Software-Defined Radar

Paul Adams & Lincoln Young

8/16/2016

Individual Contributions

Paul Adams

o Imaging, set-up and configuration of the Raspberry Pi 2

o All real-time Python software o All Matlab prototyping and tool software

o All automating and utility bash scripts and Linux service interaction

o Testing and configuration of 3rd libraries on Linux

o All system testing and integration o Development of all algorithms that deviated from reference design

ABSTRACT

This report describes the design and implementation of software defined radar (SDR) that transmits and receives signals for range and detection. The design of the Coffee Can Radar referenced from the work of Dr. Gregory L. Chavat of MIT Lincoln Labortary. This inexpensive SDR encourages students to learn the fundamentals of Radio Frequency generation within the ISM band and its useful applications for range and detection. We have prototyped this design from both the hardware and software perspectives. The hardware design produces a tuned 40

ms ramp signal for transmission at 2.4GHz and receives reflected signals for amplification and processing to construct range indications. Synchronization of the transmitted and received data is to characterize the timing constraints and error handling of the signals. This implemented radar processing algorithms is executed on a Raspberry Pi microcontroller using Python scripts which had the ability to detect a target's range.

INTRODUCTION

This project began as a conceived innovation to the MIT Coffee Can Radar [1], hereafter referred to as the reference design. The reference design demonstrated three different radar modes - Continuous-Wave (CW) Doppler, Frequency Modulated CW, and crude Synthetic Aperture Radar (SAR) - with minor hardware adjustments and using different processing algorithms. The system was interfaced with a laptop in order to acquire a block of data and then process the data offline and show results in Matlab. A block diagram is shown for reference.

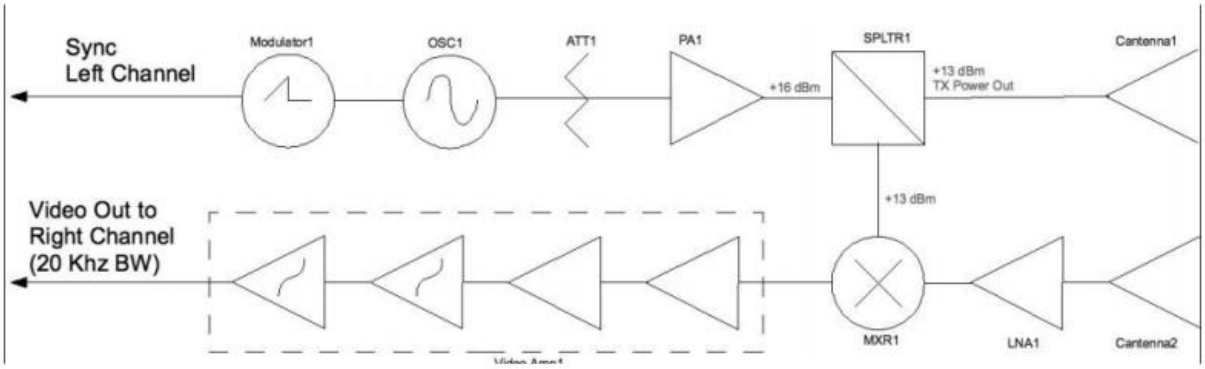


Figure 1: MIT Coffee Can Radar Block Diagram

Our goal was to implement two of these modes using streaming processing and human-in-the-loop mode switching, thus demonstrating a Software-Defined Radar (SDR) system. This project could be extended to interface the control software with waveform tuning hardware and demonstrate Cognitive Radar (CR). Our initial innovation to the reference design was to design hardware improvements and add a real-time control loop with network offloading of results. The control portion was chosen to be implemented on a Raspberry Pi 2 running Arch Linux for ARM whereas the reference design interfaced with a laptop and used offline batch-mode processing.

The modified high-level design is shown below for reference.

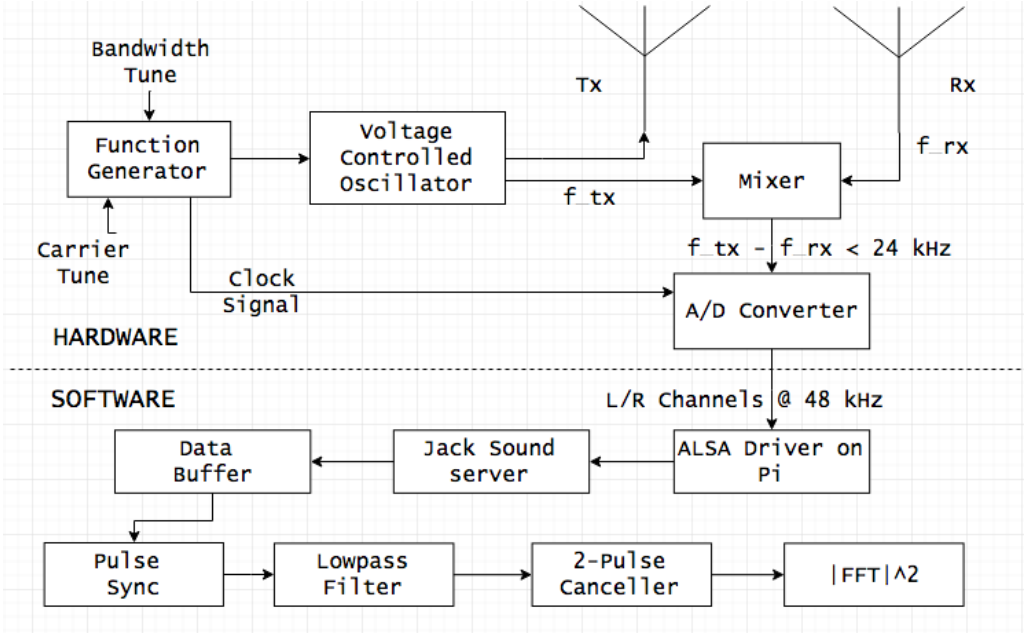


Figure 2: SDR Block Diagram

DISCUSSION OF PROJECT

DESIGN PROCEDURE

The SDR design can be summarized into two components:

1. The design improvements to the reference hardware system and

2. The software design

HARDWARE IMPROVEMENTS

The original hardware circuits consist of mix signal devices as well as specialized Radio Frequency modules that form the backbone of producing a tuned signal. This signal is transmitted at 2.4GHz via a coffee can antenna for range and detection of objects that will absorb and reflect the incident signal. The target areas for the hardware improvement are power supply, function generator and the video amplifier low pass filter.

The reference power supply is implemented with a breadboard and small pieces of connecting wires of size #28, American Wire Gauge. These interconnections are expected to carry a maximum of one ampere (1A). Therefore the conductor's reliability may be reduced for an extended period of time, perhaps 3-5 years. For a printed circuit board (PCB) equivalent conductor, the trace width and its implementation in either stripline or microstrip routing must be accounted for. The layout study has the 5V voltage rail uses a complete plane since most of its load devices require 5V. The 12V and 6V rails are designed for 20 mil trace width. This is acceptable for loads it supports.

On the bread board the function generator device has many of its discrete devices tightly connected. The risk with this implementation is shorting a Power to Ground connection. In the PCB layout the GND shape allows for better isolation from the Power connections. Hence the leads of the through-hole components are much shorter which would reduce parasitic capacitance to the function generator device.

The Video Amplifier circuit that receives the reflective signal and amplifies it is very cumbersome on the breadboard. The layout structure of the PCB shows the efficient placement of the discrete devices with the similar benefit of reduced lead lengths and a better return path for Signal_IN and Signal_OUT that is feed to the Raspberry Pi for processing.

We have used similar RF circuits as the reference design. This is sufficient for a prototype case study. It was demonstrated by Dr. Jim Carrol of National Instruments that the RF circuits and the antennas (Vivaldi – flat horn shape) can be created in the form of PCB for a total cost of $60. His presentation illustrated the advantage of circuit simulation and design analysis for cost optimization.

SOFTWARE DESIGN

The reference design provided Matlab scripts for ingesting a block of audio data and then processing to show results. There was no detection or tracking software included in this reference and neither did we attempt to develop sophisticated detection or tracking software. Rather, the data can be transformed and displayed as an image, which then allows the eye to perform the job of a target detector/tracker system.

Initial design began with experimenting and rewriting the reference Matlab code to handle processing the data as a stream and displaying the results in an animated waterfall image. This process was repeated for each of the two radar modes - Doppler and FMCW. These Matlab prototypes then became the reference point for developing Python on the Pi.

This process naturally led to the development of two tools which became essential throughout the development - software audio oscilloscopes that plugged into various source data formats. Eventually this settled into one program used to pull various results from the Pi to the development laptop and display the data in real-time in Matlab figures.

The second program is similar except that is fetches data from either the disk or the sound card and performs some processing and then displays those results in a streaming fashion.

Finally, algorithms were migrated to Python on the Linux system and development proceeded over remote shell only. During this phase, publishing results to sockets that could be read from my laptop using the tool mentioned above was critical for testing and integration.

The design procedure relied on having the reference design as a starting point, signal processing knowledge, an understanding of the mathematics and physics involved, and the ability to iterate

in rapid-prototyping environments like Matlab and Python until results agreed with expectations. We feel that this approach is suitable for research projects intending to demonstrate capability. While it is acknowledged that a more formal top-down methodology ensures success in a production environment, working in research and development, we believe that creativity and persistence produce demonstration results quicker and cheaper than the process-oriented nature of production.

SYSTEM DESCRIPTION

SPECIFICATION OF THE PUBLIC INTERFACE

Inputs

The transmitted signals can be channeled to either an object that is stationary or moving in a direction that is within its range. The reflected signals are received and amplified for algorithmic processing by the Raspberry Pi. Objects in our study included the motion of a moving vehicle, a class room wall and a person's motion. In each case both directions toward and away from the transmitted signal observed and processed thus indicating range with graphical plot. The temperature range for each case study varied between 64 F° - 75 F°.

Outputs

The Raspberry Pi does have the capability of running a desktop environment with graphics in order to display results, yet the overhead is significant and so the decision was made leave the Pi running in a headless manner and offload the resulting data over a local network to the development laptop for display. For test and debugging purposes, it was desirable to have the ability to visualize the data as it progressed throughout the processing chain. The system output interface is visualized below.

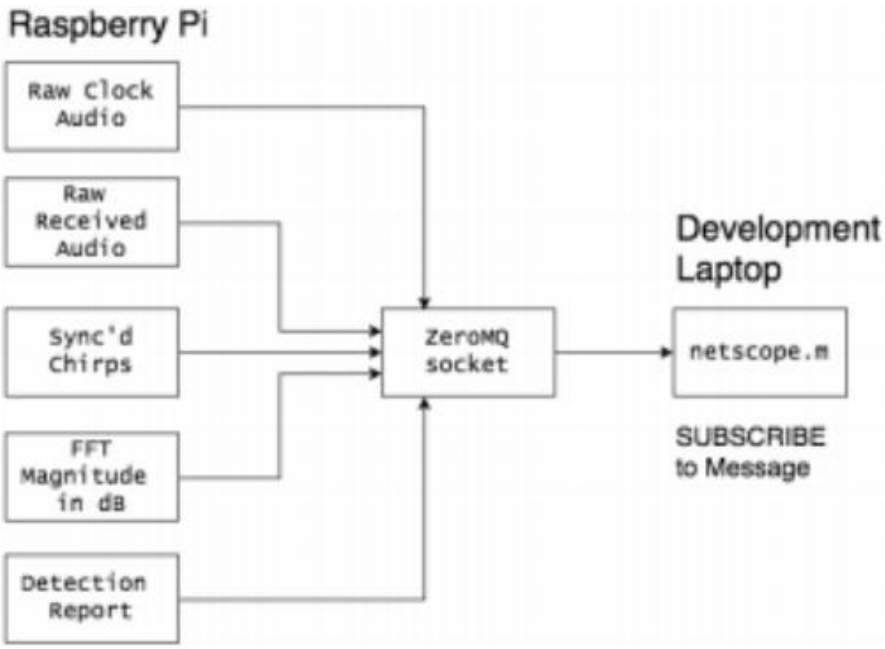


Figure 3: SDR to Development Laptop Interface

As seen, the program netscope.m takes a message name string for which a ZeroMQ socket is created which subscribes to that message type. The PUBLISH/SUBSCRIBE topology enables the Pi to artificially put all the data on the wire, yet only transfer the data that is requested. Matlab does not actually have a ZeroMQ implementation, yet it does expose a Python interface, through which ZeroMQ can be reached seamlessly.

In addition to the above the system also prints out status messages to the Raspberry Pi console.

Were the system to progress into a more productized instantiation, offloading reports could be achieved over a ZigBee link. Of course, this would limit the data rate to discrete detection measurements as opposed to the current toolset which allows viewing the data at any stage in the processing chain.

ALGORITHIM DESCRIPTION

Doppler CW

A Doppler CW system is able to measure the instantaneous radial velocity of a moving object. When an electromagnetic wave reflects off of a moving object, say a car, the wave is shifted in frequency by an amount proportional to the wavelength of the RF signal and the projection of the car's velocity onto the line from whence the wave originated. Doppler CW systems use a continuous sinusoid shifted to the carrier frequency. When the received signal is mixed with the transmitted signal, the difference is output. By performing a Fourier Transform on the received

data, over some period of coherency, the radar is able to measure the magnitude response of the data at various frequencies. These are then related to speed using the wavelength.

