# Lab #5 DSP Number Systems

## Objectives

1. Explore how several data types commonly used in DSP (byte, signed integer, long integer, float) affect memory usage and computation speed.
2. Explore the effects of finite precision math on calculations.
3. Measure the execution speed for addition and multiplication and compare how this speed changes using various number data types.

## Reading and Reference

Chapter 4 in Smith. http://Arduino.cc

## Intended Learning Outcomes

1. Demonstrate the use of an internal timer function to measure the execution time for a section of code.
2. Understand the impact of data types on execution speed and on memory usage.

## Overview and Background Material

In the previous lab on Analog to Digital Conversion (ADC), you collected temperature reading values and averaged these values in order to increase the effective resolution of the temperature measurement.

In embedded systems, the averaging must be done in real-time in a microprocessor or digital signal processing chip, not in a post-processing step such as MATLAB analysis. Consider the example of a medical instrument that is actively monitoring a patient. The averaging algorithm will have to execute quickly (e.g. < 1 millisecond) on a microprocessor chip with limited memory and speed rather than on a PC with 16GB RAM and a 3 GHz i7 processor.

How fast an algorithm runs depends on the number of instructions that have to be executed and how long each instruction takes to execute. Some instructions are faster than others e.g. addition in a microcontroller is faster than multiplication.

The datatype of the variable also affects how long it takes to execute an instruction. Typically, adding two integers is faster than adding two floating point numbers. However, it depends on how the microprocessor or DSP chip is built. If the chip has a hardware floating point processor, it may be able to quickly perform floating point calculations. Otherwise, floating point operations are typically slower than fixed point (integer) operations.

Knowing the execution speed of your particular microprocessor or DSP chip is critical because it affects the number of filter calculations that can be run in a given amount of time. If the filter takes longer to run, the bandwidth of the system will be lower. For example, you may not be able to perform a particular audio processing algorithm and have it run at the MP3 standard sample rate of 44.1kHz if there are too many calculations required for each sample.

Characterizing your system’s execution time can help you understand why it may misbehave when trying to run a particularly long filter.

**NOTE:** For this lab it is easier to use the serial monitor than MATLAB to capture output data. You will then make tables of your results using Excel. A set of blank Excel tables has been provided for you in the file “Lab 05 Tables.xlsx” located in myCourses.

# Section 1 Basic Math Operations and Data Types

In this section, you will add two numbers and examine the results obtained when different data types are used.

## Description of the code:

This code adds two numbers and prints the results.

## Procedure:

1. Paste the program code from the following text box into the Arduino IDE. This code adds two numbers and prints the results. **\* You will not use the temperature sensor or the dither circuit.**
2. Run the code initially using byte variables.
3. Record the results in the **Table 5-1a.**
4. Next, add your own code to the main loop, as indicated by the TODO, to run through a series of calculations using an array rather than manually changing the numbers each time. You may see some unexpected results.
5. Repeat step 4 using int, long, and float datatypes. Record the results in **Table 5-1a**.
6. In the first row of **Table 5-1b**, determine what positive integer value for the ‘b’ number will result in a ‘yv’ that is “254” when the datatype is byte. You do not need to fill in other values for this row.
7. In the last row of **Table 5-1b**, determine what positive integer value for the ‘b’ number will result in a ‘yv’ that is “-10” when the datatype is long. You do not need to fill in other values for this row.

**Question 5-0:** Were any of the floating point calculations for “yv” incorrect (i.e. different than you would expect)? If so, explain why. If not, explain why not.

