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

Sourabh Shirhatti and Nelson Wu, Spring 2015

**Objectives**

The objective of this lab ­­was to introduce us to the Tiva TM4C123 LaunchPad and some of its capabilities. We used an interrupting serial port (employing both hardware and software FIFOs) and a command line interpreter to issue commands to the ADC and LCD. A graphics LCD driver split the display into two segments to be accessed by different threads in future labs. Our ADC driver could be triggered by both software and timer interrupts, providing for flexibility. Periodic timer interrupts were developed for use in thread scheduling.

**Hardware Design**

None

**Software Design**

We have only included the portions of the files we modified. We also submitted a .zip.

1. Low level LCD driver (ST7735\_.c and ST7735\_.h files)

//------------ST7735\_MessageString------------

// Divide the LCD into two logical partitions and provide

// an interface to output a string

// inputs: device specifies top(0) or bottom(1)

// line specifies line number (0-7)

// string pointer to NULL-terminated ASCII string

// outputs: none

void ST7735\_MessageString (int device, int line, unsigned char \*string) {

// Sanitize inputs

if (device < 0 || device > 1) return;

if (line <0 || line > 7) return;

Output\_Color(((device)? ST7735\_RED: ST7735\_YELLOW));

// Move cursor

ST7735\_SetCursor(0, (device \* 8) + line);

// Output

ST7735\_OutString(string);

}

//------------ST7735\_MessageInteger------------

// Divide the LCD into two logical partitions and provide

// an interface to output a string

// inputs: device specifies top(0) or bottom(1)

// line specifies line number (0-7)

// value 32-bit number in unsigned decimal format

// outputs: none

void ST7735\_MessageInteger (int device, int line, long value){

// Sanitize inputs

if (device < 0 || device > 1) return;

if (line < 0 || line > 7) return;

Output\_Color(((device)? ST7735\_RED: ST7735\_YELLOW));

// Move cursor

ST7735\_SetCursor(0, (device \* 8) + line);

// Output

ST7735\_OutUDec(value);

}

//------------ST7735\_MessageString------------

// Divide the LCD into two logical partitions and provide

// an interface to output a string

// inputs: device specifies top(0) or bottom(1)

// line specifies line number (0-7)

// string pointer to NULL-terminated ASCII string

// outputs: none

void ST7735\_MessageString (int device, int line, unsigned char \*string);

//------------ST7735\_MessageInteger------------

// Divide the LCD into two logical partitions and provide

// an interface to output a string

// inputs: device specifies top(0) or bottom(1)

// line specifies line number (0-7)

// value 32-bit number in unsigned decimal format

// outputs: none

void ST7735\_MessageInteger (int device, int line, long value);

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

//------------ADC0\_Open------------

// Set up a ADC channel for SW triggering using

// sequencer 3

// inputs: channelNum ADC channel number

// outputs: none

void ADC0\_Open(uint8\_t channelNum) {

// Open the specified channel with SW triggering

ADC0\_InitSWTriggerSeq3(channelNum);

ChannelNumber = channelNum;

}

//------------ADC0\_In------------

// Read ADC value on the open channel using SW triger

// inputs: none

// outputs: -1 Channel hasn't been opened

// -2 Channel in use by timer

int16\_t ADC\_In(void) {

// Read a sample on the open channel using SW triggering

int16\_t result;

// If channel hasn't been opened return error

if (ChannelNumber == -1) return -1;

// Ensure channel is not being used by timer

if (ChannelNumber == TimerChannelNumber) return -2;

ADC0\_PSSI\_R = 0x0008; // 1) initiate SS3

while((ADC0\_RIS\_R&0x08)==0){}; // 2) wait for conversion done

// if you have an A0-A3 revision number, you need to add an 8 usec wait here

result = ADC0\_SSFIFO3\_R&0xFFF; // 3) read result

ADC0\_ISC\_R = 0x0008; // 4) acknowledge completion

return result;

}

//------------ADC0\_Collect------------

// Sample ADC at specified Fs and store into a buffer

// sequencer 3

// inputs: channelNum ADC channel number

// Fs Sampling frequency

// buffer Name of buffer to store output

// numberOfSample Number of samples requested

// outputs: 1 Requested succeeded

// 0 Timer already in use

int16\_t ADC\_Collect(uint8\_t channelNum, uint32\_t Fs, uint16\_t buffer[], uint16\_t numberOfSamples) {

