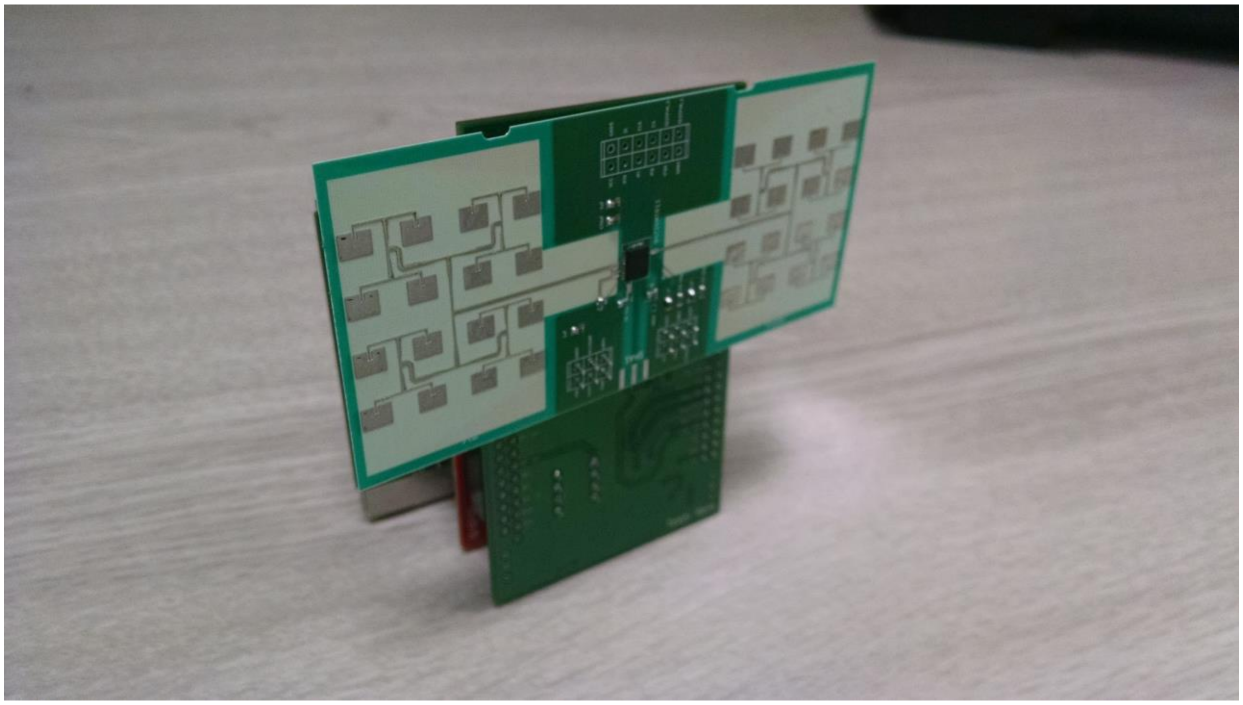
EEC 134 Final Report
RF/Microwave System Design

FALL-WINTER 2015

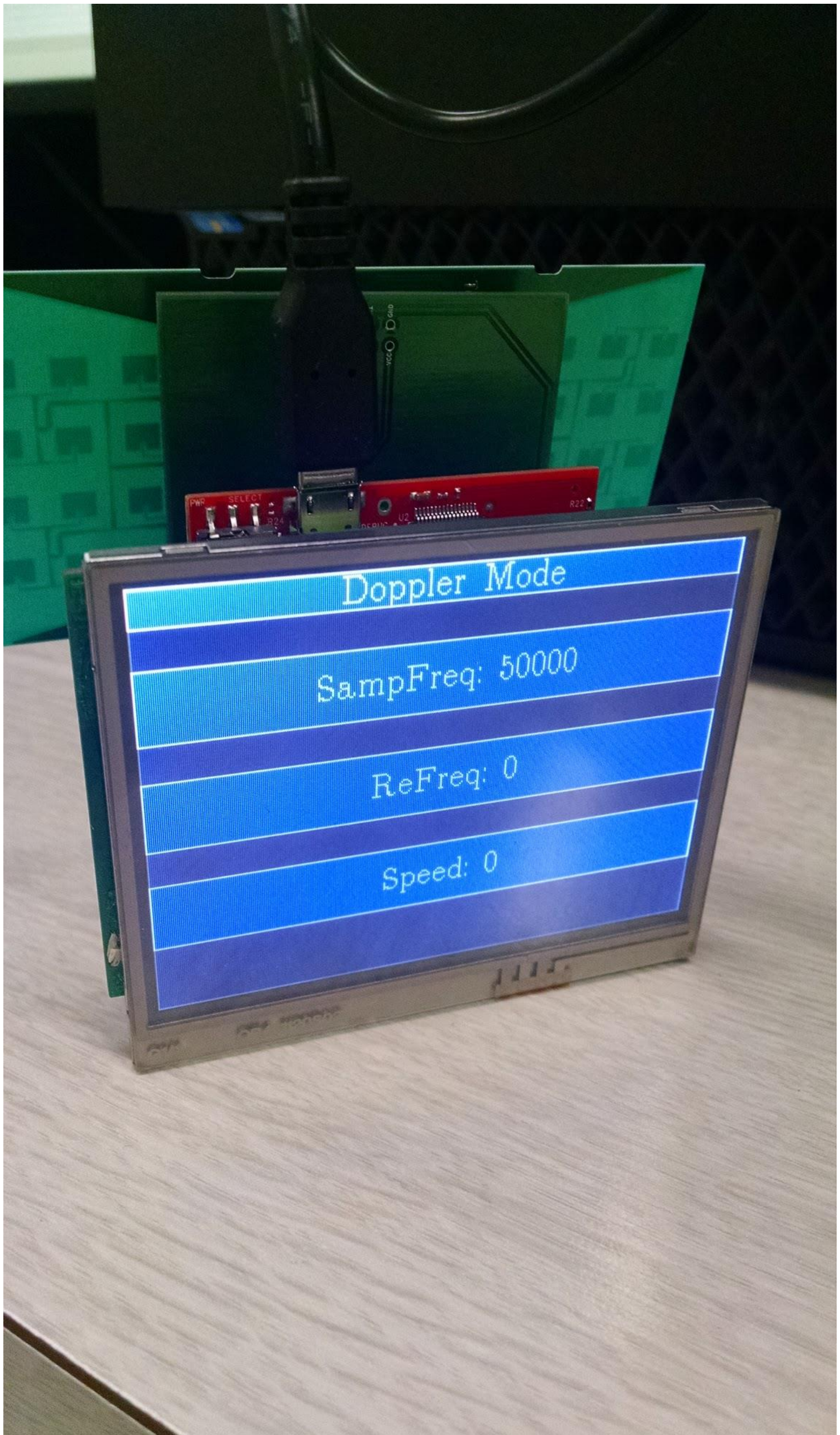
Professor Liu



University of California, Davis

Team Hero

Vincent Li	**997277506**
Patrick Huynh	**997494460**
Ehsan Fanaian	**998499273**


Doppler Mode
SampFreq: 50000
ReFreq: 0
Speed: 0

Abstract

At high frequency in the milliwave range, it allows for smaller frequency reuse distances than lower frequencies. The short wavelength allows modest size antennas to have a small beam width, further increasing frequency reuse potential. Even though radio waves in this band have high atmospheric attenuation, they still have a range within one kilometer, which is suitable for our testing purpose. For the senior design project, we hope to build a 24 Ghz radar system to win the competition in range and speed detection. We aimed at smaller antenna to reduce the overall weight, thinner PCB with advance material, and improve accuracy with high performance electronics.

Introduction

Our team took the challenge to build a 24 Ghz Radar system that utilized the Infineon BGT24MTR11 Chip as the transceiver processor. This system involved touch screen on board DSP, Tiva Programming, baseband PCB Design, and Antenna PCB Design. The 24 Ghz PCB uses RO4350B Laminates material in order for the patch antennas to radiate out the signal at the appropriate frequency. Our vision was to create high frequency speed detection and range detection with a low weight and low power system.

<u>Speed Detection:</u>

Doppler's effect will allow the detection of speed. As the waves reflect back from the object depending on its distance relative to the antennas, the incoming wave will change. By processing the observed frequency with the input frequency, we can calculate the speed. The Doppler frequency shift for active radar is as follows, where F_D is Doppler frequency, F_T is transmit frequency, V_R is radial velocity, and $C$is the speed of light:

$$F_D = 2 \times F_T \times \left(\frac{V_R}{C}\right)$$

<u>Range Detection:</u>

As for the range detection, we will need to vary the emitting frequency with our DAC. Instead of timing the system for distance, we are doing a frequency modulation. Frequency comparison between two signals is considerably more accurate, even with older electronics, than timing the signal. By measuring the frequency of the returned signal and comparing that with the original, the difference can be easily measured.

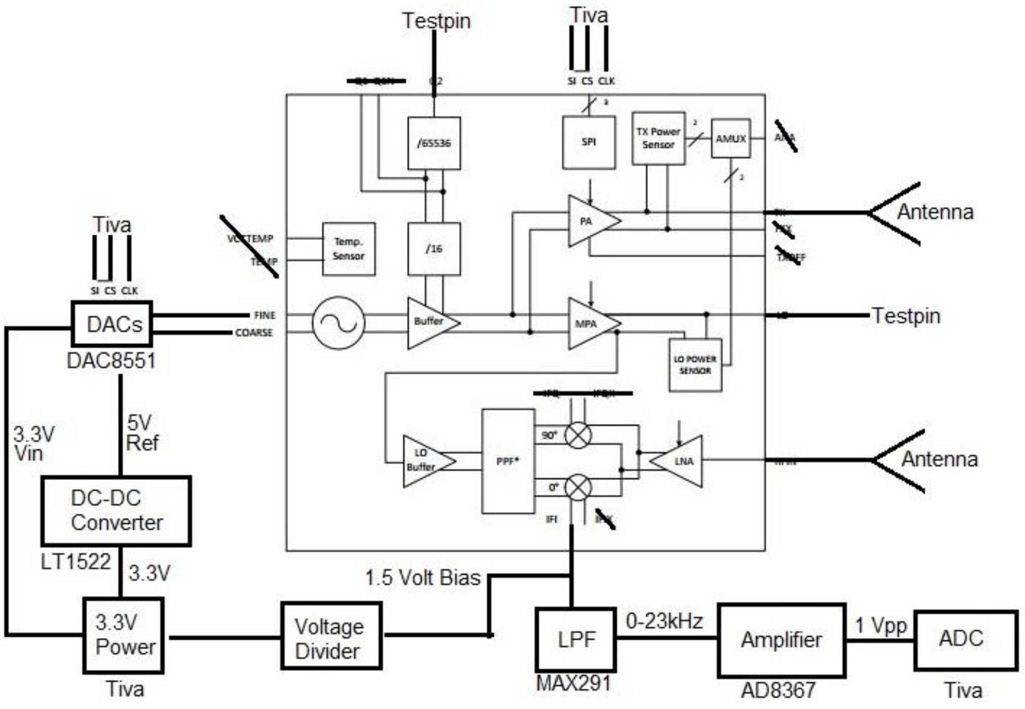
Goals

- Put together all the components into a single radar.
- Improve Signal to Noise ratio
- Implement on board DSP
- Minimize production cost by reducing the size of the PCB’s
- Control the system with a microprocessor and monitor the detection with a touch screen.

Description of the project

We divided the project into three team members - PCB Designer, Code Architect, and DSP.

The first part of the radar system was signal generation, transmission and reception. We used the BGT24MTR11 Chip to save the work of system architecture. A triangle wave at 60 hertz was generated from the Microcontroller with the DAC's for the Voltage Control Oscilloscope inside the chip. The VCO would generate a 24 Ghz signal that pass onto a buffer. The buffer would send out part of the signal at 23 kHz for testing purpose to ensure that the VCO working properly. We could test it with the pin Q2. The rest of the signal would be split into a Power Amplifier and Modular Port Adapter. After the power amplification, the signal radiates out at the TX, which would be our patch antenna. The other signal that went into the Modular Port Adapter would be feed into the LO Buffer and PPF, which then mixed with the received signal. We could take the receiving end of the signal from IFQ, IFQX, IFI and IFIX.



The received signal would feed to the Low Pass Filter and VGA Amplifier to produce a 1 Vpp wave with a DC bias of 1.5V. This signal would be fed into the Tiva.

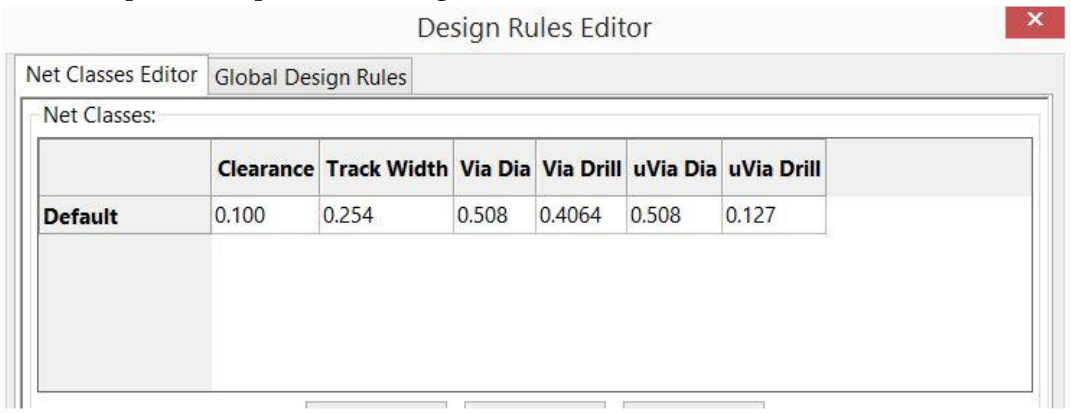
The second part was signal processing with Tiva Microcontroller. The input signal was fed into one of ADCs on the Tiva board, turning the analog voltage in a digital value of size 16 bits. Using the Tivaware library, the ADC was set to trigger at a rate of 50 kHz. Each time the ADC obtained a sample, the value would be stored in a buffer of up to 1024 samples. When the buffer became full, the uDMA would transfer the 1024 samples into an array of size 2048. The process would repeat once more, creating an array of 2048 samples.