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Datatypes.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// modified by: Clark Hochgraf Sept 28, 2019 -- added number arrays

// modified by: S. Ciccarelli 02/19/2023 -- Modified references/plot info to Lab #5

// purpose: Illustration of datatype effect on adding numbers

#define DATATYPE byte // change declaration to byte, int, long, float

DATATYPE av, bv, yv; // uses #define value to change datatype for each variable

// change declaration to byte, int, long, float

DATATYPE avl[16] = {100, 200, 20000, 100, 100, 100, 255, 255, 25500, 20000, 20000, 20000, 20000, 1000000000, 2000000000, 2000000000};

DATATYPE bvl[16] = {100, 200, 20000, 155, 156, 157, 100, 10, 25500, 12767, 12768, 12769, 12770, 1000000000, 147483649, 147483649};

void setup()

{

Serial.begin(9600);

Serial.println(F("Lab 5 datatypes 2225"));

Serial.println(F("\nEnter 'g' to go ....."));

while (Serial.read() != 'g'); // spin until 'g' entry

Serial.println("Data type is byte"); // byte, int, long, float

av = 100;

bv = 100;

yv = av + bv;

Serial.println("Add two numbers (100+100)");

Serial.print("a = "); Serial.print(av); Serial.print("\t"); Serial.print("\t");

Serial.print("b = "); Serial.print(bv); Serial.print("\t"); Serial.print("\t");

Serial.print("y = a+b = "); Serial.println(yv);

av = 200;

bv = 200;

yv = av + bv;

Serial.println("\nAdd two numbers (200+200)");

Serial.print("a = "); Serial.print(av); Serial.print("\t"); Serial.print("\t");

Serial.print("b = "); Serial.print(bv); Serial.print("\t"); Serial.print("\t");

Serial.print("y = a+b = "); Serial.println(yv);

av = 20000;

bv = 20000;

yv = av + bv;

Serial.println("\nAdd two numbers (20,000+20,000)");

Serial.print("a = "); Serial.print(av); Serial.print("\t"); Serial.print("\t");

Serial.print("b = "); Serial.print(bv); Serial.print("\t"); Serial.print("\t");

Serial.print("y = a+b = "); Serial.println(yv);

}

void loop() {

Serial.println("\nAdd numbers from array");

// TODO: Add a for-loop to run through all the 'a' and 'b' values in the

// arrays 'avl' and 'bvl'. Start at index 0 and run through all 16 values in the array.

// Inside the for-loop:

// 1) set av and bv to values picked from 'avl' and 'bvl'

// 2) perform the addition

// 3) print the a and b values and the result yv

// INSERT YOUR CODE HERE

// END INSERTION OF CODE

while (true) {}; // spin forever

}

**Table 5-1a – Addition with Various Data Types**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| input  a | input  b | Correct Mathematical Result a+b | actual result byte | actual result int | actual result long | actual  result float |
| 100 | 100 | 200 |  |  |  |  |
| 200 | 200 | 400 |  |  |  |  |
| 20,000 | 20,000 | 40,000 |  |  |  |  |
| 100 | 155 | ? |  |  |  |  |
| 100 | 156 | ? |  |  |  |  |
| 100 | 157 | ? |  |  |  |  |
| 255 | 100 | ? |  |  |  |  |
| 255 | 10 | ? |  |  |  |  |
| 25,500 | 25,500 | ? |  |  |  |  |
| 20,000 | 12,767 | ? |  |  |  |  |
| 20,000 | 12,768 | ? |  |  |  |  |
| 20,000 | 12,769 | ? |  |  |  |  |
| 20,000 | 12,770 | ? |  |  |  |  |
| 1,000,000, 000 | 1,000,000, 000 | ? |  |  |  |  |
| 2,000,000, 000 | 147,483, 649 | ? |  |  |  |  |
| 2,000,000,000 | 147483649 | ? |  |  |  |  |

**Table 5-1b. – Addition Errors with BYTE and LONG Data Types**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| input  a | Find Positive number b | Correct Mathematical Result a+b | actual result byte | actual result int | actual result long |  | actual  result float |
| 255 | (?) | //skip// | 254 | //skip// | //skip// |  | //skip// |
| 2,000,000,000 | (?) | //skip// | //skip// | //skip// | -10 |  | //skip// |

# Section 2 Illustration of Round-off Error with Floats

The round-off error associated with floating point data types can be illustrated by adding a random number to a float value and then subtracting the same number later. In this case, we add random numbers A and B to a constant value of 1.0 and then subtract them both from the original number. By comparing the ideal value to the actual value, you can see the roundoff error.

