Corey Hobbs – UC Davis
EEC134AB – RF/Microwave System Design
Professor Xiaoguang Liu – April 2015
Beagle Bone Black “All in One System”

1. Introduction

"First, a bit of vocab: the Beaglebone Black is a single-board computer, like the Raspberry Pi. A single-board computer is pretty much what it sounds like—all the hardware you would expect to find in a desktop or laptop computer, shrunken down and soldered to a single circuit board. A processor, memory, and graphics acceleration are all present as chips on the board. To contrast, Arduino boards also have a processor and memory on board, but are orders of magnitude less powerful, and lack the specialized I/O hardware you need to connect the board to a monitor. In more concrete terms, you can hook a Beaglebone Black up to a display, speakers, a keyboard and mouse and an Ethernet network, and boot into a Linux-based operating system. From there, you can do anything you could do with a (low-powered) Linux computer. You can’t do that with Arduino." - ALEX CASTLE of TESTED.COM

In this course, I am attempting to implement the BBB as a function generator and digital signal processor hybrid unit. In hopes to create a closed feedback loop with an external RF Antennae system that will attached to the output/input of the Beagle Bone Black's USB audio jacks. Turning it into an all-in-one radar unit.

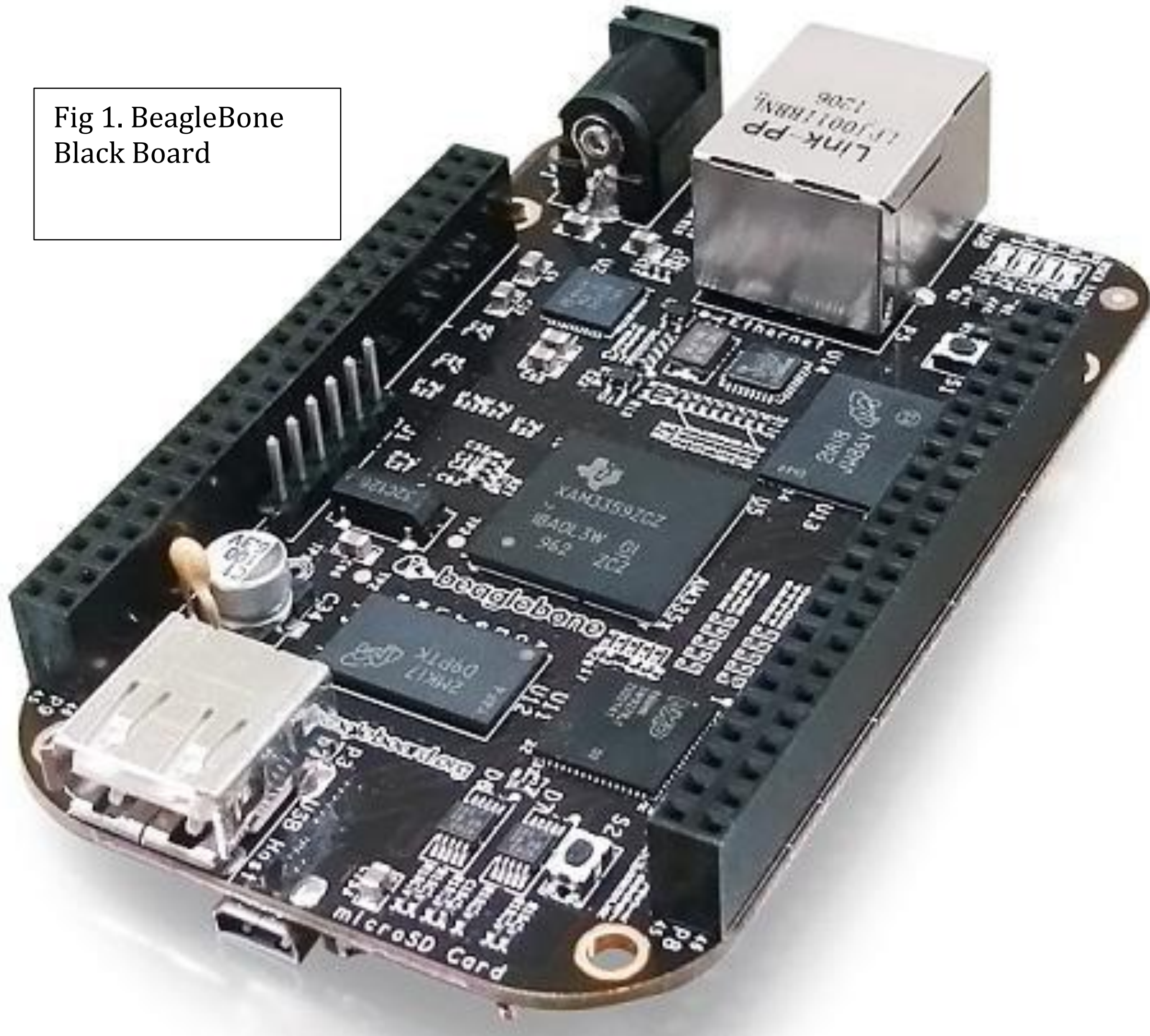

Fig 1. BeagleBone Black Board

2. Setting Up BeagleBone Black (BBB)

Since I started with BBB revision 6a, and I wanted it to run an Ubuntu based linux environment, I had to flash its firmware to change the operating system.

A) Items Needed to Flash BBB:

1. ***5V, 2A BBB Power Supply***
2. ***Micro SD Card***
3. ***Micro HDMI-HDMI CABLE***
4. ***External Monitor***

5. ***USB Keyboard***
6. ***USB Mouse***
7. ***USB Port Extender***

B) Instructions to Flash

(http://elinux.org/Beagleboard:Booting_Ubuntu_on_BeagleBoard_Black) ← reference link

The steps described here are steps you must take before installing any version of Ubuntu described in this Wiki.