These 2048 samples would be then processed. A certain value was subtracted from each of the samples to cancel out the DC component. The samples were then multiplied by a hamming window in order to filter out the low frequency components of the sampled signal. A function in the Tiva DSP library performed FFT on the samples, transforming it from the time domain to the frequency domain. The max of the spectrum became obtainable. The index of the maximum value would correspond to the frequency of the input signal. Once this frequency obtained, calculations could be made to either the speed or the distance, depending on the mode of the radar. Both the calculated frequency and the speed or distance would then be displayed on the touchscreen attached to the Tiva board.

The third part was the PCB implementation of the circuit in the first part. We used two double layer PCB. The first PCB was the antenna, and the second PCB was the baseband. We followed the application guide for the BGT24MTR11 chip to design the antenna PCB. It consisted of two parts: the array patch antenna and the central chip layout for the infineon as the heart of the board. Our antennas array was made with 4x4 patches with a feeding network, designed using ADL and HFSS. The Low Pass filter PCB also included two DAC’s, DC to DC converter, and amplifier. We also biased the output signal by 1.5V to account for clipping.

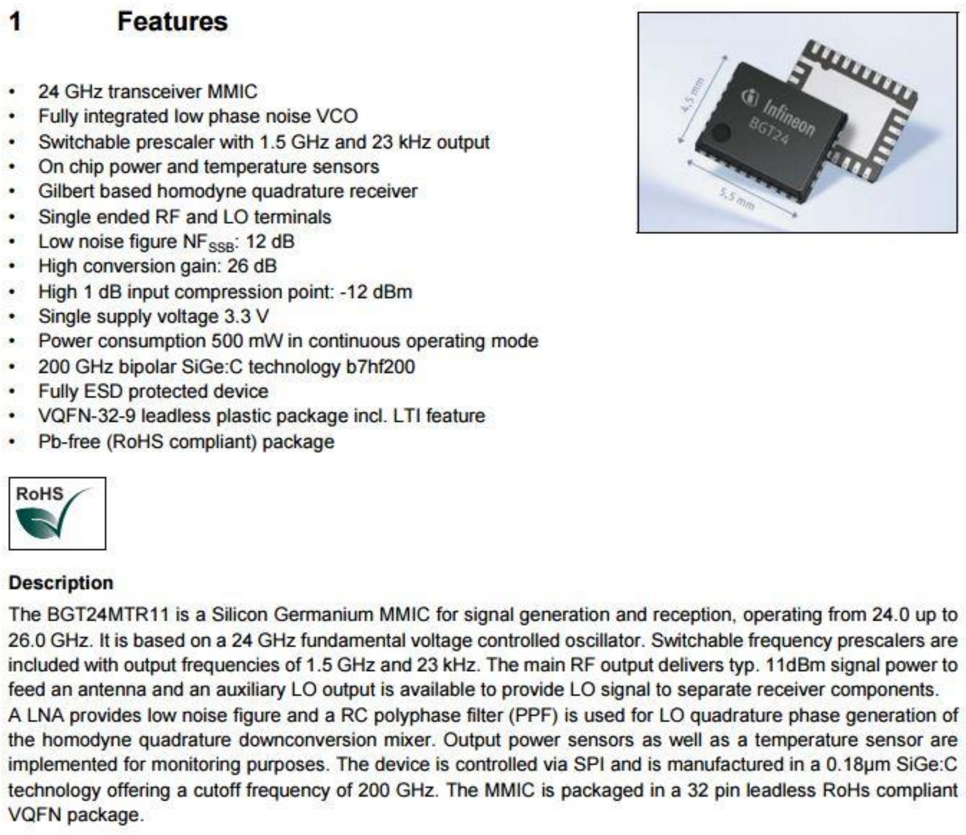
Design Details

The BGT24MTR11 chip integrated all the design elements into one layout. With that convenience, we followed the datasheet to design the PCB. Even though the layout seemed very straight forward, we need to take into the consideration of minimal noise. To achieve it, we placed the power line as far from the signal line as possible. We also varied the width of the line to guarantee the power output, and the signal. We used both 0.254 mm and 0.381 mm tracks.



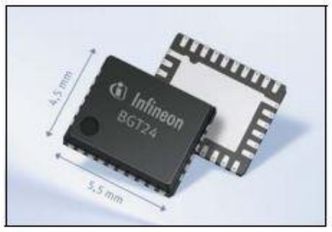
	Clearance	Track Width	Via Dia	Via Drill	uVia Dia	uVia Drill
Default	0.100	0.254	0.508	0.4064	0.508	0.127

BGT24MTR11



1 Features

- 24 GHz transceiver MMIC
- Fully integrated low phase noise VCO
- Switchable prescaler with 1.5 GHz and 23 kHz output
- On chip power and temperature sensors
- Gilbert based homodyne quadrature receiver
- Single ended RF and LO terminals
- Low noise figure NF_{SSB}: 12 dB
- High conversion gain: 26 dB
- High 1 dB input compression point: -12 dBm
- Single supply voltage 3.3 V
- Power consumption 500 mW in continuous operating mode
- 200 GHz bipolar SiGe:C technology b7hf200
- Fully ESD protected device
- VQFN-32-9 leadless plastic package incl. LTI feature
- Pb-free (RoHS compliant) package

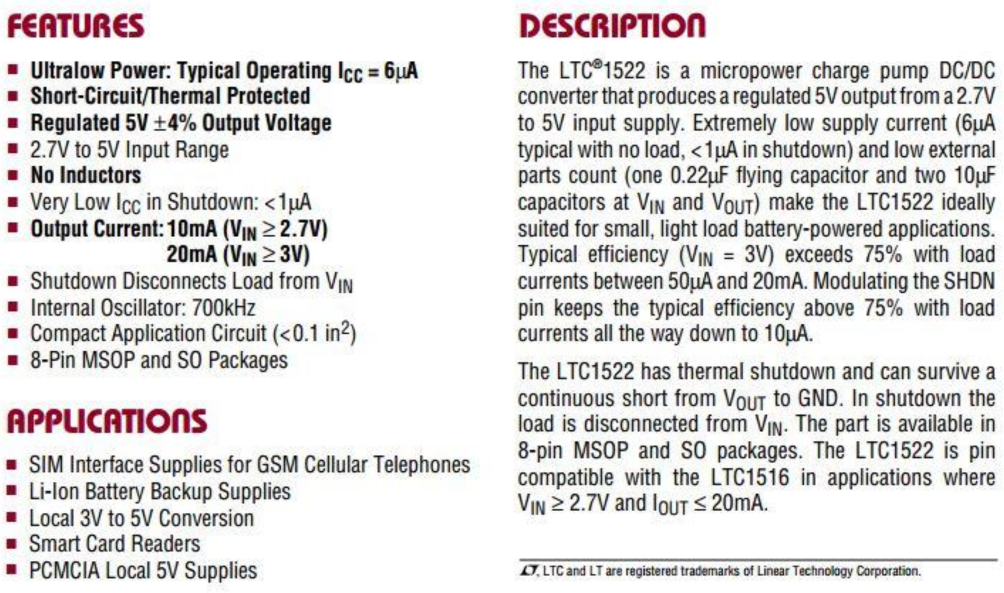


RoHS

Description

The BGT24MTR11 is a Silicon Germanium MMIC for signal generation and reception, operating from 24.0 up to 26.0 GHz. It is based on a 24 GHz fundamental voltage controlled oscillator. Switchable frequency prescalers are included with output frequencies of 1.5 GHz and 23 kHz. The main RF output delivers typ. 11dBm signal power to feed an antenna and an auxiliary LO output is available to provide LO signal to separate receiver components.
A LNA provides low noise figure and a RC polyphase filter (PPF) is used for LO quadrature phase generation of the homodyne quadrature downconversion mixer. Output power sensors as well as a temperature sensor are implemented for monitoring purposes. The device is controlled via SPI and is manufactured in a 0.18µm SiGe:C technology offering a cutoff frequency of 200 GHz. The MMIC is packaged in a 32 pin leadless RoHs compliant VQFN package.

LTC1522



FEATURES

- **Ultralow Power: Typical Operating I_{CC} = 6µA**
- **Short-Circuit/Thermal Protected**
- **Regulated 5V ±4% Output Voltage**
- 2.7V to 5V Input Range
- **No Inductors**
- Very Low I_{CC} in Shutdown: <1µA
- **Output Current: 10mA ($V_{IN} \geq 2.7V$)**
  20mA ($V_{IN} \geq 3V$)
- Shutdown Disconnects Load from V_{IN}
- Internal Oscillator: 700kHz
- Compact Application Circuit (<0.1 in^2)
- 8-Pin MSOP and SO Packages

APPLICATIONS

- SIM Interface Supplies for GSM Cellular Telephones
- Li-Ion Battery Backup Supplies
- Local 3V to 5V Conversion
- Smart Card Readers
- PCMCIA Local 5V Supplies

DESCRIPTION

The LTC®1522 is a micropower charge pump DC/DC converter that produces a regulated 5V output from a 2.7V to 5V input supply. Extremely low supply current (6µA typical with no load, <1µA in shutdown) and low external parts count (one 0.22µF flying capacitor and two 10µF capacitors at V_{IN} and V_{OUT}) make the LTC1522 ideally suited for small, light load battery-powered applications. Typical efficiency (V_{IN} = 3V) exceeds 75% with load currents between 50µA and 20mA. Modulating the SHDN pin keeps the typical efficiency above 75% with load currents all the way down to 10µA.

The LTC1522 has thermal shutdown and can survive a continuous short from V_{OUT} to GND. In shutdown the load is disconnected from V_{IN}. The part is available in 8-pin MSOP and SO packages. The LTC1522 is pin compatible with the LTC1516 in applications where $V_{IN} \geq 2.7V$ and $I_{OUT} \leq 20mA$.

LT, LTC and LT are registered trademarks of Linear Technology Corporation.

AD8367

FEATURES

Broad-range analog variable gain: −2.5 dB to +42.5 dB
3 dB cutoff frequency of 500 MHz
Gain up and gain down modes
Linear-in-dB, scaled 20 mV/dB
Resistive ground referenced input
Nominal Z_{IN} = 200 Ω
On-chip, square-law detector
Single-supply operation: 2.7 V to 5.5 V

APPLICATIONS

Cellular base stations
Broadband access
Power amplifier control loops
Complete, linear IF AGC amplifiers
High speed data I/O

FUNCTIONAL BLOCK DIAGRAM

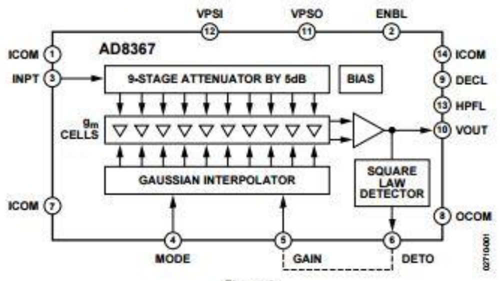


Figure 1.

MAX291

General Description

The MAX291/MAX292/MAX295/MAX296 are easy-to-use, 8th-order, lowpass, switched-capacitor filters that can be set up with corner frequencies from 0.1Hz to 25kHz (MAX291/MAX292) or 0.1Hz to 50kHz (MAX295/MAX296).

The MAX291/MAX295 Butterworth filters provide maximally flat passband response, and the MAX292/MAX296 Bessel filters provide low overshoot and fast settling. All four filters have fixed responses, so the design task is limited to selecting the clock frequency that controls the filter's corner frequency.

An external capacitor is used to generate a clock using the internal oscillator, or an external clock signal can be used. An uncommitted operational amplifier (noninverting input grounded) is provided for building a continuous-time lowpass filter for post-filtering or anti-aliasing.

Produced in an 8-pin DIP/SO and a 16-pin wide SO package, and requiring a minimum of external components, the MAX291 series delivers very aggressive performance from a tiny area.

Features

- ♦ **8th-Order Lowpass Filters:**
  Butterworth (MAX291/MAX295)
  Bessel (MAX292/MAX296)
- ♦ **Clock-Tunable Corner-Frequency Range:**
  0.1Hz to 25kHz (MAX291/MAX292)
  0.1Hz to 50kHz (MAX295/MAX296)
- ♦ **No External Resistors or Capacitors Required**
- ♦ **Internal or External Clock**
- ♦ **Clock to Corner Frequency Ratio:**
  100:1 (MAX291/MAX292)
  50:1 (MAX295/MAX296)
- ♦ **Low Noise: -70dB THD + Noise (Typ)**
- ♦ **Operate with a Single +5V Supply or Dual ±5V Supplies**
- ♦ **Uncommitted Op Amp for Anti-Aliasing or Clock-Noise Filtering**
- ♦ **8-Pin DIP and SO Packages**

DAC8551

16-BIT, ULTRA-LOW GLITCH, VOLTAGE OUTPUT DIGITAL-TO-ANALOG CONVERTER

FEATURES

- **Relative Accuracy: 3LSB**
- **Glitch Energy: 0.1nV-s**
- ***Micro*Power Operation: 140µA at 2.7V**
- **Power-On Reset to Zero**
- **Power Supply: +2.7V to +5.5V**
- **16-Bit Monotonic Over Temperature**
- **Settling Time: 10µs to ±0.003% FSR**
- **Low-Power Serial Interface with Schmitt-Triggered Inputs**
- **On-Chip Output Buffer Amplifier with Rail-to-Rail Operation**
- **Power-Down Capability**
- **Binary Input**
- **$\overline{\text{SYNC}}$ Interrupt Facility**
- **Drop-In Compatible With DAC8531/01 and DAC8550 (2's Complement Input)**
- **Available in a Tiny MSOP-8 Package**

APPLICATIONS

- **Process Control**
- **Data Acquisition Systems**
- **Closed-Loop Servo-Control**
- **PC Peripherals**
- **Portable Instrumentation**
- **Programmable Attenuation**

DESCRIPTION

The DAC8551 is a small, low-power, voltage output, 16-bit digital-to-analog converter (DAC). It is monotonic, provides good linearity, and minimizes undesired code-to-code transient voltages. The DAC8551 uses a versatile 3-wire serial interface that operates at clock rates to 30MHz and is compatible with standard SPI™, QSPI™, Microwire™, and digital signal processor (DSP) interfaces.