// check if timer not in use

if (channelNum == TimerChannelNumber)

return 0;

// If channelNum is closed, open it

if (ChannelNumber == -1) ChannelNumber = channelNum;

// save buffer name

BufferName = buffer;

NumberSamples = numberOfSamples;

TimerChannelNumber = channelNum;

// Switch to timer triggering

ADC0\_InitTimer0ATriggerSeq3(channelNum, (50000000/Fs));

return 0;

}

//------------ADC0\_Open------------

// Set up a ADC channel for SW triggering using

// sequencer 3

// inputs: channelNum ADC channel number

// outputs: none

void ADC0\_Open(uint8\_t channelNum);

//------------ADC0\_In------------

// Read ADC value on the open channel using SW triger

// inputs: none

// outputs: -1 Channel hasn't been opened

// -2 Channel in use by timer

int16\_t ADC\_In(void);

//------------ADC0\_Collect------------

// Sample ADC at specified Fs and store into a buffer

// sequencer 3

// inputs: channelNum ADC channel number

// Fs Sampling frequency

// buffer Name of buffer to store output

// numberOfSample Number of samples requested

// outputs: 1 Requested succeeded

// 0 Timer already in use

int16\_t ADC\_Collect(uint8\_t channelNum, uint32\_t Fs, uint16\_t buffer[], uint16\_t numberOfSamples);

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

//\*\*\*\*\*\*Filename: OS.c\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//\*\*\*\*\*\*\*Authors: Sourabh Shirhatti\*//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Nelson Wu\*\*\*\*\*\*\*\*\*//

//\*\*\*\*\*\*\*Created: Jan 24, 2015\*\*\*\*\*\*//

//\*\*\*Description: \*\*\*//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*//

//\*\*\*\*\*\*\*\*\*Lab #: 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//\*\*\*\*\*\*\*\*\*\*\*TAs: ,\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//\*\*\*\*\*\*\*Revised: Feb 8, 2015\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//\*\*\*\*\*HW Config: None\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

#include <stdint.h>

#include "inc/tm4c123gh6pm.h"

#include "OS.h"

#define NVIC\_EN0\_INT21 0x00200000 // Interrupt 21 enable

#define TIMER\_CFG\_32\_BIT\_TIMER 0x00000000 // 32-bit timer configuration

#define TIMER\_TAMR\_TACDIR 0x00000010 // GPTM Timer A Count Direction

#define TIMER\_TAMR\_TAMR\_PERIOD 0x00000002 // Periodic Timer mode

#define TIMER\_CTL\_TAEN 0x00000001 // GPTM TimerA Enable

#define TIMER\_IMR\_TATOIM 0x00000001 // GPTM TimerA Time-Out Interrupt

// Mask

#define TIMER\_ICR\_TATOCINT 0x00000001 // GPTM TimerA Time-Out Raw

// Interrupt

#define TIMER\_TAILR\_M 0xFFFFFFFF // GPTM Timer A Interval Load

// Register

#define GPIO\_PORTF2 (\*((volatile uint32\_t \*)0x40025010))

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

void (\*PeriodicTask)(void); // user function

static unsigned long Counter = 0;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*OS\_AddPeriodicThread\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Activate Timer1 interrupts to run user task periodically

Input: task is a pointer to a user function

period in ms

priority (0-7)

Outputs: error (1); success (0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int OS\_AddPeriodicThread(void (\*task)(void), unsigned long period, unsigned long priority) {

long sr;

if(priority > 7) { return 1; }

Counter = 0;

sr = StartCritical();

SYSCTL\_RCGCTIMER\_R |= 0x02;

PeriodicTask = task; // user function

TIMER1\_CTL\_R &= ~TIMER\_CTL\_TAEN; // 1) disable timer1A during setup

// 2) configure for 32-bit timer mode

TIMER1\_CFG\_R = TIMER\_CFG\_32\_BIT\_TIMER;

