Arduino-Based Dataloggers: Hardware and Software

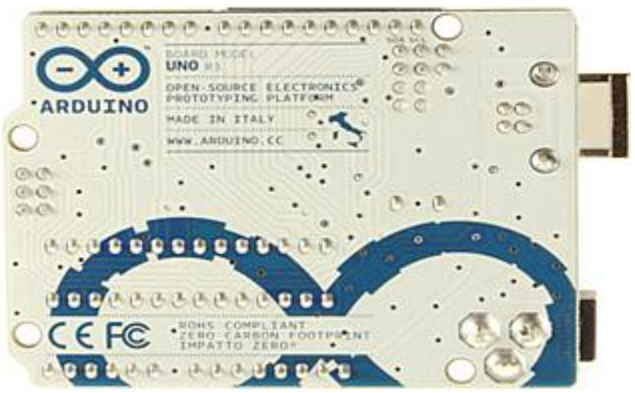
David R. Brooks

Institute for Earth Science Research and Education

V 1.3, February, 2016

© 2014, 2015, 2016

An introduction to Arduino microcontrollers and their programming language, with the goal of building a high-resolution datalogger to record data from external sensors.



A file containing the text of all sketches used in this document can be downloaded at www.instesre.org/ProgrammingGuideSketches.txt.

CONTENTS

1. INTRODUCTION

In recent years, there has been an explosion of interest in microcontrollers. One of the most successful and widely used systems is the Arduino, started as a student project in 2005 at the Italian *Interaction Design Institute Ivrea*. Since then, that Institute has closed but the Arduino project lived on. Now, this open-source hardware/software system has spawned numerous "cloned" versions and given birth to a new and rapidly growing industry devoted to making use of its capabilities. I believe it is reasonable to equate the impact of the microcontroller revolution in the early 21st century to the personal computer revolution at the end of the 20th century. It might even be fair to conclude that being comfortable working with microcontrollers is as essential for any technically competent individual in the 21st century as "personal computing skills" (which many of us now take for granted) were in the late 20th century.

Because microcontrollers provide essentially unlimited opportunities for interfacing with hardware, skills can be developed in many ways with projects ranging from the frivolous to the profound. This document is not intended as a general-purpose Arduino reference guide, but only as a record (with apologies for any errors from a "newbie") of the path I followed to develop an Arduino-based high-resolution datalogger. I started with a scientific programming background, but with absolutely no previous microcontroller programming experience. The document includes many links to online sources which I found essential for acquiring the information I needed to reach my goal.

Here is the hardware used for this project. Prices from www.adafruit.com and www.sparkfun.com are approximate as of July 2014. Quantity discounts may be available.

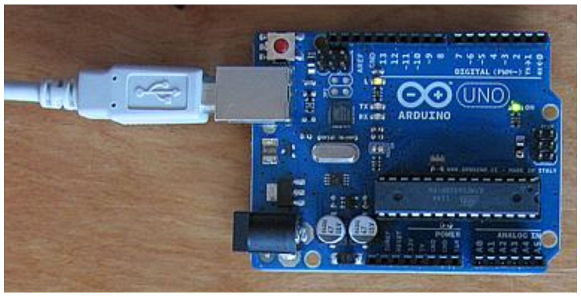


Figure 1. Arduino Uno R3 microcontroller.

• Arduino Uno R3 microcontroller (ID 50, $25) or Adafruit Arduino Uno R3 starter pack (PID 68, $65.00)
• Adafruit data logging shield with light and temperature sensors (PID 249, $37.50), and, optionally, additional data logging shields (PID 1141, $20)
• Adafruit ADS1115 16-bit analog-to-digital conversion (ADC) board (ID1085, $15)
• Sparkfun Arduino Pro microcontroller (DEV-10915, $15)
• Sparkfun FTDI board (DEV-09716, $15)
• A few other components for testing code, as described in figures below

http://arduino.cc/en/Main/arduinoBoardUno contains a description of the "classic" Uno R3 board. http://www.gammon.com.au/forum/?id=11473 has a useful pin diagram.

The software component of any Arduino project requires some general programming knowledge plus details about Arduino hardware and the Arduino programming language. If you already have experience programming with C/C++ or related languages, you may find much of this document to be unnecessarily tedious and you will be able to skip over large portions of it, with perhaps some occasional detours (here or online) to check specific features of the Arduino language.

Arduino-compatible boards and accessories like the Arduino Pro are available from www.sparkfun.com and other sources. Some of these boards have the advantage of using less power than the Uno R3, but I recommend the Uno R3 board as a starting point to learn about using these microcontrollers.

2. THE ARDUINO INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

2.1 Up and Running…

The Arduino project development environment, or integrated development environment (IDE) is a free download for Windows, Mac, or Linux systems from http://arduino.cc/en/main/software. There is no point reading this document until you have installed the IDE software. The work described in this document has been done on a Windows XP computer. Once installed in an Arduino folder, everything is in place to try some of the examples in the `\examples` folder, which will be created when you install the IDE. Connect the Arduino board to your computer through a USB port, which will provide enough power for the Arduino board to operate without an external power supply. Note that the Arduino board uses relatively a lot of power compared, for example, to a commercial datalogger such as the Onset Computer Corporation's UX120-006M 4-channel voltage logger (http://www.onsetcomp.com/products/data-loggers/ux120-series), which will run for many months on two AAA batteries. An Arduino (plus some accessories) will run continuously from a powered USB port, but you will need a relatively hefty battery supply to run an Arduino continuously on its own for extended periods of time. (See the Section 4.3 for more details.)

The `arduino.exe` file opens the IDE with a window for writing code. The source code for any Arduino application has a `.ino` extension. Every source code file is contained in its own folder, with the same name as the `.ino` file. This folder is created automatically whenever you create a new code file. In Arduino-speak, source code written using English/math-like instructions is called a "sketch." (The name is based on The Arduino language's origins in *Processing*, a programming environment for graphic design.) As a first example, open the `Blink.ino` sketch file found in the `\examples\01.Basics\Blink` folder. Choose Upload from the file menu. The code will be compiled into machine language and sent to the Arduino. If there are errors, messages will be displayed. As is often the case, error messages may or may not be helpful for fixing your code! If everything is working, the LED on the Arduino board should blink – one second on and one second off. If this simple sketch works, it is an indication that software and hardware are working together as required.

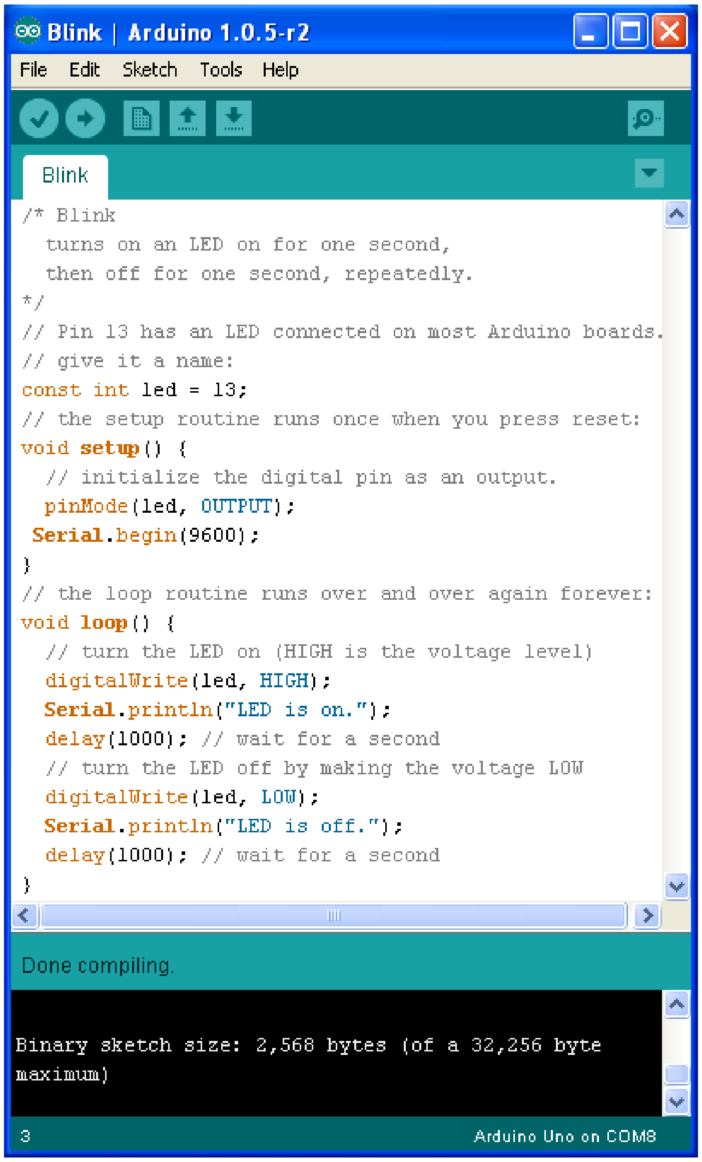
Note that you cannot "turn off" a program once it is sent to the Arduino. If you remove the power (either by removing the USB cable or by unplugging a power supply), the program will stop running. But it is still in the Arduino's memory and that same program will start running again if you power up the board again. It will stay in memory until you upload a different sketch. (Try it with Sketch 1.)

Sketch 1 is a modification of the `Blink.ino` sketch. It includes code which uses the `Serial` object to display output from a program in a serial port monitor window. This is how you keep track of what your program is doing and, often, display intermediate values to help with code debugging. As will be shown later, the serial port monitor window can also be used to provide input data to a sketch while it is running. The `Serial.begin(9600)` method opens the port at a communication speed appropriate for the Uno R3 board. See *2.2.5 Serial communication and displaying output* for information about the `Serial` object and its methods. See *2.3.1 Digital pins* for information on `digitalWrite()`.

You can make changes at any time to a sketch, re-compile it to make sure you haven't introduced any new errors, and then upload the new version. It is not necessary to save changes before you upload them. Do not save your own sketches in the "read only" `\examples` folder. You can create a new sketch there, but the IDE will not save a modification of that sketch in `\examples`. You can create another folder for your sketches in the `\Arduino` folder, or you can just save them in the `\Arduino` folder.

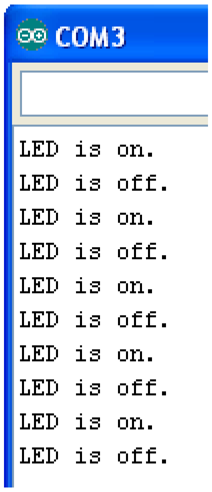
Because the Arduino system relies on software and hardware working together, there are many opportunities for problems to arise. It is far beyond the scope of this document to try to provide troubleshooting advice. But, extensive online support is available for this widely supported open source system. I have registered with online forums for Arduino and Adafruit customers. (You have to register to post questions.) I have always gotten prompt and useful advice from these forums and other online sources when I encountered a problem.

The two most common reasons why a successfully compiled sketch won't work are: (1) the correct Arduino board has not been selected; (2) the wrong COM port has been selected. These values are accessed through the Tools tab. The current board and COM port values are shown in the lower right-hand corner of the IDE window. The COM port may be different for different boards even of the same model. You may need to disconnect the USB cable and plug it in again to get the IDE to select the appropriate COM port.



```
/* Blink
  turns on an LED on for one second,
  then off for one second, repeatedly.
*/
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
const int led = 13;
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
 Serial.begin(9600);
}
// the loop routine runs over and over again forever:
void loop() {
  // turn the LED on (HIGH is the voltage level)
  digitalWrite(led, HIGH);
  Serial.println("LED is on.");
  delay(1000); // wait for a second
  // turn the LED off by making the voltage LOW
  digitalWrite(led, LOW);
  Serial.println("LED is off.");
  delay(1000); // wait for a second
}
```

```
Done compiling.

Binary sketch size: 2,568 bytes (of a 32,256 byte
maximum)
```



```
LED is on.
LED is off.
LED is on.
LED is off.
LED is on.
LED is off.
LED is on.
LED is off.
LED is on.
LED is off.
```

Sketch 1. Turn an LED on and off.

2.2 The Arduino Programming Language

Arduino boards are deceptively small – the Uno R3 board is about the size of a credit card – but they have many of the capabilities of a "real" computer. The maximum size of the allowed code is smaller than allowed on a bigger computer, but within that constraint, the computational possibilities are extensive. The major difference between microcontroller programming and "conventional" programming (for scientific and engineering computation, for example) is that the essential purpose of microcontroller programming is to control hardware. In this document, the hardware of interest is restricted mostly just to those devices needed to construct a datalogger. This section will mostly ignore hardware interfaces in favor of focusing on some programming fundamentals as they are implemented with Arduino. A few language syntax elements are shown in Table 1. Many more syntax elements are discussed in the following sections.

Arduino uses a C/C++-based language. Its syntax and structure are similar to other languages such as JavaScript and PHP. Anyone with experience programming in any of these languages should have no difficulty with Arduino programming logic, although some hardware-specific language components may be unfamiliar.

The Arduino language makes use of the "object" concept. In programming, objects are code constructions which define "attributes" that describe properties of the object and "methods" which define procedures associated with the object. For example, in Arduino programming there is a `Serial` object which includes methods for displaying output on and reading output from the serial communications port (for example, `Serial.print()` and `Serial.read()`), as will be described below.[1] For almost all Arduino projects, you do not need to know anything about how programming languages define and implement objects – it is enough to understand how to use them in their proper context.

Table 1. Some language syntax.

Language syntax	symbol or word
`;`	Required to terminate code statement. Multiple semicolon-terminated statements can appear on the same line.
`{…}`	Define a block of code.
`//`	Define a one-line comment.
`/*…*/`	Define multi-line comments.
`#define`	Give a name to a constant value to be used throughout a sketch. No equal sign or semicolon line terminator. `#define PI 3.14`
`#include`	Include external libraries. No semicolon line terminator. `#include <RTClib.h>`
`const`	A keyword to mark defined variables as "read only." The value of such variables cannot be redefined later in the code. Usually preferred over `#define`. `const float x=3.333;` … `x=0; // Not allowed!`

If you wish to make use of Arduino's capabilities there is no substitute for understanding its language and writing your own code! If you have no previous programming experience, you can learn a lot about programming in general and Arduino programming in particular by studying the examples in this document and from a huge amount of code available online. You can find a language reference at http://arduino.cc/en/Reference/HomePage and there is an Arduino programming "style guide" at http://arduino.cc/en/Reference/StyleGuide.You can find an Arduino programming tutorial with many examples at http://playground.arduino.cc/uploads/Main/arduino_notebook_v1-1.pdf. There are also dozens (hundreds?) of books about Arduino programming.

2.2.1 The Minimum Sketch

Every Arduino sketch requires both a `setup()` and `loop()` function even if `loop()` doesn't do anything.

[1] Newer computers, including modern laptops, don't have an external serial port connector, but internally they will support serial communications.

If the `loop()` function is empty, there must be something in the `setup()` function or else the sketch won't do anything at all. Some of the examples in this section don't do anything inside the `loop()` function.

```
void setup() {
  // Put your setup code here, to run once.
}
void loop() {
  // Put your main code here, to run repeatedly.
}
```

2.2.2 Data type examples, type conversions, and operators

Table 2 shows the data types supported by Arduino programming: integers and real numbers, characters, and Boolean values. See the discussion of the real time clock code, below, for more information about using long integer variables. Note the use of scientific notation (with e or `E`) for expressing real numbers. In some cases, it is possible to convert variables from one type to another. For example, it might be desirable to convert an integer value into its corresponding floating point value, or an `int` to a `long` integer (but not from `long` to `int`!) A `char()` conversion will convert an integer value to its corresponding ASCII character. Applying `int()` to a character will return the ASCII value for that character. Boolean variables, `true` or `false`, are represented by one-byte integer values equal to 1 or 0.

Table 2. Data types and conversions.

Data type	Example	Type conversion
`int`	`int a, b=0, c;`	`int(x)`
`long`	`long e,f,g;`	`long(x)`
`float`	`float x, y=.333, z=17e-6;`	`float(x)`
`char`	`char c1='A';`	`char(x)`
`boolean`	(only two possible values, `true` or `false`) `boolean pinHigh = false;`	

There is no built-in data type for strings (of characters) in Arduino programming. There are two ways to construct strings. There is a `String` object which includes methods for creating and manipulating strings. See http://arduino.cc/en/Reference/StringObject for more information. A simpler but less flexible method is to define a string as an array of characters. See *2.2.7 Arrays* for more information about arrays. Creating a string as an array of characters requires less computing resources than the `String` object, and should be used unless you actually need `String` object methods.

Sketch 2 shows some examples of data type conversions and strings. Note that strings constructed as an array of characters do not have to be displayed one character at a time in a `for`… loop. (See Section *2.2.3 Conditional and repetitive execution*.) Look up a table of ASCII characters to check the character-to-integer and integer-to-character conversions done in Sketch 2: for example, the character c has a base-10 ASCII value of 99.

Table 3. Math operators.

