A) Objectives (1/2 page maximum)

B) Hardware Design

N/A

C) Software Design

1) Low level oLED driver (rit128x96x4.c and rit128x96x4.h files)

See valvano files for other functions.

```
//***************************************************************************|
//                                   |------------------|
                                                              |
//                                   |         EE345M Stuff   |
                                                 |
//                                   |------------------|         [Not written by TI]
                                                                                  |
//***************************************************************************|


//***************************************************************************
// Prints to OLED as a split screen
//
//output string or integer to top / bottom half of the screen
// Top = 1, bottom = 0
// works by printing the string, then printing the number where the string stopped printing
// no limit on length, will fall off screen
// if -0 given as integer 'value' will not be printed
//***************************************************************************

void oLED_Message(int device, int line, char *string, long value){
	char valueToString[21];
	//char combined[21];
	int stringLength;
	line=line%4; //limit line to value 0-3
	sprintf(valueToString,"%d",value); //convert integer to string
	stringLength = strlen(string);
	if(device == top){
		line=((line)*12);
		RIT128x96x4StringDraw(string , 0, line, 15);
		if(value != -0){
			RIT128x96x4StringDraw(valueToString , stringLength*COLUMWIDTH, line, 15);}
	} else if(device == bottom){
		line=(line*12)+48;
		RIT128x96x4StringDraw(string, 0, line, 15);
		if(value != -0){
			RIT128x96x4StringDraw(valueToString , stringLength*COLUMWIDTH, line, 15);}

	}//else printf("ERROR: Top/Bottom not specified");

	return;
}

//***************************************************************************
// Prints to OLED with scrolling
//
//
// works by printing the string, then printing the number where the string stopped printing
// no limit on length, will fall off screen
//
// store strings in global scrollLines[8][21] array, manipulates that to save lines between calls
//
//NOTE: not fancy, incredibly heavy and bloated, should be avoided unless absolutely necessary
//***************************************************************************



void oLED_Message_Scroll(char *string, long value){
	char valueToString[21];
	//char combined[21];
//	int stringLength;
	int x;
	int y;
	//line=line%8; //limit line to value 0-3
	sprintf(valueToString,"%d",value); //convert integer to string
	//stringLength = strlen(string);

	if((strlen(valueToString)+strlen(string))> 20){
		//scrollLines[0][]="Error: Line Too Long ";
		scrollLines[0][0]='E';
		scrollLines[0][1]='R';
		scrollLines[0][2]='R';
		scrollLines[0][3]=0;
	} else{

		for(x=7;x>0;x--){		//copy all lines up one line
			for(y=0;y<21;y++){
				scrollLines[x][y] = scrollLines[x-1][y];
			}
		}
		for(y=0;y<strlen(string);y++){	//write bottom line ;  text part
			scrollLines[0][y] = string[y];
			}
			for(y=strlen(string);y<strlen(string)+strlen(valueToString);y++){	//concattenate bottom
line : number part
				scrollLines[0][y]=valueToString[y-strlen(string)];
			}
		}
		for(x=0;x<8;x++){		//print it all out
			RIT128x96x4StringDraw(scrollLines[x] , 0, (98-(x*12)), 15);		//98 is used to make it
print bottom up, remote the '98-' and it will print top down

		}

		return;
}
```

