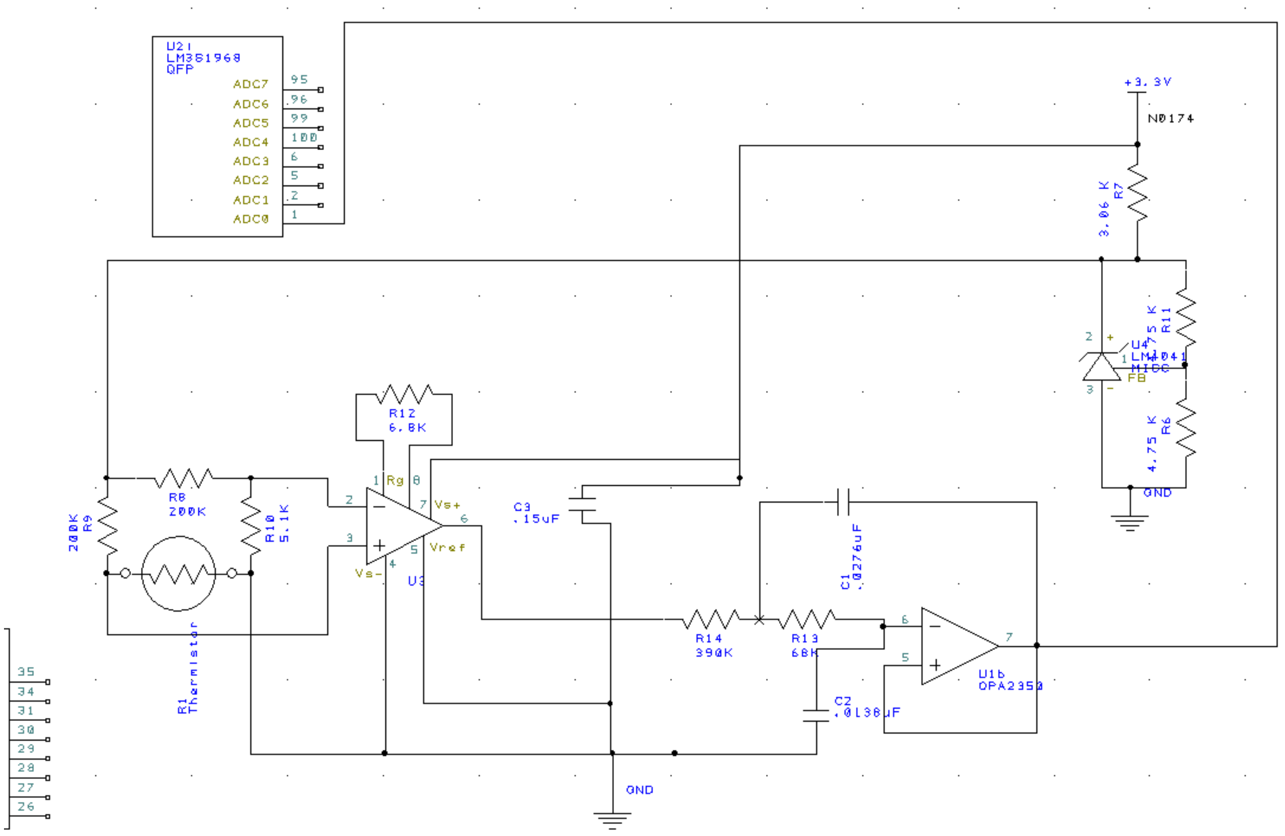
EE445L – Lab9: Temperature Data Acquisition System

Harley Ross and Dalton Altstaetter
4/8/2014

OBJECTIVES

The objectives of this lab are to study ADC conversion, how to use Nyquist Theorem, and the reason for using Valvano Postulate. The main objective of this lab is to develop a temperature measurement system using a thermistor.

HARWARE DESIGN



SOFTWARE DATA

calib.h

```
// calib.h
// extra calibration offset after building the entire system
//#define CALIBRATION_OFFSET 70
```

ADC.c

```
#include "ADC.h"
#include "lm3s1968.h"

volatile unsigned long ADCvalue = 0;;

//unsigned short const ADCdata[SIZE]={
//        0,2,27,53,79,107,135,165,196,228,262,
//     296,332,370,409,449,491,535,581,628,677,
//     728,781,837,894,954,1016,1023,1023,1023,1023,

//     1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,

//     1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1024};

//              unsigned short const ADCdata[SIZE]={
//        0,2,27,53,79,107,135,165,196,228,262,
//     296,332,370,409,449,491,535,581,628,677,
//     728,781,837,894,954,1016,1,2,3,4,
//     5,6,7,8,9,10,11,12,13,14,
//     15,16,17,18,19,20,21,22,23,24,25,1024};

// 20-40
//-------------------------------------------------------------------------------------
unsigned short const ADCdata[SIZE]={0,4,8,21,34,48,62,76,90,104,119,

     134,150,165,181,198,214,231,249,266,284,
     302,321,340,359,379,399,419,440,462,483,
     505,528,551,574,598,623,647,673,699,725,
     752,779,807,836,865,895,925,956,987,1020,1023,1024};


// 0-40
//unsigned short const Tdata[SIZE]={
//     4000,4000,3920,3840,3760,3680,3600,3520,3440,3360,3280,
//     3200,3120,3040,2960,2880,2800,2720,2640,2560,2480,
//     2400,2320,2240,2160,2080,2000,1920,1840,1760,1680,
//     1600,1520,1440,1360,1280,1200,1120,1040,960,880,
//     800,720,640,560,480,400,320,240,160,80,0,0};

//20-40
//-------------------------------------------------------------------------------------
```

```c
unsigned short const Tdata[SIZE]={4000,4000,3960,3920,3880,3840,3800,3760,3720,3680,3640,

     3600,3560,3520,3480,3440,3400,3360,3320,3280,3240,
     3200,3160,3120,3080,3040,3000,2960,2920,2880,2840,
     2800,2760,2720,2680,2640,2600,2560,2520,2480,2440,
     2400,2360,2320,2280,2240,2200,2160,2120,2080,2040,2000,2000};


//0-40
//unsigned short const Rdata[SIZE]={
//        512,512,530,548,567,587,608,629,652,675,700,725,
//        751,779,808,838,869,901,935,971,1008,1046,1086,
//        1128,1172,1218,1266,1316,1368,1423,1480,1540,1603,
//     1668,1737,1809,1884,1963,2046,2132,2223,2318,2418,
//     2523,2633,2748,2870,2997,3131,3271,3419,3574,3574};
//20-40
//-----------------------------------------------------------------------------------------
unsigned short const Rdata[SIZE]={517,517,526,534,543,552,561,570,580,589,599,

     609,619,630,640,651,662,673,685,697,708,

     721,733,746,759,772,785,799,813,827,841,

     856,871,887,903,919,935,952,969,986,1004,

     1022,1041,1060,1079,1099,1119,1140,1161,1182,1204,1226,1226};
//-----------------------------------------------------------------------------------------
// There are many choices to make when using the ADC, and many
// different combinations of settings will all do basically the
// same thing.  For simplicity, this function makes some choices
// for you.  When calling this function, be sure that it does
// not conflict with any other software that may be running on
// the microcontroller.  Particularly, ADC sample sequencer 3
// is used here because it only takes one sample, and only one
// sample is absolutely needed.  Sample sequencer 3 generates a
// raw interrupt when the conversion is complete, but it is not
// promoted to a controller interrupt.  Software triggers the
// ADC conversion and waits for the conversion to finish.  If
// somewhat precise periodic measurements are required, the
// software trigger can occur in a periodic interrupt.  This
// approach has the advantage of being simple.  However, it does
// not guarantee real-time.
//
// A better approach would be to use a hardware timer to trigger
// the ADC conversion independently from software and generate
// an interrupt when the conversion is finished.  Then, the
// software can transfer the conversion result to memory and
// process it after all measurements are complete.

// This initialization function sets up the ADC according to the
// following parameters.  Any parameters not explicitly listed
// below are not modified:
// Max sample rate: <=125,000 samples/second
// Sequencer 0 priority: 1st (highest)
// Sequencer 1 priority: 2nd
// Sequencer 2 priority: 3rd
// Sequencer 3 priority: 4th (lowest)
// SS3 triggering event: software trigger
// SS3 1st sample source: programmable using variable 'channelNum' [0:7]
// SS3 interrupts: enabled but not promoted to controller
//------------------------------------------------------------------------------------------

void ADC_InitSWTriggerSeq3(unsigned char channelNum)
{
  if(channelNum > 7)
       {
    return;   // 0 to 7 are valid channels on the LM3S1968
  }

  SYSCTL_RCGC0_R |= 0x00010000;   // 1) activate ADC clock
  SYSCTL_RCGC0_R &= ~0x00000300;  // 2) configure for 125K
  ADC_SSPRI_R = 0x3210;           // 3) Sequencer 3 is lowest priority
  ADC_ACTSS_R &= ~0x0008;         // 4) disable sample sequencer 3
  ADC_EMUX_R &= ~0xF000;          // 5) seq3 is software trigger
  ADC_SSMUX3_R &= ~0x0007;        // 6) clear SS3 field
  ADC_SSMUX3_R += channelNum;     //    set channel
  ADC_SSCTL3_R = 0x0006;          // 7) no TS0 D0, yes IE0 END0
  ADC_IM_R &= ~0x0008;            // 8) disable SS3 interrupts
  ADC_ACTSS_R |= 0x0008;          // 9) enable sample sequencer 3
}
//------------------------------------------------------------------------------------------
//------------ADC_InSeq3------------
// Busy-wait Analog to digital conversion
// Input: none
// Output: 10-bit result of ADC conversion
unsigned long ADC_InSeq3(void)
{
       unsigned long result;
  ADC_PSSI_R = 0x0008;             // 1) initiate SS3
  while((ADC_RIS_R&0x08)==0){};    // 2) wait for conversion done
  result = ADC_SSFIFO3_R&0x3FF;    // 3) read result
  ADC_ISC_R = 0x0008;              // 4) acknowledge completion
  return result;
}
//------------------------------------------------------------------------------------------
unsigned short ADC2Temp(unsigned short adcSample, int* index)
{
       int i;

       for(i = 0; i < SIZE; i++)
       {
              if(adcSample < ADCdata[i])
              {
                     break;
              }
       }
       *index = i-1;
       return Tdata[i];
}
//------------------------------------------------------------------------------------------
unsigned short interpolate(unsigned short rawADC, int i)
{
       int deltaADC;
       int deltaT;
       int scaleADC;
       int percentChange;
```

```
	int tempDiff;
	int newTemp;

	deltaADC = ADCdata[i+1] - ADCdata[i];
	scaleADC = rawADC - ADCdata[i]; // should be a + number always
	percentChange = (scaleADC*1000+(deltaADC/2))/deltaADC;

	deltaT = Tdata[i] - Tdata[i+1];
	tempDiff = (deltaT*percentChange+500)/1000;
	newTemp = Tdata[i]-tempDiff;

	return newTemp;
}
```

ADC.h

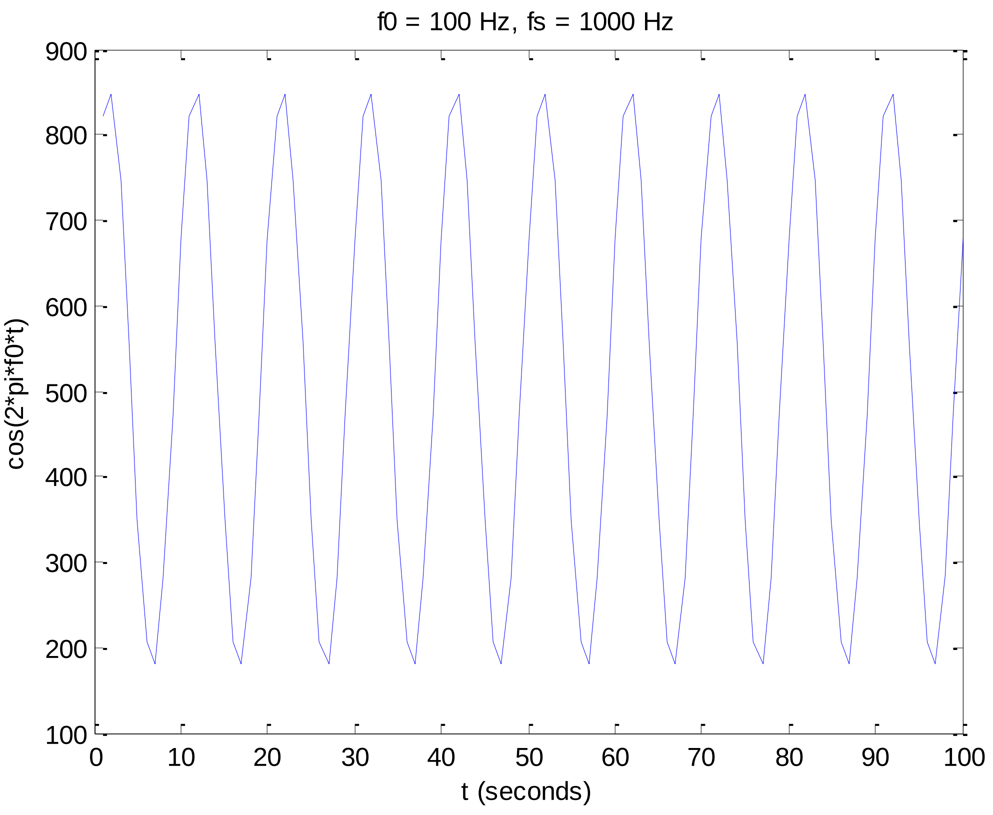
```
// ADC.h
#define SIZE 53

extern unsigned short const Rdata[SIZE];
extern unsigned short const Tdata[SIZE];
extern unsigned short const ADCdata[SIZE];
extern volatile unsigned long ADCvalue;

unsigned long ADC_InSeq3(void);
void ADC_InitSWTriggerSeq3(unsigned char channelNum);
unsigned short interpolate(unsigned short rawADC, int i);
unsigned short ADC2Temp(unsigned short adcSample, int* index);
```