Math operation	Operator, compound operator
assignment	`=`
addition	`+, +=`
subtraction	`-, -=`
multiplication	`*, *=`
division	`/, /=`

integer modulo	%, %=

Table 3 gives math operators along with their compound versions. The assignment operator looks like the algebraic “equals” sign, but its interpretation in programming is *entirely* different. It means, “Evaluate an expression on the right side of the assignment operator and assign that value to the variable name on the left side of the assignment operator.” Hence,

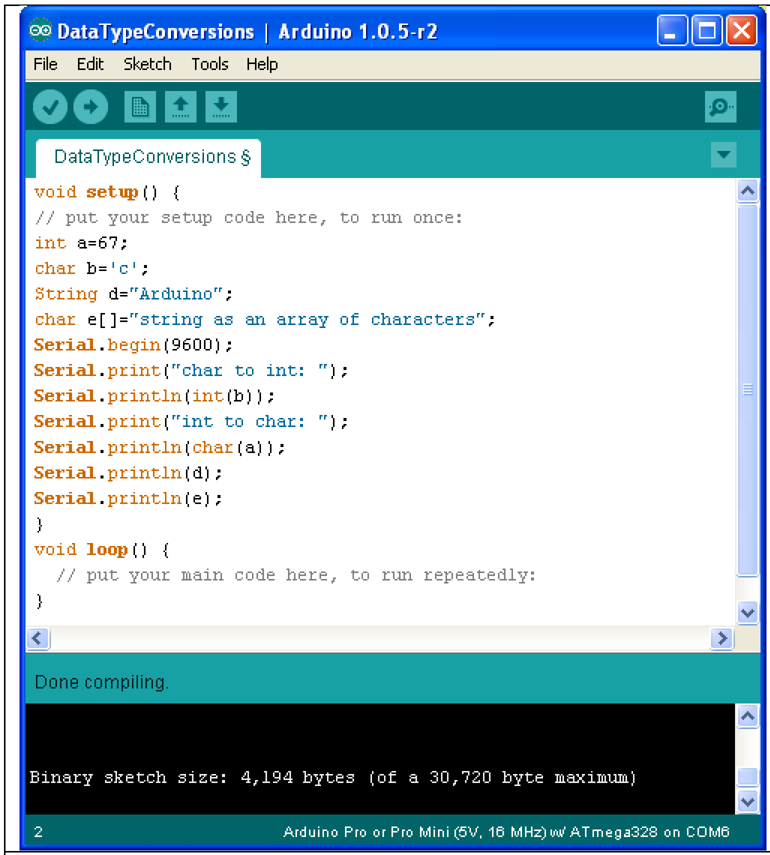
```
int x=3; x=x+3;
```

makes no algebraic sense, because x cannot be equal to itself plus 3, but makes perfectly good sense in programming. When these two statements are executed, x has a value of 6.

Compound operators provide a shorthand method for certain arithmetic operations. For example,

```
int x=3; x+=3;
```

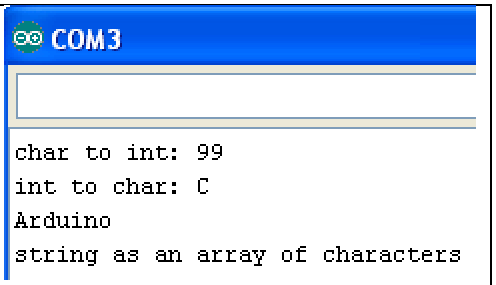
is completely equivalent to the previous two statements.


DataTypeConversions | Arduino 1.0.5-r2
File Edit Sketch Tools Help
DataTypeConversions §

```
void setup() {
// put your setup code here, to run once:
int a=67;
char b='c';
String d="Arduino";
char e[]="string as an array of characters";
Serial.begin(9600);
Serial.print("char to int: ");
Serial.println(int(b));
Serial.print("int to char: ");
Serial.println(char(a));
Serial.println(d);
Serial.println(e);
}
void loop() {
  // put your main code here, to run repeatedly:
}
```

```
Done compiling.

Binary sketch size: 4,194 bytes (of a 30,720 byte maximum)
```

```
2          Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328 on COM6
```



```
char to int: 99
int to char: C
Arduino
string as an array of characters
```

Sketch 2. Examples of data type conversions and strings.

The result of a division operation depends on the nature of the numerator and denominator. Hence,

```
int x=3,y=6; z=x/y;
```

gives a value of 0, but either

```
float x=3,y=6; z=x/y;
```

or

```
int x=3; float y=6; z=x/y;
```

gives a value of 0.5.

When you use numerical values in an expression for which you expect a real-number division result, at least one of those values should include a decimal point in order for values to be treated as real (floating point) numbers rather than integers. For example,

```
float z; z=2./3.;
```

rather than

```
float z; z=2/3; // The result is 0 even though z is declared as float.
```

or

```
float z; int x=2; z=x/3.;
```

rather than

```
float z; int x=2; z=x/3; // (the result is 0)
```

Unlike integer arithmetic, real-number math calculations are not necessarily exact. So, when determining whether the results of two real-number calculations are identical, it is a better idea to compare the absolute magnitude of their difference to some arbitrarily small value rather than testing them for equality. (See `if`… constructs in *2.2.3 Conditional and repetitive execution*, and Sketch 8.)

```
if (fabs(x – y)<10e-10) …; // then x and y are considered equal?
```

The `%` operator returns the remainder from integer division: `7%5` equals 2. There is no modulo operator for real numbers, but see the `fmod(x,y)` math function in Sketch 8, below.

There are also some bitwise operators, which are discussed in the language reference linked above. There are comparison and Boolean operators, as shown in Table 4.

Table 4. Comparison and Boolean operators.

Operation	Operator
Comparison operators	
equal to	`==`
not equal to	`!=`
less than	`<`
greater than	`>`
less than or equal to	`<=`
Greater than or equal to	`>=`
Boolean operators	
AND	`&&`
OR	`||`
NOT	`!`

There are many subtleties involved in using arithmetic, comparison, and Boolean operators. If you are unfamiliar with programming fundamentals, at some point you *will* have problems! As just one example, remember that `x==y`; is NOT the same thing as the assignment statement `x=y;`. The former statement tests for equality and the latter assigns the current value of `y` to `x`. Confusing the equality operator with the assignment operator is a common coding error that is very difficult to debug. Be careful!

2.2.3 Conditional and repetitive execution

The standard Arduino IDE installation includes several sketches that demonstrate some of the language features in this section. Those examples tend to be more hardware-oriented. For example, the `IfStatementConditional.ino` sketch uses a potentiometer attached to an analog pin.

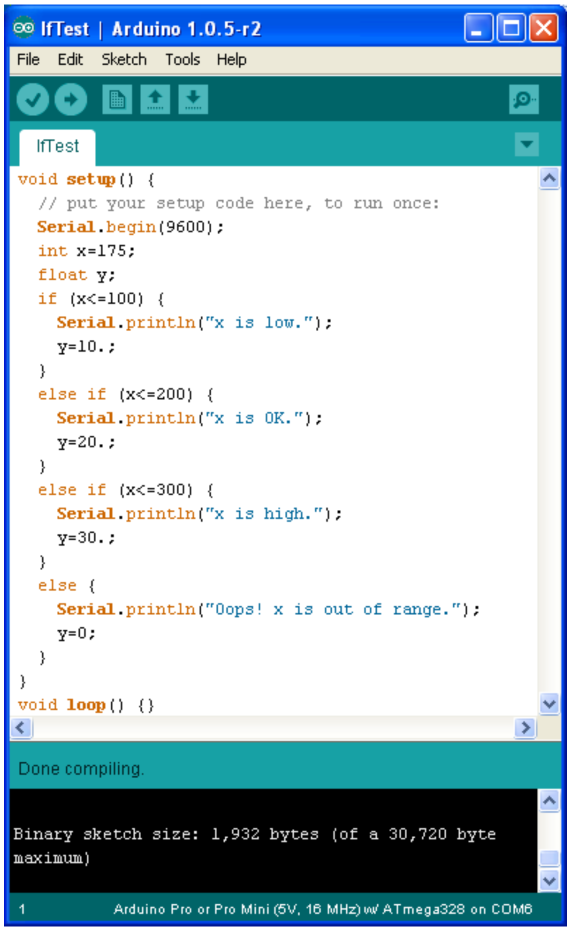
`if…` constructs

The `if…` construct allows blocks of code to be executed or not, depending on the value of a Boolean expression (true or false). In compound `if…` statements, only the block of statements corresponding to the first "true" condition (if such a condition exists) is executed. (Here and elsewhere in this document, code or "pseudocode" enclosed in square brackets means that that code is optional.)

```
if (Boolean expression) {
  …
}
[else if (Boolean expression) {
  …
}]
[else (Boolean expression) {
  …
}]
```

For processing more than two conditions, you can use as many `else if`… statement blocks as you need.

For Sketch 3, with x=175, the message "`x is OK`." is displayed on the serial port monitor. Then the rest of the `if`… construct is ignored. That is, even though it is also true that x is less than 300, this portion of the construct is never executed.

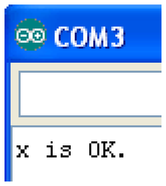


```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  int x=175;
  float y;
  if (x<=100) {
    Serial.println("x is low.");
    y=10.;
  }
  else if (x<=200) {
    Serial.println("x is OK.");
    y=20.;
  }
  else if (x<=300) {
    Serial.println("x is high.");
    y=30.;
  }
  else {
    Serial.println("Oops! x is out of range.");
    y=0;
  }
}
void loop() {}
```

Done compiling.

```
Binary sketch size: 1,932 bytes (of a 30,720 byte
maximum)
```

1 Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328 on COM6



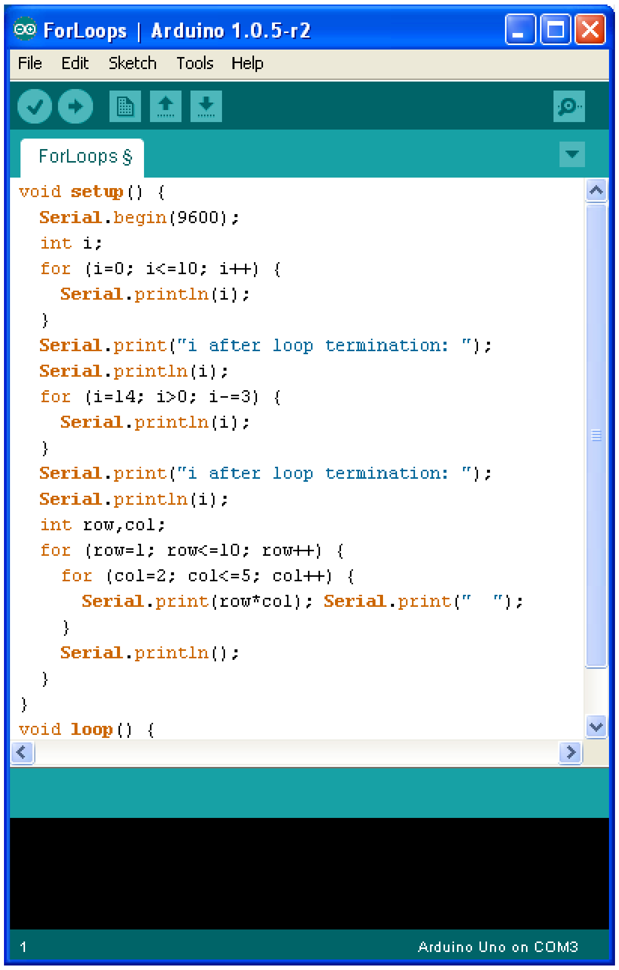
```
x is OK.
```

Sketch 3. Example of `if`… construct.

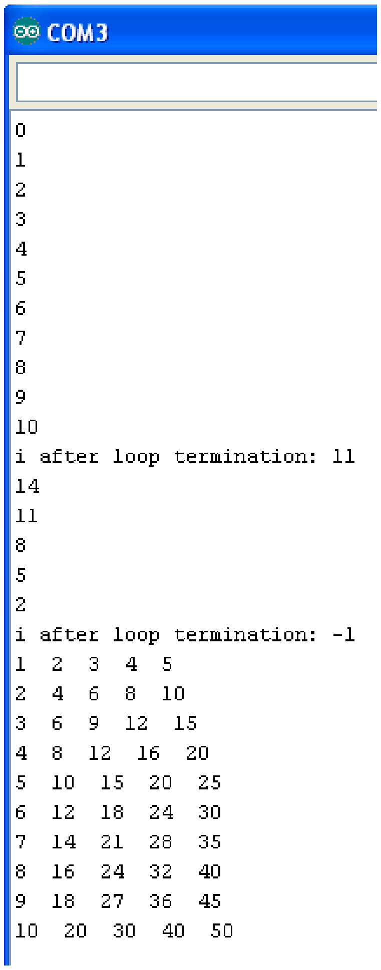
<u>`for`… loops</u>

`for`… loop constructs are for executing a block of statements a specified number of times. These loops are often used to access and process elements of an array (see *2.2.7 Arrays*, below). They can also be used to control hardware operations.

```
for (knt = lower or upper
limit; test knt against upper or lower limit; decrement or increment
operation on knt ) {
…
}
```



```
void setup() {
  Serial.begin(9600);
  int i;
  for (i=0; i<=10; i++) {
    Serial.println(i);
  }
  Serial.print("i after loop termination: ");
  Serial.println(i);
  for (i=14; i>0; i-=3) {
    Serial.println(i);
  }
  Serial.print("i after loop termination: ");
  Serial.println(i);
  int row,col;
  for (row=1; row<=10; row++) {
    for (col=2; col<=5; col++) {
      Serial.print(row*col); Serial.print("  ");
    }
    Serial.println();
  }
}
void loop() {
```



```
0
1
2
3
4
5
6
7
8
9
10
i after loop termination: 11
14
11
8
5
2
i after loop termination: -1
1  2  3  4  5
2  4  6  8  10
3  6  9  12  15
4  8  12  16  20
5  10  15  20  25
6  12  18  24  30
7  14  21  28  35
8  16  24  32  40
9  18  27  36  45
10  20  30  40  50
```

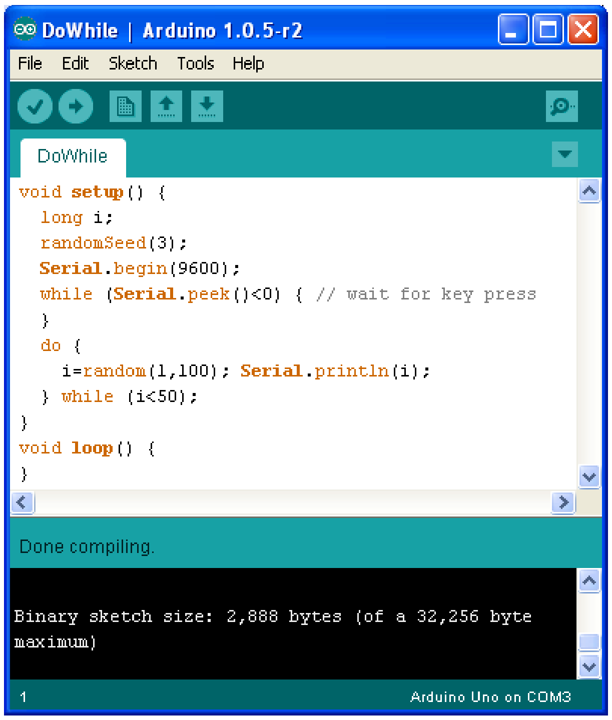
Sketch 4. Some examples of `for`… loops.

The integer loop control variable `knt` (you can name it whatever you want) is assigned an initial value. It is then incremented or decremented as specified and continues to be incremented or decremented as long as its value meets the defined condition. Often, `knt` is incremented or decremented by 1 for each pass through the loop, but other values are perfectly reasonable as long as the logic is correct and the termination condition is defined properly – with improper increment/decrement and termination conditions, it is possible to define a loop that will never terminate. The increment or decrement operation is done (automatically) after the last statement inside the loop. When the loop is complete, `knt` has a value equal to one increment or decrement step past the last value for which the loop statements were executed. Sketch 4 should make this clear. It is also clear from this sketch that the loop counter variable (named `i` in this example) can be re-used in more than one loop in the sketch. Finally, Sketch 4 shows that it possible to nest loops. In this example, nested loops are used to calculate values in the rows and columns of a two-dimensional table.

It is possible to use the `break` command (see *2.2.4 Some other program flow control statements*) to exit from a loop when some condition other than the specified loop terminating condition is met, but this kind of "conditional" control is best implemented with the constructs discussed next.

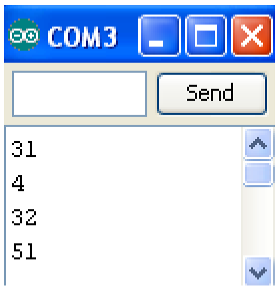
`while`… and `do`… `while` loops

These conditional execution constructs allow a block of statements to be executed repetitively as long as certain conditions are met, as opposed to `for…` loops, in which the number of executions is set ahead of time. The statements inside a `while…` loop may not be executed at all, depending on the initial value of a Boolean expression controlling execution. `do… while` loops are always executed at least once because the comparison with the Boolean expression is done at the end of the loop rather than at the beginning. It is possible to write loops that will *never* terminate, and it is a programmer's responsibility to make sure this doesn't happen.



```
void setup() {
  long i;
  randomSeed(3);
  Serial.begin(9600);
  while (Serial.peek()<0) { // wait for key press
  }
  do {
    i=random(1,100); Serial.println(i);
  } while (i<50);
}
void loop() {
}
```

```
Done compiling.

Binary sketch size: 2,888 bytes (of a 32,256 byte
maximum)
```



```
31
4
32
51
```

Sketch 5. Conditional execution loops.

In Sketch 5, the code waits for you to press a key and [Enter] in the serial port monitor. See *2.2.6 Math functions*, below, for more information about using the random number generator. The output from this sketch is always the same set of values between 1 and 100. These values are "random" in the sense that they would pass statistical tests for randomness. Note that the loop terminates *after* seeing a value ≥50 because the test is done at the bottom of the loop.