Frequency Modulated CW

In FMCW, a triangle wave is generated rather than a sinusoid. When the triangle is passed through a voltage-controlled oscillator (VCO) the ramp produces a linear frequency modulation (LFM) known as a chirp. This enables measurement of object distance from transmitter over the period of the ramp. As with Doppler, a frequency difference is measured and related to range by the speed of light and the bandwidth of the frequency ramp. However, for FMCW the coherent period is constrained to the period of the triangle ramp. Therefore, FMCW signals must be synchronized to the reference clock signal. This enables knowing the time the ramp was transmitted which can be differenced with the peak return infrequency for a measurement of distance.

Another difference is the dominance of stationary objects on the response spectrum. These are collectively referred to as clutter and reside at zero Hz (DC), though the energy bleeds into the nearby frequencies as well. If a radar is primarily interested in objects that move, clutter can be mitigated by taking a slow-time derivative of the data. Slow-time in that the time interval is measured in ramps instead of samples. By subtracting the previous response from the current response, much of the DC energy is removed and moving objects are left in the response. This clutter-mitigation strategy is known as a two-pulse canceller.

Timing Constraints

For a sensor of any type, the time scale is ultimately driven by the kinematics of the objects of interest in the environment. At some point a simplification must be made and a period established over which it is assumed that the environment is stationary. For the case of our SDR, the objects of interest are of the human-walking and car-driving variety. Each of these is slow relative to airborne jets and so our time scale has some margin compared to typical radars.

If we assume the upper bound of stationarity to be a car moving at 35 mph that translates to about 1 foot in 20 milliseconds and 5 feet in 100 milliseconds. Somewhere between 20 and 100 milliseconds we can reasonably assume our sensed environment is static. Our chirp ramps are tuned to last for 20 milliseconds, and so this becomes the fundamental unit of time for processing. Additionally, we can average over 1 to 5 of these pulses to smooth the output results and still have some confidence of the scene remaining roughly stable.

The other time constraint in this case is the ability of processing resources to handle the throughput requirements. Fortunately, the relatively slow speed of the objects of interest coupled with the fact that our system does not have the power to see beyond 1 km for a 10 square-meter target, means the information of interest is contained within a very narrow region of spectrum near DC. In fact, it is within the bandwidth of human auditory sensing. This pleasant coincidence results in an abundance of analog-to-digital converters with the requisite sample rates. This also

means that our incoming data rate of 48000 samples/second is manageable for modern processor chips. Ultimately, required processing throughput depends on the data rate and the Raspberry Pi can handle a few audio signal processing operations within the required time frame.

Error Handling

We can categorize the classes of errors as those which are induced by unexpected inputs, those induced by uncaught exceptions within the primary Python routine, and those induced by kernel scheduling, causing lag or loss of flow.

The first occur when the routine is unable to synchronize to the reference clock signal. This exception is caught by wrapping the sync block in a try/catch structure. If we are unable to sync, we want the routine to keep trying without falling apart. This reflects the software's inability to control external events, such as low battery power, fried circuits, or some other failure of the signal chain. Similarly, the routines downstream of sync need to predicate their execution on the indication from sync that everything is working. This is achieved with control flags once sync is achieved.

Additionally, each pulse iteration checks the period of the sync interval to validate stability and throws a syncLost flag if sync is lost. Therefore, acquireSync is the initial state to which the routine returns if exceptions are encountered. The second, exceptions within the main Python routine, are due to software bugs and unexpected corner cases. For our prototyping system, these are addressed using the built-in Linux kernel control wrapper called systemd. We stress that when the input is operating correctly and the kernel is able to keep up with the data processing and throughput requirements, there have as yet not been uncaught errors within the main loop.

However, in order to handle the unexpected, three systemd unit modules were written to indicate what actions should be taken when one of the main programs crashes unexpectedly. The modules can be enabled as services which allow them be started automatically after a reboot once the kernel boot reaches the point where the device drivers have been initialized. Additionally, event actions can be specified, such as OnFailure or OnWatchdog. Precedence may be set such that program two waits until program one has successfully initialized. In this way, we are leveraging the existing tool set within the wider Unix community to handle simple control flow and error handling. This is opposed to writing a custom routine to interact with the kernel scheduler. For a rapid-prototyped demonstrator, this reduces risk and cost. A production system might require more extensive effort to guarantee proper exception protocol and real-time scheduling priority.

This framework also enables handling the third failure mode where the kernel scheduling causes the audio server to fall behind resulting in audio discontinuities. In this case, the audio server, which is implemented using the third-party Jack library with ALSA as the device backend, is controlled by a custom systemd service which stops and restarts the server in the case where errors occurs due to overruns.

HARDWARE IMPLEMENTATION

The system hardware design consists of analog, digital and modular RF devices. These circuits are connected to coffee can antennas that transmit and receive signals for processing by the Raspberry Pi. In addition, these devices are supported with discrete components such as resistors, capacitors and trimmer potentiometers. Circuit theory principles and techniques are used to construct a breadboard layout prototype and a PCB layout.

The electrical, temperature and physical package information from each device was reviewed to extract critical information such as minimum and maximum operation conditions. The breadboard layout needed more focus since the number of rows and column are limited for the quantity of parts to be placed. This is definitely a disadvantage that could easily lead to potential shorts between power and electrical ground potentials. On the other hand, in software, the physical layout boundaries can scale accordingly optimum performance as cost permits.

The power supply circuit is constructed with low voltage regulator. This regulator is a three terminal device that requires a minimum of 6V at its input to produce a steady state output of 5V. Unlike the reference design we have implemented an input of 12V from eight AA batteries. The reference designed used 6V from eight AA batteries. This indicates the source input for the voltage regulator is connected in series for our implementation while the reference design used a parallel approach. Another delta between the two prototype designs is the power indicator LED. We connected the LED to output of the voltage regulator whereas the reference design used input. This makes a difference in the luminous intensity of the LED, approximately 9.5 mA compared to 2.5 mA. An application of ohm's law $[(Vcc-V_f)/R]$, where "Vcc" is the source voltage, "V_f "is the LED forward voltage and "R" is the limiting series resistor.

The function generator circuit is similar to the reference design. At the heart of this circuit is the linear amplitude modulation device, XR-2206. It is conducive for low power application with the capability to generate several types of wave forms. We have utilized the triangle wave signal to feed the two port cascaded network of the RF component chain. The signal is buffered and amplified before it is split where one half of the signal to be transmitted thru the antenna channel and the other half is mixed with the amplified received signal. No external tuning is done to the amplifier modules or mixer. The devices are all powered on the 5V rail that is supplied by the voltage regulator output.

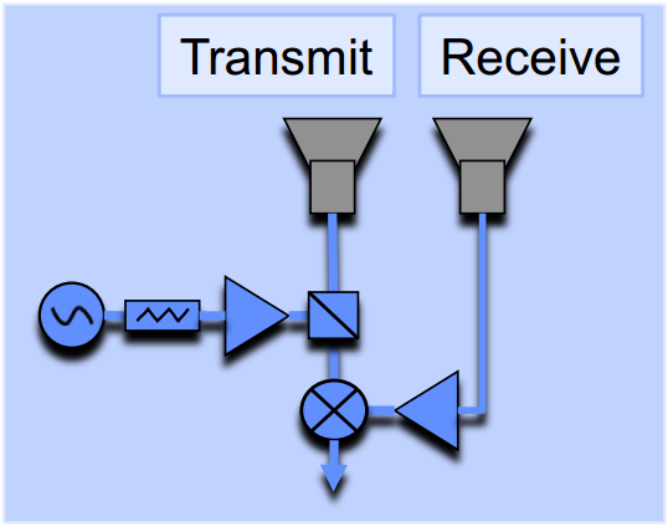


Figure 4: RF Components (Courtesy of MIT Lincoln Laboratory)

The Coffee Can antennas support coaxial cables to transmit and receive microwave signals. A time-varying signal (VTUNE) is applied to the transmit antenna. Hence an electrical current is induced on the antenna and generates electromagnetic radiation. This radiation occurs at the speed of light away from the antenna and is reflected when it comes in contact with any object that is stationary or in motion. The energy reflected from the object is absorb by the receive antenna and induces an electrical current that propagates as a signal on the coaxial cable. The initial length of each antenna was 2 inches. The length of each antenna reduced to 1.5 inches during the tuning process. The Coffee Cans structure provides cost effective method to design a "Circular Waveguide" antennas.

SOFTWARE IMPLEMENTATION

For brevity, we will focus on the Python real-time code here and not on the Matlab tools or prototyping routines. A flowchart is presented below for reference.

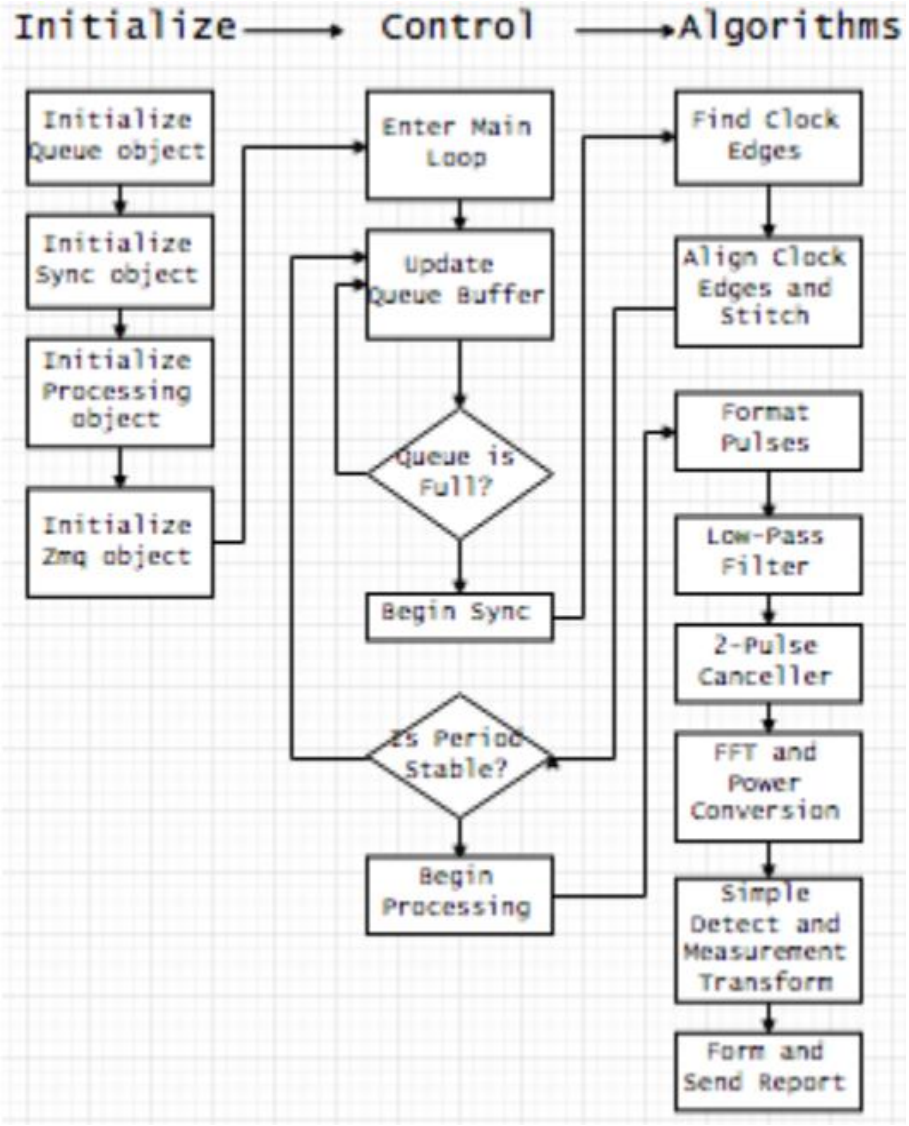


Figure 5: Software Flow Chart

TEST PLAN

There are four main areas of the hardware design that test coverage must be supported - The power supply, the function genitor, the video amplifier and the coffee can antennas. The order of testing must be done as previously tested. The expected results must align with the design specifications for each section for a fully functional and robust design within the temperature range of 64 F° - 75 F°.

The input source to the voltage regulator is required to maintain a minimum of 6V. The two 4-pack of AA batteries has three loads that need 12V rail, the function generator, the video amplifier LPF and the input of the voltage regulator. The power supply is expected to supply 5V consistently for at least four hours with in operation temperatures to support the function generator, the video amplifier LPF and the RF component link devices.

The function generator is powered on 12V, however its SYNC and VTUNE signals are pulled up to 5V. This is to ensure the output signal strength is sufficient to propagate along the conducting paths. Essentially this means a device is powered on a 12V rail and driving an output signal at a lower voltage. The receiving device (RF component) is rated to tolerate the 5V signaling from the function generator.

Like the function generator, the video amplifier LPF is powered on 12V. Three of the four operation amplifiers are used from the MAX414 low power device. The op amps are configured to use one as a gain amplifier and the two are cascaded to from a Low Pass Filter. The output of the LPF stage is driven into the Raspberry Pi microcontroller for processing. This output signal is 5V. Each of the feedback loops for the op amps are powered by 5V.

The Coffee Can antennas consist of 1.5 inches of #28 AWG is connected to a coaxial SMA junction that also connects to the 1.5 inches of coax cable from both the RF split module (transmit) and the MAX414 input amplifier. The antennas are placed 1.8” from the bottom circular surface of the Can. This is to ensure that a “Guide Wavelength” is achieved, that is, ¼ of the signal wavelength in free space.