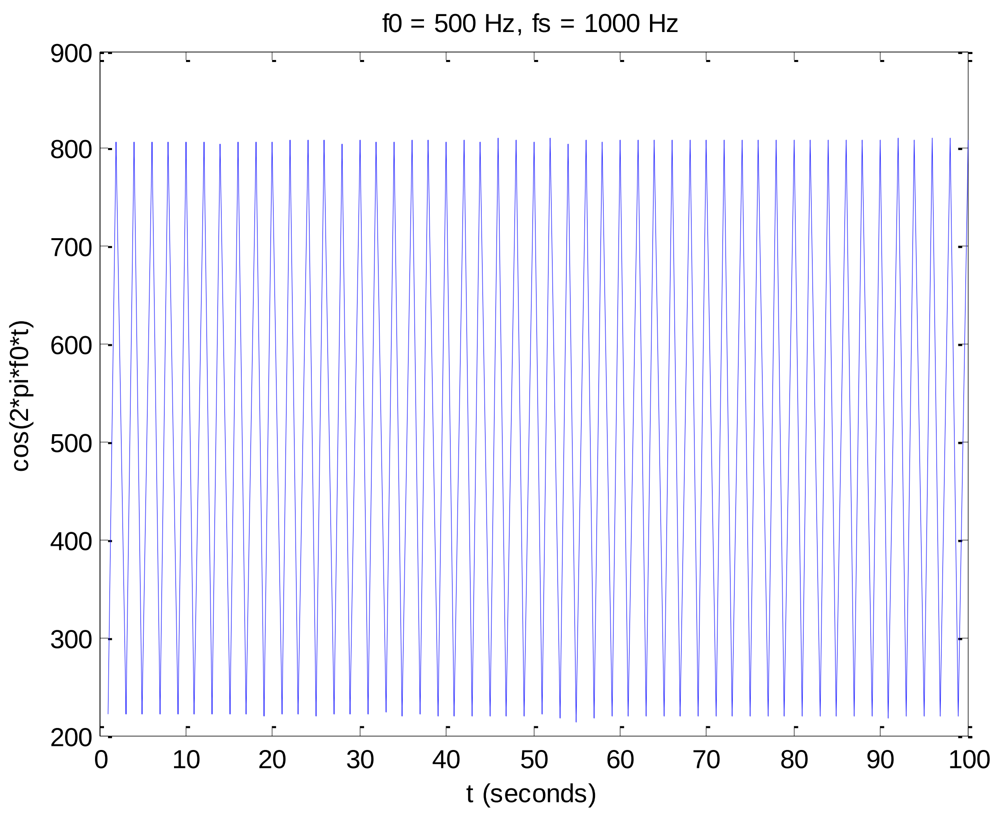
Main

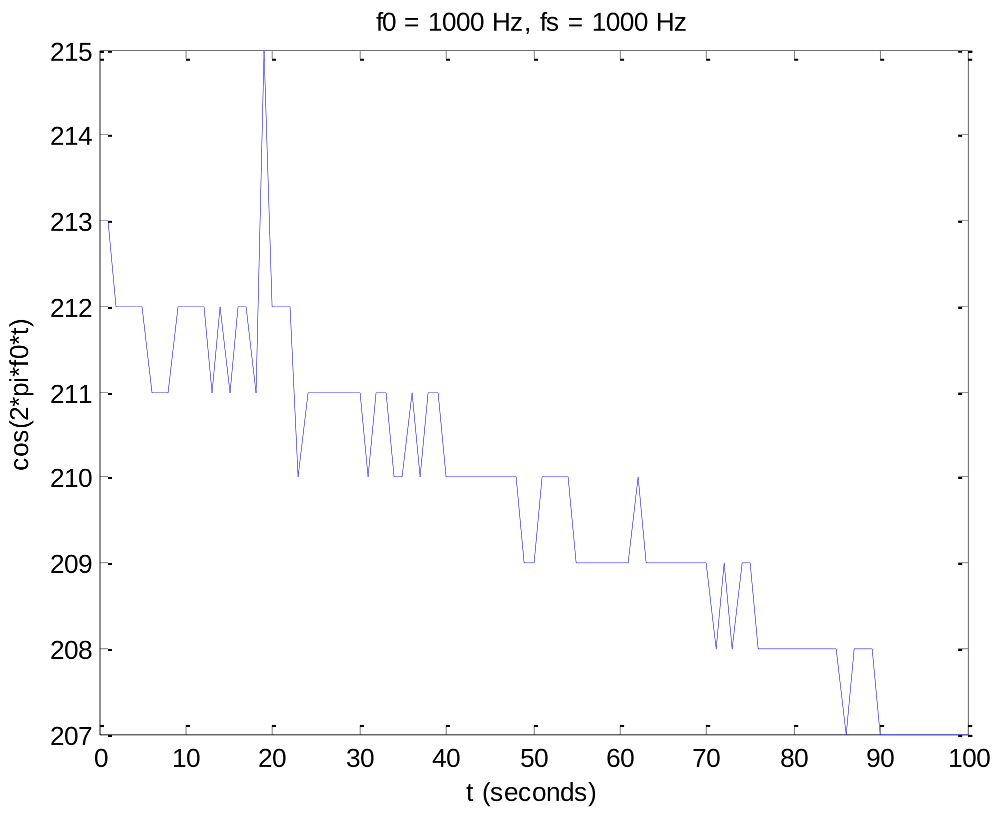
```
int main(void)
{
  PLL_Init();      // 25 MHz Clock
	PortG_Init();    // Initialize the Heartbeat

	Output_Init();
  Output_Color(15);
  Delay(4000000);
	PG2 = 1;
	Delay(4000000);
	PG2 = 0;

  ADC_InitSWTriggerSeq3(0);     // allow time to finish activating ADC0
  Timer0A_Init100HzInt();       // set up Timer0A for 100 Hz interrupts
	//ADCvalue = 0;

	RIT128x96x4PlotClear(2000,4000,20,27,34,40);
	EnableInterrupts();
  while(1)
	{
    WaitForInterrupt();
  }
}
```

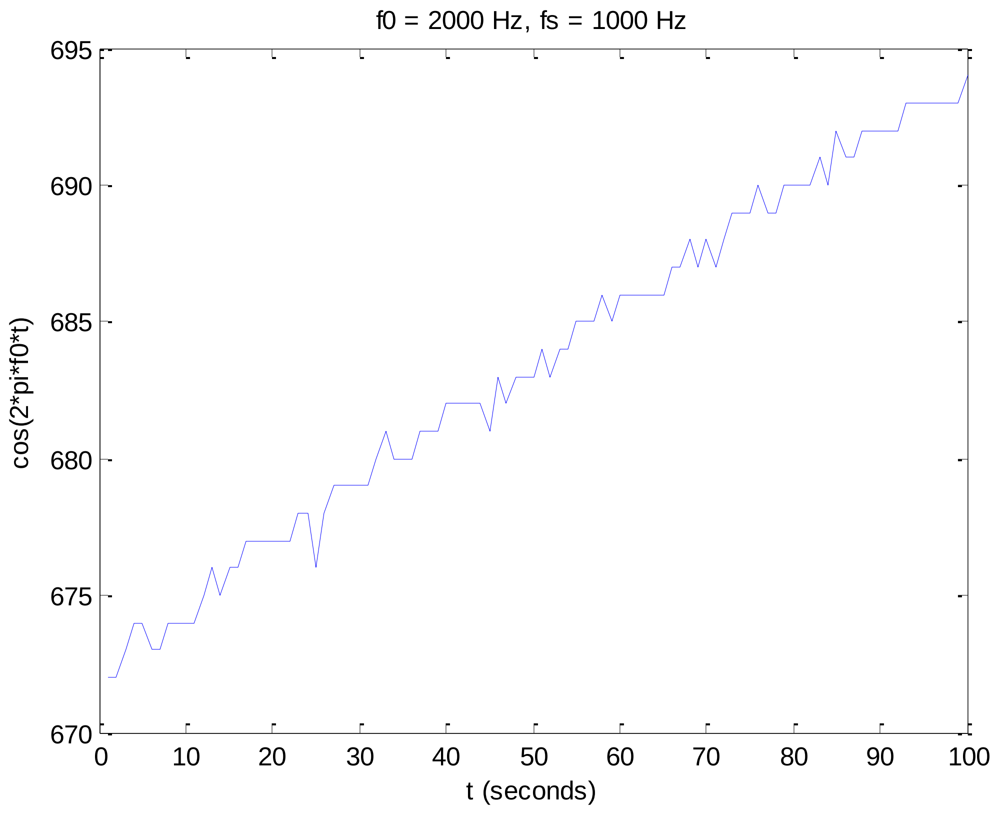
MEASUREMENT DATA

Wave Forms




f0 = 500 Hz, fs = 1000 Hz
cos(2*pi*f0*t)
t (seconds)
900
800
700
600
500
400
300
200
0
10
20
30
40
50
60
70
80
90
100


f0 = 1000 Hz, fs = 1000 Hz
cos(2*pi*f0*t)
t (seconds)
215
214
213
212
211
210
209
208
207
0
10
20
30
40
50
60
70
80
90
100



Average accuracy (with units in °C) = $(1/n) \sum_{i=1}^{n} |xti - xmi|$ = .066 °C with n = 5