You must first decide which version you want to run.

This page has the image files for 3 options for installing Ubuntu/Debian on a micro SD card to boot from Your 3 options are :

-Ubuntu Precise 12.04.2 LTS

-Ubuntu Raring 13.04

-Debian Wheezy 7.0.0

You have two current options to flash the eMMC

Your 2 options are [Ubuntu] [Debian]

After you have downloaded the .img.xz file you want, use a program such as 7zip to extract the image file Insert the micro SD card into your computer. Using a micro SD to USB adapter is fine. Use Win32 Disk Imager to write the image onto your micro SD. Run Win32 Disk Imager.

Make sure the drive letter corresponding to your micro SD card is selected. Select the unpacked image file.

Press 'Write' and wait for Win32 Disk Imager to finish - this should take a few minutes.

After Win32 Disk Imager has finished remove the micro SD card from your computer.

You are now ready to follow directions below that correspond to the version of Ubuntu/Debian you downloaded.

Put the micro SD card into the powered-off BeagleBoard Black. Make sure your BeagleBoard Black is connected to a powered-on display via HDMI.

While holding down the 'boot' button (button closest to the micro SD card slot), apply power to the BeagleBoard Black You must use an external 5V power supply, USB power will not work.

Continue to hold down the 'boot' button until the 4 LED lights begin to flicker. If more than 15 seconds have passed without the lights beginning to flicker, remove the power and try again.

The Linux penguin should flash in the upper left-hand corner of your display. After approximately 2-3 minutes you should be prompted for a username and password

Username is: ubuntu **Password** is: ubuntu

Note: echo is turned off for typing in password

You should now be in the command terminal for Ubuntu Anytime you want to boot from micro SD to Ubuntu.

C) Instructions for Audio Drivers (http://andicelabs.com/2014/03/usb-audio-beaglebone/)

Once on BBB **command line** interface do:

1. sudo apt-get install alsa-base alsa-utils
2. I added the following line to uEnv.txt:

   optargs=capemgr.disable_partno=BB-BONELT-HDMI,BB-BONELT-HDMIN
3. Be sure not to disable "BB-BONE-EMMC-2G" if you're using the on-board eMMC.
4. type: aplay -l

   **** List of PLAYBACK Hardware Devices ****

   card 1: Device [C-Media USB Audio Device], device 0: USB Audio [USB Audio]

   Subdevices: 1/1

   Subdevice #0: subdevice #0
5. sudo vi /etc/modprobe.d/alsa-base.conf
6. # Keep snd-usb-audio from beeing loaded as first soundcard

   options snd-usb-audio index=-2

   Change the index from -2 to 0 and now the USB

   adapter should be allowed to become card 0.

3. Function Generation Algorithm Tests

I first started by searching different open source websites and libraries for Function Generation code. I needed to be assured that the code would be able to output a triangle

wave, sine wave, and square wave all at 1Khz. My group chose the 1Khz set of waves because it worked best with our antennae design. The hardest part about generating a 1Khz wave from the BBB is that you have to be concerned with how much the BBB does not have available.

The BBB had only 256MB of DDR2 and a 700-MHz super-scalar ARM Cortex™-A8 Processor. To draw a comparison, this is not even as powerful as the modern smartphone (iPhone 5 has 1 GB Ram, 1300 MHz ARM – A7). With that said, here are my attempts and slight successes with getting the BBB to cooperate and not stall.

A) Attempt One:

Open Source MIT Code by Original idea copyright, (C)2009, B.Walker, G0LCU.
attached as part of the materials/code section. Also link to github source code for this function is at:
https://github.com/chobberoni/beagle134/blob/master/proj/auto_function_generator.py

Finding this set of open source code was a great start to figuring out the complexities behind signal generation from a ARM based micro-controller. It taught me a lot about signal distortion, signal loss, and throughput (processing). The downside to this set of code is that it is large and extremely complex to decipher and manipulate. As mentioned in the comment section of this function's code, the author mentions that he only allows for the signals to be played at 1Khz to "keep the length of the code down." This was a "Python program to generate a Sine, Square, Triangle, Sawtooth, and Pulse waveforms using STANDARD Python at the earphone sockets." Moreover, he used hex codes to build each signal that was to be generated; however, he did not release any information on how these hex codes were chosen or how they could be manipulated to support different frequencies. Another downside to the is code set was this it was extremely difficult to change it's function so it would only output one type of signal for a definite amount of time. This set of code also required user input on the command line, which may not be optimal for many of our radar system applications. All in all, code worked, outputted every wave that it said it could, but just not in the manner that I needed it to work.

B) Attempt Two:

A Manipulated Version of B. Walker's Code (only Triangle Wave Generation)
link:
https://github.com/chobberoni/beagle134/blob/master/proj/broken_triangle.py and in code section.

After a couple forums searches and lab failures, I was able to figure out what was going wrong the output of the triangle wave output. So I thought… I was successfully able to output a triangle wave that was clean and reliable at 1Khz. The only downside was that the code would not stop running. I attempted to extract only the code blocks that linked to the generation of a triangle wave, but when testing it in the lab the code would run in an infinite while loop.

Here is a photo of a successful triangle wave on the oscilloscope.