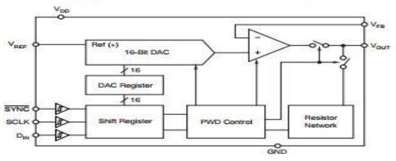
The DAC8551 requires an external reference voltage to set its output range. The DAC8551 incorporates a power-on-reset circuit that ensures the DAC output powers up at 0V and remains there until a valid write takes place to the device. The DAC8551 contains a power-down feature, accessed over the serial interface, that reduces the current consumption of the device to 200nA at 5V.

The low-power consumption of this device in normal operation makes it ideally suited for portable, battery-operated equipment. The power consumption is 0.38mW at 2.7V, reducing to less than 1µW in power-down mode.

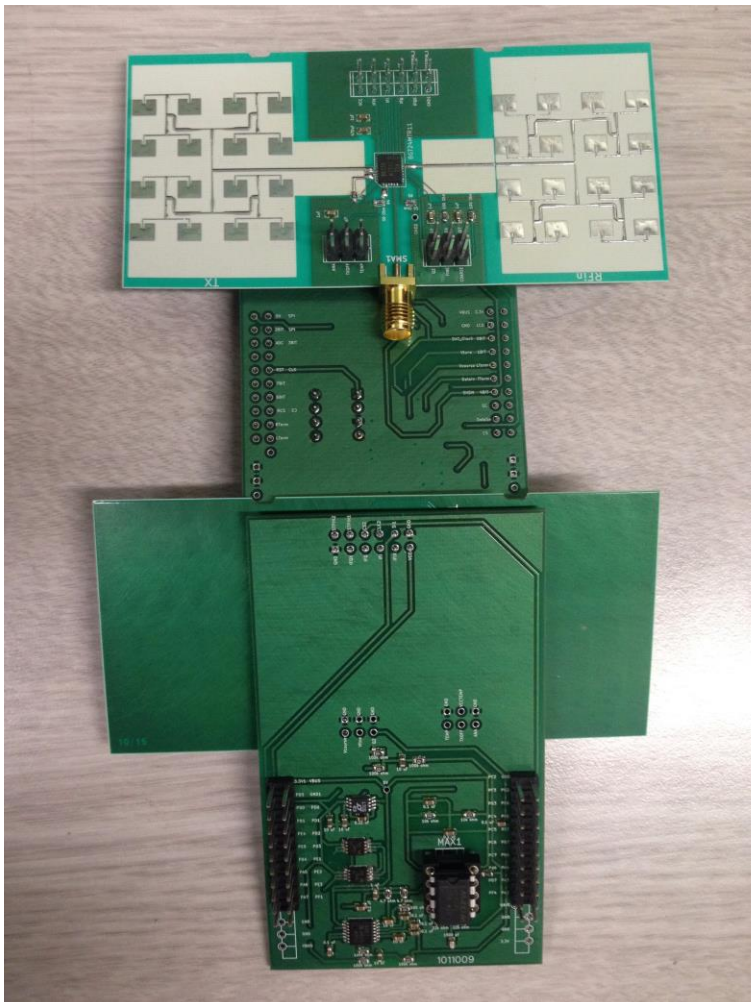
The DAC8551 is available in an MSOP-8 package.

For additional flexibilty, see the DAC8550, a 2's complement-input counterpart to the DAC8551.

FUNCTIONAL BLOCK DIAGRAM



Printed Circuit Board



We built the PCB for portable use. The SMA connector would allow testing for the signal at 1.5Ghz that transmit from the Infineon. We designed the radar to enable maximum frequency detection. The antennas faced opposite from the rest of the PCB's for effective transmission by lowering power dissipation and paths. We also thought of the practicality of use the radar system in reality. This design allowed users to control the signal with the touch screen in the front and point to the source with the antenna in the back.

Design Goal

Our goal is to design a compact, low power and light weight radar system that would also be powerful at detecting short distance object with great accuracy and resolution. At millimeter wave, the system could do imaging such as detecting the physical appearance of a rough surface or the speed of micro fluid in a biomedical chip. With these ambitious goals in mind, we wanted to build the best PCB, and Code with minimal noise.

Schematic

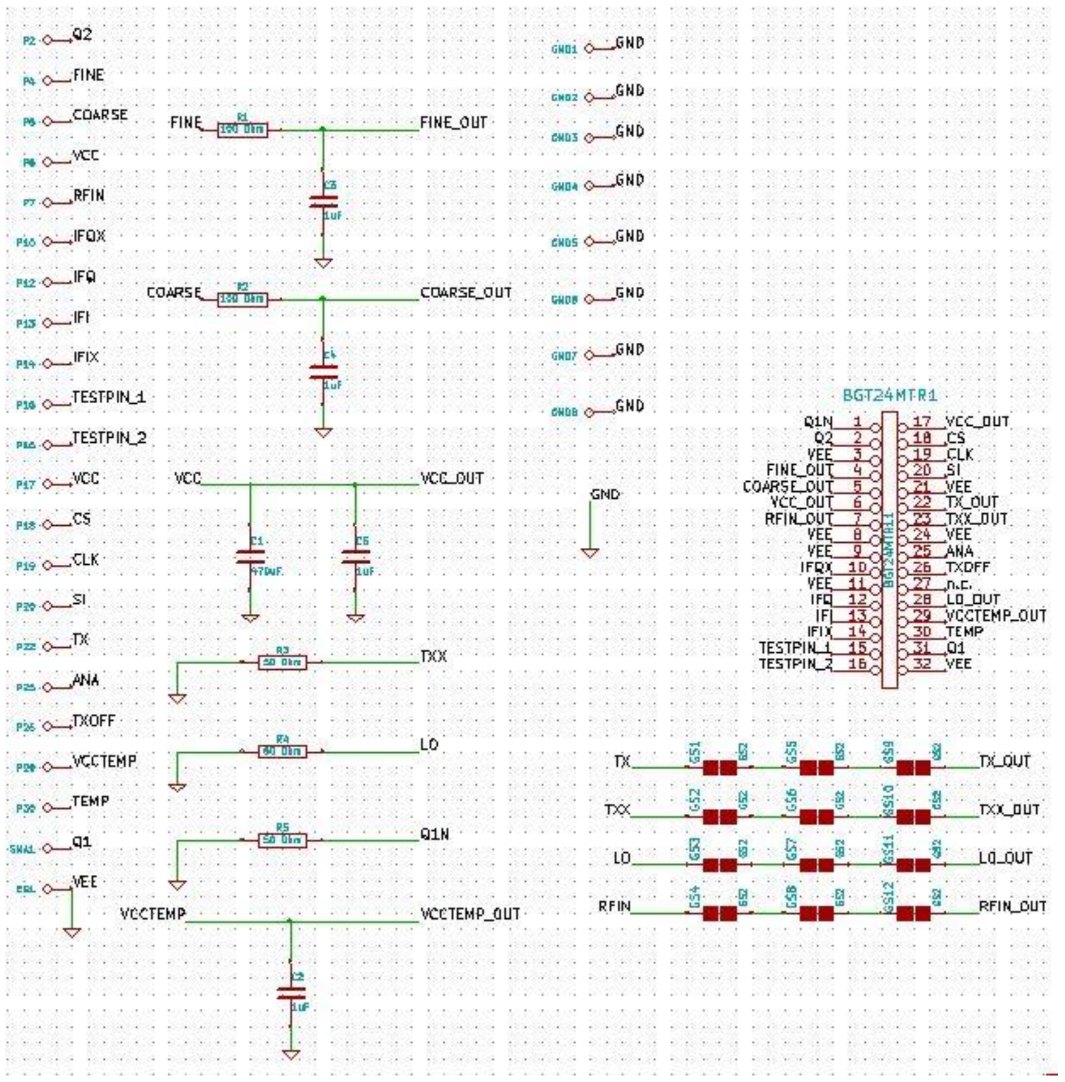
The schematic would be the earliest development of the radar project. It was determined by the datasheet as well as calculations that we did off paperwork. We did extensive reading and confirmation with experts to ensure the connections and all the circuit elements were appropriate.

For this design, we used KiCad. Our PCB Designer explored all the possible source of information to learn about the tool. The result was a clean cut schematic that was easy to understand with all the customized IC library.

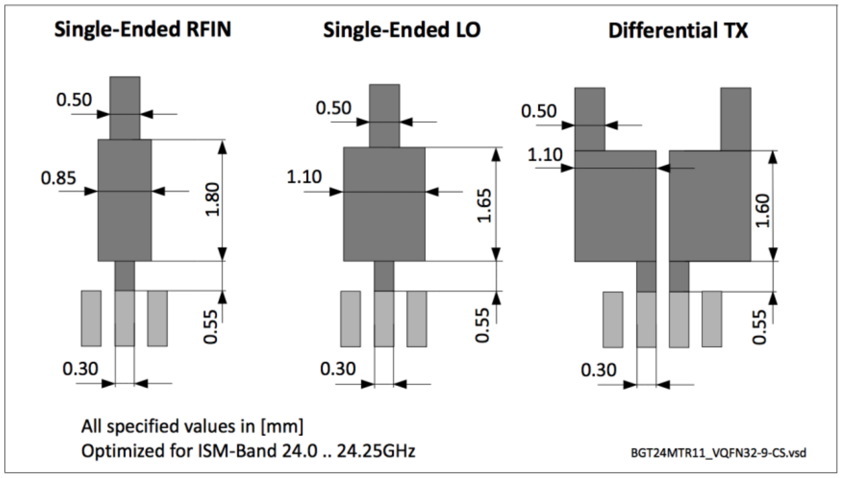
The original KiCad library had very confusing items and most components that we used were not found. Thus, we must draw our own. The schematic drawing was relatively easy because it only involved the right pins, labels and location. Then, we had to make sure the schematic components transfer properly. This process was transforming a rough sketch into the full Printed Circuit Board, which would be what the actually board looked like with all the pins and modules. When we did the schematic component to layout component matching, we rarely found the one that fit exactly. Some of the pins size or distance were not correct, and some shape was not found. Often, we needed to go into the editor and fix the size, shape, length, distance of the surface mount or through hole pins. Sometimes, we would look for the online KiCad library, but finding these designs could take up more time than designing them on our own. We finally became very careful at designing and paying attention to details.

There were also difficulty in following the proper procedure in order to load the component into the layout properly. We tried to load our antenna into the BGT24MTR11 Board so that we could put together the design. KiCad did not allow HFSS file import directly. It turned out that we must convert the HFSS file into a BIM file, and use the Component Creator function in KiCad to generate the antenna. We also had to go into the text file of the converted file to change the layout from front mask to solder mask, in order to have our antenna fully exposed. Even then, we experienced resolution problem with the antenna layout. We went an extra mile to enlarge and increase pixels of the antenna to make it appeared correctly on the layout. We needed to round the edges of the antenna so that the radiation could have minimal loss. Fortunately through the process of trials and error, we found a way to rough the feed in network of the array antenna, which gave us a good testing results.

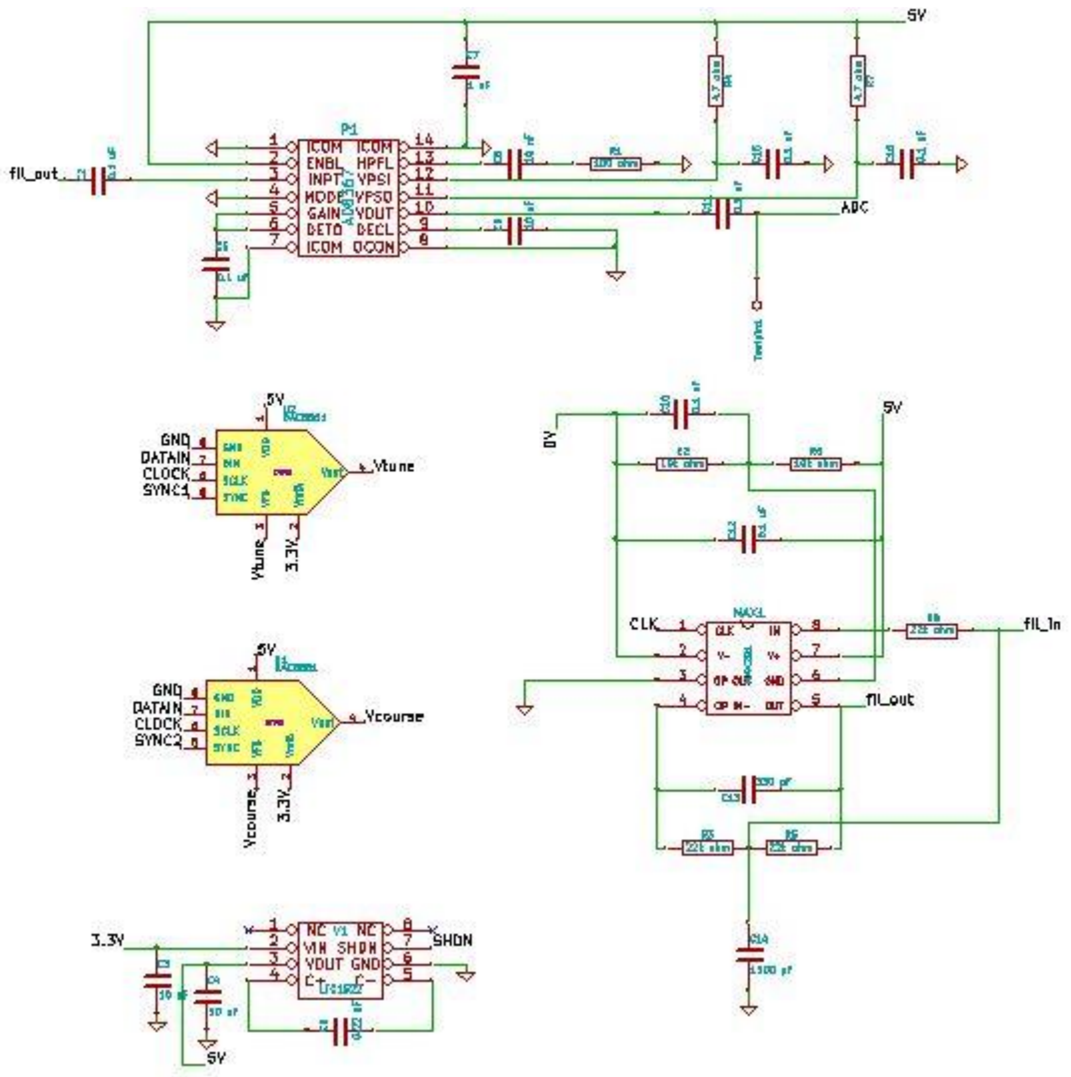
<u>BGT24MTR11</u>
The infineon chip has 32 pins. There were many double pins for noise reduction, and testing. For the pins that we did not need, we tied a 50 ohm resistor to it. We added a large and small capacitor to the DC power supply line for AC reduction. We used Q2 for VCO testing, LO for transmission testing, IFI for receiving testing, VEE for ground, TX for transmission antenna, RFin for receiving Antenna, Vfine and Vcoarse for control VCO and CS, SLK, SI for SPI. For LO and Q1, we need SMA connector due to its high frequency at 1.5 Ghz. We tested Q1 and LO with the spectrometer.
.