// 3) configure for periodic mode, default down-count settings

TIMER1\_TAMR\_R = TIMER\_TAMR\_TAMR\_PERIOD;

TIMER1\_TAILR\_R = (period\*50000) - 1; // 4) reload value

// 5) clear timer1A timeout flag

TIMER1\_ICR\_R = TIMER\_ICR\_TATOCINT;

TIMER1\_IMR\_R |= TIMER\_IMR\_TATOIM;// 6) arm timeout interrupt

// 7) priority shifted to bits 15-13 for timer1A

NVIC\_PRI5\_R = (NVIC\_PRI5\_R&0xFFFF00FF)|(priority << 13);

NVIC\_EN0\_R = NVIC\_EN0\_INT21; // 8) enable interrupt 21 in NVIC

TIMER1\_TAPR\_R = 0;

TIMER1\_CTL\_R |= TIMER\_CTL\_TAEN; // 9) enable timer1A

EndCritical(sr);

return 0;

}

void Timer1A\_Handler(void){

TIMER1\_ICR\_R = TIMER\_ICR\_TATOCINT;// acknowledge timer1A timeout

Counter++;

(\*PeriodicTask)(); // execute user task

}

// GPIO\_PORTF2 = 0x04;

// TIMER1\_ICR\_R = TIMER\_ICR\_TATOCINT;// acknowledge timer1A timeout

// Counter++;

//

// /\*if(Counter == 0xFFFFFFFF) {

// OverflowCount++;

// Counter = 0;

// } \*/

//

// (\*PeriodicTask)(); // execute user task

// GPIO\_PORTF2 = 0x00;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*OS\_ClearPeriodicTime\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

clear the 32-bit global counter

Input: none

Output: none

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void OS\_ClearPeriodicTime(void) {

Counter = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*OS\_ReadPeriodicTime\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reads the current value of the 32-bit global counter

Input: none

Output: current counter value

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned long OS\_ReadPeriodicTime(void) {

return Counter;

}

void OS\_StartPeriodicThread(void) {

TIMER1\_CTL\_R |= TIMER\_CTL\_TAEN;

}

void OS\_StopPeriodicThread(void) {

TIMER1\_CTL\_R |= ~TIMER\_CTL\_TAEN;

}

#ifndef \_\_OS\_H\_\_ // do not include more than once

#define \_\_OS\_H\_\_

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*OS\_AddPeriodicThread\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Activate Timer0 interrupts to run user task periodically

Input: task is a pointer to a user function

period in units (1/clockfreq)

priority (0-7)

Outputs: error (1); success (0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int OS\_AddPeriodicThread(void (\*task)(void), unsigned long period, unsigned long priority);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*OS\_ClearPeriodicTime\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

clear the 32-bit global counter

Input: none

Output: none

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void OS\_ClearPeriodicTime(void);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*OS\_ReadPeriodicTime\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reads the current value of the 32-bit global counter

Input: none

Output: current counter value

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned long OS\_ReadPeriodicTime(void);

void OS\_StartPeriodicThread(void);

void OS\_StopPeriodicThread(void);

#endif // \_\_OS\_H\_\_

1. High level main program (the interpreter)

// Used for UART commands

char HelpADC[] = "Send commands to the ADC";

char HelpLCD[] = "Send commands to the LCD";

char HelpOS[] = "Send commands to the OS";

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CommandADC\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Send debugging commands to the ADC

Input: ui8Argc is the number of commands parsed

g\_ppcArgv is an array containing the commands

Outputs: error (-1); success (channel #)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CommandADC(uint\_fast8\_t ui8Argc, char \*g\_ppcArgv[]) {

uint32\_t channel, Fs, numSamples;

uint16\_t buffer[100];

//Check if number of arguments are correct

if (ui8Argc != 3) {

OutCRLF();

UART\_OutString("All adc functions require exactly 1 parameter");

return;

