

## ECE 375: Computer Organization and Assembly Language Programming

### Lab 3 – Data Manipulation & the LCD

## SECTION OVERVIEW

### Complete the following objectives:

- Understand the basics of data manipulation in AVR assembly.
- Learn the basic steps of initializing a program, such as setting up the stack pointer, initializing registers, configuring peripheral devices, etc.
- Use the X, Y, and Z pointers to perform indirect addressing.
- Declare constant data in the program memory, and iteratively move that data into the data memory.
- Use pre-written library functions to interact with a peripheral device.

## PRELAB

To complete this prelab, you may find it useful to look at the AVR Assembler Guide and the AVR Instruction Set Manual. If you consult any online sources to help answer the prelab questions, you **must** list them as references in your prelab.

1. What is the stack pointer? How is the stack pointer used, and how do you initialize it? Provide pseudocode (**not** actual assembly code) that illustrates how to initialize the stack pointer.
2. What does the AVR instruction LPM do, and how do you use it? Provide pseudocode (**not** actual assembly code) that shows how to setup and use the LPM instruction.
3. Take a look at the definition file `m32U4def.inc`. What is contained within this definition file? What are some of the benefits of using a definition file like this? Please be specific, and give a couple examples if possible.

## BACKGROUND

### Introduction

For this lab, you will learn to interact with the LCD included on the mega32 microcontroller board. In order to use the LCD, you will first need to learn how to properly initialize an assembly program. Next, you will learn how to move data from the program memory into the data memory, and then display that data as characters on the LCD.

To help you along the way, a skeleton code has been provided for this lab. Similar to the Lab 2 skeleton C file, this file contains some code that is already written, and some comments indicating where you need to add some code of your own. This skeleton file also provides a good foundation for writing well-structured and well-commented assembly code, as defined in the AVR Assembler Guide, and **as required for full credit on your lab write-up**.

### Initialization

A program initialization (INIT) consists of any code that is only run once, at the beginning of the program. This code does not belong in the MAIN portion of the program, which is why most assembly programs begin with INIT and not MAIN. INIT is not a function (you wouldn't ever jump to it from within the MAIN part of your program), but rather a collection of instructions that are executed at power on (and at reset). Traditionally, INIT finishes with a jump to MAIN.

There are several things that should be included in the INIT section of a program. The first and foremost is the initialization of the stack pointer. The stack in AVR operates in a higher-to-lower address fashion, meaning that the most recent element placed on the stack is at a lower address than the previous element placed on the stack. Therefore, the stack pointer should initially point to the upper end of the data memory; in other words, to **the location with the highest address in the data memory space**.

In addition to initializing the stack pointer, which **must** be done for your program to perform function calls correctly, there are several other things that are typically initialized within the INIT section of a program, such as:

- I/O ports
- Timer/counters
- Interrupts
- Peripheral devices

The exact contents of `INIT` will depend on which features of the `mega32` board you need to use in your program. For this lab, you will need to initialize the LCD. To do so, you will perform a function call to the LCD initialization subroutine, `LCDInit`, which is described later in this document. You will also need to move data from the program memory to the data memory in this lab, so you may want to plan on doing that in your `INIT` section as well.

### Using the LCD Driver

To successfully interact with the LCD, you will need to use some pre-written functions that are part of an LCD driver file provided on the lab webpage (`LCDDriver.asm`). **Make sure to download this file and include it in the same directory with your AVR assembly code.** Unlike the `m32U4def.inc` definition file, which is purely pre-compiler directives, the LCD driver contains actual assembly instructions and thus cannot be included at the beginning of your main program file. As indicated in the skeleton code, any included **code files** are included at the end of the main program, i.e. in the last line(s).

After including the LCD driver into your program, you will still need to properly setup and call the functions defined in the LCD driver to interact with the LCD. (Remember, for any function calls to work correctly, the stack pointer must have already been initialized.) Detailed descriptions of the LCD driver functions are provided in Appendix A at the end of this handout, but here is an overview:

- `LCDInit` – Initialize the LCD (call once from within `INIT`)
- `LCDBacklightOn` – enables the backlight
- `LCDBacklightOff` – disables the backlight
- `LCDWrite` – writes both lines of text to the LCD
- `LCDWrLn1` – writes the top line (16 characters) to the LCD
- `LCDWrLn2` – `LCDWrLn2` - writes the bottom line (16 characters) to the LCD
- `LCDClr` – Clear (write “space” to) all characters of both lines
- `LCDClrLn1` – Clear all characters of line 1 only
- `LCDClrLn2` – Clear all characters of line 2 only
- `LCDWriteByte` – Write directly to a single character of the LCD
- `Bin2ASCII` – Convert an 8-bit unsigned value into an ASCII string

There are other functions within the LCD driver, but they simply support the functions listed above, so they should not be called directly from your program.

### Data Manipulation

To move data from one memory type to another, you first must understand how the available memory is organized. The ATmega32 microcontroller has an 8-bit AVR architecture. This means that all registers and data memory locations are 8 bits wide, and all data is handled 1 byte at a time (disregarding certain instructions that can treat certain pairs of registers as one 16-bit “word”).