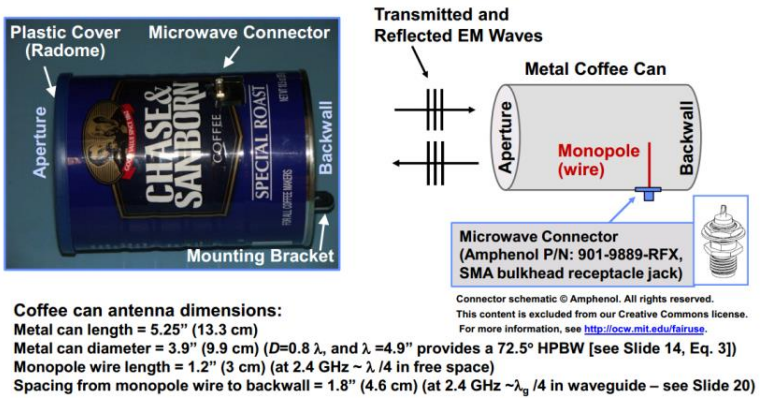


Figure 6: Circular Waveguide Antenna (Courtesy of MIT Lincoln Laboratory)

PRESENTATION, DISCUSSION, AND ANALYSIS OF RESULTS

The hardware prototype on breadboard layout enabled early testing with software, co-design.

ANALYSIS OF ERRORS

Construct and tested the circuits from a modular approach. Each subsection power delivery, function generator and the video amplifier LPF were verified using a digital multi-meter, an oscilloscope and and simulation of PCB stack up.

ANALYSIS OF WHY?

The Project may not have worked

Three areas the project may not have worked are the power supplied by the VR regulator, attempt to design a PCB, and the antenna.

We had deviated from the reference design with the power delivery of 12V to the input of the voltage regulator instead of 6V. This meant that the current this 12V was delivering was the same for the three loads (function generator, video amplifier and the voltage regulator). This meant more stress on the two 4-pack of batteries, thereby reducing their energy faster. In the case of a parallel connection where two loads would see 12V and the other load 6V. This indicates the LM2940 device would its share of current consistently as it is not shared directly with the other two loads.

The PCB design attempt began once we had proved out the breadboard prototype was known function delivering the VTUNE and the 15kHz SYNC signals. The initial work began with the circuit layout in Eagle Software. The goal was to design a 4-layer board, that is, with a stackup which consisted of a two signal layers and two power planes (GND and Power). The signal layers would be microstrip and power planes would be stripline. The intent also was to us surface mount components to reduce via count. Unfortunately the license purchased – Maker Version was not provide by the distributer (still pending – a refund). The Maker version allows for six layer design.

Initially the antenna monopoles for both transmit and receive coffee-can were 2 inches. We tested the antenna with movement of a person (toward and away directions) to observe a graphical output in Matlab.

Efforts made to Identify root cause Issues.

For the power delivery case we reviewed the reference schematic against our breadboard layout by measuring voltage at the input and output of the voltage regulator device, LM2940, every 10

minutes for an hour with a DMM and noticed that the output voltage remained steady while the input voltage (12V) diminished by 2.5%. This is acceptable since we are targeting a 4hr continuous supply which meets the +/-10% tolerance for 12V supply.

Appearance can be deceiving! The circuit layout using the Eagle free tool turned out very elegant. However, the limitations of the tool are its physical dimensions (3”x3”) and the layer count.

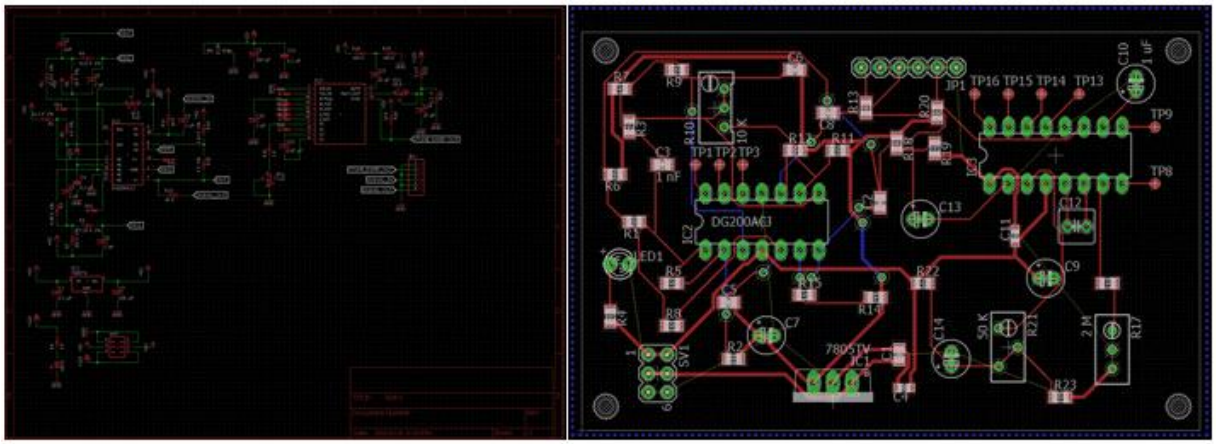


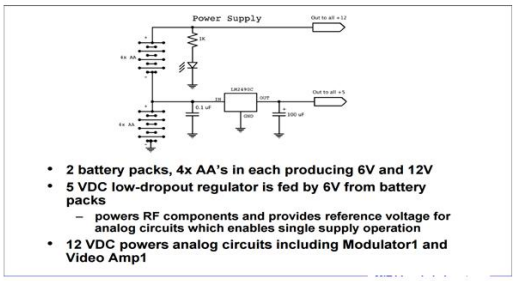
Figure 7: SDR Circuit and Board Layout with Eagle Software

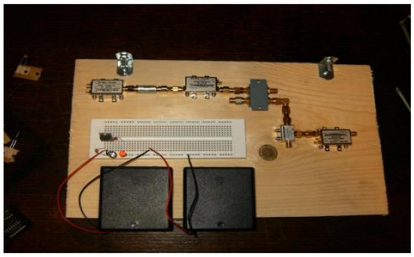
SUMMARY AND CONCLUSION

APPENDICES

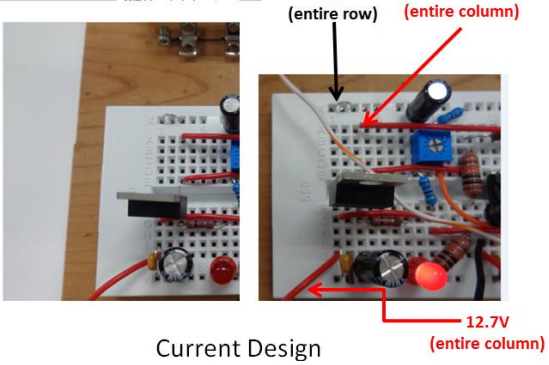
Prototype Build

Power Supply Circuit:





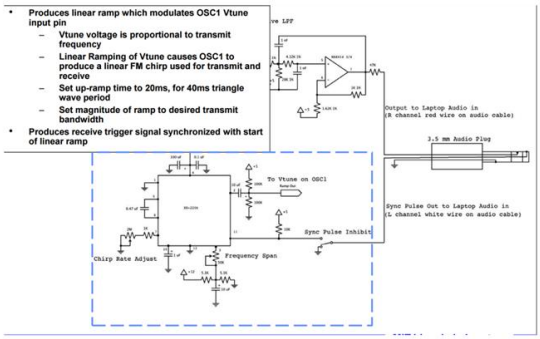
Reference Design

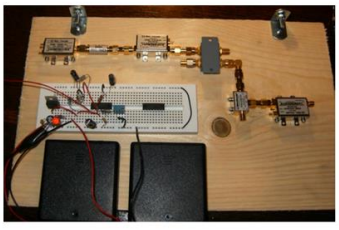


Current Design

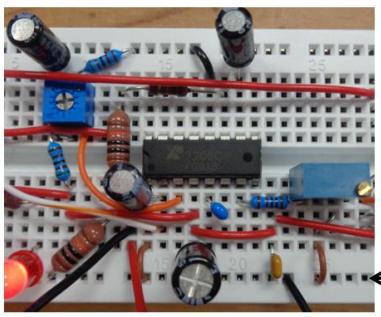
Function Generator:

Modulator1 Circuit:





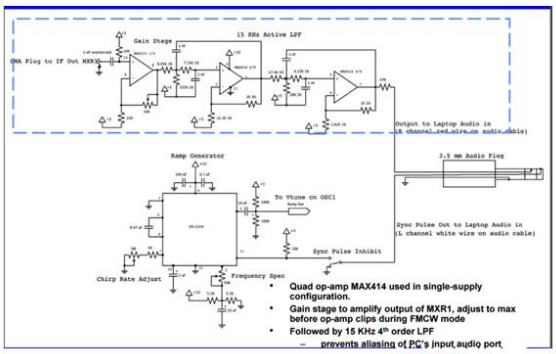
Reference Design

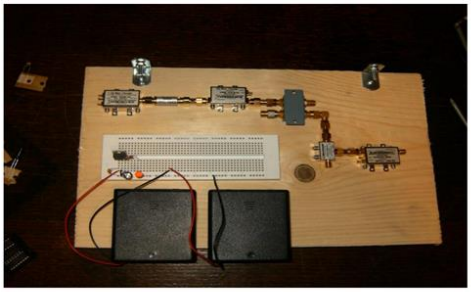


Current Design

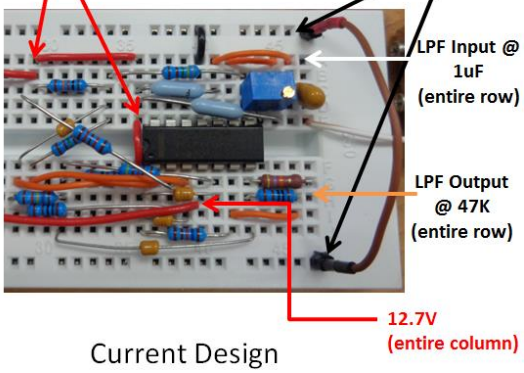
Received Signal Amplification and Filtering:

Video Amp Circuit:



