EEC 134 Application Note
Tiva Programming

Patrick Huynh

Introduction

This application note will cover Tiva programming for radar signal processing. The intended audience are students that have a bit of background in wireless systems and C programming, but not necessarily much knowledge in embedded systems.

There will be two parts to this application note. Because it was so difficult to figure out how to properly learn Tiva programming, the first part will be a meta-tutorial on how to learn programming for the Tiva. It will not necessarily go in-depth, step-by-step how to use Code Composer Studio. Rather, it will provide an outline of useful resources on how to learn Tiva programming. In addition, some less known, but useful features of Code Code Composer Studio will be demonstrated.

The second part of this application note will go over the code for the radar. This section is the same as the programming section of the Team Hero Group Report.

Tiva Programming Meta-Tutorial

Starting out

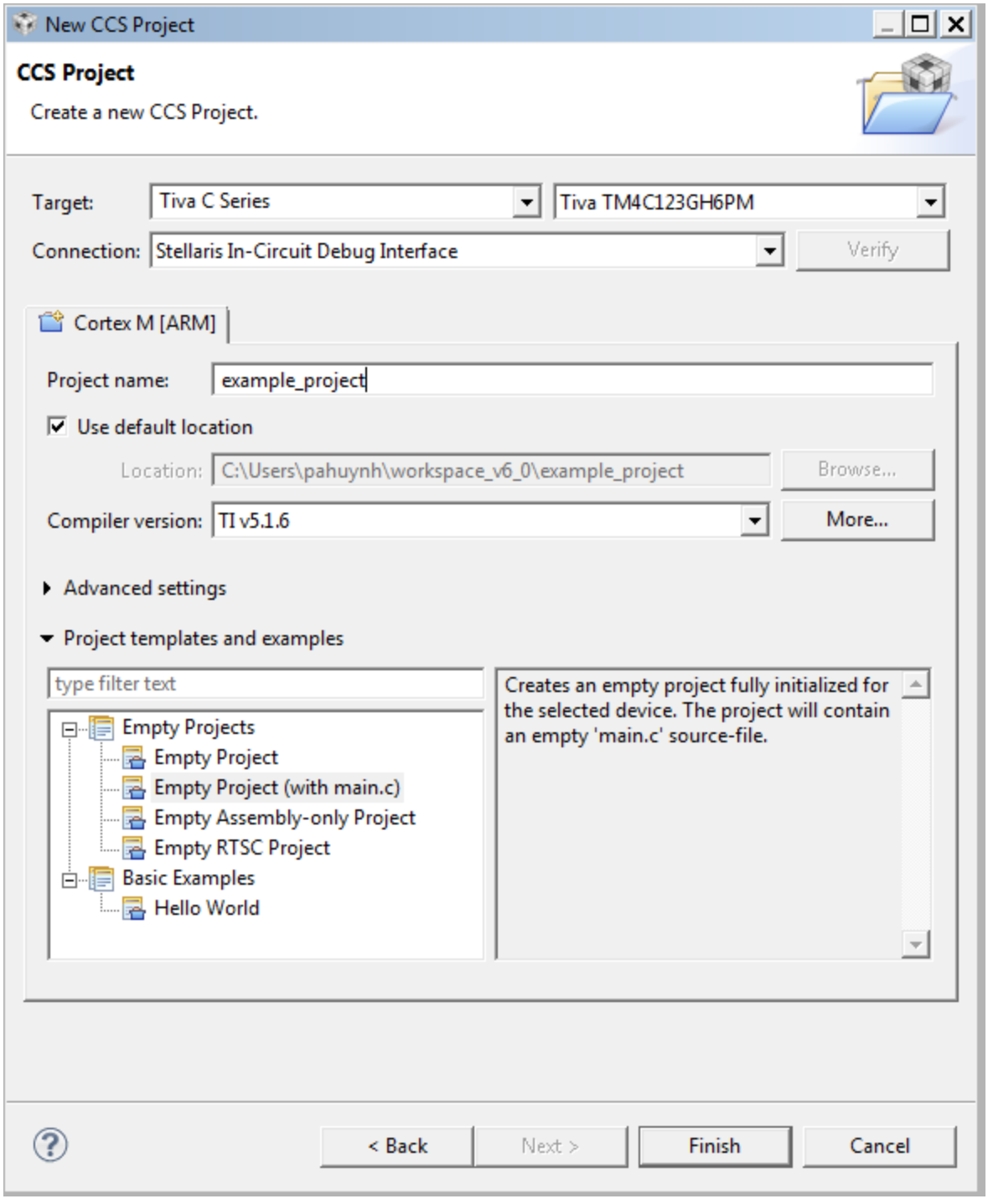
In order to use the Tiva, special software is needed to program the Tiva. There are several different types of software, with Energia being the easiest, Kiel being of medium difficulty, and Code Composer Studio having the steepest learning curve. That being said, this tutorial will assume that you use Code Composer Studio. The reason is that there is a lot of resources for programming with Code Composer Studio. There is also some useful features related to signal processing that only Code Composer Studio can perform.

The first step to learning how to use Code Composer Studio and Tiva Programming in general is to read the Tiva Workshop Student Guide. Lab 0 will cover how to go over how to install Code Composer Studio and obtain the Tivaware library, which is a contains many useful functions for programming the Tiva. Tivaware is very important; without it, programming the Tiva would be doing direct register accesses. The reader should then go over labs 1-5, 8-11, and 13. This goes over the basics of Tiva programming such as how interrupts work or how to set up a uDMA control table.