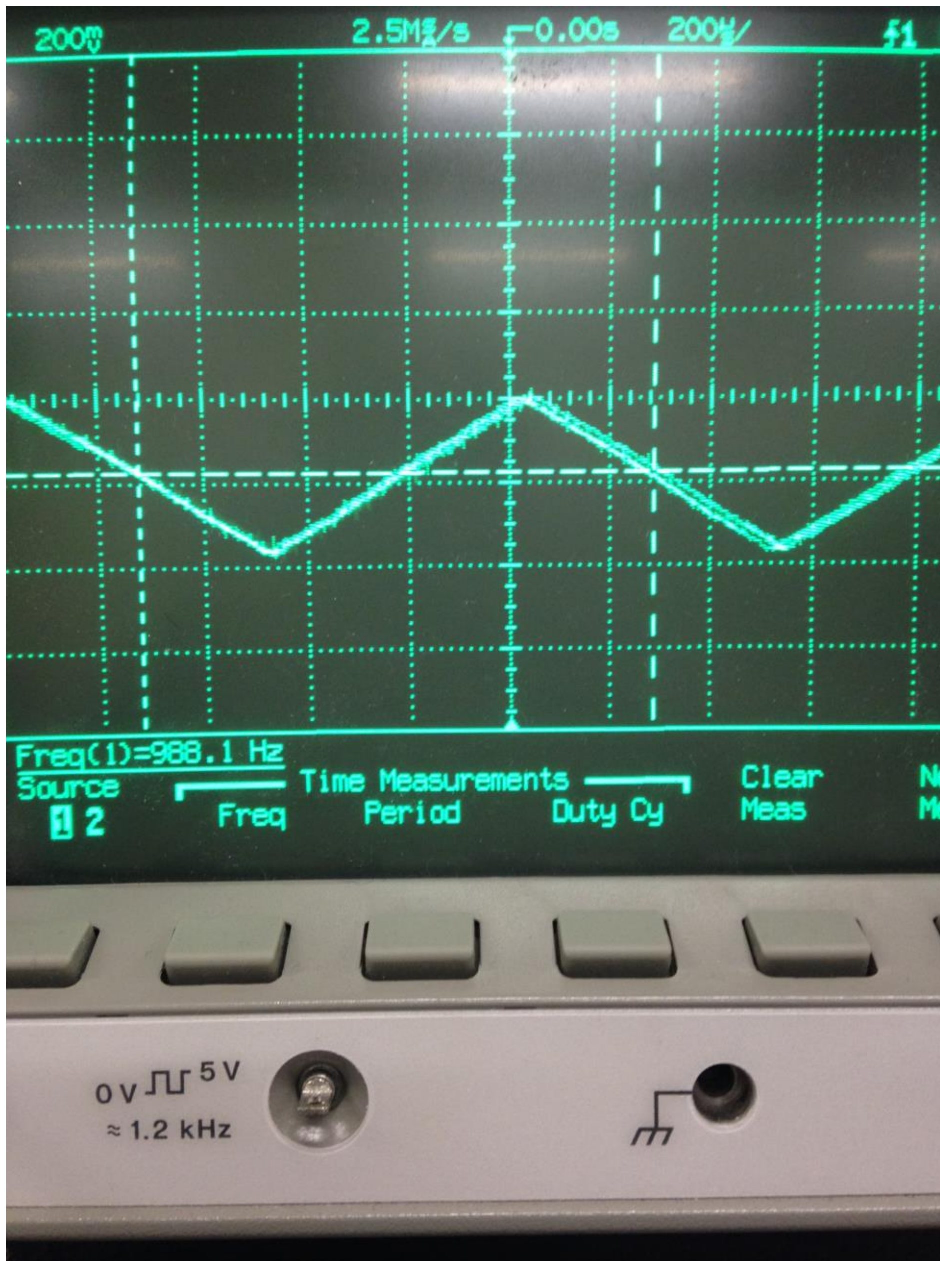

Fig 2. Triangle wave from BBB.

C) Attempt Three (The Success):
Recording 1Khz, 1.5Khz, 2Khz triangle waves and using PyAudio
link at: https://github.com/chobberoni/beagle134/blob/master/proj/triwave.py

Upon realizing that this system may not work by using cookie cutter methods that I had found on forums and other micro-controller websites. I decided to literally build triangle waves at 1Khz, 1.5Khz, and 2Khz using audio software that allowed me to make each signal

up to 10 seconds long. Since I knew we were probably going to pulse the signal anyhow, it didn't really need to be longer than 10 seconds. If need be, just put it into a timed loop. The code consists of simple Python wave parsing. All I had to do was grab the frame rate, sample width, # of channels, and then make sure I parsed each sample bit at 1024 bit chunk size.

Successful triangle (with noise) wave at 1Khz shown below:

