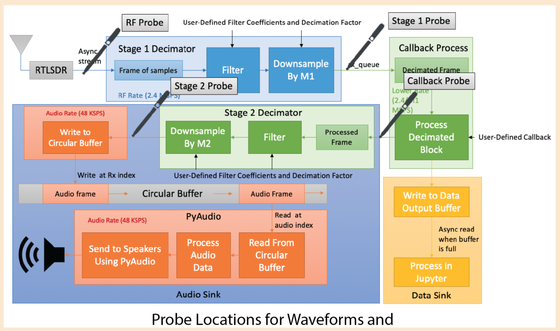
# Real-Time Software Defined Radio Using Python

**Andrew Smit (andrew@yosmitty.com) and Mark Wickert (mwickert@uccs.edu)**
**University of Colorado Colorado Springs, Colorado Springs, CO 80918**
**Jupyter Notebook sample in docs at https://github.com/mwickert/scikit-dsp-comm**

## Introduction

- We describe a software defined radio Python package that can be used to interface with radio hardware for real-time streaming applications
- Designed for high-level rapid prototyping development
- Utilizes Jupyter environments (notebook and lab)
- Interfaces with PyAudio for real-time audio message demodulation
- Out-of-the box tunable FM receiver example

## System Block Diagram

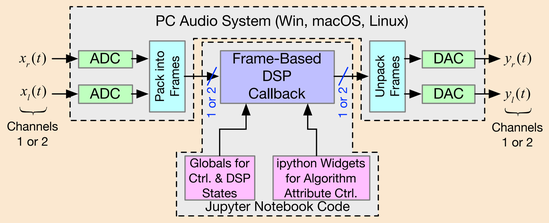


Probe Locations for Waveforms and

- The process begins with the RTL-SDR which receives over-the-air signals from an antenna and digitizes the signal at 2.4 MSPS
- We utilize Python's asynchronous functions to read in samples from the radio hardware
- The stage 1 decimator block filters the incoming signal and downsamples it to a lower rate. The user is able to define the filter coefficients and decimation factor
- The decimated signal is stored in a queue and read into a user-defined callback process:

```
def callback(samples,fs,user_var):
    '''
    User-defined callback function used to process decimated data
    in an RTLSDR_streaming object.

    parameters:
    ----------
    samples: frame of decimated data coming from the stage 1 decimator
    fs: decimated sample rate
    user_var: user-defined variable that is maintained in the class

    returns:
    -------
    samples: an ndarray of processed samples if using the audio sink
        or an array of data to be sent to the data sink
    user_var: updated user variable that will be updated in the class
        and returned on the following callback call.
    '''
    # Discriminator
    x = samples
    X=np.real(x)                    # X is the real part of the received signal
    Y=np.imag(x)                    # Y is the imaginary part of the received signal
    b=np.array([1, -1])             # filter coefficients for discrete derivative
    a=np.array([1, 0])              # filter coefficients for discrete derivative
    derY=signal.lfilter(b,a,Y)  # derivative of Y,
    derX=signal.lfilter(b,a,X)  #    "          X,
    z_bb=(X*derY-Y*derX)/(X**2+Y**2)
    return z_bb,user_var
```

- This is where the main processing algorithm can be implemented
- The processed samples can then be sent to an audio sink (if the message can be sent to an audio device) or a data sink (if you are receiving a file or binary sequence)
- The audio sink provides a second stage decimator where the user can define the decimation filter and decimation factor; The resultant rate of the second stage decimation filter should be the rate of the audio sink
- After the signal is decimated, it is stored in an audio buffer that is read out to an audio output device (PC speakers, headphones, etc.)