While going through the workshop, make sure to consult the Tivaware Device Peripheral Library User Guide as well as the Tivaware Graphics Library User Guide. If there is some function that does not quite make sense, it is documented in one of these user guides. If for some reason, the reader needs to do a direct register access in order to program the Tiva, consult the Tiva datasheet. It has useful documentation on GPIO masking works or the specific pin that an ADC channel corresponds to.

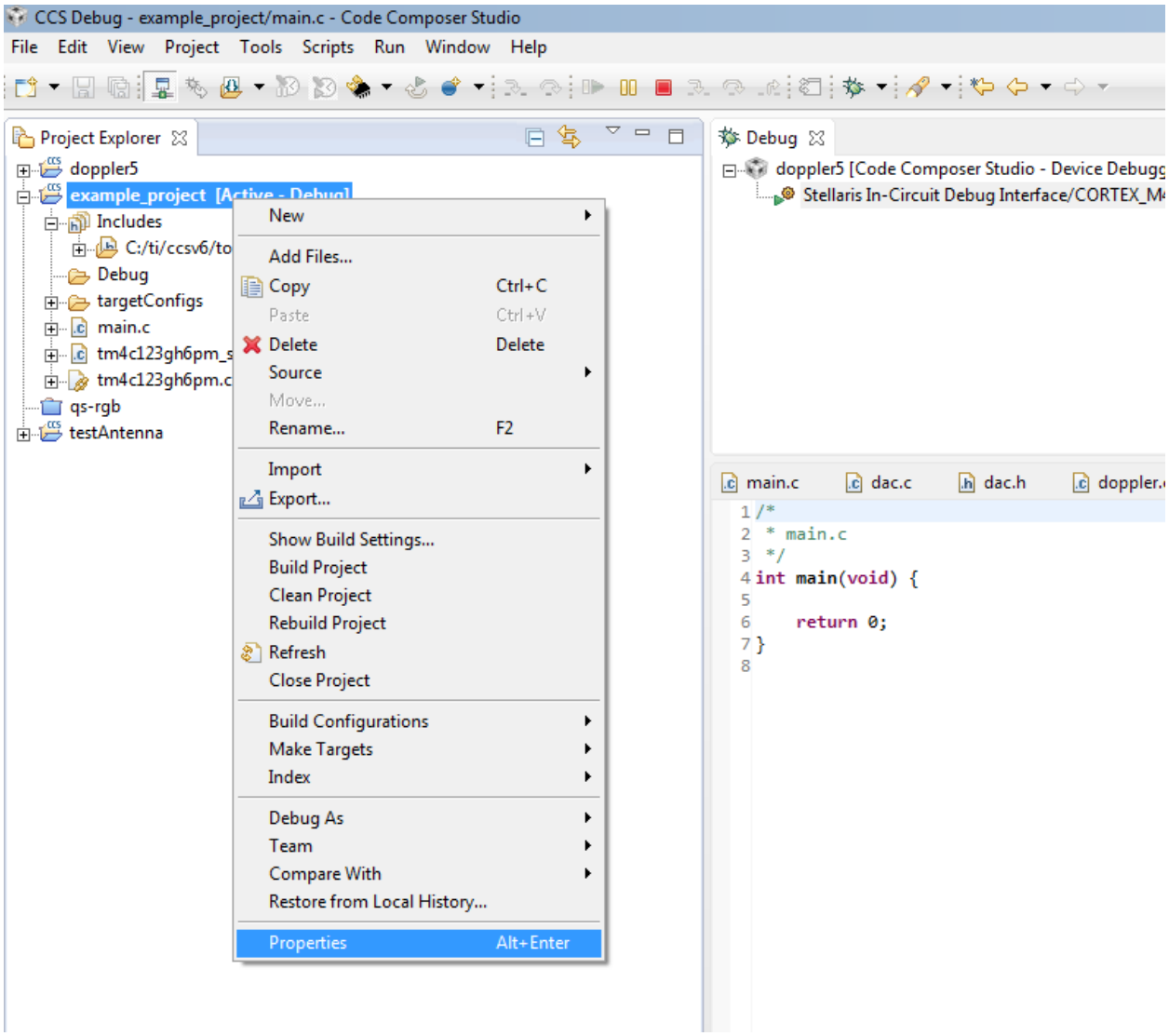
After going through the tutorials, the reader is ready to program the Tiva. Before immediately going over some code, here are some hints on using Code Composer Studio. These examples are either common problems that seem to occur while programming the Tiva or some less known features.