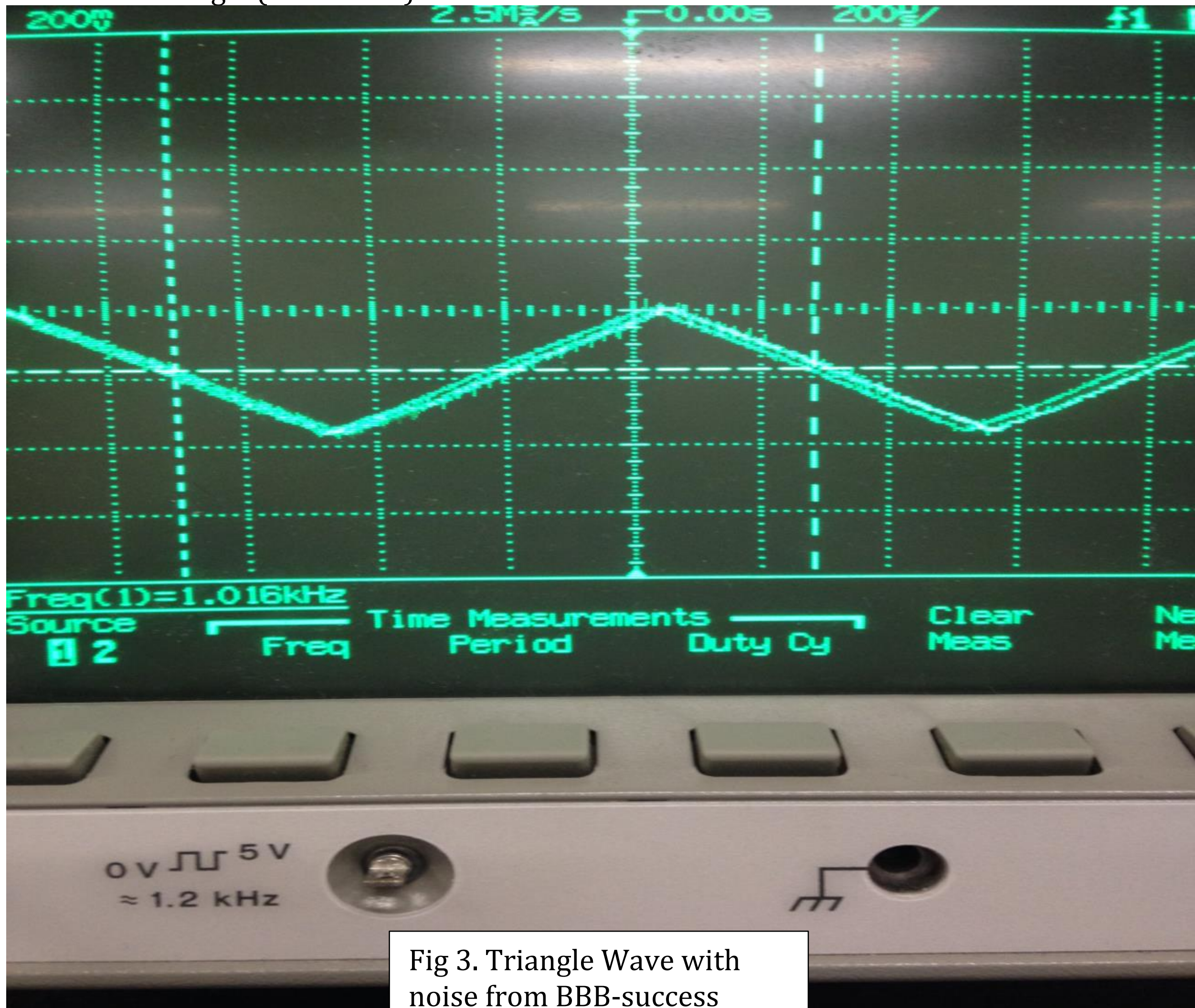

Fig 3. Triangle Wave with noise from BBB-success

It should be mentioned that this new piece of code generates a lossier signal than using the hexadecimal bit-code from B. Walker's solution. This is due to unreliability of file compression and audio compression when downloading the various waves online and then replaying the sound-waves via the BBB, which is not a microcontroller that is optimized for audio manipulation or audio processing.

4. Signal Retrieval (Recording of Returned, phase shifted waves):

A) Attempt One: A Failed Success at Recording Phase Modulated Wave

link at: https://github.com/chobberoni/beagle134/blob/master/proj/twopyrecord.py

from source: http://stackoverflow.com/questions/892199/detect-record-audio-in-python

In the retrieval of the signal section of this project is where a learned a great deal about signal distortion, phase modulation, and loss in signals. Signal distortion came about partially from the cheap USB audio dongle that allowed me to do many test cases of how the BBB would interact with the actual antennae system. I quickly learned that if I did not filter the signal on the return end, it more than likely would come back distorted. In many forums, people stated that without a dedicated sound card or BBB audio cape many audio recordings will be affected by distortion and appear as a lossy signal. In my test I was able to build a function that simultaneously run the function generation code and record the return signal in the same instance. This was done with a bash script that called triwave.py and then on an inset timer called the twopyrecord.py function. It worked and did exactly as expected; however, on observing the returned and recorded signal/wave, it came back as a distorted triangle wave that looked like a slanted square wave function. Also, this new wave was only at about 920 Hz.

Some call this occurrence the windowing effect and it can be combatted with creating an envelope filter for your retrieval/recording function so that many of the unwanted signals and spectrums will be disregarded.

All in all, the dual function worked. It was able to send out the triangle function and in real time record the "receiver's" return signal. But, back to the issue of performance and processing power on the BBB probably limited its ability to do this more effective. Better code could have helped as well.

Here is a link to the output(retrieved) wave:
https://github.com/chobberoni/beagle134/blob/master/proj/outwavetest.wav

5. Digital Signal Processing

The fail or true success of this project was always going to come down to this section. And to say the least, I failed in this area. Focusing too much time on the function generation and retrieval of smooth even waveforms caused the DSP algorithms and research area to fail as a whole.

I started with code packets provided by Professor Liu in lab6:

Focusing in on the the doppler_wav.py function
(https://github.com/chobberoni/beagle134/blob/master/Lab6/doppler_wav.py)
This code actually works and runs on the BBB, but has a runtime of 1 minute, and 48 seconds at best. When run on my i5 2015 Macbook Pro, runtime was about 18 seconds, and the BBB doesn't even have half the processing capability of my Mac.

This is where throughput became an issue for the BBB. It's tiny A8 Arm Cortex processor doesn't have enough power to handle as many jobs as the Doppler_wav.py function was asking of it. The plan was to reduce the number of samples taken for the FFT, fast fourier transform, and then take chances from there and hope for a faster run time. But this also still had a very long run time.