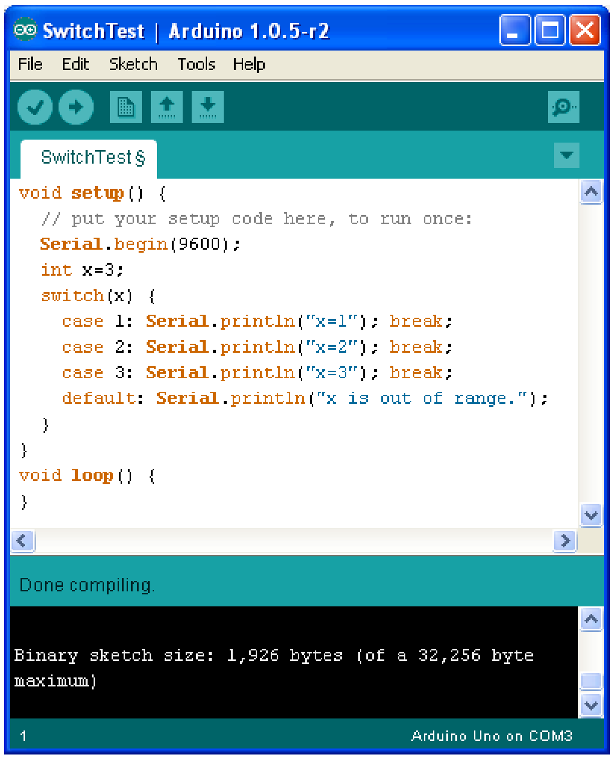
`switch` construct

The `switch` construct controls execution based on matching a value with a list of integer or character values. (It won't work with real numbers.) The `case` values in the list don't have to be in any particular order. This construct is often more clear than using a lengthy `if… else if…` statement. However, unlike `if…` statements, which execute only the first true branch, each `case` in the `switch` construct requires a `break`; statement to exit when the first match with `int` *`value or variable`* is found; without a `break;` all the other remaining operations will also be executed. The `default` keyword provides the opportunity for responding to *not* finding a match. Often, this response might be to display a message explaining that no match was found.

```
switch (toMatch) {
  case choice1:
      Do something when toMatch equals cboice1.
      break;
  case choice2:
      Do something when toMatch equals choice2.
      break;
  [additional cases…]
  [default:
      Do something if toMatch doesn't match any available choice.]
}
```

The switch construct cannot be used for matching with real numbers. Use an `if…` construct instead.

In Sketch 6, the message `x=3` is displayed in the serial port monitor. If the value of `x` is changed to 4, the message `x is out of range` will displayed.



```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  int x=3;
  switch(x) {
    case 1: Serial.println("x=1"); break;
    case 2: Serial.println("x=2"); break;
    case 3: Serial.println("x=3"); break;
    default: Serial.println("x is out of range.");
  }
}
void loop() {
}
```

```
Done compiling.

Binary sketch size: 1,926 bytes (of a 32,256 byte
maximum)
```

Sketch 6. Example of `switch` construct.

<u>Pre-compile directives for conditional execution</u>

Table 1 gave two examples of pre-compile directives, `#include` and `#define`. These directives can be used to alter what the Arduino IDE "sees" when it compiles your sketch. For example, the `#include` directive results in having the specified `.h` file literally copied into your sketch before it is compiled.

The `#if … #endif` pre-compile directive is used for including or excluding blocks of statements *before* a sketch is compiled. In this example, the pre-compile directive is used to turn output to the serial port on or off:

```
#define ECHO_TO_SERIAL 1 //"true" (1) echoes data, "false" (0) doesn't
…
#if ECHO_TO_SERIAL
  Serial.println(… whatever you wish to display);
  …
#endif //ECHO_TO_SERIAL
```

You can call the test "variable" whatever you want. `ECHO_TO_SERIAL` was chosen because it describes the purpose of using the directive.

Pre-compile directives are very useful language features when you are writing and debugging code. You can "turn on" printing to the serial port while you are developing your code and then, just by changing the value of `ECHO_TO_SERIAL` from 1 to 0, turn it off when everything is working correctly, rather than having to remove or comment out all the code that is no longer needed. For a stand-alone data logger, there is probably no reason to waste code writing results to the serial port in addition to writing data to a file on an SD card. If you need to make changes and monitor the results, just change the value back to 1 again.

Note that the same results could be obtained by setting a variable name to true or false and then using `if`… statements to include or bypass code. But in that case, *all* the code is still included in the compiled sketch and that could waste a lot of memory. With the `#if`… `#endif` directive, excluded code is simply ignored when the sketch is compiled – a potentially important consideration given the Arduino's relatively restricted memory for code.

2.2.4 Some other program flow control statements

```
break
continue
goto… label:
return [value]
```

The `break` statement is required for use in the `switch` construct described above. It is also used to exit from loops, but some programmers deprecate this practice. Most programmers believe that `goto` statements should *never* be used because they can result in code that is difficult to debug and maintain. Nonetheless, both these statements have some legitimate uses if they are used sparingly and appropriately.

The `return` statement is typically used to return a value from a user-defined function (see *2.2.8 User-defined functions*), for example:

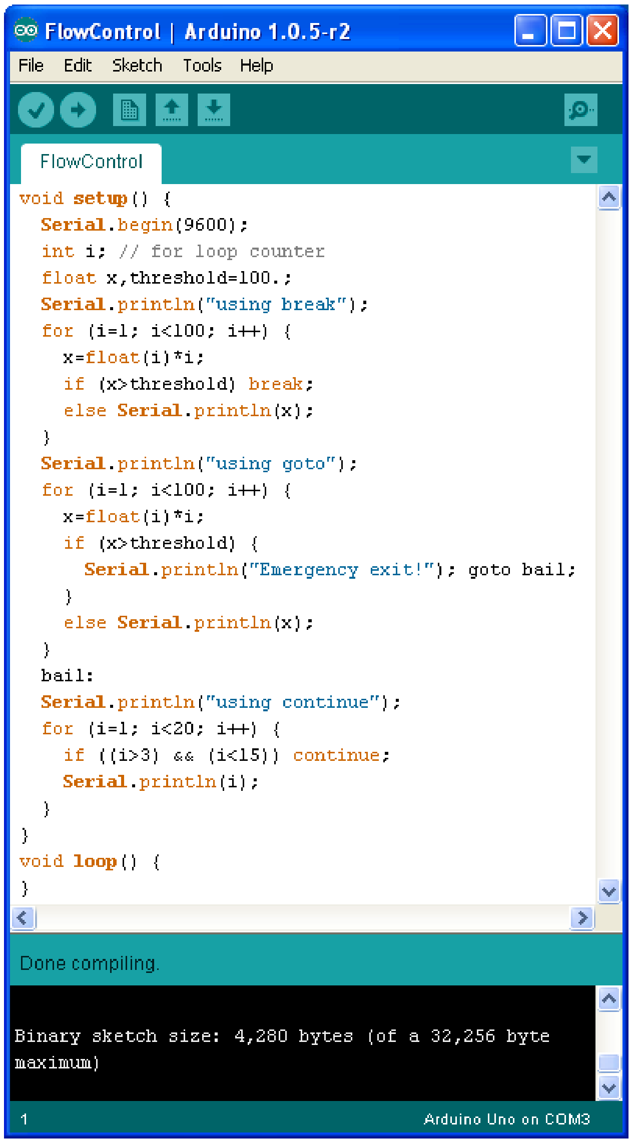
```
int checkSensor(){
    if (analogRead(0) > 400) {
        return 1;
    else{
        return 0;
    }
}
```

See below for more information about functions.

A `return;` statement can also be used to exit a function before some code is executed. Sometimes this is more convenient than commenting out unwanted code with /*… `*/`.

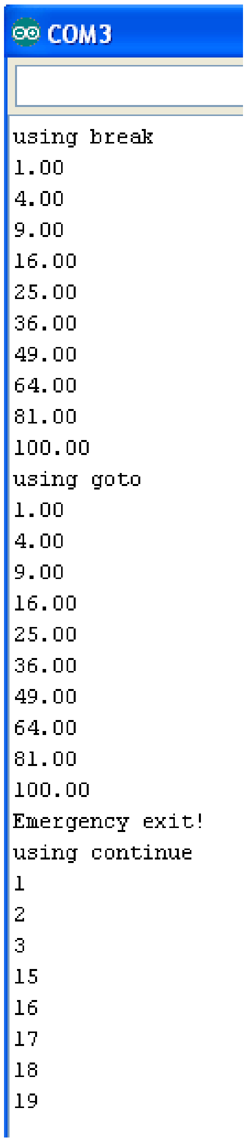
```
void myFunction(){
// good code here
return;
// code you want to be ignored
}
```

Sketch 7 shows some examples of using these statements.



```
void setup() {
  Serial.begin(9600);
  int i; // for loop counter
  float x,threshold=100.;
  Serial.println("using break");
  for (i=1; i<100; i++) {
    x=float(i)*i;
    if (x>threshold) break;
    else Serial.println(x);
  }
  Serial.println("using goto");
  for (i=1; i<100; i++) {
    x=float(i)*i;
    if (x>threshold) {
      Serial.println("Emergency exit!"); goto bail;
    }
    else Serial.println(x);
  }
  bail:
  Serial.println("using continue");
  for (i=1; i<20; i++) {
    if ((i>3) && (i<15)) continue;
    Serial.println(i);
  }
}
void loop() {
}
```

```
Binary sketch size: 4,280 bytes (of a 32,256 byte
maximum)
```



```
using break
1.00
4.00
9.00
16.00
25.00
36.00
49.00
64.00
81.00
100.00
using goto
1.00
4.00
9.00
16.00
25.00
36.00
49.00
64.00
81.00
100.00
Emergency exit!
using continue
1
2
3
15
16
17
18
19
```

Sketch 7. Some program flow control statements.

2.2.5 Serial communication and displaying output

As is evident from the sketches shown so far, output from a sketch – text or numerical values – can be displayed by sending the output to a serial port with `Serial.print()` and `Serial.println()`. Here's a summary of those and other methods.

`Serial.begin(`*`baud_rate`*`)`

Opens a serial port.

Example:

```
Serial.begin(9600) //9600 is the baud rate for many applications.
```

`Serial.peek()` reads a byte in the serial buffer, but doesn't move past that character. One use of this method is to wait until a user presses a key in the serial port monitor window. (See Sketch 5.)

```
Serial.print(x[,n])
Serial.print(str)
```

Displays value of `x` or a string in the serial port monitor. The optional parameter *`n`* specifies the number of digits to the right of the decimal point to include in the display of a floating point number. The default value, without specifying *`n`*, is two digits to the right of the decimal point.

```
Serial.println(x[,n])
Serial.println(str)
```

Like `Serial.print()`, but appends an end-of-line mark to the output.
Example:

```
Serial.print("Here is some output: ");
Serial.println(3.333,6);
```

The serial port monitor will display: `Here is some output: 3.333000`. As is the case with other high-level programming languages, the process whereby numerical values are translated into printable characters is interesting, but almost certainly of no concern to users of the language.

`Serial.read()` reads and returns a byte in the serial buffer and moves to the next byte.

```
Serial.write(val)
[int bytesSent =] Serial.write(str)
```

The first method will write a single byte to the serial port. *`val`* could be an integer or a character that can be represented in one byte, e.g., 57 or Z. The second method will write a string of characters to the serial port. Optionally, you can read the number of bytes written to the port.

2.2.6 Math functions

The Arduino microcontroller's small size can be deceptive! The Arduino language supports many math functions which make it possible to do sophisticated data processing in a datalogger sketch that would otherwise have to be done offline in some other software application.

As noted previously, the Arduino language supports integers and real (floating point) numbers. See the discussion of the real time clock code for more information about signed and unsigned integer constants and variables. The Arduino language does not have a separate "double" floating point number type for higher-precision calculations – all real numbers are treated as "double."

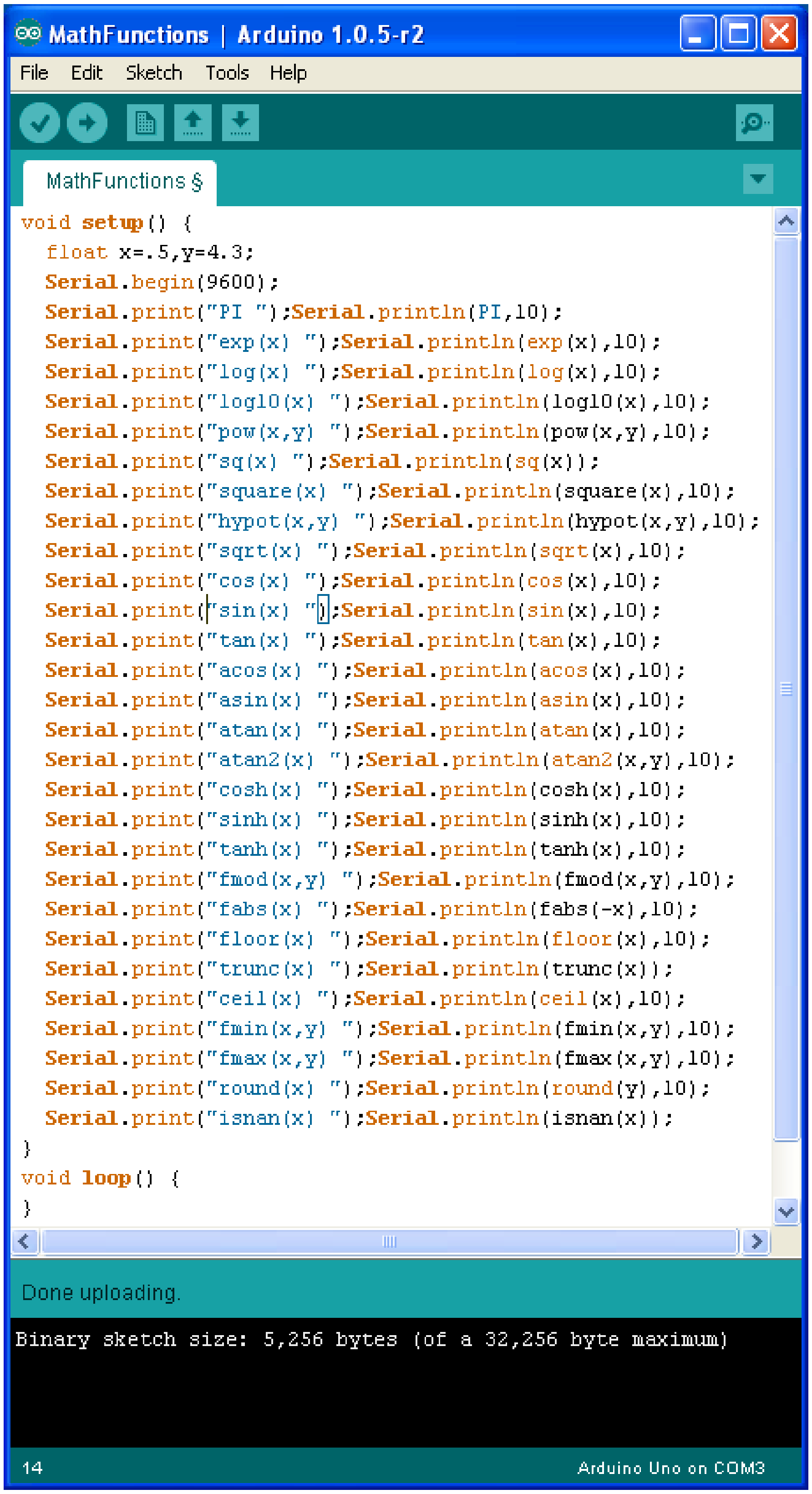
The language reference home page, at http://arduino.cc/en/Reference/HomePage, gives a short and incomplete list of "built-in" (if that's the right term) math functions. However, Arduino includes support for the much more extensive set of functions found in the `Math.h` library (http://www.nongnu.org/avr-libc/user-manual/group__avr__math.html) even though no `Math.h` library folder is shown as part of the Arduino IDE installation. In Sketch 8, the "built-in" math functions display in orange font, but other functions don't. For example, the "`log`" in `log(x)` is displayed in orange font, but the "`log10`" in `log10(x)` isn't. (Why? I don't know. All that matters is that both functions work.)

Note that `PI` (uppercase) is a built-in defined constant. As is the case for C-based and many other languages, the names of everything are case-sensitive, so `pi` is not the same as `PI`.

See Sketch 5, above, for an example of using Arduino's random number generator. The `random([min[,max])` function generates pseudo-random long integers, optionally between specified minimum and maximum values. `randomSeed(i)`, where `i` is an integer value, causes the random number always to start at the same point in its sequence (depending on the value of `i`).

"Pseudo-random" numbers are not really "random." They are generated by an algorithm based on an initial value (a "seed") in a way that the resulting numbers should pass statistical tests for randomness. See http://arduino.cc/en/Reference/RandomSeed for more information on these two functions, including how to start the random number sequence at a different position every time the sketch runs.