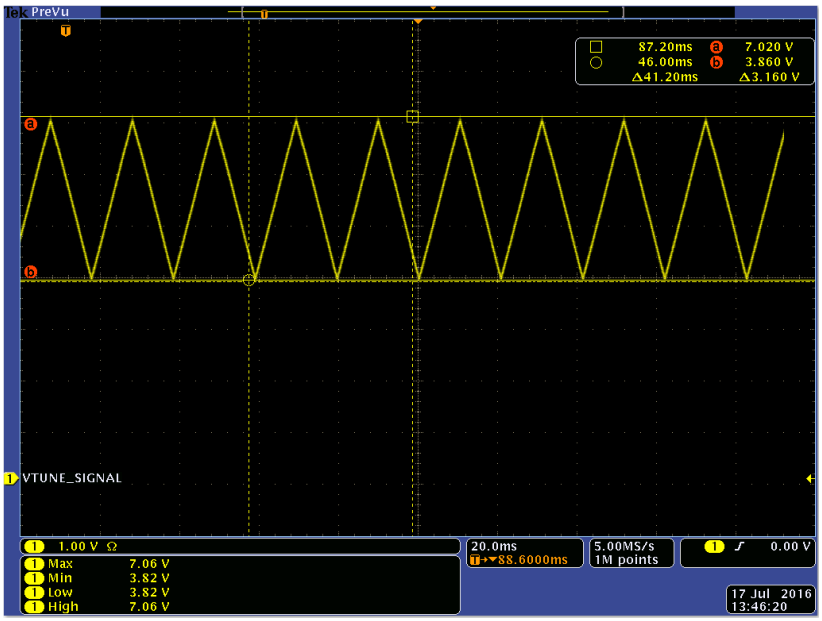


Reference Design

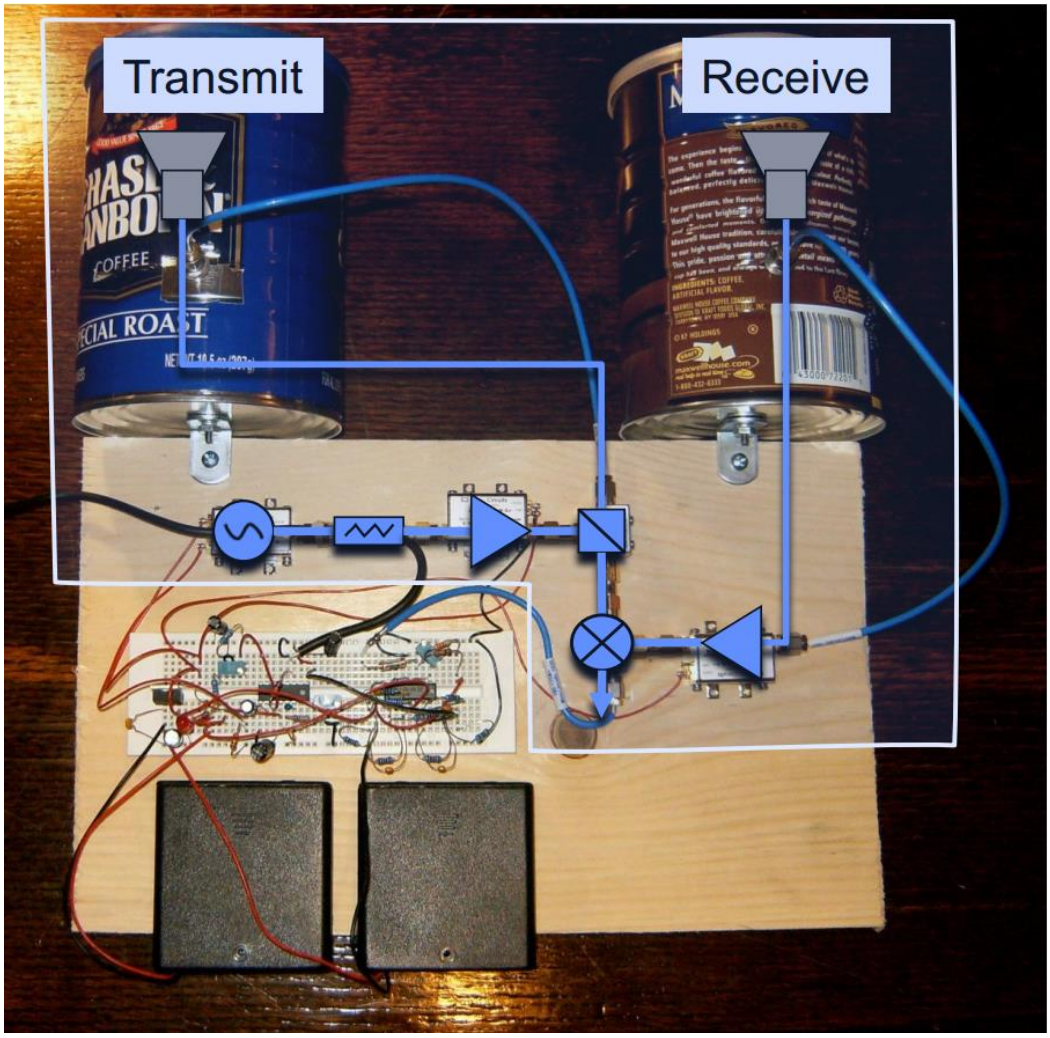


Current Design

VTUNE SIGNAL OUT: 40ms ramp per a cycle



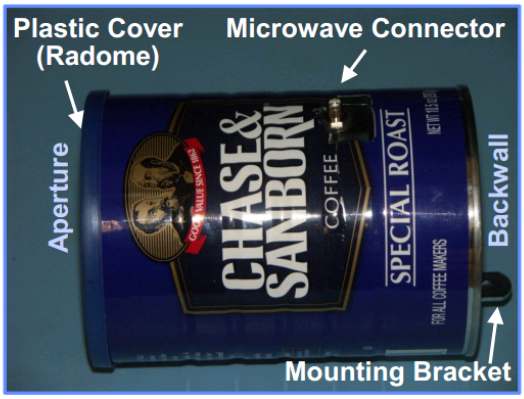
MIT LINCOLN LABORTORY COFFEE CAN PROTOTYPE

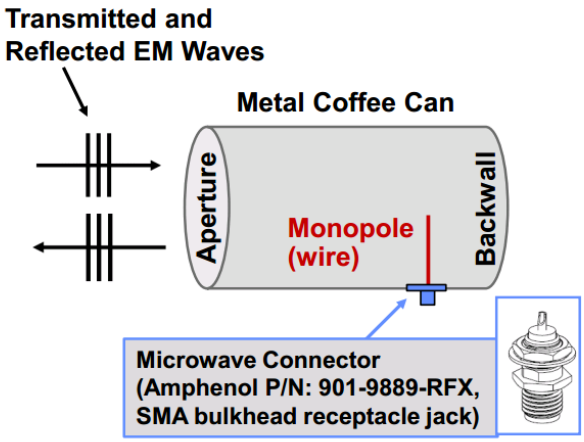


MIT LINCOLN LABORTORY COFFEE CAN PROTOTYPE

Example Metal Can Antenna Design for 2.4 GHz

Circular Waveguide Antenna



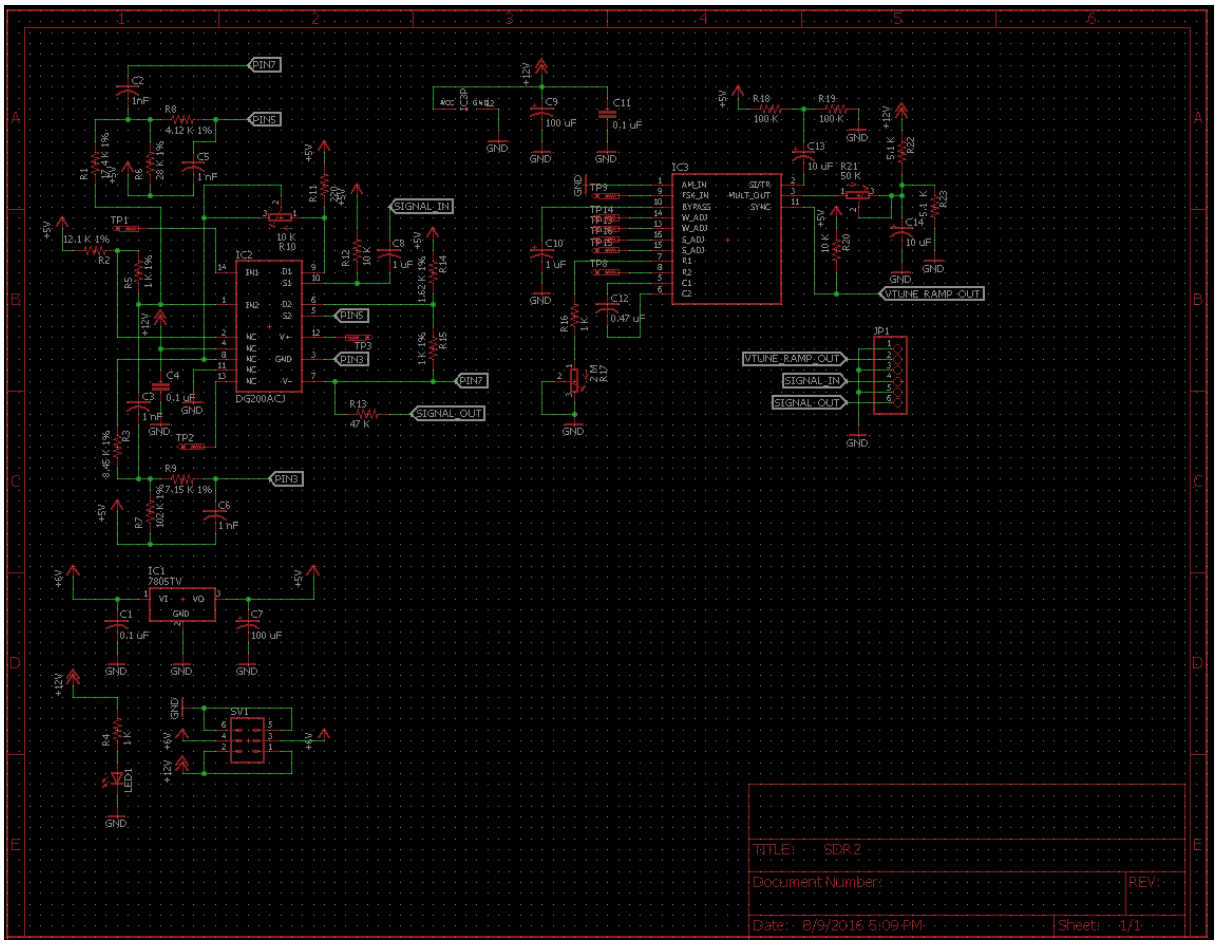


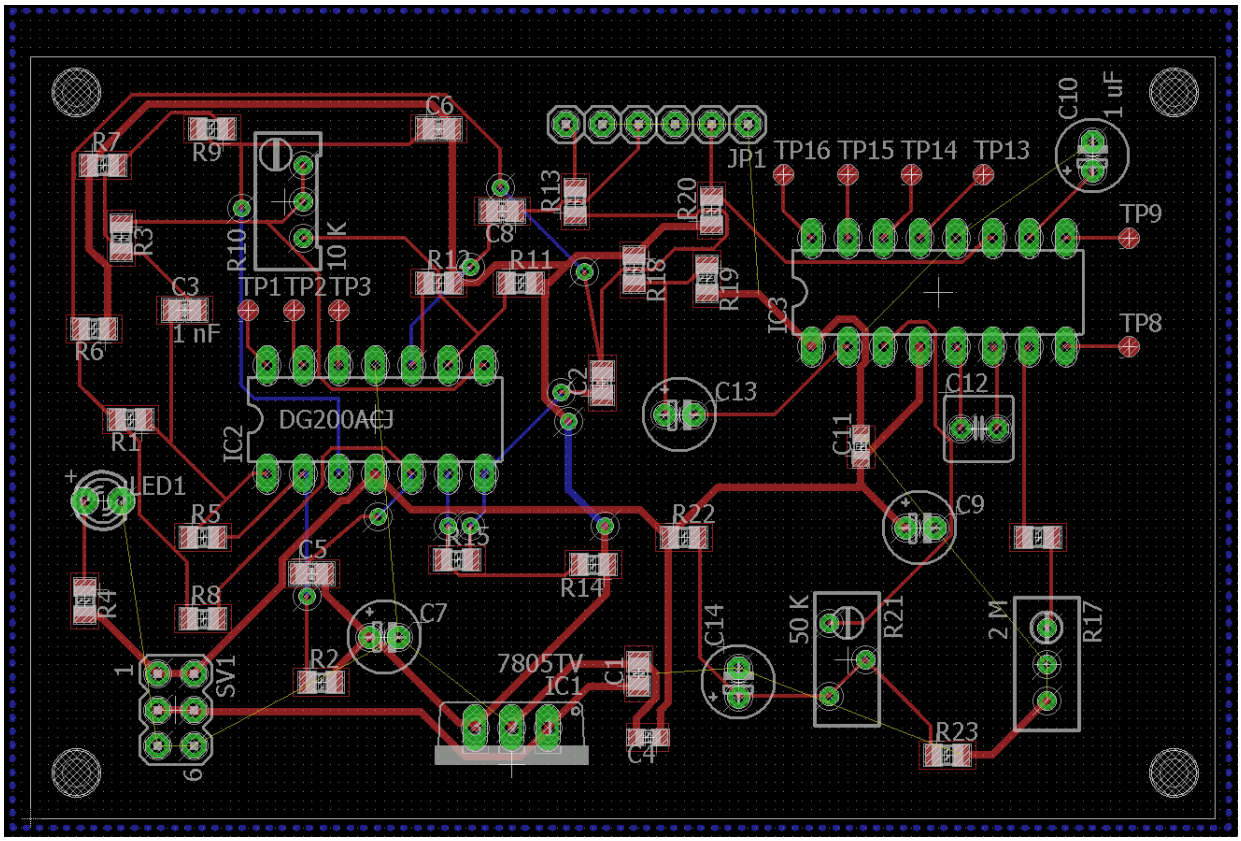
Coffee can antenna dimensions:
Metal can length = 5.25" (13.3 cm)
Metal can diameter = 3.9" (9.9 cm) (*D*=0.8 λ, and λ =4.9" provides a 72.5° HPBW [see Slide 14, Eq. 3])
Monopole wire length = 1.2" (3 cm) (at 2.4 GHz ~ λ /4 in free space)
Spacing from monopole wire to backwall = 1.8" (4.6 cm) (at 2.4 GHz ~λ_g /4 in waveguide – see Slide 20)

Ref: W.W.S. Lee and E.K.N. Yung, "The Input Impedance of a Coaxial Line Fed Probe in a Cylindrical Waveguide,"
IEEE Trans. Microwave Theory and Techniques, Vol. 42, No. 8, August 1994, pp. 1468-1473.

