EE445M – Lab 1: Graphics, LCD, Timer and Interpreter

Kristian Doggett and Akshar Patel

1/13/15

A. Objectives

Goals:
Refreshing knowledge of TM4C123 Launchpad
Interrupt driven UART
TFT driver for two logical displays
Multi-channel ADC driver
Periodic interrupts with GPTimer
Build command line interpreter

Summary:

In lab one, we are refamiliarizing ourselves with our development board, the TM4C123 Lanchpad, by designing basic uController functions such as ADC, UART, and timers. Additionally, we are laying a framework for future labs by designing a command line interpreter and adding drivers for our TFT screen. We are enhancing our prior ADC drivers to allow for multiple channels and specifying sampling rates and number of samples dynamically. Instead of busy-wait, our UART driver now uses interrupts, which allows for future development of data exchanges using FIFOs. We measured the ISR times of our general purpose timer so we can get an idea of the overhead for ISRs in general. Finally, we separated our TFT into two logical displays in anticipation of more advanced graphics capabilities and built the foundation of a CLI to interact with future systems.

B. Hardware Design

None for Lab 1.

C. Software Design

1. LCD Driver
2. ADC Driver
3. Timer Driver
4. Interpreter

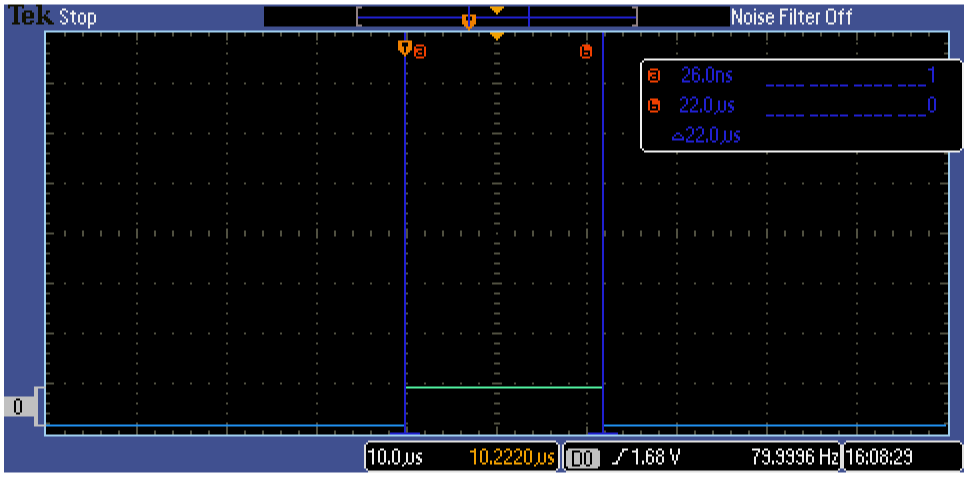
All code is included at end of report.

D. Measurement Data

1. Estimated time of periodic interrupt:

(# of instructions in ISR handler) * 12.5 ns = 16 * 12.5 ns = .2 us

2. Measured time to run periodic interrupt: 22 us



E. Analysis and Discussion

1. ADC Range/Resolution/Precision

   a. Range (max/min ADC inputs): 0 to 3 V
   b. Precision (number of distinguishable inputs): 12 bits
   c. Resolution (smallest distinguishable change): 1 mV

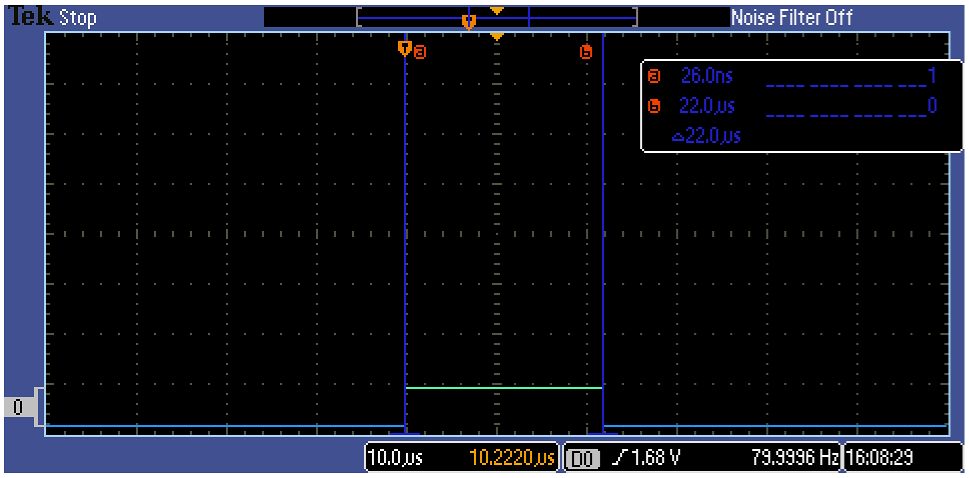
2. List the ways the ADC conversion can be started. Explain why you choose the way you did.

