 4: Walkie Talkie

	expected	actual
Task 17: Design and Implement UI	15h	19h
Task 18: Implementation AM	15h	0h
Task 19: Implementation FM	10h	18h
Task 20: Implementation SSB	10h	14h
Task 21: Testing and Bug Fixing	10h	14h
Total	60h	65h

Table 5 shows the actual time spent on each task in the Walkie Talkie feature. We underestimated every task. The implementation of AM was excluded, because it is not widely used.

Figure 2 shows the distribution of the workload over the time for the beta release. We see that there was no activity during the last week of December and first week of January (other than planned). The weeks after that are roughly evenly distributed. In total we already spent 194 hours (20h initiation + 86h alpha + 88h beta) on this project, such that we have 66 hours left. We are planning to use this time to implement SSTV, finalize the documentation and to prepare the final presentation.

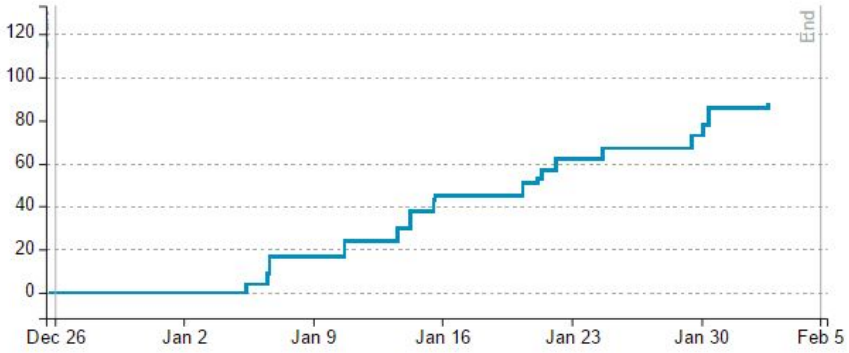

Figure 2: Workload Distribution for Beta Release (23.12.2016-02.02.2017)

2.3 FINAL RELEASE: 09.03.2017

In this final release, we completed the documentation and prepared the final presentation. Other than initially planned, we decided after the beta release to implement SSTV instead of improving BPSK and implementing QAM. Therefore, the expected workloads for those features became irrelevant. We estimated that we would be able to implement the SSTV feature in 50 hours.

Table 6: Expected and Actual Workload for Features in final Release

	expected	actual
SSTV	50h	58h+
Documentation	25h	21h
Presentation	15h	15h
Total	90h	94h

Table 6 shows the actual workload for this release. We see that we spent 58 hours on the SSTV feature and there are still parts that are currently not working properly. The documentation and presentation have been completed in the expected time, but other than initially planned we spent a good amount on the documentation in the previous releases. Therefore, the total documentation time is higher than expected.

Figure 3 shows the distribution of the workload over the time frame. The majority of the work was done in the last 3 weeks, due to other exams and vacation.

Figure 3: Workload Distribution for Final Release (03.02.2017-17.03.2017)

3 IMPLEMENTATION

This chapter describes the implementation of the features that were added to AnSiAn. We roughly describe the theoretical background, the design choices made and the actual implementation in Java. Additionally, we extensively describe the challenges we faced during the implementation.

3.1 FEATURE 1: TRANSMISSION CHAIN

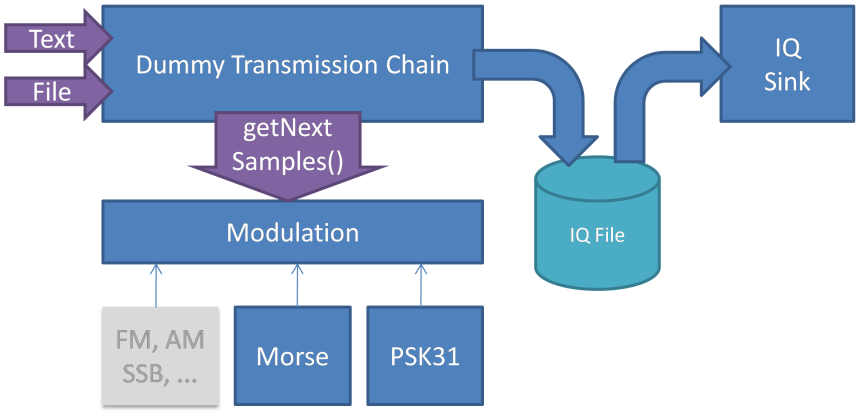

Figure 4: Transmission in the current Version of AnSiAn [8]

Figure 4 illustrates the implementation of the transmission before our project started. We roughly analyze how this has been implemented to find out the main problems of the implementation. The DummyTransmissionChain is the entry point of a transmission. It is called when the user initiates a transmission. The different modulations (currently PSK31 and Morse) are implemented as a subclass of Modulation. The DummyTransmissionChain repeatedly calls getNextSamples() to get a SamplePacket of the selected modulation until it returns null. The SamplePacket is basically a complex array (i.e. 2 float arrays containing in-phase and quadrature components). The DummyTransmissionChain then serializes this to a signed 8 bit integer IQ file, i.e. converts floats to bytes and interleaves I and Q samples. Once the modulation is done, the IQSink is launched which reads the file and passes it to the HackRF driver using the BlockingQueue of byte buffers provided by the HackRF driver.

With this implementation we see the following problems:

- The modulation is entirely done before the transmission is started. This is not only very inconvenient for the user since he has to wait for a long time before the transmission starts, but it also is not suitable for the features we want to implement where we require continuous transmission of recorded audio. We want to implement this in a way where the modulation is done simultaneously to the transmission. Mantz and Engelhardt [8] already described a concept to use BlockingQueues, i.e. we queue up N packets and then block until a packet has been removed from the queue. This helps to have enough samples ready, while not using too much memory, which is a rare resource on smartphones.
- Writing to a file introduces additional overhead. This was done to save memory, but is not required if we can do the modulation just-in-time.

3.1.1 *Structural Changes*

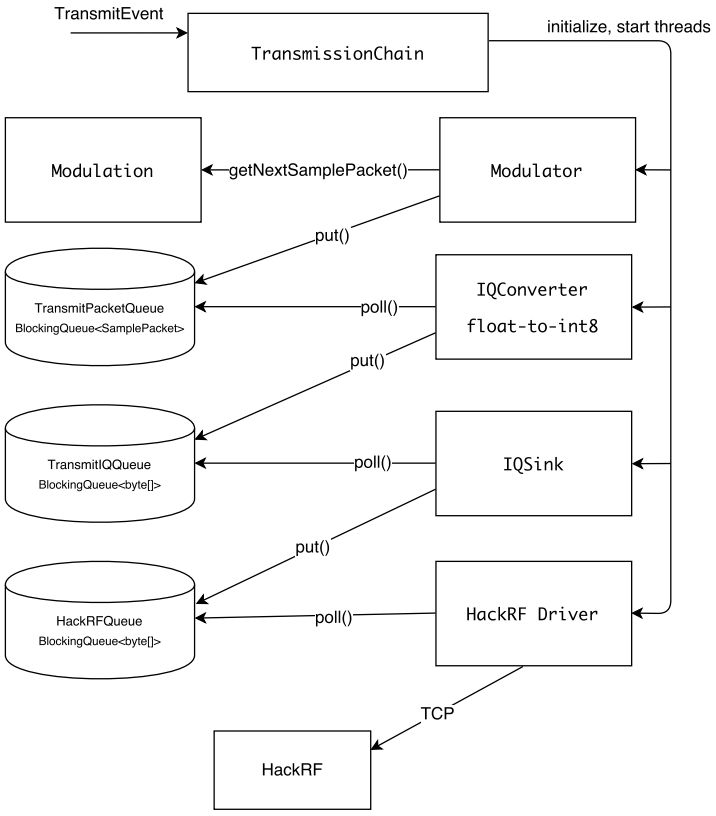

Figure 5: Implementation of the Transmission Chain

To solve the problems detected with the old implementation we needed to adjust the architecture of the transmission chain. The cur-

rent transmission chain is outlined in Figure 5. We briefly describe the responsibilities of each component:

- **TransmissionChain**: Responsible to launch the transmission chain. It listens to TransmitEvents that were sent by the UI. It is the only object that is created on app startup and is kept in memory for the entire time - all other parts of the transmission chain are started for a single transmission and die afterwards. Ideally this class is the only one that contains Android specific / dependent code.
- **Modulator**: The Modulator continuously gets SamplePackets from the correct modulation class and pushes them to the TransmitPacketQueue.
- **IQConverter**: Converts the SamplePackets (float) from the TransmitPacketQueue to the required data format (for HackRF: 8 bit signed integer IQ samples). The data is filled into byte buffers which are then enqueued into the TransmitIQQueue.
- **IQSink**: Configures the HackRF Driver. Then continuously takes the buffers from the TransmitIQQueue and passes them to the driver specific queue.