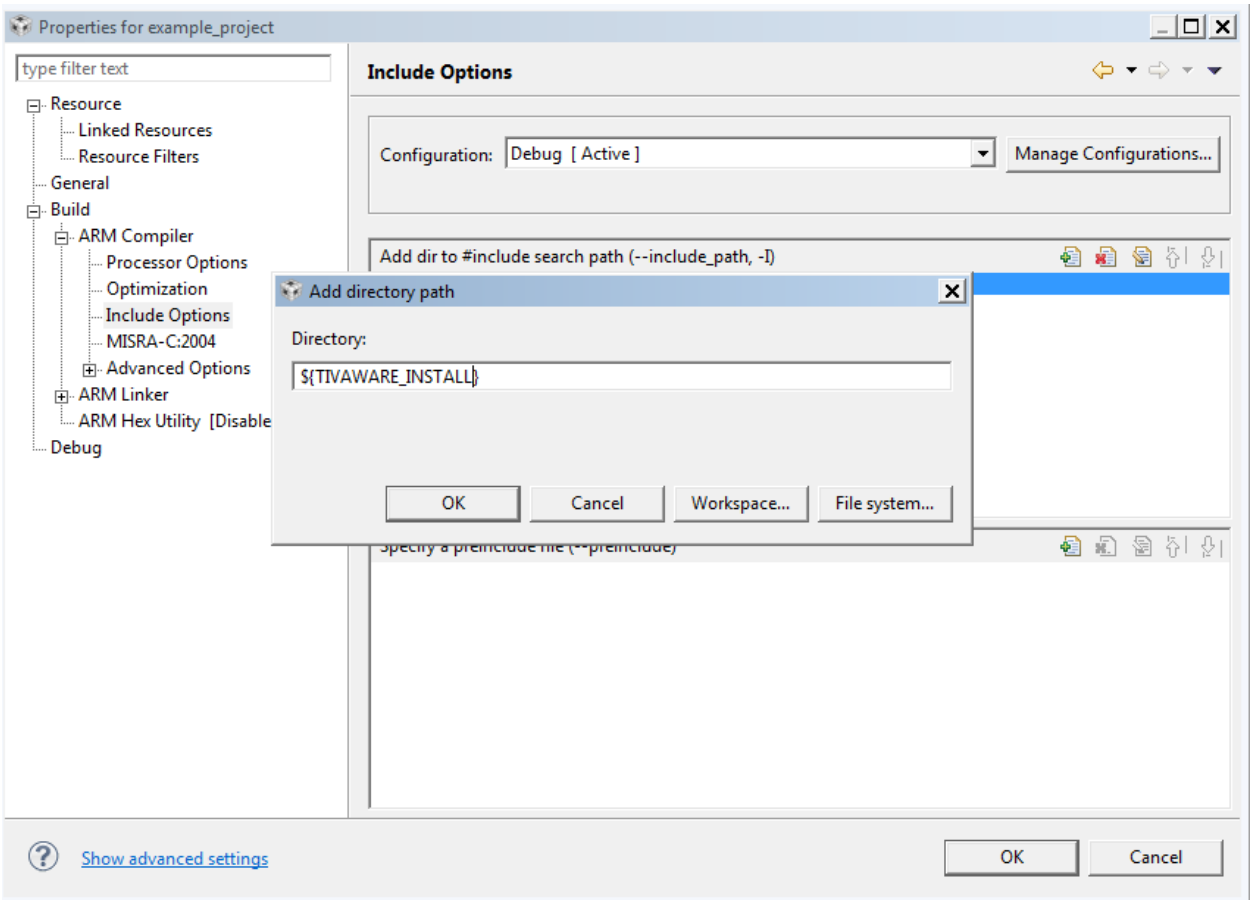
Making a New Project



When making a new project, make sure to define the Target as Tiva and the Connection as Stellaris In-Circuit Debug Interface. Without this, there is no target config file in the project

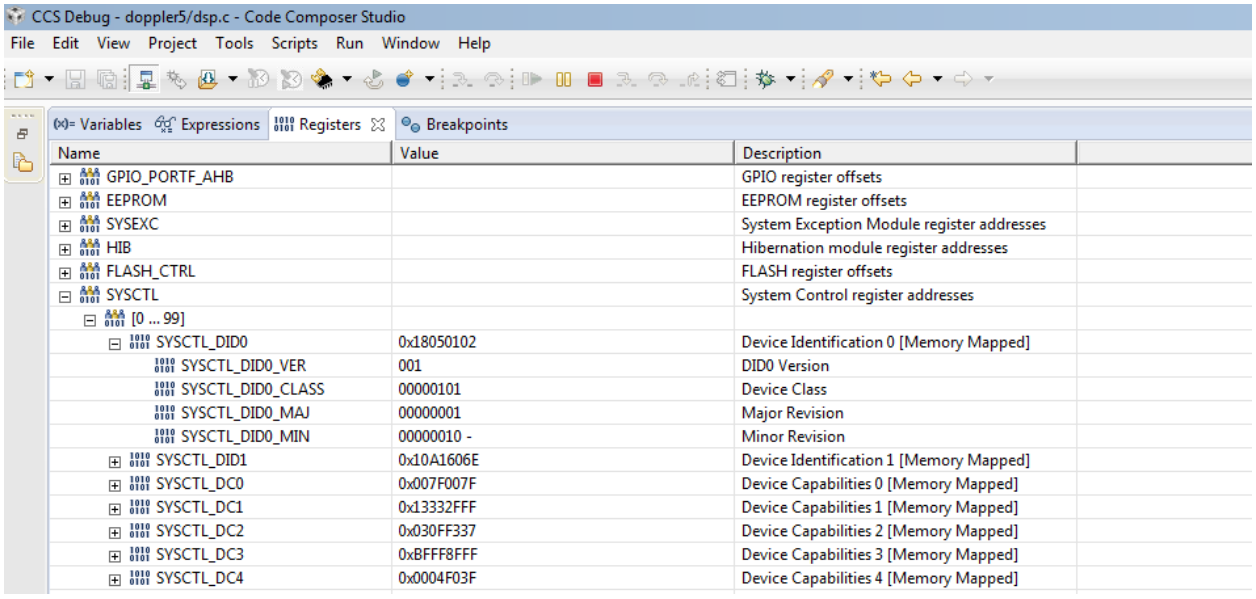
directory and the program will not compile. While this seems very simple, this problem came up many times with multiple people.





Also make sure that you know how to properly link the Tivaware library to the project and include it in the build options. While programming the Tiva, you will need to use functions from either the Driver library or the Graphics library. Without linking or including it in the build options, the program will not compile. Reread lab 1 and 2 of the Tiva Workshop Student Guide if necessary.