   Sample triggering for each sample sequencer is defined in the ADC Event Multiplexer Select
   (ADCEMUX) register.

   Trigger sources include:
   1. processor (default)
   2. analog comparators
   3. external signal on a GPIO
   4. GP Timer
   5. PWM generator
   6. continuous sampling

3. Measured time to run periodic interrupt.

We used the method of toggling a bit within the ISR. Out measured time to run the ISR was 22 us. We did not use the alternate method of calculating time lost because it was simpler to calculate ISR time.



4. There is a large discrepancy in our measured time versus estimated time.

We measured 22 us and assuming the ISR takes 16 instructions:

time to execute ISR/# of instructions = 22us/16 = 1 us bus speed, which is clearly incorrect.

Assuming clock is executing an instruction every 12.5 ns:

# of instructions = time to execute ISR/system clock = 22 us/12.6 ns = 1,760 instructions, which is clearly not correct.

This means we are either measuring the ISR time incorrectly or estimating the number of instructions in correctly. Most likely measuring the ISR time incorrectly since the calculated instruction number of 1,760 makes our measured ISR time off by two orders of magnitude.

If the ISR uses 16 instructions with a bus speed of 12.5 ns we would anticipate the ISR time as

ISR time = bus speed * # of instructions = 12.5 ns * 16 = .2 us

Estimated instructions: 16
Calculated instructions: 1,760

Measured ISR time: 22 us

Calculated ISR time based on 16 instructions: .2 us

We were discussing this on Piazza (https://piazza.com/class/i4ye1tm4y9s4ve?cid=15) but never resolved it by the time of the report.

5. What is range, resolution, and precision of SysTick timer?

   Range: 0 to 2^24
   Resolution: 12.5 ns
   Precision: 24 bits

PreLab 1

1.

a.	The following are lines of code that determine which UART to activate. You must also activate the GPIO port corresponding to that UART.

```
  SYSCTL_RCGCUART_R |= 0x01;           // activate UART0
  SYSCTL_RCGCGPIO_R |= 0x01;           // activate port A
```

b.	The following determines the baud rate, parity, data bits, and stop bits.

```
#define UART_LCRH_WLEN_8        0x00000060  // 8 bit word length
#define UART_LCRH_FEN           0x00000010  // UART Enable FIFOs

UART0_IBRD_R = 27;                  // IBRD = int(50,000,000 / (16 * 115,200)) =
int(27.1267)
UART0_FBRD_R = 8;                   // FBRD = int(0.1267 * 64 + 0.5) = 8
                                    // 8 bit word length (no parity bits, one stop bit, FIFOs)
UART0_LCRH_R = (UART_LCRH_WLEN_8|UART_LCRH_FEN);
```

c.	The port pins vary by UART used. In this example UART0 uses PA0 to receive and PA1 to transmit.

d.	The following code inputs one byte from the UART port.

```
// input ASCII character from UART
// spin if RxFifo is empty
char UART_InChar(void){
  char letter;
  while(RxFifo_Get(&letter) == FIFOFAIL){};
  return(letter);
}
```

e.	The following code outputs one byte to the UART port.

```
// output ASCII character to UART
// spin if TxFifo is full
void UART_OutChar(char data){
  while(TxFifo_Put(data) == FIFOFAIL){};
  UART0_IM_R &= ~UART_IM_TXIM;         // disable TX FIFO interrupt
  copySoftwareToHardware();
  UART0_IM_R |= UART_IM_TXIM;           // enable TX FIFO interrupt
}
```

f. The system defines the ISR vector in the Startup.s file. Each exception has a 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory. All interrupts are enabled by setting the CPSIE bit.

g. The TM4C123 has large 16 byte hardware FIFOs like the PC.