We decided not to implement an interpolator at first, which would have enabled to use different sampling rates for modulating than for the actual transmission. We did this because currently we are using 1 MHz everywhere. If we require different sampling rates later this can be easily added.

3.1.2 *Implementation*

After we described the structural changes and required classes in the last section, we now point out a few specific implementation choices that were made during the development: Since we need to run the components concurrently and we do not want to do a lot of work on the main thread, we need to make use of multiple threads. When a transmission is initialized the TransmissionChain object creates an object of each component and then runs each in a separate thread.
We already described the concept of blocking queues. This highly simplifies the implementation of the chain, since we do not need to take care of the communication and memory management. The threads just do their work and enqueue the results in the next queue and the rest is controlled by the queue limits. We only need to decide how long the queues are. The byte buffers have a size of 16 KiB (as defined by the HackRF driver), while the SamplePackets can have arbitrary size (usually a lot bigger). Therefore, we currently choose 5

as the maximum size of the TransmitPacketQueue and 200 for the TransmitIQQueue.

It is important to keep memory management in mind. If every component just keeps allocating new memory the application will run out of memory or the Java garbage collector will be too busy and drastically slow down the application. Therefore, we use the byte buffer pool of the HackRF driver. This again uses BlockingQueues, which makes sure that only a fixed number of buffers are used throughout the application.

All components need to be interruptible. This needs to be implemented by the class itself. The TransmissionChain can only call the .interrupt() method of the thread, but the thread itself needs to take care that this interrupt is received and correctly handled.

The implementation of the IQConverter was more challenging than expected. The main challenge is that the size of the SamplePackets can be different than and even not divisible by the buffer size required by the IQSink. We decided to memorize the last samples of the SamplePacket until the next SamplePacket is processed. However, we cannot know if a SamplePacket is the last one. Therefore, we implemented that that if no SamplePacket arrives for a specified time (here 1000 ms), we fill the rest of the buffer with zeros and transmit it.

3.1.2.1 *Debugging and Testing*

Since all software contains bugs, we needed to find a way to debug and test our implementation in a convenient and fast way. We developed unit tests for the IQConverter, since it contains logic that can be easily tested by a unit test.

To test the other components we wanted to monitor the bytes that are actually passed to the HackRF driver. To do so we implemented a FileIQSink, that can be launched instead of the normal IQSink. This FileIQSink writes the samples to a file. Then we transfered this file to a computer and analyzed them with Matlab, Audacity or a hex editor. We included this settings in the App preferences, such that the user can enable the FileIQSink. After we successfully fixed a lot of smaller bugs in our implementation, we integrated it into the current application and performed manual regression tests of the already implemented transmission features. We tested our implementation with the 3 currently implemented transmission modes:

- Morse Code Modulation
- PSK31 with USB Modulation
- Directly transmitting a IQ File

All three transmission modes are now working without any problems and do no longer require long calculations before the transmission.

3.2 FEATURE 2: RDS TRANSMISSION

As the second feature we implemented the synthesis of RDS signals. RDS has been standardized by the RDS Forum [4]. It is a good example for a non-trivial protocol that is very widely used. We wanted to demonstrate that these signals can be synthesized on a smartphone, such that a commodity kitchen or car radio is able to decode them.

One of the main features of RDS is the transmission of the station information (station name, audio information, station location and other metadata about the transmission). Most radios only display the station name. Thus, we are also concentrating on this feature.

The RDS signals are transmitted together with the audio signal using frequency modulation. By implementing RDS in AnSiAn, we enables the user to broadcast his own radio signals with a station name specified by the user.

3.2.1 *User Interface*

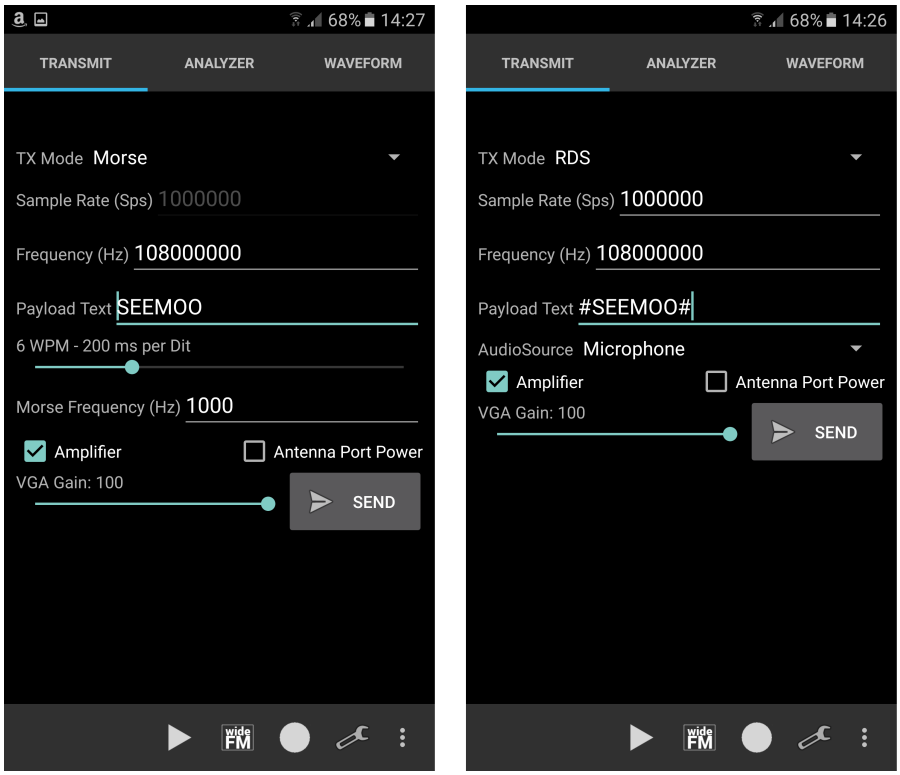

Figure 6: AnSiAn Transmission Tab: Morse (old, left) and RDS (new, right)

Figure 6 shows a screenshot of the extended transmission tab of AnSiAn. For reference we printed the already existing Morse transmission screen on the left. On the right you can see the newly added option which enables the user to transmit RDS signals. The user can enter his own station name (at most 8 characters) and select an audio source (currently we are supporting the integrated microphone and an audio file source). The other settings are the same as for the other transmission modes.

3.2.2 *Implementation*

To transmit a radio signal including RDS, we need to do the following steps (c.f. [4]):

1. Generate a bit string according to RDS specification (4 groups, each 64 bit)
2. Calculate 10 bit check words for every 16-bit block (resulting in 4 groups, each 104 bits)
3. Calculate differential encoding of the bit string
4. Manchester encode the bit string
5. Apply a low-pass filter to the signal to cut off frequencies higher than 2800 Hz
6. Upconvert the generated signal to 57 kHz
7. Add a sinusoidal pilot tone at 19 kHz
8. Add the baseband audio signal. Most audio signals are sampled at 44100 Hz, therefore, we need to resample these signals to match our sampling rate of 1MHz.
9. Calculate the frequency modulation on the signal with a frequency deviation of 75 kHz

The steps after step 4 are illustrated in the GNU radio flow graph in Figure 7. Since most of these steps are self-explanatory, we only explain the more complex ones in detail here. For the bit string generation and check word calculation refer to the documentation of the RDS decoding functionality implemented in AnSiAn [8, Section 4.2.3].

