Advanced Computer Systems 2017/2018 Project– Spectrum Analyzer

Omer Zentner

1. Description:

A spectrum Analyzer for sub-1Ghz frequencies, using TI's Launchpad-cc1350XL.

The Board is used to sample RSSI values of requested frequency ranges and communicates the results to a PC front end.

There are 2 working modes, using either a direct serial connection, or a Python front end.

The main front end is the Python one, enabling saving samples to files, and later viewing of saved records. The direct connection mode was used for initial work and is maintained to provide an alternative mode of work.

When taking samples, a live feed of the results is displayed in the console.

The user can configure various parameters of the sampling: The frequency range, the resolution of the scan, how long to sample – either in terms of time, or of how many iterations of the range to do, and also to set a delay between each iteration of the range.

Results are saved in two types of files – a private format, used for offline viewing of records, and as CSV.

Saved results can be viewed using another part of the frontend, in two modes: a frame by frame view, showing each range scan separately, or as a colored contour of the complete sample.

Results viewed offline use matplotlib for representation, which enables more features like zooming and saving outputs as pictures.

2. Required Installations and Setup:

Direct Serial:

For this mode, all that's needed is some terminal program which can connect to serial ports.

When working on the project, I used Putty.

Python Front End:

The python front end is written for Python 2.7.

The program uses 2 external packages: PySerial and matplotlib.

Installing the packages can be done using pip:

pip install pyserial

pip install matplotlib

Board code:

The board itself should run the project's code.

To do so you can Import the project into Code Composer Studio and burn it to the board.

3. How to Run:

Note: Instructions are for a Windows 7 machine, connecting the board using a usb cable.

First thing's first:

Connect the board to the PC, using the usb cable

Direct Serial:

1. Find out which serial port the board is connected to:
   Once the board is connected, open *Device Manager*
   Open *Ports (COM & LPT)*
   You should see 2 XDS110 ports. The one we'd like to connect to is the *Application/User UART.*
   The name of the port is the name in the parenthesis (e.g. COM4)

2. Set the following properties for the terminal connection:
   Name ("serial line"): the name you found in step 1.
   Speed (baud): 115200
   Data bits: 8
   Stop bits: 1
   Parity: None
   Flow control: None

3. Open the connection & reset the board (button next to the usb connector)
   You should now see a message asking you to choose a working mode.
   Type '1'

4. Insert requested parameters according to on screen prompt

5. A live feed of the results will be printed to the console.

Python front end ("serial_comm.py"):

1. Run the program

Once the program starts it will prompt you to reset the board.

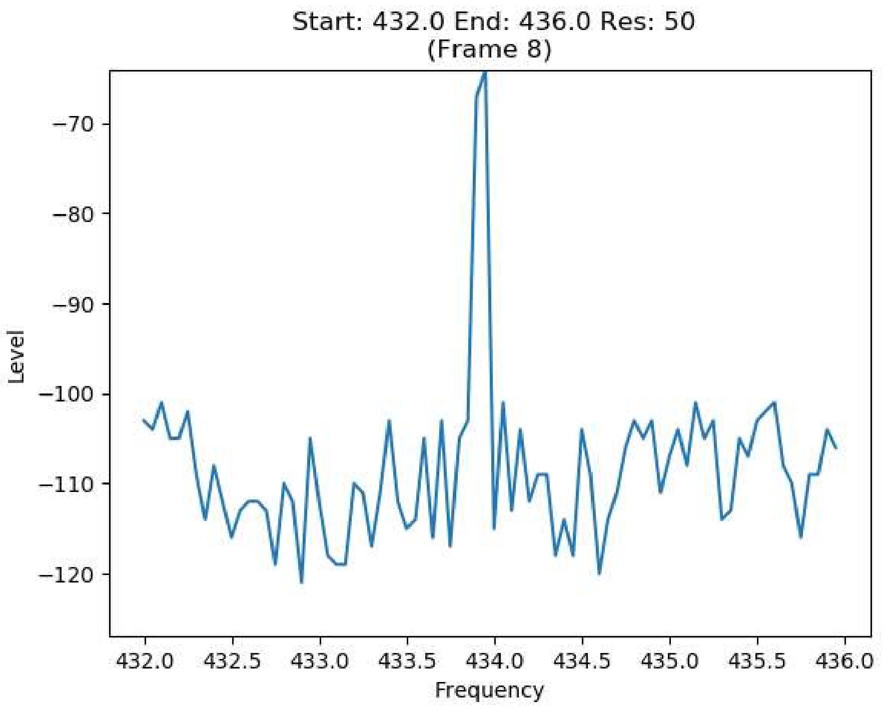
2. Insert requested parameters according to on screen prompt.