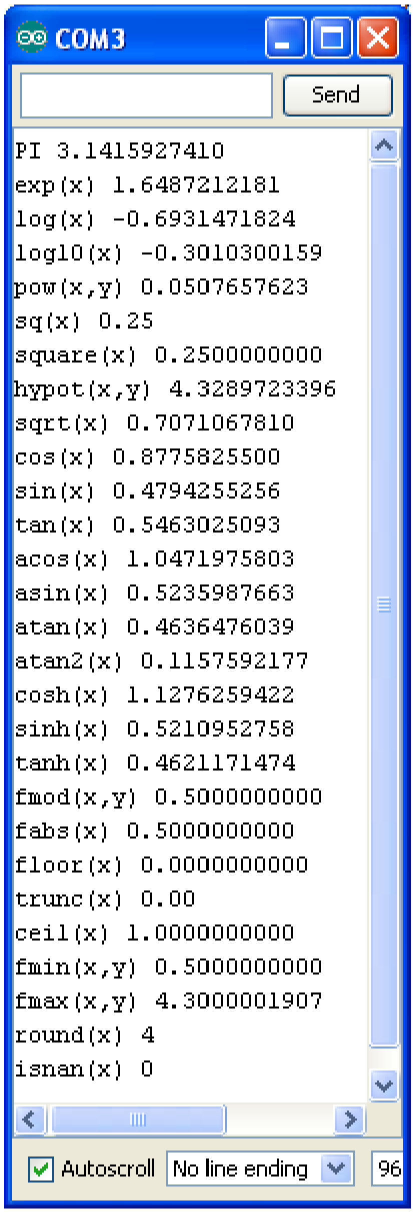
All trigonometric functions accept as input and return as output angles in radians, not degrees: radians = degrees × π/180 and vice versa to convert radians back to degrees. `sin(30)` will cause no problems when you compile your sketch, but if you really want the sine of 30º, you must use `sin(30*PI/180.)`. It is up to the programmer to use all math functions appropriately – for example, by not asking for the square root of a negative number. It is possible that some of these functions might be computationally intensive enough to cause memory and/or performance problems with sketches. If so, that would favor minimizing the numerical processing done within a datalogger program. The only way to find problems is to try your code!



```
void setup() {
  float x=.5,y=4.3;
  Serial.begin(9600);
  Serial.print("PI ");Serial.println(PI,10);
  Serial.print("exp(x) ");Serial.println(exp(x),10);
  Serial.print("log(x) ");Serial.println(log(x),10);
  Serial.print("log10(x) ");Serial.println(log10(x),10);
  Serial.print("pow(x,y) ");Serial.println(pow(x,y),10);
  Serial.print("sq(x) ");Serial.println(sq(x));
  Serial.print("square(x) ");Serial.println(square(x),10);
  Serial.print("hypot(x,y) ");Serial.println(hypot(x,y),10);
  Serial.print("sqrt(x) ");Serial.println(sqrt(x),10);
  Serial.print("cos(x) ");Serial.println(cos(x),10);
  Serial.print("sin(x) ");Serial.println(sin(x),10);
  Serial.print("tan(x) ");Serial.println(tan(x),10);
  Serial.print("acos(x) ");Serial.println(acos(x),10);
  Serial.print("asin(x) ");Serial.println(asin(x),10);
  Serial.print("atan(x) ");Serial.println(atan(x),10);
  Serial.print("atan2(x) ");Serial.println(atan2(x,y),10);
  Serial.print("cosh(x) ");Serial.println(cosh(x),10);
  Serial.print("sinh(x) ");Serial.println(sinh(x),10);
  Serial.print("tanh(x) ");Serial.println(tanh(x),10);
  Serial.print("fmod(x,y) ");Serial.println(fmod(x,y),10);
  Serial.print("fabs(x) ");Serial.println(fabs(-x),10);
  Serial.print("floor(x) ");Serial.println(floor(x),10);
  Serial.print("trunc(x) ");Serial.println(trunc(x));
  Serial.print("ceil(x) ");Serial.println(ceil(x),10);
  Serial.print("fmin(x,y) ");Serial.println(fmin(x,y),10);
  Serial.print("fmax(x,y) ");Serial.println(fmax(x,y),10);
  Serial.print("round(x) ");Serial.println(round(y),10);
  Serial.print("isnan(x) ");Serial.println(isnan(x));
}
void loop() {
}
```

Done uploading.

```
Binary sketch size: 5,256 bytes (of a 32,256 byte maximum)
```



```
PI 3.1415927410
exp(x) 1.6487212181
log(x) -0.6931471824
log10(x) -0.3010300159
pow(x,y) 0.0507657623
sq(x) 0.25
square(x) 0.2500000000
hypot(x,y) 4.3289723396
sqrt(x) 0.7071067810
cos(x) 0.8775825500
sin(x) 0.4794255256
tan(x) 0.5463025093
acos(x) 1.0471975803
asin(x) 0.5235987663
atan(x) 0.4636476039
atan2(x) 0.1157592177
cosh(x) 1.1276259422
sinh(x) 0.5210952758
tanh(x) 0.4621171474
fmod(x,y) 0.5000000000
fabs(x) 0.5000000000
floor(x) 0.0000000000
trunc(x) 0.00
ceil(x) 1.0000000000
fmin(x,y) 0.5000000000
fmax(x,y) 4.3000001907
round(x) 4
isnan(x) 0
```

Sketch 8. Examples of Arduino math functions.

2.2.7 Arrays

An array is a collection of values (elements) that can be accessed by name and an index number. Index values always start at 0, not 1. The size of an array must be part of its declaration, either explicitly or implicitly. That is, memory space for arrays is allocated statically, not dynamically. This means that unlike with some languages, such as PHP, you cannot define additional array elements later in your code. You must either declare the size without specifying values, or leave the size blank and declare values enclosed in curly brackets (from which the code compiler will infer the size). Here are some examples of array declarations:

```
int IntArray[10];
int counters[]={1,2,3,4,5};
float data[]={3.3,4.,-.5};
char greeting[4] = "hi!";
// length of hello[] will be 14 characters, including null
char hello[]="Hello, world.";
```

As noted above, the Arduino programming language does not have a separate "string" data type. Strings defined as arrays of characters must contain one more element than the number of characters, to allow for a required null character (`'\0'`) at the end.

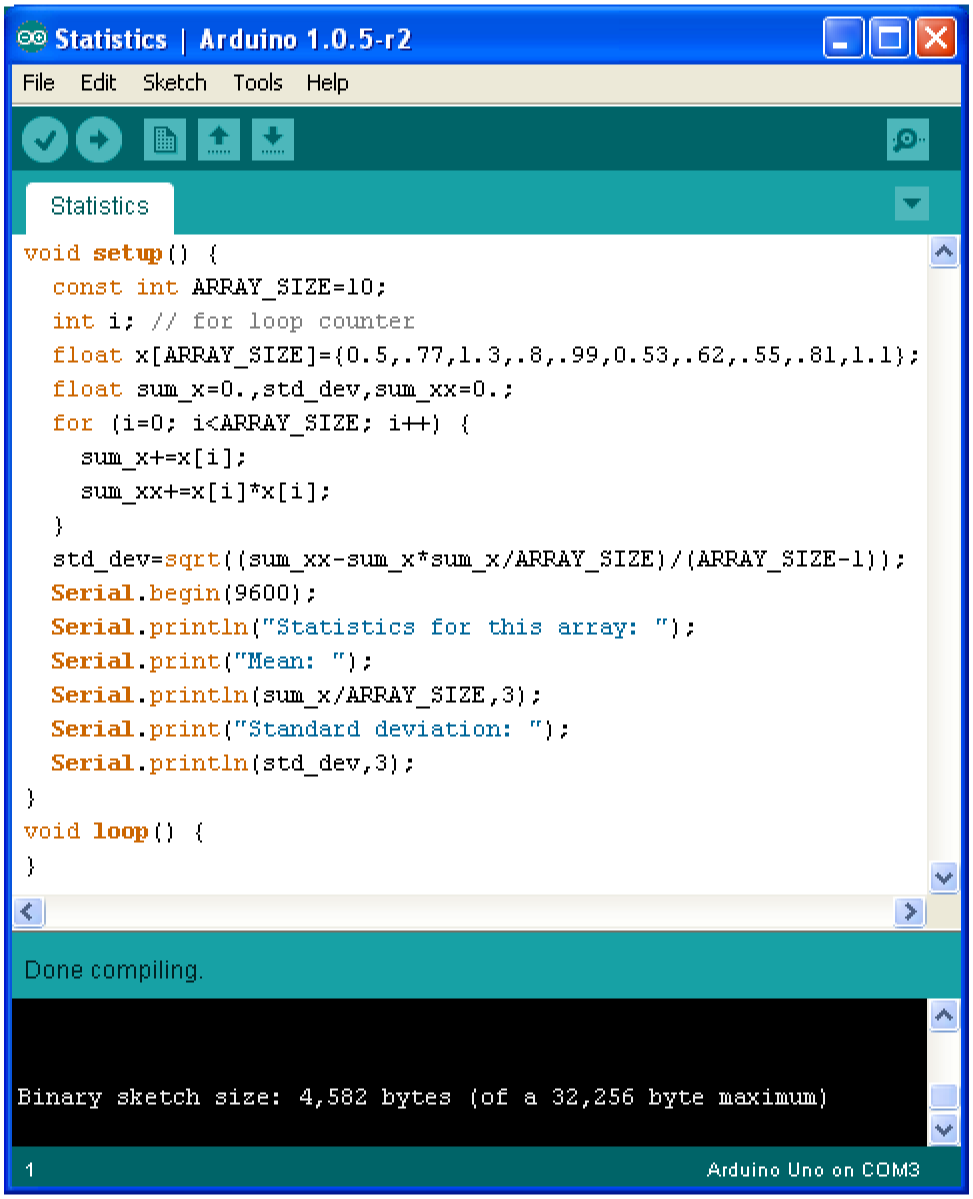
Elements of an array are accessed through an index value, which can be an integer constant, an integer variable, or a calculation that returns an integer result. The Arduino language does not check to see if an index value refers to non-existent elements beyond the array declaration boundaries. Trying to access values outside the defined boundaries will cause problems that can be very difficult to debug. Often, `for…` loops are used to access array elements. For an array with 10 elements, the appropriate index values are 0 through 9, *not* 1 through 10. Reading from element 10 in a 10-element array will not produce an error, but it will produce junk – whatever happens to be in that memory location at the time. Trying to assign a value to element 10 in a 10-element array could destroy values in memory that you really didn't want to lose!

Sketch 9 shows a typical calculation performed on an array of numerical values: find the mean and standard deviation of values in the array. This code will be of interest in a datalogger sketch.

2.2.8 User-defined functions

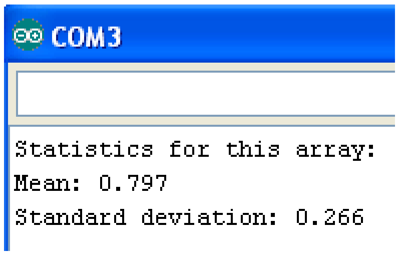
User-defined functions serve two important purposes. They make it easier to organize code and they facilitate calculations that must be done more than once, but with different input values. The two important points to know about functions is that: (1) variables defined within a function (local variables) are isolated from variables in your main code or in other functions; (2) functions can have multiple inputs, but they can return only one value.

Sketch 10 shows a computationally trivially simple example: Write a function which accepts the radius of a circle as input and returns the circumference as output. If the function returns a value, the data type must be included in the function definition. If it doesn't return a value, its function type should be `void`. For this example, the return value is a floating point number. The data type(s) of the input parameter(s) must be specified, as shown.



```
void setup() {
  const int ARRAY_SIZE=10;
  int i; // for loop counter
  float x[ARRAY_SIZE]={0.5,.77,1.3,.8,.99,0.53,.62,.55,.81,1.1};
  float sum_x=0.,std_dev,sum_xx=0.;
  for (i=0; i<ARRAY_SIZE; i++) {
    sum_x+=x[i];
    sum_xx+=x[i]*x[i];
  }
  std_dev=sqrt((sum_xx-sum_x*sum_x/ARRAY_SIZE)/(ARRAY_SIZE-1));
  Serial.begin(9600);
  Serial.println("Statistics for this array: ");
  Serial.print("Mean: ");
  Serial.println(sum_x/ARRAY_SIZE,3);
  Serial.print("Standard deviation: ");
  Serial.println(std_dev,3);
}
void loop() {
}
```

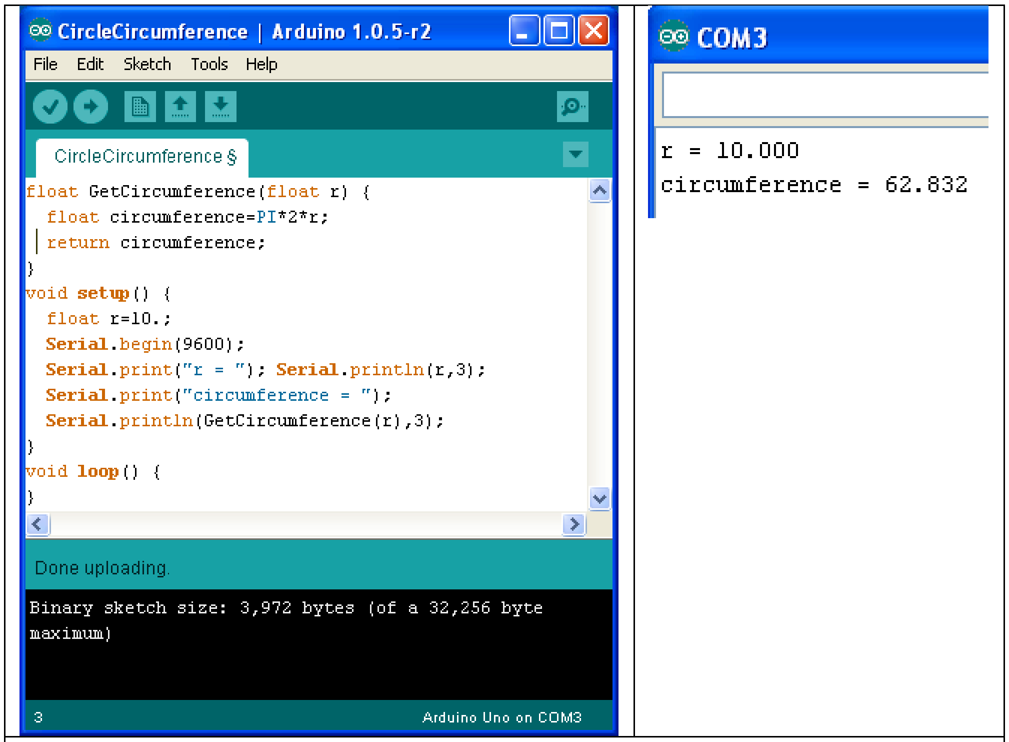
```
Done compiling.

Binary sketch size: 4,582 bytes (of a 32,256 byte maximum)
```



```
Statistics for this array:
Mean: 0.797
Standard deviation: 0.266
```

Sketch 9. Using arrays.

There are many situations in which it would be desirable to return more than one value from a function. For example, suppose you wish the function in Sketch 10, perhaps renamed to `CircleStuff`, to return both the circumference and area of a circle. Arduino functions (like C functions, on which the Arduino programming language is based) cannot return multiple values directly. One way around this problem is to store multiple values in an array and to define that array as a global variable. This is done simply by declaring the array before any other code in the sketch. A globally declared variable should not be re-declared within a function. With no direct return value, the function type should be `void`. When you write the code, it is up to you to keep track of which values are held in which array element. A possibly significant restriction is that all the return values must be of the same data type because all the array elements must be of the same data type.


CircleCircumference | Arduino 1.0.5-r2
File Edit Sketch Tools Help
CircleCircumference §
Done uploading.
3
Arduino Uno on COM3
COM3

```
float GetCircumference(float r) {
  float circumference=PI*2*r;
  return circumference;
}
void setup() {
  float r=10.;
  Serial.begin(9600);
  Serial.print("r = "); Serial.println(r,3);
  Serial.print("circumference = ");
  Serial.println(GetCircumference(r),3);
}
void loop() {
}
```

```
Binary sketch size: 3,972 bytes (of a 32,256 byte
maximum)
```

```
r = 10.000
circumference = 62.832
```

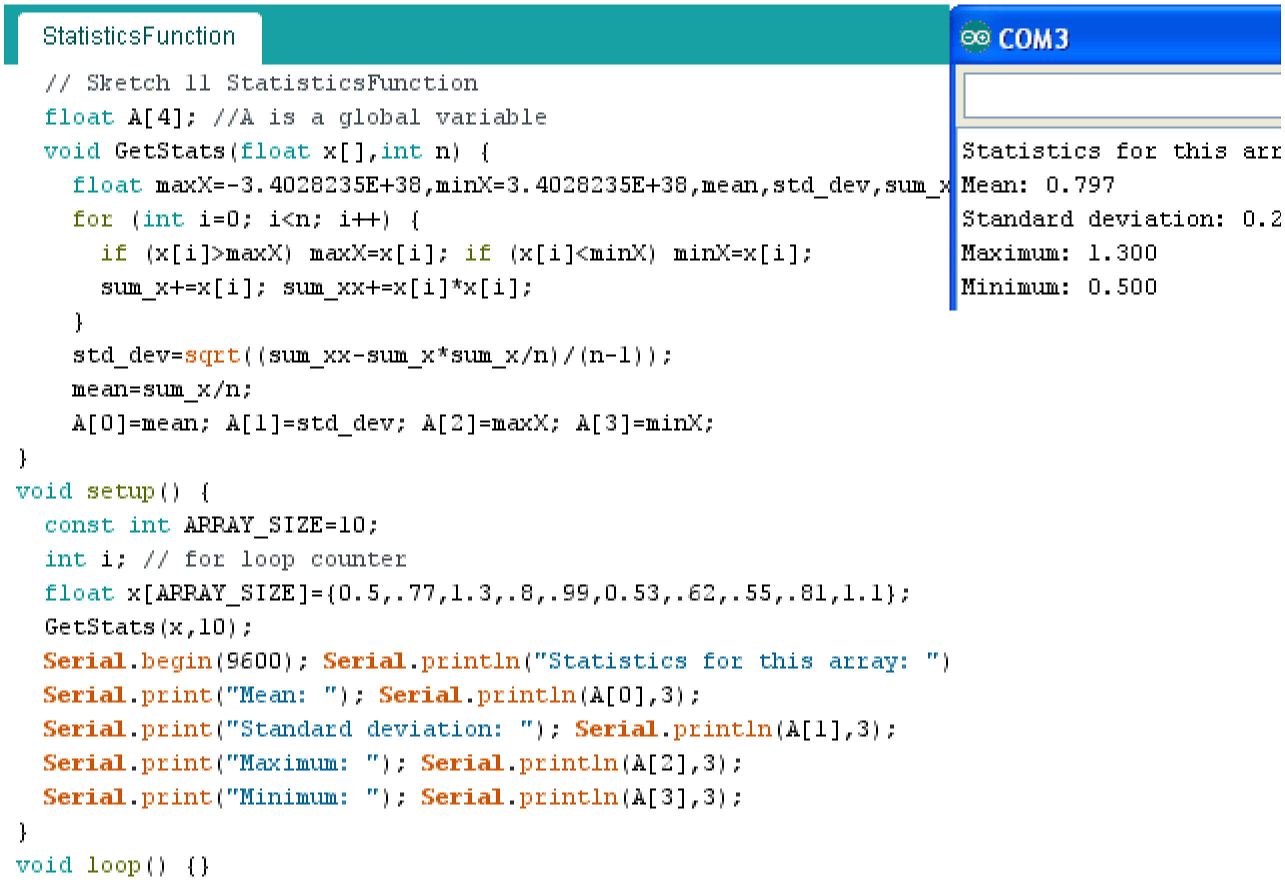
Sketch 10. Example of a user-defined function.