Mean = 24.94 °C

Standard Deviation = .0399 °C

ANALYSIS AND DISCUSSION

1. The Nyquist theorem is rate at which you must sample in order to faithfully represent the signal from the digital signals. It is used in this lab to sample the analog signal to get more accurate results.
2. Resolution is the number of discrete values that are produced over a range of analog values. Accuracy is how correct the measurement is to the actual value.
3. Reproducibility = $((1/n) \sum_{i=1}^{n}(\text{average} - \text{result})^2)^{1/2}$
4. The purpose of the Low Pass Filter is to filter out the higher frequencies before it gets to the ADC.
5. Using the conversions from the excel spreadsheet and calibrating our thermistor in software accounts for the nonlinearity in the Resistance vs Temp plot and converts the Voltage vs Temp to a linear plot.
6. We had the excel spreadsheet available to us and it only calculated ~50 points . So, using those points we applied linear interpolation in between each of our 50 data points. This method has advantages for being more memory efficient. The downside is that it loses a little precision since we approximate using a linear approximation and it takes more computation power that could be used on other tasks.

```
// Timer.c
#include "lm3s1968.h"
#include "Timer.h"
#include "rit128x96x4.h"
#include <stdio.h>
#include "Output.h"
#include "calib.h"

extern void Delay(unsigned long ulCount);
extern unsigned short plotPoints[100];

#define CALIBRATION_OFFSET 30

// This debug function initializes Timer0A to request interrupts
// at a 10 Hz frequency.  It is similar to FreqMeasure.c.
void Timer0A_Init100HzInt(void)
{
  volatile unsigned long delay;
  DisableInterrupts();
  // **** general initialization ****
  SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER0;// activate timer0
  delay = SYSCTL_RCGC1_R;          // allow time to finish activating
  TIMER0_CTL_R &= ~TIMER_CTL_TAEN; // disable timer0A during setup
  TIMER0_CFG_R = TIMER_CFG_16_BIT; // configure for 16-bit timer mode
  // **** timer0A initialization ****
                                   // configure for periodic mode
  TIMER0_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
  TIMER0_TAPR_R = 249;             // prescale value for 10us
  TIMER0_TAILR_R = 999;           // start value for 100 Hz interrupts
  TIMER0_IMR_R |= TIMER_IMR_TATOIM;// enable timeout (rollover) interrupt
  TIMER0_ICR_R = TIMER_ICR_TATOCINT;// clear timer0A timeout flag
  TIMER0_CTL_R |= TIMER_CTL_TAEN;  // enable timer0A 16-bit, periodic, interrupts
  // **** interrupt initialization ****
                                   // Timer0A=priority 2
  NVIC_PRI4_R = (NVIC_PRI4_R&0x00FFFFFF)|0x40000000; // top 3 bits
  NVIC_EN0_R |= NVIC_EN0_INT19;    // enable interrupt 19 in NVIC
}
//------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------
void Timer0A_Handler(void)
{
	static int i = 0;
	static int k = 0;
	unsigned short interpData = 0;
	DisableInterrupts();
  TIMER0_ICR_R = TIMER_ICR_TATOCINT;    // acknowledge timer0A timeout
  //PG2 = 0x04;                           // profile
  ADCvalue = ADC_InSeq3();
	ADC2Temp(ADCvalue, &i);
	interpData = interpolate(ADCvalue,i) + CALIBRATION_OFFSET;
	plotPoints[k] = interpData;
	RIT128x96x4UDecOut4(ADCvalue,50,10,12);
	RIT128x96x4FixOut2(interpData,75, 10, 15);

	RIT128x96x4PlotPoint(plotPoints[k]);
	RIT128x96x4PlotNext(); // called 108 times
	RIT128x96x4ShowPlot();
```

```
        RIT128x96x4UDecOut4(ADCvalue,50,10,12);
        RIT128x96x4FixOut2(interpData,75, 10, 15);

        Delay(100000);
        PG2 ^= 0xFF;

        k++;
        if(k == 100)
        {
               k = 0;
               RIT128x96x4PlotReClear();
        }

        EnableInterrupts();
}

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
//*****************************************************************************
//
// rit128x96x4.c - Driver for the RIT 128x96x4 graphical OLED display.
//
// Copyright (c) 2007-2010 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 6075 of the EK-LM3S1968 Firmware Package.
//
//*****************************************************************************

//*****************************************************************************
//
//! \addtogroup display_api
//! @{
//
//*****************************************************************************

#include "inc/hw_ssi.h"
#include "inc/hw_memmap.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
```

```
#include "rit128x96x4.h"

//*****************************************************************************
//
// Macros that define the peripheral, port, and pin used for the OLEDDC
// panel control signal.
//
//*****************************************************************************
#define SYSCTL_PERIPH_GPIO_OLEDDC   SYSCTL_PERIPH_GPIOH
#define GPIO_OLEDDC_BASE            GPIO_PORTH_BASE
#define GPIO_OLEDDC_PIN             GPIO_PIN_2
#define GPIO_OLEDEN_PIN             GPIO_PIN_3

//*****************************************************************************
//
// Flags to indicate the state of the SSI interface to the display.
//
//*****************************************************************************
static volatile unsigned long g_ulSSIFlags;
#define FLAG_SSI_ENABLED        0
#define FLAG_DC_HIGH            1

//*****************************************************************************
//
// Buffer for storing sequences of command and data for the display.
//
//*****************************************************************************
static unsigned char g_pucBuffer[8];

//*****************************************************************************
//
// Define the SSD1329 128x96x4 Remap Setting(s).  This will be used in
// several places in the code to switch between vertical and horizontal
// address incrementing.  Note that the controller support 128 rows while
// the RIT display only uses 96.
//
// The Remap Command (0xA0) takes one 8-bit parameter.  The parameter is
// defined as follows.
//
// Bit 7: Reserved
// Bit 6: Disable(0)/Enable(1) COM Split Odd Even
//        When enabled, the COM signals are split Odd on one side, even on
//        the other.  Otherwise, they are split 0-63 on one side, 64-127 on
//        the other.
// Bit 5: Reserved
// Bit 4: Disable(0)/Enable(1) COM Remap
//        When Enabled, ROW 0-127 map to COM 127-0 (that is, reverse row order)
// Bit 3: Reserved
// Bit 2: Horizontal(0)/Vertical(1) Address Increment
//        When set, data RAM address will increment along the column rather
//        than along the row.
// Bit 1: Disable(0)/Enable(1) Nibble Remap
//        When enabled, the upper and lower nibbles in the DATA bus for access
//        to the data RAM are swapped.
// Bit 0: Disable(0)/Enable(1) Column Address Remap
//        When enabled, DATA RAM columns 0-63 are remapped to Segment Columns
//        127-0.
//
```

```
//*****************************************************************************
#define RIT_INIT_REMAP      0x52 // app note says 0x51
#define RIT_INIT_OFFSET     0x00
static const unsigned char g_pucRIT128x96x4VerticalInc[]   = { 0xA0, 0x56 };
static const unsigned char g_pucRIT128x96x4HorizontalInc[] = { 0xA0, 0x52 };

//*****************************************************************************
//
// A 5x7 font (in a 6x8 cell, where the sixth column is omitted from this
// table) for displaying text on the OLED display.  The data is organized as
// bytes from the left column to the right column, with each byte containing
// the top row in the LSB and the bottom row in the MSB.
//
// Note:  This is the same font data that is used in the EK-LM3S811
// osram96x16x1 driver.  The single bit-per-pixel is expaned in the StringDraw
// function to the appropriate four bit-per-pixel gray scale format.
//
//*****************************************************************************
static const unsigned char g_pucFont[96][5] =
{
    { 0x00, 0x00, 0x00, 0x00, 0x00 }, // " "
    { 0x00, 0x00, 0x4f, 0x00, 0x00 }, // !
    { 0x00, 0x07, 0x00, 0x07, 0x00 }, // "
    { 0x14, 0x7f, 0x14, 0x7f, 0x14 }, // #
    { 0x24, 0x2a, 0x7f, 0x2a, 0x12 }, // $
    { 0x23, 0x13, 0x08, 0x64, 0x62 }, // %
    { 0x36, 0x49, 0x55, 0x22, 0x50 }, // &
    { 0x00, 0x05, 0x03, 0x00, 0x00 }, // '
    { 0x00, 0x1c, 0x22, 0x41, 0x00 }, // (
    { 0x00, 0x41, 0x22, 0x1c, 0x00 }, // )
    { 0x14, 0x08, 0x3e, 0x08, 0x14 }, // *
    { 0x08, 0x08, 0x3e, 0x08, 0x08 }, // +
    { 0x00, 0x50, 0x30, 0x00, 0x00 }, // ,
    { 0x08, 0x08, 0x08, 0x08, 0x08 }, // -
    { 0x00, 0x60, 0x60, 0x00, 0x00 }, // .
    { 0x20, 0x10, 0x08, 0x04, 0x02 }, // /
    { 0x3e, 0x51, 0x49, 0x45, 0x3e }, // 0
    { 0x00, 0x42, 0x7f, 0x40, 0x00 }, // 1
    { 0x42, 0x61, 0x51, 0x49, 0x46 }, // 2
    { 0x21, 0x41, 0x45, 0x4b, 0x31 }, // 3
    { 0x18, 0x14, 0x12, 0x7f, 0x10 }, // 4
    { 0x27, 0x45, 0x45, 0x45, 0x39 }, // 5
    { 0x3c, 0x4a, 0x49, 0x49, 0x30 }, // 6
    { 0x01, 0x71, 0x09, 0x05, 0x03 }, // 7
    { 0x36, 0x49, 0x49, 0x49, 0x36 }, // 8
    { 0x06, 0x49, 0x49, 0x29, 0x1e }, // 9
    { 0x00, 0x36, 0x36, 0x00, 0x00 }, // :
    { 0x00, 0x56, 0x36, 0x00, 0x00 }, // ;
    { 0x08, 0x14, 0x22, 0x41, 0x00 }, // <
    { 0x14, 0x14, 0x14, 0x14, 0x14 }, // =
    { 0x00, 0x41, 0x22, 0x14, 0x08 }, // >
    { 0x02, 0x01, 0x51, 0x09, 0x06 }, // ?
    { 0x32, 0x49, 0x79, 0x41, 0x3e }, // @
    { 0x7e, 0x11, 0x11, 0x11, 0x7e }, // A
    { 0x7f, 0x49, 0x49, 0x49, 0x36 }, // B
    { 0x3e, 0x41, 0x41, 0x41, 0x22 }, // C
    { 0x7f, 0x41, 0x41, 0x22, 0x1c }, // D
    { 0x7f, 0x49, 0x49, 0x49, 0x41 }, // E
    { 0x7f, 0x09, 0x09, 0x09, 0x01 }, // F
    { 0x3e, 0x41, 0x49, 0x49, 0x7a }, // G
    { 0x7f, 0x08, 0x08, 0x08, 0x7f }, // H
    { 0x00, 0x41, 0x7f, 0x41, 0x00 }, // I
    { 0x20, 0x40, 0x41, 0x3f, 0x01 }, // J
    { 0x7f, 0x08, 0x14, 0x22, 0x41 }, // K
    { 0x7f, 0x40, 0x40, 0x40, 0x40 }, // L
    { 0x7f, 0x02, 0x0c, 0x02, 0x7f }, // M
    { 0x7f, 0x04, 0x08, 0x10, 0x7f }, // N
    { 0x3e, 0x41, 0x41, 0x41, 0x3e }, // O
    { 0x7f, 0x09, 0x09, 0x09, 0x06 }, // P
    { 0x3e, 0x41, 0x51, 0x21, 0x5e }, // Q
    { 0x7f, 0x09, 0x19, 0x29, 0x46 }, // R
    { 0x46, 0x49, 0x49, 0x49, 0x31 }, // S
    { 0x01, 0x01, 0x7f, 0x01, 0x01 }, // T
    { 0x3f, 0x40, 0x40, 0x40, 0x3f }, // U
    { 0x1f, 0x20, 0x40, 0x20, 0x1f }, // V
    { 0x3f, 0x40, 0x38, 0x40, 0x3f }, // W
    { 0x63, 0x14, 0x08, 0x14, 0x63 }, // X
    { 0x07, 0x08, 0x70, 0x08, 0x07 }, // Y
    { 0x61, 0x51, 0x49, 0x45, 0x43 }, // Z
    { 0x00, 0x7f, 0x41, 0x41, 0x00 }, // [
    { 0x02, 0x04, 0x08, 0x10, 0x20 }, // "\"
    { 0x00, 0x41, 0x41, 0x7f, 0x00 }, // ]
    { 0x04, 0x02, 0x01, 0x02, 0x04 }, // ^
    { 0x40, 0x40, 0x40, 0x40, 0x40 }, // _
    { 0x00, 0x01, 0x02, 0x04, 0x00 }, // `
    { 0x20, 0x54, 0x54, 0x54, 0x78 }, // a
    { 0x7f, 0x48, 0x44, 0x44, 0x38 }, // b
    { 0x38, 0x44, 0x44, 0x44, 0x20 }, // c
    { 0x38, 0x44, 0x44, 0x48, 0x7f }, // d
    { 0x38, 0x54, 0x54, 0x54, 0x18 }, // e
    { 0x08, 0x7e, 0x09, 0x01, 0x02 }, // f
    { 0x0c, 0x52, 0x52, 0x52, 0x3e }, // g
    { 0x7f, 0x08, 0x04, 0x04, 0x78 }, // h
    { 0x00, 0x44, 0x7d, 0x40, 0x00 }, // i
    { 0x20, 0x40, 0x44, 0x3d, 0x00 }, // j
    { 0x7f, 0x10, 0x28, 0x44, 0x00 }, // k
    { 0x00, 0x41, 0x7f, 0x40, 0x00 }, // l
    { 0x7c, 0x04, 0x18, 0x04, 0x78 }, // m
    { 0x7c, 0x08, 0x04, 0x04, 0x78 }, // n
    { 0x38, 0x44, 0x44, 0x44, 0x38 }, // o
    { 0x7c, 0x14, 0x14, 0x14, 0x08 }, // p
    { 0x08, 0x14, 0x14, 0x18, 0x7c }, // q
    { 0x7c, 0x08, 0x04, 0x04, 0x08 }, // r
    { 0x48, 0x54, 0x54, 0x54, 0x20 }, // s
    { 0x04, 0x3f, 0x44, 0x40, 0x20 }, // t
    { 0x3c, 0x40, 0x40, 0x20, 0x7c }, // u
    { 0x1c, 0x20, 0x40, 0x20, 0x1c }, // v
    { 0x3c, 0x40, 0x30, 0x40, 0x3c }, // w
    { 0x44, 0x28, 0x10, 0x28, 0x44 }, // x
    { 0x0c, 0x50, 0x50, 0x50, 0x3c }, // y
    { 0x44, 0x64, 0x54, 0x4c, 0x44 }, // z
    { 0x00, 0x08, 0x36, 0x41, 0x00 }, // {
    { 0x00, 0x00, 0x7f, 0x00, 0x00 }, // |
    { 0x00, 0x41, 0x36, 0x08, 0x00 }, // }
//    { 0x02, 0x01, 0x02, 0x04, 0x02 }, // ~
    { 0x00, 0x06, 0x09, 0x09, 0x06 }, // ~ changed to degree symbol
    { 0x5C, 0x62, 0x02, 0x62, 0x5C }  // omega symbol
};

//*****************************************************************************
//
// The sequence of commands used to initialize the SSD1329 controller.  Each
// command is described as follows:  there is a byte specifying the number of
// bytes in the command sequence, followed by that many bytes of command data.
// Note:  This initialization sequence is derived from RIT App Note for
// the P14201.  Values used are from the RIT app note, except where noted.
//
//*****************************************************************************
static const unsigned char g_pucRIT128x96x4Init[] =
{
    //
    // Unlock commands
    //
    3, 0xFD, 0x12, 0xe3,

    //
    // Display off
    //
    2, 0xAE, 0xe3,

    //
    // Icon off
    //
    3, 0x94, 0, 0xe3,

    //
    // Multiplex ratio
    //
    3, 0xA8, 95, 0xe3,

    //
    // Contrast
    //
    3, 0x81, 0xb7, 0xe3,

    //
    // Pre-charge current
    //
    3, 0x82, 0x3f, 0xe3,

    //
    // Display Re-map
    //
    3, 0xA0, RIT_INIT_REMAP, 0xe3,

    //
    // Display Start Line
    //
    3, 0xA1, 0, 0xe3,

    //
    // Display Offset
    //
    3, 0xA2, RIT_INIT_OFFSET, 0xe3,
    //
    // Display Mode Normal
    //
    2, 0xA4, 0xe3,

    //
    // Phase Length
    //
    3, 0xB1, 0x11, 0xe3,

    //
    // Frame frequency
    //