3. A live feed for results will be shown

4. **Saving results:**
By default, all samples taken using the python front end are saved in *C:\My_RF_samples*
Results are named based on the date and time when they were recorded.
Results are kept in two formats: a “private” one (.rfs), used by the record viewer program, and also as a CSV file – so the results can be easily used with any datasheet program.

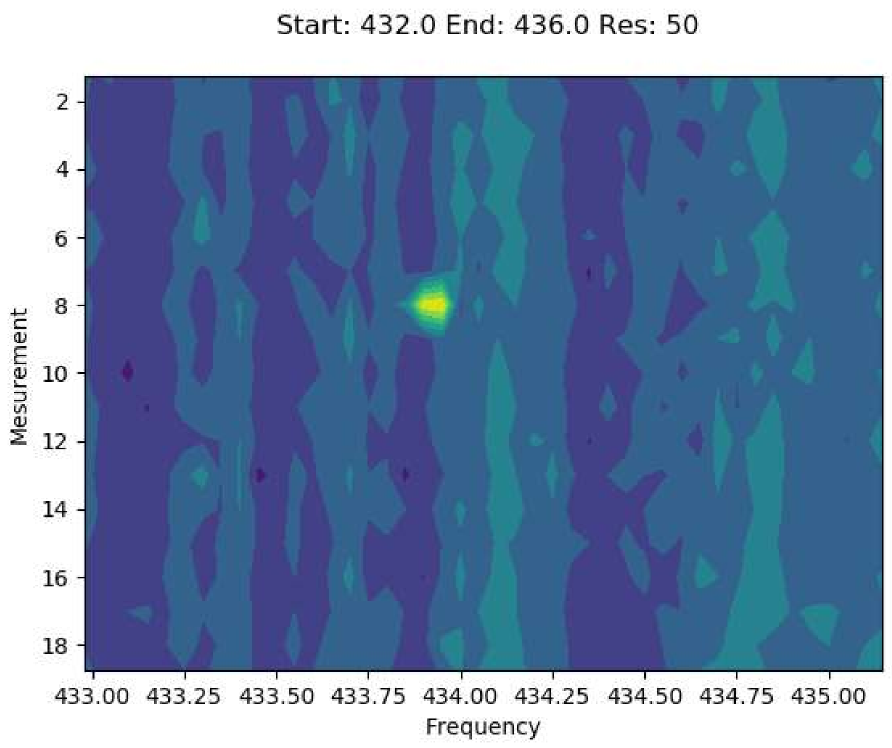
5. **Viewing Saved records**
In order the view saved records, run *record_viewer.py*
The program will show you a list of available records from which you can choose.
After selecting a record, you will choose a viewing method.
Two viewing methods are available: *frame-by-frame* or *contour*

The frame by frame method shows a 2D graph of a single spectrum measurement, and you can navigate forward and backwards within the measurements in the sample.



In the picture: One frame from a sample containing a capture of a car key fob

The contour method shows a spectrogram of the entire sample, where colors indicate the frequencies values.



In the picture: the contour view of the same sample. The yellow part is the car key fob signal.