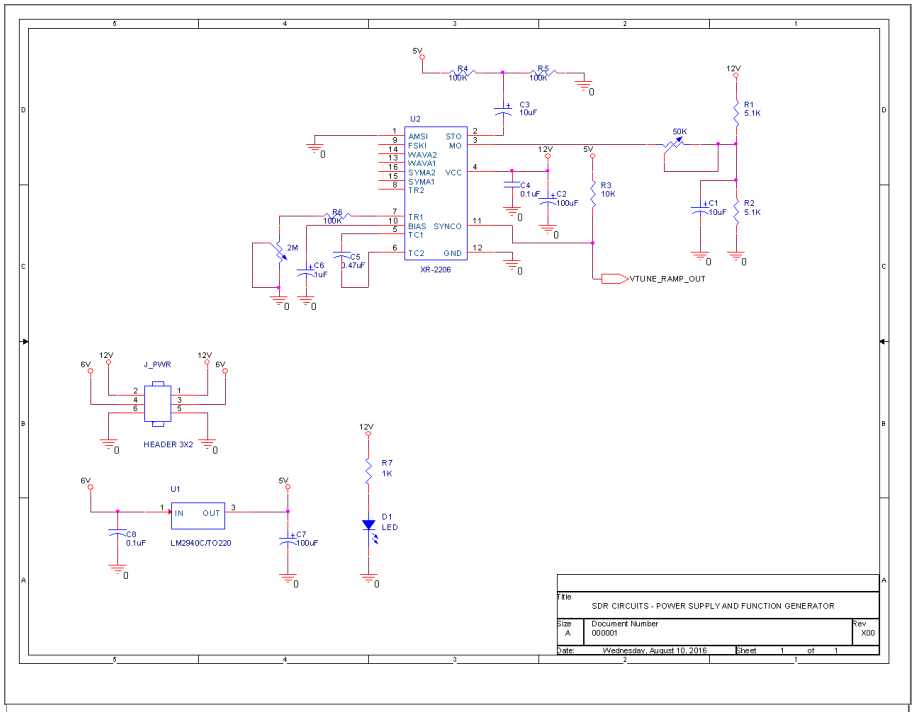
Schematics

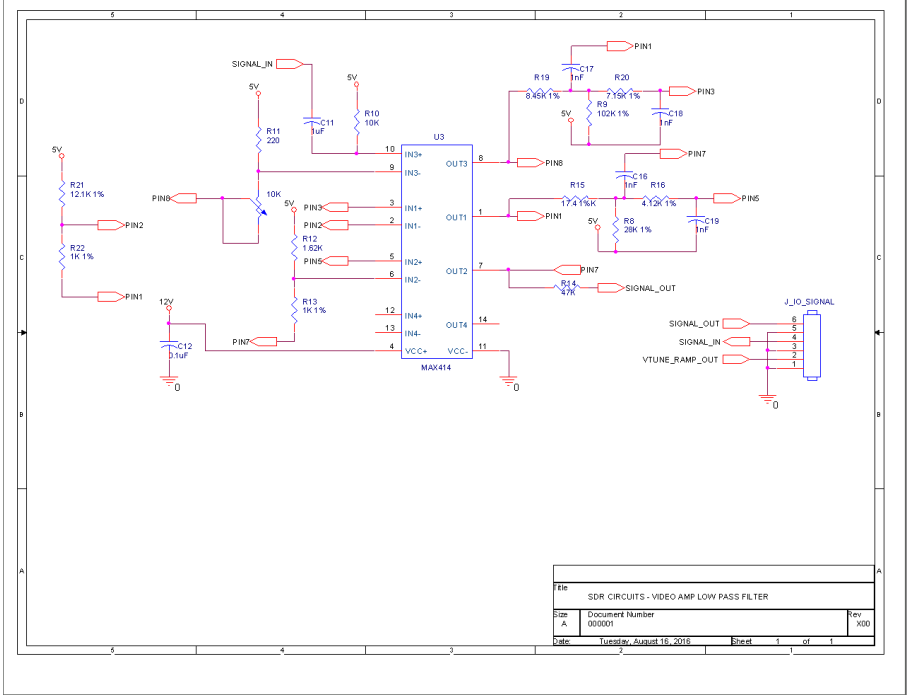
<u>PCB layout using Surface Mount Components:</u> Using Eagle CAD Tools (free version)



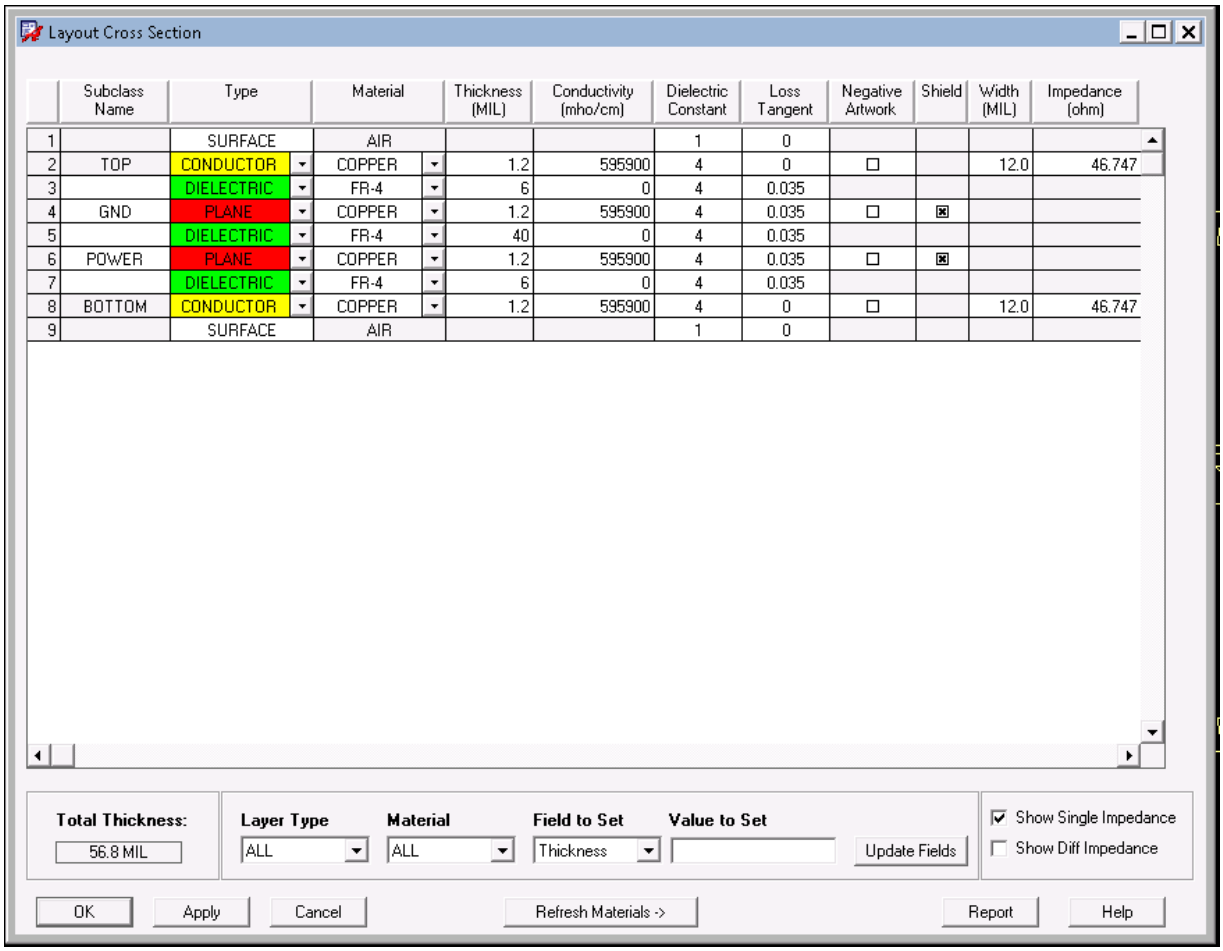

C6
R7
R9
JP1
TP16 TP15 TP14 TP13
C10
1 uF
TP9
R3
R10
10 K
C8
R13
R20
R12
R11
R18
R19
IC3
C3
1 nF
TP1 TP2 TP3
TP8
R6
C2
C13
C12
DG200ACJ
IC2
R1
C11
LED1
C9
R5
R22
C5
R15
R14
R4
R8
C7
50 K
R21
2 M
R17
C14
SV1
R2
7805TV
IC1
C1
R23
C4