## Procedure:

1. Read the code from the text-box below to understand how it works.
2. Paste the code from the text-box into the Arduino IDE, load and run the file.
3. Compare the ideal output value and the actual output value and record the results in Table 5-2.
4. Modify the code so that the scale factor for the random numbers A and B is 10.0 instead of 1.0. Compile, upload and run the code, record your results and note how the error changes.
5. Compare the ideal output value, the actual value and record your results for all of the different random number scale factors listed in table 5-2.

**Table 5-2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scale factor for random numbers | Starting value of x | Ending value of x | The total error in percent after 1000 calculations | Average error in percent for one calculation |
| 1.0 | 1.0000000000 |  |  |  |
| 10.0 | 1.0000000000 |  |  |  |
| 100.0 | 1.0000000000 |  |  |  |
| 1000.0 | 1.0000000000 |  |  |  |
| 10000.0 | 1.0000000000 |  |  |  |
| 100000.0 | 1.0000000000 |  |  |  |

**Q5-1:** If the signal is the original value of x and the final value of x contains the signal plus noise, what is the resulting signal to noise ratio (in dB) of the final result when the random scale factor is 10,000.0?

**Q5-2:** Explain why the difference between the starting value of X (1.0) and the ending value of X was larger when the random numbers (A and B) were larger? In other words, why does the size of A and B affect the result?

**Q5-3:** Your lab partner says that if you use floating point numbers, that the calculations will always be accurate enough. Write a paragraph illustrating a case where this is not true and write a paragraph explaining what feature or characteristic of floating point numbers causes the problem.

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Roundoff\_1.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// modified by: S. Ciccarelli 02/19/2023 - fixed %error calculations and

// changed references to Lab #5

// purpose: Illustration of roundoff error when using float

const int NUM\_CALC = 1000;

float xv = 0.0, A, B, error, scale\_factor;

void setup()

{

Serial.begin(9600);

Serial.println(F("Lab 5 S2 datatypes 2225"));

Serial.println(F("\nEnter 'g' to go ....."));

while (Serial.read() != 'g'); // spin until 'g' entry

randomSeed(425);

scale\_factor = 100000.0; // start with 1.0

xv = 1.0;

Serial.print("Adding random floating point numbers with scale factor ");

Serial.println(scale\_factor);

Serial.print("x value starts at x = ");

Serial.println(xv,10); // print to 10 decimal places

for (int i = 1; i < NUM\_CALC; i++)

{

A = random\_float() \* scale\_factor;

B = random\_float() \* scale\_factor;

xv = xv+A; // add A to x

xv = xv+B; // add B to x

xv = xv-A; // subtract A from x

xv = xv-B; // subtract B from x

} // for

error=(xv-1.0);

// end result should just be the original value of x (i.e. 1.0)

Serial.print("x value finishes at x = "); Serial.println(xv,10);

Serial.print("Total percent error after 1000 calculations = ");

Serial.print(error\*100.0,10); Serial.println(" %");

Serial.print("Average percent error per calculation = ");

Serial.print(error\*100.0/(4.0\*NUM\_CALC),10); Serial.println(" %");

Serial.println();

} // setup()

void loop(){} // spin forever

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

float random\_float() { return (random(2147483648)/2147483648.0); }

# Section 3 Unexpected Behavior Due to Finite Precision Effects with Floats

The finite precision associated with floating point data types can cause unexpected results in many ways. Examine the code in the text-box below. The difference between two floating point numbers A=18000002.0 and B=18000001.0 is exactly 1. The two numbers are first printed and then the difference between the two numbers is calculated and printed.

Next in the code, the value of A is set to a counter variable and B is set to the counter variable minus 1. The counter value is continuously incremented until the value of A-B does not equal exactly 1, which should never happen, hence resulting in an infinite loop. Let the code run for a while and observe what happens.

