ECE/CSE 412 INTRO TO EMBEDDED SYSTEMS
LAB 2
Assembly, C, Flash Storage, EEPROM
by Eugene Rockey
Copyright 2025, All Rights Reserved

Lab 2 is designed, in part, to continue with Assembly thereby advancing assembly-programming concepts. These concepts are necessary to understand low-level modular and structured programming as well as the important relationship between Assembly and C code. Also, Lab 2 will further demonstrate the relationship between software and the hardware. Accordingly, configuration and control of some of the MCU architecture and peripherals is explored using Assembly and or C programming.

Yellow highlight points out lab related action required from the student.

Green highlight points out report related action required from the student.

Turquoise highlight emphasizes certain terms and information.

*Important: Always be aware of static electricity and take actions to prevent it from destroying computer equipment by using the anti-static mats and the anti-static wrist straps. If possible, wear 100% cotton clothing and nothing synthetic. Store your coats and sweaters in lockers to help minimize static near the workbench. Lay hands on the anti-static mats to discharge yourself. Rinsing your hands with warm water helps dissipate static charge. Hand lotion helps reduce static charge. Washing and drying clothes using fabric softener reduces static buildup. **Keep the AVR board in its anti-static box when not in use.**

Part 1: Modular Programming: In this part, the student will write some given modular code and then expand on that code in a continuing modular form and according to the description of the program. Modular programming consists of a MAIN section along with SUBROUTINES. The MAIN section is used to initialize everything global about the system software and system hardware. The MAIN section then calls the subroutines. And subroutines can call other subroutines. Subroutines can perform their own local initialization but ultimately perform some sort of function or operation.

Program Description: In the 328PB MCU, there is no divide instruction, therefore; enter the given assembly program in Microchip Studio to divide two numbers. The input data for the program comes from the internal FLASH memory. This data contains two constant 8-bit numbers, one is the 8-bit dividend and one the 8-bit divisor. The output from the program consists of two 8-bit variables, one is the 8-bit quotient and one the 8-bit remainder. After building the following code, look at the assembly-listing file (.lss) to discover the issued addresses for these declared constants and variables. Discuss in your report where dividend, divisor, count, quotient, and remainder are located in memory. Create a description, flowchart, and pseudo code from the given assembly code. In your report, put the program description, flowchart, and pseudo code under the BODY heading. Put the assembly program shown below under the SOFTWARE heading. Create the comments that are asked for in the code. *Align the Assembly columns just as shown here*

```
/*
* lab2p1.asm
* Positive 8-bit Integer Division
* Created: 2:07:42 PM
* Author: Eugene Rockey
*/
; Declare Variables
;******************************
           .dseg
           .org  0x100              ;originate data storage at address 0x100
quotient:  .byte 1                  ;uninitialized quotient variable stored in SRAM aka data segment
remainder: .byte 1                  ;uninitialized remainder variable stored in SRAM
           .set  count = 0          ;initialized count variable stored in SRAM
;******************************
           .cseg                    ;Declare and Initialize Constants (modify them for different results)
           .equ  dividend = 13      ;8-bit dividend constant (positive integer) stored in FLASH memory aka code segment
           .equ  divisor = 3        ;8-bit divisor constant (positive integer) stored in FLASH memory
;******************************
;* Vector Table (partial)
;******************************
           .org  0x0
reset:     jmp   main               ;RESET Vector at address 0x0 in FLASH memory (handled by MAIN)
int0v:     jmp   int0h              ;External interrupt vector at address 0x2 in Flash memory (handled by int0)
;******************************
;* MAIN entry point to program*
;******************************
           .org  0x100              ;originate MAIN at address 0x100 in FLASH memory (step through the code)
main:      call  init               ;initialize variables subroutine, set break point here, check the STACK,SP,PC
           call  getnums            ;Check the STACK,SP,PC here.
           call  test               ;Check the STACK,SP,PC here.
           call  divide             ;Check the STACK,SP,PC here.
endmain:   jmp   endmain
init:      lds   r0,count           ;get initial count, set break point here and check the STACK,SP,PC
           sts   quotient,r0        ;use the same r0 value to clear the quotient-
           sts   remainder,r0       ;and the remainder storage locations
           ret                      ;return from subroutine, check the STACK,SP,PC here.
getnums:   ldi   r30,dividend       ;Check the STACK,SP,PC here.
           Ldi   r31,divisor
           ret                      ;Check the STACK,SP,PC here.
test:      cpi   r30,0              ;is dividend == 0 ?
           brne  test2
test1:     jmp   test1              ;halt program, output = 0 quotient and 0 remainder
test2:     cpi   r31,0              ;is divisor == 0 ?
           brne  test4
           ldi   r30,0xEE           ;set output to all EE's = Error division by 0
           sts   quotient,r30
           sts   remainder,r30
test3:     jmp   test3              ;halt program, look at output
test4:     cp    r30,r31            ;is dividend == divisor ?
           brne  test6
           ldi   r30,1              ;then set output accordingly
           sts   quotient,r30
test5:     jmp   test5              ;halt program, look at output
test6:     brpl  test8              ;is dividend < divisor ?
           ser   r30
           sts   quotient,r30
           sts   remainder,r30      ;set output to all FF's = not solving Fractions in this program
test7:     jmp   test7              ;halt program look at output
test8:     ret                      ;otherwise, return to do positive integer division
divide:    lds   r0,count           ;student comment goes here
divide1:   inc   r0                 ;student comment goes here
           sub   r30,r31            ;student comment goes here
           brpl  divide1            ;student comment goes here
           dec   r0                 ;student comment goes here
           add   r30,r31            ;student comment goes here
           sts   quotient,r0        ;student comment goes here
           sts   remainder,r30      ;student comment goes here
divide2:   ret                      ;student comment goes here
int0h:     jmp   int0h              ;interrupt 0 handler goes here
.exit
```

In your report, discuss what is happening to the STACK in SRAM, pointed to by the Stack Pointer, and as the Program Counter steps through the code during the CALLs and RETs to and from the subroutines in this program. Re-write this division program so that only one CALL instruction is in MAIN and that each subroutine CALLs the next appropriate subroutine immediately before executing its RET instruction. In other words, nest the entire set of subroutines and step through using the debugger. In your report, discuss the differences concerning the STACK and its content when executing the original division program versus your nested re-write. In your report, put your nested division program under the SOFTWARE heading and include descriptive comments about each line of code.