}

// Check if a valid channel

if (strcmp(g\_ppcArgv[2],"0")==0 || (strcmp(g\_ppcArgv[2],"1")==0) || strcmp(g\_ppcArgv[2],"2")==0 || (strcmp(g\_ppcArgv[2],"3")==0) || strcmp(g\_ppcArgv[2],"4")==0 || (strcmp(g\_ppcArgv[2],"5")==0) || strcmp(g\_ppcArgv[2],"6")==0 || (strcmp(g\_ppcArgv[2],"7")==0) || strcmp(g\_ppcArgv[2],"8")==0 || (strcmp(g\_ppcArgv[2],"9")==0) || strcmp(g\_ppcArgv[2],"10")==0 || (strcmp(g\_ppcArgv[2],"11")==0)) {

channel = atoi(g\_ppcArgv[2]);

} else {

OutCRLF();

UART\_OutString("Invalid channel number specified to adc function");

return;

}

// Check for command open

if (strcmp("open", g\_ppcArgv[1])==0) {

ADC0\_Open(channel);

} else if(strcmp("collect",g\_ppcArgv[1]) == 0) {

ADC\_Collect(channel, Fs, buffer, numSamples);

} else if(strcmp("read",g\_ppcArgv[1]) == 0) {

UART\_OutUDec(ADC\_In());

//Display value to screen/UART?

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CommandLCD\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Send debugging commands to the LCD

Input: ui8Argc is the number of commands parsed

g\_ppcArgv is an array containing the commands

Outputs: error (-1); success (0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CommandLCD(uint\_fast8\_t ui8Argc, char \*g\_ppcArgv[]) {

int16\_t screen = 0, line = 0;

if (ui8Argc <= 3) {

OutCRLF();

UART\_OutString("Insufficient arguments for command lcd");

return;

}

//Check for valid screen number

if (strcmp(g\_ppcArgv[1],"0")==0 || (strcmp(g\_ppcArgv[1],"1")==0)) {

screen = atoi(g\_ppcArgv[1]);

} else {

OutCRLF();

UART\_OutString("Invalid parameter for screen number");

return;

}

// Check for valid line number

if (strcmp(g\_ppcArgv[2],"0")==0 || (strcmp(g\_ppcArgv[2],"1")==0) || strcmp(g\_ppcArgv[2],"2")==0 || (strcmp(g\_ppcArgv[2],"3")==0) || strcmp(g\_ppcArgv[2],"4")==0 || (strcmp(g\_ppcArgv[2],"5")==0) || strcmp(g\_ppcArgv[2],"6")==0 || (strcmp(g\_ppcArgv[2],"7")==0)) {

line = atoi(g\_ppcArgv[2]);

} else {

OutCRLF();

UART\_OutString("Invalid parameter for line number");

return;

}

// Display message

char message[80];

message[0] = '\0';

for (uint16\_t i = 3; i < ui8Argc; i++) {

sprintf(message, "%s %s", message, g\_ppcArgv[i]);

}

ST7735\_MessageString(screen, line, message);

OutCRLF(); UART\_OutString(message);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CommandOS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Send debugging commands to the OS

Input: ui8Argc is the number of commands parsed

g\_ppcArgv is an array containing the commands

Outputs: error (-1); success (0)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void CommandOS(uint\_fast8\_t ui8Argc, char \*g\_ppcArgv[]) {

if (ui8Argc < 2) {

OutCRLF();

UART\_OutString("Insufficient arguments for command os");

return;

}

if (strcmp(g\_ppcArgv[1],"clear") == 0) {

OS\_ClearPeriodicTime();

}

else if (strcmp(g\_ppcArgv[1],"read") == 0) {

UART\_OutString(" ");

UART\_OutUDec(OS\_ReadPeriodicTime());

}

else if (strcmp("stop", g\_ppcArgv[1]) == 0) {

OS\_StopPeriodicThread();

}

else if (strcmp(g\_ppcArgv[1],"start") == 0) {

OS\_StartPeriodicThread();

}

else {

UART\_OutString("command os argument not recognized");

}

}

// Command Table as defined by Tivaware

tCmdLineEntry g\_psCmdTable[] = {

{ "adc", CommandADC, HelpADC },

{ "lcd", CommandLCD, HelpLCD },

{ "os", CommandOS, HelpOS }

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Created by Sourabh Shirhatti and Nelson Wu for EE 445M, Spring 2015

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdint.h>

#include "inc/tm4c123gh6pm.h"

#include "cmdline.h"

