NAME(S) _____

CS 341 – Lab 2
Computer Architecture and Organization
Introduction to the Harvard Architecture

Equipment: Arduino UNO microcomputer, PC with Arduino IDE installed, and a USB cable.

The ulab system CPU and the PC processors in our SAPC systems are built along the lines of the Von Neumann architecture. In this architecture, the processor stores both program code and program data (including the stack) in a single memory space. Code and data must be located at different addresses within that single memory space. The UNIX and SAPC memory maps shown in the lecture notes indicate the non-overlapping ranges of memory addresses used for code, data, and stack in each of those systems.

The processor used in the Arduino UNO (the ATMEGA328) is built along the lines of the Harvard architecture. In this architecture, the processor stores program code and program data/stack in separate memory spaces. The memory addresses used for code may appear to overlap the memory addresses used for data or the stack, but there is no conflict since they are located in separate memory spaces.

There are three memory spaces in the Arduino UNO processor architecture. See Figure 1. Note that addresses for each of the three memory spaces start with 0x0000 and go to a symbolic constant as the end address - FLASHEND, RAMEND, or E2END. These constants are defined in the IDE compiler and you will determine their hex values below.

The Flash (program code) and EEPROM (configuration data) are non-volatile meaning that their contents are preserved across power cycles. The RAM is volatile so its contents are lost when the power is turned off. The contents of RAM are initialized during the power on sequence in the program code. Note that the 32 general purpose registers and the I/O device registers are “memory mapped” into the data memory address space from 0x0000 to 0x00ff. The program memory is organized as 16 bit words while the data memory and EEPROM are organized as bytes.

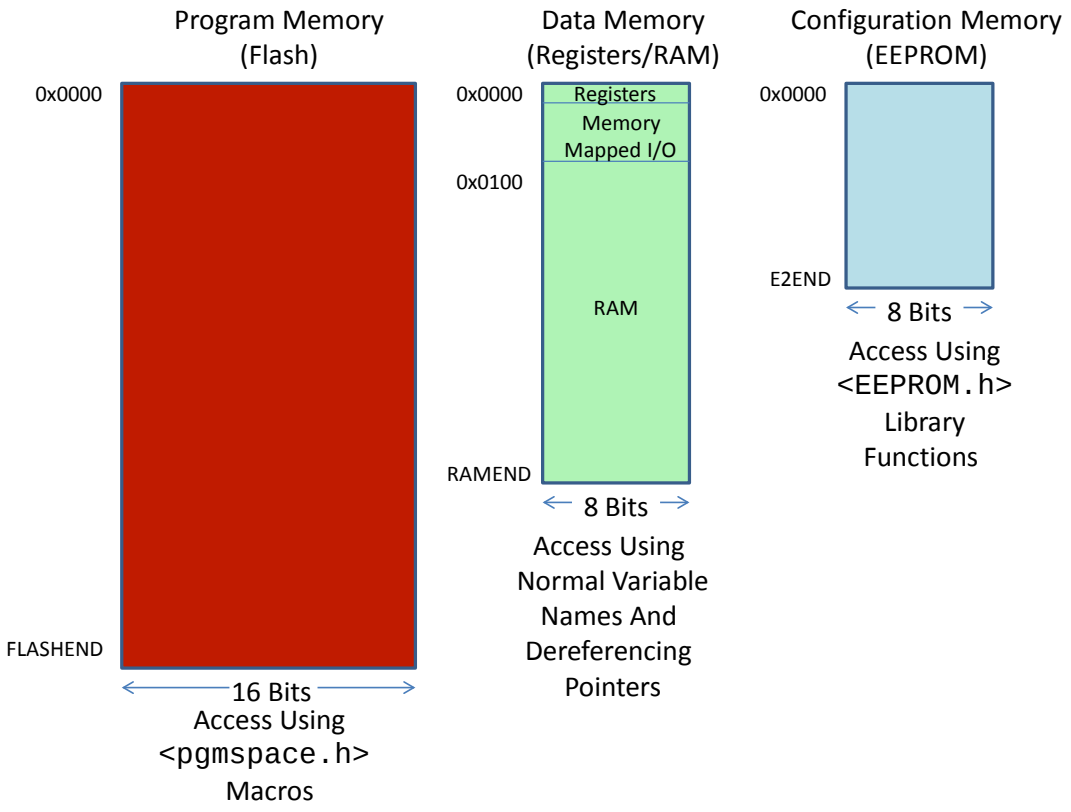

Figure 1

In this lab, you will explore the program and data memory spaces. In Lab 3, you will explore the EEPROM memory space. In Lab 4, you will explore the memory mapped I/O.

