San José State University
Department of Computer Engineering
CMPE 146-03, Real-Time Embedded System Co-Design, Fall 2019

Lab Assignment 2

Due date: Friday, 9/20/2019

1. Description

In this assignment, you will be familarized with the two acclerator units on the MSP432 MCU: CRC-32 and AES.

2. Exercise 1. CRC-32

In this exercise, you will use the CRC-32 checksum generator to compute the CRC checksum of a block of data. You will duplicate the example project, /Software/SimpleLink MSP432P4 SDK (3.20.00.06)/Examples/Development Tools/MSP432P401R LaunchPad - Red 2.x (Red)/DriverLib/ crc32_32-bit_signature_calculation/No RTOS/CCS Compiler/crc32_32-bit_signature_calculation, and modify it for this exercise. You will only need to modify the C file, *crc32_32-bit_signature_calculation.c*.

Exercise 1.1

Change the array size of *myData* to be much bigger, like 1024*10. Add some code near the beginning of *main()* to initialize the array with some random numbers. You can use the C library function *rand()* or some other way you prefer. You also need to make the numbers odd and even alternatively, for example, 12, 15, 34, 255, 70, … So, this is the block of data we are going to compute the checksums on. You may want to use *printf()* to print out some beginning numbers to convince yourself that the data pattern is good.

Lab Report Submission

List the code in your report.

Exercise 1.2

Write a small function to implement a very simple checksum algorithm. Here is the prototype:

uint32_t compute_simple_checksum(uint8_t data, uint32_t length)*

The function will just add all the bytes in the data array together into a 32-bit word. Upon returning from the function, flip all the bits in the sum value. (This is kind of a general practice to avoid some specific situation.)

Lab Report Submission

List the function in your report.

Exercise 1.3

At this point, you should have three ways to compute a 32-bit checksum of the data block. The first two are given in the example project. One is using the DriverLib functions *MAP_CRC32_xxx()*, which uses the hardware accelerator. The second method is the *calculateCRC32()* function, located after *main()*. The third one is *compute_simple_checksum()* that you just wrote. The results from the first two should be identical. (Don't forget to flip all the bits after calling *MAP_CRC32_getResultReversed()*, just like what the example code does.)

We are going to see how fast each method does. Use the *Timer32_xxx* DriverLib functions that you might have used in Lab 1. By default, the timer should run at 3 MHz, and it counts down from its maximum value. Use the timer function to measure how long the three methods take to compute the checksum. Print out the time in µs and the checksum for each method.

Lab Report Submission

List the code that did the measurements. Also show the times and checksums. For the CRC checksum methods, compute the speedup of using the accelerator over the pure software method.

Exercise 1.4

We are going to make some small changes to the data and see how the checksum changes. We will just use two methods: *MAP_CRC32_xxx()* and *compute_simple_checksum()*. Pick a data point in the data block, say, *myData[20]*. Flip the least significant bit. Compute the checksums and print them. Then, flip the least significant bit of *myData[21]*. Compute the checksums and print them.

Lab Report Submission

List the code that did the above actions. Describe the results and explain what you saw. How do they compare to the ones before the data are modified. Which checksum method will you use and why?

3. Exercise 2. AES

In this exercise, you will use the AES accelerator to encrypt and decrypt a block of data. You will use a terminal program to connect to LaunchPad through a COM port on your laptop or PC. User types in a command on the terminal and LaunchPad will execute the command and return the results.

Exercise 2.1

You will duplicate the example project, /Software/SimpleLink MSP432P4 SDK (3.20.00.06)/Examples/Development Tools/MSP432P401R LaunchPad - Red 2.x (Red)/DriverLib/uart_pc_echo_12mhz_brclk/No RTOS/CCS Compiler/uart_pc_echo_12mhz_brclk. Install a terminal program on your laptop if you have not done so before. There are many such tools available from the Web. For example, you can download PuTTY from https://www.puttygen.com/download-putty. (CCS Cloud has a terminal function, too. But it is more convenient to be able to use a stand-alone program without relying on an Internet connection.)