**Procedure:**

1. Read the code from the text-box below to understand how it works.
2. Paste the code from the text-box into the Arduino IDE, load and run the file.
3. Examine the results. **Answer questions Q5-4 and Q5-5**
4. Modify the code so that variables A and B have long datatypes instead of float datatypes and change the definition of the initial values of A and B to:

A=18000002;

B=18000001;

**Not** the original:

A=18000002.0;

B=18000001.0;

1. Upload the code and observe the results. **Answer questions Q5-6, Q5-7**
2. Modify the code so that variables A and B have long datatypes instead of float but change the definition of the initial values of A and B to:

A=18000002.0;

B=18000001.0;

Be sure to include the decimal points. Upload and run the code. How are the results different than when no decimal point was included? **Answer question Q5-8**

**Q5-4:** In the first section of the code, why doesn’t the value of B print out as expected?

**Q5-5:** In the second section of the code, at what value of the counter does the program produce an unexpected result? Why does the error occur at this particular value of the counter? (hint: What is 2^24?)

**Q5-6:** When the data type was set to long, did the computer calculate the correct value of A-B when A= 18000002, and B=18000001?

**Q5-7:** When the datatypes were set to long, did the computer break out of the while loop at the same value of counter as when the datatype was float? What did happen?

**Q5-8:** What happened when the datatypes were set to long, but the initial values of A and B included a decimal point? Why did this happen?

// OPEN NEW ARDUINO SKETCH.

// CLICK IN THIS TEXT BOX. CTRL-A, CTRL-C.

// CLICK IN SKETCH. CTRL-A, CTRL-V.

// file: Roundoff\_2.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// modified by: Mark Thompson December 31, 2019

// purpose: Illustration of unexpected behavior due to roundoff error

float A = 18000002.0;

float B = 18000001.0;

//long A = 18000002;

//long B = 18000001;

//long A = 18000002.0;

//long B = 18000001.0;

void setup()

{

Serial.begin(9600);

Serial.print("A = float(18000002.0) value printed out is "); Serial.println(A);

Serial.print("B = float(18000001.0) value printed out is "); Serial.println(B);

Serial.print("A - B is "); Serial.println(A-B);

//-------------------------------------------------------------------

Serial.println("\nAnother illustration: Counting up by 1 ");

Serial.println("A is the counter value");

Serial.println("B is the counter value minus 1");

Serial.println();

long counter=16000000;

A = counter;

B = A-1.0;

Serial.print("Starting at a counter value of = "); Serial.println(counter);

Serial.print("A value is "); Serial.println(A);

Serial.print("B value is "); Serial.println(B);

Serial.print("A - B is "); Serial.println(A-B);

Serial.println();

Serial.println("Continue to count up by 1. Print the counter every 50,000 counts");

Serial.println("Check to see that the difference between A and B is always 1, error otherwise");

Serial.println();

while ((A-B) == 1)

{

if (counter%50000 == 0) Serial.println(counter);

counter = counter+1;

A=counter;

B=counter-1;

}

Serial.print("\nUnexpected result: Counter = ");

Serial.println(counter);

Serial.print("A value is "); Serial.println(A);

Serial.print("B value is "); Serial.println(B);

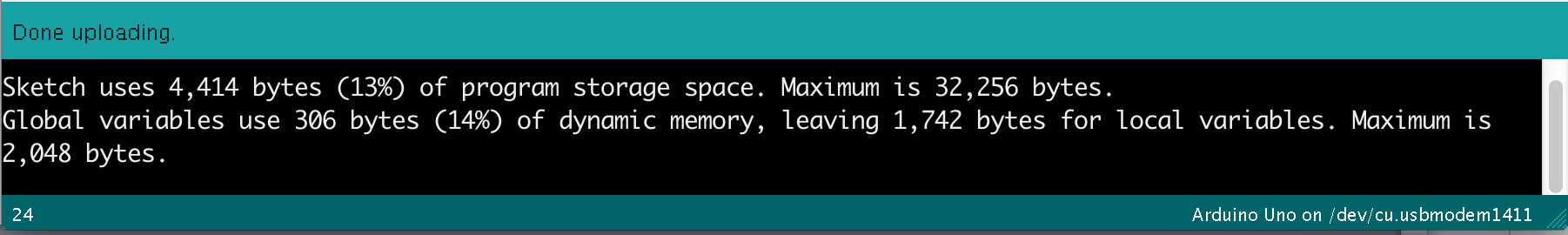
Serial.print("A - B is "); Serial.println(A-B);