2.

a. writedata uses Busy-Wait Synchronization.

```
void static writedata(uint8_t c) {
  while((SSI0_SR_R&SSI_SR_TNF)==0){};   // wait until transmit FIFO not full
  DC = DC_DATA;
  SSI0_DR_R = c;                        // data out
}
```

b. ST7735_DrawChar has five parameters:

x - horizontal of the top left corner of the character
y - vertical position of the top left corner of the character
c - character to be printed
textColor - 16-bit color of character
bgColor - 16-bit color of background
size - number of pixels per character pixel

To use this function, you pass the x,y position where you want the top left of the character to begin, the character to print, the color you would like to set the character, the color you would like to set as the background behind the character, and the size. You must keep in mind that x,y position is based on columns and the size parameter prints each pixel in the designated size.

c. Below are the pin assignments for two different versions of the ST7735 we are using.

```
/* Sain Smart Connections (ST7735)

            Name        Pin         Board
            VCC         1           +3.3V
```

```
		GND		2		GND

		SCL		3		PA2 (SSI0Clk)
		SDA		4		PA5 (SSI0Tx)
		RS/DC		5		PA6 (GPIO)
		RES		6		PA7 (GPIO)
		CS		7		PA3 (SSI0Fss)

		Pins below are for the SD-Card:
		MSO		8		nc
		SCLK		9		nc
		MOSI		10		nc
		CS		11		nc

		http://www.sainsmart.com/zen/documents/20-011-920/Manual.pdf
*/

/* Ada Fruit Connections (ST7735)

		Name			Pin		Board

		Backlight		10		+3.3V
			MISO			9		nc
		SCK			8		PA2 (SSI0Clk)
		MOSI			7		PA5 (SSI0Tx)
		TFT_CS			6		PA3 (SSI0Fss)
		CARD_CS			5		nc
		Data/Command		4		PA6 (GPIO)
		RESET			3		PA7 (GPIO)
		VCC			2		+3.3V

		Gnd			1		GND

*/
```

d. PA2, PA5, and PA3 are only used for GPIO and the SSI function. One can use PA6 and PA7 as I2C and/or PWM functionality.

3.

a. The following code defines the period of the SysTick interrupt. PLL_Init() will define user specified clock speed, based on that, you will call SysTick_Init and pass it a unsigned 32 bit value which is labeled as period which will then be used to set the reload value:

```
  NVIC_ST_RELOAD_R = period-1;           // reload value
```

b. The clock is set by calling the PLL_Init() function. Specifically, the system clock is determined by the SYSDIV2 filed in the SYSCTL_RCC2_R register. Based on which number you select SYSDIV2 to be, the developer can change the clock cycle.

c. The SysTick interrupt is triggered when the CURRENT value counts down from 1 to 0, which sets the COUNT flag. The interrupt does not need to be acknowledged because it automatically reloads the CURRENT value and continues decrementing.

4.

a. The first assembly instruction is PUSH and the final instruction is BX LR.

b. It pushes eight registers on the stack: R0-R3, R12, LR, PC, PSR. R0 is on top and more registers are pushed if floating point is enabled.