Sketch 11a is a rewrite of Sketch 9, with some statistics calculations done in a user-defined function. Because there are four calculated values (mean, standard deviation, maximum, and minimum – all floating point numbers), the results are returned in an array.

With this approach, it is easy to add more return values to a function. For example, in Sketch 11, you could define the global array A with 5 elements instead of 4 and calculate the median in the function.

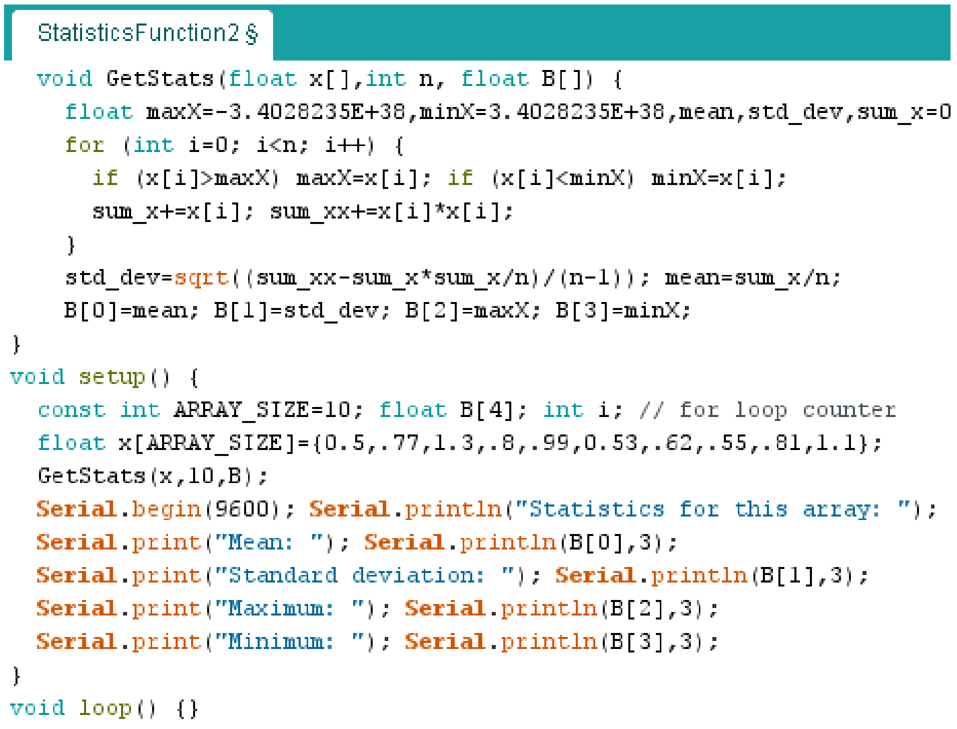
It is not necessary to use an array to access multiple values calculated within a function. You can define all the desired return values as global variables, which are available inside any function. There are two advantages to this approach: (1) each value has its own variable name rather than being just an indexed array element; (2) the values don't have to have the same data types. The only disadvantage is that you have to be careful not to redefine those variable names elsewhere in your sketch. This might be a significant problem in an environment with less restrictive maximum code size requirements, but it shouldn't be a problem for the code that the Arduino can handle.

There is a third option for "returning" multiple values from a function. This involves passing a "pointer" to a variable name and modifying the contents of the memory location to which that pointer points. In that case, the function doesn't actually "return" anything, so its type is `void`. Sketch 12 shows an example. Because using too many global variables can create problems with possible variable name conflicts, many programmers prefer to use pointers rather than global variables. Note that arrays are always passed by reference. Hence, you can pass them as input to a function and modify their elements in a function without having to "return" them, thereby avoiding the use of a global variable for the array. See Sketch 11b.



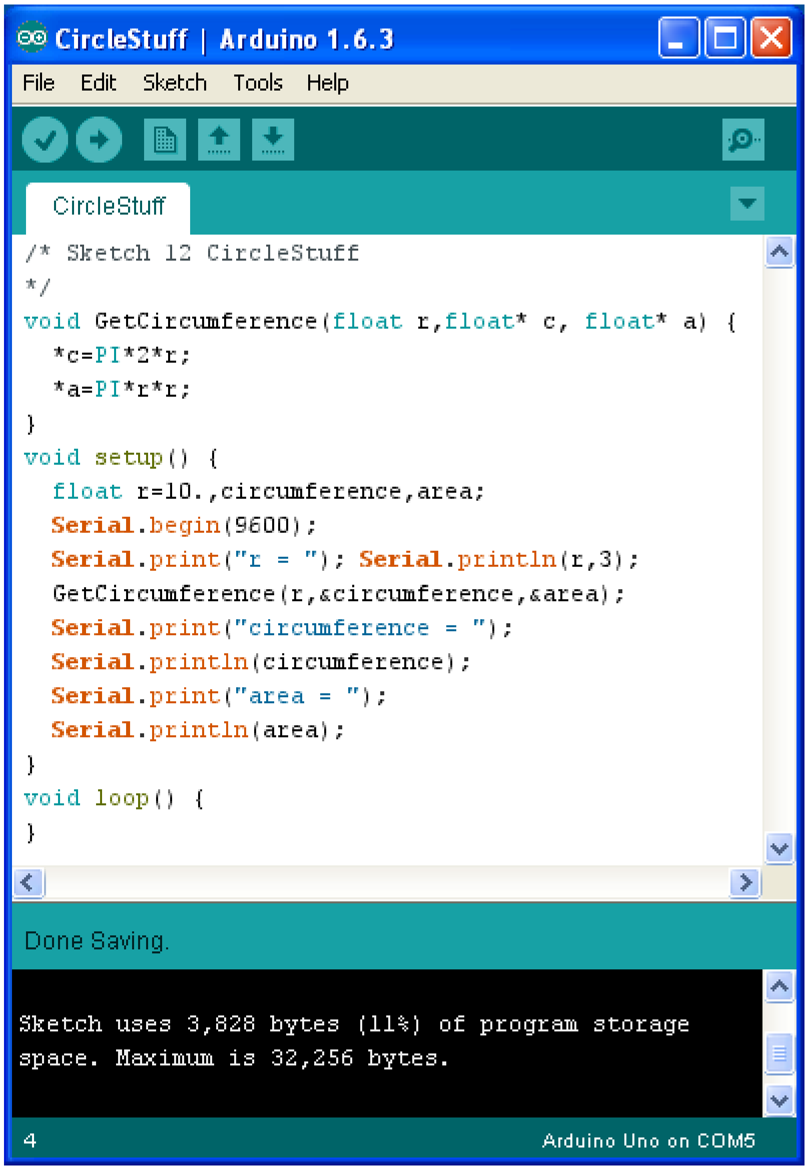
```
  // Sketch 11 StatisticsFunction
  float A[4]; //A is a global variable
  void GetStats(float x[],int n) {
    float maxX=-3.4028235E+38,minX=3.4028235E+38,mean,std_dev,sum_x
    for (int i=0; i<n; i++) {
      if (x[i]>maxX) maxX=x[i]; if (x[i]<minX) minX=x[i];
      sum_x+=x[i]; sum_xx+=x[i]*x[i];
    }
    std_dev=sqrt((sum_xx-sum_x*sum_x/n)/(n-1));
    mean=sum_x/n;
    A[0]=mean; A[1]=std_dev; A[2]=maxX; A[3]=minX;
}
void setup() {
  const int ARRAY_SIZE=10;
  int i; // for loop counter
  float x[ARRAY_SIZE]={0.5,.77,1.3,.8,.99,0.53,.62,.55,.81,1.1};
  GetStats(x,10);
  Serial.begin(9600); Serial.println("Statistics for this array: ")
  Serial.print("Mean: "); Serial.println(A[0],3);
  Serial.print("Standard deviation: "); Serial.println(A[1],3);
  Serial.print("Maximum: "); Serial.println(A[2],3);
  Serial.print("Minimum: "); Serial.println(A[3],3);
}
void loop() {}
```

```
Statistics for this arr
Mean: 0.797
Standard deviation: 0.2
Maximum: 1.300
Minimum: 0.500
```



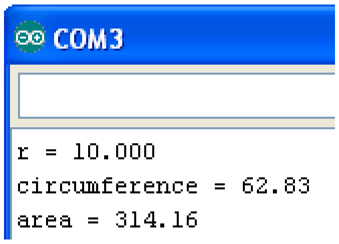
```
  void GetStats(float x[],int n, float B[]) {
    float maxX=-3.4028235E+38,minX=3.4028235E+38,mean,std_dev,sum_x=0
    for (int i=0; i<n; i++) {
      if (x[i]>maxX) maxX=x[i]; if (x[i]<minX) minX=x[i];
      sum_x+=x[i]; sum_xx+=x[i]*x[i];
    }
    std_dev=sqrt((sum_xx-sum_x*sum_x/n)/(n-1)); mean=sum_x/n;
    B[0]=mean; B[1]=std_dev; B[2]=maxX; B[3]=minX;
}
void setup() {
  const int ARRAY_SIZE=10; float B[4]; int i; // for loop counter
  float x[ARRAY_SIZE]={0.5,.77,1.3,.8,.99,0.53,.62,.55,.81,1.1};
  GetStats(x,10,B);
  Serial.begin(9600); Serial.println("Statistics for this array: ");
  Serial.print("Mean: "); Serial.println(B[0],3);
  Serial.print("Standard deviation: "); Serial.println(B[1],3);
  Serial.print("Maximum: "); Serial.println(B[2],3);
  Serial.print("Minimum: "); Serial.println(B[3],3);
}
void loop() {}
```

Sketches 11a and 11b. Returning multiple values from a user-defined function using arrays.



```
/* Sketch 12 CircleStuff
*/
void GetCircumference(float r,float* c, float* a) {
  *c=PI*2*r;
  *a=PI*r*r;
}
void setup() {
  float r=10.,circumference,area;
  Serial.begin(9600);
  Serial.print("r = "); Serial.println(r,3);
  GetCircumference(r,&circumference,&area);
  Serial.print("circumference = ");
  Serial.println(circumference);
  Serial.print("area = ");
  Serial.println(area);
}
void loop() {
}
```

```
Sketch uses 3,828 bytes (11%) of program storage
space. Maximum is 32,256 bytes.
```



```
r = 10.000
circumference = 62.83
area = 314.16
```

Sketch 12. Using pointers to “return” multiple values from a user-defined function.

It is never *required* to use pointers in Arduino programming, but it is sometimes very helpful for getting around the single-return restriction of user-defined functions, as Sketch 12 shows. In some situations it can be a more efficient way to change values. Arduino programming syntax for referencing and dereferencing pointers, using `&` and `*` in front of variable names, respectively, is identical to the C language. Oddly, the Arduino programming reference (http://arduino.cc/en/Reference/Pointer) is singularly unhelpful on this topic – it basically advises you to look elsewhere. Fortunately, there are many online discussions of how to use pointers in C programming.

2.3 Digital and Analog I/O

As noted previously, a fundamental purpose of programming the Arduino is to control the hardware interface. This is done through pins attached (both literally and in the software sense) to various devices. In this document, we will just scratch the surface of this topic. Just as there is no substitute for writing your own code to learn how to program, there is no substitute for wiring up devices to the Arduino board. Get some prototyping breadboards, a few electronics components, some hookup wire, and get started. All these parts are widely available from places like RadioShack, AllElectronics (www.allelectronics.com), and other electronics suppliers. Or, buy the Arduino starter pack mentioned in Chapter 1.Time spent with these simple devices absolutely will *not* be wasted in preparation for putting together a datalogger.

2.3.1 Digital pins

Digital pins can be set to either a "high" or "low" state. See here for a tutorial on digital pins: http://arduino.cc/en/Tutorial/DigitalPins.

There are three functions available for controlling and accessing digital pins:

```
pinMode(pin_number, INPUT or OUTPUT)
digitalWrite(pin_number, HIGH or LOW)
digitalRead(pin_number)
```

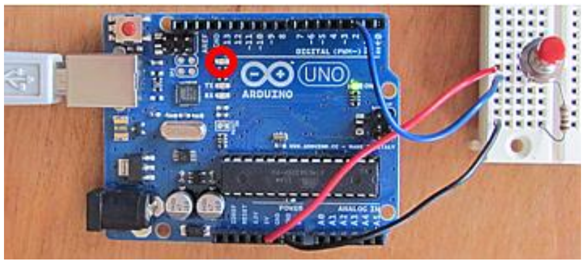
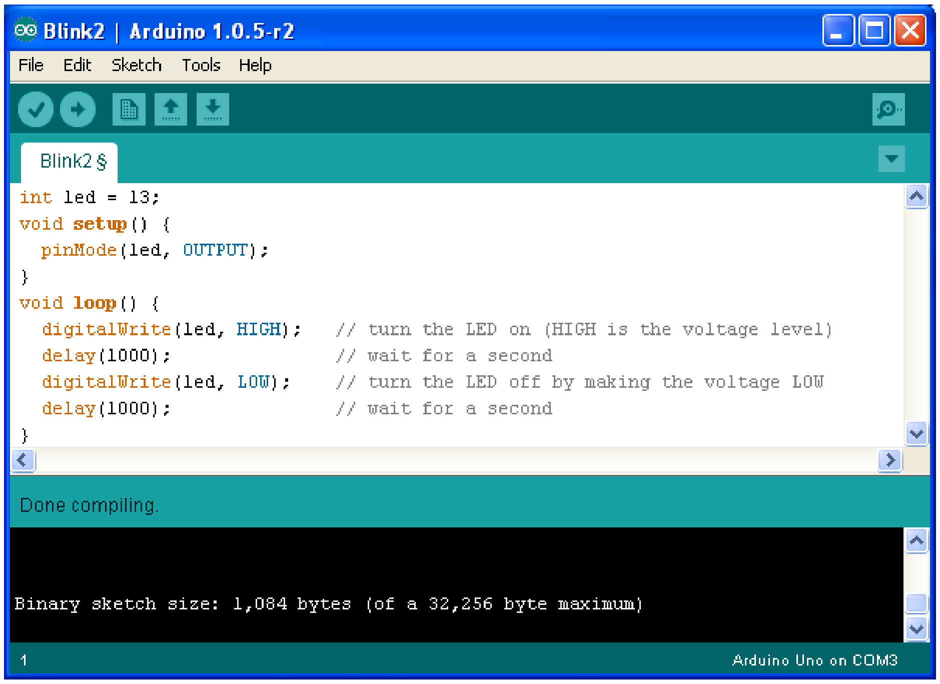


Figure 2. Use a digital pin to read the status of a pushbutton.

Pins configured as OUTPUT (the default state) can provide a constant current of up to 20 mA or an intermittend current of up to 40 mA to a connected device – enough to power an LED, for example, but probably not a relay or motor. Pins configured as INPUT are used to detect changes in state of a connected devices such as a pushbutton.

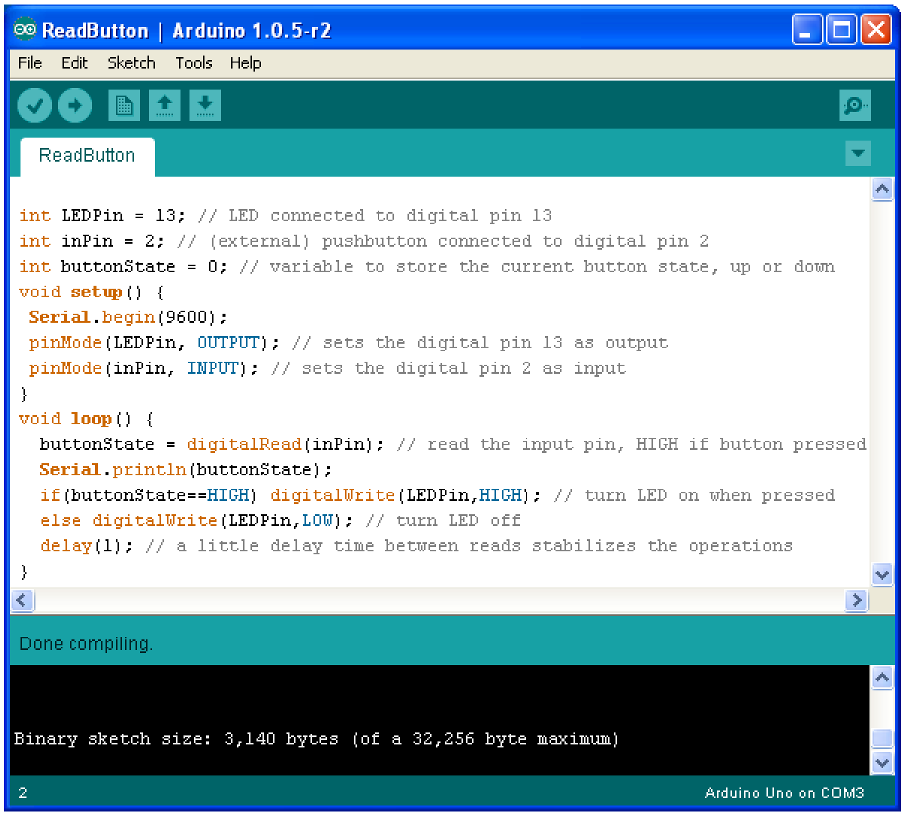
Sketches 13 and 14 show two examples, slight modifications of examples from the standard Arduino installation library. The Uno R3 has an LED already connected to pin 13. Sketch 13 turns an LED (the small yellow LED marked with the red circle in the image) on and off, with one second in each state. This example uses only the hardware already on the board.



```
int led = 13;
void setup() {
  pinMode(led, OUTPUT);
}
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

```
Done compiling.