    3, 0xB2, 0x23, 0xe3,

    //
    // Front Clock Divider
    //
    3, 0xB3, 0xe2, 0xe3,

    //
    // Set gray scale table.  App note uses default command:
    // 2, 0xB7, 0xe3
    // This gray scale attempts some gamma correction to reduce the
    // the brightness of the low levels.
    //
    17, 0xB8, 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 19, 22, 26, 30, 0xe3,

    //
    // Second pre-charge period. App note uses value 0x04.
    //
    3, 0xBB, 0x01, 0xe3,

    //
    // Pre-charge voltage
    //
    3, 0xBC, 0x3f, 0xe3,

    //
    // Display ON
    //
    2, 0xAF, 0xe3,
};

//*****************************************************************************
//
//! \internal
//!
//! Write a sequence of command bytes to the SSD1329 controller.
//!
//! The data is written in a polled fashion; this function will not return
//! until the entire byte sequence has been written to the controller.
//!
//! \return None.
//
//*****************************************************************************
static void
```

```
RITWriteCommand(const unsigned char *pucBuffer, unsigned long ulCount)
{
    //
    // Return if SSI port is not enabled for RIT display.
    //
    if(!HWREGBITW(&g_ulSSIFlags, FLAG_SSI_ENABLED))
    {
        return;
    }

    //
    // See if data mode is enabled.
    //
    if(HWREGBITW(&g_ulSSIFlags, FLAG_DC_HIGH))
    {
        //
        // Wait until the SSI is not busy, meaning that all previous data has
        // been transmitted.
        //
        while(SSIBusy(SSI0_BASE))
        {
        }

        //
        // Clear the command/control bit to enable command mode.
        //
        GPIOPinWrite(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN, 0);
        HWREGBITW(&g_ulSSIFlags, FLAG_DC_HIGH) = 0;
    }

    //
    // Loop while there are more bytes left to be transferred.
    //
    while(ulCount != 0)
    {
        //
        // Write the next byte to the controller.
        //
        SSIDataPut(SSI0_BASE, *pucBuffer++);

        //
        // Decrement the BYTE counter.
        //
        ulCount--;
    }
}

//*****************************************************************************
//
//! \internal
//!
//! Write a sequence of data bytes to the SSD1329 controller.
//!
//! The data is written in a polled fashion; this function will not return
//! until the entire byte sequence has been written to the controller.
//!
//! \return None.
//
```

```
//*****************************************************************************
static void
RITWriteData(const unsigned char *pucBuffer, unsigned long ulCount)
{
    //
    // Return if SSI port is not enabled for RIT display.
    //
    if(!HWREGBITW(&g_ulSSIFlags, FLAG_SSI_ENABLED))
    {
        return;
    }

    //
    // See if command mode is enabled.
    //
    if(!HWREGBITW(&g_ulSSIFlags, FLAG_DC_HIGH))
    {
        //
        // Wait until the SSI is not busy, meaning that all previous commands
        // have been transmitted.
        //
        while(SSIBusy(SSI0_BASE))
        {
        }

        //
        // Set the command/control bit to enable data mode.
        //
        GPIOPinWrite(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN, GPIO_OLEDDC_PIN);
        HWREGBITW(&g_ulSSIFlags, FLAG_DC_HIGH) = 1;
    }

    //
    // Loop while there are more bytes left to be transferred.
    //
    while(ulCount != 0)
    {
        //
        // Write the next byte to the controller.
        //
        SSIDataPut(SSI0_BASE, *pucBuffer++);

        //
        // Decrement the BYTE counter.
        //
        ulCount--;
    }
}

//*****************************************************************************
//
//! Clears the OLED display.
//!
//! This function will clear the display RAM.  All pixels in the display will
//! be turned off.
//!
//! \return None.
//
```

```
//*****************************************************************************
void
RIT128x96x4Clear(void)
{
    static const unsigned char pucCommand1[] = { 0x15, 0, 63 };
    static const unsigned char pucCommand2[] = { 0x75, 0, 127 };
    unsigned long ulRow, ulColumn;

    //
    // Clear out the buffer used for sending bytes to the display.
    //
    *(unsigned long *)&g_pucBuffer[0] = 0;
    *(unsigned long *)&g_pucBuffer[4] = 0;

    //
    // Set the window to fill the entire display.
    //
    RITWriteCommand(pucCommand1, sizeof(pucCommand1));
    RITWriteCommand(pucCommand2, sizeof(pucCommand2));
    RITWriteCommand(g_pucRIT128x96x4HorizontalInc,
                    sizeof(g_pucRIT128x96x4HorizontalInc));

    //
    // Loop through the rows
    //
    for(ulRow = 0; ulRow < 96; ulRow++)
    {
        //
        // Loop through the columns.  Each byte is two pixels,
        // and the buffer hold 8 bytes, so 16 pixels are cleared
        // at a time.
        //
        for(ulColumn = 0; ulColumn < 128; ulColumn += 8 * 2)
        {
            //
            // Write 8 clearing bytes to the display, which will
            // clear 16 pixels across.
            //
            RITWriteData(g_pucBuffer, sizeof(g_pucBuffer));
        }
    }
}

//*****************************************************************************
//
//! Displays a string on the OLED display.
//!
//! \param pcStr is a pointer to the string to display.
//! \param ulX is the horizontal position to display the string, specified in
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display the string, specified in
//! rows from the top edge of the display.
//! \param ucLevel is the 4-bit gray scale value to be used for displayed text.
//!
//! This function will draw a string on the display.  Only the ASCII characters
//! between 32 (space) and 126 (tilde) are supported; other characters will
//! result in random data being draw on the display (based on whatever appears
//! before/after the font in memory).  The font is mono-spaced, so characters
//! such as ``i'' and ``l'' have more white space around them than characters
//! such as ``m'' or ``w''.
//!
//! If the drawing of the string reaches the right edge of the display, no more
//! characters will be drawn.  Therefore, special care is not required to avoid
//! supplying a string that is ``too long'' to display.
//!
//! \note Because the OLED display packs 2 pixels of data in a single byte, the
//! parameter \e ulX must be an even column number (for example, 0, 2, 4, and
//! so on).
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4StringDraw(const char *pcStr, unsigned long ulX,
                      unsigned long ulY, unsigned char ucLevel)
{
    unsigned long ulIdx1, ulIdx2;
    unsigned char ucTemp;

    //
    // Check the arguments.
    //
    ASSERT(ulX < 128);
    ASSERT((ulX & 1) == 0);
    ASSERT(ulY < 96);
    ASSERT(ucLevel < 16);

    //
    // Setup a window starting at the specified column and row, ending
    // at the right edge of the display and 8 rows down (single character row).
    //
    g_pucBuffer[0] = 0x15;
    g_pucBuffer[1] = ulX / 2;
    g_pucBuffer[2] = 63;
    RITWriteCommand(g_pucBuffer, 3);
    g_pucBuffer[0] = 0x75;
    g_pucBuffer[1] = ulY;
    g_pucBuffer[2] = ulY + 7;
    RITWriteCommand(g_pucBuffer, 3);
    RITWriteCommand(g_pucRIT128x96x4VerticalInc,
                    sizeof(g_pucRIT128x96x4VerticalInc));

    //
    // Loop while there are more characters in the string.
    //
    while(*pcStr != 0)
    {
        //
        // Get a working copy of the current character and convert to an
        // index into the character bit-map array.
        //
        ucTemp = *pcStr++ & 0x7f;
        if(ucTemp < ' ')
        {
            ucTemp = 0;
        }
        else
        {
            ucTemp -= ' ';
        }

        //
        // Build and display the character buffer.
        //
        for(ulIdx1 = 0; ulIdx1 < 6; ulIdx1 += 2)
        {
            //
            // Convert two columns of 1-bit font data into a single data
            // byte column of 4-bit font data.
            //
            for(ulIdx2 = 0; ulIdx2 < 8; ulIdx2++)
            {
                g_pucBuffer[ulIdx2] = 0;
                if(g_pucFont[ucTemp][ulIdx1] & (1 << ulIdx2))
                {
                    g_pucBuffer[ulIdx2] = (ucLevel << 4) & 0xf0;
                }
                if((ulIdx1 < 4) &&
                   (g_pucFont[ucTemp][ulIdx1 + 1] & (1 << ulIdx2)))
                {
                    g_pucBuffer[ulIdx2] |= (ucLevel << 0) & 0x0f;
                }
            }

            //
            // Send this byte column to the display.
            //
            RITWriteData(g_pucBuffer, 8);
            ulX += 2;

            //
            // Return if the right side of the display has been reached.
            //
            if(ulX == 128)
            {
                return;
            }
        }
    }
}

//*****************************************************************************
//
//! Displays an image on the OLED display.
//!
//! \param pucImage is a pointer to the image data.
//! \param ulX is the horizontal position to display this image, specified in
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display this image, specified in
//! rows from the top of the display.
//! \param ulWidth is the width of the image, specified in columns.
//! \param ulHeight is the height of the image, specified in rows.
//!
//! This function will display a bitmap graphic on the display.  Because of the
//! format of the display RAM, the starting column (\e ulX) and the number of
//! columns (\e ulWidth) must be an integer multiple of two.
//!
//! The image data is organized with the first row of image data appearing left
//! to right, followed immediately by the second row of image data.  Each byte
//! contains the data for two columns in the current row, with the leftmost
//! column being contained in bits 7:4 and the rightmost column being contained
//! in bits 3:0.
//!
//! For example, an image six columns wide and seven scan lines tall would
//! be arranged as follows (showing how the twenty one bytes of the image would
//! appear on the display):
//!
//! \verbatim
//!     +-------------------+-------------------+-------------------+
//!     |      Byte 0       |      Byte 1       |      Byte 2       |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//!     |      Byte 3       |      Byte 4       |      Byte 5       |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//!     |      Byte 6       |      Byte 7       |      Byte 8       |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//!     |      Byte 9       |      Byte 10      |      Byte 11      |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//!     |      Byte 12      |      Byte 13      |      Byte 14      |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//!     |      Byte 15      |      Byte 16      |      Byte 17      |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//!     |      Byte 18      |      Byte 19      |      Byte 20      |
//!     +---------+---------+---------+---------+---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+---------+---------+
//! \endverbatim
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4ImageDraw(const unsigned char *pucImage, unsigned long ulX,
                     unsigned long ulY, unsigned long ulWidth,
                     unsigned long ulHeight)
{
    //
    // Check the arguments.
    //
    ASSERT(ulX < 128);
    ASSERT((ulX & 1) == 0);
    ASSERT(ulY < 96);
    ASSERT((ulX + ulWidth) <= 128);
    ASSERT((ulY + ulHeight) <= 96);
    ASSERT((ulWidth & 1) == 0);

    //
    // Setup a window starting at the specified column and row, and ending
    // at the column + width and row+height.
    //
    g_pucBuffer[0] = 0x15;
    g_pucBuffer[1] = ulX / 2;
    g_pucBuffer[2] = (ulX + ulWidth - 2) / 2;
    RITWriteCommand(g_pucBuffer, 3);
    g_pucBuffer[0] = 0x75;
    g_pucBuffer[1] = ulY;
    g_pucBuffer[2] = ulY + ulHeight - 1;
    RITWriteCommand(g_pucBuffer, 3);
    RITWriteCommand(g_pucRIT128x96x4HorizontalInc,
                    sizeof(g_pucRIT128x96x4HorizontalInc));

    //
    // Loop while there are more rows to display.
    //
    while(ulHeight--)
    {
        //
        // Write this row of image data.
        //
        RITWriteData(pucImage, (ulWidth / 2));

        //
        // Advance to the next row of the image.
        //
        pucImage += (ulWidth / 2);
    }
}

//*****************************************************************************
//
//! Enable the SSI component of the OLED display driver.
//!
//! \param ulFrequency specifies the SSI Clock Frequency to be used.
//!
//! This function initializes the SSI interface to the OLED display.
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4Enable(unsigned long ulFrequency)
{
    //
    // Disable the SSI port.
    //
    SSIDisable(SSI0_BASE);

    //
    // Configure the SSI0 port for master mode.
    //
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
                       SSI_MODE_MASTER, ulFrequency, 8);

    //
    // (Re)Enable SSI control of the FSS pin.
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_3);
    GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_STRENGTH_8MA,
                     GPIO_PIN_TYPE_STD_WPU);

    //
    // Enable the SSI port.
    //
    SSIEnable(SSI0_BASE);

    //
    // Indicate that the RIT driver can use the SSI Port.
    //
    HWREGBITW(&g_ulSSIFlags, FLAG_SSI_ENABLED) = 1;
}