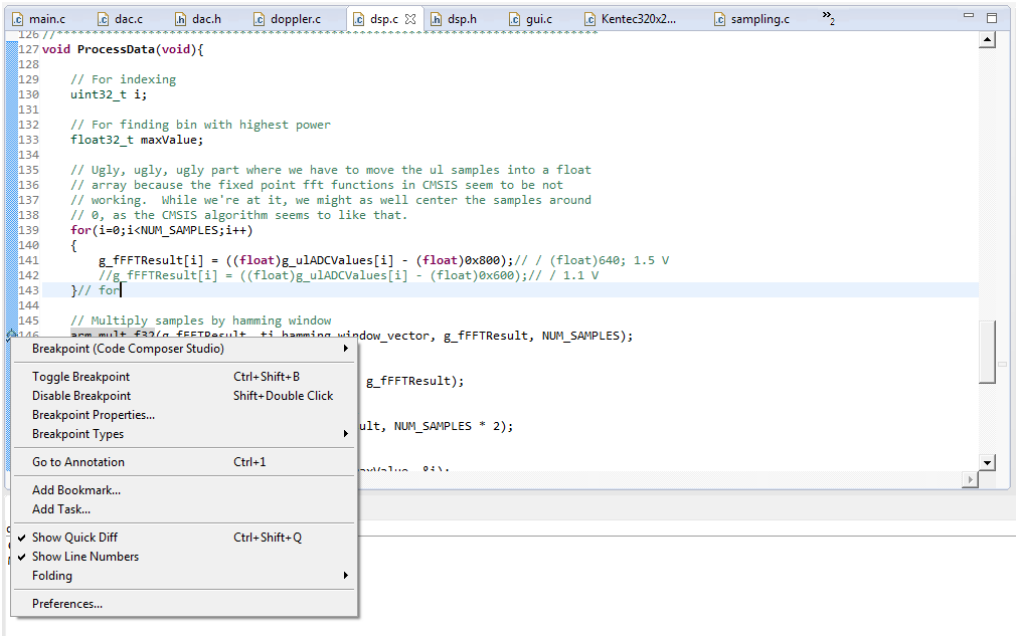
Tiva Model Number

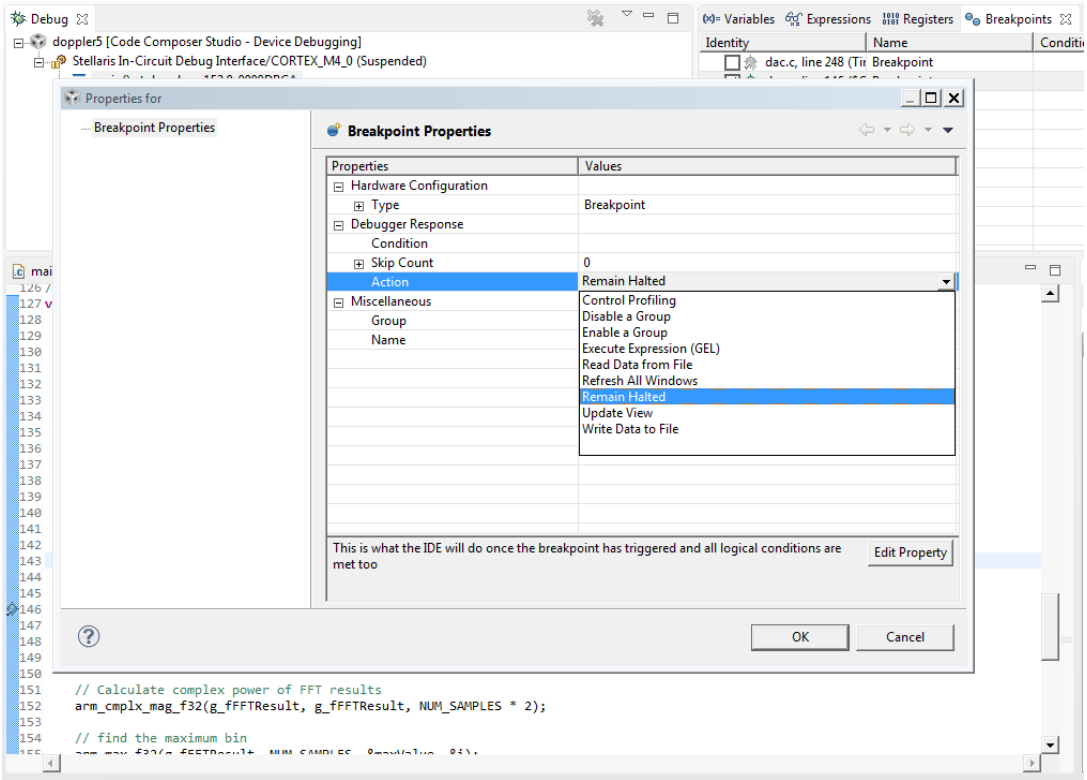


At some point while working with the Tiva, you may need to use ROM functions instead of the normal versions of the function in order to save memory space. The problem with using these functions is that in order to use them, you need to define the version of the Tiva in use. This is because different versions of the Tiva have their functions located in different parts of the ROM. So instead of doing *#define TARGET_IS_BLIZZARD_RA1* (Blizzard is nickname for the Tiva), you would need to use *#define TARGET_IS_BLIZZARD_RB2*. You can check the model by checking the certain registers while the program is in debug mode. The information is found in SYSCTL_DIC0_MAJ and SYSCTL_DID0_MIN. You can check the silicon revision guide for specific details on which numbers correspond to which model.