Going to the doppler_realtime_spi_adc.py function from Lab6:

(https://github.com/chobberoni/beagle134/blob/master/Lab6/doppler_realtime_spi_adc.py)
this function is similar to the code above, but it allows us to read and output a graph in real-time. Note, the spidev package is not supported on the BBB, so this is where we would implement our twopyrecord.py function instead and then throw the retrieved signal into the FFT function for plotting.
Here is a photo of the plot from the doppler_wav.py graph:

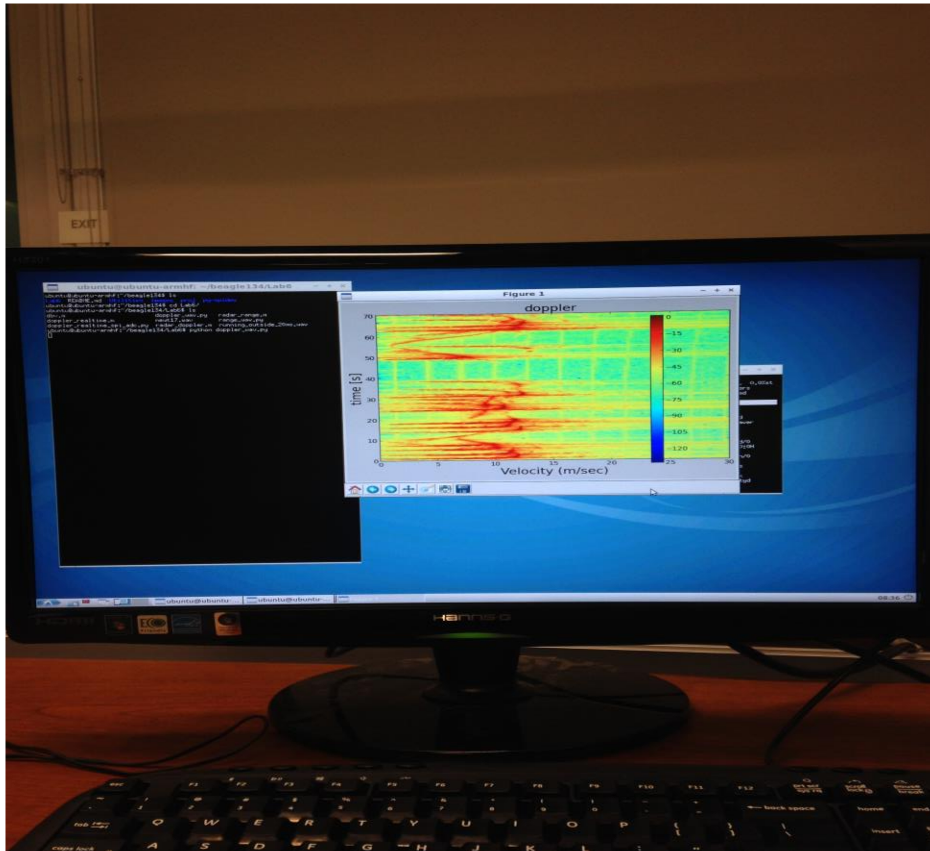

Fig 4. Doppler Wave graph from BBB. Runtime +55seconds.

Overall, the DSP code that was given from lab 6 was not optimal for the BBB. As the spidev library, an optimization library, was not supported it greatly affected the success of the BBB being able to complete such difficult number crunching.

6. How To Succeed with Micro-Controller for this Entire Project

Shortly after deciding to go with BBB the Raspberry Pi 2 came out. The Raspberry Pi 2 easily outperforms the BBB in every aspect possible. The RP2 has 900MHz processor, 1GB Ram, 4 USB outputs, 1 dedicate 3.5mm sound jack, hdmi out, and a dedicated video processing card. This setup alone on the RP2 would alleviate the headache of carrying an entire array of dongles and usb adapters everywhere you want to develop on the BBB.

Here is a photo of what my setup looked like (described in section 2A):

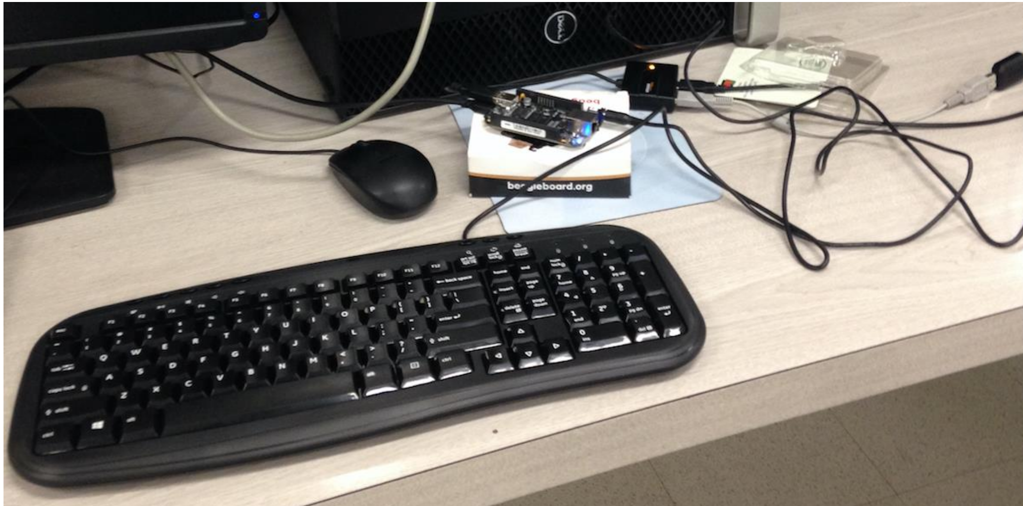

Fig 5. BBB Setup. Keyboard, Mouse, USB extender, Power Supply, microSD card

The RP2's dedicated video card automatically tackles the issue of not having enough throughput and processing power to do the real-time FFT and plotting. Also, the fact that it has an on-board audio jack it handles the issue of signal distortion. The RP2 also runs Windows 10 which would allow you to run a lite version of Matlab to get better results on your doppler and range tests. Not to mention the processor is a quad-core A7, unlike the BBB single core.