//*****************************************************************************
//
//! Enable the SSI component of the OLED display driver.
//!
//! This function initializes the SSI interface to the OLED display.
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4Disable(void)
{
    unsigned long ulTemp;

    //
    // Indicate that the RIT driver can no longer use the SSI Port.
    //
    HWREGBITW(&g_ulSSIFlags, FLAG_SSI_ENABLED) = 0;

    //
    // Drain the receive fifo.
    //
    while(SSIDataGetNonBlocking(SSI0_BASE, &ulTemp) != 0)
    {
    }

    //
    // Disable the SSI port.
    //
    SSIDisable(SSI0_BASE);

    //
    // Disable SSI control of the FSS pin.
    //
    GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_3);
    GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_STRENGTH_8MA,
```

```c
                     GPIO_PIN_TYPE_STD_WPU);
    GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, GPIO_PIN_3);
}

//*****************************************************************************
//
//! Initialize the OLED display.
//!
//! \param ulFrequency specifies the SSI Clock Frequency to be used.
//!
//! This function initializes the SSI interface to the OLED display and
//! configures the SSD1329 controller on the panel.
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4Init(unsigned long ulFrequency)
{
    unsigned long ulIdx;

    //
    // Enable the SSI0 and GPIO port blocks as they are needed by this driver.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO_OLEDDC);

    //
    // Configure the SSI0CLK and SSIOTX pins for SSI operation.
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_5);
    GPIOPadConfigSet(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_5,
                     GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_STD_WPU);

    //
    // Configure the GPIO port pin used as a D/Cn signal for OLED device,
    // and the port pin used to enable power to the OLED panel.
    //
    GPIOPinTypeGPIOOutput(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN);
    GPIOPadConfigSet(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN,
                     GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_STD);
    GPIOPinWrite(GPIO_OLEDDC_BASE, GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN,
                 GPIO_OLEDDC_PIN | GPIO_OLEDEN_PIN);
    HWREGBITW(&g_ulSSIFlags, FLAG_DC_HIGH) = 1;

    //
    // Configure and enable the SSI0 port for master mode.
    //
    RIT128x96x4Enable(ulFrequency);

    //
    // Clear the frame buffer.
    //
    RIT128x96x4Clear();

    //
    // Initialize the SSD1329 controller.  Loop through the initialization
    // sequence array, sending each command "string" to the controller.
    //
    for(ulIdx = 0; ulIdx < sizeof(g_pucRIT128x96x4Init);
        ulIdx += g_pucRIT128x96x4Init[ulIdx] + 1)
    {
        //
        // Send this command.
        //
        RITWriteCommand(g_pucRIT128x96x4Init + ulIdx + 1,
                        g_pucRIT128x96x4Init[ulIdx] - 1);
    }
}

//*****************************************************************************
//
//! Turns on the OLED display.
//!
//! This function will turn on the OLED display, causing it to display the
//! contents of its internal frame buffer.
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4DisplayOn(void)
{
    unsigned long ulIdx;

    //
    // Initialize the SSD1329 controller.  Loop through the initialization
    // sequence array, sending each command "string" to the controller.
    //
    for(ulIdx = 0; ulIdx < sizeof(g_pucRIT128x96x4Init);
        ulIdx += g_pucRIT128x96x4Init[ulIdx] + 1)
    {
        //
        // Send this command.
        //
        RITWriteCommand(g_pucRIT128x96x4Init + ulIdx + 1,
                        g_pucRIT128x96x4Init[ulIdx] - 1);
    }
}

//*****************************************************************************
//
//! Turns off the OLED display.
//!
//! This function will turn off the OLED display.  This will stop the scanning
//! of the panel and turn off the on-chip DC-DC converter, preventing damage to
//! the panel due to burn-in (it has similar characters to a CRT in this
//! respect).
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4DisplayOff(void)
{
    static const unsigned char pucCommand1[] =
    {
        0xAE, 0xe3
    };

    //
    // Put the display to sleep.
    //
    RITWriteCommand(pucCommand1, sizeof(pucCommand1));
}

/***************Fix3Str***************
 converts fixed point number to ASCII string
 format signed 32-bit with resolution 0.01
 range -99.999 to +99.999
 Input: signed 32-bit integer part of fixed point number
 Output: null-terminated string exactly 7 characters plus null
 Examples
  12345 to " 12.345"
 -82100 to "-82.100"
  -1002 to " -1.002"
     31 to "  0.031"

 */
void Fix3Str(long const num, char *string){
  long n;

  if(num<0){
    n = -num;
    string[0] = '-';
  } else{
    n = num;
    string[0] = ' ';
  }
  if(n>99999){      // too big
    string[1] = '*';
    string[2] = '*';
    string[3] = '.';
    string[4] = '*';
    string[5] = '*';
    string[6] = '*';
    string[7] = 0;
    return;
  }
  if(n>9999){    // 10000 to 99999
    string[1] = '0'+n/10000;
    n = n%10000;
  } else{        // 0 to 9999
    if(num<0){
      string[0] = ' ';
      string[1] = '-';
    } else {
      string[1] = ' ';
    }
  }
  string[2] = '0'+n/1000;
  n = n%1000;
  string[3] = '.';
  string[4] = '0'+n/100;
  n = n%100;
  string[5] = '0'+n/10;
  n = n%10;
  string[6] = '0'+n;
  string[7] = 0;
}

/****************Fix4Str***************
 converts fixed point number to ASCII string
 format signed 32-bit with resolution 0.0001
 range -9.9999 to +9.9999
 Input: signed 32-bit integer part of fixed point number
 Output: null-terminated string exactly 7 characters plus null
 Examples
  12345 to " 1.2345"
 -82100 to "-8.2100"
  -1002 to " -.1002"
     31 to " 0.0031"

 */
void Fix4Str(long const num, char *string){
  long n;

  if(num<0){
    n = -num;
    string[0] = '-';
  } else{
    n = num;
    string[0] = ' ';
  }
  if(n>99999){       // too big
    string[1] = '*';
    string[2] = '.';
    string[3] = '*';
    string[4] = '*';
    string[5] = '*';
    string[6] = '*';
    string[7] = 0;
    return;
  }
  string[1] = '0'+n/10000;
  n = n%10000;
  string[2] = '.';
  string[3] = '0'+n/1000;
  n = n%1000;
  string[4] = '0'+n/100;
  n = n%100;
  string[5] = '0'+n/10;
  n = n%10;
  string[6] = '0'+n;
  string[7] = 0;
}
/****************Fix2Str***************
 converts fixed point number to ASCII string
 format signed 32-bit with resolution 0.001
 range -999.99 to +999.99
 Input: signed 32-bit integer part of fixed point number
 Output: null-terminated string exactly 7 characters plus null
 Examples
  12345 to " 123.45"
 -82100 to "-821.00"
   -102 to "  -1.02"
     31 to "   0.31"
 */
void Fix2Str(long const num, char *string){
  long n;

  if(num<0){
    n = -num;
    string[0] = '-';
  } else{
    n = num;
    string[0] = ' ';
  }
  if(n>99999){       // too big
    string[1] = '*';
    string[2] = '*';
    string[3] = '*';
    string[4] = '.';
    string[5] = '*';
    string[6] = '*';
    string[7] = 0;
    return;
  }  if(n>9999){
    string[1] = '0'+n/10000;
    n = n%10000;
    string[2] = '0'+n/1000;
  } else{
    if(n>999){
      if(num<0){
        string[0] = ' ';
        string[1] = '-';
      } else {
        string[1] = ' ';
      }
      string[2] = '0'+n/1000;
    } else{
      if(num<0){
        string[0] = ' ';
        string[1] = ' ';
        string[2] = '-';
      } else {
        string[1] = ' ';
        string[2] = ' ';
      }
    }
  }
  n = n%1000;
  string[3] = '0'+n/100;
  n = n%100;
  string[4] = '.';
  string[5] = '0'+n/10;
  n = n%10;
  string[6] = '0'+n;
  string[7] = 0;
```

```c
}
/****************Fix1Str***************
 converts fixed point number to ASCII string
 format signed 32-bit with resolution 0.01
 range -9999.9 to +9999.9
 Input: signed 32-bit integer part of fixed point number
 Output: null-terminated string exactly 8 characters plus null
 Examples
  12345 to " 1234.5"
 -82100 to "-8210.0"
   -102 to "  -10.2"
     31 to "    3.1"

 */
void Fix1Str(long const num, char *string)
{
  long n;

  if(num<0){
    n = -num;
    string[0] = '-';
  } else{
    n = num;
    string[0] = ' ';
  }
  if(n>99999){      // too big
    string[1] = '*';
    string[2] = '*';
    string[3] = '*';
    string[4] = '*';
    string[5] = '*';
    string[6] = '*';
    string[7] = 0;
    return;
  }  if(n>9999){ //10000-99999
    string[1] = '0'+n/10000;
    n = n%10000;
    string[2] = '0'+n/1000;
    n = n%1000;
    string[3] = '0'+n/100;
  } else{
    if(n>999){  //1000-9999
      if(num<0){
        string[0] = ' ';
        string[1] = '-';
      } else {
        string[1] = ' ';
      }
      string[2] = '0'+n/1000;
      n = n%1000;
      string[3] = '0'+n/100;
    } else{
      if(n>99){  //100-999
        if(num<0){
          string[0] = ' ';
          string[1] = ' ';
          string[2] = '-';
        } else {
          string[2] = ' ';
        }
        string[3] = '0'+n/100;
      } else{    //0-99
        if(num<0){
          string[0] = ' ';
          string[1] = ' ';
          string[2] = ' ';
          string[3] = '-';
        }  else{
          string[0] = ' ';
          string[1] = ' ';
          string[2] = ' ';
          string[3] = ' ';
       }
      }
    }
  }
  // 0 to 99
  n = n%100;
  string[4] = '0'+n/10;
  string[5] = '.';
  string[6] = '0'+n%10;
  string[7] = 0;
}

//*****************************************************************************
//
//! Displays a fixed-point number (0.01 resolution) on the OLED display.
//!
//! \param num is the integer part of the fixed-point number to display.
//! \param ulX is the horizontal position to display the string, specified in
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display the string, specified in
//! rows from the top edge of the display.
//! \param ucLevel is the 4-bit gray scale value to be used for displayed text.
//!
//! This function will display the number on the display.
//! The num should be between -99999 and 99999
//!
//! If the drawing of the string reaches the right edge of the display, no more
//! characters will be drawn.  Therefore, special care is not required to avoid
//! supplying a string that is ``too long'' to display.
//!
//! \note Because the OLED display packs 2 pixels of data in a single byte, the
//! parameter \e ulX must be an even column number (for example, 0, 2, 4, and
//! so on).
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4FixOut2(long num, unsigned long ulX,
                      unsigned long ulY, unsigned char ucLevel)  {
char string[10];
  Fix2Str(num, string);
  RIT128x96x4StringDraw(string, ulX, ulY, ucLevel);
}
```

```c
/****************Fix2Str***************
 converts fixed point number to ASCII string
 format signed 32-bit with resolution 0.01
 range -999.99 to +999.99
 Input: signed 32-bit integer part of fixed point number
 Output: null-terminated string exactly 8 characters plus null
 Examples
  345 to "3.45"
  100 to "1.00"
   99 to "0.99"
   31 to "0.31"

 */
void Fix22Str(long const num, char *string){
  long n;

  if(num<0){
    n = 0;
  }else{
    n= num;
  }
  if(n>999){      // too big
    string[0] = '*';
    string[1] = '.';
    string[2] = '*';
    string[3] = '*';
    string[4] = 0;
    return;
  }
  string[0] = '0'+n/100;
  n = n%100;
  string[1] = '.';
  string[2] = '0'+n/10;
  n = n%10;
  string[3] = '0'+n;
  string[4] = 0;
}
//*****************************************************************************
//
//! Displays a fixed-point number (0.01 resolution) on the OLED display.
//!
//! \param num is the integer part of the fixed-point number to display.
//! \param ulX is the horizontal position to display the string, specified in
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display the string, specified in
//! rows from the top edge of the display.
//! \param ucLevel is the 4-bit gray scale value to be used for displayed text.
//!
//! This function will display the number on the display.
//! The num should be between -99999 and 99999
//!
//! If the drawing of the string reaches the right edge of the display, no more
//! characters will be drawn.  Therefore, special care is not required to avoid
//! supplying a string that is ``too long'' to display.
//!
//! \note Because the OLED display packs 2 pixels of data in a single byte, the
//! parameter \e ulX must be an even column number (for example, 0, 2, 4, and
//! so on).
//!
//! \return None.
//
//*****************************************************************************
void  RIT128x96x4FixOut22(long num, unsigned long ulX,
                          unsigned long ulY, unsigned char ucLevel)
{
       char string[10];
  Fix22Str(num, string);
  RIT128x96x4StringDraw(string, ulX, ulY, ucLevel);
}


/****************Int2Str***************
 converts signed integer number to ASCII string
 format signed 32-bit
 range -99999 to +99999
 Input: signed 32-bit integer
 Output: null-terminated string exactly 7 characters plus null
 Examples
  12345 to " 12345"
 -82100 to "-82100"
   -102 to "  -102"
     31 to "    31"

 */
void Int2Str(long const num, char *string)
{
  long n;

  if(num<0){
    n = -num;
  } else{
    n = num;
  }
  if(n>99999){      // too big
    string[0] = '*';
    string[1] = '*';
    string[2] = '*';
    string[3] = '*';
    string[4] = '*';