} // setup

void loop() {} // spin forever

# Section 4: Measuring Execution Time and Memory Usage versus Datatype

The Arduino Uno microprocessor has a RAM memory space of 2048 bytes. While this may seem like a lot, it is very easy to use it all up. If the free RAM memory falls below 300 bytes, the microprocessor may produce irregular results or completely stop working. The amount of RAM usage is estimated and displayed each time you compile a program (Arduino IDE 1.57 or higher). An example is shown in the textbox below



Note that there are two types of memory usage displayed by the Arduino IDE. The first memory usage refers to FLASH memory where the program code is stored. The UNO has 32kB of FLASH for storing the program. The second is the RAM memory used for storing variables. The UNO has 2kB of RAM. You need to make sure that your program does not use up all of the RAM as the processor needs about 300 bytes of RAM for stack operations, etc.

## Execution Time

In this section you will measure the time it takes for a section of code to execute. The Arduino’s internal timer 0 can be used to count the number of microseconds that have elapsed since the microprocessor was reset. The timer value can be accessed by calling micros().

To measure how long a section of code takes to execute, first call micros() and record its value. Then, after the code runs, call micros() again to record the clock value at the end of the routine. The difference between these gives you the elapsed time in microseconds.

Using this approach, you will measure how long it takes to add two numbers, to multiply two numbers, and to multiply and add in one combined operation (called multiply and accumulate or MAC). The amount of time each operation takes will depend on the datatype (byte, integer, long, float) of the numbers you put into operation.

To change the datatype, you can use the **cast** function in C. For example, you can temporarily change an integer datatype to a float data type using the C statement:

xv[i]=(float)i\*(float)i;

This takes the integer variable i and makes it a float datatype for the calculation.

If the x variable is already a float, then the memory usage and execution speed will reflect the float datatype. To make the calculation with long datatype, change the code to:

xv[i]=(long)i\*(long)i;

and declare the xv[] array as a long datatype.

long xv[DATA\_LEN]={0}; // array of values initialized to zero

For this lab, the code is set up so you only need to change the datatype of result and xv[]. You should not need the cast function.

Alternative Algorithm Timing Measurement Method

Sometimes you may not be able to use the micros() command to measure execution time. For example, if your code modifies the operation of Timer 0 in the Arduino chip. Another way to measure algorithm execution time is to use a spare digital I/O pin and set it HIGH when the algorithm starts and then set it LOW when the algorithm ends. Use an oscilloscope to measure the pin pulse width to ascertain run time. Example:

startTime = micros(); becomes digitalWrite(timingPin, HIGH);

end Time = micros(); becomes digitalWrite(timingPin, LOW);

## Program Description

Unpack the code in the text-box below to a new Arduino sketch. Variables are declared to store the time values as measured in microseconds since the microprocessor was reset. The number of variables stored is changed by changing the value of:

const int DATA\_LEN = 200;

To modify the input array and resulting variable’s datatype, change the datatype in line

float xv[DATA\_LEN] = {0}, res = 1.0;

The execution time of adds, multiplies and multiply-accumulates is calculated using three different methods:

1. First, a set of 20 adds is computed without using a for loop. The timing result is calculated and displayed. The process is repeated for multiplications and multiply accumulates.
2. Second, a for loop is setup to perform the operations and the timing results are displayed.
3. Third, a for loop is used where the variables are arrays, indexed by the for loop iterator variable.

When using the for-loop, the calculations take longer because of the loop instruction’s overhead. To get the execution time per operation, total time is divided by the total number of operations.