PCB layout using Through-Hole Components: Using Orcad Allegro Tools

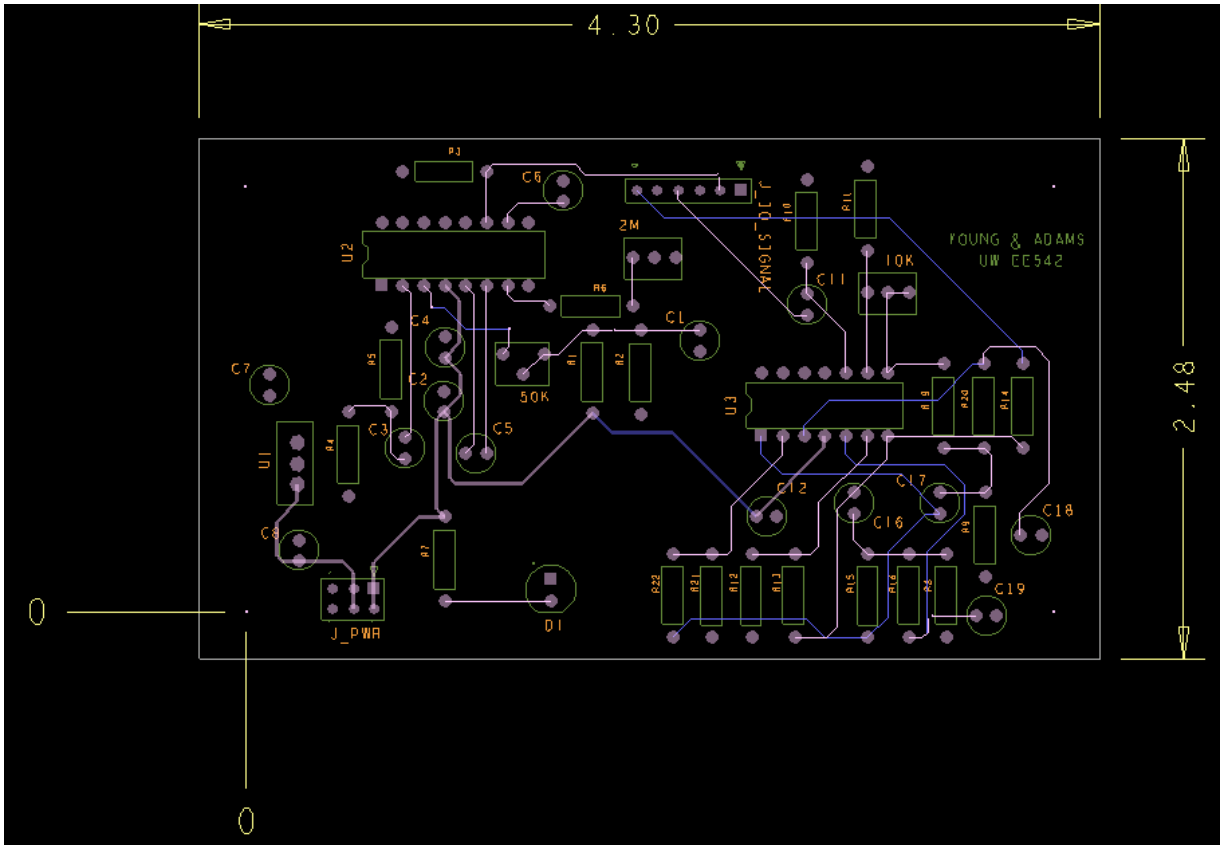




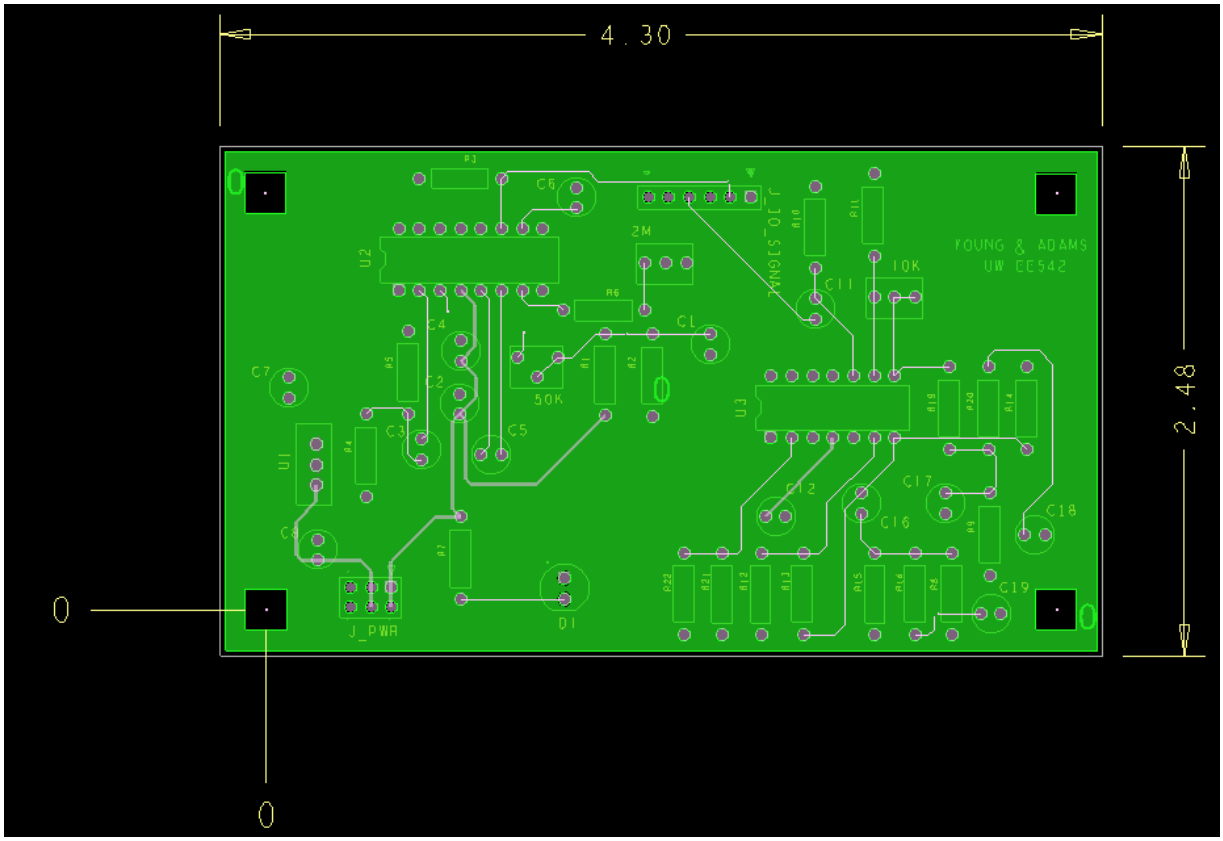
Proposed PCB Stack Up



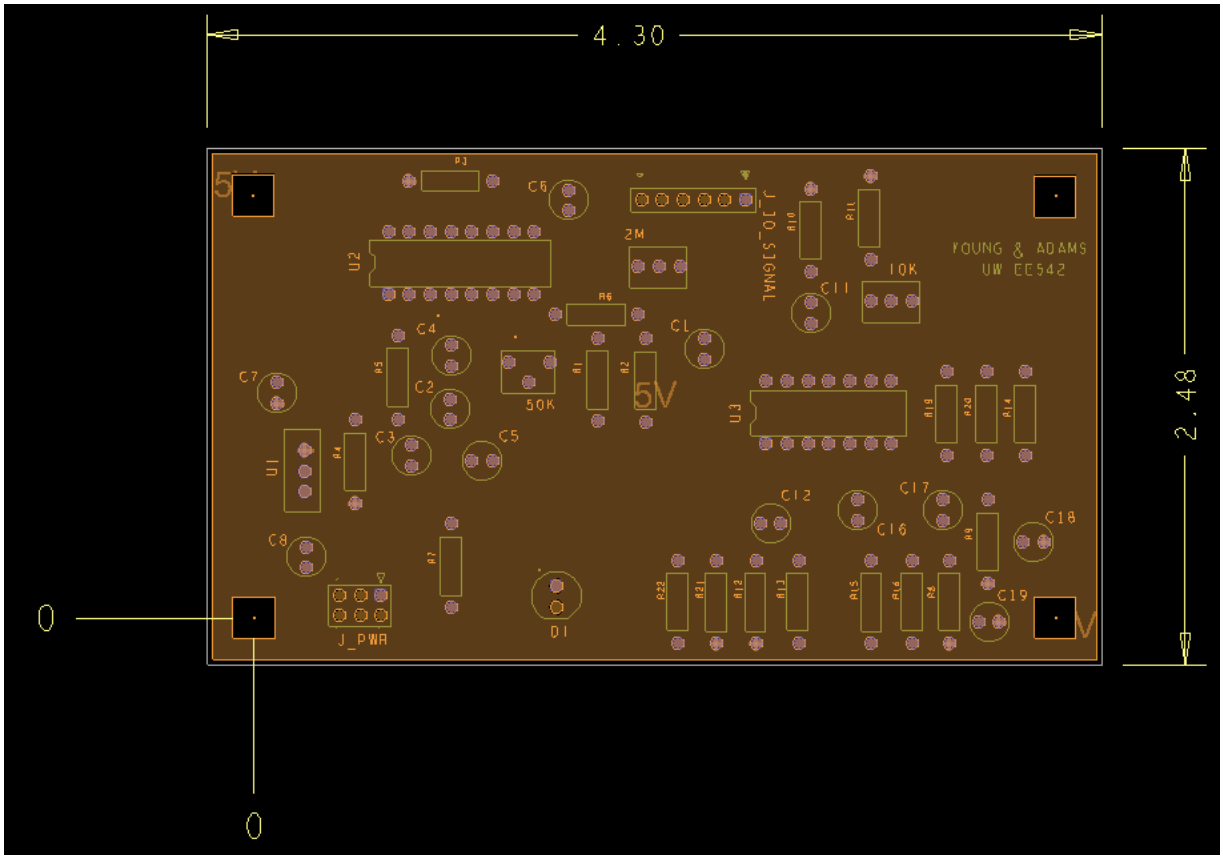
Top layer shows Pink Traces



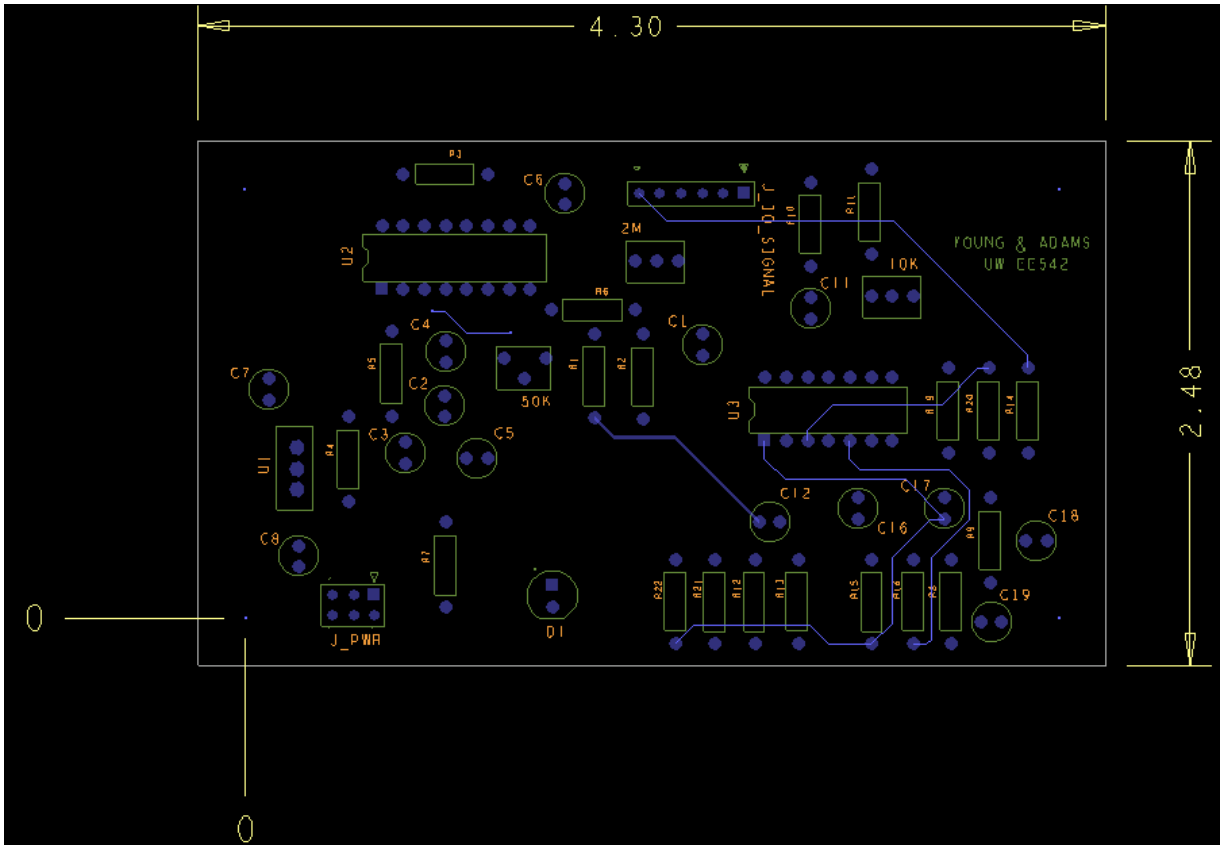
Top layer reference layer 2 (GND)



Layer 3 shows 5V Plane shape.



Bottom layer shows Blue Traces



References

1. TBD

2. TBD

Code Listings

Matlab Visualization Tools

```
% filename: netscope.m
% author: Paul Adams

function netscope(varargin)
    % parse input args
    args = containers.Map(varargin(1:2:end), varargin(2:2:end));
    %socket_ip =char(py.socket.gethostbyname('thebes')));
    socket_ip = get_arg(args, 'socket_ip', 'thebes');
    sub_topic = get_arg(args, 'sub_topic', 'raw');
    v2db = @(x) 10*log10(x);
    dev = init_data_device(socket_ip, sub_topic);
    not_init = true;
    figure(1); clf; figure(2); clf;
    i_frame = 0;

    % main loop
    while true
        % update data
        data = dev.recv();
        idx = strfind(char(data), ';;;');
        header = data(1:idx);
        data = data(idx+3:end); % remove header
        shape = regexp(char(header), '.*n_row: (\d+).*', 'tokens');
        n_row = str2double(shape{1});
        if n_row > 10
            n_row = 1;
        end
        v = cell2mat(cell(py.list(py.numpy.fromstring(data))));
        v = reshape(v, length(v)/n_row, n_row).';

        if strcmp(sub_topic, 'filt') || strcmp(sub_topic, 'avg')
            v = v2db(v);
        end

        if not_init
            not_init = false;
            [x, wf, pl, img] = init_figures(v);
            n_frame = size(wf, 1);
        end

        % update waterfall
        i_frame = i_frame + n_row;
        if i_frame <=  n_frame
            idx = i_frame;
        else
            idx = n_frame;
            wf = circshift(wf, [-n_row, 0]);
        end
        wf((0:n_row - 1) + idx, 1:length(v)) = v;
        x(1:n_row, 1:length(v)) = v;
```

```
        %update plots
        %caxis([6, 30] + median(wf(:)));
        img.CData = wf;
        if length(pl(1).YData) < length(x)
            [x, wf, pl, img] = init_figures(v);
        end
        for i = 1:n_row
            try
                pl(i).YData = x(i, :);
            catch
            end
        end
        drawnow;
    end
end

function val = get_arg(args, key, default)
    if isKey(args, key)
        val = args(key);
    else
        val = default;
    end
end

function dev = init_data_device(socket_ip, sub_topic)
    tcp = sprintf('tcp://%s:5556', socket_ip);
    ctx = py.zmq.Context();
    dev = ctx.socket(py.zmq.SUB);
    dev.connect(tcp);
    dev.setsockopt(py.zmq.SUBSCRIBE, py.str(sub_topic));
end

function [x, wf, pl, img] = init_figures(v)
    n_frame = 500;

    % init figures
    f1 = figure(1);
    f1.WindowStyle = 'docked';
    t = (0:length(v) - 1)/48000;
    x = zeros(length(t), size(v, 1))';
    pl = plot(t*1000, v, '.-');
    grid on;
    xlabel('time [ms]')

    f2 = figure(2);
    f2.WindowStyle = 'docked';
    wf = zeros(n_frame, length(v));

    wf((0:size(v, 1) - 1) + 1, :) = v;
    %img = imagr(wf, 'fignum', 5);
    img = imagesc(wf);
    colorbar; colormap hot
    xlabel('sample')
```

```
    ylabel('time')
    %caxis([-20, 20])
    ui_init();
end

function cb_clim(src, evnt)
    obj = src.Parent.Children(end - 1);
    cur_value = obj.Limits;
    dval = 1;
    if strcmp(src.Tag, 'nf_up')
        new_val = cur_value + dval;
    elseif strcmp(src.Tag, 'nf_dn')
        new_val = cur_value - dval;
    elseif strcmp(src.Tag, 'cal')
        new_val = [3, 30] + median(src.Parent.Children(end).Children(1).CData(:));
    elseif strcmp(src.Tag, 'dr_up')
        new_val(1) = cur_value(1) - dval;
        new_val(2) = cur_value(2) + dval;
    elseif strcmp(src.Tag, 'dr_dn')
        new_val(1) = cur_value(1) + dval;
        new_val(2) = cur_value(2) - dval;
    elseif strcmp(src.Tag, 'dr')
        new_val = cur_value*src.Value;
    elseif strcmp(src.Tag, 'nf')
        new_val = cur_value + src.Value;
    end
    obj.Limits = new_val;
    caxis(new_val);
end

function ui_init()
    cb = colorbar();

    x0 = cb.Position(1);
    dx = cb.Position(3);
    y0 = cb.Position(2);
    dy = cb.Position(4);

    uicontrol('Style', 'text', 'string', 'NF', ...
        'units', 'normalized',...
        'position', [x0 + 1.4*dx, y0, dx, dx], ...
        'Fontweight', 'bold', 'fontsize', 14);

    uicontrol('Style', 'pushbutton', 'string', 'cal',...
        'units', 'normalized',...
        'position', [x0 + 1.2*dx, y0 + 5*dx, dx, dx], ...
        'callback', @cb_clim, 'tag', 'cal');

    uicontrol('Style', 'pushbutton', 'string', '+', ...
        'units', 'normalized',...
        'position', [x0 + 1.2*dx, y0 + dx, dx, dx], ...
        'callback', @cb_clim, 'tag', 'nf_up');

    uicontrol('Style', 'pushbutton', 'string', '-', ...
```

```
        'units', 'normalized',...
        'position', [x0 + 2.4*dx, y0 + dx, dx, dx], ...
        'callback', @cb_clim, 'tag', 'nf_dn');

    uicontrol('Style', 'text', 'string', 'DR', ...
        'units', 'normalized',...
        'position', [x0 + 1.4*dx, y0 + dy - dx, dx, dx], ...
        'Fontweight', 'bold', 'fontsize', 14);

    uicontrol('Style', 'pushbutton', 'string', '+', ...
        'units', 'normalized',...
        'position', [x0 + 1.2*dx, y0 + dy - 2*dx, dx, dx], ...
        'callback', @cb_clim, 'tag', 'dr_up');

    uicontrol('Style', 'pushbutton', 'string', '-',...
        'units', 'normalized',...
        'position', [x0 + 2.4*dx, y0 + dy - 2*dx, dx, dx], ...
        'callback', @cb_clim, 'tag', 'dr_dn');
end
```