    string[5] = '*';
    string[6] = '*';
    string[7] = 0;
    return;
  }
  if(n>9999){  // 10000 to 99999
    if(num<0){
      string[0] = '-';
    } else {
      string[0] = ' ';
    }
    string[1] = '0'+n/10000;
    n = n%10000;
    string[2] = '0'+n/1000;
    n = n%1000;
    string[3] = '0'+n/100;
    n = n%100;
    string[4] = '0'+n/10;
    n = n%10;
    string[5] = '0'+n;
    string[6] = ' ';
    string[7] = 0;
    return;
  }
  if(n>999){   // 1000 to 9999
    string[0] = ' ';
    if(num<0){
      string[1] = '-';
    } else {
      string[1] = ' ';
    }
    string[2] = '0'+n/1000;
    n = n%1000;
    string[3] = '0'+n/100;
    n = n%100;
    string[4] = '0'+n/10;
    n = n%10;
    string[5] = '0'+n;
    string[6] = ' ';
    string[7] = 0;
    return;
  }
  if(n>99){   // 100 to 999
    string[0] = ' ';
    string[1] = ' ';
    if(num<0){
      string[2] = '-';
    } else {
      string[2] = ' ';
    }
    string[3] = '0'+n/100;
    n = n%100;
    string[4] = '0'+n/10;
    n = n%10;
    string[5] = '0'+n;
    string[6] = ' ';
    string[7] = 0;
    return;
  }
  if(n>9){   // 10 to 99
    string[0] = ' ';
    string[1] = ' ';
    string[2] = ' ';
    if(num<0){
      string[3] = '-';
    } else {
      string[3] = ' ';
    }
    string[4] = '0'+n/10;
    n = n%10;
    string[5] = '0'+n;
    string[6] = ' ';
    string[7] = 0;
    return;
  }
  // 0 to 9
  string[0] = ' ';
  string[1] = ' ';
  string[2] = ' ';
  string[3] = ' ';
  if(num<0){
    string[4] = '-';
  } else {
    string[4] = ' ';
  }
  string[5] = '0'+n;
  string[6] = ' ';
  string[7] = 0;
}
//*****************************************************************************
//
//! Displays an integer on the OLED display.
//!
//! \param num is the integer to display.
//! \param ulX is the horizontal position to display the string, specified in
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display the string, specified in
//! rows from the top edge of the display.
//! \param ucLevel is the 4-bit gray scale value to be used for displayed text.
//!
//! This function will display the number on the display.
//! The num should be between -99999 and 99999
//!
//! If the drawing of the string reaches the right edge of the display, no more
//! characters will be drawn.  Therefore, special care is not required to avoid
//! supplying a string that is ``too long'' to display.
//!
//! \note Because the OLED display packs 2 pixels of data in a single byte, the
//! parameter \e ulX must be an even column number (for example, 0, 2, 4, and
//! so on).
//!
//! \return None.
//
//*****************************************************************************
void RIT128x96x4DecOut5(long num, unsigned long ulX,
                      unsigned long ulY, unsigned char ucLevel)
{
       char string[10];
  Int2Str(num, string);
  RIT128x96x4StringDraw(string, ulX, ulY, ucLevel);
}

/***************UInt2Str4***************
 converts unsigned integer number to ASCII string
 format unsigned 32-bit
 range 0 to 9999
 Input: unsigned 32-bit integer
 Output: null-terminated string exactly 4 characters plus null
 Examples
  1234 to "1234"
  821  to " 821"
  10   to "  10"
  3    to "   3"

 */
void UInt2Str4(unsigned long const num, char *string){
  unsigned long n=num;

  if(n>9999){      // too big
    string[0] = '*';
    string[1] = '*';
    string[2] = '*';
    string[3] = '*';
    string[4] = 0;
    return;
  }
  if(n>999){   // 1000 to 9999
    string[0] = '0'+n/1000;
    n = n%1000;
    string[1] = '0'+n/100;
    n = n%100;
    string[2] = '0'+n/10;
    n = n%10;
    string[3] = '0'+n;
    string[4] = 0;
    return;
  }
  if(n>99){   // 100 to 999
    string[0] = ' ';
    string[1] = '0'+n/100;
    n = n%100;
    string[2] = '0'+n/10;
    n = n%10;
    string[3] = '0'+n;
    string[4] = 0;
    return;
  }
  if(n>9){   // 10 to 99
    string[0] = ' ';
    string[1] = ' ';
    string[2] = '0'+n/10;
    n = n%10;
    string[3] = '0'+n;
    string[4] = 0;
    return;
  }
  // 0 to 9
  string[0] = ' ';
  string[1] = ' ';
  string[2] = ' ';
  string[3] = '0'+n;
  string[4] = 0;
}

//*****************************************************************************
//
//! Displays an integer on the OLED display.
//!
//! \param num is the integer to display.
//! \param ulX is the horizontal position to display the string, specified in
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display the string, specified in
//! rows from the top edge of the display.
//! \param ucLevel is the 4-bit gray scale value to be used for displayed text.
//!
//! This function will display the number on the display.
//! The num should be between 0 and 9999
//!
//! If the drawing of the string reaches the right edge of the display, no more
//! characters will be drawn.  Therefore, special care is not required to avoid
//! supplying a string that is ``too long'' to display.
//!
//! \note Because the OLED display packs 2 pixels of data in a single byte, the
//! parameter \e ulX must be an even column number (for example, 0, 2, 4, and
//! so on).
//!
//! \return None.
//
//*****************************************************************************
void RIT128x96x4UDecOut4(unsigned long num, unsigned long ulX,
                     unsigned long ulY, unsigned char ucLevel)
{
       char string[10];
  UInt2Str4(num, string);
  RIT128x96x4StringDraw(string, ulX, ulY, ucLevel);
}

/***************UInt2Str3***************
 converts unsigned integer number to ASCII string
 format unsigned 32-bit
 range 0 to 999
 Input: unsigned 32-bit integer
 Output: null-terminated string exactly 3 characters plus null
 Examples
  821  to "821"
  10   to " 10"
  3    to "  3"

 */
void UInt2Str3(unsigned long const num, char *string){
  unsigned long n=num;

  if(n>999){      // too big
    string[0] = '*';
    string[1] = '*';
    string[2] = '*';
    string[3] = 0;
    return;
  }

  if(n>99){   // 100 to 999
    string[0] = '0'+n/100;
    n = n%100;
    string[1] = '0'+n/10;
    n = n%10;
    string[2] = '0'+n;
    string[3] = 0;
    return;
  }
  if(n>9){   // 10 to 99
    string[0] = ' ';
    string[1] = '0'+n/10;
    n = n%10;
    string[2] = '0'+n;
    string[3] = 0;
    return;
  }
  // 0 to 9
  string[0] = ' ';
  string[1] = ' ';
  string[2] = '0'+n;
  string[3] = 0;
}
/****************Int2Str2**************
 converts signed integer number to ASCII string
 format signed 32-bit
 range -9 to 99
 Input: signed 32-bit integer
 Output: null-terminated string exactly 2 characters plus null
 Examples
  82  to "82"
  1   to " 1"
 -3   to "-3"
 */
void Int2Str2(long const n, char *string){
  if((n>99)||(n<-9)){      // too big, too small
    string[0] = ' ';
    string[1] = ' ';
    string[2] = 0;
    return;
  }

  if(n>9){   // 10 to 99
    string[0] = '0'+n/10;
    string[1] = '0'+n%10;
    string[2] = 0;
    return;
  }
  if(n>=0){   // 0 to 9
    string[0] = ' ';
    string[1] = '0'+n;
    string[2] = 0;
    return;
  }
  // -9 to -1
  string[0] = '-';
  string[1] = '0'-n;
  string[2] = 0;
}
//*****************************************************************************
//
//! Displays an integer on the OLED display.
//!
//! \param num is the integer to display.
//! \param ulX is the horizontal position to display the string, specified in
```

```c
//! columns from the left edge of the display.
//! \param ulY is the vertical position to display the string, specified in
//! rows from the top edge of the display.
//! \param ucLevel is the 4-bit gray scale value to be used for displayed text.
//!
//! This function will display the number on the display.
//! The num should be between 0 and 999
//!
//! If the drawing of the string reaches the right edge of the display, no more
//! characters will be drawn.  Therefore, special care is not required to avoid
//! supplying a string that is ``too long'' to display.
//!
//! \note Because the OLED display packs 2 pixels of data in a single byte, the
//! parameter \e ulX must be an even column number (for example, 0, 2, 4, and
//! so on).
//!
//! \return None.
//
//*****************************************************************************
void
RIT128x96x4UDecOut3(unsigned long num, unsigned long ulX,
                      unsigned long ulY, unsigned char ucLevel)  {
char string[10];
  UInt2Str3(num, string);
  RIT128x96x4StringDraw(string, ulX, ulY, ucLevel);
}
/***************RIT128x96x4DecOut2**************
 output 2 digit signed integer number to ASCII string
 format signed 32-bit
 range -9 to 99
 Input: signed 32-bit integer, position, level
 Output: none
 Examples
  82  to "82"
  1   to " 1"
 -3   to "-3"
 */
void RIT128x96x4DecOut2(unsigned long num, unsigned long ulX,
                      unsigned long ulY, unsigned char ucLevel)
{
       char string[4];
  Int2Str2(num, string);
  RIT128x96x4StringDraw(string, ulX, ulY, ucLevel);
}
//! Graphics plot, an image 128 columns wide and 80 scan lines tall would
//! be arranged as follows (showing how the twenty one bytes of the image would
//! appear on the display):
//!
//!
//!     +-------------------+-------------------+  +-------------------+
//!     |      Byte 0       |      Byte 1       |  |      Byte 55      |
//!     +---------+---------+---------+---------+  +---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |  | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+  +---------+---------+
//!     |      Byte 56      |      Byte 57      |  |      Byte 111     |
//!     +---------+---------+---------+---------+  +---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |  | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+  +---------+---------+
//! row j, j from 0 to 79
//!     +---------+---------+---------+---------+  +---------+---------+
//!     |      Byte 56*j    |      Byte 56*j+1  |  |  Byte 56*j+127    |
//!     +---------+---------+---------+---------+  +---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |  | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+  +---------+---------+
//!
//!     +---------+---------+---------+---------+  +---------+---------+
//!     |    Byte 4424      |    Byte 4425      |  |   Byte 4479       |
//!     +---------+---------+---------+---------+  +---------+---------+
//!     | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |  | 7 6 5 4 | 3 2 1 0 |
//!     +---------+---------+---------+---------+  +---------+---------+
unsigned char PlotImage[4480];  // 112 wide by 80 tall plot
// 0-12 have hashes
// 3 is yaxis
// 4 to 111 is data (108 points)
long Ymax,Ymin,X;  // X goes from 4 to 111
long Yrange,YrangeDiv2;
long Y0 = 1; long Y1 = 2; long Y2 = 3; long Y3 = 4;
void RIT128x96x4PlotReClear(void)
{
       unsigned long i;
  for(i=0; i<4480; i++)
       {
    PlotImage[i] = 0; // clear, blank
  }
  X = 4;  // 4 to 111

  for(i=0;i<80;i=i+13)  // 7 hashes at 0,13,26,39,52,65,78
       {
    PlotImage[0+56*i] = 0xAA;  // x=0,1
    PlotImage[1+56*i] = 0xAA;  // x=2,3
  }
  for(i=0;i<79;i=i+1){          // y axis
    PlotImage[1+56*i] |= 0x0A;  // x=3
  }
}
// *************** RIT128x96x4PlotClear ********************
// Clear the graphics buffer, set X coordinate to 0
// It does not output to display until RIT128x96x4ShowPlot called
// Inputs: ymin and ymax are range of the plot
// four numbers are displayed along left edge of plot
// y0,y1,y2,y3, can be -9 to 99, any number outside this range is skipped
// y3 --          hash marks at number           Ymax
//     |
//    --          hash marks between numbers     Ymin+(5*Yrange)/6
//     |
// y2 --                                         Ymin+(4*Yrange)/6
//     |
//    --                                         Ymin+(3*Yrange)/6
//     |
// y1 --                                         Ymin+(2*Yrange)/6
//     |
//    --                                         Ymin+(1*Yrange)/6
//     |
// y0 --                                         Ymin
// Outputs: none
void RIT128x96x4PlotClear(long ymin, long ymax, long y0, long y1, long y2, long y3){
  if(ymax>ymin){
    Ymax = ymax;
    Ymin = ymin;
    Yrange = ymax-ymin;
  } else{
    Ymax = ymin;
    Ymin = ymax;
    Yrange = ymax-ymin;
  }
  YrangeDiv2 = Yrange/2;
  Y0 = y0;
  Y1 = y1;
  Y2 = y2;
  Y3 = y3;

  RIT128x96x4PlotReClear();
  RIT128x96x4DecOut2(Y0,0,84,10);
  RIT128x96x4DecOut2(Y1,0,60,10);
  RIT128x96x4DecOut2(Y2,0,34,10);
  RIT128x96x4DecOut2(Y3,0,10,10);
}

// *************** RIT128x96x4PlotPoint ********************
// Used in the voltage versus time plot, plot one point at y
// It does not output to display until RIT128x96x4ShowPlot called
// Inputs: y is the y coordinate of the point plotted
// Outputs: none
void RIT128x96x4PlotPoint(long y){
long i,j;
  if(y<Ymin) y=Ymin;
  if(y>Ymax) y=Ymax;
  // i goes from 0 to 55
  i = X/2;  // X goes from to 111
  // if X is even, set bits 7-4
  // if X is odd,  set bits 3-0
  // j goes from 0 to 79
  // y=Ymax maps to j=0
  // y=Ymin maps to j=79
  j = (79*(Ymax-y)+YrangeDiv2)/Yrange;
  if(j<0) j = 0;
  if(j>79) j = 79;
  i = 56*j+i;
  if(X&0x01){    // if X is odd,  set bits 3-0
    if(PlotImage[i]&0x0F){
      if((PlotImage[i]&0x0F)<14){
        PlotImage[i] += 2;  // 10,12,14
      }
    } else{
      PlotImage[i] |= 0x08;
    }
  } else{       // if X is even, set bits 7-4
    if(PlotImage[i]&0xF0){
      if((PlotImage[i]&0xF0)<0xE0){
        PlotImage[i] += 0x20;  // 10,12,14
      }
    } else{
      PlotImage[i] |= 0x80;
    }
  }
}
// *************** RIT128x96x4PlotBar ********************
// Used in the voltage versus time bar, plot one bar at y
// It does not output to display until RIT128x96x4ShowPlot called
// Inputs: y is the y coordinate of the bar plotted
// Outputs: none
void RIT128x96x4PlotBar(long y){
long i,j;
  if(y<Ymin) y=Ymin;
  if(y>Ymax) y=Ymax;
  // i goes from 0 to 55
  i = X/2;  // X goes from to 111
  // if X is even, set bits 7-4
  // if X is odd,  set bits 3-0
  // j goes from 0 to 79
  // y=Ymax maps to j=0
  // y=Ymin maps to j=79
  j = (79*(Ymax-y)+YrangeDiv2)/Yrange;
  if(j<0) j = 0;
  if(j>79) j = 79;
  if(X&0x01){    // if X is odd,  set bits 3-0
    for(; j<80; j++){
      PlotImage[56*j+i] |= 0x0C;
    }
  } else{
    for(; j<80; j++){
      PlotImage[56*j+i] |= 0xC0;
    }
  }
}
/*
// full scale defined as 1.25V
unsigned char const dBfs[512]={
79, 79, 79, 79, 79, 79, 77, 75, 72, 70, 68, 67, 65, 64, 63, 61, 60, 59, 58, 57, 56, 55,
55, 54, 53, 52, 52, 51,
50, 50, 49, 49, 48, 48, 47, 47, 46, 46, 45, 45, 44, 44, 43, 43, 43, 42, 42, 41, 41, 41,
40, 40, 40, 39, 39, 39,
38, 38, 38, 38, 37, 37, 37, 36, 36, 36, 36, 35, 35, 35, 35, 34, 34, 34, 34, 33, 33, 33,
33, 32, 32, 32, 32, 32,
31, 31, 31, 31, 31, 30, 30, 30, 30, 30, 29, 29, 29, 29, 29, 29, 28, 28, 28, 28, 28, 28,
27, 27, 27, 27, 27, 27,
26, 26, 26, 26, 26, 26, 25, 25, 25, 25, 25, 25, 25, 24, 24, 24, 24, 24, 24, 24, 24, 23,
23, 23, 23, 23, 23, 23,
23, 22, 22, 22, 22, 22, 22, 22, 22, 21, 21, 21, 21, 21, 21, 21, 21, 21, 20, 20, 20, 20,
20, 20, 20, 20, 20, 19,
19, 19, 19, 19, 19, 19, 19, 19, 19, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 17, 17, 17,
17, 17, 17, 17, 17, 17,
17, 17, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 14,
14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
13, 13, 13, 13, 12, 12,
12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 11, 11, 11, 11, 11, 11, 11, 11, 11,
11, 11, 11, 11, 11, 11,
10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 9, 9, 9, 9, 9, 9, 9,
9, 9, 9, 9, 9, 9, 9, 9,
9, 9, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
*/
// full scaled defined as 0.625V
unsigned char const dBfs[512]={
79,79,79,77,72,68,65,63,60,58,56,55,53,52,50,49,48,47,46,45,44,43,43,42,41,40,40,39,38,38
,37,37,36,36,35,35,34,34,
33,33,32,32,31,31,31,30,30,29,29,29,28,28,28,27,27,27,26,26,26,25,25,25,25,24,24,24,24,23
,23,23,23,22,22,22,22,21,
21,21,21,20,20,20,20,20,19,19,19,19,19,18,18,18,18,18,17,17,17,17,17,17,16,16,16,16,16,15
,15,15,15,15,15,15,14,14,
14,14,14,14,13,13,13,13,13,13,13,12,12,12,12,12,12,12,12,11,11,11,11,11,11,11,10,10,10,10
,10,10,10,10,10,9,9,9,9,9,
9,9,9,8,8,8,8,8,8,8,8,8,8,7,7,7,7,7,7,7,7,7,7,6,6,6,6,6,6,6,6,6,6,5,5,5,5,5,5,5,5,5,5,5,4
,4,4,4,4,4,4,4,4,4,4,4,3,
3,3,3,3,3,3,3,3,3,3,3,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};

// *************** RIT128x96x4PlotdBfs ********************
// Used in the amplitude versus frequency plot, plot bar point at y
// 0 to 0.625V scaled on a log plot from min to max
// It does not output to display until RIT128x96x4ShowPlot called
// Inputs: y is the y ADC value of the bar plotted
// Outputs: none
void RIT128x96x4PlotdBfs(long y){
long i,j;
  if(y<0) y=0;
  if(y>511) y=511;
  // i goes from 0 to 63
  i = X/2;  // X goes from to 128
  // if X is even, set bits 7-4
  // if X is odd,  set bits 3-0
  // j goes from 0 to 79
  // y=511 maps to j=0
  // y=0 maps to j=79
  j = dBfs[y];
  if(X&0x01){    // if X is odd,  set bits 3-0
    for(; j<80; j++){
      PlotImage[64*j+i] |= 0x0C;
    }
  } else{
    for(; j<80; j++){
      PlotImage[64*j+i] |= 0xC0;
    }
  }
}

// *************** RIT128x96x4PlotNext ********************
// Used in all the plots to step the X coordinate one pixel
// X steps from 4 to 111, then back to 4 again
// It does not output to display until RIT128x96x4ShowPlot called
// Inputs: none
// Outputs: none
void RIT128x96x4PlotNext(void){
  if(X==111){
//    RIT128x96x4ImageDraw(PlotImage, 0, 10, 112, 80);
//    RIT128x96x4PlotClear(Ymax,Ymin);
    X = 4;  // start past axis
  } else{
    X++;
  }
}

// *************** RIT128x96x4ShowPlot ********************
// Used in all the plots to write buffer to LED
// Example 1 Voltage versus time
//    RIT128x96x4PlotClear(0,1023);  // range from 0 to 1023
//    RIT128x96x4PlotPoint(data); RIT128x96x4PlotNext(); // called 108 times
//    RIT128x96x4ShowPlot();
// Example 2 Voltage versus time (N data points/pixel, time scale)
//    RIT128x96x4PlotClear(0,1023);  // range from 0 to 1023
//    {
//        RIT128x96x4PlotPoint(data); // called N times
//        RIT128x96x4PlotNext();
//    }   // called 108 times
//    RIT128x96x4ShowPlot();
// Example 3 Voltage versus frequency (512 points)
//    perform FFT to get 512 magnitudes
//    RIT128x96x4PlotClear(0,511);  // parameters ignored
//    {
//        RIT128x96x4PlotPoint(mag); // called 4 times
//        RIT128x96x4PlotPoint(mag);
//        RIT128x96x4PlotPoint(mag);
//        RIT128x96x4PlotPoint(mag);
//        RIT128x96x4PlotNext();
//    }   // called 128 times
//    RIT128x96x4ShowPlot();
// Inputs: none
// Outputs: none
void RIT128x96x4ShowPlot(void){
  RIT128x96x4ImageDraw(PlotImage, 16, 10, 112, 80);
}

//*****************************************************************************
//
// Close the Doxygen group.
//! @}
//
```

```
//*****************************************************************************
//---------------------Fixed_uDecOut2---------------------
// create a fixed point number string with precision=65535
// and resolution=1/100, ranging from 0 to 65534
// Input: number contains an unsigned short to be converted into fixed point
// Output: pointer to the beginning of a string containing the fixed point number
void Fixed_uDecOut2(unsigned short number, char* string)
{
  unsigned short dec_part, frac_part;        //hold decimal part and fractional part

  if (number > 65534)
      {                        //error condition
    string = "***.**";
  }

  dec_part = number / 100;                        //calculate decimal part (quotient)
  frac_part = number % 100;                       //calculate fractional part (remainder)
  sprintf(string,"%d.%.2d",dec_part,frac_part); //put together fixed point number string
}



// PLL.c
#include "PLL.h"


void PLL_Init(void)
{
  // program 4.6 volume 1
  // 1) bypass PLL and system clock divider while initializing
  SYSCTL_RCC_R |=  0x00000800;   // PLL bypass
  SYSCTL_RCC_R &= ~0x00400000;   // do not use system divider
  // 2) select the crystal value and oscillator source
  SYSCTL_RCC_R &= ~0x000003C0;   // clear XTAL field, bits 9-6
  SYSCTL_RCC_R +=  0x00000380;   // 0x0E, configure for 8 MHz crystal
  SYSCTL_RCC_R &= ~0x00000030;   // clear oscillator source field
  SYSCTL_RCC_R +=  0x00000000;   // configure for main oscillator source
  // 3) activate PLL by clearing PWRDN and OEN
  SYSCTL_RCC_R &= ~(0x00002000|0x00001000);
  // 4) set the desired system divider and the USESYSDIV bit
  SYSCTL_RCC_R &= ~0x07800000;   // system clock divider field
  SYSCTL_RCC_R +=  0x03800000;   // configure for 25 MHz clock
  SYSCTL_RCC_R |=  0x00400000;   // Enable System Clock Divider
  // 5) wait for the PLL to lock by polling PLLLRIS
  while((SYSCTL_RIS_R&0x00000040)==0){};  // wait for PLLRIS bit
  // 6) enable use of PLL by clearing BYPASS
  SYSCTL_RCC_R &= ~0x00000800;
}


// Output.c
// Runs on LM3S1968
// Implement the fputc() function in stdio.h to enable useful
// functions such as printf() to write text to the onboard
// organic LED display.  Remember that the OLED is vulnerable
// to screen image "burn-in" like old CRT monitors, so be sure
// to turn the OLED off when it is not in use.
// Daniel Valvano
// July 28, 2011

/* This example accompanies the book
   "Embedded Systems: Real Time Interfacing to the Arm Cortex M3",
   ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2011
   Section 3.4.5

 Copyright 2011 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */

#include <stdio.h>
#include "rit128x96x4.h"
#include "Output.h"

#define CHARCOLS                6    // a character is 6 columns wide (right column
blank)
#define CHARROWS                8    // a character is 8 rows tall (bottom row blank)
#define TOTALCHARCOLUMNS        21   // (128 pixels)/(6 pixels/char) columns
#define TOTALCHARROWS           12   // (96 pixels)/(8 pixels/char) rows
#define WRAP                    1    // automatically wrap to next line

// Cursor x-position [0:126] of next character
static unsigned short CursorX = 0;
// Cursor y-position [0:88] of next character
static unsigned short CursorY = 0;
// Color [0:15] of next character
static unsigned char Color = 15;
// Character 2-D array of the screen character contents
static unsigned char CharBuffer[TOTALCHARROWS][TOTALCHARCOLUMNS];
// Color 2-D array of the screen color contents
static unsigned char ColorBuffer[TOTALCHARROWS][TOTALCHARCOLUMNS];
// Status of OLED display (1 = on; 0 = off)
static unsigned char Status = 0;
// Shift everything up; clear last line; re-print all
void shiftEverythingUp(void){
  int i, j;
  unsigned char outStr[2];
  outStr[1] = 0;              // output string terminated with NULL
                              // put contents of line i+1 in line i and print
  for(i=0; i<(TOTALCHARROWS-1); i=i+1){ // every row but the last one
    for(j=0; j<TOTALCHARCOLUMNS; j=j+1){// every column
      CharBuffer[i][j] = CharBuffer[i+1][j];
      ColorBuffer[i][j] = ColorBuffer[i+1][j];
      outStr[0] = CharBuffer[i][j];
      RIT128x96x4StringDraw((const char *)outStr, j*CHARCOLS, i*CHARROWS,
ColorBuffer[i][j]);
    }
  }
                              // clear the last line
  outStr[0] = ' ';
  for(j=0; j<TOTALCHARCOLUMNS; j=j+1){ // every column
    CharBuffer[TOTALCHARROWS-1][j] = ' ';
    ColorBuffer[TOTALCHARROWS-1][j] = Color;
    RIT128x96x4StringDraw((const char *)outStr, j*CHARCOLS, (TOTALCHARROWS-1)*CHARROWS,
ColorBuffer[TOTALCHARROWS-1][j]);
  }
}
// Print a character to OLED.
int fputc(int ch, FILE *f){
  unsigned char outStr[2];
  if(Status == 0){            // verify that OLED display is on
    return EOF;               // error
  }
  if(Color > 15){             // verify 'Color' is valid
    Color = 15;
  }
  // special case characters require moving the cursor
  if((ch == BACKSPACE) && (CursorX >= CHARCOLS)){
    CursorX = CursorX - CHARCOLS;// back up one character
  }
  else if(ch == TAB){
    while(CursorX < 63){      // still on left side of screen
                              // insert spaces
      CharBuffer[CursorY/CHARROWS][CursorX/CHARCOLS] = ' ';
      ColorBuffer[CursorY/CHARROWS][CursorX/CHARCOLS] = Color;
      outStr[0] = ' ';        // build output string
      outStr[1] = 0;          // terminate with NULL
                              // print
      RIT128x96x4StringDraw((const char *)outStr, CursorX, CursorY, Color);
      CursorX = CursorX + CHARCOLS;
    }
  }
  else if((ch == LF) || (ch == HOME)){
    CursorX = 0;              // move cursor all the way left on current line
  }
  else if((ch == NEWLINE) || (ch == RETURN) || (ch == CR)){
    // fill in the remainder of the current line with spaces
    while((CursorX/CHARCOLS) < TOTALCHARCOLUMNS){
                              // insert spaces
      CharBuffer[CursorY/CHARROWS][CursorX/CHARCOLS] = ' ';
      ColorBuffer[CursorY/CHARROWS][CursorX/CHARCOLS] = Color;
      outStr[0] = ' ';        // build output string
      outStr[1] = 0;          // terminate with NULL
                              // print
      RIT128x96x4StringDraw((const char *)outStr, CursorX, CursorY, Color);
      CursorX = CursorX + CHARCOLS;
    }
    CursorX = 0;              // move cursor all the way left
    if((CursorY/CHARROWS) == (TOTALCHARROWS - 1)){
                              // on the last line
      shiftEverythingUp();
    }
    else{                     // not on the last line; go to next line
      CursorY = CursorY + CHARROWS;
    }
  }
  else{                       // regular character
```

```
                              // check if there is space to print
    if((CursorX/CHARCOLS) == TOTALCHARCOLUMNS){
                              // current line is full
      if(WRAP == 0){          // wrapping is disabled
        return EOF;           // error
      }
      else{                   // wrapping is enabled
        CursorX = 0;          // move cursor all the way left
        if((CursorY/CHARROWS) == (TOTALCHARROWS - 1)){
                              // on the last line
          shiftEverythingUp();
        }
        else{                 // not on the last line; go to next line
          CursorY = CursorY + CHARROWS;
        }
      }
    }
    outStr[0] = ch;           // build output string
    outStr[1] = 0;            // terminate with NULL
                              // print
    RIT128x96x4StringDraw((const char *)outStr, CursorX, CursorY, Color);
                              // store character in the buffer
    CharBuffer[CursorY/CHARROWS][CursorX/CHARCOLS] = ch;
    ColorBuffer[CursorY/CHARROWS][CursorX/CHARCOLS] = Color;
                              // increment cursor
    CursorX = CursorX + CHARCOLS;
  }
  return 1;
}
// No input from OLED, always return 0.
int fgetc (FILE *f){
  return 0;
}
// Function called when file error occurs.
int ferror(FILE *f){
  /* Your implementation of ferror */
  return EOF;
}
//------------Output_Init------------
// Initializes the OLED interface.
// Input: none
// Output: none
void Output_Init(void){
  int i, j;
  RIT128x96x4Init(1000000);   // initialize OLED