However, the AVR instructions supported by the ATmega32 are 16 bits wide (or 32 bits wide, for a small number of instructions), and so for efficiency reasons the ATmega32’s program memory is 16 bits (2 bytes) wide. This has an important consequence: Unlike pointers to data memory, pointers to program memory **initially point to a 16-bit memory location**, and you will have to take certain steps to read just the low byte or just the high byte of that location.

When writing (and especially when testing) your program, it is often useful to include some data directly into program memory. For example, imagine you are simulating your program, and you are testing a function which uses several data memory locations as input. Instead of manually entering test input values into the data memory via the Memory window in Atmel Studio, you can use the `.DB` (Data Byte) directive to have the compiler place your test values into the program memory during compilation, and then write a simple loop or function that copies your test values from program memory into data memory at runtime.

The following example shows how to place data into the program memory:

```
DATA:
    .DB $05, $F4
```

The hexadecimal values `$05` and `$F4` can then be accessed using the label `DATA`, which the compiler converts to a program memory address. To read this data, use the `LPM` (Load Program Memory) instruction. You can also enter strings (i.e., a sequence of ASCII characters) into program memory using the `.DB` directive; just take a look at the Lab 3 skeleton file to see how.

Note: Since the program memory is 16 bits wide, the total amount of data you specify **using a single instance** of the `.DB` directive should be an integer multiple of 16 bits. Otherwise, the compiler will pad the program memory with an extra byte of data, usually `$FF`.

Movement within data memory is accomplished using variations of load and store instructions, and with two main addressing modes: *direct addressing* and *indirect addressing*. Direct addressing, where a memory address is provided directly as an operand, is useful when you want to move a single byte to/from a single memory location (such as an extended I/O register). Indirect addressing,

```

• Z <- program memory address of first character
• Y <- data memory address of character destination
• do { mpr <- ProgramMemory[Z], Z++,
    DataMemory[Y] <- mpr, Y++ }
    while (Z != program memory address after last character)

```

Figure 1: Pseudocode for String Copy from Program Memory to Data Memory

where the X, Y, and Z-pointers are an operand and they *contain an effective address*, is useful for moving blocks of data into contiguous locations in the data memory. By using indirect addressing in conjunction with a loop, you can efficiently move lots of data around in memory. Figure 1 shows some pseudocode you can use to properly read a bunch of data from program memory.

## PROCEDURE

The display should be initially blank. To receive full credit, you must: (1) properly initialize your program, (2) place your two strings into program memory using the .DB directive. (3) Use two unique strings, for example one string could contain your name, and another your partners. (4) copy the strings from program memory to the appropriate data memory locations **using a loop**, and (5) there must not be any unintended trailing characters on the LCD.

You will use four buttons; PD4, PD5, PD6, and PD7 to write and clear text on the LCD display. Write a program that does the following:

1. When you press PD4: The content should be cleared.
2. When you press PD5: Displays your name on the first line of the LCD, and a phrase like "Hello World!" (or your partners name) on the second line of the screen.
3. When you press PD6: The contents should swap so that a phrase like "Hello World" is shown on line 1 and your name is shown on line 2.
4. When you press PD7: Not being content with displaying a static message, you would like to add some flair by displaying a scrolling, marquee-style message. Modify your program so that the text scrolls across the two lines of the LCD from left to right. Include some wait time (delay) between each "scroll" event, so that there is enough time to read the characters in their current position before

they move again. When a character reaches the end of a line, display it next at the beginning **of the opposite line**. In other words, a character at index 15 of line 1 will move to index 0 of line 2, and a character at index 15 of line 2 will move to slot 0 of line 1.

Button PD4:     Line 1:  
                  Line 2:

Button PD5:     Line 1: Your name  
                  Line 2: Hello, World (or partner's name)

Button PD6:     Line 1: Hello, World (or partner's name)  
                  Line 2: Your name

Button PD7:

- 1) Line 1: \_\_\_\_My\_Name\_is\_  
   Line 2: \_\_\_\_\_Jane\_Doe\_  
      (wait for .25 seconds)
- 2) Line 1: \_\_\_\_\_My\_Name\_is  
   Line 2: \_\_\_\_\_Jane\_Doe  
      (wait for .25 seconds)
- 3) Line 1: e\_\_\_\_\_My\_Name\_i  
   Line 2: s\_\_\_\_\_Jane\_Do  
      (wait for .25 seconds)
- 4) Line 1: oe\_\_\_\_\_My\_Name\_  
   Line 2: is\_\_\_\_\_Jane\_D  
      etc.