Matlab Algorithm Protyping

```
% filename: batch_fmcw_detection.m
% author: Paul Adams
%
function x = batch_fmcw_detection(varargin)
    % init and params
    dbv = @(x) 20*log10(abs(x));
    fname = dir('*radar_data*');
    [~, idx] = max([fname.datenum]);
    fname = fname(idx).name;

    % parse input args
    args = containers.Map(varargin(1:2:end), varargin(2:2:end));
    oversample = get_arg(args, 'oversample', 5);
    window_func = get_arg(args, 'window_func', @rectwin);
    fname = get_arg(args, 'fname', fname);
    detrend_flag = get_arg(args, 'detrend_flag', false);

    sync_chan = 1;
    signal_chan = 2;

    % fetch data
    [x, fs] = audioread(fname);

    % sync pulses
    [pulse_idx, n_samp_pulse] = pulse_sync(x, sync_chan);

    % derived params
    n_pulse = length(pulse_idx) - 1;
    N = size(x, 1);
    t = (0:N - 1)/fs;
    n_fft = 2^nextpow2(n_samp_pulse)*8;
    f = (0:n_fft/2-1)*fs/n_fft;
    taper = repmat(window(window_func, n_samp_pulse)', n_pulse, 1);
    pulse_proto = 1 - abs(linspace(-1, 1, n_samp_pulse));

    %radar parameters
    c = 3E8; %(m/s) speed of light
    fstart = 2260E6; %(Hz) LFM start frequency for example
    fstop = 2590E6; %(Hz) LFM stop frequency for example
    BW = fstop-fstart; %(Hz) transmti bandwidth

    %range resolution
    rr = c/(2*BW);
    max_range = rr*n_samp_pulse/2;
    r = linspace(0, max_range, n_fft);

    % form pulse matrix (optionally detrend)
    y = zeros(n_pulse, n_samp_pulse);
    for i = 1:n_pulse
        pulse = x((1:n_samp_pulse) + pulse_idx(i), signal_chan)';
        y(i, :) = pulse - detrend_flag*mean(pulse);
    end
```

```
    % clutter suppression
    dy = [zeros(1, n_samp_pulse); y(1:n_pulse - 1, :)];
    dyy = [zeros(2, n_samp_pulse); y(1:n_pulse - 2, :)];

    z2 = y - dy; % 2-pulse canceller
    z3 = y - 2*dy + dyy; % 3-pulse canceller

    if 1
        % Taper, FFT, Power
        Z1 = fft(y.*taper, n_fft, 2);
        Z1= dbv(Z1(:, 1:n_fft/2));

        im = imagr(Z1, 'x', r, 'y', t, 'fignum', 101, 'colormap', 'hot');
        xlabel('range bin');
        ylabel('time');
        %%xlim([0, 50]);
        title('Range Time Indicator - No clutter suppresion');

        % Taper, FFT, Power
        Z2 = fft(z2.*taper, n_fft, 2);
        Z2= dbv(Z2(:, 1:n_fft/2));

        im = imagr(Z2, 'x', r, 'y', t, 'fignum', 102, 'colormap', 'hot');
        xlabel('range bin');
        ylabel('time');
        %xlim([0, 50]);
        title('Range Time Indicator - 2-Pulse Canceller');
    end

    % Taper, FFT, Power
    Z3 = fft(z3.*taper, n_fft, 2);
    Z3= dbv(Z3(:, 1:n_fft/2));
    %Z3 = dbv(Z3);

    im = imagr(Z3, 'x', r, 'y', t, 'fignum', 103, 'colormap', 'hot');
    xlabel('range bin');
    ylabel('time');
    %xlim([0, 50]);
    title('Range Time Indicator - 3-Pulse Canceller');

    [~, range_gates] = max(Z3, [], 2);
    figure(3)
    plot(r(range_gates), t(pulse_idx(1:end-1)), 'o')
    xlabel('range [m]'); ylabel('time')
    %xlim([0, 50]);
    grid on;
    ax = gca;
    ax.YAxis.Direction = 'reverse';
    title(fname);
end

function [pulse_idx, n_samp_pulse] = pulse_sync(x, sync_chan)
    % perform pulse sync
```

```
    a = x(:, sync_chan) > 0; % square wave
    b = diff([0; a]) > 0.5; % true on rising edges
    c = diff([0; a]) < -0.5; % true on falling edges

    rise_idx = find(b);
    fall_idx = find(c);
    n = min(length(rise_idx), length(fall_idx));
    ramp_idx = fall_idx(1:n) - rise_idx(1:n);

    % guaranteed shortest ramp time so we don't accidentally integrate over boundaries
    n_samp_pulse = min(ramp_idx);
    pulse_idx = rise_idx;
end

function val = get_arg(args, key, default)
    if isKey(args, key)
        val = args(key);
    else
        val = default;
    end
end
```

```
% filename: DopplerConfig.m
% author: Paul Adams

classdef DopplerConfig < handle
    properties
        ax
        state
        wav
        is_file
        n_total
        n_samp
        Tp
        oversample
        n_dwell_detect
        taper
        i_samp
        i_chan
        i_dwell
        n_filt
        v_mph
        funcs
        max_filt
        debug_level
        noise_est
        alpha_n
        beta_n
    end

    properties(Constant)
        MPH_CF = 2.23694;
        FC = 2590E6; %(Hz) Center frequency (connected VCO Vtune to +5 for example)
        C_LIGHT = 3e8; %(m/s) speed of light
```

```
        AUDIO_DEVICE_ID = 2;
        FS = 44100;
        % Thresholds
        pass_thresh = 0.5; % instantaneous noise level indicating vehicle crossing
        actv_thresh = 28; % peak must be X dB above noise est
        v_max_mph = 4000;% suppress speeds above 50 mph
    end

    methods
        function me = DopplerConfig(args)
            taper_func = eval(sprintf('@%swin', args('windowing_function')));
            me.debug_level = args('debug_level');

            me.Tp = args('dwell_period_ms')/1000;
            me.oversample = args('oversample_factor');
            me.n_dwell_detect = args('n_detection_dwell');
            me.i_samp = 0;
            me.i_dwell = 0;
            me.wav = args('wav_file');
            me.i_chan = args('channel_vector');
            if ~isempty(me.wav)
                I = audioinfo(me.wav);
                me.n_total = I.TotalSamples;
                me.FS = I.SampleRate;
                me.is_file = true;
            else
                me.n_total = inf;
                me.is_file = false;
                me.wav = audiorecorder(me.FS, 16, (me.i_chan), me.AUDIO_DEVICE_ID);
                me.wav.record(2*me.Tp);
            end
            me.noise_est = [];
            me.n_samp = round(me.Tp*me.FS);
            me.Tp = me.n_samp/me.FS;
            me.taper = window(taper_func, me.n_samp).';
            me.n_filt = me.oversample*2^nextpow2(me.n_samp);

            % State
            me.state.real_time = args('real_time');
            me.state.Active = false; % indicates there is moving object in frame,
            % assuming peak value is x dB above noise
            me.state.Passing = false; % indicates an object is passing the radar,
            % assumes instantaneous noise level is x dB above historical value
            me.state.VehicleCount = 0;

            % calculate velocity
            lambda = me.C_LIGHT/me.FC;
            df = me.FS/me.n_filt;
            doppler = 0:df:me.FS/2 - df; % doppler spectrum
            me.funcs.dop2mph = @(dop) me.MPH_CF*dop*lambda/2;
            me.funcs.mph2dop = @(mph) 2*mph/(me.MPH_CF*lambda);
            me.v_mph = me.MPH_CF*doppler*lambda/2;
            max_doppler = 2*me.v_max_mph/(me.MPH_CF*lambda);
            [~, me.max_filt] = min(abs(doppler - max_doppler));
```

```
            % iir filter coeffs
            % noise filter
            me.alpha_n = 0.90; % historical value weighting
            me.beta_n = 1 - me.alpha_n; % current measurement weighting

            plot(me.v_mph, me.v_mph, '.-'); hold on;
            plot(me.v_mph(1), 1, 'r*'); hold off;
            xlim([0, 200])
            ylim([-50, 50])
            xlabel('velocity [mph]'); ylabel('dB'); title('Audio Spectrum');
            me.ax = gca;
            grid on;
        end
    end
end
```

```
% filename: event_doppler.m
% author: Paul Adams
%
function dc = event_doppler(varargin)
    % config
    args = containers.Map(varargin(1:2:end), varargin(2:2:end));
    dc = DopplerConfig(args);
    dc.wav.record(dc.n_dwell_detect*dc.Tp);

    % Loop until EOF
    eof = false;
    while ~eof
        tic;
        x = fetch_and_format(dc);
        [v, i, n0] = reduce_doppler(x, dc);
        update_state(v, i, n0, dc);

        % Check for EOF and enforce real-time
        eof = dc.i_samp >= dc.n_total - dc.n_dwell_detect*dc.n_samp; % within one dwell of eof

        show_state(dc, v, n0);

        margin = dc.Tp*dc.n_dwell_detect - toc;
        if args('real_time')
            pause(margin);
        else
            drawnow
        end
    end
end

function x = fetch_and_format(dc)
    if dc.is_file
        start_samp = dc.i_samp + 1;
        dc.i_samp = start_samp + dc.n_samp*dc.n_dwell_detect - 1;
        if dc.debug_level >= 2; fprintf('Fetch and Format... samples: %d to %d...', start_samp, dc.i_sa
        [v, ~] = audioread(dc.wav, [start_samp, dc.i_samp]);
```

```
    else
        while dc.wav.get('TotalSamples') < dc.n_samp*dc.n_dwell_detect; end
        v = dc.wav.getaudiodata();
        dc.wav.record(dc.n_dwell_detect*dc.Tp);
    end
    x = reshape(v(:, dc.i_chan)', dc.n_samp, dc.n_dwell_detect).'; % let each row have n_samp samples
end

function [v, i, n0] = reduce_doppler(x, dc)
    if dc.debug_level >= 2; fprintf('Doppler Process... \n'); end
    x_w = bsxfun(@times, x, dc.taper); % apply taper to each row
    X = fft(x_w, dc.n_filt, 2); % fft along rows
    %X = 20*log10(abs(X(:, 1:dc.max_filt)));
    X = 20*log10(abs(X(:, 1:dc.n_filt/2)));
    n0 = mean(X(:));
    [v, i] = max(mean(X, 1)); % take mean over rows, then select the max filter bin
    dc.ax.Children(2).YData = mean(X, 1);
    dc.ax.Children(1).XData = dc.v_mph(i);
    dc.ax.Children(1).YData = v;
end

function update_state(v, i, n0, dc)
    if isempty(dc.noise_est)
        dc.noise_est = n0; % initial value
    else
        % current noise est. is weight last est. + weighted current measurement
        dc.noise_est = dc.alpha_n*dc.noise_est + dc.beta_n*n0;
    end

    dc.state.SpeedEst = nan;
    if n0 > dc.noise_est + dc.pass_thresh
        if ~dc.state.Passing % if changing state, inc. counter
            dc.state.VehicleCount = dc.state.VehicleCount + 1;
        end
        dc.state.Passing = true;
    elseif v > dc.noise_est + dc.actv_thresh
        dc.state.Passing = false; % turn off Passing
        dc.state.Active = true;
        dc.state.SpeedEst = dc.v_mph(i);
    end
    if v <= dc.noise_est + dc.actv_thresh; dc.state.Active = false; end

    % update debug variables
    if dc.debug_level >= 1
        dc.i_dwell = dc.i_dwell + 1;
        dc.state.n_i(dc.i_dwell) = n0;
        dc.state.n_a(dc.i_dwell) = dc.noise_est;
        dc.state.v_mph(dc.i_dwell) = dc.state.SpeedEst;
    end
end

function show_state(dc, v, n0)
    fprintf('Time: %.2f\tnoise: %.2f\tpower: %.2f\tPeak: %.2f\tPassing: %d\tActive: %d\tSpeed: %0.1f\tVe
        dc.i_samp/dc.FS, dc.noise_est, n0, v, dc.state.Passing, dc.state.Active, round(dc.state.SpeedEst
```

```
end
```

Python Main Programs

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
# __file__ sdr_main.py
# __author__ Paul Adams

import sys

MODE = sys.argv[1]

def main():
    if MODE == 'fmcw':
        import fmcw_serv
        fmcw_serv.main()
    elif MODE == 'dopp':
        import dopp_serv
        dopp_serv.main()

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
# __file__ sdr.py
# __author__ Paul Adams

# third-party imports
import numpy as np
import socket
from scipy.signal import lfilter
import zmq
import time

SYNC_CHAN = 0
SGNL_CHAN = 1
N_CHAN = 2
SUB_PORT = 5555
PUB_PORT = 5556
lpf_b = np.fromfile('/home/paul/ee542/lpf.npy')

class Zmq():
    def __init__(self):
        # setup zmq
        self.ctx = zmq.Context()
        self.sub = self.ctx.socket(zmq.SUB)
        self.sub.setsockopt(zmq.SUBSCRIBE, 'pcm_raw')
        self.ip = socket.gethostbyname('thebes')
        tcp = "tcp://%s:%s" % (self.ip, SUB_PORT)
        self.sub.connect(tcp)
        print 'SDR: Listening on %s' % tcp

        self.pub = self.ctx.socket(zmq.PUB)
        tcp = "tcp://%s:%s" % (self.ip, PUB_PORT)
```