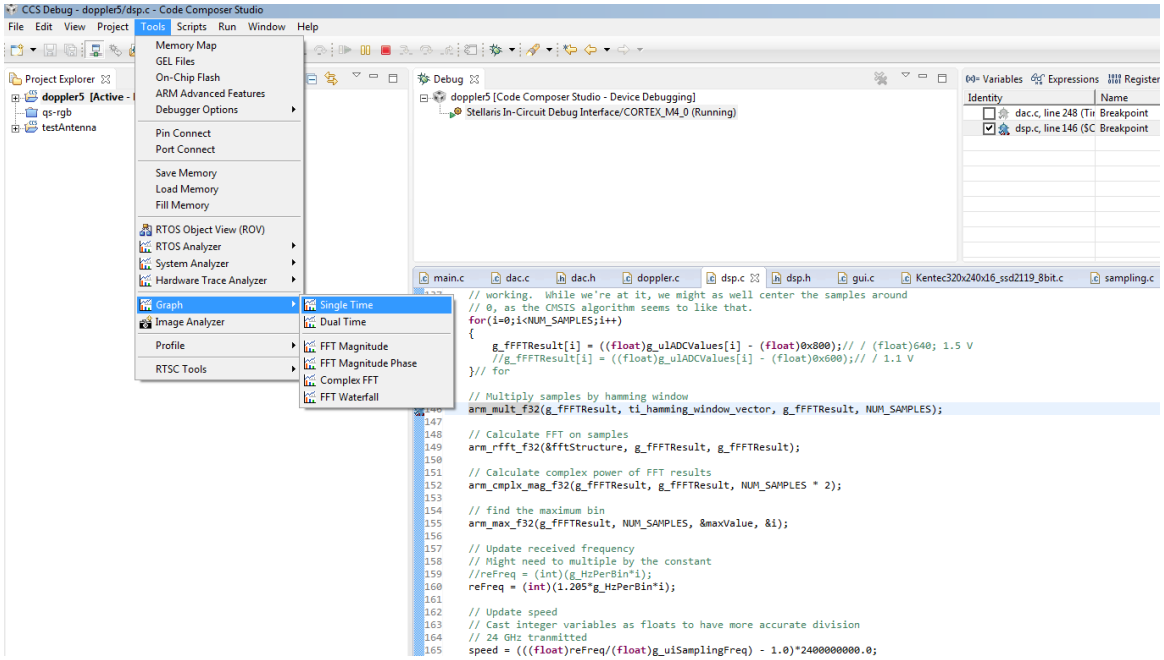
How to Display a Signal on Code Composer Studio

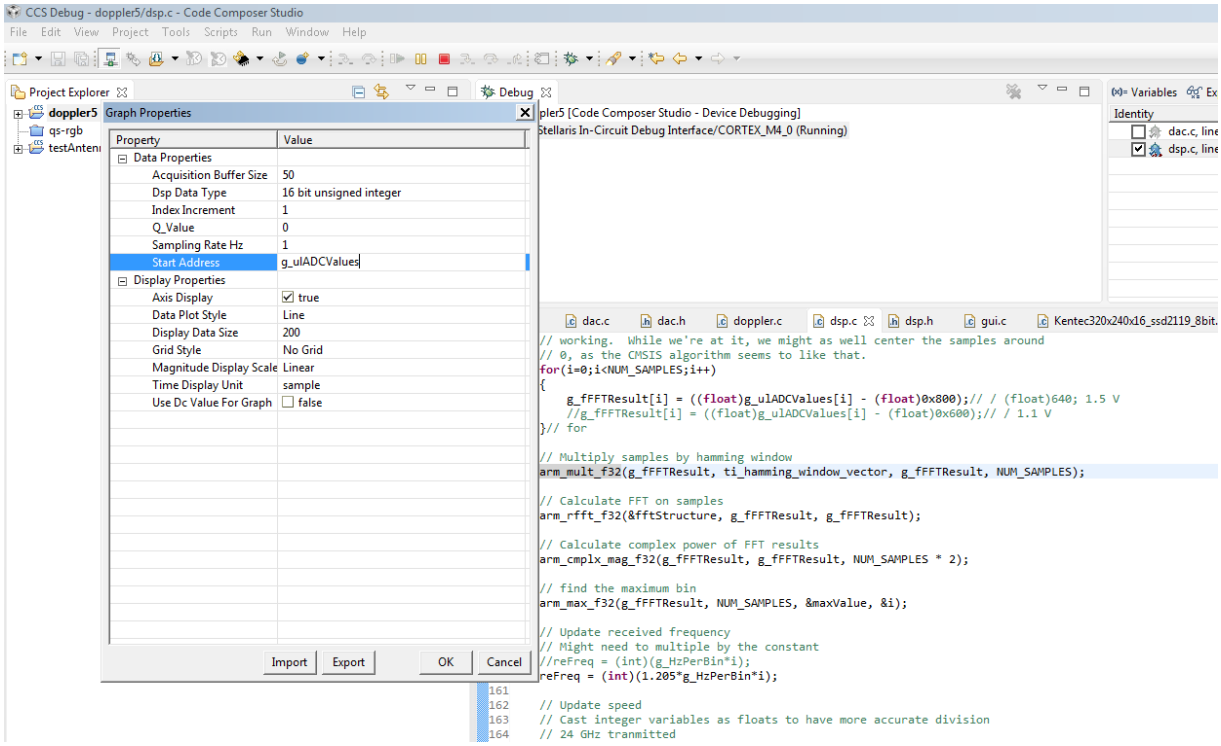
The radar program will be sampling a signal and putting the resulting data in an array. Using Code Composer Studio, it is possible to takes values from this array and display the signal in the time or frequency domain.