c. When the ISR is complete, the LR contains a special value that that indicates a return from interrupt. Because the LR contains the special value, the BX LR instruction pulls the eight registers from the stack and returns control to the main program.

```
________________________________________LCD DRIVER________________________________________

//-----------ST7735_Message--------------
//Draws a string and long value one one of the two split screens
//Each logically seperate screen contains four lines
//Used for interaction with the CLI, debugging, and displaying data

void ST7735_Message(int screen,int line,char *string,long value){

    if(screen == 0){
        if(line == 0){
            ST7735_FillRect(0,0,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 0);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
        else if(line == 1){
            ST7735_FillRect(0,20,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 2);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
        else if(line == 2){
            ST7735_FillRect(0,40,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 4);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
        else if(line == 3){
            ST7735_FillRect(0,60,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 6);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
    }
    else if(screen == 1){
        if(line == 0){
            ST7735_FillRect(0,90,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 9);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
        else if(line == 1){
            ST7735_FillRect(0,110,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 11);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
        else if(line == 2){
            ST7735_FillRect(0,130,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 13);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
        else if(line == 3){
            ST7735_FillRect(0,150,128,8,ST7735_BLACK);
            ST7735_SetCursor(0, 15);
            printf("%s",string);
            ST7735_OutUDec(value);
        }
    }
}

______________________________________ADC DRIVER______________________________________

AddIndexFifo(ADCBuffer, 1000, uint32_t, FIFOSUCCESS, FIFOFAIL)

volatile uint32_t ADCvalue;

void ADC0Seq3_Handler(void){
  ADC0_ISC_R = 0x08;          // acknowledge ADC sequence 3 completion
    ADCvalue = ADC0_SSFIFO3_R;
    ADCBufferFifo_Put(ADCvalue);
}

uint32_t ADC_In(void){
    return ADCvalue;
}

uint16_t* ADC_Collect(uint32_t channelNum, uint32_t fs, uint16_t buffer[], uint32_t 
numberOfSamples) { int i = 0;
    ADCBufferFifo_Init();
    uint32_t value;
    uint32_t period = 0;
    period = (80000000 / fs);                     // Divide clock cycel by the specified frequency
    ADC_Open(channelNum, period);             //Use ADC_Open to properly open up the specified 
    channel at the designmated frequency
    while(ADCBufferFifo_Size() != numberOfSamples){}
    uint32_t counter;
    for(counter = 0; counter < numberOfSamples; counter++){
        buffer[counter] = ADCBufferFifo_Get(&value);
    }
    return buffer;
}

______________________________________TIMER DRIVER______________________________________


int OS_AddPeriodicThread(void(*task)(void),unsigned long period,unsigned long priority){
    SYSCTL_RCGCTIMER_R |= 0x20;
    PeriodicTask = task;
    TIMER5_CTL_R = 0x00;                    //disable during setup
    TIMER5_CFG_R = 0x00;                    //32 bit mode
    TIMER5_TAMR_R = 0x02;                   //periodic mode
    TIMER5_TAILR_R = period - 1;    //requested reload value
    TIMER5_TAPR_R = 0x00;                   //bus clock resolution, no prescale
    TIMER5_ICR_R = 0x01;                    //clear timeout flag
    TIMER5_IMR_R = 0x01;                    //arm timeout interrupt
    NVIC_PRI23_R = (NVIC_PRI23_R&0xFFFFFF00) | 0x80; //priority 4   (need to change priority,
    maybe left shift 5 times?)
    NVIC_EN2_R = 0x10000000;            //enable IRQ 92
    TIMER5_CTL_R = 0x01;                    //enable timer 5A
    EnableInterrupts();
    return 0;
}

void Timer5A_Handler(void){
    TIMER5_ICR_R = 0x01;                    //acknowledge timeout
    PF1 = PF1^0x02;             // toggle red LED, PF1
    (*PeriodicTask)();
    PF1 = PF1^0x02;             // toggle red LED, PF1
}

void OS_ClearPeriodicTime(void){
    TIMER5_TAILR_R = 0;                     //resets counter to 0, TAILR register
}

unsigned long OS_ReadPeriodicTimer(void){
    return TIMER5_TAILR_R;
}

______________________________________INTERPRETER DRIVER______________________________________

void ProcessCommand(char *command){
    char commandType[COMMAND_MAX];
// Initialize commandType buffer
    for(int j = 0; j < COMMAND_MAX; j++) {
        commandType[j] = 0;
    }
    uint32_t i = 0;
    char commandNum;
    while(1) {
        if(command[i] == ' ')
        {
            break;
        }
        else if(command[i] == NULL) {
            break;
        }
        else {
            commandType[i] = command[i];
        }
        i++;
    }
    if (strcmp(commandType,"ADC") == 0){
        commandType[i] = ' ';
        i++;
        commandNum = 1;
    }
    if (strcmp(commandType,"Timer") == 0){
        commandType[i] = ' ';
        i++;
        commandNum = 2;
    }
    if (strcmp(commandType,"LCD") == 0){
        commandType[i] = ' ';
        i++;
        commandNum = 3;
    }
        switch(commandNum){
            case 1:
                uint32_t ADCValue = ADC_In();
                ST7735_Message(1,3,commandType, ADCValue);
                UART_OutUDec(ADCValue);
                break;
            case 2:
                while(command[i] != 0){
                    commandType[i] = command[i];
                    i++;
                }
                UART_OutString(commandType);
                break;
            case 3:
                while(command[i] != 0){
                    commandType[i] = command[i];
                    i++;
                }
                UART_OutString(commandType);
                ST7735_Message(0,3,commandType,0);
                break;
            default:
                ST7735_Message(2,1,"Default",1);
                UART_OutString(commandType);
                break;
        }
        i = 0;
}
```