2) Low level ADC driver (ADC.c and ADC.h files)

```
/**
Filename:	adc.c
Name:		Cruz Monrreal II, Austin Blackstone
Creation Date:  01/25/2012
Lab #:		1
TA:		Zahidul
Last Revision:  10/30/2012
Description:	Helper functions for Timer-based ADC operation
*/

#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/interrupt.h"

#include "lm3s1968.h"
#include "driverlib/adc.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"

#include "adc.h"

unsigned long adc_last_value;
unsigned int adc_samples;
unsigned char adc_status;
```

```
// ADC Interrupt Handler
void ADC0IntHandler(){
  // Clear Interrupt
  ADCIntClear(ADC0_BASE, 0);

  // Update value
  ADCSequenceDataGet(ADC_BASE, 3, &adc_last_value);

  // Check to see if we're done
  if (--adc_samples == 0){
    // Disable ADC Interrupts
    ADCIntDisable(ADC0_BASE, 0);

    // Update status
    adc_status = ADC_IDLE;
  }
}

// Initialize ADC w/ Timer0
void ADC_Init(unsigned long freq){
    // Enable Peripheral
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC);
    IntEnable(INT_ADC0);

    // ADC Pins are dedicated.
    // No need to get GPIO Type

    // Init ADC Timer  (Default @ 1KHz)
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
    TimerControlTrigger(TIMER0_BASE, TIMER_A, true);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 1000);
    TimerEnable(TIMER0_BASE, TIMER_A);

    // Set status
    adc_status = ADC_IDLE;
}

// Get status of ADC
unsigned short ADC_Status(){  return adc_status; }

// Internally start ADC
void ADC_Enable(unsigned int samples){
    // Set number of samples before finished
    adc_samples = samples;

    // Change status
    adc_status = ADC_BUSY;
```

```
   // Enable ADC Interrupts
   ADCIntEnable(ADC0_BASE, 0);
}

// Perform ADC on Channel
unsigned short ADC_Read(unsigned int channelNum){
   // Make sure channelNum is value 0 to 3
   ASSERT(channelNum < 4);

   // Setup ADC Sequence
   ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
   ADCSequenceStepConfigure(ADC0_BASE, 0, channelNum, channelNum | ADC_CTL_IE |
ADC_CTL_END);

   // Start Conversion
   ADC_Enable(1);

   // Wait for ADC to finish
     while (adc_status != ADC_IDLE);

   // Return value
   return (short) adc_last_value;
}

// Collect multiple samples from single ADC Channel
void ADC_Collect(unsigned int channelNum, unsigned int freq, unsigned short buffer[], unsigned int
samples){
  unsigned int i;

  // Check parameters
  ASSERT(channelNum < 4);
  ASSERT(freq >= 100);
  ASSERT(freq <= 10000);

  // Set channel to sample from
  ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
  ADCSequenceStepConfigure(ADC0_BASE, 0, channelNum, channelNum | ADC_CTL_IE | ADC_CTL_END);

  // Set Sample Frequency
  TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / freq);

  // Start conversions
  ADC_Enable(samples);

  // Collect samples as they become availble
  adc_last_value = ADC_SAMPLE_NOT_READY;
  for(i=0; i<samples; i++){
   // Idle until sample is ready
   while(adc_last_value == ADC_SAMPLE_NOT_READY);

   // Read new sample
   buffer[i] = (unsigned short) adc_last_value;

   // Reset adc_last_sample
   adc_last_value = ADC_SAMPLE_NOT_READY;

   /* Should not need this
   if (adc_status == ADC_IDLE)
     break;
   */
  }

  // Set status to finished
  adc_status = ADC_IDLE;
}
```

```
/**
Filename:      adc.h
Name:          Cruz Monrreal II, Austin Blackstone
Creation Date:  01/30/2012
Lab #:         1
TA:            Zahidul
Last Revision:  10/30/2012
Description:    Helper functions for Timer-based ADC operation
*/

#define ADC_IDLE 0
#define ADC_BUSY 1

// ADC Range: 0 to 1023
#define ADC_SAMPLE_NOT_READY 1024

// ADC Interrupt Handler
void ADC0IntHandler(void);

// Initialize ADC w/ Timer0
void ADC_Init(unsigned long freq);

// Get status of ADC
unsigned short ADC_Status(void);

// Internally start ADC
void ADC_Enable(unsigned int samples);

// Perform ADC on Channel
unsigned short ADC_Read(unsigned int channelNum);
```

```
// Collect multiple samples from single ADC Channel
void ADC_Collect(unsigned int channelNum, unsigned int freq, unsigned short buffer[], unsigned int
samples);
```