Binary sketch size: 1,084 bytes (of a 32,256 byte maximum)
```

Sketch 13. Blink an LED.



```
int LEDPin = 13; // LED connected to digital pin 13
int inPin = 2; // (external) pushbutton connected to digital pin 2
int buttonState = 0; // variable to store the current button state, up or down
void setup() {
 Serial.begin(9600);
 pinMode(LEDPin, OUTPUT); // sets the digital pin 13 as output
 pinMode(inPin, INPUT); // sets the digital pin 2 as input
}
void loop() {
  buttonState = digitalRead(inPin); // read the input pin, HIGH if button pressed
  Serial.println(buttonState);
  if(buttonState==HIGH) digitalWrite(LEDPin,HIGH); // turn LED on when pressed
  else digitalWrite(LEDPin,LOW); // turn LED off
  delay(1); // a little delay time between reads stabilizes the operations
}
```

```
Done compiling.

Binary sketch size: 3,140 bytes (of a 32,256 byte maximum)
```

Sketch 14. Use LED to display state of a pushbutton.

Sketch 14 uses the hardware shown in Figure 2 – a pushbutton connected to a digital pin, through a 10KΩ resistor to ground. The purpose of the resistor is to limit the amount of current that will flow from the pin to ground when the button is pushed; this current should not exceed 40 mA. From Ohm's law, the current through the 10KΩ resistor is I = V/R = 5/10000 = 0.5 mA. Do NOT connect a pushbutton directly from the pin to ground! The digital pins are at the top right. The blue wire is connected to pin 2. The red and black wires are connected to the +5V pin and ground. In the code below, the `digitalRead()` function reads the state of pin 2, HIGH (when the button is pressed) or LOW. For software help, see these tutorials: http://arduino.cc/en/Reference/digitalRead and http://arduino.cc/en/Reference/digitalWrite. If everything is working OK, the small yellow LED marked with the red circle will light up only while the button is pushed and held down.

2.3.2 Analog pins

Reading signals on analog input pins is no more difficult than reading digital pins, except the return is an integer value whose interpretation is based on the input voltage relative to the reference voltage applied to the pin. (See http://arduino.cc/en/Reference/AnalogReference?from=Reference.AREF) and http://arduino.cc/en/Tutorial/AnalogInputPins.) The functions available for controlling and accessing analog pins include:

`analogReference(`*`type`*`)`

Determines the reference voltage for analog input – the value used as the top of the input range. The allowed *`type`* values include:

`DEFAULT`, 5V for 5V boards or 3.3V for 3.3V boards
`INTERNAL`, a built-in reference of 1.1 V
`EXTERNAL`, a voltage applied to the `AREF` pin, between 0 and 3.3V or 5V, depending on the board.

Additional TYPE values are not available on the Arduino R3 or equivalent boards. The reference voltage for all analog pins is set with `analogReference()` – you cannot set different values simultaneously for different pins, although you can write code to change the reference while a sketch is running.

`analogRead(pin_number)`

Reads a value from the specified pin. For the Arduino's 10-bit analog-to-digital conversion, this is an integer value A between 0 and 1023 and the conversion to voltage is `( A/1023.)*REF;`, where `REF` is the default reference voltage (5V or 3.3V, depending on the board) or the reference voltage set with `analogReference()`. The decimal point after 1023 is *required* because A is an integer value and `(A/1023)*REF;` will return a value of 0, or 1 if `A` is 1023.

`analogWrite(pin_number,dutyCycle)`

The `analogWrite()` function is not, as might be a reasonable assumption, the "inverse" of `analogRead()`. Its purpose is to modify the duty cycle of a pulse-width modulated (PWM) square wave (see Figure 3) sent to a pin configured to accept this input. The frequency of the square wave is about 500 Hz. The allowed values for `dutyCycle` are between 0 and 255. For a 50% duty cycle, set `dutyCycle=128`. For a 5% cycle, set `dutyCycle=13`.

On the Arduino Uno R3 and similar boards, this function works on pins 3, 5, 6, 9, 10, and 11. On the Uno R3 these pins are marked with a dash before the pin number. On the Arduino Pro, the pins are noted with PWM before the pin number.

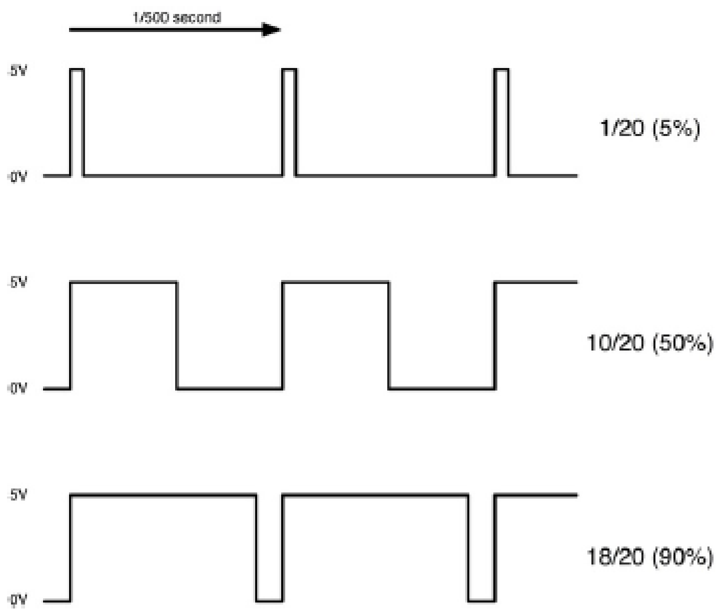
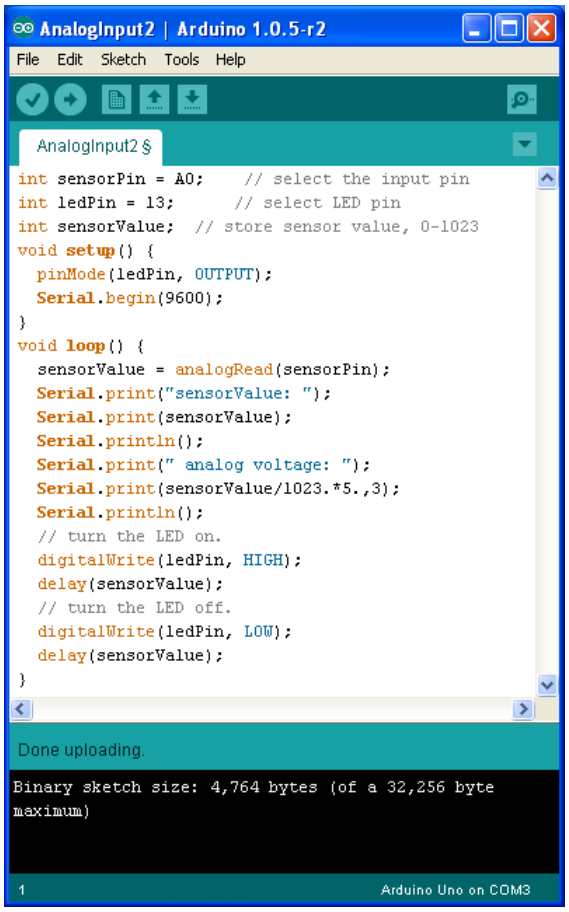


Figure 3. Pulse width modulation. (See https://learn.adafruit.com/downloads/pdf/adafruit-arduino-lesson-3-rgb-leds.pdf)

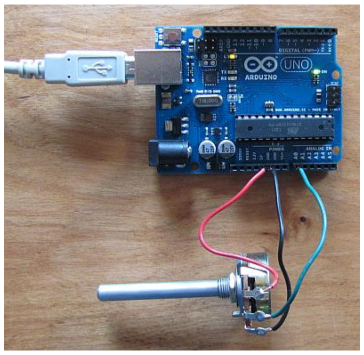
Sketch 15, shows how to use `analogRead()`. The two outer leads of a 10kΩ potentiometer are connected to the 5V pin and ground. The center lead is connected to an analog pin. The code reads the integer value on the pin and blinks the LED on digital pin 13 at a rate that depends on the position of the potentiometer shaft (with `delay(sensorValue)` having a value between 0 and 1023 milliseconds). Sketch 15 also displays the voltage value – `sensorValue/1023.*5.` – at the analog pin, as shown in some output cut-and-pasted from the serial port window. As noted above, the decimal point after the 1023 is required. The decimal point after the 5 is optional in this case.

For the Uno R3 board, which operates at 5V, it is *essential* not to apply a voltage outside the range 0–5 V to any pin, digital or analog. Exceeding this range will destroy that pin's function and may destroy the entire Uno board. In Sketch 15, the voltage applied to the input pin is, by definition, some fraction of the 5 V applied to the digital pin, so an appropriate value is guaranteed. This is a potential problem with an external sensor providing input to an analog pin. Some Arduino boards are powered at 3.3 V, in which case that is the maximum allowed input voltage on a pin. The ”input voltage” on a pin is different from the supply voltage which powers the board; that should be in the 7-12V range. (The board contains an onboard voltage regulator to provide +5 V to the board components when an external power supply is used.)



```
int sensorPin = A0;    // select the input pin
int ledPin = 13;       // select LED pin
int sensorValue;  // store sensor value, 0-1023
void setup() {
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}
void loop() {
  sensorValue = analogRead(sensorPin);
  Serial.print("sensorValue: ");
  Serial.print(sensorValue);
  Serial.println();
  Serial.print(" analog voltage: ");
  Serial.print(sensorValue/1023.*5.,3);
  Serial.println();
  // turn the LED on.
  digitalWrite(ledPin, HIGH);
  delay(sensorValue);
  // turn the LED off.
  digitalWrite(ledPin, LOW);
  delay(sensorValue);
}
```

```
Binary sketch size: 4,764 bytes (of a 32,256 byte
maximum)
```

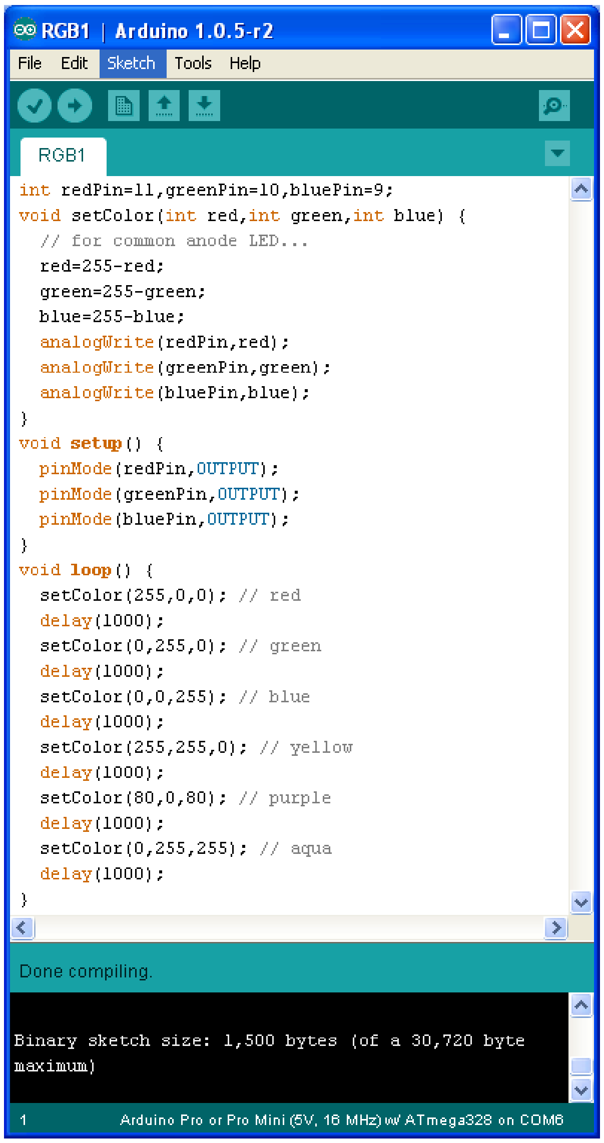


```
sensorValue: 0
 analog voltage: 0.000
sensorValue: 0
 analog voltage: 0.000
sensorValue: 27
 analog voltage: 0.132
sensorValue: 175
 analog voltage: 0.855
sensorValue: 451
 analog voltage: 2.204
sensorValue: 805
 analog voltage: 3.935
sensorValue: 1023
 analog voltage: 5.000
sensorValue: 1023
 analog voltage: 5.000
```

Sketch 15. Reading voltages on an analog pin.

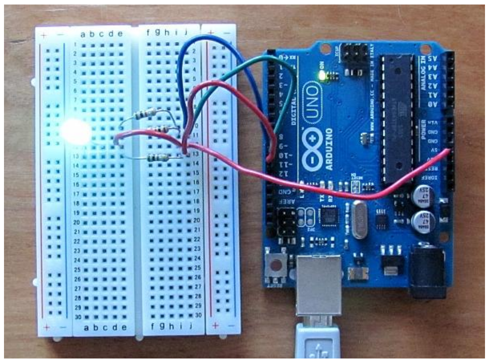
Sketch 16 shows how to use `analogWrite()`. This sketch has absolutely nothing to do with data logging, but it does show an interesting use of the PWM pins – varying the effective voltage to the three LEDs in a three-component (RGB) LED. It uses three 1K current-limiting resistors and a common-anode LED (Adafruit PID 159) in a diffusing 5-mm . By varying the width of the voltage pulse supplied to each LED, the visible output can be dimmed or turned off to select and blend colors quite smoothly – the pulse frequency is high enough that the LED output doesn't appear to flicker. The image shows the LED in its “aqua” mode. In a simpler application, `analogWrite()` could be used to control the brightness of any LED – otherwise, this would have to be done by changing the resistor value.

Note: *never* connect an LED to a pin without including a current-limiting resistor. For a typical LED, this value shouldn’t be less than about 270Ω (a standard resistor value). For typical LEDs in 3- and 5-mm housings, the current through the LED and resistor should be about 20 mA and no more than 30 mA for full brightness. In many cases, a smaller current will still provide adequate light and may be desirable to minimize power consumption.



```
int redPin=11,greenPin=10,bluePin=9;
void setColor(int red,int green,int blue) {
  // for common anode LED...
  red=255-red;
  green=255-green;
  blue=255-blue;
  analogWrite(redPin,red);
  analogWrite(greenPin,green);
  analogWrite(bluePin,blue);
}
void setup() {
  pinMode(redPin,OUTPUT);
  pinMode(greenPin,OUTPUT);
  pinMode(bluePin,OUTPUT);
}
void loop() {
  setColor(255,0,0); // red
  delay(1000);
  setColor(0,255,0); // green
  delay(1000);
  setColor(0,0,255); // blue
  delay(1000);
  setColor(255,255,0); // yellow
  delay(1000);
  setColor(80,0,80); // purple
  delay(1000);
  setColor(0,255,255); // aqua
  delay(1000);
}
```

```
Done compiling.