The feed-in system for TX, TXX, LO, RFin took some time to design. We needed to adjust the width of the wire blocks connecting out from the BGT24MTR11 chip and into the chip. The purpose is to reduce reflection from the load by impedance matching.



<u>Baseband</u>



Sending out the PCB

We sent out the PCB to Bayarea Circuit. Due to the customization of the 24Ghz antenna PCB on RO4350B material, we waited three weeks before the fabrications finished. Before sending out the antenna, we setup the Chamber for testing the antenna, making sure that it has -15 dbm at 25 Ghz so we know it operates at the right frequency. We also designed multiple versions of the PCB for testing purposes. The regular PCB came back within a week. We ordered all the electrical components from Digikey. These came within a week as well. With everything done, we were at week 7. Everything came back to us at week 10, which put us into the soldering and testing phrase.

PCB Soldering

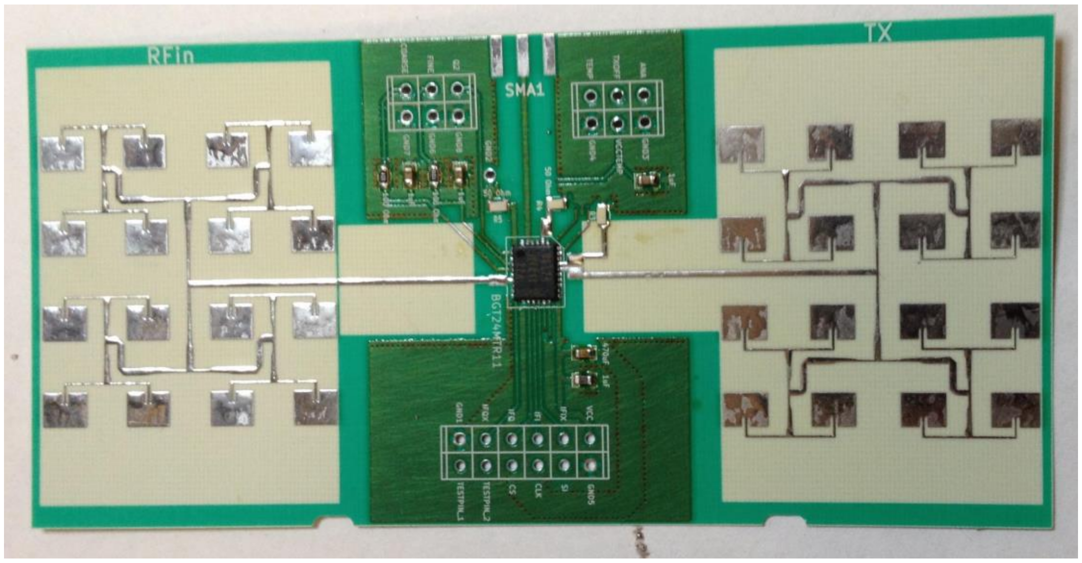
For our designs, traditional soldering would not work with just rosin wires and hot iron. Due to the compact design and small components, we used Rosin Flux (a type of soft rosin material) and hot plates. We carefully put Flux onto all the solder mask and smaller amount on the Surface mount components; such as the DAC's, and amplifier. We set the hot plate to 200 degree celsius for roughly 2-4 minutes. When the PCB heats up, the Flux turned into metal that glues all the components onto the PCB. We examined the binding carefully under a microscope to ensure that we are not shorting any of the pins or misplacing any components.

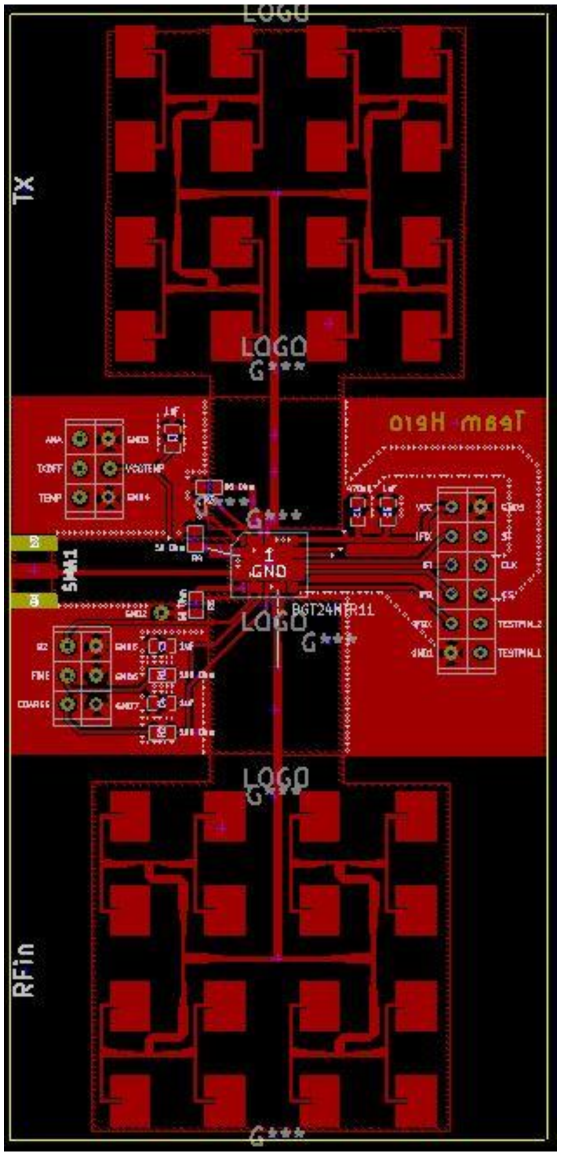
PCB difficulty

The most difficult part was making sure that the multiple pin surface mount chip would not be shorting. When we placed the Flux onto the solder mask, there were always chances that we would put extra. Since the solder mask pins were very small, a small leak of the Flux would glue two pins together. We had to be patient about the soldering process to move the components in circular motion so that the flux would move around evenly. In case there was noticeable electric shortage, we must remove the components and take out the extra flux. Another concern was that the IC cannot tolerate temperature over 180 degree celcius. Otherwise, the chip itself would degenerate. Thus, we must move fast and accurately so that we would not waste time in high temperature, exposing our IC to risk.

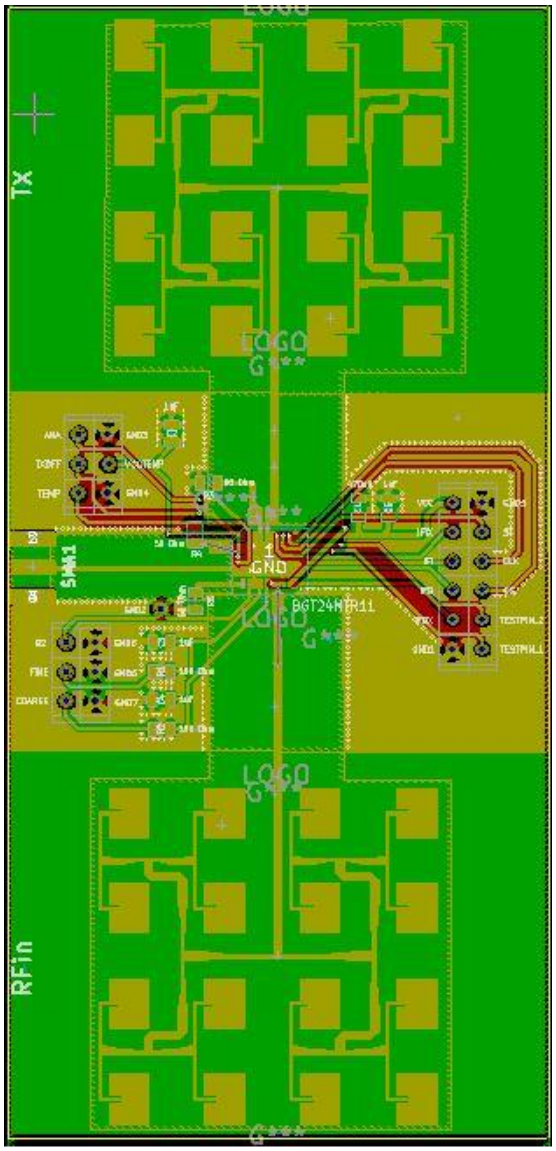
PCB results

BGT24MTR11





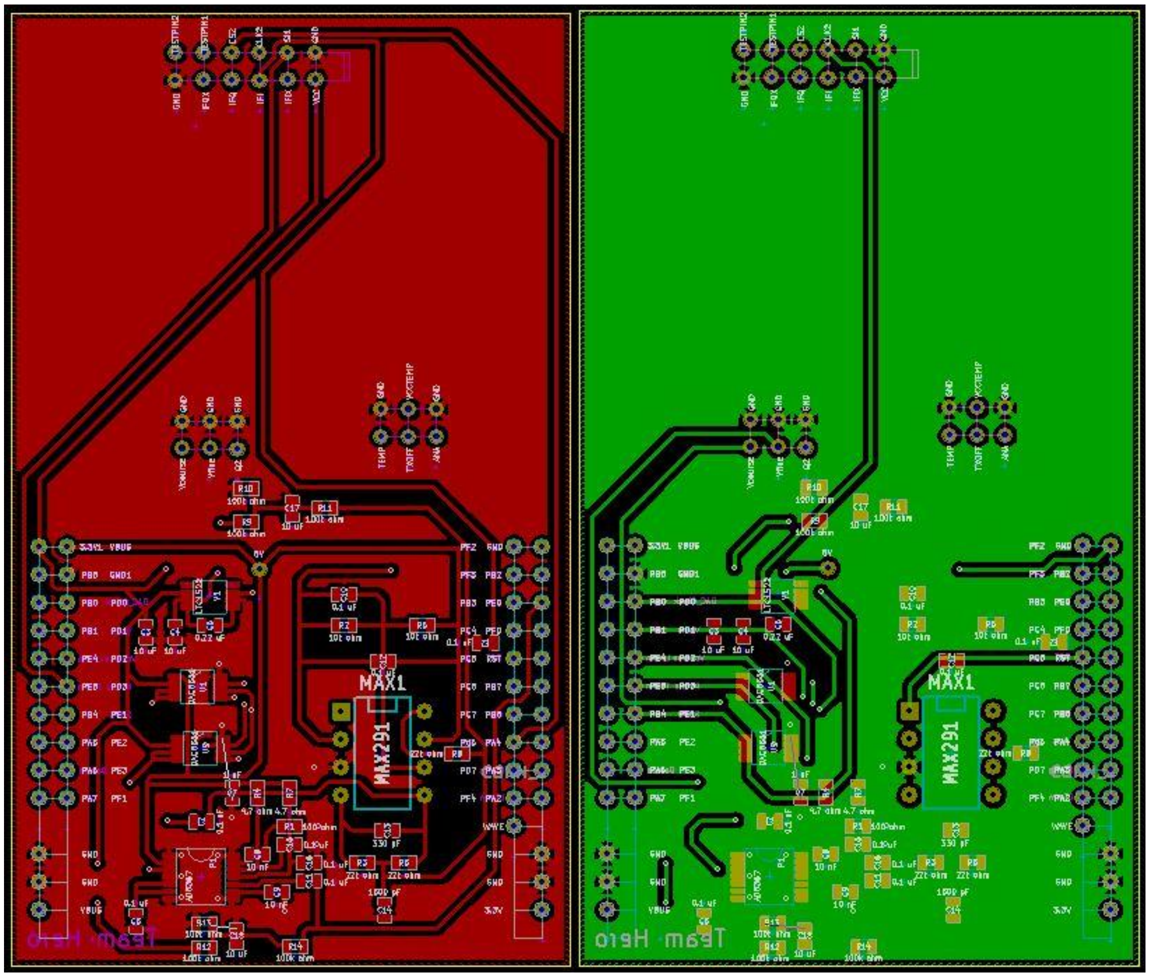
This was the antenna



PCB in KiCad. The array antennas were on the top and bottom of the board for transmission and reception. The feed in system was designed for 50 ohm resistance, following the guideline from the Data sheet. Notice that we placed resistors to some of the pins. It was because the board would transmit the signal at two phases with 90 degree difference to cancel out the noise. To utilize the feature, it require more complicated setup. We only used one phase of the signal and tie the other end to a high frequency 50 ohm resistor.

Low Pass Filter, DAC, DC-DC Converter, Amplifier

We designed a very compact circuit to fit everything onto one chip. The process took a lot of planning, and trials & error. We used both side of the PCB to allocate space for wiring.



The 3.3V would feed into the DC-DC convert (LTC1522) to ramp up to 5V for some of our DAC, and low pass filter. The two DAC were hooked to the SPI in the TIVA. They would be used for range detection with the VCO. The antenna signal would come back from IFI, which would feed into the lower pass filter and AGC Amplifier. The final signal would be processed by the TIVA at pin PE0.

Microcontroller Coding

Design

The intended design of the Tiva code was to have both the receiver and transmitter code for the radar on the same Tiva launchpad. The Tiva was to initialize the SPI for the DACs and the Infineon, the touchscreen, the ADC and the uDMA. After initializing all the needed peripherals, the Tiva would enter an infinite while loop which only contains functions for processing the data and the touchscreen inputs. At certain points during the while loop, interrupts for the DAC, ADC, and the touchscreen would occur. The DAC interrupts would write to the DACs on the PCB and modulate the signal on the Infineon. The ADC interrupts would take samples from the received signal. The touchscreen interrupts would both update the screen and sample the touchscreen at various points, searching for a finger press. The finger press interrupt would allow the Tiva to switch between Doppler and Range modes.

The overall structure of the program can be seen from the main function of the Tiva. The purpose and role of each function will be explained later in the report.

```
int main(void) {

   InitBasics();
   InitGui();
   InitGuiTimer();
   InitSSI();
   InitDACTimer();
   InitDSP();
   InitSamplingTimer();
   InitADC3Transfer();

   IntEnable(INT_ADC0SS3);
   IntMasterEnable();

         while(1)
         {
             if (g_ucDataReady)
             {
                   ProcessData();
              }// if

         WidgetMessageQueueProcess();
         }// while
```

}// main()

While this was the intended design of the Tiva, certain bugs were discovered during the development of the code. One bug caused the touch part of the touchscreen to fail. Since the touchscreen could not receive input, the two different modes, Doppler and Range, could not be toggled. As such, original program had to be separated into two programs: one for Doppler and one for Range. A possible solution was to use the buttons on the Tiva board to toggle the two modes, but were not sued due to lack of time.

The other bug caused the touchscreen to fail to display when the DAC code was running. This bug was complicated and had multiple aspects to it, which will be explained in a later section. This caused the program to separate into two separate programs, one for transmitting and receiving, which are loaded into different Tiva boards.The end design is one Tiva doing all the receiver code and one Tiva doing all the transmitter code.

Devices

There are two main devices that are used in the signal processing: the Tiva and the display.

The device used is the Tiva launchpad, model EK-TM4C123GXL. The particular microcontroller on the launchpad is the TM4C123GH6PM. The two Tiva boards used are of different silicon revisions. One is silicon revision 6, also called Blizzard_RB1. The other is silicon revision 7, called Blizzard_RB2. The silicon revision numbers affect the ROM calls in the Tivaware library, as well as the types of bugs that the Tiva exhibit.

The display is the Kentec display originally meant to be a Stellaris Launchpad boosterpack, model number EB-LM4F120-L35. Since the Tiva is an updated version of the Stellaris, the Kentec display can be adapted for Tiva use.

Development of Code

Most of the code for the project was developed on Code Composer Studio, developed by TI. This was for several reasons. The tutorial for the Tiva, the Tivaware Workshop, uses Code Composer Studio as opposed to Kiel, Energia, or some other toolchain. Code Composer Studio also provides an internal FFT function for the Tiva, allowing the plotting of the frequency domain of the input signal. The final reason to use Code Composer Studio is due to the fact that much of the code involving DSP is based on the code from the EuphonistiHack blog, which mainly uses Code Composer Studio.

The code for the Tiva had borrows from three different sources: the EuphonistiHack blog, the Tiva Workshop, and the Stefan and Joe.

Originally, the EuphonistiHack code was for a Frequency Analyzer for audio files. It was designed using a Stellaris board along with the Kentec display. What was borrowed for this radar project was the code involving sampling and the signal processing. Some parts, however, had to be changed. For example, the EuphonistiHack code originally had two modes of operation for DSP and sampling: DMA_METHOD_FAST and DMA_METHOD_SLOW. Which mode is used depends on the sampling frequency of the project. DMA_METHOD_SLOW was used when the sampling could not obtain 1024 samples before the next screen update. In order to compensate, this mode used Ping-Pong buffers to transmit 256 samples at a time. This way, the DSP could be performed on a set of samples while attempting to obtain more samples. DMA_METHOD_FAST was used when the sampling speed was fast enough to obtain 1024 samples before the next screen update. Only one buffer of size 1024 is used instead of two size 256 buffers. For our radar project, only DMA_METHOD_FAST was used. This is because the screen update for the radar is much slower than that our the EuphonistiHack Frequency Analyzer, 1 Hz instead of 15 Hz.

While porting code, however, there were a lot of errors involving the variable types and naming. For example, in the gui code for the EuphonistiHack project, the sRect struct has an element sYMin. When compiling for the Tiva, however, this throws an error. This is because Tivaware and Stellarisware structs have different element names. In Tivaware, sYmin is i16YMin. While porting code over, this document by TI is used: http://www.ti.com/lit/an/spma050a/spma050a.pdf.

Some of the code involving the SPI was borrowed from Stefan and Joe. The code that sets proper sequence of bits for initialization of the Infineon, for example, is from their code. They also showed us how to perform the SPI for DACs; that particular piece of code was later modified by us. Their original code had SPI run in loop as follows (comments are removed for better visibility):

```
for(;;)
{
    int a = 0;
    while (a <= 4080)
    {
         GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
         outputValue = a;
        ROM_GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0);

        data = highByte(outputValue);
        data = 0x0F & data;
         data = 0x30 | data;
        writeData(data);
```

```
       data =lowByte(outputValue);
      writeData(data);

       ROM_GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, GPIO_PIN_6);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
      a = a+4;
   }

    int b = 4080;
    while (b >= 4)
    {
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
            outputValue = b;
            ROM_GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0);

           data =highByte(outputValue);
           data = 0x0F & data;
           data = 0x30 | data;

          writeData(data);
          data =lowByte(outputValue);

            writeData(data);
            ROM_GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, GPIO_PIN_6);
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 1);
            b = b-4;
    }
}
```

Since we were originally going to have the SPI code for the DAC running parallel to the sampling code on the same Tiva, we had to change the DAC code to utilize interrupts instead. Based on examples in the Tiva workshop, we changed the code to the following:

```
void
Timer2AIntHandler(void)
{
    TimerIntClear(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

    // If on the rising edge
    if (edge) {
        // Increment the data bits
        outputValue += step;

        // Check if at the peak or above the wave
        if (outputValue >= peak) {

            // Go from rising edge to falling edge
            edge = !edge;

        }// if
    }// if

    // Else on the falling edge of the wave
    else {

        // Decrement the data bits
        outputValue -= step;

        // Check if at zero or below (below zero is bad)
        if (outputValue <= 0) {

            // Go from rising edge to falling edge
            edge = !edge;

        }// if
    }// else

    // Write data
    write12bit();

}// Timer2AIntHandler()
```

The above is the interrupt handler for the DAC code. Instead of defining the output value before a while loop and decrementing inside of the loop, we defined the output value globally. Each time the interrupt handler is called, outputValue is incremented or decremented depending on whether the code is currently dealing with the rising or falling edge of the triangle wave.

Both the DAC code and DSP code will be explained in further detail later in the report. These snippets of code are merely examples of how we borrowed code from various sources and modified them to our needs.

Organization of Code

The code is organized into several different source files as well as well as their corresponding header files.

The files arm_common_table.h, arm_math.h, cmsis_ccs.h, core_cm0.h, core_cm3.h, core_cm4.h, core_cmFunc.h, core_cmInstr.h, and dsplib-cm4f.lib provide the functions for used during signal processing. All these files are from the EuphonistiHack github.

The files doppler.c and doppler.h hold the main function of the program. They call functions from the other source files.

The files dac.c and dac.h control the DAC code of the Tiva.

The files dsp.c, dsp.h, sampling.c, sampling.h, ti_hamming_windows_vector.c, have to do with the sampling and DSP code for the Tiva. Much of the code found here is borrowed from the EuhponistiHack blog.

The files gui.c, gui.h, Kentec320x240x16_ssd2119_8bit.c, Kentec320x240x16_ssd2119_8bit.h, touch.c, and touch.h deal with properly displaying the results onto the screen. The Kentec320x240x16_ssd2119_8bit and touch files are from the EuphonistiHack blog.

The remaining files are ustdlib.c, driverlib.lib, grlib.lib, and tm4c123gh6pm_startup_cs.c. The library files are linked files from the Tivaware Library. The ustdlib.c file is also from the Tivaware library and is used to properly format text strings onto the Kentec display. The last file contains the interrupt vector table.

Receiver Code

This section will walk through step by step how the receiver code works.

First, the InitBasics() function in doppler.c initializes the clock speed to 80 MHz and allows the use of floating point calculations, which will be important during the DSP step.

```
void InitBasics(void) {

   ROM_FPUEnable();
   ROM_FPULazyStackingEnable();

   // Set clock to 80 MHz
   ROM_SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL |   SYSCTL_XTAL_16MHZ
|SYSCTL_OSC_MAIN);

}// IntiBasics()
```

Then the initialization of the GUI is performed in the functions InitGui() and InitGuiTimer(), both found in gui.c.

```
void InitGui(void){

        //
        // Initialize LCD screen
        //
        Kentec320x240x16_SSD2119Init();

        TouchScreenInit();
        TouchScreenCallbackSet(WidgetPointerMessage);

        //
        // Add Widgets to screen.
        //
        WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);
        WidgetAdd((tWidget *)&g_sBackground, (tWidget *)&g_sTitle);
        WidgetAdd((tWidget *)&g_sBackground, (tWidget *)&g_sSampFreq);
        WidgetAdd((tWidget *)&g_sBackground, (tWidget *)&g_sReceivedFreq);
        WidgetAdd((tWidget *)&g_sBackground, (tWidget *)&g_sSpeed);
        WidgetPaint(WIDGET_ROOT);

        //
        // Update Sampling Frequency
        //
        usprintf(sampFreqText,"SampFreq: %d", g_uiSamplingFreq);
        CanvasTextSet(&g_sSampFreq, sampFreqText);
        WidgetPaint((tWidget *)&g_sSampFreq);

}// InitGui
```

InitGui() first calls Kentec320x240x16_SSD2119Init(), which is from Kentec320x240x16_ssd2119_8bit.c, which is from the Euphonitihack blog. How Kentec320x240x16_SSD2119Init() works is not too important; it enables pins on port A and B, writes to the Kentec display and turns it on.

TouchScreenInit() is a bit problematic. This is a function from touch.c, found on both the EuphonistiHack Github and the Tiva Workshop. The problem is that the two touch.c files are different. In the original Tiva Workshop version of touch.c, the TouchScreenInit() prepares and ADC with the following code:

```
        ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_TIMER, 0);
```

And later ….

```
        TimerConfigure(TIMER1_BASE, (TIMER_CFG_SPLIT_PAIR |
                        TIMER_CFG_A_PERIODIC |
                        TIMER_CFG_B_PERIODIC));
        TimerLoadSet(TIMER1_BASE, TIMER_A, (SysCtlClockGet() / 1000) - 1);
        TimerControlTrigger(TIMER1_BASE, TIMER_A, true);
```

The Kentec display needs to use an ADC on the Tiva in order to process finger presses; where and how hard the finger press on the touchscreen corresponds to an analog voltage value, which is then converted to a digital value on the Tiva through an ADC. The ADC is called every one millisecond through Timer1. Using TimerControlTrigger(), the ADC is hardware triggered, meaning that it bypasses the CPU or processor.

The code from touch.c from EuphonisticHack is different as seen here:

```
ADCSequenceConfigure(ADC1_BASE, 3, ADC_TRIGGER_PROCESSOR, 1);
```

And later ….

```
void
Timer1AIntHandler(void)
{
         // Clear interrupt
        TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

        // Trigger ADC to sample
        ADCProcessorTrigger(ADC1_BASE, 3);
}// Timer1AIntHandler()
```

Instead of having the ADC be hardware triggered, the ADC is software triggered. When the ADC was hardware triggered using TimerControlTrigger(), Timer1 did not need at interrupt handler because the CPU was always bypassed when Timer1 calls the ADC. Now, Timer1 calls Timer1AIntHandler(), which manually calls the ADC with ADCProcessorTrigger(). The reason for this is because the DSP needs to use an ADC as well. Notice that TimerControlTrigger() does not have any function parameters; it does not differentiate between ADCs. So when TimerControlTrigger is called with both ADCs being hardware triggered, both the touchscreen and DSP ADC will be called. In order to prevent that, only the DSP ADC is hardware triggered and the touchscreen ADC is software triggered.

Going back to InitGui(), the next function is TouchScreenCallbackSet(). It also initializes the touchscreen for processing presses. Since the touchscreen does not work, which will be explained later, it is not important how this function works.

The next few lines of code attaches the canvases to the widget tree. Canvases are rectangles that show up on the the Kentec Display. They can be modified to have different colors or show text. They are defined in the Graphics section of the Tivaware library. They are initialized as a global struct as seen here:

```
Canvas(g_sSampFreq, 0, 0, 0,
        &g_sKentec320x240x16_SSD2119, 0, 40, 320, 40,
        (CANVAS_STYLE_FILL | CANVAS_STYLE_OUTLINE | CANVAS_STYLE_TEXT),
        ClrBlue, ClrWhite, ClrWhite, g_psFontCm20,
        sampFreqText, 0, 0);
```

This particular canvas is for displaying the sampling frequency on the screen. When the line of code, *WidgetAdd((tWidget *)&g_sBackground, (tWidget *)&g_sSampFreq);* runs, the this canvas becomes a child of the background canvas. This means that when displaying different canvases on the Kentec display, the box containing the sampling frequency is above the background picture.

The last few lines of InitGui() are:

```
        usprintf(sampFreqText,"SampFreq: %d", g_uiSamplingFreq);
        CanvasTextSet(&g_sSampFreq, sampFreqText);
        WidgetPaint((tWidget *)&g_sSampFreq);
```

Usprintf() is from the file ustdlib.c. The function works much like sprintf() in the C library, which allows the conversion from integer to float. If one were to use the sprintf() function, however, nothing shows up on the Kentec display. It appears that the normal sprintf() function does not format the string correctly on a Tiva.

CanvasTextSet is macro found in the Graphics library which updates a canvas with a new string. The reason that this macro is needed is because normal pointer reassignment does not work with the Canvas struct. The canvas struct takes in a const char pointer for the its string, which means that the string cannot update during runtime. Since the canvas struct is created at compile time, we could not dynamically construct a new Canvas during runtime. The workaround is to use this macro which creates a new const char pointer based on the new string and reassigns it to the struct.

The last part, WidgetPaint() simply updates the screen based on the Canvas parameter. Passing in the the Sampling Frequency Canvas means that only that particular canvas in updated.