3) Low level timer driver (OS.c and OS.h files)

```
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/interrupt.h"

#include "lm3s1968.h"
#include "driverlib/timer.h"

unsigned long os_counter;
void(*func)(void) = NULL;

// Timer2 Interrupt Handler
void Timer2A_Handler()
{
   // Ack Timer2
   TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

   // Inc OS Counter
   os_counter++;

   // Run Timer Task
   func();
}

int OS_AddPeriodicThread(void(*task)(void) , unsigned long period, unsigned long priority){
   // Save
   func = task;

   // Reconfigure Timer2
   SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
   TimerConfigure(TIMER2_BASE, TIMER_CFG_32_BIT_PER);
   TimerControlTrigger(TIMER2_BASE, TIMER_A, true);
   TimerLoadSet(TIMER2_BASE, TIMER_A, period);
   TimerEnable(TIMER2_BASE, TIMER_A);

   // Priority?
}

void OS_ClearMsTime(void){
   // Reset timer counter
   os_counter; = 0;

   // Disable hardware timer
   TimerDisable(TIMER2_BASE, TIMER_A);
}

unsigned long OS_MsTime(void){ return os_counter; }

void dummy(void){}
```

```
// filename **********OS.H***********
// Real Time Operating System for Labs 2 and 3
// Jonathan W. Valvano 2/3/11, valvano@mail.utexas.edu
//***********Ready to go*************
// You may use, edit, run or distribute this file
// You are free to change the syntax/organization of this file

#ifndef __OS_H
#define __OS_H  1

// fill these depending on your clock
#define TIME_1MS  50000
#define TIME_2MS  2*TIME_1MS

// feel free to change the type of semaphore, there are lots of good solutions
struct  Sema4{
  int Value;   // >0 means free, otherwise means busy
// add other components here, if necessary to implement blocking
};
typedef struct Sema4 Sema4Type;

// ******** OS_Init ************
// initialize operating system, disable interrupts until OS_Launch
// initialize OS controlled I/O: serial, ADC, systick, select switch and timer2
// input:  none
// output: none
void OS_Init(void);

// ******** OS_InitSemaphore ************
// initialize semaphore
// input:  pointer to a semaphore
// output: none
void OS_InitSemaphore(Sema4Type *semaPt, long value);

// ******** OS_Wait ************
// decrement semaphore and spin/block if less than zero
// input:  pointer to a counting semaphore
// output: none
void OS_Wait(Sema4Type *semaPt);

// ******** OS_Signal ************
// increment semaphore, wakeup blocked thread if appropriate
// input:  pointer to a counting semaphore
// output: none
void OS_Signal(Sema4Type *semaPt);

// ******** OS_bWait ************
// if the semaphore is 0 then spin/block
// if the semaphore is 1, then clear semaphore to 0
// input:  pointer to a binary semaphore
// output: none
void OS_bWait(Sema4Type *semaPt);

// ******** OS_bSignal ************
// set semaphore to 1, wakeup blocked thread if appropriate
// input:  pointer to a binary semaphore
// output: none
void OS_bSignal(Sema4Type *semaPt);

//******** OS_AddThread ***************
// add a foregound thread to the scheduler
// Inputs: pointer to a void/void foreground task
//        number of bytes allocated for its stack
//        priority (0 is highest)
// Outputs: 1 if successful, 0 if this thread can not be added
// stack size must be divisable by 8 (aligned to double word boundary)
// In Lab 2, you can ignore both the stackSize and priority fields
// In Lab 3, you can ignore the stackSize fields
int OS_AddThread(void(*task)(void),
   unsigned long stackSize, unsigned long priority);

//******** OS_Id ***************
// returns the thread ID for the currently running thread
// Inputs: none
// Outputs: Thread ID, number greater than zero
unsigned long OS_Id(void);

//******** OS_AddPeriodicThread ***************
// add a background periodic task
// typically this function receives the highest priority
// Inputs: pointer to a void/void background function
//        period given in system time units
//        priority 0 is highest, 5 is lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// It is assumed that the user task will run to completion and return
// This task can not spin, block, loop, sleep, or kill
// This task can call OS_Signal  OS_bSignal          OS_AddThread
// You are free to select the time resolution for this function
// This task does not have a Thread ID
// In lab 2, this command will be called 0 or 1 times
// In lab 2, the priority field can be ignored
// In lab 3, this command will be called 0 1 or 2 times
// In lab 3, there will be up to four background threads, and this priority field
//         determines the relative priority of these four threads
int OS_AddPeriodicThread(void(*task)(void),
   unsigned long period, unsigned long priority);

//******** OS_AddButtonTask ***************
// add a background task to run whenever the Select button is pushed
// Inputs: pointer to a void/void background function
//        priority 0 is highest, 5 is lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// It is assumed that the user task will run to completion and return
// This task can not spin, block, loop, sleep, or kill
// This task can call OS_Signal  OS_bSignal          OS_AddThread
// This task does not have a Thread ID
// In labs 2 and 3, this command will be called 0 or 1 times
// In lab 2, the priority field can be ignored
// In lab 3, there will be up to four background threads, and this priority field
//         determines the relative priority of these four threads
int OS_AddButtonTask(void(*task)(void), unsigned long priority);

//******** OS_AddDownTask ***************
// add a background task to run whenever the Down arror button is pushed
// Inputs: pointer to a void/void background function
//        priority 0 is highest, 5 is lowest
// Outputs: 1 if successful, 0 if this thread can not be added
// It is assumed user task will run to completion and return
// This task can not spin block loop sleep or kill
// It can call issue OS_Signal, it can call OS_AddThread
// This task does not have a Thread ID
// In lab 2, this function can be ignored
// In lab 3, this command will be called will be called 0 or 1 times
// In lab 3, there will be up to four background threads, and this priority field
//         determines the relative priority of these four threads
int OS_AddDownTask(void(*task)(void), unsigned long priority);

// ******** OS_Sleep ************
// place this thread into a dormant state
// input:  number of msec to sleep
// output: none
// You are free to select the time resolution for this function
// OS_Sleep(0) implements cooperative multitasking
```

```
void OS_Sleep(unsigned long sleepTime);

// ******** OS_Kill ************
// kill the currently running thread, release its TCB memory
// input:  none
// output: none
void OS_Kill(void);

// ******** OS_Suspend ************
// suspend execution of currently running thread
// scheduler will choose another thread to execute
// Can be used to implement cooperative multitasking
// Same function as OS_Sleep(0)
// input:  none
// output: none
void OS_Suspend(void);

// ******** OS_Fifo_Init ************
// Initialize the Fifo to be empty
// Inputs: size
// Outputs: none
// In Lab 2, you can ignore the size field
// In Lab 3, you should implement the user-defined fifo size
// In Lab 3, you can put whatever restrictions you want on size
//    e.g., 4 to 64 elements
//    e.g., must be a power of 2,4,8,16,32,64,128
void OS_Fifo_Init(unsigned long size);

// ******** OS_Fifo_Put ************
// Enter one data sample into the Fifo
// Called from the background, so no waiting
// Inputs:  data
// Outputs: true if data is properly saved,
//          false if data not saved, because it was full
// Since this is called by interrupt handlers
//  this function can not disable or enable interrupts
int OS_Fifo_Put(unsigned long data);

// ******** OS_Fifo_Get ************
// Remove one data sample from the Fifo
// Called in foreground, will spin/block if empty
// Inputs:  none
// Outputs: data
unsigned long OS_Fifo_Get(void);

// ******** OS_Fifo_Size ************
// Check the status of the Fifo
// Inputs: none
// Outputs: returns the number of elements in the Fifo
//         greater than zero if a call to OS_Fifo_Get will return right away
//         zero or less than zero if the Fifo is empty
//         zero or less than zero  if a call to OS_Fifo_Get will spin or block
long OS_Fifo_Size(void);

// ******** OS_MailBox_Init ************
// Initialize communication channel
// Inputs:  none
// Outputs: none
void OS_MailBox_Init(void);

// ******** OS_MailBox_Send ************
// enter mail into the MailBox
// Inputs:  data to be sent
// Outputs: none
// This function will be called from a foreground thread
// It will spin/block if the MailBox contains data not yet received
void OS_MailBox_Send(unsigned long data);

// ******** OS_MailBox_Recv ************
// remove mail from the MailBox
// Inputs:  none
// Outputs: data received
// This function will be called from a foreground thread
// It will spin/block if the MailBox is empty
unsigned long OS_MailBox_Recv(void);

// ******** OS_Time ************
// reads a timer value
// Inputs:  none
// Outputs: time in 20ns units, 0 to max
// The time resolution should be at least 1us, and the precision at least 12 bits
// It is ok to change the resolution and precision of this function as long as
//   this function and OS_TimeDifference have the same resolution and precision
unsigned long OS_Time(void);

// ******** OS_TimeDifference ************
// Calculates difference between two times
// Inputs:  two times measured with OS_Time
// Outputs: time difference in 20ns units
// The time resolution should be at least 1us, and the precision at least 12 bits
// It is ok to change the resolution and precision of this function as long as
//   this function and OS_Time have the same resolution and precision
unsigned long OS_TimeDifference(unsigned long start, unsigned long stop);

// ******** OS_ClearMsTime ************
// sets the system time to zero from Lab 1)
// Inputs:  none
// Outputs: none
// You are free to change how this works
void OS_ClearMsTime(void);

// ******** OS_MsTime ************
// reads the current time in msec (from Lab 1)
// Inputs:  none
// Outputs: time in ms units
// You are free to select the time resolution for this function
unsigned long OS_MsTime(void);

//******** OS_Launch ***************
// start the scheduler, enable interrupts
// Inputs: number of 20ns clock cycles for each time slice
//        you may select the units of this parameter
// Outputs: none (does not return)
// In Lab 2, you can ignore the theTimeSlice field
// In Lab 3, you should implement the user-defined TimeSlice field
void OS_Launch(unsigned long theTimeSlice);

#endif
```