#include "PLL.h"

#include "UART.h"

#include "ST7735.h"

#include "OS.h"

#define GPIO\_PORTF2 (\*((volatile uint32\_t \*)0x40025010))

void dummy(void) {

// UART\_OutUDec(OS\_ReadPeriodicTime());

}

int main(void){

char string[80]; // global to assist in debugging

PLL\_Init(); // set system clock to 50 MHz

Output\_Init();

UART\_Init(); // initialize UART

OS\_AddPeriodicThread(dummy, 1000, 3);

SYSCTL\_RCGC2\_R |= SYSCTL\_RCGC2\_GPIOF; // activate port F

GPIO\_PORTF\_DIR\_R |= 0x04; // make PF2 out (built-in LED)

GPIO\_PORTF\_AFSEL\_R &= ~0x04; // disable alt funct on PF2

GPIO\_PORTF\_DEN\_R |= 0x04; // enable digital I/O on PF2

// configure PF2 as GPIO

GPIO\_PORTF\_PCTL\_R = (GPIO\_PORTF\_PCTL\_R&0xFFFFF0FF)+0x00000000;

GPIO\_PORTF\_AMSEL\_R = 0; // disable analog functionality on PF

GPIO\_PORTF2 = 0; // turn off LED

while(1){

OutCRLF(); UART\_OutString(">");

UART\_InString(string,79);

if(CmdLineProcess(string) == -1) {

UART\_OutString("command not recognized");

}

}

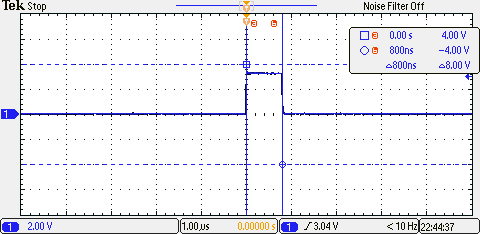
}

**Measurement Data**

1. Estimated time: 15 instructions \* 2 CPI \* 20 ns/cycle = 600 ns

As stated in the lab descriptor, it is difficult to estimate how long the ISR will take due to compiler optimizations and pipelining. On the ARM site, most instructions take one cycle added to some pipelining time, and branch instructions take two plus the time to clean up if the branch prediction failed, which is why we have used 2 CPI as the average time.

1. Measured time: 800 ns



**Analysis and Discussion**

1. ADC Range = 0 - 3.3V

ADC Resolution = 0.00080566406 V

ADC Precision = 12 bits (4096 alternatives)

1. ADC conversion can be started through software (manually calling for a sample), with timer interrupts (at a periodic rate), analog comparators (voltage level), PWM (external pulse signal), and GPIO trigger (pin toggle). We chose to use a software trigger and a timer (periodic) trigger. Software triggers are useful for debugging because we can call for a sample at some point while a program is running to see what the current input is. A periodic trigger allows us to sample some external signal and accurately analyze it for use in the rest of the program.
2. We used method one (setting an output port high at the start of the ISR, and clearing it at the end of the ISR). Profiling methods other than the two mentioned could include reading SysTick (or another timer) at the beginning and end of the ISR, subtracting the two counts, and multiplying by the SysTick period. Our method required the use of a scope or logic analyzer with the appropriate resolution, while the SysTick method would all be in software. Toggling a pin in the ISR is easier than measuring it indirectly because the main program would need to directly access the port (without a read-modify-write sequence); otherwise, the interrupt could occur in the middle, leading to an inaccurate measurement. Toggling in the ISR only requires the addition of two lines of C code, so it is simple and fast. The software method requires more overhead, and it also takes time to read SysTick and output the value.
3. 800 ns/(15 assembly instructions) = 53.3 ns

It takes an average of 53.3 ns to execute one instruction. Running at a system clock frequency of 50 MHz (20 ns clock period), one instruction completes in an average of 2.665 clock cycles.

1. SysTick Range: fbus/(n+1) = interrupt frequency (n is the value in NVIC\_ST\_CURRENT\_R)

SysTick Resolution: 1/fbus = seconds between each tick

SysTick Precision: 24 bits (size of value in NVIC\_ST\_CURRENT\_R)