Homework 4 Due: Monday (10/13) in Class (turn in two pieces of paper)

1) Read EdX Chapters 7 and 9: all sections; watch the videos
http://users.ece.utexas.edu/~valvano/Volume1/E-Book/

2) Log into Zybooks and read Sections 4.1, 4.2, 4.8, 5.1, 5.2, and 5.3

The purpose of this homework is to learn arrays in C programming.

You are allowed to work in groups of 2 on homework. Each student must turn in their own solution. If you will miss class you are allowed to turn in homework to your professor before class. To get credit for homework you must complete all questions, but the official score will be completion. I.e., we will not check the answers. However, the professors have answers to the homework, so if you are uncertain about your answers go to their office hours to check your answers against the solution key. We will not post the answers.

The exercises are practice programs to write, but not turn in. The assignment is a program you will write, debug and turn in. Create a one page print out of a screen shot showing some of the program code and run time output. In this homework we will look at basics of arrays and revisit the for-loop as a natural fit for traversing an array. We will also look at subroutines and parameter passing. If you are referring to Yale Patt's book, you may want to read chapters 13, 14 and 16.

Exercise 4.1: Do all activities in section 5.1, 5.2 and 5.4.
Assignment 4.1: Submit one sheet (screenshot) your code for Homework challenge 5.4.1 and 5.4.3 from the Zybooks C book.

To declare an array of a particular type of size **N** we use the declaration:

```
type arrayname[N];
```

The **arrayname** is the name by which the array will be referred. The choices for **type** are

unsigned char	means 8-bit unsigned.
char	means 8-bit signed.
signed char	means 8-bit signed.
unsigned short	means 16-bit unsigned.
short	means 16-bit signed.
signed short	means 16-bit signed.
unsigned long	means 32-bit unsigned.
long	means 32-bit signed.
signed long	means 32-bit signed.

Examples:

```
unsigned char anums[5];      // array of 5 8-bit unsigned numbers
signed char bnums[8];        // array of 8 8-bit signed numbers
```

```
  unsigned short scores[25];   // array of 25 16-bit unsigned numbers
  long data[200];              // array of 200 32-bit signed numbers
```

Notice the size of the array is a constant defined at compile time. To access individual elements of the array you use the square brackets with the index of the element you wish to access. Note that indexes start from 0. So, the first element's index is 0 and the last element's index is (Size-1). Starting at an index of zero is called "zero-indexing" and C always uses zero-indexing.

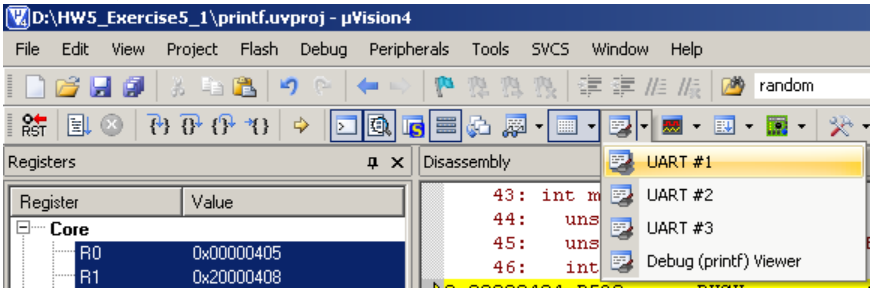
Exercise 4.1: Download and unzip **HW4_Exercise4_1.zip**. You can run this in the simulator and observe the output in a UART#1 window. Assume someone has given us the function **`Sum(n)`** to test. They claim this function calculates the sum of the numbers n down to 1. For example **`Sum(5)`** is 5+4+3+2+1 which equals 15. One approach to functional debugging is to collect input and output data for the software for typical input values. In particular we define a set of input values, run the software, and record the output data generated by the software. In this exercise we will test the software using input values from 0 to N-1, where N is 25. Furthermore, we will test the hypothesis that the sum of n numbers can be expressed as the simple calculation of $(n*(n+1))/2$.

