EEL 4930 – Fall 2018

Revision **0**

Lab 1: ARM Assembly Checksum, LED Drivers, ISRs

OBJECTIVES

- Familiarize yourself with C callable ARM thumb assembly and assembly callable C functions via matrix checksum
- Create the drivers for the LED drivers via I2C
- Familiarize yourself with NVIC, interrupts, SYSTICK, and via writing ISRs

PART A: ARM Assembly Checksum

MOTIVATION

Writing assembly code is needed when creating low level performance critical code. In Lab 2, you will be writing the core sections of the G8RTOS operating systems. Some of these sections will need to be written in assembly. The first part of this lab will help you familiarize yourself with ARM Thumb Assembly and learn the useful concept of checksums.

Checksum Basics:

Checksums are used to check to see if a data transfer was valid. Checksums are extremely important in the world of digital communication and embedded computing. This is even more so true for space embedded computing since solar radiation actively flips bits during a data transfer.

Fast checksums, lower computation intensity, many errors can get through unnoticed [2]:

1) Parity (even, odd, mark, space)
2) Xor checksum (Longitudinal Redundancy Checks)
    a. Perform an xor on a set of words being transferred and transfer the result for comparison
3) Additive checksums
    a. Perform a sum (typically take one's complement of all numbers to be added first) on a set of words being transferred and transfer the result for comparison

Moderately fast checksums, Intermediate computation intensity, only some errors can get through unnoticed [2]:

1) Fletcher checksum [3]
    a. Data is divided into blocks and modular sum of blocks is computed
2) Adler checksum [2]
    a. Typically not as good at Fletcher checksum

Slow checksum, high computational intensity, very few errors can get through unnoticed

1) Cyclic Redundancy Checks (CRC)
    a. There are numerous CRC implementations that very in computation intensity
    b. Somewhat configurable for the application

The Fletcher checksum works by the following (Straightforward, un-optimized implementation) [3]:

1) Take in a pointer to a vector (1D array) of data and a size
2) Define two temporary variable sum1 and sum2 that are twice as large as the data type of the data vector
3) Accumulate sum1 over the data set (for each data point, add it to sum1)
4) Each time sum1 is updated, the new value of sum1 is added onto sum2
5) To assure that sum1 and sum2 don't overflow, use a modulo operation each time a sum is computed
6) Return a concatenation of sum2 (MSBs) and sum1 (LSBs)

Assignment:

You need to write an assembly function that implements the fletcher-16 algorithm and a C main function that calls it and compares it to a C implementation. The main.c program should create a 4x4 square matrix of uint8_t. Initialize the matrix to be a magic square (you may hard code this part, unless you want to write a magic square function yourself!) [1].

Implement a Fletcher 16 checksum on the matrix in assembly [2] [3]. You need to call this assembly function from within your main.c program. To compute the mod 255 operation create a modulo function in C in the main.c and call it from within the assembly routine.

With this you will learn how to call C from assembly and assembly from C.

Hint: Be careful how you address the individual bytes in assembly (look at the instruction set carefully)

To help you get started, look over the Implementation section of [3]. You must implement the **straightforward** implementation from [3] and create the assembly callable function for the modulus operation.

In addition you **must** implement the fletcher-16 function in C for comparison. You can use the C code snippet in [3] for this task. At the end of your main.c program you need to compare the result of the fletcher C routine and the fletcher assembly routine.

You can check whether the comparison is correct or not either by using a breakpoint and debugging the code or using the serial port to send a message to the screen. We recommend using the serial port so that you can become familiar with its operation as you will need it in later labs.

IN-LAB REQUIREMENTS

Demo the Fletcher checksum for the magic matrix and a matrix that will be provided in lab.

Lab 1: ARM Assembly Checksum, LED Drivers, ISRs

RESOURCES:

[1] https://en.wikipedia.org/wiki/Magic_square

[2] https://betterembsw.blogspot.com/2010/05/which-error-detection-code-should-you.html

[3] https://en.wikipedia.org/wiki/Fletcher's_checksum

[4] https://betterembsw.blogspot.com/2010/05/whats-best-crc-polynomial-to-use.html

PART B: LED Drivers

MOTIVATION

When working with embedded systems, you will be required to interface with external devices. Communication with external devices can be parallel or serial in nature. Serial methods of communication are extremely common because they decrease the number of lines or traces that need to connect the master and slaves. During this semester, and for years to come, you will find I^2C useful because a very large number of devices can be addressed using only two lines or traces.
This section will introduce you to the registers used by the MSP432 for I^2C communications, and prepare you to interface with external devices.