3.2.2.1 *Low-Pass Filtering*

Listing 1: AnSiAn Blackman Low-Pass Filter

```
// create a filter
```

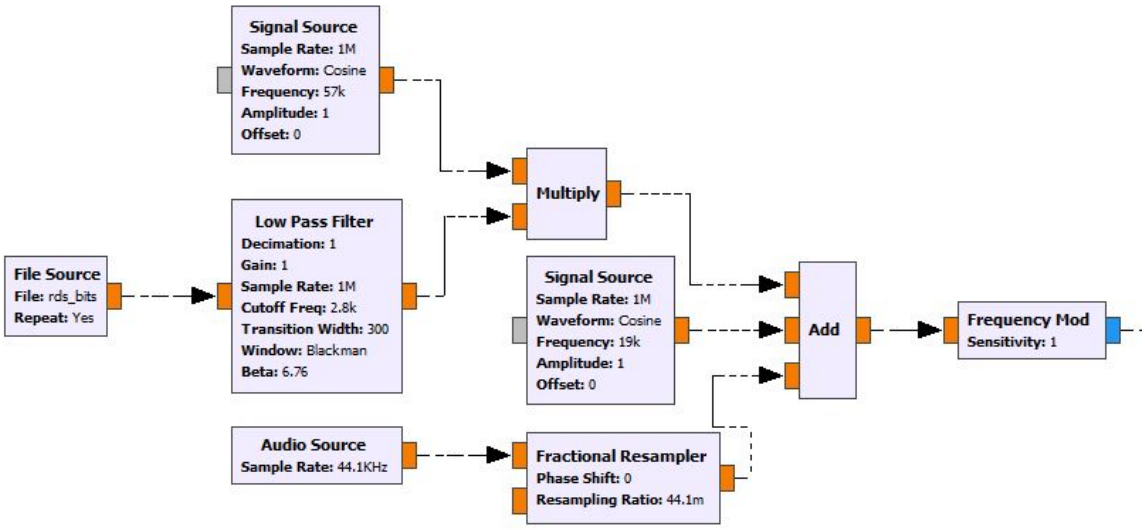

Figure 7: GNU Radio Flow Graph of RDS Signal Generation

```
public static FirFilter createLowPass(int decimation, float gain,
float sampling_freq, // Hz
float cutoff_freq, // Hz BEGINNING of transition band
float transition_width, // Hz width of transition band
float attenuation_dB) // attenuation dB

// filter
public int filterReal(SamplePacket in, SamplePacket out, int
    offset, int length)
```

AnSiAn already contains an implementation of a Blackman low-pass filter. The signature of the low pass filter function is shown in Listing 1.

However, the choice of the parameters highly influences the performance of the filter. A bad parameter choice either leads to a badly filtered signal or to a very long filtering time. Since we are very constrained in terms of computing power on the Android platform, we want to choose the parameter such that the filter is just good enough but still very fast. We found that the best parameter choice for us was cutoff_freq=2800Hz, transition_width=300Hz and attenuation_dB=3dB.
The parameters decimation=1, gain=1 and sampling_freq=1000000 are fixed for us.

3.2.2.2 *Frequency Modulation*

For implementing frequency modulation, we looked the implementation in the Octave communications package. Listing 2 shows how Octave implements fmmod.

Listing 2: Octave Implementation of Frequency Modulation [9]

```
function s = fmmod (m, fc, fs, freqdev)
   l = length (m);
   t = 0:1./fs:(l-1)./fs;
   int_m = cumsum (m)./fs;
   s = cos (2*pi.*fc.*t + 2*pi.*freqdev.*int_m);
endfunction
```

It implements the following function

$$s(t) = \cos\left(2 \cdot \pi \cdot f_c \cdot t + 2 \cdot \pi \cdot f_\Delta \int_0^t m(t')dt'\right) \tag{1}$$

However, this modulates the signal in the passband, but we require it in the baseband and the up conversion is done later by the HackRF. Thus, we can either set $f_c = 0$ or use

$$s(t) = \cos\left(2 \cdot \pi \cdot f_\Delta \cdot \int_0^t m(t')dt'\right) \tag{2}$$

The implementation can be easily adapted to Java and is shown in Listing 3.

Listing 3: Java Implementation of fmmod

```
public static SamplePacket fmmod(float[] x, float fs, float
    freqdev ){
   SamplePacket packet = new SamplePacket(x.length);
   packet.setSize(x.length);
   float[] sum = cumsum(x);
   for(int i=0;i<sum.length;i++){
      sum[i] = sum[i]/fs;
   }

   float[] re = packet.getRe();
   float[] im = packet.getIm();
   for(int i=0;i<sum.length;i++){
      re[i] = (float) Math.cos(2*Math.PI*freqdev*sum[i]);
      im[i] = (float) Math.sin(2*Math.PI*freqdev*sum[i]);
   }
   return packet;
}
```

3.2.3 *Testing*

Testing a transmission implementation can be very tedious, since even small bugs lead to a non-decodable signal. Therefore, we again used the FileSink to write the generated signals to a file instead of transmitting them with the HackRF. We analyzed these signals and

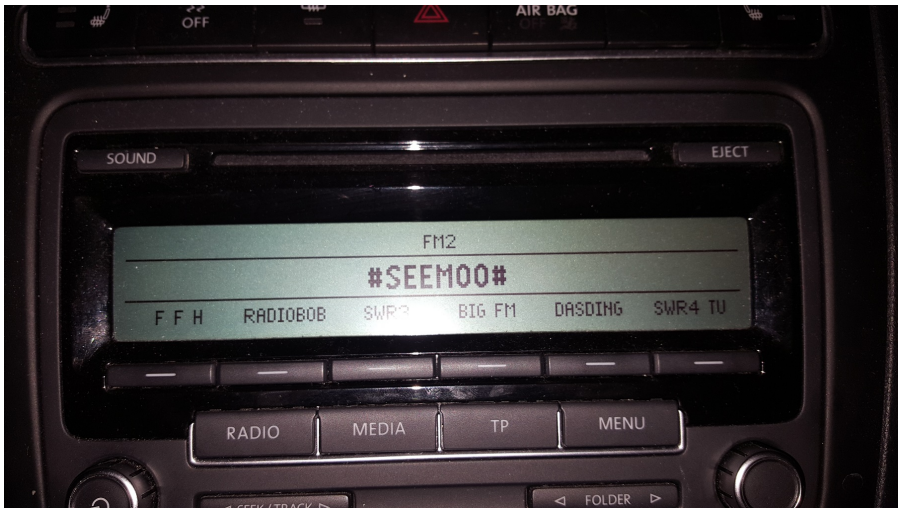

Figure 8: Decoded RDS Signal on Car Radio

compared the intermediate results to a reference implementation of RDS in Matlab.

Now AnSiAn is able to send RDS signals which can be decoded by commodity radios like the car radio in Figure 8.

3.2.4 *Optimization*

Since the modulation of the RDS signals requires quite some computation we decided to do the calculation until step 7 before the actual transmission is started. This precalculated RDS frame does not change over time, so we can just reuse it in every new SamplePacket that is requested. The actual getNextSamplePacket() then only retrieves the audio samples (from file or from microphone), adds them to the RDS frame and does the frequency modulation. This computation can be done in real-time without any problems.

Since our transmission chain and the HackRF work with 1 MHz sampling rate, we initially also used 1MHz to calculate the RDS signals. This, however, proved to be extremely expensive. Especially the filtering of the signal does require a lot of time. This resulted in a precalculation time of up to 6 seconds even on our high end testing devices. Since the signal only contains frequencies up to around 60 kHz, we do not require such a high sampling rate. We improved the code (basically steps 4 to 7) to work with a variable sampling rate. After step 7 the signal is then upsampled to 1 MHz, such that the real-time computation (step 8 and 9) remains unchanged.

To keep the upsampling step easy (both from implementation and computational view), we only used factors of 1 MHz. We tested 500 kHz, 250 kHz, 200 kHz and 125 kHz. However, for 125 kHz, the signal was no longer decodable by our testing radio. Therefore, we used 200 kHz as the lowest sampling rate.

Table 7: RDS Optimization: Using Different Sampling Rates for the Calculation of 4x4 RDS Groups (around 1400ms)

sampling rate	avg. execution time (Samsung Galaxy S6)	avg. execution time (Google Nexus 6p)
1MHz	4840 ms	5684 ms
500kHz	1340 ms	1566 ms
250kHz	441 ms	480 ms
200kHz	339 ms	340 ms

We performed experiments on how the sampling rate affects the execution time. The results are shown in Table 7. Since the experiments are performed on physical devices, the processor might also be busy with other computations. Therefore, we repeated the experiment five times for each sampling rate and device and used the mean value (the variance was small for all measurements). We see that the sampling rate drastically affects the performance. We have been able to reduce the computation time from 4840 ms on the Galaxy S6 (or 5684 ms on the Nexus 6p) to 339 ms (340 ms), i.e. the optimized version only needed 7% (6%) of execution time. With the optimized implementation we could in theory do the modulation in real-time and not ahead of the transmission start. However, since we also need to do other computation and we do not want to unnecessarily strain the smartphone, we did not change that part.