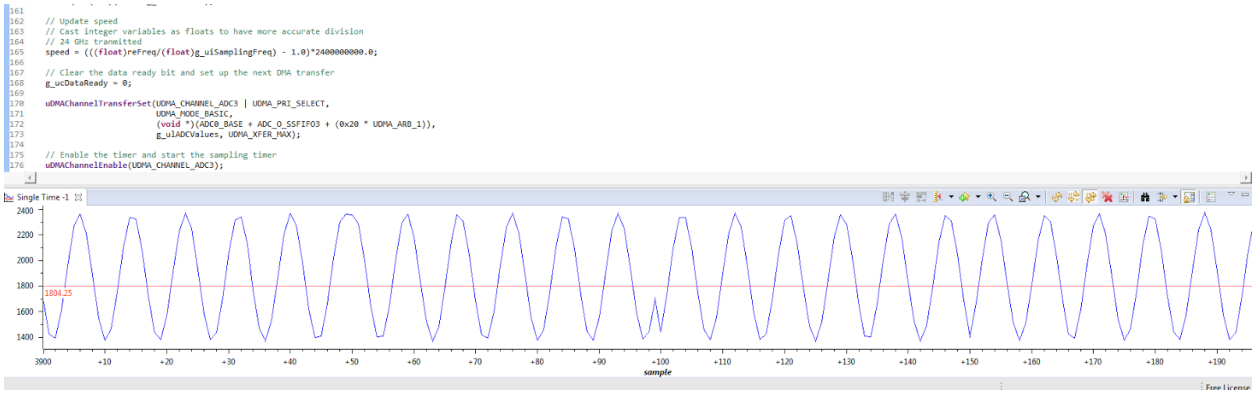




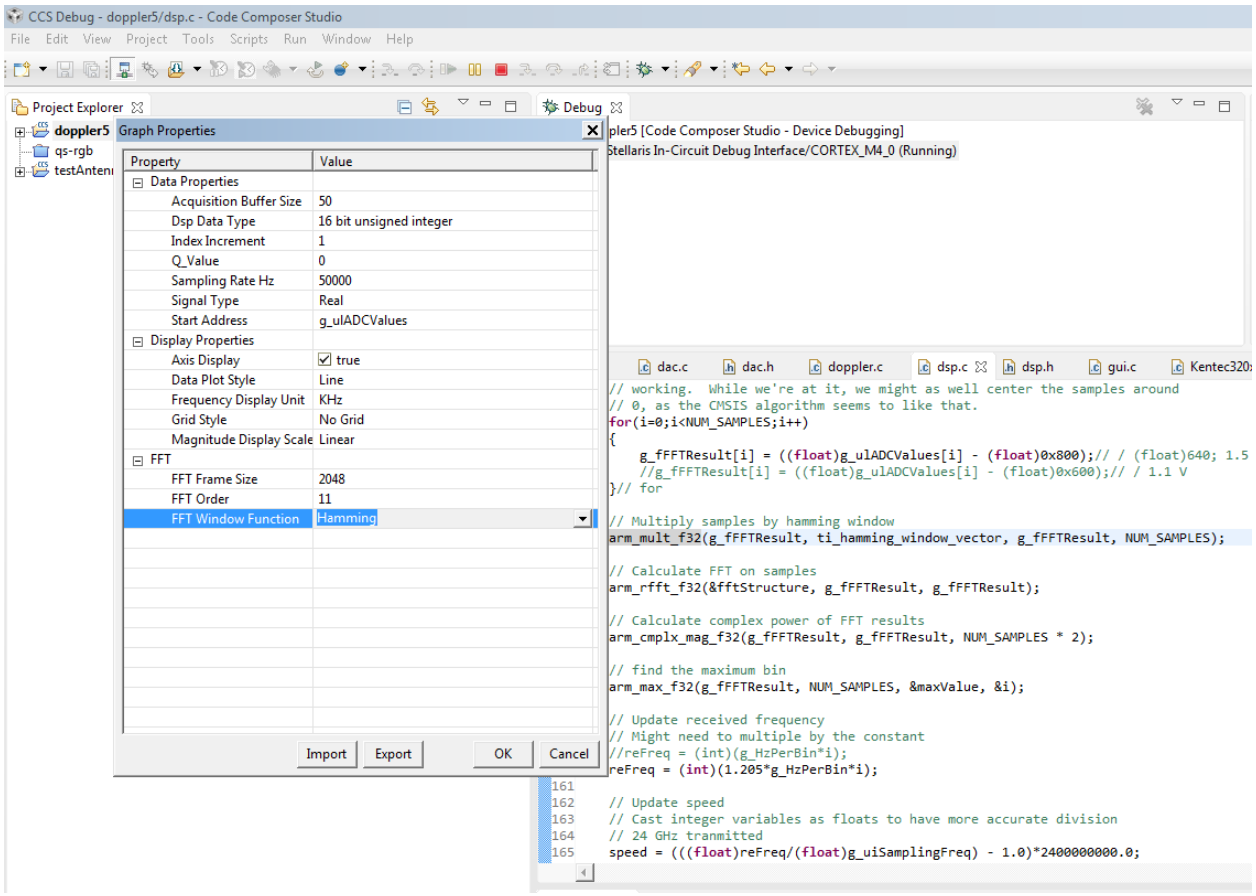
First set a breakpoint on some line. Go to Breakpoint Properties and change the Action from Remain Halted to Refresh All Windows. Run the program in debug mode.