## STUDY QUESTIONS / REPORT

### Study Questions

1. In this lab, you were required to move data between two memory types: program memory and data memory. Explain the intended uses and key differences of these two memory types.
2. You also learned how to make function calls. Explain how making a function call works (including its connection to the stack), and explain why a RET instruction must be used to return from a function.
3. To help you understand why the stack pointer is important, comment out the stack pointer initialization at the beginning of your program, and then

try running the program on your mega32U4 board and also in the simulator. What behavior do you observe when the stack pointer is never initialized? In detail, explain what happens (or no longer happens) and why it happens. After answering the question, **make sure to include the stack pointer back in your code before submitting in Canvas.**

## APPENDIX A – LCD FUNCTIONS

### LCDInit

This subroutine initializes the serial interface that is used to communicate with the LCD, and also initializes the display itself, configuring it to display 2 lines/rows of 16 characters (2x16). This function can be called directly, e.g.:

```
rcall  LCDInit      ; Initialize LCD peripheral interface
```

### LCDWrite

This function writes data to both lines of the LCD. First, line data is retrieved from the following data memory addresses:

Line 1: \$0100 – \$010F

Line 2: \$0110 – \$011F

Next, that data is actually written out to the LCD. So, for this function to work properly, **you must first move the data** (e.g., an ASCII string) that you want displayed on the LCD into the appropriate locations in data memory. Then, call the function as follows:

```
rcall  LCDWrite     ; Write to both lines of the LCD
```

### LCDWrLn1

This function writes data to the first/top line of the LCD. First, the data is retrieved from data memory addresses \$0100 – \$010F, and then the data is written out to the first line of the LCD.

To use this function, make sure the data you want to display is in the appropriate locations in data memory, and then call the function as follows:

```
rcall  LCDWrLn1     ; Write to first line (Line 1) of the LCD
```

### LCDWrLn2

This function writes data to the second/bottom line of the LCD. First, the data is retrieved from data memory addresses \$0110 – \$011F, and then the data is written out to the second line of the LCD.

To use this function, make sure the data you want to display is in the appropriate locations in data memory, and then call the function as follows:

```
rcall  LCDWrLn2     ; Write to second line (Line 2) of the LCD
```

### LCDClr

This subroutine clears both lines of the LCD. Specifically, it writes the “space” character (ASCII value \$20) to data memory locations \$0100 – \$010F and \$0110 – \$011F, and then writes to both lines of the LCD. To call it, use:

```
rcall  LCDClr       ; Clear both lines of the LCD
```

### LCDClrLn1

This subroutine works similarly to LCDClr, except it only clears the first/top line of the LCD.

```
rcall  LCDClrLn1    ; Clear first line (Line 1) of the LCD
```

### LCDClrLn2

This subroutine works similarly to LCDClr, except it only clears the second/bottom line of the LCD.

```
rcall  LCDClrLn2    ; Clear second line (Line 2) of the LCD
```

### LCDWriteByte

This function allows you to write a single ASCII character (or byte) to anywhere on the LCD. It allows direct control over where data is displayed on the LCD, and does not require anything to be stored in data memory first (unlike many of the previous functions). There are three registers that must be loaded with input parameters before this function is called:

- `mpr` – Contains the byte/character that you want to display on the LCD (must be a value between 0 and 255)

- **line** – Contains the line number where you want to display the byte (must be either 1 for first line, or 2 for second line))
- **count** – Contains the index number where you want to display the byte (must be a number between 0 and 15, with 0 specifying the leftmost position). Note: The LCD actually accepts index values up to 39, but only indexes 0 through 15 are actually visible on the display. Any index value between 16 and 39 will result in the character being display “off screen”

The following example uses the `LCDWriteByte` function to write the character ‘D’ to Line 2, index 7 of the LCD:

```
; load parameters for LCDWriteByte
ldi    mpr, 'D'      ; load character 'D' into mpr
ldi    line, 2       ; load line number 2 into line register
ldi    count, 7      ; load index 7 into count register

; call LCDWriteByte function
rcall  LCDWriteByte ; write character to LCD
```

### Bin2ASCII

This function converts an unsigned 8-bit value into its numerically equivalent ASCII string, i.e. from  $138_{10} \rightarrow$  “138”. The resulting string (with length between 1 and 3 bytes) is stored in the data memory. This function is useful for displaying a value that changes throughout the course of a program (such as a counter).

Two registers must be loaded with input parameters before this function is called, and a third register must be available to store a returned value:

- **mpr** – Contains the 8-bit value that will be converted
- **XH:XL** – Contains the beginning 16-bit data memory address where the ASCII string will be stored
- **count** – Will contain the return value (the length of the created string)

The following example uses the `Bin2ASCII` function to convert the value  $138_{10}$  into “138”, and then store it in data memory beginning at location \$0112:

```
; load parameters for Bin2ASCII
ldi    mpr, 138      ; load to-be-converted value into mpr
ldi    XL, low($0112) ; load X with beginning address
```

```
ldi    XH, high($0112) ; of where result will be stored

; call Bin2ASCII function
rcall  Bin2ASCII      ; convert value in ASCII
```

After the above example is executed, the data memory locations \$0112, \$0113, and \$0114 will contain the characters ‘1’, ‘3’, and ‘8’, respectively, and **count** will contain the value 3.

If a smaller value like  $75_{10}$  had instead been converted, only locations \$0112 and \$0113 would contain characters, and **count** would have a value of 2.