The next function to discuss is InitGuiTimer() which is shown here:

```
void
InitGuiTimer()
{
	 // Enable the timer
	SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);

	// Full Width Periodic Timer using Timer
	TimerConfigure(TIMER3_BASE, TIMER_CFG_PERIODIC);

	// Set timer
	TimerLoadSet(TIMER3_BASE, TIMER_A, SysCtlClockGet()-1);

	// Enable the gui interrupt
	IntEnable(INT_TIMER3A);

	// When timer hits zero, call interrupt
	TimerIntEnable(TIMER3_BASE, TIMER_TIMA_TIMEOUT);

	// Start the gui timer
	TimerEnable(TIMER3_BASE, TIMER_A);
}// InitGuiTimer()
```

The code here is pretty straightforward. It sets Timer3 to trigger at a rate of one Hz. Each time Timer3 hits zero, updateGui() is called:

```
void updateGui(void) {

   //
   // Update Received Frequency
   //
	usprintf(reFreqText,"ReFreq: %d", reFreq);
	CanvasTextSet(&g_sReceivedFreq, reFreqText);
	WidgetPaint((tWidget *)&g_sReceivedFreq);
```

```
        //
        // Update Received Frequency
        //
        usprintf(speedText,"Speed: %d", speed);
        CanvasTextSet(&g_sSpeed, speedText);
        WidgetPaint((tWidget *)&g_sSpeed);

}// updateGui()
```

Much like in the later part of InitGui(), this function updates the calculated frequency of the received signal as well as the calculated speed.

Now that the Kentec display is on and the GUI timer is ticking down, the next step is to initialize the DSP and sampling. This is achieved by InitDSP(), InitSamplingTimer(), and InitADC3Transfer().

InitDSP() is simple, as seen here:

```
void InitDSP(void){

   // Determine the
   g_HzPerBin = (float)g_uiSamplingFreq / (float)NUM_SAMPLES;

   // Call the CMSIS real fft init function
   arm_rfft_init_f32(&fftStructure, &cfftStructure, NUM_SAMPLES, INVERT_FFT,
                           BIT_ORDER_FFT);

}// InitDSP()
```

When the 2048 point FFT function is called, the output is a spectrum with 2048 bins. The range of each frequency that each bin represents depends on the sampling frequency divided by the 2048. g_HzPerBin is that value. The other function is a function from the DSP library which initializes the FFT function.

The next step is to initialize the sampling timer, as follows:

```
void
InitSamplingTimer()
{
        // Enable the timer0
```

```
	SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

	// Full Width Periodic Timer using Timer0
	TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

	// Enables ADC trigger output
	TimerControlTrigger(TIMER0_BASE, TIMER_A, true);

	// Set timer by dividing system clock freq by sampling freq
	// to get the # of clock cycles per period
	TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet()/(g_uiSamplingFreq - 1));

	// Enable the sampling interrupt
	IntEnable(INT_TIMER0A);

	// When timer hits zero, call interrupt
	TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

	// Start the sampling timer
	 TimerEnable(TIMER0_BASE, TIMER_A);
}// InitSamplingTimer()
```

The code is very similar to the GUI timer. One difference is that instead of using timer 3, it uses timer 0. The other difference is that instead triggering at a rate of one Hz, it is now the Sampling Frequency.

The next step is to set up the transferring of data between the ADC to the place to be processed. This is initialized in the following function:

```
void InitADC3Transfer(void)
{
	 // Index of g_ulADCValues
	unsigned int uIdx;

	// Set data as not ready to be processed
	g_ucDataReady = 0;

	// Init buffers by setting them all to 0
	// Should go from 0 to 2048
	for(uIdx = 0; uIdx < NUM_SAMPLES; uIdx++)
	{
```

```
 g_ulADCValues[uIdx] = 0;
}// for

// Configure and enable the uDMA controller
SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);

// Enable the uDMA error interrupt
IntEnable(INT_UDMAERR);

 // Enable uDMA
 uDMAEnable();

// Sets the base address of the control table
 // The control table is the 1024-byte-aligned base address
// that was set up with a preprocessor statement earlier
uDMAControlBaseSet(ucControlTable);

//
// Configure the ADC to capture one sample per sampling timer tick
// which is controled by Timer0
//

 // Enable and reset the ADC
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
SysCtlPeripheralReset(SYSCTL_PERIPH_ADC0);

// Set up the ADC so that it will sample when Timer0 times out
// It is Timer0 which activates the ADC because it was configured
// with TimerControlTrigger()
ADCSequenceConfigure(ADC0_BASE, ADC_SEQUENCER,
                                    ADC_TRIGGER_TIMER,0);
ADCSequenceStepConfigure(ADC0_BASE, ADC_SEQUENCER, 0, ADC_CTL_CH0 |
                                                       ADC_CTL_IE | ADC_CTL_END);

// Enable the sequencer
// ADC_SEQUENCER should be 3
ADCSequenceEnable(ADC0_BASE, ADC_SEQUENCER);
ADCIntEnable(ADC0_BASE, ADC_SEQUENCER);

//
// Configure the DMA channel
```

```
        //
        uDMAChannelAttributeDisable(UDMA_CHANNEL_ADC3,
                                    UDMA_ATTR_ALTSELECT |
                                    UDMA_ATTR_USEBURST |
                                    UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK);

        // Use primary data structure for ADC3
        // Uses 16 bit words
        // Do not increment source address
        // Increment destination address by 16 bits
        // What's arbitration size
        uDMAChannelControlSet(UDMA_CHANNEL_ADC3 | UDMA_PRI_SELECT,
                              UDMA_SIZE_16 | UDMA_SRC_INC_NONE |
                              UDMA_DST_INC_16 | UDMA_ARB_1);


    // Use primary data structure and use ADC3
    // Use basic transfer
    // Source is something to do with ADC3 address
    // Destination is g_ulADCValues
    // Transfer UDMA_XFER_MAX (1024) samples
    uDMAChannelTransferSet(UDMA_CHANNEL_ADC3 | UDMA_PRI_SELECT,
                           UDMA_MODE_BASIC,
                           (void *)(ADC0_BASE + ADC_O_SSFIFO3
                           + (0x20 * UDMA_ARB_1)),
                           g_ulADCValues, UDMA_XFER_MAX);

        // Enable the DMA channel
        uDMAChannelEnable(UDMA_CHANNEL_ADC3);
}// InitADC3Transfer()
```

The first half of the code deals with initializing the uDMA. The uDMA requires a control table which aligns everything in sets of 1024 bytes and is set in the line

uDMAControlBaseSet(ucControlTable). This is reason why even though the program performs 2048 point FFT, the uDMA can only transfer up to 1024 samples at a time. The interrupt in the function IntEnable(INT_UDMAERR) does not actually control the transfer of data for the ADC but deals with errors generated by the uDMA. g_ucDataReady is set to zero to signify that there are not enough samples for the signal processing to begin.

The second half of the code deals with how the sampling and transferring of data should be like. ADCSequenceStepConfigure() sets the ADC so that ADC0 is used, it will only take one sample at a time, put the value in the sequencer 3 (which is a buffer which only holds one sample), and that the ADC will be controlled by an interrupt. uDMAChannelTransferSet makes it so that the uDMA always takes a sample from the same place, the ADC, and puts the sample into g_ulADCValues, moving one space over each time.

While in the context of the ADC, it is important to introduce the ADC interrupt handler which will be called after 1024 samples are obtained.

```
void
ADC3IntHandler(void)
{
   unsigned long ulStatus;
   static unsigned long uluDMACount = 0;
   static unsigned long ulDataXferd = 0;
   unsigned long ulNextuDMAXferSize = 0;


   // Clear the ADC interrupt
   ADCIntClear(ADC0_BASE, ADC_SEQUENCER);

   // If the channel's not done capturing, we have an error
   if(uDMAChannelIsEnabled(UDMA_CHANNEL_ADC3))
   {
         // Increment error counter
         g_ulBadPeriphIsr2++;

         // Disable the ADC interrupt
         ADCIntDisable(ADC0_BASE, ADC_SEQUENCER);

         // Drop pending interrupts associated with ADC0
         IntPendClear(INT_ADC0SS3);

         // Exit interrupt
         return;
   }// if

   ulStatus = uDMAChannelSizeGet(UDMA_CHANNEL_ADC3);

   // If non-zero items are left in the transfer buffer
   // Something went wrong
   if(ulStatus)
   {
         // Increment error counter
         g_ulBadPeriphIsr1++;

         // Exit interrupt handler
         return;
   }// if

   // Disable the sampling timer
   TimerDisable(TIMER0_BASE, TIMER_A);

   uluDMACount++;

   // The amount of data transferred increments in sets of 1024
   ulDataXferd += UDMA_XFER_MAX;


   if(NUM_SAMPLES > ulDataXferd)
   {


         if((NUM_SAMPLES - ulDataXferd) > UDMA_XFER_MAX)
         {
                 ulNextuDMAXferSize = UDMA_XFER_MAX;
         }// if


         else
         {
                 ulNextuDMAXferSize = NUM_SAMPLES - ulDataXferd;
         }// else


         uDMAChannelTransferSet(UDMA_CHANNEL_ADC3 | UDMA_PRI_SELECT,
                                                             UDMA_MODE_BASIC,
                                                             (void *)(ADC0_BASE + ADC_O_SSFIFO3
                                                                     + (0x20 * UDMA_ARB_1)),
                                                             g_ulADCValues + (UDMA_XFER_MAX *
                                                                     uluDMACount),
                                                             ulNextuDMAXferSize);

         // Enable channel with new settings
         uDMAChannelEnable(UDMA_CHANNEL_ADC3);

         // Reset the timer to maximum
         TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet()/(g_uiSamplingFreq - 1));

         // Enable the timer with new settings
         TimerEnable(TIMER0_BASE, TIMER_A);
   }// if

   else
   {
         // Since data will be processed, set counters back to 0
         uluDMACount = 0;
         ulDataXferd = 0;

         // Disable sampling for now while processing
         ADCIntDisable(ADC0_BASE, ADC_SEQUENCER);

         // Remove pending interrupts for the ADC
         IntPendClear(INT_ADC0SS3);

         // Signal that we have new data to be processed
         g_ucDataReady = 1;
   }// else

}// ADC3IntHandler()
```

There are two main parts to the interrupt handler. One part runs when there is not enough samples to run the DSP. The other runs when there is enough samples in g_ulADCValues to run the DSP.

The if block runs when there is only 1024 samples in g_ulADCValues. Since there needs to be 2048 samples before the data can be processed, the program must start sampling again. It seems that 1024 samples is the maximum number of samples that can be transferred at a time before uDMA has to be restarted again because the uDMA control table is only 1024 bytes in width. The uDMAChannelTransferSet() function restarts the transferring of data into the next 1024 blocks in g_ulADCValues.

The else block runs when there is 2048 samples in g_ulADCValues. Since there are enough samples to perform FFT, the sampling and uDMA transfer is turned off until the data can be processed.

Now that everything has been initialized and the ADC interrupt handler has been introduced, the next step is to have an infinite while loop run as seen here:

```
while(1)
{
        if(g_ucDataReady)
        {
                ProcessData();
        }// if

       WidgetMessageQueueProcess();
}// while
```

At various points through the while loop, interrupts will trigger based on the frequency set earlier. For example, before the if statement is processed, a hardware trigger will interrupt 1024 times, making the ADC sample each time. After 1024 times, the ADC interrupt handler will be called. Since there are less than 2048 samples, the sampling will resume. The touchscreen ADC interrupt is called, attempting to trigger the ADC, but fails. Evaluating the if statement, g_ucDataReady is still False because there are not enough samples. It skips over to WidgetMessageQueueProcess() which looks for any touchscreen finger presses. There are none because the touchscreen is broken, so the loop starts again.

The gui updates, but there is nothing to print yet. Another 1024 samples are obtained so the ADC interrupt handler is called again. With 2048 samples, the ADC turns off. The if statement evaluates to True. So ProcessData() runs.

ProcessData() is defined in dsp.c and is presented here:

```
void ProcessData(void){

   // For indexing
   uint32_t i;

   // For finding bin with highest power
   float32_t maxValue;

   for(i=0;i<NUM_SAMPLES;i++)
```

```
    {
            g_fFFTResult[i] = ((float)g_ulADCValues[i] - (float)0x800);// / (float)640;
    }// for

    // Multiply samples by hamming window
    arm_mult_f32(g_fFFTResult, ti_hamming_window_vector, g_fFFTResult, NUM_SAMPLES);

    // Calculate FFT on samples
    arm_rfft_f32(&fftStructure, g_fFFTResult, g_fFFTResult);

    // Calculate complex power of FFT results
    arm_cmplx_mag_f32(g_fFFTResult, g_fFFTResult, NUM_SAMPLES * 2);

    // find the maximum bin
    arm_max_f32(g_fFFTResult, NUM_SAMPLES, &maxValue, &i);

    // Update received frequency
    reFreq = (int)(1.205*g_HzPerBin*i);

    // Update speed
    // Cast integer variables as floats to have more accurate division
    // 24 GHz tranmitted
    speed = (((float)reFreq/(float)g_uiSamplingFreq) - 1.0)*2400000000.0;

    // Clear the data ready bit and set up the next DMA transfer
    g_ucDataReady = 0;

    uDMAChannelTransferSet(UDMA_CHANNEL_ADC3 | UDMA_PRI_SELECT,
                                        UDMA_MODE_BASIC,
                                        (void *)(ADC0_BASE + ADC_O_SSFIFO3 +
                                                (0x20 * UDMA_ARB_1)),
                                        g_ulADCValues, UDMA_XFER_MAX);

    // Enable the timer and start the sampling timer
    uDMAChannelEnable(UDMA_CHANNEL_ADC3);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet()/(g_uiSamplingFreq - 1));
    TimerEnable(TIMER0_BASE, TIMER_A);

}// ProcessData()
```

ProcessData() first subtracts all the sampled data by 0x800, which is approximately 1.5 V based on a scale from 0 to 3.3 V. The data is then multiplied by a hamming window in order to removed the low frequency components. This hamming window is an array of values taken from EuphonistiHack. Afterwards, FFT is performed on the data set. The program then searches for the index with the maximum power. This bin corresponds to the frequency of the input signal. By multiplying with the g_HzPerBin, the input frequency is obtained. For some reason, the frequency obtained is always off by a constant factor. This factor is determined experimentally and is used to correct the result. The speed is calculated from this result. ADC and uDMA is restarted and ProcessData() exits.

Back to the while loop, updateGui() runs and prints the received frequency and the speed onto the screen. And it loops back to the beginning.