3.3 FEATURE 3: WALKIE TALKIE

AnSiAn in the version of December 2016 is capable of receiving and demodulating analogue signals, transmitting and receiving simple digital signals and now even implemented the first advanced digital modulation. However, it stills lacks the ability to transmit analogue signals. This is implemented as a part of feature 3.

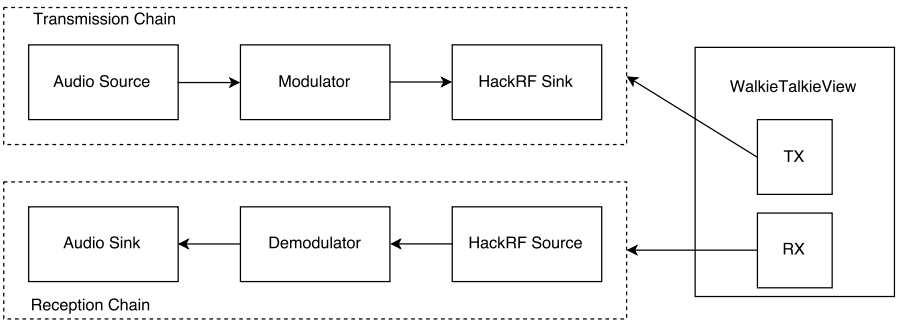

Figure 9: Design of the Walkie Talkie Feature (Simplified Tx and Rx chain)

Since AnSiAn with this feature is capable of providing all functionality of an amateur radio walkie talkie, we call this feature Walkie Talkie and extend AnSiAn with an additional view that provides modest access to this feature. Figure 9 shows the components that are required for this feature. First, we need a new view called "WalkieTalkieView". This view basically allows the user to switch between reception and transmission mode. Then, either the reception chain or the transmission chain is active. The reception chain retrieves samples from the HackRF, demodulates them and plays the audio signal using the built-in speakers. The reception chain is already fully implemented in the previous version of AnSiAn. So, only very minor changes are required here. The transmission chain records samples from the integrated microphone, modulates them and transmits them with the HackRF. Most required parts here have been implemented in Feature 1 (see Section 3.1). However, we need to implement new modulations here. We want to implement FM, LSB and USB. FM will be integrated, because it was already used in feature 2 (Section 3.2), and therefore, we expect the implementation to be straight-forward. LSB and USB will be integrated because they are the most widely used modulation schemes in amateur radio.

3.3.1 *User Interface*

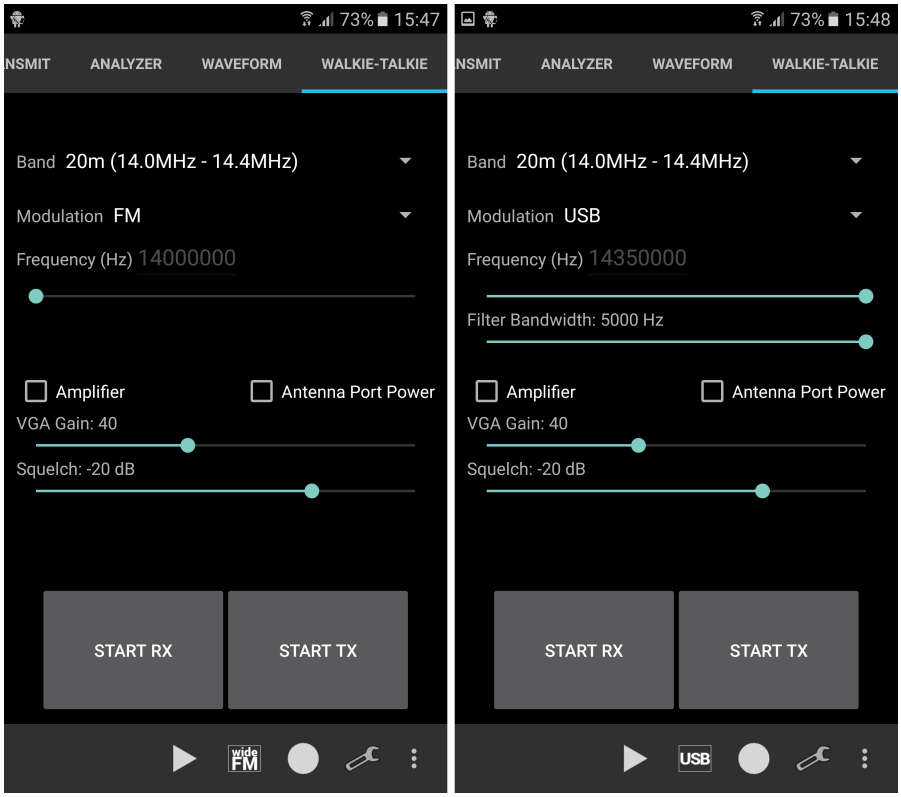

Figure 10: AnSiAn WalkieTalkie Tab: FM (left) and USB (right)

Figure 10 shows the new WalkieTalkie tab implemented as a part of this feature. On the left side you can see the UI when using FM and on the right side is the UI when using USB (or LSB). The only difference is that, when using USB or LSB, the user can specify the width of the filter applied to the signal (see Section 3.3.2.3). To ease the selection of a legal amateur radio frequency, we provide a selection of popular frequency bands that can be used legally in Germany if the user possesses an amateur radio license [11]:

- 80m Band (3.5MHz - 3.8MHz)
- 60m Band (5.3MHz - 5.4MHz)
- 40m Band (7.0MHz - 7.2MHz)
- 30m Band (10.1MHz - 10.2MHz)
- 20m Band (14.0MHz - 14.4MHz)

The choice of the frequency might depend on various factors, e.g. used antenna, hardware of communication partner, frequency occupancy and intended communication range. Additionally, we provide a "other" band, which enables the user to freely choose a frequency within the range supported by HackRF (1 MHz to 6 GHz). However, the user is responsible for using this feature legally. To fine tune the frequency the user can use a seek bar to select a frequency within the selected band.

After selecting the frequency band, the user can select one of three modulation schemes, namely FM, LSB and USB. Usually LSB is used for frequencies below 10 MHz and USB is used for frequencies above. However, we let the user decide which modulation scheme he wants to use. The settings for Amplifier, Antenna Port Power and VGA Gain are parameters for the HackRF hardware, that can be used to optimize the transmission/reception quality. Since they are the same as in the TransmitView, we do not explain them in detail here and refer to [8].

The last parameter that can be set in the WalkieTalkieView is the squelch threshold setting. It is used to suppress noise when there is no signal.

At the bottom of the screen, there are two buttons to start/stop reception and transmission. Since we cannot receive and transmit at the same time, we need to switch between these modes. We assume a classical walkie talkie use case, where we receive for the majority of the time and switch to transmission mode for short periods in between. Therefore, we implemented the state machine shown in Figure 11.

At first, the radio is turned off and the user can enter the parameters. Then, the transitions 1 to 4 are straight-forward: The user can either enable RX or TX. Accordingly, the reception or transmission is started. By pressing the same button again, the radio is turned off

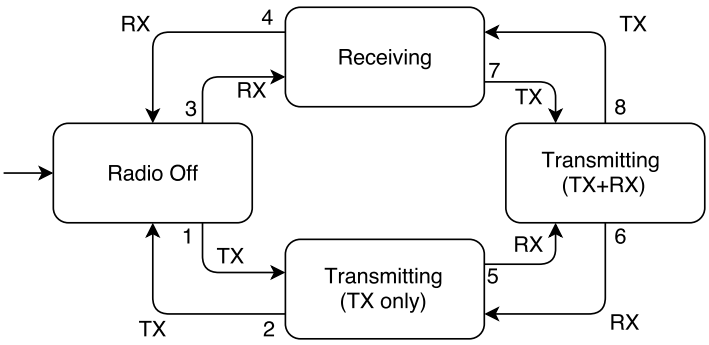

Figure 11: State Machine of WalkieTalkie View

again. The other transitions are similar to the behavior we would expect from a walkie talkie:

- **Transition 5** - The HackRF is in transmission mode and the user pressed the RX button: No action is taken, but a RX flag is set, such that once the transmission is stopped, we change into reception mode and not off.
- **Transition 6** - The HackRF is in transmission mode, the RX flag is set and the user presses the RX button: No action is taken, but the RX flag is unset, such that the reception is not resumed once the transmission is stopped.
- **Transition 7** - The HackRF is in reception mode and the user presses TX: The reception is paused and the transmission is started.
- **Transition 8** - The HackRF is in transmission mode, the RX flag is set and the user presses TX: The transmission is stopped and the reception is resumed.