Binary sketch size: 1,500 bytes (of a 30,720 byte
maximum)
```



Sketch 16. Using `analogWrite()` on PWM pins to change the apparent color of an RGB LED.

3. AN ARDUINO-BASED DATALOGGER

With the programming background presented in Chapter 2, it is now possible to develop datalogger applications. For our purposes, a "datalogger" is defined as a device which will operate independently to store analog data in digital form. The datalogger application may also include programming to do some internal data processing of input, such as averaging multiple values collected over a specified time interval. To operate independently, the device should include onboard data storage. Microcontrollers make it possible to design such standalone devices. In general, such a project requires four components:

1. microcontroller
2. analog-to-digital (ADC) converter
3. clock
4. data storage device (an SD card)

3.1 Using the Adafruit Datalogger Shield to Explore a Datalogger Application

The hardware required for a basic standalone datalogger includes an Arduino board and a datalogger shield from Adafruit (https://www.adafruit.com/product/1141). (Boards which "piggyback" on the Arduino board are called "shields.") The shield includes a real time clock and an SD card interface. A datalogger shield packaged with components for sensing light (CdS photoresistor) and temperature (Analog Devices TMP36, see https://learn.adafruit.com/tmp36-temperature-sensor) is used here, see http://www.adafruit.com/products/249. This is an instructive choice of hardware. The photoresistor requires that current flow through it, supplied by the Arduino board. The temperature sensor requires power supplied by the Arduino.

The datalogger shield without any input sensors is preassembled. Connecting the light and temperature sensors requires a little soldering, as shown in Figure 4, but there are instructions in a complete user's guide (https://learn.adafruit.com/downloads/pdf/adafruit-data-logger-shield.pdf).

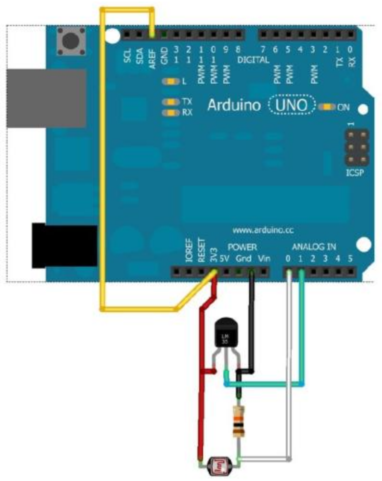


Figure 4(a). Connections for temperature and light sensors.

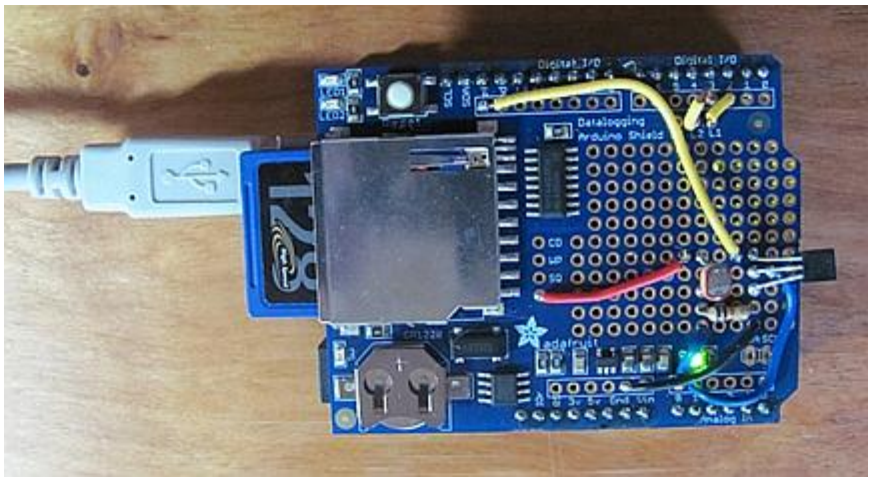


Figure 4(b). Components installed in the work area of an Adafruit datalogger shield, mounted on Uno R3 board.

As shown in Figure 4, the outputs from these two sensors are connected to analog pins A0 and A1(the two blue wires in the lower right hand corner in Figure 4(b) – the shorter wire passes just over the left-hand side of the green LED). They make use of the built-in 10-bit analog-to-digital conversion capabilities of the Arduino board to read the output. These sensors are very useful for learning how to program a datalogger, but the relatively low 10-bit ADC conversion resolution on the Arduino analog input over a 0-5V (default) input range – 5/1023≈ 5 mV – will not be suitable for sensors with lower voltage outputs; For example, the output from a silicon photodiode-based pyranometer, (http://www.instesre.org/construction/pyranometer/pyranometer.htm), available from the Institute for Earth Science Research and Education has an output of about 250mV in full summer sunlight. Under full summer sun, the solar radiation reaching Earth's surface is about 1000 W/m^2, so for this instrument, the logger will provide a resolution of only about 20 W/m^2 – a resolution too poor for monitoring solar radiation. The datalogger shield software uses a 3.3V power source from the Arduino Uno R3 board. This improves the resolution a little, but not significantly.

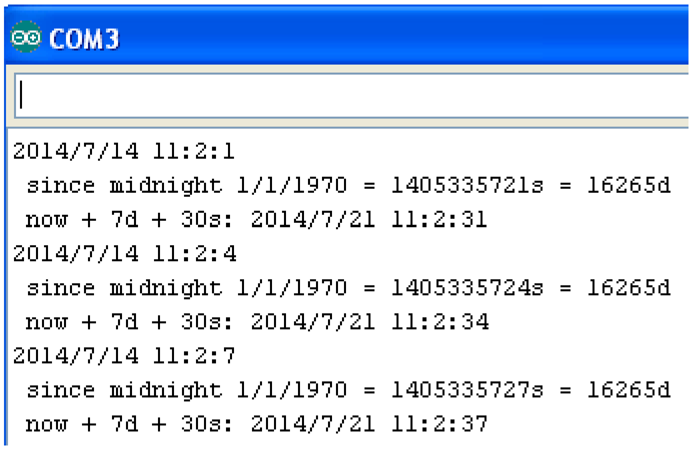
Let's test two subsystems on the datalogger shield – the clock and the SD card interface.

3.1.1 Real time clock (RTC)

The datalogger shield includes a real-time clock – an essential component of a system for logging data. The coin cell battery will last for several *years*, so it is simply left in place once installed. A library is required to use the clock. Download the `RTClib.cpp` and `RTClib.h` files at https://github.com/adafruit/RTClib. Create a folder within the `\libraries` folder, `\libraries\RTClib`, and copy both the `.cpp` and `.h` files there. You should install libraries only when the Arduino IDE is not running because libraries installed while the IDE is running will not be recognized until the IDE is closed and restarted.

Communications with the clock are handled through the widely used "Inter-Integrated Circuit" (I2C) protocol, which allows various devices to communicate with each other (http://tronixstuff.com/2010/10/20/tutorial-arduino-and-the-i2c-bus/). That communication is managed by the `Wire.h` library, which is part of the standard Arduino installation.

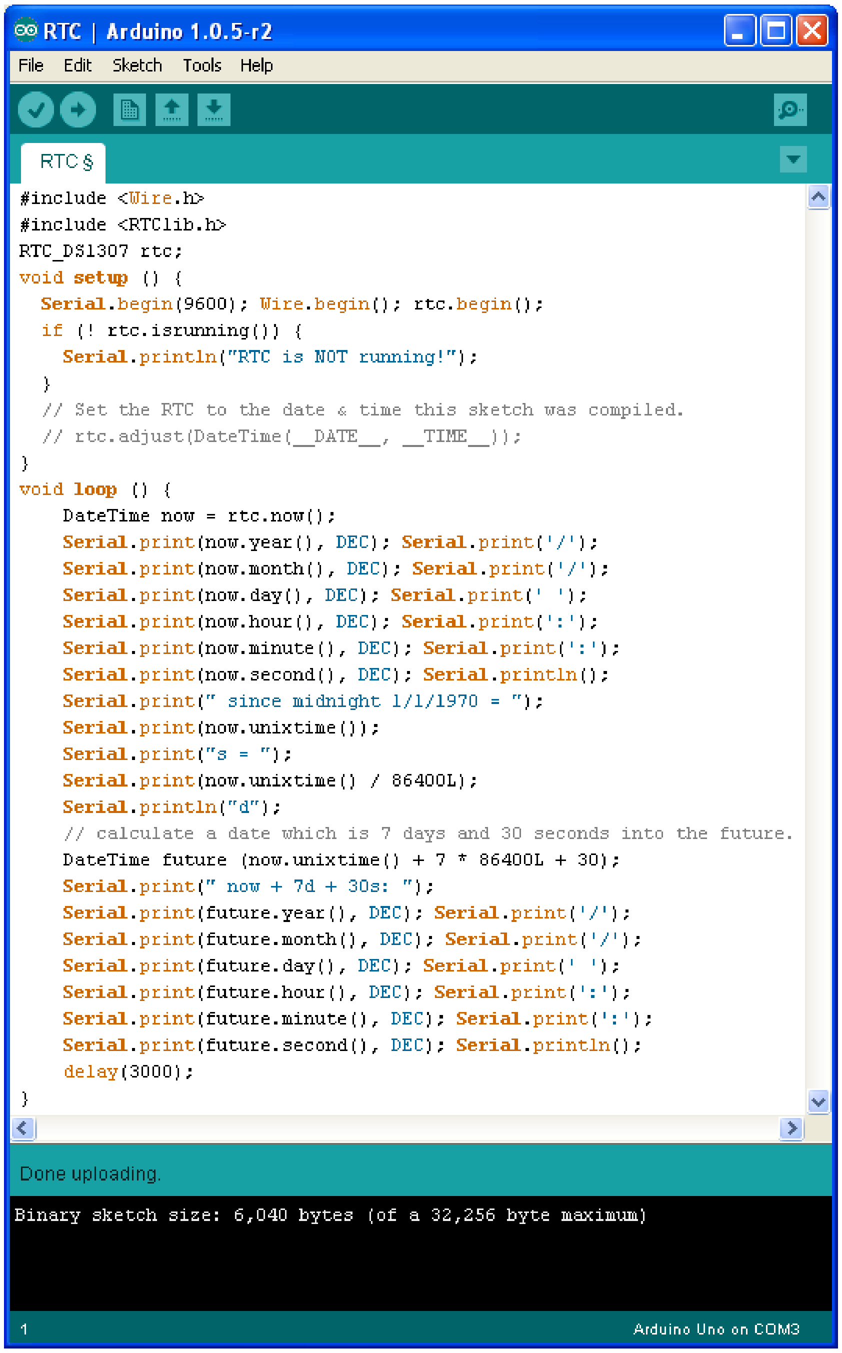
Sketch 17 shows how to use the real time clock and its library. The `rtc.now()` method provides access to year, month, day, hours, minutes, and seconds. Some output from that sketch is shown in Figure 5. The first time you run Sketch 17 with a new clock, the output will not agree with your computer clock. Remove the line comment (`//`) from line 10 and reload the script. This will set the clock according to your computer clock. After this has been done once, you shouldn't have to do it again (for years!) as long as you don't remove the battery. In the sample code from Adafruit, the `Serial.begin(57600)` statement must be changed to `Serial.begin(9600)`. The sampling interval is every 3 seconds (3000 milliseconds) – see `delay(3000)` at the end of the loop.



```
2014/7/14 11:2:1
 since midnight 1/1/1970 = 1405335721s = 16265d
 now + 7d + 30s: 2014/7/21 11:2:31
2014/7/14 11:2:4
 since midnight 1/1/1970 = 1405335724s = 16265d
 now + 7d + 30s: 2014/7/21 11:2:34
2014/7/14 11:2:7
 since midnight 1/1/1970 = 1405335727s = 16265d
 now + 7d + 30s: 2014/7/21 11:2:37
```

Figure 5. Output from Sketch 17.

Time calculations are based on the number of seconds since the beginning of the day on 1/1/1970 (not counting any leap seconds that may have been added). This may seem like an odd way to keep track of time but, because integer calculations can be done exactly and very efficiently, this is typical of how programming languages handle time calculations. One result is that integers can become too large for the standard 2-byte `int` data type (±32787 or 65,535 for an unsigned integer). Note statements such as `Serial.print(now.unixtime()/86400L);`, in which the "L" forces the `86400` (the number of seconds in one day) to be treated as a "long integer" stored in 4 bytes. It could also be specified as an unsigned long integer (UL or ul) (See http://arduino.cc/en/Reference/IntegerConstants for more information about typing integers.) For specifying long integers, lowercase L will work, but it is a good idea to use uppercase L rather than lowercase l, which can too easily be mistaken for the digit 1. An unsigned long integer can store a value up to 4,294,967,295.



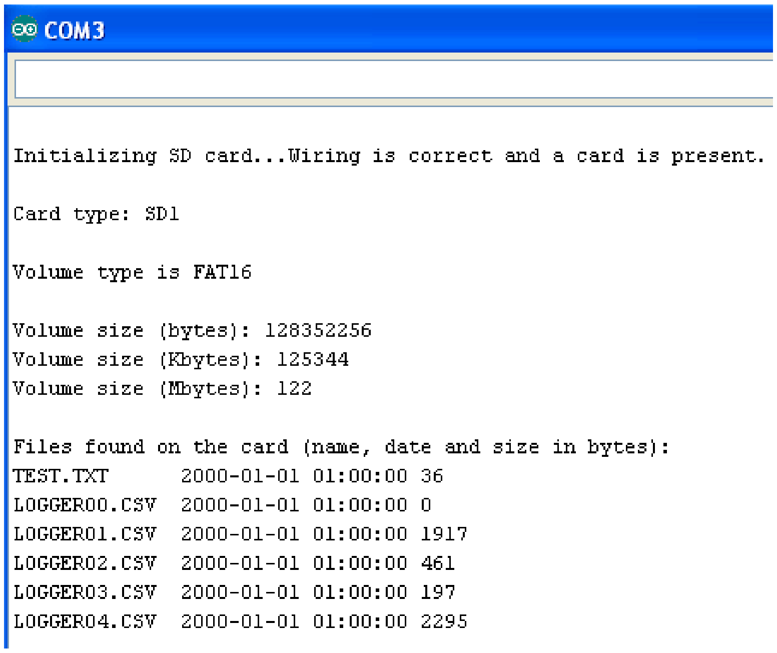
```
#include <Wire.h>
#include <RTClib.h>
RTC_DS1307 rtc;
void setup () {
  Serial.begin(9600); Wire.begin(); rtc.begin();
  if (! rtc.isrunning()) {
    Serial.println("RTC is NOT running!");
  }
  // Set the RTC to the date & time this sketch was compiled.
  // rtc.adjust(DateTime(__DATE__, __TIME__));
}
void loop () {
    DateTime now = rtc.now();
    Serial.print(now.year(), DEC); Serial.print('/');
    Serial.print(now.month(), DEC); Serial.print('/');
    Serial.print(now.day(), DEC); Serial.print(' ');
    Serial.print(now.hour(), DEC); Serial.print(':');
    Serial.print(now.minute(), DEC); Serial.print(':');
    Serial.print(now.second(), DEC); Serial.println();
    Serial.print(" since midnight 1/1/1970 = ");
    Serial.print(now.unixtime());
    Serial.print("s = ");
    Serial.print(now.unixtime() / 86400L);
    Serial.println("d");
    // calculate a date which is 7 days and 30 seconds into the future.
    DateTime future (now.unixtime() + 7 * 86400L + 30);
    Serial.print(" now + 7d + 30s: ");
    Serial.print(future.year(), DEC); Serial.print('/');
    Serial.print(future.month(), DEC); Serial.print('/');
    Serial.print(future.day(), DEC); Serial.print(' ');
    Serial.print(future.hour(), DEC); Serial.print(':');
    Serial.print(future.minute(), DEC); Serial.print(':');
    Serial.print(future.second(), DEC); Serial.println();
    delay(3000);
}
```

```
Done uploading.
Binary sketch size: 6,040 bytes (of a 32,256 byte maximum)
1                                        Arduino Uno on COM3
```

Sketch 17. Testing the real time clock.

3.1.2 SD card interface

Like the clock, the SD card requires a library. Download the `SD.h` and `SD.cpp` files at https://github.com/adafruit/SD and copy them into a `\libraries\SD` folder. Sketch 18, which tests communication with an SD card, is taken from the SD library. Figure 6 shows output from the `CardInfo.ino` sketch available along with the SD library. The code line `const int chipSelect = 4;` in the downloaded version must be changed to const `int chipSelect = 10;` to work with this shield. In this case, the code reports that 6 files (including an empty file) have already been saved to the card. The file creation data and time are clearly junk. This is because a different library, `SdFat` (http://code.google.com/p/sdfatlib/downloads/list), is required to provide proper time stamps for files written to SD cards. For these purposes, this capability does not seem to be worth the extra code; typically, logged data should include date and time fields from the real time clock as part of the output saved in the file.



```
Initializing SD card...Wiring is correct and a card is present.

Card type: SD1

Volume type is FAT16

Volume size (bytes): 128352256
Volume size (Kbytes): 125344
Volume size (Mbytes): 122

Files found on the card (name, date and size in bytes):
TEST.TXT      2000-01-01 01:00:00 36
LOGGER00.CSV  2000-01-01 01:00:00 0
LOGGER01.CSV  2000-01-01 01:00:00 1917
LOGGER02.CSV  2000-01-01 01:00:00 461
LOGGER03.CSV  2000-01-01 01:00:00 197
LOGGER04.CSV  2000-01-01 01:00:00 2295
```

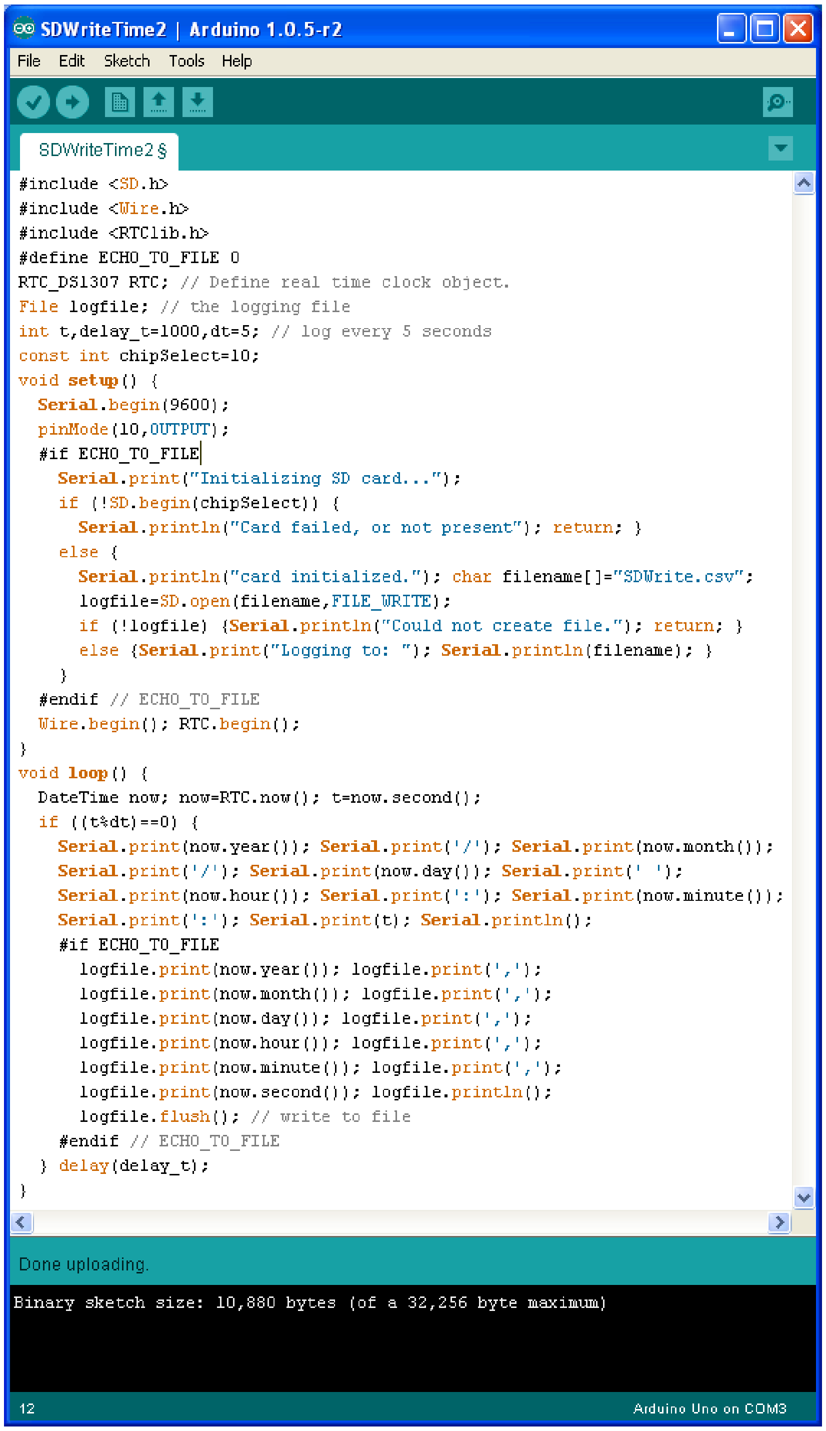
Figure 6. Output from `CardInfo.ino` (modified).

A reasonable next step is to write date and time data from the real time clock code to an SD card file. First, recall the real time clock code shown previously in Sketch 17. That code read time and date values from the `DateTime` object, did a couple of calculations, printed results to the serial port, and then waited for 3 seconds (`delay(3000)`) before doing it again.

Sketch 17 does *not* really produce results every three seconds. The delay between getting one set of time and date values and the next is three seconds, because of `delay(3000)`, *plus* the time required to do everything else inside the loop. In some applications, the additional delay of at least several tens of milliseconds might not even be noticeable, but as a result the output from Sketch 17 will periodically "skip" a second.

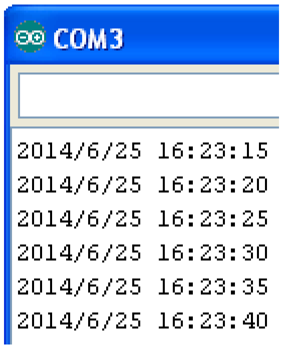
The way to fix this problem (assuming that you think it *is* a problem) is to get data from the `DateTime` object more often and process data only when the seconds returned from `DateTime` is an integer multiple of the desired interval. If `t` is seconds and `dt` is the desired sampling interval, then data should be processed only when `t%dt` is 0. Getting data from the `DateTime` object more often may put more demands on the processor, but you will have more control over the results. If `dt` is at least 2 s, then your code should include a `delay(1000)` to ensure that the same second isn't processed twice. Then your logged data will always be at the desired interval relative to clock time. If this doesn't matter, don't bother!

In Sketch 18, the 6th line, `File logfile;`, defines a logical name, a "handle," which is then associated with a physical file name. You can use whatever name you like for the handle. The logical name is not the same as the physical file name.