Upon having an RP2, there is also no need to boot any additional operating system on it because it comes pre-loaded with Ubuntu on it. Without even changing the code from Lab6 you'd be able to have a working realtime doppler radar system.

Additionally, you could simply use the BBB or another microcontroller and partition the tasks to other smaller, lighter microcontrollers that perform one task. For example using the TeensyDuino or Particle.io Photon to do number crunching while tasking the primary microcontroller to do only DSP. When adding another system in cohort, you always have to be aware of the power consumption vs performance and knowing when it becomes less efficient.

7. Final Thoughts and Conclusions

If you plan to use an RP2 or BBB be aware that the output signal that you are sending to your antennae is only at about 400mv, not 3-5V. I know for our system this was a major issue and I figured that we would use a step up amp and potentiometer to boost our output signal. However, this may cause issues if you don't filter properly going into your antennae network. I highly recommend having two sides to your project: one "dream" case, and the other a more practical tested approach. This all-in-one system was definitely our dream case because it would have allowed us to have a much lower power draw, over the air (wifi-ssh) controls of the system, and a system that could perform DSP outside of the typical external laptop setup that most groups used. If any group plans to take this approach I urge you to use a Raspberry Pi 2 or something better that doesn't require a high level of power consumption. This is easily doable if you are willing to put in the work to make it work. All

of the code is available via github and dropbox (in the materials section of this report).

Thank you Professor Liu,
Corey Hobbs

Acknowledgements:
UC Davis Electrical Engineering Department
Professor Xiaouguang "Leo" Liu
Senior Design Partners: Daniel Hunchard & Francis Ambion
Stackoverflow.com
B. Walker (MIT License)

Materials Section (Code Snippets & Links)

Contact: Email – chobberoni@gmail.com
Linkedin - https://www.linkedin.com/in/coreyhobbsofdavis

Dropbox link to entire project files:

https://www.dropbox.com/sh/3uew4p67k78w5fx/AADw01dsYtKM7D6l5h9BAZjea?dl=0

Github Link to entire Repo:

https://github.com/chobberoni/beagle134

1) auto_function_generator.py

```
# Audio
Function
Generator;
"afg.py"..
.
# ------------------------------------
#
# A fun Python program to generate a Sine, Square, Triangle, Sawtooth,
# and Pulse waveforms using STANDARD Python at the earphone sockets.
# This is for (PC)Linux(OS), (ONLY?), and was done purely for fun.
# (It has now been tested on Debian 6.0.0 and Knoppix 5.1.1!)
#
# Although it is fairly easy to alter the frequency within limits I
# have left it at approximately 1KHz to keep the code size down...
#
# It would be tricky to change the output level using STANDARD Python
# for (PC)Linux(OS) 2009 using this idea to generate waveforms, but it
```

```
# is possible within limits.
#
# (Original idea copyright, (C)2009, B.Walker, G0LCU.)
# Issued as Public Domain, (to LXF) and you may do with it as you please.
#
# It is assumed that /dev/audio exists; if NOT, then install oss-compat
# from the distro`s repository.
#
# Ensure the sound system is not already in use.
#
# Copy the file to the Lib folder(/drawer/directory) or where the modules
# reside as "afg.py"...
#
# For a quick way to run just use at the ">>>" prompt:-
#
# >>> import afg[RETURN/ENTER]
#
# And away we go...
#
# The waveforms generated are unfiltered and therefore not "pure",
# but hey, an audio function generator signal source, for free, without
# external hardware, AND, using standard Python, what more do you want... :)
#
# Using my notebook about 150mV p-p was generated at the earphone
# socket(s).
#
# Coded on a(n) HP dual core notebook running PCLinuxOS 2009 and
# Python 2.5.2 for Linux...
#
# You will need an oscilloscope connected to the earphone socket(s)
# to see the resultant waveform(s) generated, or listen to the
# harshness of the sound... ;o)
#
# It is EASILY possible to generate pseudo-random noise also but
# I'll leave that for you to work out... :)

# Import any modules...
import os

# Clear a terminal window ready to run this program.
print os.system("clear"),chr(13),"  ",chr(13),

# The program proper...
def main():
	# Make all variables global, a quirk of mine... :)
	global sine
```

```
global square
global triangle
global sawtoothplus
global sawtoothminus
global pulseplus
global pulseminus
global waveform
global select
global count

# Allocate values to variables.
# Any discrepancy between random soundcards may require small changes
# in the numeric values inside each waveform mode...
# These all oscillate at around 1KHz.
sine=chr(15)+chr(45)+chr(63)+chr(45)+chr(15)+chr(3)+chr(0)+chr(3)
square=chr(63)+chr(63)+chr(63)+chr(63)+chr(0)+chr(0)+chr(0)+chr(0)
triangle=chr(0)+chr(7)+chr(15)+chr(29)+chr(63)+chr(29)+chr(15)+chr(7)
sawtoothplus=chr(63)+chr(39)+chr(26)+chr(18)+chr(12)+chr(8)+chr(4)+chr(0)
sawtoothminus=chr(0)+chr(4)+chr(8)+chr(12)+chr(18)+chr(26)+chr(39)+chr(63)
pulseplus=chr(0)+chr(63)+chr(63)+chr(63)+chr(63)+chr(63)+chr(63)+chr(63)
pulseminus=chr(63)+chr(0)+chr(0)+chr(0)+chr(0)+chr(0)+chr(0)+chr(0)

# This is the INITIAL default waveform, the Square Wave.
waveform=square
select="G0LCU."
count=1

# A continuous loop to change modes as required...
while 1:
	# Set up a basic user window.
	print os.system("clear"),chr(13),"  ",chr(13),
	print
	print "Simple Function Generator using STANDARD Python 2.5.2"
	print "for PCLinuxOS 2009, issued as Public Domain to LXF..."
	print
	print "Original idea copyright, (C)2009, B.Walker, G0LCU."
	print
	print "1) Sinewave."
	print "2) Squarewave."
	print "3) Triangle."
	print "4) Positive going sawtooth."
	print "5) Negative going sawtooth."
	print "6) Positive going pulse."
	print "7) Negative going pulse."
	print "Q) or q) to quit..."
```

```
		print
		# Enter a number for the mode required.
		select=raw_input("Select a number/letter and press RETURN/ENTER:-
")
		if select=="": select="2"
		if len(select)!=1: break
		if select=="Q": break
		if select=="q": break
		if select=="1": waveform=sine
		if select=="2": waveform=square
		if select=="3": waveform=triangle
		if select=="4": waveform=sawtoothplus
		if select=="5": waveform=sawtoothminus
		if select=="6": waveform=pulseplus
		if select=="7": waveform=pulseminus
		# Re-use the variable ~select~ again...
		if select<=chr(48): select="Default OR last"
		if select>=chr(56): select="Default OR last"
		if select=="1": select="Sine wave"
		if select=="2": select="Square wave"
		if select=="3": select="Triangle wave"
		if select=="4": select="Positive going sawtooth"
		if select=="5": select="Negative going sawtooth"
		if select=="6": select="Positive going pulse"
		if select=="7": select="Negative going pulse"
		print os.system("clear"),chr(13),"  ",chr(13),
		print
		print select+" audio waveform generation..."
		print
		# Open up the audio channel(s) to write directly to.
		audio=file('/dev/audio', 'wb')
		# Make the tone generation time finite in milliseconds...
		# A count of 10000 is 10 seconds of tone burst...
		count=0
		while count<10000:
			# Write the waveform to the audio device.
			audio.write(waveform)
			count=count+1
		# Close the audio device when finished.
		audio.close()
main()

# End of demo...
# Enjoy finding simple solutions to often very difficult problems...
```