A classical state transition flow would be 3, 7, 8, 4.

3.3.2 *Implementation*

This section describes the required changes in the reception and transmission chain of AnSiAn. Since the reception of FM, USB and LSB was already implemented in the last version, the only task here was to correctly call the API of the reception chain. For setting the required parameters (e.g. frequency) AnSiAn uses Android Preferences, which handle the serialization and deserialization of the settings to/from a file to persist the the settings beyond the runtime of Android applications. For more details we refer to the Android documentation [2] and the documentation of the first AnSiAn version [6].

The start of the reception is then initiated by posting a RequestStateEvent to the EventBus (which is a publish/subscribe library by

greenrobot). The communication within AnSiAn is mainly based on such events. For greater details on the large variety of available events in AnSiAn, we again refer to [6]. The only change required in the reception chain was to extend the preferences, such that a custom filter bandwidth can be specified for the LSB and USB demodulation.

The most changes for this feature were required in the transmission chain. Firstly, we needed to generalize the recording of audio from the microphone from the RDS feature. Additionally, we implemented the three modulation schemes FM, LSB and USB. FM was already used for RDS, so we just needed to generalize the implementation and exclude the RDS signals. LSB and USB are newly added. The following sections describe some key aspects in greater detail.

3.3.2.1 *AudioSource*

For all three modulations it is required to retrieve samples from the microphone. We implemented this functionality in the self-contained AudioSource class, such that it can also be used in other scenarios. Basically, the class provides three methods: One for starting the collection of audio samples (startRecording()), one to retrieve the audio samples (getNextSamples()) and one to stop the recording and release the resources (stopRecording()).

Listing 4: Retrieving PCM Samples from the Microphone

```
1  public class AudioSource {
2  //...
3   private int AUDIO_SOURCE = MediaRecorder.AudioSource.MIC;
4   private int AUDIO_SAMPLERATE = 44100;
5   private int AUDIO_CHANNELS = AudioFormat.CHANNEL_IN_MONO;
6   private int AUDIO_ENCODING = AudioFormat.ENCODING_PCM_FLOAT;
7
8   public boolean startRecording(){
9      this.recorder = new AudioRecord(AUDIO_SOURCE,
10         AUDIO_SAMPLERATE, AUDIO_CHANNELS,
11         AUDIO_ENCODING, this.bufferSize*4);
12     this.recorder.startRecording();
13     // ... error handling ...
14  }
15
16  public float[] getNextSamples(){
17     if(this.recorder == null) return null;
18     float[] buffer = new float[this.bufferSize];
19     this.recorder.read(buffer, 0, buffer.length, AudioRecord.
           READ_BLOCKING);
20     return buffer;
21  }
22  //...
23 }
```

Luckily, Android does provide a API to retrieve float PCM samples through the AudioRecord class [1]. Listing 4 shows the implementation of startRecording() and getNextSamples(). For the purpose of presentation, some implementation details for error handling have been removed. An instance of AudioRecord can be created using the following parameters:

- **audioSource**: MediaRecorder.AudioSource.MIC is the most reasonable for us. Other options are uplink and/or downlink of a current call.
- **sampleRateInHz**: 44100 Hz is guaranteed to work on all devices. Some devices might provide other sample rates as well.
- **channelConfig**: Mono is guaranteed to work on all devices. Stereo might not be supported.
- **audioFormat**: We use floats, since our entire modulation works with floats. Other options would be 8-bit (i.e. bytes) or 16-bit (i.e. shorts).
- **bufferSizeInBytes**: size of internal buffer. We use a default value of 32 KiB.

If the initialization (line 9) finishes successfully, the recording can be started using startRecording() (line 12). From now on Android fills up the buffer and we need to make sure that we retrieve the samples in time. Otherwise the buffer becomes full and Android throws away audio samples.

The retrieval of audio samples is implemented in the getNextSamples() method. It allocates a new float buffer and calls the read method on the AudioRecord. Since Android API Level 23 (6.0 / Marshmallow) this can be called with a AudioRecord.READ_BLOCKING argument, such that the call blocks until enough audio samples are available to fill the buffer. This simplifies our implementation, since we want the Modulator to block until enough samples are available. If we wanted to support lower API levels, we would need to implement this blocking behavior ourselves.

Additionally, to this getNextSamples() method, we provide two other methods to retrieve the samples: getNextSamplesUpsampled() (returns the samples upsampled to a higher sampling rate, which is specified in the constructor) and getNextSamplePacket() (returns a SamplePacket instead of a float array).

3.3.2.2 *FM*

The central method that needs to be implemented for each modulation is the getNextSamplePacket() method, which is repeatedly called by the modulator until null is returned or the transmission is stopped.

Using the fmmod from Section 3.2 and the AudioSource class, a frequency modulation implementation is straight-forward.

Listing 5: Modulating Microphone Samples using Frequency Modulation

```
1 @Override
2 public SamplePacket getNextSamplePacket() {
3         if(this.audioSource == null) return null;
4         float[] upsampled = this.audioSource.
              getNextSamplesUpsampled();
5         if(upsampled == null) return null;
6         SamplePacket resultPacket = FM.fmmod(upsampled,
              sampleRate, 75000);
7         return resultPacket;
8 }
```

Listing 5 shows the basic steps. It retrieves the audio samples from the AudioSource (l. 4). This might block until enough samples are available. Since this audio is sampled with 44100 Hz, we need to upsample the signal to 1MHz. This is already integrated in the AudioSource class with the getNextSamplesUpsampled(). If an error occurs during the retrieval of the samples, null is returned and we also return null to the Modulator (l. 5). This will cause the transmission to stop. Otherwise we pass the upsampled signal to fmmod, which implements the frequency modulation (l. 6).

3.3.2.3 *LSB/USB*

In amateur radio the most widely used modulation scheme is SSB, which is an advancement of amplitude modulation. Figure 12 illustrates why this is done. It contains the time and frequency domains of a modulated audio signal (to be more precise: the Star Wars Imperial March). Subplot 1 contains the time domain of the source signal. Subplot 2 shows the frequency domain of this signal. Since we are working with complex signals, we have positive and negative frequencies which are mirrored at 0 Hz. This is no problem in the baseband (red), but when upmixing this signal to the carrier frequency (green), we see that both sidebands are upmixed. This has two disadvantages: Firstly, it is a waste of transmission power, since the second sideband does not contain additional information. Secondly, our signal occupies more bandwidth, which is a very rare resource. This problem is solved by SSB. We either use the upper sideband (positive frequencies) or the lower sideband (negative frequencies). The ideal frequency domains are shown in Subplot 2 and 3.

There are multiple ways to implement SSB modulation. The plots in Figure 12 are generated in Matlab using the Hilbert transformation to create a complex signal and then multiplying the imaginary part by either i (USB) or -i (LSB). However, this requires both the implementation of the Hilbert transformation and the complex multiplication, both is currently not available in AnSiAn.