6. **Main challenges:**
    a. **Working with the RF functionality:**
    In order to get RSSI measurements, I've used the ger_rssi() direct radio command.
    In order for the command to return a correct value, the RF core must be in active RX mode when the command is called.
    In order to do that, I've used the RX_TEST command, which sets the RF core into RX mode.
    One challenge in that regard was that using RF commands without Smart RF Studio:
    It seems that the main use cases for the RF core are for a limited number of set frequencies, which is appropriate for communication applications which use one or few frequencies.
    TI's recommendations are usually to use Smart RF Studio in order to configure the RF setup and copy the results into the project.
    In my case, I needed to programmatically change the commands structures within my code, when moving between frequencies, and so instead of using Smart RF studio, I've went over the code and user manual documentation of the commands and added and edited them within my project.

    One later challenge in that regard was to understand and set the radio to work correctly for different frequency ranges: At a certain point I've discovered that I was only able to get

readings around 433Mhz, even after I was already manipulating the RF_SETUP command to set my appropriate center-frequency and receive bandwidth.
It turns out that I also needed to change the LO divider value, which is the value by which the basic Oscillator frequency needs to be divided in order to get the appropriate frequency (e.g. the oscillator's frequency is about 4Ghz, and so for 433Mhz, a 10 Lo-divider value is used, while for 868Mhz, the value is 5). After making this change I was able to get readings for almost the entire sub-1Ghz spectrum, though, as only a subset of LO-divider values are supported (according to documentation, and also partially confirmed by testing with unsupported values), there are still parts of the spectrum for which I couldn't get a reading, which seems to simply be unsupported.

b. Communication with the PC frontend:
The main challenge in this regard comes from having the application consist of two independent components that need to interact with each other and maintain a consistent state between them.
The communication channel I've used was a serial connection between the front end and the board, using the UART module for the board side.
What I've attempted to do in this regard is to define my communication protocol in a consistent way that will be easy to implement correctly on both sides.
I had some issues with it along the way, but nothing too bad.
At one stage, I wanted to improve my board's code – the main thing I wanted to change was the amount of information kept per sample, and the amount of data transferred between the board and the PC.
My initial implementation stored the frequency of each measurement along with the measurement and would also send both the frequency and the measurement to the PC.
As the work is done in batches of iterations of the same frequencies range, that seems wasteful. So, I've changed the code so that for a range of frequencies I would only keep the frequencies data once, as I can infer which measurement is of each frequency.
This change required changes on both sides, as it changed the communication protocol a bit, but It went without too many issues.

c. Plotting the output:
I needed a way to view the samples, both live and at a later stage, offline.
As a baseline I've created a basic ASCII contour showing real-time values, which runs from the board itself, so it can be viewed even without a front-end program.
This was before I've started to work on the PC front-end, what I supported at this stage was connecting to the board using a direct serial connection (I used putty for that), and having all relevant code run on the card. That mode is still maintained as an alternative option.
Currently, the basic representation is actually implemented twice – the original version is implemented within the board's code itself and is used for the direct connection mode.
After adding the python front end, I've also attempted to use PyPlot for the real time view, but It was rather challenging to get it to work the way I wanted, especially at high rates of data. As a result, I've decided to implement a similar real time view for the front-end side.

PyPlot is used for offline record viewing, as it produces very nice graphs, and adds nice capabilities, like zooming and saving graphs as picture files (see examples below).

d. File saving and offline viewing:

The file saving part wasn't much of a challenge as it is quite easy to do from python code.
In order to enable offline viewing, I've shared code that extracts the relevant measurement information from the data received from the board, so that the data itself can be saved in "raw" form in a file, which can later be parsed by the offline viewing tool in the same way it is parsed by the main front-end.
Later, I've added the capability to output CSV files as well, to enable easy porting of the results to other programs.

7. Possible Future enhancements:

There are many things I thought it would be nice to add to the project, but I didn't get to.

Following are a few of those:

a. Adding a GUI to the python front end
b. Adding analysis features (peak/signal detection, capture & retransmission, …)
c. Adding a BLE support and front end