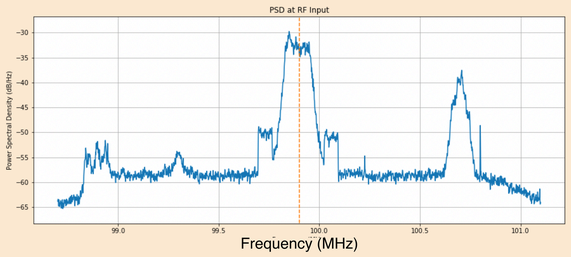



- The data sink can be used to store processed samples in an output buffer that is available to the Jupyter notebook where data can be further processed or saved out to a file; The data sink is used mostly in digital modulation types

## Probe Tools

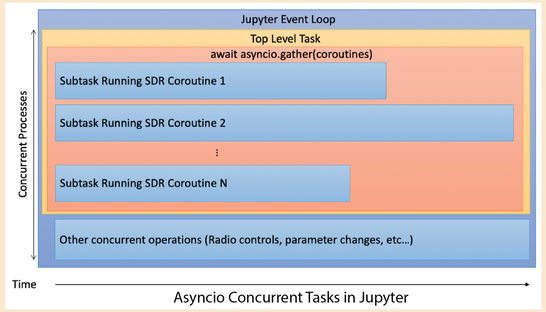
- There are several "probe" methods that allow users to asynchronously read out frames of samples at various points in the block diagram
- The RF Probe allows users to read samples coming directly from the radio; This data can then be plotted in the time domain or converted to the frequency domain for spectral analysis
- The stage 1 probe provides samples after the stage 1 decimator at rate $f_s/M_1$
- The callback probe provides samples after the user-defined callback
- The stage 2 probe provides samples after the stage 2 decimator if the audio sink is used at rate $f_s/(M_1 \cdot M_2)$

- A spectrum analyzer plot can be instantiated in the Jupyter Notebook at any of the probe points.




## Async Processes

- The package makes use of Python asyncIO concepts including tasks, coroutines, and the Jupyter event loop in order to run concurrent processes
- The package uses `asyncio.gather()` to run several coroutines concurrently while leaving the Jupyter event loop unblocked in order to run async radio controls and make parameter changes while the SDR streaming functions continue to run in real-time



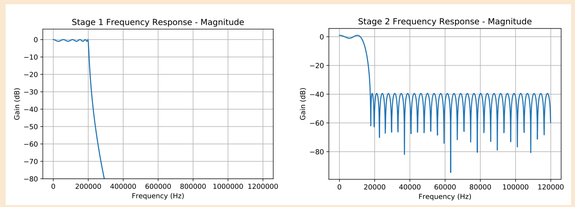

## Jupyter Widgets

- Jupyter widgets are used in the package in order to be able to start and stop radio/audio streaming and implement other controls (gain, center frequency, etc)
- The FM receiver example automatically instantiates widgets that can be used to tune the radio in real-time
- Users of the package can also instantiate their own widgets in order to control their software defined radio prototypes

```
sdr_stream = sdrh.RTLSDR_stream(0,rtl_buffer_size=2**15,audio_out=3)

sdr_stream.interactive_FM_Rx(88.7e6,40,3,2048,48000)

Sample Rate: 2400000.0
Center Frequency: 88700000
Gain: 40.2
Interactive FM Receiver
        Start Streaming        Stop Streaming
Status: Stopped
Gain (dB)                                  -9.3
f_c                                        88.7
f_c : 88.7 MHz

sdr_stream.show_logs()

Stopping SDR
Stopping Audio
Starting SDR and Audio Event Loop
Stopping SDR
Stopping Audio
```
Screen Shot from the Notebook (if not displayable in readthedocs)

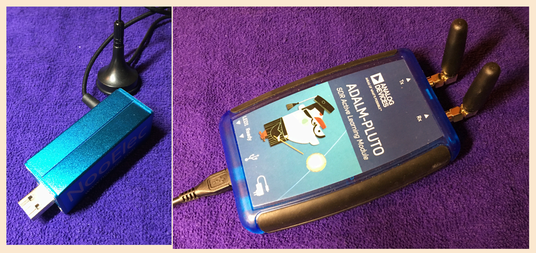
## Custom Filter Design

- The scikit-dsp-comm package contains helpful libraries for custom filter design that can be used for the decimation filters
- Both IIR and FIR filters can be designed and implemented depending on the application needs




## Hardware Support

- The package was initially designed for the RTL-SDR, but it is modular enough that other SDR hardware can be supported with only slight modification
- The Analog Devices ADALM-PLUTO SDR is now also a supported device
- The Pluto has both an Rx and Tx antenna, so full transmit-receive loopback type applications can be implemented




- While transmit is not currently supported in the package, this capability will become available for the Pluto
- The Scikit-DSP-Comm team is also moving forward with plans use the SOAPY SDR API