The example project expects these communication settings: 9600 baud rate, 1 stop bit and no parity. So, set these up accordingly on the terminal program. Enable local echo on the terminal. Also enable the feature to automatically insert a <LF> when a <CR> is received. When you are running the program, after a key is pressed, you'll see two characters displayed on terminal: one echoed by the terminal and the other by

LaunchPad. When you press the <Enter> key, you should see the cursor on the terminal advance a couple of lines.

After you get the program working, change the baud rate to be much higher; bring it to a more modern standard. Modify the data structure *uartConfig* in *uart_pc_echo_12mhz_brclk.c* to use 460,800 baud. You may want to consult the MCU technical reference on how to compute for certain fields in the data structure.

Lab Report Submission

List the data structure. Show a sample of interactions on the terminal.

Exercise 2.2

In this exercise, you will do some command-line processing after receiving a command from the terminal. The command line is termined by a <CR> character. In the ISR *EUSCIA0_IRQHandler()*, disable the echoing. Instead, store the received characters in a buffer. The while loop in *main()* should constantly check the buffer to see if a CR is received. If so, process the entire command line received and return the results.

Here is the format of the command line: command_character : text_string<CR>
The first character is the command, followed by ':', then followed by the argument, an ASCII string. The whole command line is terminated by the CR character.

Here are the commands to be supported:

H: Convert the string into an ASCII hex string. For example, "H:ABC" should produce "414243".
T: Convert the string, assumed to be a hex string, back to ASCII. For example, "T:414243" should produce "ABC".

Add some simple error checking. Send an error message back to the terminal for bad commands.

In the main C file, you should have a small function to send a string to the UART so that user can see the results. Make sure you also send a <CR> at the end as well. In the function, you can check if the UART is busy or not. There is a DriveLib function *UART_queryStatusFlags()* for that. Wait until the UART is not busy, then send one character to the transmit data register. You don't need to use interrupt for this purpose.

Lab Report Submission

List the code. Explain briefly on how you design your buffer. Show a sample of interactions on the terminal.

Exercise 2.3

In this exercise you will make use of the AES accelerator to encrypt and encrpt something user types in.

Look at the example project, project, /Software/SimpleLink MSP432P4 SDK (3.20.00.06)/Examples/Development Tools/MSP432P401R LaunchPad - Red 2.x (Red)/DriverLib/aes256_encrypt_decrypt/No RTOS/CCS Compiler/aes256_encrypt_decrypt. Incorporate the way encryption and decryption are done into the program that you wrote above. Since the results from encryption and decryption may contain invisible characters, let's use hex string as input and output. For simplicity, we will restrict the data block to be only 16 bytes long (128 bits) and the key length is 32 bytes long (256 bits). So, we only process one block of data. If the user input is shorter, fill zeroes to the rest of the block.

Add a few more commands in your command processing:
K: Store the input key for encryption and decryption. For example: “K:313233” would produce a key: 0x01,0x02,0x03, and the rests are 0x00. You can use the ‘H’ command you wrote earlier to get the hex string for an ordinary text. You may want to return a simple message like “ok” to the terminal after the command is processed.
E: Encryt a message in hex string. For example: “E:414243” would encrypt a data block with beginning bytes 0x41, 0x42 and 0x43. Your program should return the encrpted message in hex string to the terminal.
D: Decryt a message. For example: “D:414243….” would decrypt the message and return the decrypted message. Note that the input must be long enough for a 16-byte block.

Try the key “12345” and the message “This is a test” to see if you can successfully encrypt message and restore the original message.

Lab Report Submission

List the code. Show a sample of interactions on the terminal of the encryption and decryption process.

4. Grading

	Points
Exercise 1.1	1
Exercise 1.2	1
Exercise 1.3	1
Exercise 1.4	2
Exercise 2.1	1
Exercise 2.2	2
Exercise 2.3	2
Total	10