4) High level main program (the interpreter)

```
/* Init code here */
//
   // Prompt for text to be entered.
   //
   //UARTSend((unsigned char *)"$ ", 12);
            OutFifo_Put('$');                     //command line character
            OutFifo_Put(' ');
            UARTCharPutNonBlocking(UART0_BASE, '$');
   //
   // Loop forever echoing data through the UART.
   //
   while(1)
   {
                     while(InFifo_Get(&test)){
         // Echo to terminal
         OutFifo_Put(test);

         // Parse input
         switch(test){
            case '\r':
               i=0;
```

```
          // Generate String
          while(CmdFifo_Get(&tmp))
            str[i++]=tmp;

          str[i]='\n'; // Terminalte String

           // If/Else Blocks for parsing commands
          if (strstr(str, "echo")){
           // Remove 'echo' from command
           strncpy(str, str+5, strlen(str)-4);

           // Echo string to oLED
           oLED_Message(bottom, 0, str, -0);

           // New Line
           OutFifo_Put('\r');
           OutFifo_Put('\n');

           i=0;
           while((tmp=str[i++]) != '\n')
             OutFifo_Put(tmp);
          }


          // New Line
          OutFifo_Put('\r');
          OutFifo_Put('\n');

          // Reset String
          memset(str, 0, 32);

          // Prompt user for more input
                OutFifo_Put('$');
                OutFifo_Put(' ');

          break;
        default:
          CmdFifo_Put(test);
      }

               UARTIntHandler();
               }
        }
```