// OPEN A NEW SKETCH WINDOW IN ARDUINO

// CLICK IN THIS BOX, CTL-A, CTL-C (Copy code)

// CLICK IN SKETCH, CTL-A, CTL-V (Paste code into sketch)

// file: Timing.ino

// created by: Clark Hochgraf Sept 15, 2015

// modified by: David Orlicki Sept 1, 2017

// purpose: Measuring execution time and memory usage as datatype is

// changed from byte to integer, long, float

const int DATA\_LEN = 200; // number of data array values

unsigned long startUsec, endUsec, execUsec;

byte xv[DATA\_LEN]={0}, seed = 1.01, res; // array initialized to zero

void setup()

{

Serial.begin(9600);

Serial.print("Data array length = "); Serial.println(DATA\_LEN);

res = seed;

startUsec=micros();

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

res=res+res; res=res+res; res=res+res; res=res+res; res=res+res;

endUsec = micros();

execUsec = endUsec-startUsec;

Serial.print("\nuSec per individual addition = ");

Serial.println(execUsec/20.0,2);

//----------------------------------------------

res = seed;

startUsec=micros();

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

res=res\*res; res=res\*res; res=res\*res; res=res\*res; res=res\*res;

execUsec = micros()-startUsec;

Serial.print("uSec per individual multiplication = ");

Serial.println(execUsec/20.0,2);

//----------------------------------------------

res = seed;

startUsec=micros();

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

res=res+res\*res; res=res+res\*res; res=res+res\*res; res=res+res\*res;

execUsec = micros()-startUsec;

Serial.print("uSec per individual multiply and accumulate = ");

Serial.println(execUsec/20.0,2);

//----------------------------------------------

res = seed;

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++)

{

res = res+res;

// Serial.print(i); Serial.print('\t');

// Serial.println(res);

}

execUsec = micros()-startUsec;

Serial.print("uSec per loop addition = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ res = res\*res; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop multiplication = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values and add to original value (accumulate)

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ res = res+res\*res; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop multiply and accumulate = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// Add two values using arrays

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ xv[i] = xv[i]+xv[i]; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop array addition = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values using arrays

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ xv[i] = xv[i]\*xv[i]; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop array multiplication = ");

Serial.println((float)execUsec/DATA\_LEN,2);

//----------------------------------------------

// multiply two values and add to original value (accumulate) using arrays

startUsec=micros();

for (int i=0; i <DATA\_LEN; i++){ xv[i] = xv[i]+xv[i]\*xv[i]; }

execUsec = micros()-startUsec;

Serial.print("uSec per loop array multiply and accumulate = ");

Serial.println((float)execUsec/DATA\_LEN,2);