Basics:

If you haven't read the manual for the LP3943 and the I^2C section of the MSP432 Family manual, stop and do so now.

We will be controlling 16 RGB LEDs. This means that we will be powering 48 individual LED channels. This is a major reason to use an LED driver, we don't lose 48 GPIO channels to the LEDs. Consequently, we will be using three LED drivers, one for the red, blue, and green channels of each LED.

The LED drivers we have chosen are not just a simple driver, they also are small microcontrollers that can generate a PWM signal on each of their 16 LED channels. This is very useful because we can use the duty cycle of the PWM signal to dim or brighten each channel. In effect this will allow us to create any visible color by combining different brightness values of the red, green, and blue channels of each LED. This is the same process used by your phone, tablet, and computer screens. **You are not required to use the LEDs in PWM mode, but it is your choice if you'd like to.**

Assignment:

You will write a library to communicate with the LP3943s on your board. You should write a function to initialize the LEDs (the I2C peripheral and all the LEDs off). You should also create a function to change the state the LED array. **This means the function should take in a color, and a 16bit stream that indicates which LED to light up (you will need to do some bit manipulation to format the ON setting for the LEDs, refer to pg. 13 in the documentation).** It is highly suggested that you make use of the LP3943's register increment bit.

I recommend using an enum for the three colors: RED, BLUE, and GREEN.

You may also find it useful to know that the driver at 0x60 controls all the blue channels, 0x61 controls all the green channels, and 0x62 controls all the red channels. So as not interfere with other devices, you should use the eUSCI_B_2 port. Consult the pin-out to determine which pins this port corresponds to.

The DAD board has a built in I2C interpreter, which is highly recommended for debugging purposes to make sure you're seeing the address and data being sent out appropriately. You can probe these pins from the Launchpad.

PRE-LAB QUESTIONS

1. What is the maximum clock speed of the LP3943?
2. In your own words, describe the I^2C communications process for the LP3943

IN-LAB REQUIREMENTS

Run your part A code on the magic square matrix and display the result in hex on your LEDs (you must use all 3 colors to show that you can properly use all of them)

RESOURCES:

[1] http://www.ti.com/lit/ug/slau356e/slau356e.pdf

[2] http://www.ti.com/lit/ds/symlink/lp3943.pdf

PART C: ISR

MOTIVATION

There are many dynamics to the ARMv7-M exception model (how exceptions are handled). One of the reasons for ARM's incredible success in the embedded systems world is its compatibility with RTOSs. This compatibility largely comes from the exception model. This is critical to understand to fully utilize the processor.

Basics:

For an introduction to the exception model, please read over the PowerPoint lecture and refer to the manuals for more in depth knowledge.

Assignment:

Lab 1: ARM Assembly Checksum, LED Drivers, ISRs

Create at least two different multistep, animated color patterns (be creative!). Use the SYSTICK timer to create an interrupt that occurs once every 250ms. Within the ISR, communicate the next step of the pattern to the LEDs via the LED drivers made prior.

Create an external interrupt on port 4 bit 4 (this corresponds to the top button on the daughter board). You will write this ISR in C. Within this ISR, change the LED pattern you are displaying.

To initialize the NVIC, you can use the NVIC_EnableIRQ(IRQn) function.

You are not allowed to use software loops to wait for interrupts. You must enter the low-power-modes (LPMs) when waiting for interrupts.

PRE-LAB QUESTIONS

1) What are the advantages and disadvantages of using the Systick over some other timer?
2) What was your final percentage error for the periodic interrupt?

IN-LAB REQUIREMENTS

Demo the external interrupt driven by the button interrupt and the systick interrupt via the LEDs.

RESOURCES:

[1] ARM exceptions lecture
[2] ARMv7-m manual
[3]http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf