**Lab 1: Graphics, LCD, ADC, Timer and Interpreter**

**Objectives**

The main goal in this lab is to familiarize ourselves with the Launchpad board, it’s ARM Cortex-M4 microprocessor, and the µVision development system. To accomplish this, there will be several tasks. Task 1 is to implement an interpreter that accepts command-line inputs through the UART serial port. Task 2 is to implement the LCD drivers, including a multi-device simulator that splits the screen into two parts. Task 3 is to implement an ADC using a single input pin. This ADC will take in values ranging from 0V to 3.3V. Task 4 is the final task, which is to implement a system time driver using timer interrupts. This driver will be able to run other functions by passing in a function pointer.

**Software Design**

This section contains relevant code to the project.

LCD Driver – ST7735.h

LCD Driver – ST7735.c

ADC Driver – ADC.h





ADC Driver – ADC.c







Timer Driver – OS.h



Timer Driver – OS.c



High Level Main Program

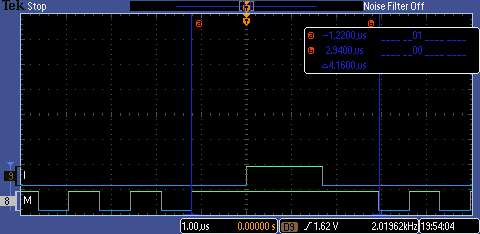




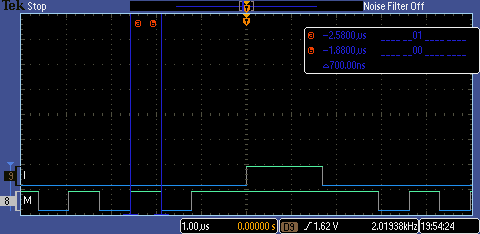
**Measurement Data**

The estimated time to run the periodic timer interrupt is .575 µs. This was calculated by taking the number of assembly instructions, multiplying them by a factor of two to estimate the instruction processing time, then adding cycles for push and pop of the registers, then multiplying that sum by 12.5 ns.

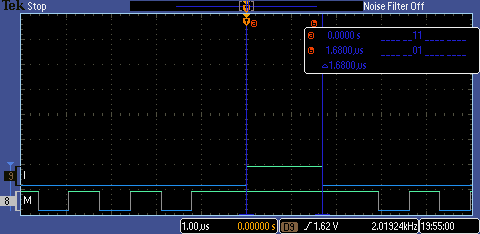
The actual time to run the periodic timer interrupt is 3 µs. This was measured by viewing the port toggling done by the main function and the interrupt routine and subtracting them.



Time spent between edges during an interrupt.



Duration of a single toggle.



Duration of the interrupt’s toggle.

**Analysis and Discussion**

The range of the ADC is 0V to 3.3V. The resolution is 0.0008. The precision is 4096.

To start the ADC conversion, we can use a timer that triggers the collection of data for conversion or we can continually run the collection in the main and hold the data for outputting. We used Timer2 to periodically sample the ADC to allow our interpreter to run continually while being ready to sample the ADC if a command is given, for instance.

We measured the time to run the periodic interrupt by having the main toggle a pin. Using the toggle and an oscilloscope, we could observe where the interrupt was happening and the effect it had on the main toggling routine. Finally, we subtracted the time for a normal toggle to occur (half a period) from the main method’s wait time plus the rest of the cycle, and that gave us the time for the interrupt to run. This is different than measuring the direct toggle, because when measuring the toggle of the pin inside the ISR it does not consider the pushes and pops required to switch subroutines.

The total number of instructions in the ISR is 13 while the run time was 3 µs. Dividing the time by the number of instructions gives 230.8 ns. This is a lot higher than the 12.5 ns system clock period, but not surprising as this calculation does not take into account multiple clock cycles required for some instructions as well as the pushes and pops at the beginning and end of the interrupt.

The range of SysTick is 0 to 16777215, as specified by the NVIC\_ST\_RELOAD\_M definition, which is the maximum reload value allowed. The precision is the difference between the actual time SysTick is triggered vs. the expected time. The resolution is the time between each SysTick interrupt.

**Preprep Questions**

2a. SYSCTL\_RCGCGPIO\_R |= 0x01; // activate port A

2b. UART0\_IBRD\_R = 43; // IBRD = int(80,000,000 / (16 \* 115,200)) = int(43.403)

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);

2c. Pins PA1 and PA0. PA1 transmits while PA0 receives.

2d. char UART\_InChar(void){

char letter;

while(RxFifo\_Get(&letter) == FIFOFAIL){};

return(letter);

}

2e. 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

}

2f. Vectors are set based on the interrupt handler. The interrupt handler corresponds to a device, in this case UART0.

2g. When an interrupt is triggered, the first step is to acknowledge the interrupt. This is done by clearing the corresponding interrupt flag.

2h. The TM4C123 has a hardware FIFO buffer.

3a. Busy-wait (wait while FIFO is full, then write)

3b. X and Y are the coordinates in pixels of the top left corner of the character to draw. C is the character. Then you can set the color of the character itself, then the color of the rectangular space encapsulating the character, then the size of the character itself in pixels per character pixel.

3c. Backlight (pin 10) connected to +3.3 V

MISO (pin 9) unconnected

SCK (pin 8) connected to PA2 (SSI0Clk)

MOSI (pin 7) connected to PA5 (SSI0Tx)

TFT\_CS (pin 6) connected to PA3 (SSI0Fss)

CARD\_CS (pin 5) unconnected

Data/Command (pin 4) connected to PA6 (GPIO), high for data, low for command

RESET (pin 3) connected to PA7 (GPIO)

VCC (pin 2) connected to +3.3 V

Gnd (pin 1) connected to ground

3d. Any devices that use the SS interface

4a. NVIC\_ST\_RELOAD\_R = period-1;// reload value

4b. PLL\_Init(Bus80MHz); // set system clock to 80 MHz

4c. STCURRENT resets the counter.

5a. First instruction is to acknowledge the flag and the last instruction is returning to the main thread.

5b. Saves registers R4-R11 to the stack.

5c. Pops those registers back off the stack.