  for(i=0; i<TOTALCHARROWS; i=i+1){
    for(j=0; j<TOTALCHARCOLUMNS; j=j+1){
      CharBuffer[i][j] = 0;   // clear screen contents
      ColorBuffer[i][j] = 0;
    }
  }
  CursorX = 0;                // initialize the cursors
  CursorY = 0;
  Color = 15;
  Status = 1;
}

//------------Output_Clear------------
```

```c
// Clears the OLED display.
// Input: none
// Output: none
void Output_Clear(void){
  int i, j;
  RIT128x96x4Clear();          // clear the screen
  for(i=0; i<TOTALCHARROWS; i=i+1){
    for(j=0; j<TOTALCHARCOLUMNS; j=j+1){
      CharBuffer[i][j] = 0;    // clear screen contents
      ColorBuffer[i][j] = 0;
    }
  }
  CursorX = 0;                 // reset the cursors
  CursorY = 0;
}

//------------Output_Off------------
// Turns off the OLED display
// Input: none
// Output: none
void Output_Off(void){         //   to prevent burn-in damage
  RIT128x96x4DisplayOff();     // turn off
  Status = 0;                  // ignore any incoming writes
}

//------------Output_On------------
// Turns on the OLED display
//  called after Output_Off to turn it back on
// Input: none
// Output: none
void Output_On(void){
  RIT128x96x4DisplayOn();      // turn on
  Status = 1;                  // resume accepting writes
}

//------------Output_Color------------
// Set the color of future characters.
// Input: 0 is off, non-zero is on
// Output: none
void Output_Color(unsigned char newColor){
  if(newColor > 15){
    Color = 15;
  }
  else{
    Color = newColor;
  }
}


// ADCSWTrigger.c
// Runs on LM3S1968
// Provide functions that initialize ADC SS3 to be triggered by
// software and trigger a conversion, wait for it to finish,
// and return the result.
// Daniel Valvano
// May 21, 2012

/* This example accompanies the book
```

```
   "Embedded Systems: Interfacing to the Arm Cortex M3",
   ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2012

 Copyright 2012 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */

#include <stdio.h>
#include "lm3s1968.h"
#include "Output.h"
#include "rit128x96x4.h"
#include "driverlib/adc.h"
#include "ADC.h"
#include "PLL.h"
#include "Timer.h"
/*
//unsigned short const ADCdata[SIZE]={
//        0,2,27,53,79,107,135,165,196,228,262,
//     296,332,370,409,449,491,535,581,628,677,
//     728,781,837,894,954,1016,1023,1023,1023,1023,

//     1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,

//     1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1024};

//              unsigned short const ADCdata[SIZE]={
//        0,2,27,53,79,107,135,165,196,228,262,
//     296,332,370,409,449,491,535,581,628,677,
//     728,781,837,894,954,1016,1,2,3,4,
//     5,6,7,8,9,10,11,12,13,14,
//     15,16,17,18,19,20,21,22,23,24,25,1024};

/////// 20-40
/////unsigned short const ADCdata[53]={0,4,8,21,34,48,62,76,90,104,119,

/////     134,150,165,181,198,214,231,249,266,284,
/////     302,321,340,359,379,399,419,440,462,483,
/////     505,528,551,574,598,623,647,673,699,725,
/////     752,779,807,836,865,895,925,956,987,1020,1023,1024};



// 0-40
//unsigned short const Tdata[SIZE]={
//     4000,4000,3920,3840,3760,3680,3600,3520,3440,3360,3280,
//     3200,3120,3040,2960,2880,2800,2720,2640,2560,2480,
//     2400,2320,2240,2160,2080,2000,1920,1840,1760,1680,
//     1600,1520,1440,1360,1280,1200,1120,1040,960,880,
//     800,720,640,560,480,400,320,240,160,80,0,0};
```

```
////////20-40
//////unsigned short const
Tdata[53]={4000,4000,3960,3920,3880,3840,3800,3760,3720,3680,3640,
//////     3600,3560,3520,3480,3440,3400,3360,3320,3280,3240,
//////     3200,3160,3120,3080,3040,3000,2960,2920,2880,2840,
//////     2800,2760,2720,2680,2640,2600,2560,2520,2480,2440,
//////     2400,2360,2320,2280,2240,2200,2160,2120,2080,2040,2000,2000};


//0-40
//unsigned short const Rdata[SIZE]={
//        512,512,530,548,567,587,608,629,652,675,700,725,
//        751,779,808,838,869,901,935,971,1008,1046,1086,
//        1128,1172,1218,1266,1316,1368,1423,1480,1540,1603,
//     1668,1737,1809,1884,1963,2046,2132,2223,2318,2418,
//     2523,2633,2748,2870,2997,3131,3271,3419,3574,3574};

////////20-40
//////unsigned short const Rdata[53]={517,517,526,534,543,552,561,570,580,589,599,

//////     609,619,630,640,651,662,673,685,697,708,

//////     721,733,746,759,772,785,799,813,827,841,

//////     856,871,887,903,919,935,952,969,986,1004,

//////     1022,1041,1060,1079,1099,1119,1140,1161,1182,1204,1226,1226};





//debug code
//
// This program periodically samples ADC channel 0 and stores the
// result to a global variable that can be accessed with the JTAG
// debugger and viewed with the variable watch feature.

//void DisableInterrupts(void); // Disable interrupts
//void EnableInterrupts(void);  // Enable interrupts
//long StartCritical (void);    // previous I bit, disable interrupts
//void EndCritical(long sr);    // restore I bit to previous value
//void WaitForInterrupt(void);  // low power mode

//------------------------------------------------------------------------------------
*/
// delay function for testing from sysctl.c
// which delays 3*ulCount cycles
#ifdef __TI_COMPILER_VERSION__
       //Code Composer Studio Code
       void Delay(unsigned long ulCount){
       __asm (       "    subs    r0, #1\n"
                     "    bne     Delay\n"
                     "    bx      lr\n");
}

#else
```

```
//Keil uVision Code
__asm void
Delay(unsigned long ulCount)
{
	subs    r0, #1
	bne     Delay
	bx      lr
}
#endif
//-----------------------------------------------------------------------------------------
void PortG_Init(void)
{
	volatile unsigned long delay;
	SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOG; // activate port G

	delay = SYSCTL_RCGC2_R;
	delay = SYSCTL_RCGC2_R;
	delay = SYSCTL_RCGC2_R;

	GPIO_PORTG_DIR_R |= 0x04;             // make PG2 out (built-in LED)
  GPIO_PORTG_AFSEL_R &= ~0x04;          // disable alt func on PG2
  GPIO_PORTG_DEN_R |= 0x04;             // enable digital I/O on PG2
  PG2 = 0;                              // turn off LED
  GPIO_PORTG_DATA_R |= 0x04;

}
//-----------------------------------------------------------------------------------------

unsigned short plotPoints[100] = {2000};


int main(void)
{
  PLL_Init();      // 25 MHz Clock
	PortG_Init();    // Initialize the Heartbeat

	Output_Init();
  Output_Color(15);
  Delay(4000000);
	PG2 = 1;
	Delay(4000000);
	PG2 = 0;

  ADC_InitSWTriggerSeq3(0);     // allow time to finish activating ADC0
  Timer0A_Init100HzInt();       // set up Timer0A for 100 Hz interrupts
	//ADCvalue = 0;

	RIT128x96x4PlotClear(2000,4000,20,27,34,40);
	EnableInterrupts();
  while(1)
	{
    WaitForInterrupt();
  }
}
//-----------------------------------------------------------------------------------------
```

```c
// ADC.c

#include "ADC.h"
#include "lm3s1968.h"

volatile unsigned long ADCvalue = 0;;

//unsigned short const ADCdata[SIZE]={
//        0,2,27,53,79,107,135,165,196,228,262,
//     296,332,370,409,449,491,535,581,628,677,
//     728,781,837,894,954,1016,1023,1023,1023,1023,

//     1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,

//     1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1023,1024};

//              unsigned short const ADCdata[SIZE]={
//        0,2,27,53,79,107,135,165,196,228,262,
//     296,332,370,409,449,491,535,581,628,677,
//     728,781,837,894,954,1016,1,2,3,4,
//     5,6,7,8,9,10,11,12,13,14,
//     15,16,17,18,19,20,21,22,23,24,25,1024};

// 20-40
//------------------------------------------------------------------------------------
unsigned short const ADCdata[SIZE]={0,4,8,21,34,48,62,76,90,104,119,

     134,150,165,181,198,214,231,249,266,284,
     302,321,340,359,379,399,419,440,462,483,
     505,528,551,574,598,623,647,673,699,725,
     752,779,807,836,865,895,925,956,987,1020,1023,1024};


// 0-40
//unsigned short const Tdata[SIZE]={
//     4000,4000,3920,3840,3760,3680,3600,3520,3440,3360,3280,
//     3200,3120,3040,2960,2880,2800,2720,2640,2560,2480,
//     2400,2320,2240,2160,2080,2000,1920,1840,1760,1680,
//     1600,1520,1440,1360,1280,1200,1120,1040,960,880,
//     800,720,640,560,480,400,320,240,160,80,0,0};

//20-40
//------------------------------------------------------------------------------------
unsigned short const Tdata[SIZE]={4000,4000,3960,3920,3880,3840,3800,3760,3720,3680,3640,

     3600,3560,3520,3480,3440,3400,3360,3320,3280,3240,
     3200,3160,3120,3080,3040,3000,2960,2920,2880,2840,
     2800,2760,2720,2680,2640,2600,2560,2520,2480,2440,
     2400,2360,2320,2280,2240,2200,2160,2120,2080,2040,2000,2000};


//0-40
//unsigned short const Rdata[SIZE]={
//        512,512,530,548,567,587,608,629,652,675,700,725,
//        751,779,808,838,869,901,935,971,1008,1046,1086,
//        1128,1172,1218,1266,1316,1368,1423,1480,1540,1603,
//     1668,1737,1809,1884,1963,2046,2132,2223,2318,2418,
//     2523,2633,2748,2870,2997,3131,3271,3419,3574,3574};
```

```c
//20-40
//------------------------------------------------------------------------------------
unsigned short const Rdata[SIZE]={517,517,526,534,543,552,561,570,580,589,599,

     609,619,630,640,651,662,673,685,697,708,

     721,733,746,759,772,785,799,813,827,841,

     856,871,887,903,919,935,952,969,986,1004,

     1022,1041,1060,1079,1099,1119,1140,1161,1182,1204,1226,1226};
//------------------------------------------------------------------------------------
// There are many choices to make when using the ADC, and many
// different combinations of settings will all do basically the
// same thing.  For simplicity, this function makes some choices
// for you.  When calling this function, be sure that it does
// not conflict with any other software that may be running on
// the microcontroller.  Particularly, ADC sample sequencer 3
// is used here because it only takes one sample, and only one
// sample is absolutely needed.  Sample sequencer 3 generates a
// raw interrupt when the conversion is complete, but it is not
// promoted to a controller interrupt.  Software triggers the
// ADC conversion and waits for the conversion to finish.  If
// somewhat precise periodic measurements are required, the
// software trigger can occur in a periodic interrupt.  This
// approach has the advantage of being simple.  However, it does
// not guarantee real-time.
//
// A better approach would be to use a hardware timer to trigger
// the ADC conversion independently from software and generate
// an interrupt when the conversion is finished.  Then, the
// software can transfer the conversion result to memory and
// process it after all measurements are complete.

// This initialization function sets up the ADC according to the
// following parameters.  Any parameters not explicitly listed
// below are not modified:
// Max sample rate: <=125,000 samples/second
// Sequencer 0 priority: 1st (highest)
// Sequencer 1 priority: 2nd
// Sequencer 2 priority: 3rd
// Sequencer 3 priority: 4th (lowest)
// SS3 triggering event: software trigger
// SS3 1st sample source: programmable using variable 'channelNum' [0:7]
// SS3 interrupts: enabled but not promoted to controller
//------------------------------------------------------------------------------------

void ADC_InitSWTriggerSeq3(unsigned char channelNum)
{
  if(channelNum > 7)
      {
    return;   // 0 to 7 are valid channels on the LM3S1968
  }

  SYSCTL_RCGC0_R |= 0x00010000;    // 1) activate ADC clock
  SYSCTL_RCGC0_R &= ~0x00000300;  // 2) configure for 125K
  ADC_SSPRI_R = 0x3210;            // 3) Sequencer 3 is lowest priority
  ADC_ACTSS_R &= ~0x0008;          // 4) disable sample sequencer 3
  ADC_EMUX_R &= ~0xF000;          // 5) seq3 is software trigger
  ADC_SSMUX3_R &= ~0x0007;        // 6) clear SS3 field
  ADC_SSMUX3_R += channelNum;     //    set channel
  ADC_SSCTL3_R = 0x0006;          // 7) no TS0 D0, yes IE0 END0
  ADC_IM_R &= ~0x0008;            // 8) disable SS3 interrupts
  ADC_ACTSS_R |= 0x0008;          // 9) enable sample sequencer 3
}
//------------------------------------------------------------------------------------------
//------------ADC_InSeq3------------
// Busy-wait Analog to digital conversion
// Input: none
// Output: 10-bit result of ADC conversion
unsigned long ADC_InSeq3(void)
{
       unsigned long result;
  ADC_PSSI_R = 0x0008;             // 1) initiate SS3
  while((ADC_RIS_R&0x08)==0){};    // 2) wait for conversion done
  result = ADC_SSFIFO3_R&0x3FF;    // 3) read result
  ADC_ISC_R = 0x0008;              // 4) acknowledge completion
  return result;
}
//------------------------------------------------------------------------------------------
unsigned short ADC2Temp(unsigned short adcSample, int* index)
{
       int i;

       for(i = 0; i < SIZE; i++)
       {
              if(adcSample < ADCdata[i])
              {
                     break;
              }
       }
       *index = i-1;
       return Tdata[i];
}
//------------------------------------------------------------------------------------------
unsigned short interpolate(unsigned short rawADC, int i)
{
       int deltaADC;
       int deltaT;
       int scaleADC;
       int percentChange;
       int tempDiff;
       int newTemp;

       deltaADC = ADCdata[i+1] - ADCdata[i];
       scaleADC = rawADC - ADCdata[i]; // should be a + number always
       percentChange = (scaleADC*1000+(deltaADC/2))/deltaADC;

       deltaT = Tdata[i] - Tdata[i+1];
       tempDiff = (deltaT*percentChange+500)/1000;
       newTemp = Tdata[i]-tempDiff;

       return newTemp;
}
```