University of Victoria

Department of Electrical and Computer Engineering

ELEC 350 - Communications Theory and Systems

Final Report

# SDR PSK31 Transceiver

Report Submitted December 10, 2013

Jason Bens
V00766510

## Abstract

This experiment develops a 20 meter PSK31 transceiver using a software defined radio, Python, and GNU Radio. The radio successfully received and decoded PSK31 messages, but was unable to transmit and accomplish a QSO. Nonetheless, this demonstrates the ease of development of a software defined radio transceiver, compared the the development of a similar conventional radio.

## Introduction

Software Defined Radio (SDR) is a rapidly developing component of communication systems. SDR differs from conventional radios in that the majority of the radio's functionality is implemented digitally, relying only on a high-bandwidth analog front-end for sampling. SDR offers significant improvements over conventional radio in that a digital system is nearly immune to component drift, is much more energy efficient, and can be easily reconfigured.

## Problem Statement

This experiment develops a PSK31 transceiver in the 20 meter band. The analog front-end chosen is a Softrock RXTX Ensemble. The software is implemented using a mixture of GNU Radio and Python.

## Theory

The front-end of an SDR is primarily used to perform complex downmixing of a real signal to allow it to be sampled by an analog to digital converter (ADC). The amount of downmixing required depends on the sampling hardware used. The Softrock RXTX Ensemble assumes the user will be sampling with PC audio card, a common piece of equipment accessible by most users. Thus, the frontend must filter and downmix the signal so that it occupies, at most, the range of frequencies from -24kHz to 24 kHz. The PC audio card can then sample the signal with no risk of aliasing, allowing the SDR to further process the signal to extract a message. On the transmit side, the front-end does a similar operation, upmixing the PC audio output to the frequency of interest and performing any power amplification needed.

The software portion of the SDR receives the quadrature inputs i(t) and q(t), the left and right channels of the audio input, and applies a map function to extract a message. The map varies for different protocols. This mapping ability is the prime strength of the SDR, as the map can be changed to change the receiver type, without modifying any hardware. To transmit, the software can take a message and apply a similar map to generate the quadrature outputs I(t) and Q(t) and output it via the PC audio output using the left and right channels, where it is then amplified and transmitted.

## Methodology

The PSK31 transciever was implemented using GNU Radio and Python. To pass messages from the controller, the python program, to the radio, the GNU Radio application, TCP sockets were used. All signal processing, such as IQ compensation, filtering, and IF mixing, were implemented in GNU Radio, while varicode encoding and decoding was implemented in Python.

## Results

The PSK31 transceiver had mixed results.  The receiver functioned very well, with many QSO's overheard in the 20 meter band.  However, the transmitter was very problematic.  No contacts were successfully made.

## Discussion

The inability to make a contact was confusing.  Examining the transmitter output on a spectrum analyzer, the transmitter was behaving as expected.  When trying to make contact, it seemed as though the transmitter was not being heard by the other party.  This belief is motivated by the observation that other operators would often cut in over the transmitter, even when calling CQ on an empty piece of spectrum.  The transmitter is capable of one watt of output, more than sufficient to make contact.  Additionally, in tests within the lab, it was possible to transmit a decodable message across the lab.  Thus, no explanation is proposed for this problem.

## Conclusions

This experiment developed an SDR PSK31 transceiver.  The receiver portion was demonstrated successfully, while the transceiver, while working in the lab, did not seem to work properly on-air, preventing any QSOs from being made.  Nonetheless, this is not suspected to be a failure of SDR, but rather some mistake in this particular implementation.  An interesting direction for future work would be to develop an internet-connected transceiver.  Similar systems already exist, such as webSDR.  The direction I would like to persue is somewhat smaller in scope, creating some kind of interface allowing me to keep the radio in the cold ham shack with a computer performing the signal processing, while sending messages from someplace less cold, possibly my bed.  The current code, using TCP sockets, seems like a good start, and can probably be developed further into something like my envisioned project.

## Appendix 1

This project was rife with problems, from start to end. The only part that wasn't a hassle was constructing the board. GNU Radio, despite having been around since 1998, has terrible documentation, a common problem across many open-source projects. Getting components to work properly often involved either a long trial-and-error process, or diving into the source code searching for any kind of internal documentation, followed by a long trial-and-error process. As an example, consider GNU Radio's tagged messages. Initially, this looked like a promising way to send and receive data outside of GNU Radio. After wasting several hours trying to decipher how this could be done from one-line documentation snippets, It was concluded that there is no way to actually add tags to messages from GNU Radio Companion, and that this approach should be abandoned. This decision was finally reached based on information that came from a blog post completely unaffiliated with GNU Radio that happened to be discussing an earlier project using SDR.

Another problem was the unnecessary complexity caused by the use of GNU Radio Companion for development, rather than using the GNU Radio libraries directly. GNU Radio can be a powerful tool, but GNU Radio Companion offers only the barest margin of this. There is no clear way to run arbitrary code not encapsulated by a block, which means that if any functionality is required that is not currently offered, a custom block must be created. The approach in this lab was to not do that, and rather to pass data into and out of the GNU Radio application using sockets. This is not really ideal, as it prevents a tight integration between the two projects. Additionally, due to GNU Radio Companions lack of support for any kind of decision structure, some code that logically fit in the GNU Radio application had to be implemented in the python project. For larger projects, this would create an unmaintainable mess. In the future, development will likely proceed using the GNU Radio libraries directly.

One final difficulty was the lack of support for the Softrock kit in Linux. A driver does exist, but it is not installed on the lab machines, and is therefore not useable. One of the most important parts of the driver is that it allows the Softrock's PTT line to be toggled programmatically. Without the driver, the PTT line either had to be toggled by using a second computer running Windows, or by shorting the PTT line to ground manually.