In C programming, there is only a single address space for pointers. You can find a review of pointers in the handout. In the Von Neumann architecture(where code and data can be accessed in one address space), a pointer's value can be the address of either data (an array) or program code (a function pointer) and dereferencing a function pointer can be used to access the code itself, i.e. the binary value of the instruction at that address. The C pointer space is mapped to the data memory space since that is the most common use for pointers. Although a function can be executed via dereferencing a function pointer, dereferencing the function pointer cannot be used to read the contents of program memory space.

```
 // correct usage of a function pointer in either architecture
   void (*pointer)() = &functionName;// defines and initializes the value of
a function pointer
   (*pointer) ();     // calls functionName via the function pointer

// incorrect usage of a function pointer in a Harvard architecture
   Serial.println((int) *((int *) pointer), HEX); // prints contents from RAM
data memory
                                                  // does NOT print contents
of program memory
```

The ATMEGA processor includes some special features that allow access to program memory as data (e.g. for downloading code). The Arduino IDE compiler and library have some extensions to allow use of these features. In the compiler, there is a special attribute called `PROGMEM`. This attribute can be used in global data definitions (outside the scope of all functions in the sketch) to tell the compiler to place the data in program memory instead of data memory. (If PROGMEM is used in an automatic variable definition (inside the scope of a function), the attribute is ignored and the memory will be allocated on the stack in RAM.) The library provides the macros `pgm_read_byte` and `pgm_read_word` which are defined in the `pgmspace.h` header file that allow your code to access program memory.

Write the setup function in a new sketch to perform the following experiments:

1. Print the hex values of the three constants for the end of each of the memory spaces. Fill in the values you get below:

   FLASHEND 0x_____
   RAMEND 0x_____
   E2END 0x_____

2. Define a function in your sketch that simply prints something to verify that the function was called. Define a function pointer and call the function via the pointer as described above. Then add code to print in hex various addresses and contents of addresses in program memory and RAM as follows:

   (int) &functionName 0x_____
   (int) &pointerName 0x_____
   (int) value of pointer variable 0x_____
   int value at pointer in RAM 0x_____
   (Cast your pointer variable to an int *, dereference it, and cast result to an int for printing.)
   int value at pointer in program memory 0x_____
   (Use one of the macros described above and cast return value to an int for printing.)

3. Define a global `char array1[]` with the PROGMEM attribute and initialize it with the string literal "Array 1". Print the array and the address of the array in Hex. (You will need to write a loop using one of the above macros to read the char value of each entry in the array before printing it.)
   Record the value: _____ (Note: This address should be in program memory.)

   Define a global `char array2[]` without the PROGMEM attribute and initialize it with the string literal "Array 2". Print the array and the address of the array in HEX.
   Record the value: _____ (Note: This address should be in the initialized data area in RAM.)

Define a local automatic variable `char array3[]` inside the { } of the setup function without the PROGMEM attribute and initialize it with the string literal “Array 3”. Print the array and the address of the array in Hex.
Record the value: _____ (Note: This address should be in the stack area in RAM.)

Define a local automatic variable `char *array4` inside the { } of the setup function without the PROGMEM attribute and initialize it by casting the return value from a call to `malloc(strlen(“Array 4”) + 1)` to `char *`. Use `strcpy` to copy “Array 4” into array4. Print the array and the address of the array (not the address of the pointer to the array) in Hex.
Record the value: _____ (Note: This address should be in the heap area in RAM.)

4. Print some other locations or ranges of locations in RAM if that would be useful to answer the question below for your report.

From the above data, try to determine the compiler’s rules for allocating addresses in RAM. What locations in RAM are used for the initialized data, the heap, and the stack? Include your conclusions and rationale supporting them in your report. Submit your report to the TA in your next lab session with a copy of the code for your final sketch.

____ / 10