D) Measurement Data

1) Estimated time to run the periodic timer2 interrupt

10 us

2) Measured time to run the periodic timer2 interrupt

4.43 us

E) Analysis and Discussion (1 page maximum) This section will consist of explicit answers to these questions

1) What are the range, resolution, and precision of the ADC?

The ADC on the LM3s8962 is a 10 bit resolution ADC that has a range from 0 to 3 volts and 0.0029296875 v/bit precision.

2) List the ways the ADC conversion can be started. Explain why you choose the way you did.

An ADC conversion is capable of being started via software, hardware timers, capture-compare interrupts, or through a sequence. For this lab, we chose to use hardware timers to ensure that the ADC was steadily capturing samples, and when the user wanted to request data, there would be very little delay, as the ADC would already be in the middle of a capture. Using the hardware timer also allowed us flexibility that would be needed for the next lab.

3) The measured time to run the periodic interrupt can be measured directly by setting a bit high at the start of the ISR and clearing that bit at the end of the ISR. It could also be measured indirectly by measuring the time lost when running a simple main program that toggles an output pin. How did you measure it? Compare and contrast your method to these two.

We set and unset the status led, while also probing the same signal. This allowed us to get a visual representation of the signal, as well as a logic signal. We used a logic analyzer to measure how long the led was turned off. Measuring the duration off in the main program would have been less invasive, but more difficult to achieve, as opposed to setting and clearing a signal.

4) Divide the time to execute once instance of the ISR by the total instructions in the ISR it to get the average time to execute an instruction. Compare this to the 20 ns system clock period (50 MHz).

At 19 assembly instructions observed in the debugger, each instruction took approximately 230 ns to execute. Unless we're clocked at the wrong frequency (8MHz instead of 50MHz), we might have measured wrong.

5) What are the range, resolution, and precision of the SysTick timer? I.e., answer this question relative to the NVIC_ST_CURRENT_R register in the Cortex M3 core peripherals.