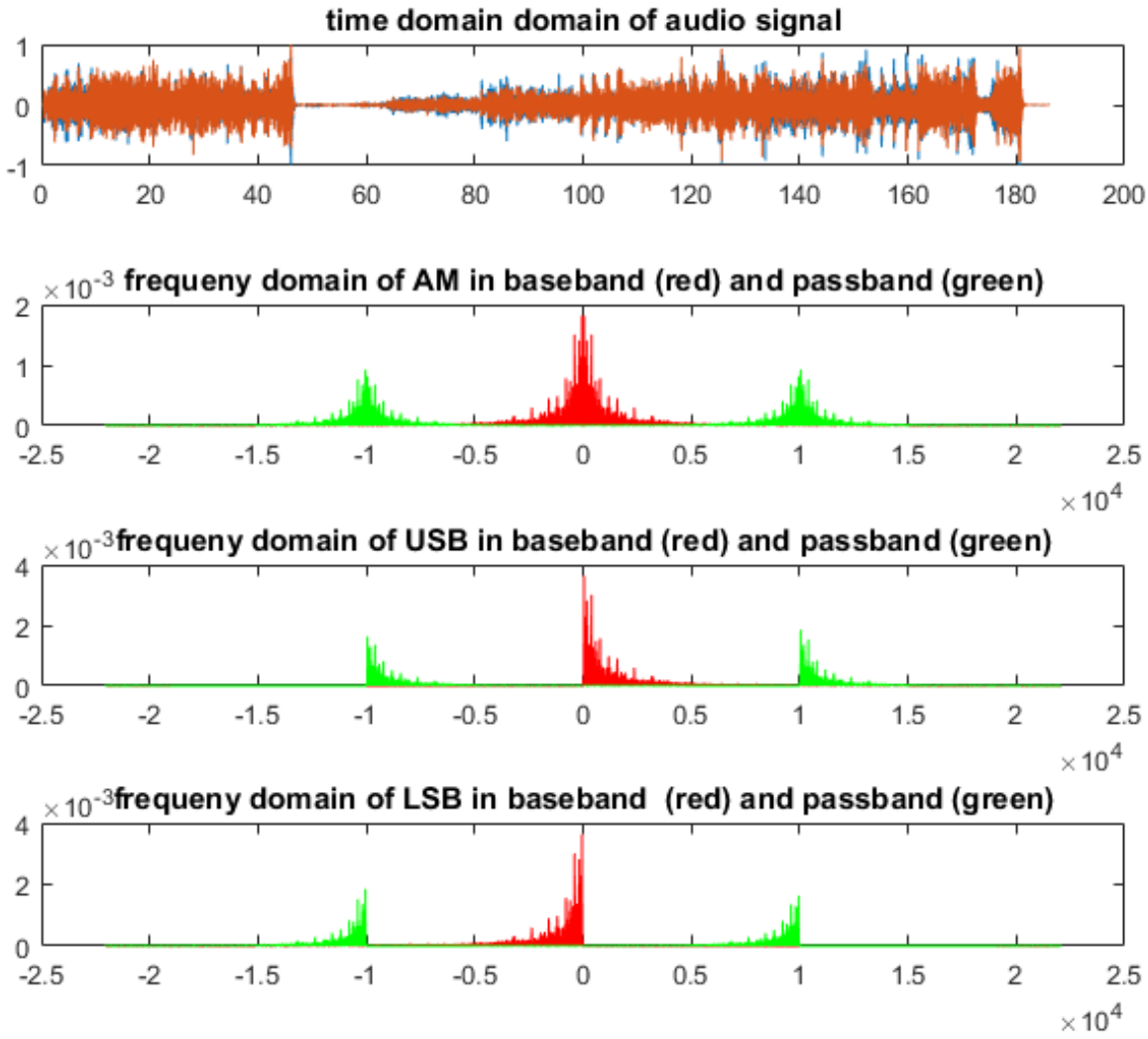

Figure 12: Time Domain and Frequency Domain of AM, LSB and USB Signals

Listing 6: Modulating Microphone Samples using SSB

```
1  @Override
2  public SamplePacket getNextSamplePacket() {
3    // get audio samples
4    SamplePacket packet = audioSource.getNextSamplePacket();
5
6    if(packet == null) return null;
7
8    // filter out the upper or lower side band
9    ComplexFirFilter filter = ComplexFirFilter.createBandPass(1,1,
         audioSource.getAudioSamplerate(),
10         isUpperSideband ? 0 : -this.filterBandWidth,
11         isUpperSideband ? this.filterBandWidth: 0,
12         audioSource.getAudioSamplerate()*0.01f, 40);
13
14   SamplePacket output = new SamplePacket(packet.size());
15   filter.filter(packet,output, 0, packet.size());
16
17   // upsample from audio frequency (usually 44.1kHz) to our
         transmission frequency (1MHz)
18   SamplePacket upsampled = output.upsample((int) Math.ceil((float
         ) this.sampleRate / audioSource.getAudioSamplerate()));
19   return upsampled;
20 }
```

Therefore, we decided to use a filter to remove the other sideband. Listing 6 shows the implementation of the SSB in AnSiAn. We retrieve the next audio samples from the AudioSource (l. 4). Then, we filter out the band we don't want using a ComplexFirFilter - the arguments are similar to the FirFilter used in Section 3.2. When using USB, the passband of the filter is from 0 Hz to the user defined filter bandwidth, e.g. 3000Hz. When using LSB this would be -3000Hz to 0 Hz. The resulting packet is then upsampled to 1MHz and returned to the modulator.

Note: The AudioSource does return a real signal, not a complex signal, i.e. all imaginary parts are zero. However, the filter does correct this, such that the output signal does contain both a real and an imaginary part.

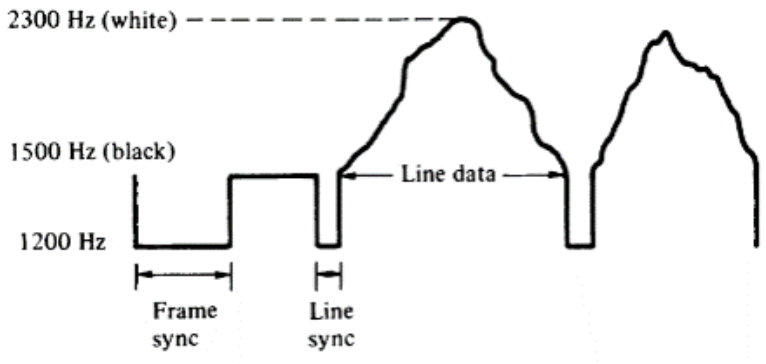

Figure 13: Modulation of SSTV Signals - Frequency Domain of Sync and Line Sequences [10, p. 239]

3.4 FEATURE 4: SLOW-SCAN TELEVISION (SSTV)

Having implemented a text and a speech mode of AnSiAn, we decided to also add a modulation schema capable of transmitting images. One amateur radio scheme for transmitting images is SSTV. It is used to transmit single images using a very small bandwidth with a relatively small frame rate. Depending on the mode used, SSTV transmits images in 8 to 400 seconds using less than 3 kHz of bandwidth, while amateur television (ATV) (also known as fast-scan television (FSTV)) uses multiple MHz of bandwidth but is capable of a much higher frame rate.

The idea of SSTV is illustrated in Figure 13. It shows the frequency domain of a SSTV signal. SSTV transmits the lines of an image one after the other. There exist multiple modes of SSTV with different resolutions, frame lengths, image types (black/white or RGB) and slightly different modulations. We concentrate on the basic black/white mode with a frame length of 8 seconds and 120 lines per frame.

A SSTV mode defines three frequencies:

- A **sync frequency**, used for finding the start of a frame (frame sync) and the start of each individual line of the image. This usually is 1200 Hz.
- A **black frequency**, the frequency that represents a black pixel. (here: 1500 Hz)
- A **white frequency**, the frequency that represents a white pixel. (here: 2300 Hz)

The frequencies between the black and white frequency are used to linearly interpolate gray-scale pixels. For example, when a pixel is represented as a byte, where 0 is black and 255 is white, a gray value of 128 would be 1902 Hz (1500 Hz + 128*(2300 Hz-1500 Hz)/255).

An SSTV frame consists of the following parts:

- **Frame sync**: Each image/frame starts with a frame sync sequence. For the B/W 8s mode the frame sync has a length of 30ms.
- **For each line:** (120 lines for B/W 8s)
  - **Line data**: Starting with the left-most pixel, each pixel is modulated as a sine of the interpolated frequency successively. In B/W 8s a line has a length of 61.6ms (excluding line sync). However, the number of pixels per line is not fixed. The transmitter and receiver can decide which horizontal resolution to use. Typical values are 128 and 256 (i.e. each pixel lasts 0.48ms or 0.24ms).
  - **Line sync**: At the end of each line a synchronization sequence is inserted. For B/W 8s this sequence has a length of 5ms. This sequence is used by the receiver to sync exactly to the beginning of the next line.

We are going to implement the B/W 8s mode in AnSiAn, which can be summarized as follows [10]:

- **Sync frequency**: 1200 Hz
- **Black frequency**: 1500 Hz
- **White frequency**: 2300 Hz
- **Frame Sync**: 30ms
- **Line length**: 1/15 s = 66.6ms
  - **Line data**: 61.6ms
  - **Line sync**: 5ms
- **Frame length**: 8s = 120*66.6ms

To integrate SSTV into AnSiAn we need to add 3 components: First, we need add an additional view that enables the user to access the feature. We decided to add another tab into AnSiAn, since this feature is different from the previous features. Secondly, we need to extend the transmission chain for an additional modulation that implements the signal synthesis described above. The last step is to extend the reception chain with an SSTV demodulation. However, we did not manage to complete the demodulation part. We have a first version of a demodulation class, but it is still having some problems. Therefore, we describe our approach and the problems we faced.