Transmitter Code

Compared to the receiver code, the transmitter code is much simpler. The only initialization is two functions, InitSSI() and InitDACTimer().
InitSSI() configures certain pins for SPI, as seen below.

```
void InitSSI()
{
   // for CLK and data
   ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI3);

   // Delay for peripheral to initialize
   SysCtlDelay(3);

   // enable slave select port
   ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

   // Delay for peripheral to initialize
   SysCtlDelay(3);

   // Enable NEW SS pin as GPIO
   GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_1 | GPIO_PIN_2 |
                                                         GPIO_PIN_6);

   // Configure pin to transmit (MOSI)
   GPIOPinConfigure(GPIO_PD3_SSI3TX);

   // Configure pins to be used as SSI Clock and Data
   GPIOPinTypeSSI(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_3);
```

```
    // Initializes SSI
    // Parameters are:     base address of the SSI
    //                              the clock supplied to the SSI
    //                              data frame format
    //                              configure SSI as master as opposed to slave
    //                              the bit rate (should be lower than system clock by at least factor of
    //                                                                                              4)
    //                              word size
    SSIConfigSetExpClk(SSI3_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
                                SSI_MODE_MASTER, SysCtlClockGet()/4, 8);

    // Configure pin as a clock
    GPIOPinConfigure(GPIO_PD0_SSI3CLK);

    // Enable SSI
    ROM_SSIEnable(SSI3_BASE);

   ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, 0);    //SS low
    writeData(0x00); // Send the upper byte
    writeData(0x18);    // Send the lower byte
    ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_6, GPIO_PIN_6;    //SS High

}// InitSSI()
```

The SPI clock is set to transmit at an eighth of the system clock through PD0. The data is set to transmit through PD3 with a work size of one byte. PD1, PD2, and PD6 are all configured as chip select pins. The first two are for the two 16 bit DACs and the last one is for the Infineon. The Infineon only needs to be initialized once. This is done by writing 0x00 and 0x18 to the Infineon.

The initialization of the DAC timer is much like the previous timers and is presented below:

```
void
InitDACTimer(void)
{
    const uint32_t timerFreq = 10000;
    uint32_t ui32Period;           // Determines the cycles

    // Enable the configuration of Timer2
          SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
```

```
        // It loads
        TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC);

   // Divide system clock freq by constant to get # of clock cycles
   // for the timer to count down
   ui32Period = SysCtlClockGet() / timerFreq;

        // Set it to the clock frequency
        TimerLoadSet(TIMER2_BASE, TIMER_A, ui32Period -1);

        // Enable the interrupt associated with this timer
        IntEnable(INT_TIMER2A);

        // Set the interrupt to be called when the timer
        // runs out
        TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

        // Enable the timer
        TimerEnable(TIMER2_BASE, TIMER_A);

}// InitDACTimer()
```

The timerFreq variable can be changed to alter the frequency of the triangle wave. By increasing the frequency of new DAC values, the frequency of the triangle wave should increase as well.

When the program enters the while loop, the receiver code should run. At certain intervals, the DAC interrupt triggers, calling the interrupt handler seen here:

```
void
Timer2AIntHandler(void)
{
        // Clear the timer interrupt.
        TimerIntClear(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

   // If on the rising edge of the wave
   if (edge) {

         // Increment the data bits
         outputValue += step;

         // Check if at the peak or above the wave
```

```
            if (outputValue >= peak) {

                    // Go from rising edge to falling edge
                    edge = !edge;

            }// if
    }// if

    // Else on the falling edge of the wave
    else {

            // Decrement the data bits
            outputValue -= step;

            // Check if at zero or below (below zero is bad)
            if (outputValue <= 0) {

                    // Go from rising edge to falling edge
                    edge = !edge;

            }// if
    }// else

    // Write data
    write16bit();

}// Timer2AIntHandler()
```

As explained earlier, the first part of the code checks whether the program should printing out values for the rising or falling part of the triangle wave. Depending on whether the program is on the rising or falling edge of the wave, the outputValue is either incremented or decremented. The there is some logic to check whether the program is at the peak or trough of the wave. If the outputValue is above 2^(N bits)-1 or below zero, the program switches to either the falling or rising edge respectively.

The last part of the interrupt handler is either write12bit() or write16bit(), depending on the which DAC is being used. If the 12 bit DAC is being used, then the code becomes something like this:

```
 void write12bit(void)
{
   ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, 0);    //SS low

   // 12 bit DAC
   data = highByte(outputValue);       // Take the upper byte
   data = 0x0F & data;
   data = 0x30 | data;
   writeData(data); // Send the upper byte

   data = lowByte(outputValue);        // Shift in the 8 lower bits
   writeData(data);    // Send the lower byte

   ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, GPIO_PIN_1);    //SS High

}// write12bit()
```

The 12 bit DAC has 4 control bits and 12 bits of data. The control bits are always 0011 and the data bits depends on the outputValue. The CS is set low to write and set high when finished. This particular piece of code only shows setting one DAC at a time. In order to test the Infineon, we set the other input of the VCO to Vcc for convenience.

Writing to the 16 bit DAC is similar as seen here.

```
void write16bit(void)
{
   ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, 0);    //SS low
   writeData(0x00); // Control bits
   writeData(outputValue); // Send the upper byte
     writeData(outputValue);    // Send the lower byte
   ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, GPIO_PIN_1);    //SS High

}// write16bit()
```

Instead of having 12 data bits, there are 16 data bits. There are two control bits that are supposed to be 00 and six don’t care bits, so the 0x00 is written in first.

Problems with the Code

As said earlier, there are a few problems with the Tiva code. One of them is the software trigger for the ADC not working. While the processor should be able to call the ADC as long as IntMasterEnable() is called, the ADC for the touchscreen does not work. Since we only have one working ADC that is hardware triggered, we reserved it for the sampling.

Another problem is the DAC code and the GUI code interfering with each other. While the Kentec display is active, the SPI stops working. It seems that as long as the initialization function for the Kentec screen runs, the DAC stops outputting a voltage.

Antenna

The patch design

In our quarter one design, we use coffee can as our antenna. The cans worked great, but they were oversized and impractical for 2.4 Ghz. We chose to do a patch antenna because it can be directly printed onto the PCB with the Infineon chip. A microstrip antenna is fed by a microstrip transmission line. It is made with high conductivity material. The patch is of length L, width W, and sitting on top of a substrate of thickness h with permittivity e. . In our case, we use the RO4350B substrate material.

The center frequency can be given by

$$f_c \approx \frac{c}{2L\sqrt{\varepsilon_r}} = \frac{1}{2L\sqrt{\varepsilon_0 \varepsilon_r \mu_0}}$$

$$\varepsilon_{reff} = \frac{\varepsilon_r + 1}{2} + \frac{\varepsilon_r - 1}{2}\left[1 + 12\frac{h}{W}\right]^{-1/2} \quad (14\text{-}1)$$

The RO4350B has dielectric constant 3.48+/- 0.05, z direction, 10 Ghz/23 Celsius.

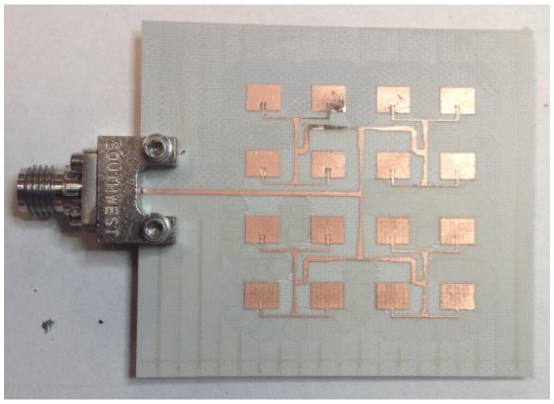
The width W of the miscrostip antenna controls the input impedance. Larger widths also can increase the bandwidth. L is the resonant dimension. The width W is usually chosen to be larger than L. W=1.5 L is typical. The advantages of using a patch antenna are low profile, easy to fabricate, feed and use and patterns are somewhat hemispherical.

Feed-in System

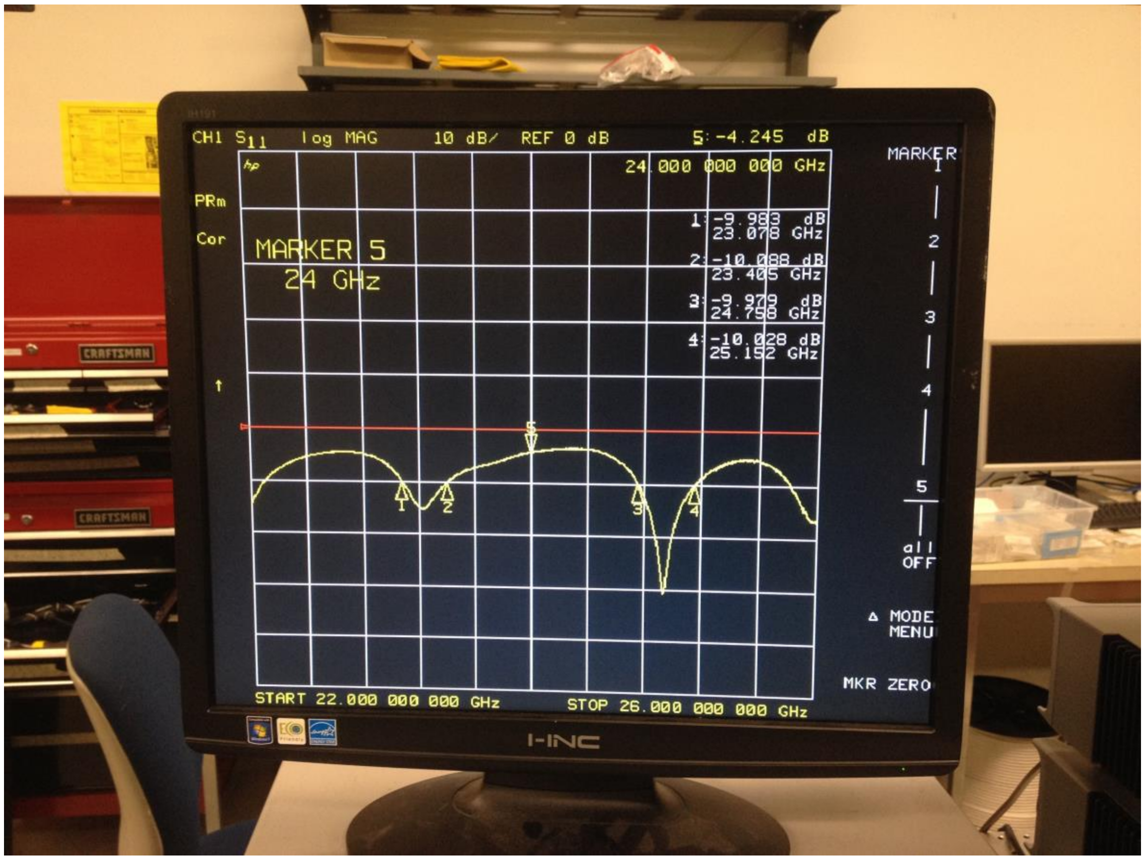
The patch antenna array uses a series of T-junction power dividers. These power dividers are three port networks. For the transmitting antenna, the input power is divided in in half to each of the other two output ports. For the receiving antenna, the input signal from each input arm is transferred to the output port. Since the power can be split in half or added together, the T-junction power divider is reciprocal. Since three port networks cannot be matched, reciprocal, and lossless at the same time, this power is lossy.

Test/Measurement Result

24 Ghz Antenna

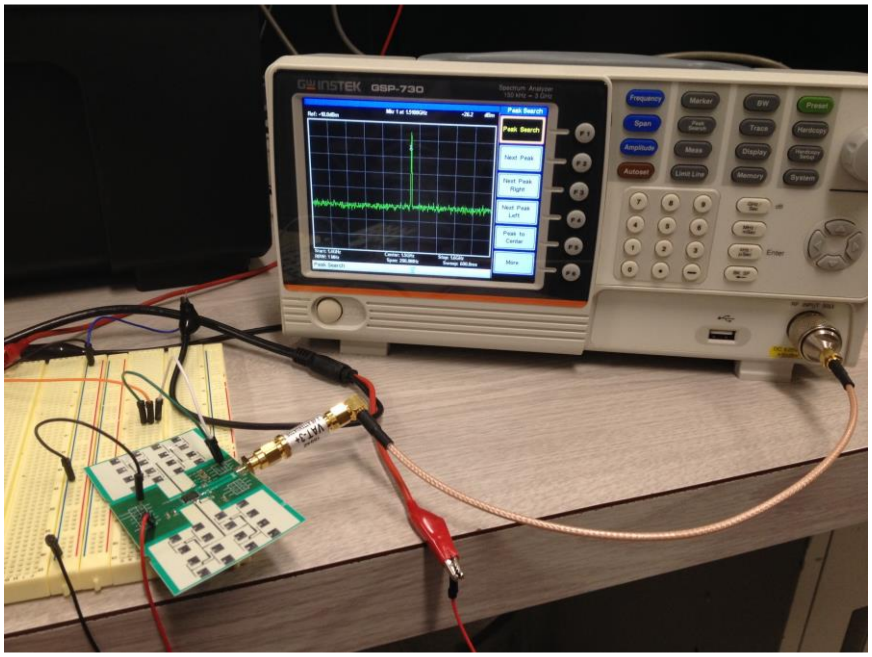


We modified Hao’s design to come up with this antenna. We used Hao’s feed system for impedance matching which saved us a lot of time for designing the antenna. We used graphical tools to round out all the edges and resized each patches. In our HFSS, we observed a high efficiency operation at 25 Ghz. The stimulation is varied by the actual graph below.