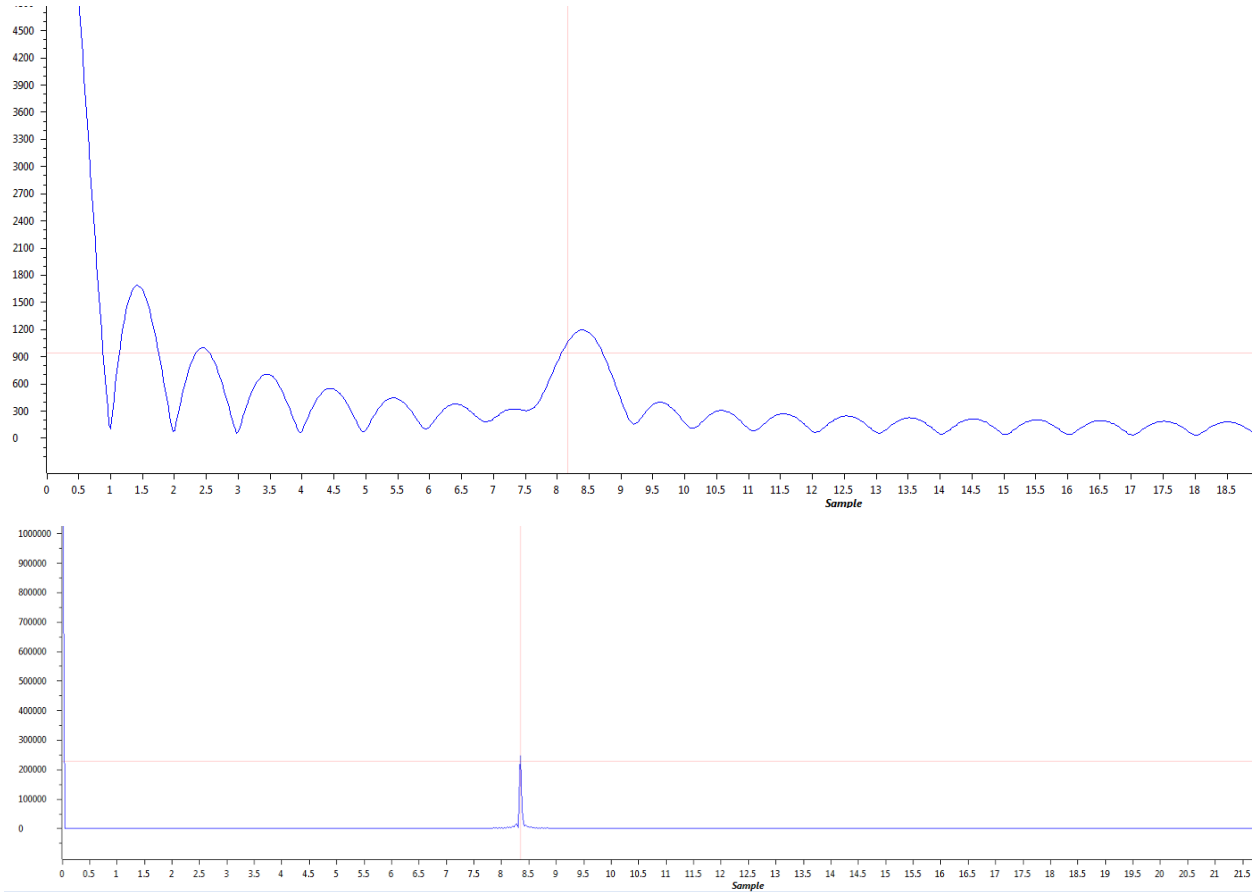

CCS Debug - doppler5/dsp.c - Code Composer Studio
File Edit View Project Tools Scripts Run Window Help
Memory Map
GEL Files
On-Chip Flash
ARM Advanced Features
Debugger Options
Pin Connect
Port Connect
Save Memory
Load Memory
Fill Memory
RTOS Object View (ROV)
RTOS Analyzer
System Analyzer
Hardware Trace Analyzer
Graph
Image Analyzer
Profile
RTSC Tools
Single Time
Dual Time
FFT Magnitude
FFT Magnitude Phase
Complex FFT
FFT Waterfall
Project Explorer
doppler5 [Active -
qs-rgb
testAntenna
Debug
doppler5 [Code Composer Studio - Device Debugging]
Stellaris In-Circuit Debug Interface/CORTEX_M4_0 (Running)
Variables
Expressions
Identity
Name
dac.c, line 248 (Tir Breakpoint
dsp.c, line 146 ($C Breakpoint
main.c
dac.c
dac.h
doppler.c
dsp.c
dsp.h
gui.c
Kentec320x240x16_ssd2119_8bit.c
sampling.c
// working. While we're at it, we might as well center the samples around
// 0, as the CMSIS algorithm seems to like that.
for(i=0;i<NUM_SAMPLES;i++)
{
g_fFFTResult[i] = ((float)g_ulADCValues[i] - (float)0x800);// / (float)640; 1.5 V
//g_fFFTResult[i] = ((float)g_ulADCValues[i] - (float)0x600);// / 1.1 V
}// 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
// Might need to multiple by the constant
//reFreq = (int)(g_HzPerBin*i);
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;


CCS Debug - doppler5/dsp.c - Code Composer Studio
File Edit View Project Tools Scripts Run Window Help
Project Explorer
doppler5
qs-rgb
testAnteni
Graph Properties
Property
Value
Data Properties
Acquisition Buffer Size 50
Dsp Data Type 16 bit unsigned integer
Index Increment 1
Q_Value 0
Sampling Rate Hz 1
Start Address g_ulADCValues
Display Properties
Axis Display true
Data Plot Style Line
Display Data Size 200
Grid Style No Grid
Magnitude Display Scale Linear
Time Display Unit sample
Use Dc Value For Graph false
Import
Export
OK
Cancel
Debug
pler5 [Code Composer Studio - Device Debugging]
Stellaris In-Circuit Debug Interface/CORTEX_M4_0 (Running)
Variables
Identity
dac.c, line
dsp.c, line
dac.c
dac.h
doppler.c
dsp.c
dsp.h
gui.c
Kentec320x240x16_ssd2119_8bit.
// working. While we're at it, we might as well center the samples around
// 0, as the CMSIS algorithm seems to like that.
for(i=0;i<NUM_SAMPLES;i++)
{
g_fFFTResult[i] = ((float)g_ulADCValues[i] - (float)0x800);// / (float)640; 1.5 V
//g_fFFTResult[i] = ((float)g_ulADCValues[i] - (float)0x600);// / 1.1 V
}// 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
// Might need to multiple by the constant
//reFreq = (int)(g_HzPerBin*i);
reFreq = (int)(1.205*g_HzPerBin*i);
// Update speed
// Cast integer variables as floats to have more accurate division
// 24 GHz tranmitted