Part 2: Data tables are important in embedded systems because they efficiently provide expected data necessary to perform some process. The data could be calculated but that would take time and resources and could be less practical depending on the hardware. In some cases, data must be calculated but using a data table where applicable is usually the better choice. The following code demonstrates a simple look-up data table to quickly convert Celsius to Fahrenheit.

```
/*
* lab2p2.asm
* Celsius to Fahrenheit Look-Up Table
* Created: 10:17:31 AM
* Author: Eugene Rockey
*/
        .dseg
        .org    0x100
 output:.byte   1               ;student comment goes here
        .cseg
        .org    0x0
        jmp     main            ;partial vector table at address 0x0
        .org    0x100           ;MAIN entry point at address 0x200 (step through the code)
main:   ldi     ZL,low(2*table) ;student comment goes here
        ldi     ZH,high(2*table);student comment goes here
        ldi     r16,Celsius     ;student comment goes here
        add     ZL,r16          ;student comment goes here
        ldi     r16,0           ;student comment goes here
        adc     ZH,r16          ;student comment goes here
        lpm                     ;lpm = lpm r0,Z in reality, what does this mean?
        sts     output,r0       ;store look-up result to SRAM
        ret                     ;consider MAIN as a subroutine to return from - but back to where??
                                ;Fahrenheit look-up table
table:  .db     32, 34, 36, 37, 39, 41, 43, 45, 46, 48, 50, 52, 54, 55, 57, 59, 61, 63, 64, 66
        .equ    celsius = 5     ;modify Celsius from 0 to 19 degrees for different results
        .exit
```

In your report, put your commented program under SOFTWARE and discuss how this Fahrenheit look-up table is indexed using the Celsius input parameter. Re-write the table program to sort 20 random values stored in the internal FLASH. Use the sorting method of your choice; sort from max to min or min to max. Store the sorted numbers to SRAM as the output. In your report, put your commented SORT program under the SOFTWARE heading.

Part 3: Relationship between C and its equivalent Assembly
When C is compiled, an intermediate Assembly source listing (.lss) is generated. This file is great for debugging the C code it represents (for logic errors) and for learning the relationship between C and the target processor's Assembly language. Understanding this C-Assembly relationship helps to make you a better C programmer and better at mixing C and Assembly source files into a

single project solution. The following C code is entered into Microchip Studio as a NEW C PROJECT for the ATMega328PB. And it is named L2P3_C. There are four MAIN codes with GLOBAL variables of different type, and only one MAIN section should be un-commented and compiled at a time. Compile the first MAIN section and then go to the output files in the solution explorer window. Double click the .lss file and examine it in the editor window. At the bottom of the file's listing will be the Assembly equivalent of the C source file's MAIN section. Ignore all the stuff above the MAIN section. The compiler will have commented some of the Assembly lines in MAIN. Comment all the lines that the compiler left empty in MAIN. Make each comment describe what its Assembly line is doing with respect to the C program. Repeat this task for the next MAIN section. In your report, discuss the relationship between each of the following four MAIN sections of C code and their equivalent Assembly code. Discuss the CHAR and INT data types and how they are represented differently in Assembly. Discuss how the Assembly in one MAIN section differs from the Assembly in the other with respect to CHAR versus INT addition. Build the INT and CHAR as SIGNED and UNSIGNED, examine the .lss listing for each; try subtraction in place of addition. Discuss the Assembly differences between SIGNED and UNSIGNED addition in your report. Put ALL four of the MAIN fully commented sections of Assembly code under the report's Software heading along with their respective C code.

```c
/*
* L2P3_C.c
* Example relationship between C and Assembly
* Created: 4:56:28 PM
* Author: Eugene Rockey
*/

//Compile and examine the .lss file beginning at <main>:,comment the lines the compiler left empty.
        signed char Global_A;
        signed char Global_B = 1;
        signed char Global_C = 2;
        int main(void)
        {
                Global_A = Global_C + Global_B;
        }

/*
//Compile and examine the .lss file beginning at <main>:,comment the lines the compiler left empty.
        unsigned char Global_A;
        unsigned char Global_B = 1;
        unsigned char Global_C = 2;
        int main(void)
        {
                Global_A = Global_C + Global_B;
        }
*/
/*
//Compile and examine the .lss file beginning at <main>:,comment the lines the compiler left empty.
        signed int Global_A;
        signed int Global_B = 1;
        signed int Global_C = 2;
        signed int main(void)
        {
                Global_A = Global_C + Global_B;
        }
*/
/*
//Compile and examine the .lss file beginning at <main>:, comment the lines the compiler left empty.
        unsigned int Global_A;
        unsigned int Global_B = 1;
        unsigned int Global_C = 2;
        int main(void)
        {
                Global_A = Global_C + Global_B;
        }
*/
```