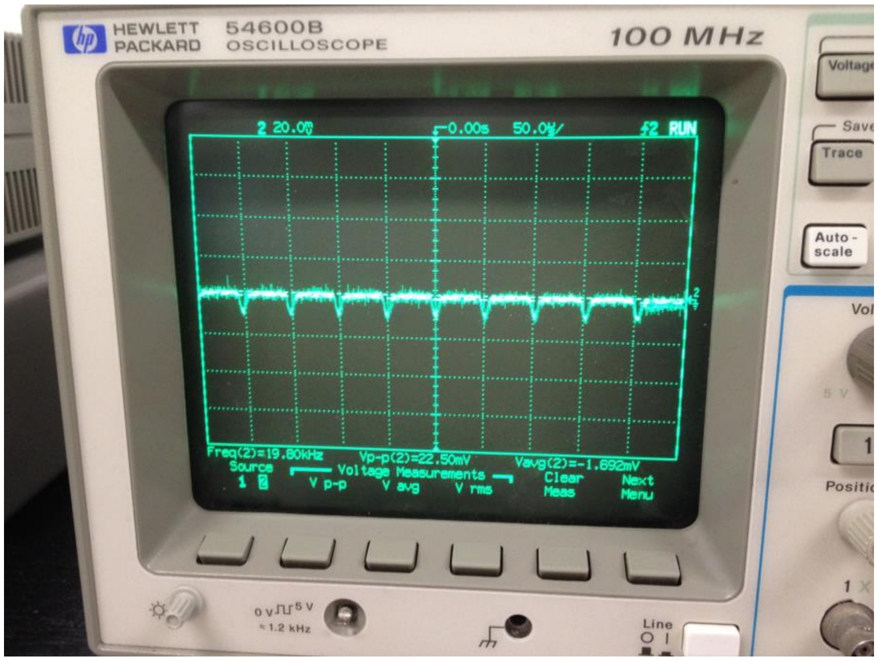
The antenna has an operating frequency at around 25 Ghz. S11 is the reflected power radio 1 trying to deliver to antenna 1. Thus, we have a reflection coefficient or return loss. Our antenna has S11 = -30 dB at 25 Ghz. This implies that if 3 dB of power is delivered to the antenna, -27 dB is the reflected power. In this case, it will be very low so majority of the power is radiated.

BGT11MTR24 LO output



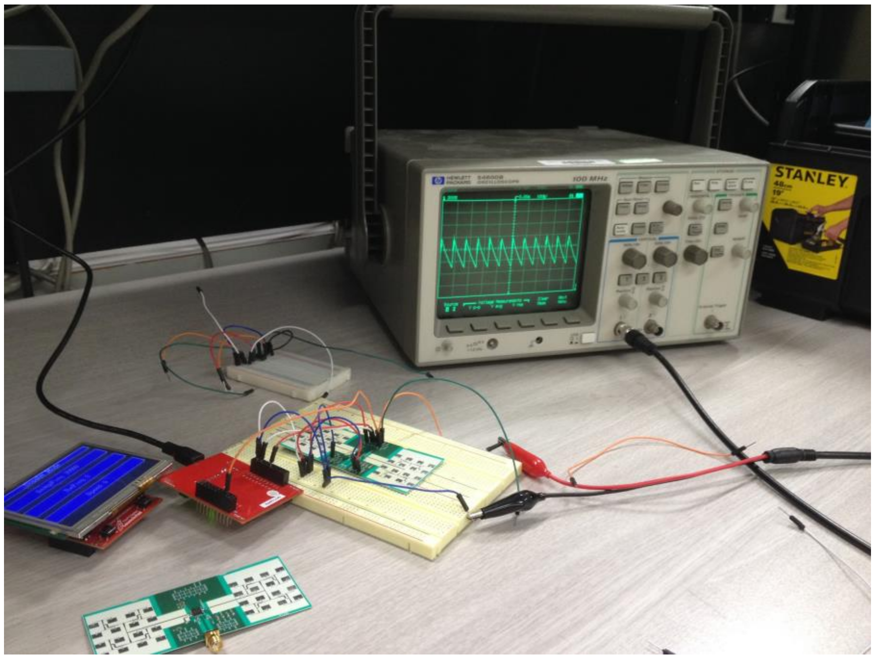
We tested the LO using the SMA connector to the Spectrometer. We observed a strong signal radiating out at the transmitter. The signal supposed to be 24 Ghz, but for testing purpose the chip puts it down to 1.5 Ghz.

BGT24MTR11 Q2 Output



```
Freq(2)=19.80kHz   Vp-p(2)=22.50mV   Vavg(2)=-1.692mV
```

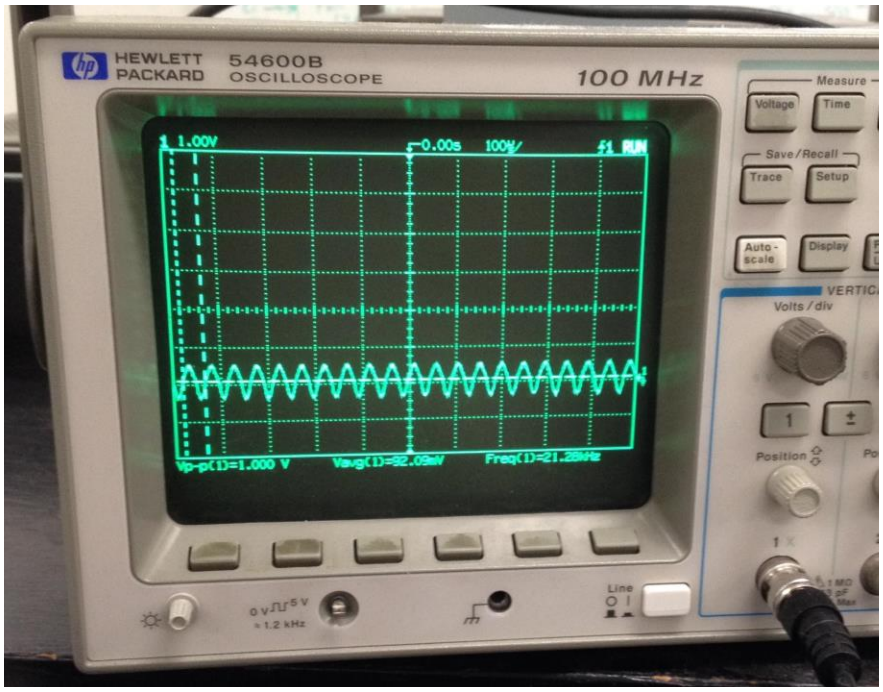
The Q2 is the testing pin for the VCO. Inside the transceiver, the original VCO frequency is divided by 16 then by 65536 to produce a 22.8kHz signal at 24Ghz. We see a 19 kHz output, which gives us a roughly 22 Ghz. We can increase the voltage for the VCO to generate a higher frequency. Nevertheless, Q2 indicates functional VCO.



BGT24MTR11 IFI, IFIX, IFQ,IFQX Output

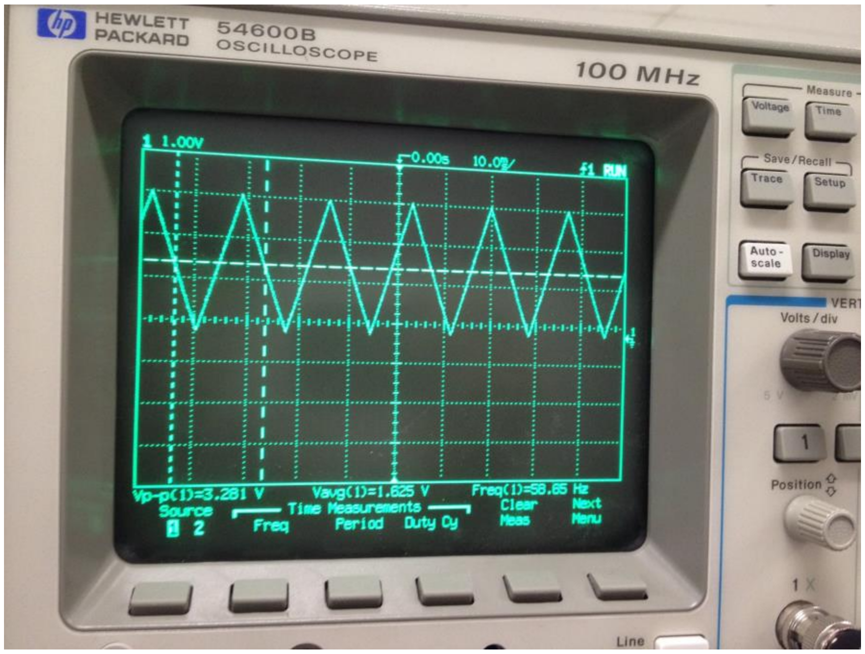
From the output of the system, we observed a saw-wave. We are not sure what causes the problem but this waveform is not what our program is designed to analyze. The wave has an amplitude of roughly 500 mV, which indicates that it is not just noise. We would expect a sinusoidal wave, but the system is not perfect. With more time and processing tools, we might still calculate the speed or the doppler's effect from these waves.

Low Pass Filter and Amplifier



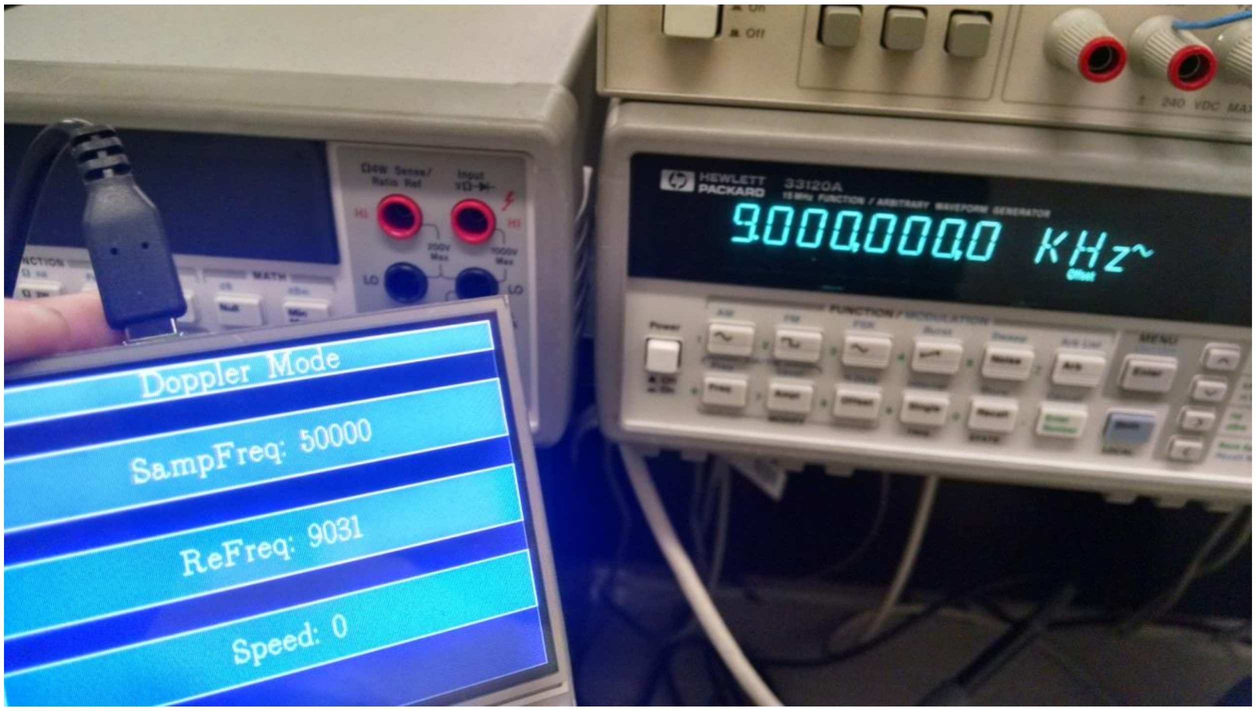
We set the function generator to 22 kHz, which act as the real signal that comes from the antenna. This signal is feed into the low pass filter which has a cut off frequency at 23 kHz. After the low pass filter, it gets amplified to 1 Vpp. Then, it returns to the TIVA ADC for processing.

DAC



We programmed the DAC to generated a clean triangle signal for the range testing. The Vpp is 3.28 V, which can activate the VCO to generate a frequency from to 21- 25 Ghz.

Frequency Measurements



The Tiva successful measures a signal of a given frequency. The frequency has a certain amount of error in it, as seen in the picture. The speed does not seem to update correctly. The following table shows the calculated frequency compared to the input and sampling frequency. All measurements are done with a function generator with 0.8 Vpp and 1.5 V offset. The 1.5 V offset is important because the DSP assumes a 1.5 V offset.

Input Frequency	Sampling Frequency		
	50 Hz	5 kHz	50 kHz
1 Hz	1	0 Hz	0 Hz
10 Hz	10	8	29
100 Hz	N/A	100	88
1 kHz		1003	1000
2 kHz		N/A	2000
3 kHz			3000
4 kHz			4030
5 kHz			5030

6 kHz			6030
7 kHz			7031
8 kHz			8031
9 kHz			9031
10 kHz			10061
11 kHz			11061
12 kHz			12061
13 kHz			13062
14 kHz			13062
15 kHz			15062
16 kHz			N/A

Instead of needing twice the sampling frequency to measure a given input frequency, about three times the input frequency is needed. This could be due to noise.

Possibility for future extension

Low Pass Filter Design

The low pass filter designated after receiving the signal, needs to be linear phase. It should be linear phase primarily because we want all the frequency component of our signal to get shifted the same amount. Otherwise we would get a different type of signal at different frequency than the actual one at the output of the low pass filter. This case would be called as phase distortion or delay distortion. The derivative of the phase response of the filter which is called the group delay of the filter represents the amount of delay a signal would face after entering into the filter. If our filter has linear phase, its group delay would be constant, which means all the frequency component of our signal would have same amount of delay. In order to have the linear phase filter, we need to select a FIR filter. FIR filter stands for finite impulse response. In order to have the FIR filter to be linear phase, its coefficients needs to be symmetric. In our design, we eventually decided to use MAX 291 chip as our low pass filter since we were allowed to use ready chips. One other possible case for our low pass filter design would be to program a FIR filter where its coefficients are evenly symmetrical around a center.

Suggestions

Using competition to decide the outcome of student’s grade may not be a very good idea. Engineering students are just exposed to reality projects, and they may not have the foresight to sense the difficulty of a project that is out of their boundary. The nature of competition requests students to build “the best” radar system. Building a system that is win worthy requires more than undergraduate knowledge. This is not feasible due to the fact that most senior students do not have the extra time to pour their full passion into the project. As a result, it leads to a situation which students dream too big, and not enough time to realize their goals. Our team tried what we could, and even worked with a couple other team on the same chip. We designed the PCB and wrote the code, but the system with on board DSP is prompt to more problems than results. We can not fulfill our project objective, which is to create a functional system.

A suggestion for next year is to have an advanced radar system in mind, and ask students to build it, and improve upon it. For example, our 24 Ghz system could be a very good practice. It will take ten weeks for students to finish. If the entire class can use the same chip, students can discuss with their classmates more often to facilitate better result. Our code and PCB can serve as the basis of their system implementation.

Conclusion

Although we were not able to successfully to create a working radar system, we were able to get many of the individual parts working. In our struggles we learned a lot of RF system design.