3.4.1 *User Interface*

The user interface for SSTV is added as an additional tab and is kept very simple. It is shown in Figure 14. The user can chose an image

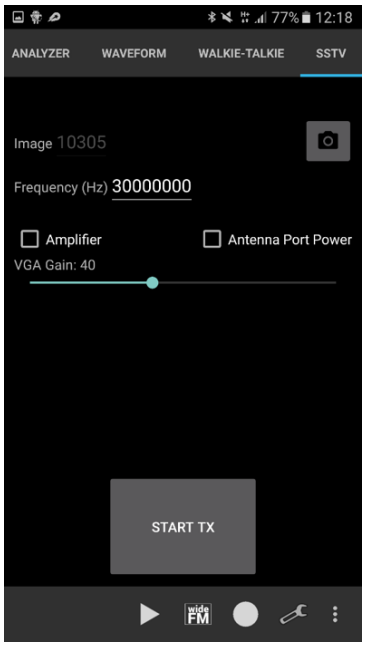

Figure 14: AnSiAn SSTV View - Tx only

from his gallery, adjust the HackRF transmission settings (frequency and gain settings) and then start transmitting the image. Since the demodulation is not working yet, we did not include it into the UI.

3.4.2 *Modulation*

The SSTV modulation is done in the model.modulation.SSTV class which is a subclass of Modulation. Before we can start creating the signal, we need to convert the image, which is probably a png or jpg, to a suitable format. Luckily, the Android SDK contains a Bitmap class that is capable of parsing such an image, scaling/cropping it to a suitable size (120x256 in our case) and converting it to gray scale pixels (bytes).

To implemented the SSTV modulation, we decided to reuse the fmmod from Section 3.2. We first create a float array containing the frequencies we want to create, then scale it to a value between 0 and 1 and finally call the fmmod with a frequency deviation of 2300 Hz. For example, for creating a frame sync at a sampling rate of 1 MHz, we create an array of length 30000 and set each value to 1200.

The time domain of a modulated signal recorded at the transmitter (i.e. perfect signal) is shown in Figure 15. It shows the start of a SSTV frame. At the beginning, there is a frame sync, i.e. a sine at 1200 Hz for 30ms. This frame sync is followed by the first line, which lasts 61.6ms. This first line consists of white pixels only, i.e. the frequency is constant at 2300 Hz. It is followed by a line sync (5ms) and the second line.

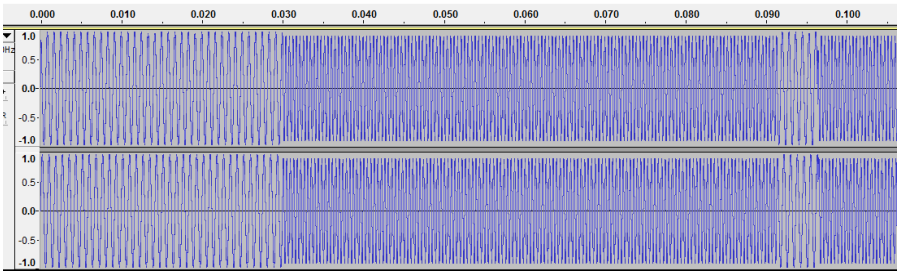

Figure 15: Time Domain of the Transmitted Signal - Start of a Frame (Frame Sync, Line, Line Sync, Line)

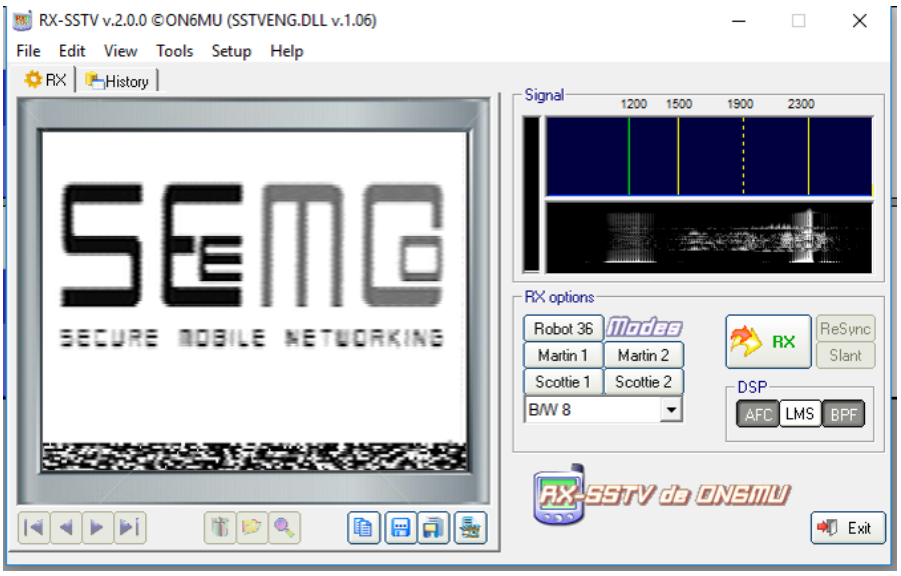

Figure 16: Validation of our Modulation using a Free SSTV Decoder [5]

For a good implementation took into account the findings from the previous sections:

- **Modulate small portions, instead of an entire frame**: To start transmission as fast as possible, we need to pass SamplePackets down the transmission chain as soon as possible. Therefore, we modulate each line into one SamplePacket.
- **Use a low sampling rate**: Since our signals use only very low frequencies, it would be a waste of resources to do the modulation with a sampling rate of 1 MHz. Thus, we use a much lower internal sampling rate (50 kHz) and upsample at the end.

For validating that our modulation is indeed correct, we used a free SSTV decoder [5]. We recorded the transmitted signal and replayed it on a PC.

Note: Since, RX-SSTV expects the samples from an audio input device, we imported our signal to audacity and played it there. Then, we forwarded the audio output to an input device using the "Stereo Mix" feature of Windows.

In Figure 16 we see the SEEMOO logo that is correctly demodulated by the RX-SSTV software. This implies that our implementation is correct.

3.4.3 *Demodulation*

We didn't manage to finish the implementation of the SSTV demodulation due to lack of time. However, the we sketch an approach to demodulate SSTV:

1. **Frequency demodulation**: First, we need to invert the frequency modulation to convert the time domain to the frequency domain. This could be done using a fast Fourier transformation (FFT), but that is computational expensive. As an alternative, Dennis Mantz implemented an approximative version of this for the FM demodulation (see model.demodulation.FM) that can hopefully be reused.
2. **Find sync sequences**: Next, we need to find the sync pulses to calculate the start of each line.
3. **Correct frequency shift**: The sync pulses can be used to correct a potential frequency shift
4. **Remove outliers**: Since our frequency demodulation is approximative, sometimes we get very large or very small values. Since we now the maximum and minimum frequencies in our signal, we can remove them easily.
5. **Decode**: Convert each pixel by taking the average frequency in this time frame and using this formula:

   $byte = 255 * (frequency - 1500)/800$.

3.4.3.1 *Problems*

When we tried to implement the SSTV demodulation in the last days of our project, we faced a couple of problems, which we want to document here. First, we used the generated signal (i.e. perfect signal - no noise, no attenuation, no shift) to implement the demodulation. The approximative frequency modulation works perfectly and it is easy to find the sync pulses. However, when this signal is transmitted and then received over an HackRF this looks differently. For illustration, we used the Matlab spectrogram function to plot the frequency domain of a signal over time. Figure 17 shows the spectrogram of a perfect SSTV signal. We can clearly differentiate the sync and the pixel frequencies, even with a low resolution. There are very few other frequencies in the signal.

Figure 17 shows the same signal recorded on the receiver side. We can still see the sync sequences, but we see two problems:

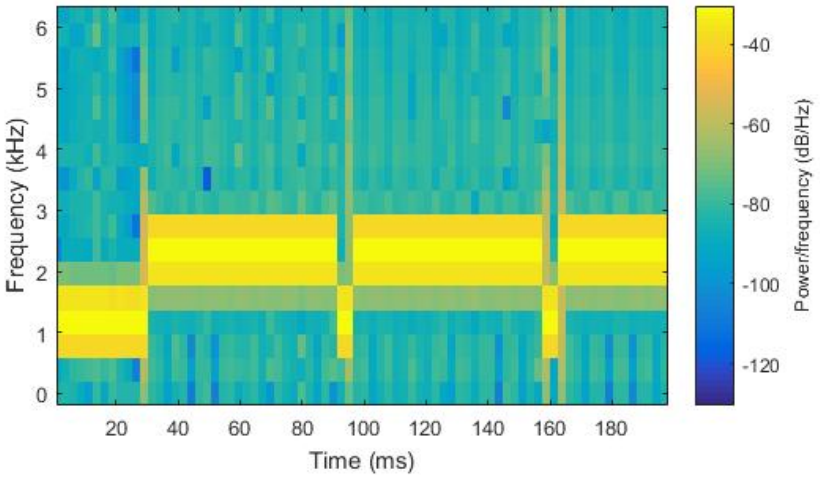

Figure 17: Frequency Domain of the Transmitted Signal - Start of a Frame (Frame Sync, Line, Line Sync, Line, Line Sync, Line)

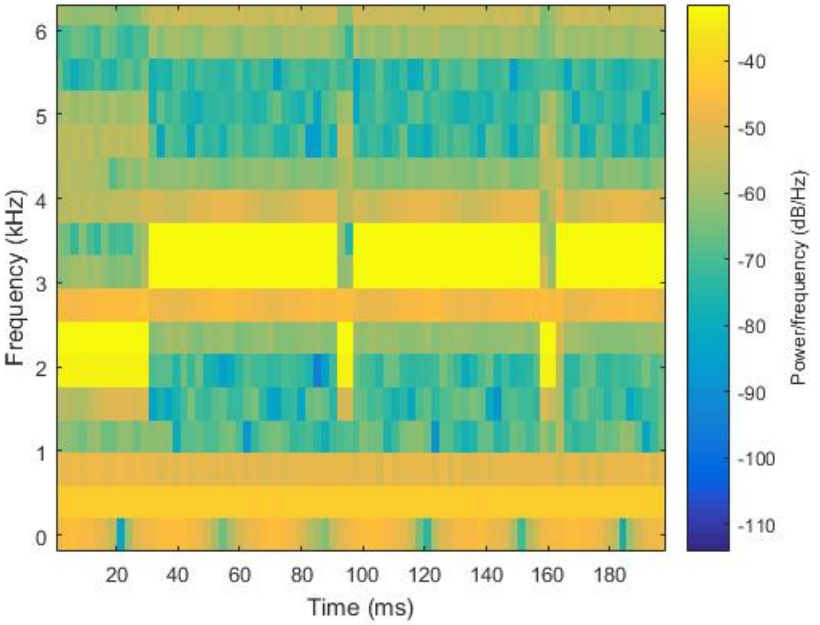

Figure 18: Frequency Domain of the Received Signal - Start of a Frame (Frame Sync, Line, Line Sync, Line, Line Sync, Line)

- **The frequencies are shifted**: The sync pulses are a little bit over 2kHz, although both HackRFs are tuned to the exact same frequency. This is not a huge problem, because we can correct that offset later. Depending on hardware and channel conditions this offset can be smaller or bigger.
- **There are other frequencies**: The signal contains other frequencies as well (although there were no other signals), especially low frequencies (<= 1KHz).

These problems cause the approximate frequency demodulation to produce bad results. Actually, the results look pretty random to us. Maybe, this can be corrected by applying a well designed bandpass to the signal.

4 CONCLUSION

This project mainly focused on the extension of AnSiAns' transmission functionality using the HackRF One. Primarily, we implemented amateur radio communication modulations. We added four larger features to AnSiAn:

Firstly, we generalized the transmission chain and enabled the real-time modulation of a wide variety of signals. We validated that the transmission capabilities of the previous versions are still working. We even improved them, because the transmission is now starting immediately.

Secondly, we added the RDS transmission capability, which enables the user to operate an own radio station which broadcasts RDS metadata like the station name. The feature combines multiple modulations techniques that can also be useful to implement other modulation schemes.

Thirdly, we integrated a Walkie-Talkie view into AnSiAn, that allows a user to transmit and receive speech signals recorded from the integrated microphone. The modulation was implemented from scratch, while for the demodulation we reused modules of older versions of AnSiAn with slight modifications. The main challenge here was to integrate the transmission chain and reception chain into one working piece that allows fast switching between reception and transmission.

Lastly, at the end of the project we started implementing SSTV, an amateur radio modulation scheme to transmit images using only a very low bandwidth of less than 3 kHz. The modulation and transmission of SSTV signals was integrated into AnSiAn which allow the user to pick an image from his device gallery and transmit it on a specified frequency. We didn't manage to integrate the demodulation of SSTV signals into AnSiAn. However, we started with a first prototype of a demodulator which still has some problems. We described the current problems and hope that it will be completed in a later version an AnSiAn.

Having implemented all four features, we can conclude that signal processing is very cumbersome on Android, mainly because of three reasons: missing library support, computational constraints and time-consuming testing and debugging. We spent hours on finding minor problems, that would be found much easier in a traditional signal processing setup.

We are actively working with Dennis Mantz to merge back parts of this project into the original RFAnalyzer (which was the basis of

AnSiAn in the first place). RFAnalyzer is a much more popular app with a lot of followers and forks. We think, that it would be great to provide the features to a large user group and that it will be received very well.

4.1 FUTURE WORK

Although, implementing signal processing on smartphones might require more effort than implementing the same functionality with other technologies (e.g. Matlab or GNU radio), it is still worth the effort, because it is much more portable and accessible. Therefore, we believe that AnSiAn should be continued and we propose the following features for the next versions:

Firstly, in our opinion it is required to redesign the structure of AnSiAn, because the code base grew over time and is very complex and unmaintainable at the moment. We especially dislike the heavy use of global preferences throughout the application, it makes the code very confusing and dependent on the Android system. Additionally, we are not sure if the use of an EventBus is a good choice here, because it is very hard to understand the control flow of the application while providing very little dependency reduction.

Secondly, we suggest to further extend the SSTV feature. The first step would be to implement the demodulation which requires to optimize the frequency modulation. It will require quite some work, but we believe that it will be possible. After this part is done, it should be easy to add more SSTV Modes (higher resolution, RGB, ...).

For more ideas on what to implement next, it might be a good starting point to look at what users want to be integrated into the original RFAnalyzer (see `https://github.com/demantz/RFAnalyzer/issues`).

BIBLIOGRAPHY

[1] *Android Developer Documentation: Audio Record*. Accessed: 2017-01-30. URL: https://developer.android.com/reference/android/media/AudioRecord.html.

[2] *Android Developer Documentation: Shared Preferences*. Accessed: 2017-01-30. URL: https://developer.android.com/reference/android/content/SharedPreferences.html.

[3] *Android Signal Analyzer Sourcecode*. https://github.com/matze765/AnSiAn.

[4] RDS Forum. *The new RDS IEC 62106:1999 standard*. IEC. 1999.

[5] *Free SSTV Decoder*. Accessed: 2017-03-12. URL: http://users.belgacom.net/hamradio/rxsstv.htm.

[6] S. Kreis and M. Grau. *AnSiAn - Android Signal Analyzer Documentation*. Secure Mobile Networking Lab, TU Darmstadt, 2016.

[7] D. Mantz. *RF Analyzer for Android*. https://github.com/demantz/RFAnalyzer.

[8] D. Mantz and M. Engelhardt. *AnSiAn - Android Signal Analyzer Documentation*. Secure Mobile Networking Lab, TU Darmstadt, 2016.

[9] *Octave Communications Package fmmod*. https://octave.sourceforge.io/communications/function/fmmod.html.

[10] J. Pritchard. *Newnes Amateur Radio Computing Handbook*. Elsevier Science, 2016. ISBN: 9781483106090. URL: https://books.google.de/books?id=1o-GDAAAQBAJ.

[11] *Verordnung zum Gesetz über den Amateurfunk, 2005*. Accessed: 2017-01-30. URL: https://www.juris.de/purl/gesetze/_ivz/AFuV.

ERKLÄRUNG

gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich die vorliegende Secure Mobile Networking Project Documentation ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. In der abgegebenen Arbeit stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, 16. März 2017

Matthias Kannwischer