Go to Tools, then Graph, and select Single Time. In the Start Address, input the name of the array containing the samples. This works because an array name in C is actually the address of the first element in the array. The data type here is 16 bit unsigned integer because that is what is used in the radar program described later. A sinusoid should appear. The small discontinuity is due to the signal processing which momentarily stops the sampling.

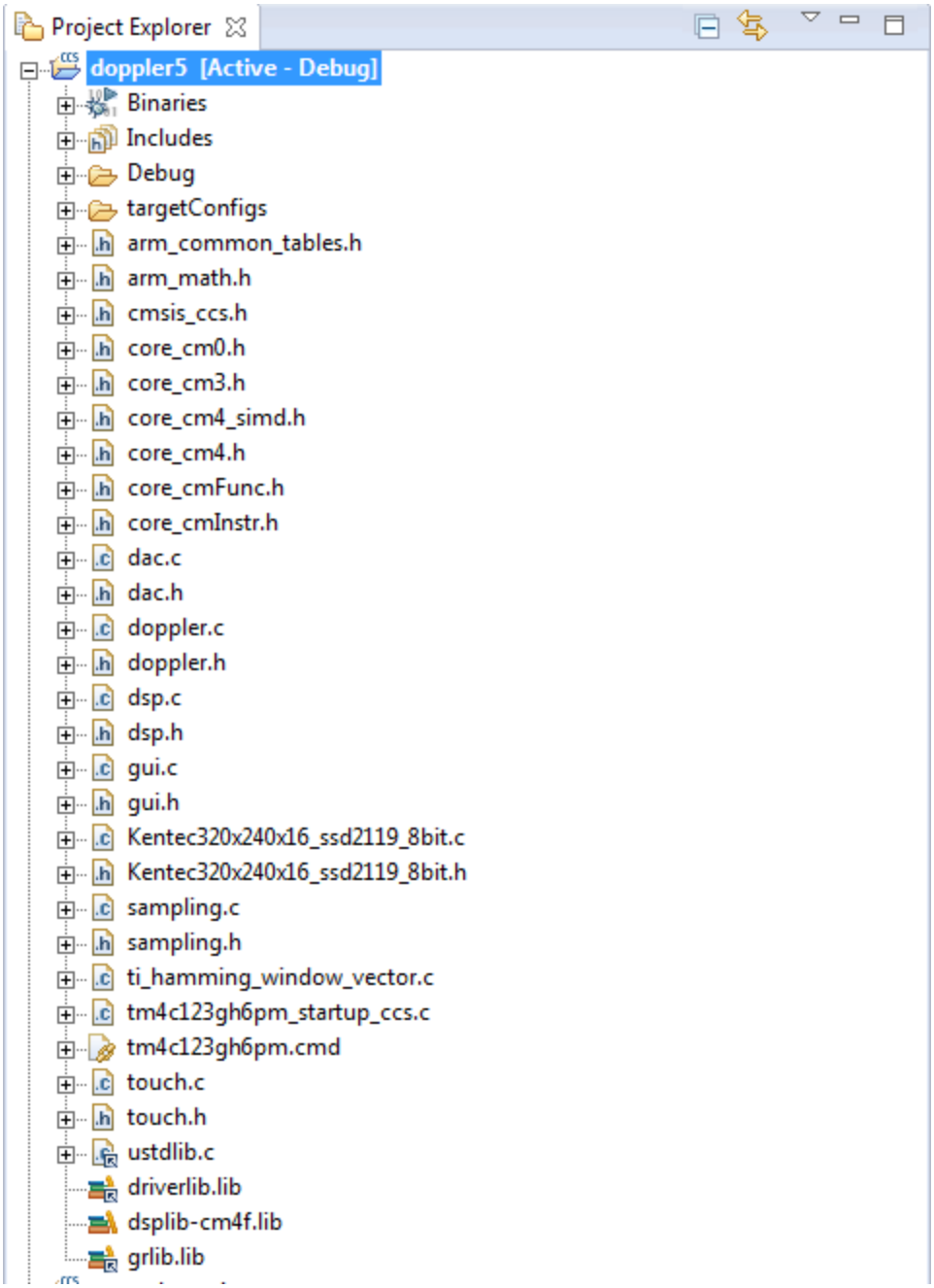




In order to display the frequency domain, go back to Graph and choose FFT Magnitude. Do the same thing as before with the time domain except now make the FFT 2048 points and include a hamming window. The two graphs are of the same signal. The upper graph has an Acquisition buffer size of 50 while the bottom one has a buffer size of 2048. It is interesting to note that the input signal was actually a 10 kHz signal. Due to problems with the Tiva, it shows a lower frequency.

Radar Code

Here is a guide to the radar code written by Team Hero. If you happen to have all the source and headers files and the libraries properly linked, it should look something like this:



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.

Conclusion

Hopefully this application note is of use any student that reads this. While our radar did not work properly, I hope that this application note will help you succeed in your senior design project.