2) broken_triangle.py

```
#/usr/bin/pytho
n
import os


#start system

print os.system("clear"),chr(13),"  ",chr(13),


def signals():
        global triangle

        triangle=chr(0)+chr(7)+chr(15)+chr(29)+chr(63)+chr(29)+chr(15)+chr(7
)

        waveform=triangle
        select="GOLCU."
        count=1

        while 1:
                print os.system("clear"),chr(13),"  ",chr(13),
                print

                waveform=triangle
                print os.system("clear"),chr(13),"  ",chr(13),
                print

                audio=file('/dev/audio', 'wb')

                count=0
                while count<20000:
                        audio.write(waveform)
                        count=count+1
                audio.close()

signals()
```

3) triwave.py

```
#!usr/bin/env
python
#coding=utf-8

import pyaudio
import wave

#define stream chunk
chunk = 1024

#open a wav format music
f = wave.open("triangle1Khz.wav","rb")
#instantiate PyAudio
p = pyaudio.PyAudio()
#open stream
stream = p.open(format = p.get_format_from_width(f.getsampwidth()),
                channels = f.getnchannels(),
                rate = f.getframerate(),
                output = True)
#read data
data = f.readframes(chunk)

#paly stream
while data != '':
    stream.write(data)
    data = f.readframes(chunk)

#stop stream
stream.stop_stream()
stream.close()

#close PyAudio
p.terminate()
```