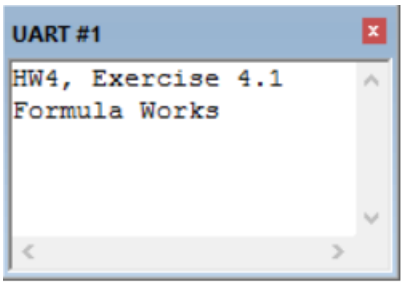
$$\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$$

This test program will evaluate the function **`Sum(n)`** and store the result in the array at **`SumBuf[n]`**. We define a second function **`Fun(n)`** that simply calculates $(n*(n+1))/2$. We have another array **`FunBuf`** that we use to store the first 25 calculations of this second function. We will check for equivalence of the two functions by comparing elements of **`SumBuf`** to the elements of **`FunBuf`**. In the previous homework examples, we employed a top-down approach by placing the main program on the top and the functions after. In this example, we will configure the system as bottom up by placing the functions first and the main program at the end. Notice that bottom up organization does not need prototypes for the functions. Run this example.

```
#define TRUE 1
#define FALSE 0
#define N 25 // The Size of the Array
unsigned long Sum(unsigned long n){
  unsigned long partial, count;
  partial = 0; // holds the running sum as we compute it
  for(count = 1; count <= n; count++){
    partial += count;
  }
  return partial;
}
unsigned long Fun(unsigned long n){
  return (n*(n+1))/2;
}
int main(void){
  unsigned long num;
  unsigned long SumBuf[N], FunBuf[N]; // arrays to store values
  int correct;
  UART_Init();              // initialize UART
  printf("HW4, Exercise 4.1 \n");
  for(num = 0; num < N; num++){
```

```
    SumBuf[num] = Sum(num);
    FunBuf[num] = Fun(num);
  }
  // Check if the formula and computation agree
  correct = TRUE; // Assume they match and change to false if mismatch
  for(num = 0; num < N; num++){
    if (SumBuf[num] != FunBuf[num]) {
      correct = FALSE;
    }
  }
  if(correct == TRUE){
    printf("Formula Works\n");
  } else{
    printf("Formula Wrong\n");
  }
}
```





Exercise 4.2: Download and unzip **HW4_Exercise4_2.zip**. You can run this in the simulator and observe the output in a UART#1 window. Write a program that searches an array of 16-bit numbers (call it haystack) to see if a particular value (call it needle) is present in it. You can declare an array and initialize it in one step like so: **`haystack[0]`** has 12, **`haystack[1]`** has 4, **`haystack[2]`** has 13, **`haystack[3]`** has 2, and **`haystack[4]`** has 15. We add the **`const`** qualifier for objects that are defined at compile time and not allowed to change during execution. On the microcontroller, constant arrays will be stored in ROM.

```
unsigned short const haystack[5] = {12, 4 , 13, 2, 15};
```

Your function will be passed a 16-bit unsigned number. Return a 1 (true) if that number is in the array **`haystack`**, and return a 0 (false if that number is not in the array **`haystack`**.

```
long InHay(unsigned short needle);  // prototype
unsigned short const haystack[5] = {12, 4 , 13, 2, 15};
void main(void){ unsigned short i;
  for(i=0; i<65535; i++){
    if(InHay(i)){
      printf("Number %5d is in the haystack.\n",i);
    }
  }
}
long InHay(unsigned short needle){
  // put your code here
```

```
  return 0; // replace this line
}
```

Your output should be something like this

```
HW4, Exercise 4.2
Number     2 is in the haystack.
Number     4 is in the haystack.
Number    12 is in the haystack.
Number    13 is in the haystack.
Number    15 is in the haystack.
```

Assignment 4.2: Download and unzip **HW4_Assignment4_2.zip**. You can run this in the simulator and observe the output in a UART#1 window. Write a function that counts the number of instances of letter in a string. The strings are null-terminated. All strings are 11 characters long (12 bytes including null.) Complete the implementation of the function **`Count`**, and test it using the following Keil project in **HW4_Assignment4_2**. You may use pointer or index syntax to access data from the string. Take a screenshot of the Keil debugger showing your code and the output results of running your code. Show this printed page to your TA at the start of class. *You are expected to write the function* ***`Count`***. It is not necessary that you understand the rest of the code. What you need to know is what the **`Count`** function is supposed to count the number of occurrences of letter in string. This main program is an example of functional debugging; it defines a set of input parameters, calls your function, and evaluates the equivalence of your output to the expected output.