```
        self.pub.bind(tcp)
        print 'SDR: Publishing on %s' % tcp

class Queue():
    def __init__(self):
        self.buff_idx = 0
        self.ref = []
        self.sig = []
        self.raw = []
        self.n_buff =  1
        self.time = []

    def re_init(self):
        self.buff_idx = 0
        self.ref = []
        self.sig = []

    def fetch_format(self):
        # fetch format data
        data = z.sub.recv()
        idx = data[0:100].find(';;;')
        header = data[0:idx]
        self.time = float(header[header.find(':')+1:header.find(';;;')])
        #import pdb; pdb.set_trace()
        x = (np.fromstring(data[idx+3::], np.int32)).astype(np.float)/2**31
        self.sig = x[SGNL_CHAN::2]
        self.ref = x[SYNC_CHAN::2]
        debug_hook(self.ref, 'clock')
        debug_hook(self.sig, 'signal')

    def update_buff(self):
        self.buff_idx += 1
        self.fetch_format()

    def is_full(self):
        return self.buff_idx == self.n_buff

def debug_hook(data, topic):
    n_row = data.shape[0]
    z.pub.send('%s n_row: %s;;;%s' % (topic, str(n_row), data.tostring()))

def lowpass(x):
    if len(x.shape) == 1:
        axis = 0
    else:
        axis = 1

    return lfilter(lpf_b, 1, x, axis=axis)

def fft_filter(x, n_fft):
    if len(x.shape) == 1:
        x = np.abs(np.fft.fft(x, n=n_fft)[0:n_fft/2])**2
    else:
        x = np.abs(np.fft.fft(x, n=n_fft)[:, 0:n_fft/2])**2
```

```
    debug_hook(x, 'filt')
    return x

def averager(x):
    x = np.mean(x, axis=0)
    debug_hook(x, 'avg')
    return x

z = Zmq()
```

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
# __file__ serv-alsa.py
# __author__ Paul Adams

# third-party imports
import socket
import pyaudio
import zmq
import time
import sys

# local imports
N_SAMP = int(sys.argv[1])
N_SAMP_BUFF = 10*N_SAMP
N_CHAN = 2
FS = 48000
PUB_PORT = 5555
pa = pyaudio.PyAudio()

# setup zmq
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
tcp = "tcp://%s:%s" % (socket.gethostbyname('thebes'), PUB_PORT)
pub.bind(tcp)
print 'ALSA: Publishing on %s' % tcp

def alsa_callback(data, frames, time, status):
    pub.send('%s %s;;;%s' % ('pcm_raw', 'time:%f' % (time['current_time']), data))
    return (data, pyaudio.paContinue)

class Alsa():
    def __init__(self):
        self.stream = pa.open(format=pyaudio.paInt32,
                rate=FS, input=True, channels=N_CHAN,
                frames_per_buffer=N_SAMP_BUFF,
                stream_callback=alsa_callback)

    def loop(self):
        while self.stream.is_active():
            time.sleep(0.01)

        # stop stream
```

```
        self.stream.stop_stream()
        self.stream.close()
        pa.terminate()

def main():
    a = Alsa()
    a.loop()

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
# __file__ serv-fmcw.py
# __author__ Paul Adams

# third-party imports
import numpy as np
import time
import json

# local imports
import sdr

# constants
M2FT = 3.28084
C_LIGHT = 3e8
BW = 300e6
N_FFT = 4096
FS = 48000
MAX_PERIOD_DELTA = 50


class Sync():
    def __init__(self):
        self.have_period = False
        self.period = []
        self.edges = {}
        self.T = []
        self.pulses ={}
        self.tail = {}
        self.tail['ref'] = np.zeros(2)
        self.head = {}
        self.stable_period_count = 0

    def get_edges(self, q):
        dref = np.diff((q.ref > 0).astype(np.float))
        # find indices of rising edges
        self.edges['rise'] = np.where(dref == -1)[0]

        # find indices of falling edges
        self.edges['fall'] = np.where(dref == +1)[0]

    def align_edges(self, q):
```

```
        # make sure fall follows rise, save head
        head_idx = np.argmax(self.edges['fall'] > self.edges['rise'][0])
        self.edges['fall'] = self.edges['fall'][head_idx::]
        head_idx = self.edges['rise'][0]

        # make sure each vector is equi-length
        if len(self.edges['rise']) > len(self.edges['fall']):
            self.edges['rise'] = self.edges['rise'][0:len(self.edges['fall'])]
        else:
            self.edges['fall'] = self.edges['fall'][0:len(self.edges['rise'])]

        # try stitch previous tail to current head
        self.head['ref'] = q.ref[0:head_idx]
        self.head['sig'] = q.sig[0:head_idx]
        self.stitch()
        self.tail['ref'] = q.ref[self.edges['fall'][-1]::]
        self.tail['sig'] = q.sig[self.edges['fall'][-1]::]

    def check_period(self):
        if self.period:
            prev_period = self.period
        else:
            prev_period = 0

        self.period = np.floor(np.mean(self.edges['fall'] - self.edges['rise'])).astype(np.int16)
        rez = np.abs(self.period - prev_period)

        if rez < MAX_PERIOD_DELTA:
            self.stable_period_count += 1
            if not self.have_period:
                print 'pulse period acquired --> %d samples' % (self.period)
                self.have_period = True
                self.T = self.period*FS

        elif self.have_period:
            self.stable_period_count = 0
            self.have_period = False
            self.period = []
            print 'pulse period lost. residual --> %d samples' % (rez)

    def stitch(self):
        if self.tail['ref'].any():
            x = np.hstack((self.tail['ref'], self.head['ref']))
            y = np.hstack((self.tail['sig'], self.head['sig']))

            # sync clock signal
            dx = np.diff((x > 0).astype(np.float))

            # find indices of rising edges
            rise = np.where(dx == -1)[0].tolist()

            while rise:
                r = rise.pop(0)
                if self.period.__class__ is list:
```

```
                    pass
                else:
                    # import pdb; pdb.set_trace()
                    ts = float(r)/self.period*(q.time - 1)
                    self.pulses[ts] = y[r:r+self.period]

    # given a buffer of audio frames, find the pulses within the clock signal and extract received chirp
    def extract_pulses(self, sig):
        rises = self.edges['rise'].tolist()
        while rises:
            idx = rises.pop(0)
            ts = float(idx)/self.period*q.time
            # import pdb; pdb.set_trace()
            self.pulses[ts] = sig[idx:idx + self.period]

class Processor():
    def __init__(self):
        self.x = 0
        self.detects = []
        self.range_rez = C_LIGHT/(2*BW)
        self.n_samp = 0
        self.report_dict = {}
        self.range_lu = np.zeros((1, 1))
        self.ranges = []
        self.prior = []
        self.string = ''
        self.t0 = 0

    def format(self, pulses):
        self.n_samp = min([len(p) for p in pulses.values()])
        self.x = np.zeros((len(pulses) , self.n_samp))
        k = pulses.keys()
        k.sort()
        for i, j in enumerate(k):
            self.x[i, :] = pulses[j][0:self.n_samp]

        sdr.debug_hook(self.x, 'raw')

    def canceller(self):
        (nrow, nsamp) = self.x.shape
        try:
            if len(self.prior) > 0:
                if len(self.prior) != nsamp:
                    nsamp = min([len(self.prior), self.x.shape[1]])
                    self.x = self.x[:, 0:nsamp]
                    self.prior = self.prior[0:nsamp]
            else:
                self.prior = np.zeros(nsamp)
        except:
            import pdb; pdb.set_trace()

        y = self.x.copy()
        dy = np.vstack((self.prior, y[0:nrow-1, :]))
        self.x = y - dy
```

```
        self.prior = y[nrow-1, :]

    def detect(self):
        #cfar = signal.lfilter(self.cfar_filt, 1, self.x)
        self.detects = [np.argmax(self.x[50:-1]) + 50]

    def transform(self):
        self.ranges = []
        if not self.range_lu.any():
            max_range = self.range_rez*self.n_samp/2
            self.range_lu = np.linspace(0, max_range, N_FFT/2)

        if len(self.detects) > 0:
            for d in self.detects:
                self.ranges.append(self.range_lu[d])

    def report(self):
        self.report_dict = {}
        for i in range(len(self.detects)):
            ts = time.time()
            self.report_dict[ts] = {}
            self.report_dict[ts]['gate'] = self.detects[i]
            self.report_dict[ts]['range'] = self.ranges[i]

        self.report_str = json.dumps(self.report_dict)

    def print_debug(self, doit):
        dt = q.time - self.t0
        self.t0 = q.time
        self.string += 'time: %.3f -> dt: %.3f ms -> pulses: %d -> samp: %d stable -> %d -> detect: %d r
        if doit:
            print self.string,
            self.string = ''

    def process_pulses(self, pulses):
        self.format(pulses)
        self.x = sdr.lowpass(self.x)
        self.canceller()
        self.x = sdr.fft_filter(self.x, N_FFT)
        self.x = sdr.averager(self.x)
        self.detect()
        self.transform()
        self.report()

# init objects
q = sdr.Queue()
s = Sync()
p = Processor()

def main():
    print 'RUN_FMCW: Queue and Sync initzd... Entering loop now... '

    while True:
        q.update_buff()
```

```
        if q.is_full():
            s.get_edges(q)
            s.align_edges(q)
            s.check_period()

            if s.have_period:
                s.extract_pulses(q.sig)
                p.process_pulses(s.pulses)
                p.print_debug(True)
                s.pulses = {} # reset pulses
            else:
                pass

            q.re_init()

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
# __file__ dopp-serv.py
# __author__ Paul Adams

# third-party imports
import numpy as np
from scipy.signal import hanning
import time
import json

# local imports
import sdr

# constants
M2FT = 3.28084
C_LIGHT = 3e8
BW = 300e6
N_FFT = 4096*4
FS = 48000

class Processor():
    def __init__(self):
        self.x = 0
        self.detects = []
        self.n_samp = 0
        self.report_dict = {}
        self.string = ''
        self.taper = np.zeros(1)
        self.t0 = 0

    def format(self):
        if not self.taper.any():
            self.taper = hanning(q.sig.shape[0])
        self.x = np.multiply(q.sig, self.taper)
```

```
        sdr.debug_hook(self.x, 'raw')

    def detect(self):
        self.detects = [np.argmax(self.x[50:-1]) + 50]

    def print_debug(self, doit):
        dt = q.time - self.t0
        self.t0 = q.time
        self.string += 'time: %.3f -> dt: %.3f ms -> detect: %d m\n' % (self.t0, dt*1e3, self.detects[0]
        if doit:
            print self.string,
            self.string = ''

    def process_pulses(self):
        self.format()
        self.x = sdr.lowpass(self.x)
        self.x = sdr.fft_filter(self.x, N_FFT)
        self.x -= np.mean(self.x)
        self.detect()

# init objects
q = sdr.Queue()
p = Processor()

def main():
    print 'RUN_DOPP: Queue initzd... Entering loop now... '

    while True:
        q.update_buff()

        if q.is_full():
            p.process_pulses()
            p.print_debug(True)
            q.re_init()

if __name__ == '__main__':
    main()
```

Shell Scripts and systemd Modules

```
#!/bin/bash
# file: eth0-startup.sh
# author: Paul Adams

carrier_state=$(</sys/class/net/eth0/carrier)
not_up=1

while [ "$not_up" -eq 1 ]; do
    if [ "$carrier_state" -eq 1 ]; then
        ip addr add 192.168.2.108/255.255.255.0 dev eth0
        ip link set eth0 up
        not_up=0
        echo "success"
    else
        echo "ethernet not connected foo"
        sleep 10
    fi
done

echo "exit program"
```

```
#!/bin/bash
# file: kill-all.sh
# author: Paul Adams

pgrep python2 | xargs kill -9
pgrep jackd | xargs kill -9
```

```
#!/bin/bash
# file: start-all.sh
# author: Paul Adams

/usr/bin/jackd -P70 -p16 -t2000 -d alsa -d hw:0 -p 128 -n 3 -r 48000 -s 2>/dev/null&
echo 'started jackd...'
sleep 1
/usr/bin/python2 /home/paul/ee542/alsa_serv.py $1 2>/dev/null&
# /usr/bin/python2 /home/paul/ee542/alsa_serv.py $1 &
echo "started serv-alsa.py N_SAMP=$1"
sleep 1
/usr/bin/python2 /home/paul/ee542/sdr_main.py $2
```

```
[Unit]
Description=my jackd management wrapper
Before=serv-alsa.service serv-fmcw.service

[Service]
ExecStart=/usr/bin/jackd -P70 -p16 -t2000 -d alsa -d hw:0 -p 128 -n 3 -r 48000 -s
Type=simple
Restart=always
RestartSec=15s

[Install]
WantedBy=multi-user.target
```

```
[Unit]
Description=my fmcw server wrapper
After=serv-alsa.service jackd.service
Requires=jackd.service serv-alsa.service

[Service]
Environment=PYTHONBUFFERED=true
ExecStart=/usr/bin/python2 /home/paul/ee542/serv-fmcw.py
Type=simple
Restart=always
RestartSec=15s

[Install]
WantedBy=multi-user.target
```

```
[Unit]
Description=my alsa server wrapper
After=jackd.service
Before=serv-fmcw.service
Requires=jackd.service

[Service]
Environment=PYTHONBUFFERED=true
ExecStart=/usr/bin/python2 /home/paul/ee542/serv-alsa.py
Type=simple
Restart=always
RestartSec=15s

[Install]
WantedBy=multi-user.target
```