4) twopyrecord.py

```
from sys import byteorder
from array import array
from struct import pack

import pyaudio
import wave

THRESHOLD = 500
CHUNK_SIZE = 1024
FORMAT = pyaudio.paInt16
RATE = 44100
```

```
def is_silent(snd_data):
    "Returns 'True' if below the 'silent' threshold"
    return max(snd_data) < THRESHOLD

def normalize(snd_data):
    "Average the volume out"
    MAXIMUM = 16384
    times = float(MAXIMUM)/max(abs(i) for i in snd_data)

    r = array('h')
    for i in snd_data:
        r.append(int(i*times))
    return r

def trim(snd_data):
    "Trim the blank spots at the start and end"
    def _trim(snd_data):
        snd_started = False
        r = array('h')

        for i in snd_data:
            if not snd_started and abs(i)>THRESHOLD:
                snd_started = True
                r.append(i)

            elif snd_started:
                r.append(i)
        return r

    # Trim to the left
    snd_data = _trim(snd_data)

    # Trim to the right
    snd_data.reverse()
    snd_data = _trim(snd_data)
    snd_data.reverse()
    return snd_data

def add_silence(snd_data, seconds):
    "Add silence to the start and end of 'snd_data' of length 'seconds' (float)"
    r = array('h', [0 for i in xrange(int(seconds*RATE))])
    r.extend(snd_data)
    r.extend([0 for i in xrange(int(seconds*RATE))])
    return r

def record():
    """
    Record a word or words from the microphone and
    return the data as an array of signed shorts.

    Normalizes the audio, trims silence from the
    start and end, and pads with 0.5 seconds of
    blank sound to make sure VLC et al can play
    it without getting chopped off.
    """
    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT, channels=1, rate=RATE,
        input=True, output=True,
        frames_per_buffer=CHUNK_SIZE)

    num_silent = 0
```

```
    snd_started = False

    r = array('h')

    while 1:
        # little endian, signed short
        snd_data = array('h', stream.read(CHUNK_SIZE))
        if byteorder == 'big':
            snd_data.byteswap()
        r.extend(snd_data)

        silent = is_silent(snd_data)

        if silent and snd_started:
            num_silent += 1
        elif not silent and not snd_started:
            snd_started = True

        if snd_started and num_silent > 30:
            break

    sample_width = p.get_sample_size(FORMAT)
    stream.stop_stream()
    stream.close()
    p.terminate()

    r = normalize(r)
    r = trim(r)
    r = add_silence(r, 0.5)
    return sample_width, r

def record_to_file(path):
    "Records from the microphone and outputs the resulting data to 'path'"
    sample_width, data = record()
    data = pack('<' + ('h'*len(data)), *data)

    wf = wave.open(path, 'wb')
    wf.setnchannels(1)
    wf.setsampwidth(sample_width)
    wf.setframerate(RATE)
    wf.writeframes(data)
    wf.close()

if __name__ == '__main__':
    print("please speak a word into the microphone")
    record_to_file('outwavetest.wav')
    print("done - result written to outwavetest.wav")
```

5) doppler_wav.py

```
# -*-
coding:
utf-8 -
*-
#Doppler Radar, Read data from a WAV file
#written by Meng Wei,  a summer exchange student (UCD GREAT Program, 2014) from
Zhejiang University, China
```

```python
import wave
from struct import unpack
from numpy.fft import ifft
from math import log
import matplotlib.pyplot as plt


#read the raw data .wave file here
wavefile = wave.open(r"newt17.wav", "rb")
framerate = wavefile.getframerate()
numframes = wavefile.getnframes()

s=[]
for i in range(numframes):
    val = wavefile.readframes(1)
    right = val[2:4]
    u = unpack('h', right )[0]
    s.append(u*(-1.0)/32768.0)

#constants
c= 3E8 #(m/s) speed of light

#radar parameters
FS=framerate
Tp = 0.250 #(s) pulse time
N = Tp*FS  ## of samples per pulse
fc=2590E6

#creat doppler vs. time plot data set here
sif=[]
for ii in range(1,int(round(len(s)/N))):
             sif.append(s[int((ii-1)*N):int(ii*N)])

#subtract the average DC term here
me=sum(s/)/len(s/)
sif = [[x - me for x in y]for y in sif]
zpad = int(8*N/2)

#doppler vs. time plot
v=ifft(sif)  #now v is a [[]]
vvv=[[20*log(abs(x),10) for x in y]for y in v]
S=[x[:int(len(vvv[0])/2)] for x in vvv]
m=max(max(S))

grid=[[x-m for x in y] for y in S]
```

```
#calculate velocity
delta_f = FS/2
lambdaa=c/fc;
velocity = delta_f*lambdaa/2;
#calculate time
time = Tp*len(v)
plt.imshow(grid, extent=[0,velocity,0,time],aspect='auto')
plt.xlim(0,30)
plt.colorbar()
plt.xlabel('Velocity (m/sec)',{'fontsize':20})
plt.ylabel('time [s]',{'fontsize':20})
plt.title('doppler',{'fontsize':20})
plt.tight_layout()
plt.show()
```

6) doppler_realtime_spi_adc.py

```
# -*- coding: utf-8 -*-
```

```
#Real Time Doppler Radar
#written by Meng Wei, a summer exchange student (UCD GREAT Program, 2014) from Zhejiang University, China

import spidev
from numpy.fft import ifft
from math import log
import pylab as plt

#constants
c=3E8 #(m/s) speed of light

#radar parameters
FS = 5600#Sampling rate with SPI port
Tp = 0.1#(s) pulse time
N = 2000# of samples per pulsRe
fc = 2590E6;#(Hz) Center frequency (connected VCO Vtune to +5)

delta_f = FS/2
lambdaa=c/fc
velocity = delta_f*lambdaa/2
Time = 1
#talk to SPI
spi=spidev.SpiDev()
spi.open(0,0)
```

```
fig=plt.figure()
plt.ion()
plt.show()

for i in range(10):
    plt.cla()
    for k in range(5000):#This loop takes nearly 1 sec
        r=spi.xfer2([1,8<<4,0],16000000,0)
        s.append((((r[1]&3)<<8)+r[2]))

    sif=[]
    for ii in range(1,int(round(len(s)/N))):
             sif.append(s[int((ii-1)*N):int(ii*N)])
    me=sum(s)/len(s)
    sif = [[x - me for x in y]for y in sif]
    v=ifft(sif)  #now v is a [[]]
    vvv=[[20*log(abs(x),10) for x in y]for y in v]
    S=[x[:int(len(vvv[0])/2)] for x in vvv]
    m=max(max(bufferVel))
    grid=[[x-m for x in y] for y in bufferVel]

    plt.imshow(grid, extent=[0,velocity,0,Time],aspect='auto')
    plt.xlim(0,5)
    plt.xlabel('Velocity (m/sec)',{'fontsize':20})
    plt.ylabel('time [s]',{'fontsize':20})
    plt.title('real_time_doppler',{'fontsize':20})
    plt.draw()
```

End of Report