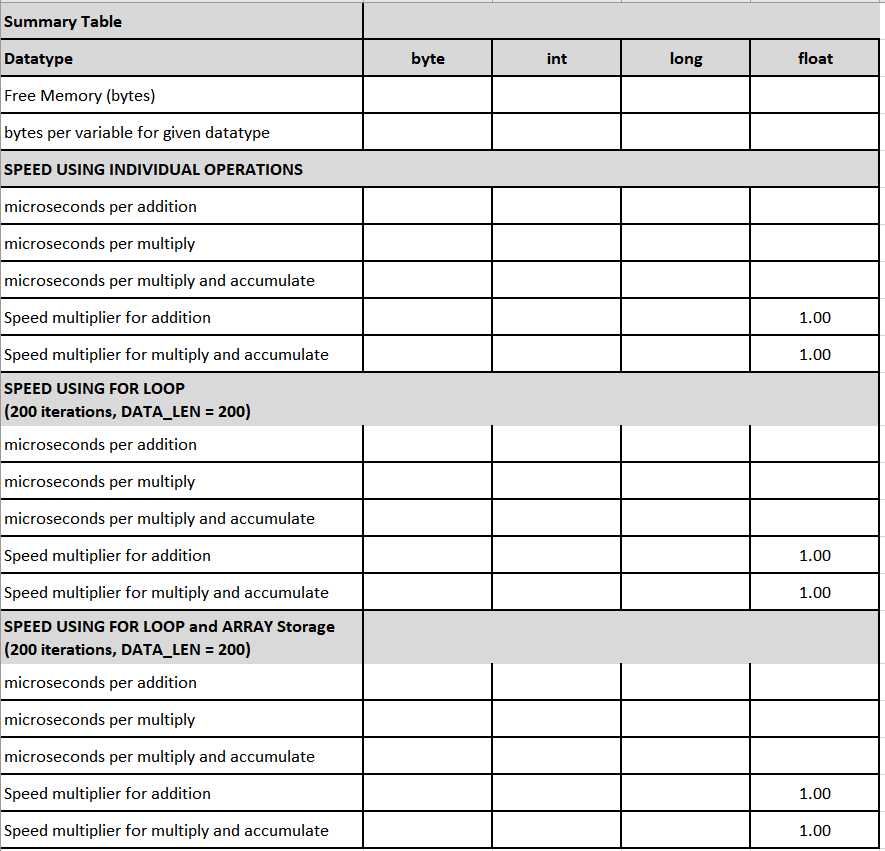
}

void loop(){ } // spin forever

## Procedure

1. Paste the file from the text-box into the Arduino IDE. Make sure to select all the text in the textbox. (Use CTRL-A, then CTRL-C.)
2. Change the datatype from float to integer, long and byte. Remember to check that the right values are being printed and divided by the number of additions, etc.
3. **Record the data on the microprocessor performance in Table 5-3.** For the speed multipliers**, calculate them in Excel** **after you have collected all the data**. The speed multiplier indicates how much faster an operation is relative to the corresponding floating point operation. For example, addition of two bytes using the for loop and array storage is about 8x faster than the addition of two floats. Results will vary slightly from board to board but the multipliers in some cases are very significant.

Table 5-3 Memory and Execution Times as a Function of Input Variable Datatypes



**Observations**

Generally, floating point operations are the slowest to execute. While byte, integer and longs are faster, you must use them carefully to ensure they produce valid results. One approach in using integer datatypes is to pre-scale numerical values coming into the algorithm so that the results stay within the dynamic range of the datatype (e.g. -32768 to 32767 for integers).

**Tips**

1. The bytes per variable is a number that you can look up in the Arduino documentation online or find on the Arduino user support forum.
2. You can also calculate the bytes per variable by running the program with DATA\_LEN =1 and recording the number of bytes of free memory. Then increase DATA\_LEN to 101. Now look at the change in free memory. Divide the number of additional bytes used by 100 (the number of additional variables) and you can get the bytes per variable.

## Report Submission:

Refer to the rubric below. Include the requested plots, tables and answers to questions in a narrative format discussing the results and their implications.

Be sure to fully label the axes and title graphs. Be sure to include descriptive captions for each figure, table and graph.

Use the IEEE Journal Format to complete your report. Maintain the report title and Author line. Use the 2-column format for the remaining portions of the report. Maintain the correct font and margins. Submit your team report in PDF format.

Make sure you also submit a relatively detailed work-breakdown document in either PDF, TXT or DOC format.

**Grading Rubric – Digital Signal Processing LAB 05: DSP Number Systems**

|  |  |  |  |
| --- | --- | --- | --- |
| Lab Section | Result Description | Points Available | Points Obtained |
| Abstract | Abstract providing a brief overview of your work, including your methods and key findings. | 10 |  |
| Section 1 | Correct and properly labeled tables 5-1a and 5-1b. Narrative answer to question Q5-0 | 15 |  |
| Section 2 | Correct and properly labeled tables 5-2. Narrative answers to question Q5-1 through Q5-3 | 15 |  |
| Section 3 | Narrative answers to question Q5-4 through Q5-8 | 15 |  |
| Section 4 | Complete and correct Table 5-3.  Narrative description of the results of this section. | 15 |  |
| Conclusion | Conclusion summarizing your main findings, along with the significance of these findings and suggestions for follow-up work. | 15 |  |
| FORMAT | IEEE Journal format followed. Proper grammar and spelling. | 15 |  |

**Instructor comments:**