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.