```
#include <SD.h>
#include <Wire.h>
#include <RTClib.h>
#define ECHO_TO_FILE 0
RTC_DS1307 RTC; // Define real time clock object.
File logfile; // the logging file
int t,delay_t=1000,dt=5; // log every 5 seconds
const int chipSelect=10;
void setup() {
  Serial.begin(9600);
  pinMode(10,OUTPUT);
  #if ECHO_TO_FILE
    Serial.print("Initializing SD card...");
    if (!SD.begin(chipSelect)) {
      Serial.println("Card failed, or not present"); return; }
    else {
      Serial.println("card initialized."); char filename[]="SDWrite.csv";
      logfile=SD.open(filename,FILE_WRITE);
      if (!logfile) {Serial.println("Could not create file."); return; }
      else {Serial.print("Logging to: "); Serial.println(filename); }
    }
  #endif // ECHO_TO_FILE
  Wire.begin(); RTC.begin();
}
void loop() {
  DateTime now; now=RTC.now(); t=now.second();
  if ((t%dt)==0) {
    Serial.print(now.year()); Serial.print('/'); Serial.print(now.month());
    Serial.print('/'); Serial.print(now.day()); Serial.print(' ');
    Serial.print(now.hour()); Serial.print(':'); Serial.print(now.minute());
    Serial.print(':'); Serial.print(t); Serial.println();
    #if ECHO_TO_FILE
      logfile.print(now.year()); logfile.print(',');
      logfile.print(now.month()); logfile.print(',');
      logfile.print(now.day()); logfile.print(',');
      logfile.print(now.hour()); logfile.print(',');
      logfile.print(now.minute()); logfile.print(',');
      logfile.print(now.second()); logfile.println();
      logfile.flush(); // write to file
    #endif // ECHO_TO_FILE
  } delay(delay_t);
}
```

```
Done uploading.
Binary sketch size: 10,880 bytes (of a 32,256 byte maximum)
```



```
2014/6/25 16:23:15
2014/6/25 16:23:20
2014/6/25 16:23:25
2014/6/25 16:23:30
2014/6/25 16:23:35
2014/6/25 16:23:40
```

Sketch 18. Write date and time data to SD card file at specified time interval.

Sketch 18 includes a pre-compile directive to turn off writing to a file, for the purpose of testing the rest of the code. All the code for writing data to a file is included between within the `#if… #endif` directives. The syntax for `logfile.print()` and `logfile.println()` is the same as for `Serial.print()` and `Serial.println()`. However, `logfile.print()` and `logfile.println()` don't actually write data to the file. Those statements temporarily store data in a buffer. The `logfile.flush()` statement actually writes data to the file – you can think of it as "flushing" the buffer by transferring data to the SD card file. In principle, this means that you can store the results of multiple calls to `print()` or `println()` before actually writing those results to a file. This may save processing time and memory space, but it doesn't seem worth the effort for the kinds of data logging applications that will be dealt with in this document.

For Arduino programming, file names are restricted to no more than 8 characters, a period, and a maximum of three characters for a file name extension. In Sketch 18, the data are written with a `.csv` file extension so they are easy to import directly into a spreadsheet. Windows computers don't distinguish between uppercase and lowercase characters in file names (saved files will be spelled in all uppercase letters), but Linux systems do, so be sure to make your spelling of file names consistent as required.

For writing date and time data to a file, my personal preference is to separate year, month, day, hour, minute, and second by commas, rather than writing them in a conventional MM/DD/YYYY HH:MM:SS format (or DD/MM/YYYY… in European notation). This makes it easy to convert a day and time into a fractional day. This calculation could be done in your sketch:

```
fractionalDay = day + hour/24. + minute/1440. + second/86400.;
```

where the decimal points force the calculation to use a real-number arithmetic – otherwise calculations with integer division (for example, `hour/24`) would always be 0.

Sketch 18 represents the last intermediate step toward writing code to log temperature and light sensor data. Yes, a pre-written logging script for those data is included with the Adafruit datalogger shield/sensor package, but it has proven much more instructive to approach this application one step at a time by learning how to write code that does just what is needed to test each component of the system, and nothing more.

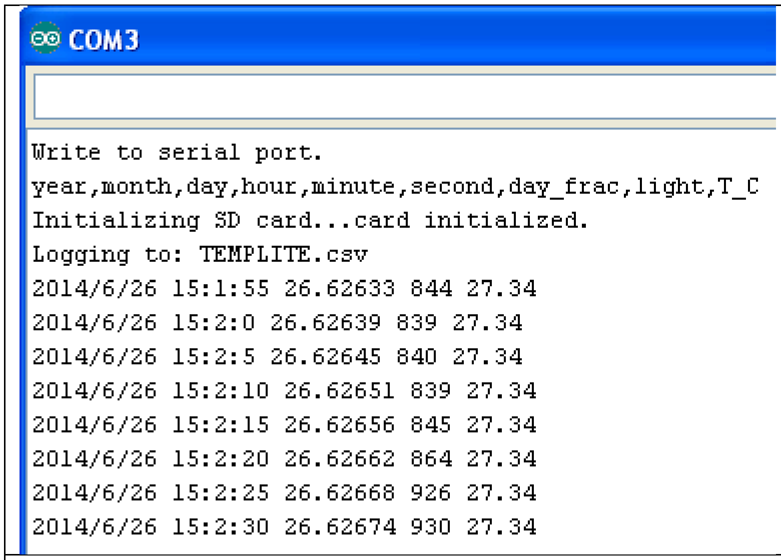
3.1.3 A simple data logging program

This section uses the Arduino data logging shield with temperature and light sensors, connected as shown above in Figure 4. Sketch 19 is a simple data logging program for these sensors. (These sketches, and some that follow, are too long to capture as screen shots of the IDE window.) It records temperature (ºC) and light sensor data (integer values between 0 and 1023) every 5 seconds. Using pre-compile directives, the output can be switched between the serial port (set `ECHO_TO_SERIAL` to 1 for testing, as shown in the sketch) and to an SD card file (`ECHO_TO_FILE` to 1), or both outputs can be turned. You can name these directives whatever you like – these just seemed like reasonable names. Some serial port output is shown in Figure 7(a) and some data from an output file opened in Excel is shown in Figure 7(b).

Sketch 19.

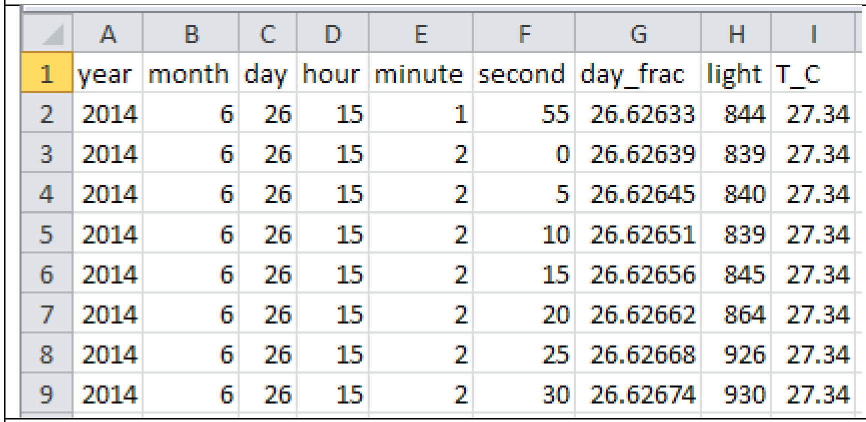
```
// MyLightTempLogger.ino
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <RTClib.h>
#define ECHO_TO_FILE 0
#define ECHO_TO_SERIAL 1
File logfile; // the logging file
int Second,delay_t=1000,dt=5;
int tempReading, photocellReading,Year,Month,Day,Hour,Minute;
float temperatureC;
const float aref_voltage=3.3;
const int photocellPin=0,tempPin=1,chipSelect=10;
RTC_DS1307 RTC; // Define real time clock object.
void setup() {
  Serial.begin(9600); pinMode(10,OUTPUT);
  #if ECHO_TO_SERIAL
    Serial.println("Write to serial port.");
    Serial.println("year,month,day,hour,minute,second,day_frac,light,T_C");
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    Serial.print("Initializing SD card...");
    if (!SD.begin(chipSelect)) {
      Serial.println("Card failed, or not present"); return; }
    else {
      Serial.println("card initialized."); }
    char filename[]="TEMPLITE.CSV";
    logfile=SD.open(filename,FILE_WRITE);
    if (!logfile) {Serial.println("Could not create file."); return; }
    else {Serial.print("Logging to: "); Serial.println(filename); }
    logfile.println("year,month,day,hour,minute,second,day_frac,light,T_C");
  #endif // ECHO_TO_FILE
  Wire.begin(); RTC.begin();
  analogReference(EXTERNAL);
}
void loop() {
  DateTime now=RTC.now();
  Year=now.year(); Month=now.month(); Day=now.day();
  Hour=now.hour(); Minute=now.minute(); Second=now.second();
  if ((Second%dt)==0) {
    photocellReading=analogRead(photocellPin); delay(10);
    tempReading=analogRead(tempPin); delay(10);
    temperatureC = (tempReading*aref_voltage/1024 - 0.5)*100;
    #if ECHO_TO_SERIAL
      Serial.print(Year); Serial.print('/'); Serial.print(Month);
      Serial.print('/'); Serial.print(Day); Serial.print(' ');
      Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
      Serial.print(':'); Serial.print(Second);
      Serial.print(' ');
      Serial.print(Day+Hour/24.+Minute/1440.+Second/86400.,5);
      Serial.print(' '); Serial.print(photocellReading);
      Serial.print(' '); Serial.print(temperatureC,2);
      Serial.println();
    #endif // ECHO_TO_SERIAL
    #if ECHO_TO_FILE
      logfile.print(Year); logfile.print(',');
      logfile.print(Month); logfile.print(',');
      logfile.print(Day); logfile.print(',');
      logfile.print(Hour); logfile.print(',');
      logfile.print(Minute); logfile.print(',');
      logfile.print(Second); logfile.print(',');
      logfile.print(Day+Hour/24.+Minute/1440.+Second/86400.,5);
      logfile.print(','); logfile.print(photocellReading);
      logfile.print(','); logfile.print(temperatureC,2);
      logfile.println();
```

```
        logfile.flush(); // write to file
      #endif // ECHO_TO_FILE
    } delay(delay_t);
  }
```



```
Write to serial port.
year,month,day,hour,minute,second,day_frac,light,T_C
Initializing SD card...card initialized.
Logging to: TEMPLITE.csv
2014/6/26 15:1:55 26.62633 844 27.34
2014/6/26 15:2:0 26.62639 839 27.34
2014/6/26 15:2:5 26.62645 840 27.34
2014/6/26 15:2:10 26.62651 839 27.34
2014/6/26 15:2:15 26.62656 845 27.34
2014/6/26 15:2:20 26.62662 864 27.34
2014/6/26 15:2:25 26.62668 926 27.34
2014/6/26 15:2:30 26.62674 930 27.34
```

Figure 7(a). Serial port output for `MyLightTemperatureLogger.ino`, with `ECHO_TO_FILE` and `ECHO_TO_SERIAL` both set to 1.



	A	B	C	D	E	F	G	H	I
1	year	month	day	hour	minute	second	day_frac	light	T_C
2	2014	6	26	15	1	55	26.62633	844	27.34
3	2014	6	26	15	2	0	26.62639	839	27.34
4	2014	6	26	15	2	5	26.62645	840	27.34
5	2014	6	26	15	2	10	26.62651	839	27.34
6	2014	6	26	15	2	15	26.62656	845	27.34
7	2014	6	26	15	2	20	26.62662	864	27.34
8	2014	6	26	15	2	25	26.62668	926	27.34
9	2014	6	26	15	2	30	26.62674	930	27.34

Figure 7(b). SD card file (`TEMPLITE.CSV`) output for `MyLightTemperatureLogger.ino`.

3.1.4 Additional software considerations for a data logging application

For a general-purpose data logging application, it is useful to be able to modify its performance with "configuration parameters." The simplest implementation samples and stores data at a specified interval; Sketch 19 is an example of such an implementation. A more flexible implementation would allow changing the sampling interval without having to make changes to the code. It might also be desirable to sample data at some specified interval and then, at some longer interval, calculate and store statistics for the individual samples. For example, sample at 10-second intervals and then, at 5-minute intervals, calculate and store average, max, min, and standard deviation for those 30 samples.

There are two possibilities for providing additional flexibility. One is to create a configuration file "offline" and store it as a text file on the same SD card used to store data. Another way is to upload the sketch and enter the configuration data from the keyboard in the serial port monitor window. Both methods require some knowledge about how Arduino reads and processes numerical and text input.

Sketch 20 shows one way to read data from a file on the SD card. A lot of the code is required just to check the status of the hardware. The rest of the code deals with interpreting data stored in the file. When a data file is opened (the default for an open file is "read only"), the code establishes a "pointer" to the beginning of the file. The `read()` method reads the byte at that position and advances the pointer to the next byte.

This is not helpful behavior for interpreting several bytes in a file as numbers or "words" – strings of characters. It is important to understand that if you write 3.14159 in a text file, it looks to you like the *number* 3.14159. But from a computer's point of view, this is just a string of bytes that happens to represent digits and a decimal point.

The Arduino programming language includes two methods to deal with extracting numbers from strings of bytes: `parseInt()` and `parseFloat()`.These functions start at the current location of the file pointer. They then look for the first byte that could be a character associated with a number – the digits 0-9, a period (decimal point, for a floating point number), a + or – character, or e or E for a floating point number expressed in scientific notation. That byte is the first character in a string. Then they keep looking at byes, adding them to the string one at a time, until they find a byte that represents a character that can't be part of a number. Finally, they convert that string of characters into an integer or floating point number; how they do this last step might be interesting, but it is not relevant to this discussion.

One of the values in the data file accessed by Sketch 20 is the name of an output file to which data will be written. If this text string represents a file name, it should contain no more than eight characters for the name, a period, and no more than three characters for the file name extension. But, the text could be used for anything, with no length limitation. The Arduino programming language includes a `String` object that makes it easy to construct a "word" from a string of bytes in a file. In this case, the `read()` method is used to skip past the comma after the last numerical value. Then characters are extracted one at a time using the `read()` method and they are "concatenated" to a variable to store the word – in this case, the output file name. See the statement `outFile += c;` . The `while`... loop to read characters terminates when there are no more characters in the file. The `delay(10)` command may or may not be necessary. (But, see Sketch 20.) Finally, the `trim()` method is used to strip "white space" characters (spaces and tabs) that might exist at the end of the line of characters in the data file, or even at the beginning if there is a space between a comma and a string of characters. The output shows the extracted values. Note that the 3.3 is printed as 3.300 – three digits to the right of the decimal point – to demonstrate that the code really has interpreted the characters 3.3 as the floating point number 3.3.

The second approach to getting configuration parameters reads data directly from the serial port monitor rather than from a file. The code in Sketch 21 is shorter than that required to read from a data file on an SD card because there is less checking for hardware status.

Some Arduino programmers advise against using the `String` method unless it is absolutely necessary because it is a memory hog. It also allows dynamic allocation of array space in memory because the length of a character string doesn't have to be specified in advance. This can cause problems with code and memory space. Sketch 21 uses the same `parseInt()` and `parseFloat()` methods to extract numerical values as Sketch 20, but it doesn't use `String` methods to extract the text. The only disadvantage of this approach is that the number of characters in the file name string must be known ahead of time. If the character array dimension for the file name in your code is 13 – 8 characters for the file name plus a period, a three-character extension, and a null character, the file name string you enter must be in the format `XXXXXXXX.XXX`. Note that the output doesn't display the length of the text string. This is because `length()` is a method of the `String` object, which isn't available here.

Initially the serial port buffer is empty. The code

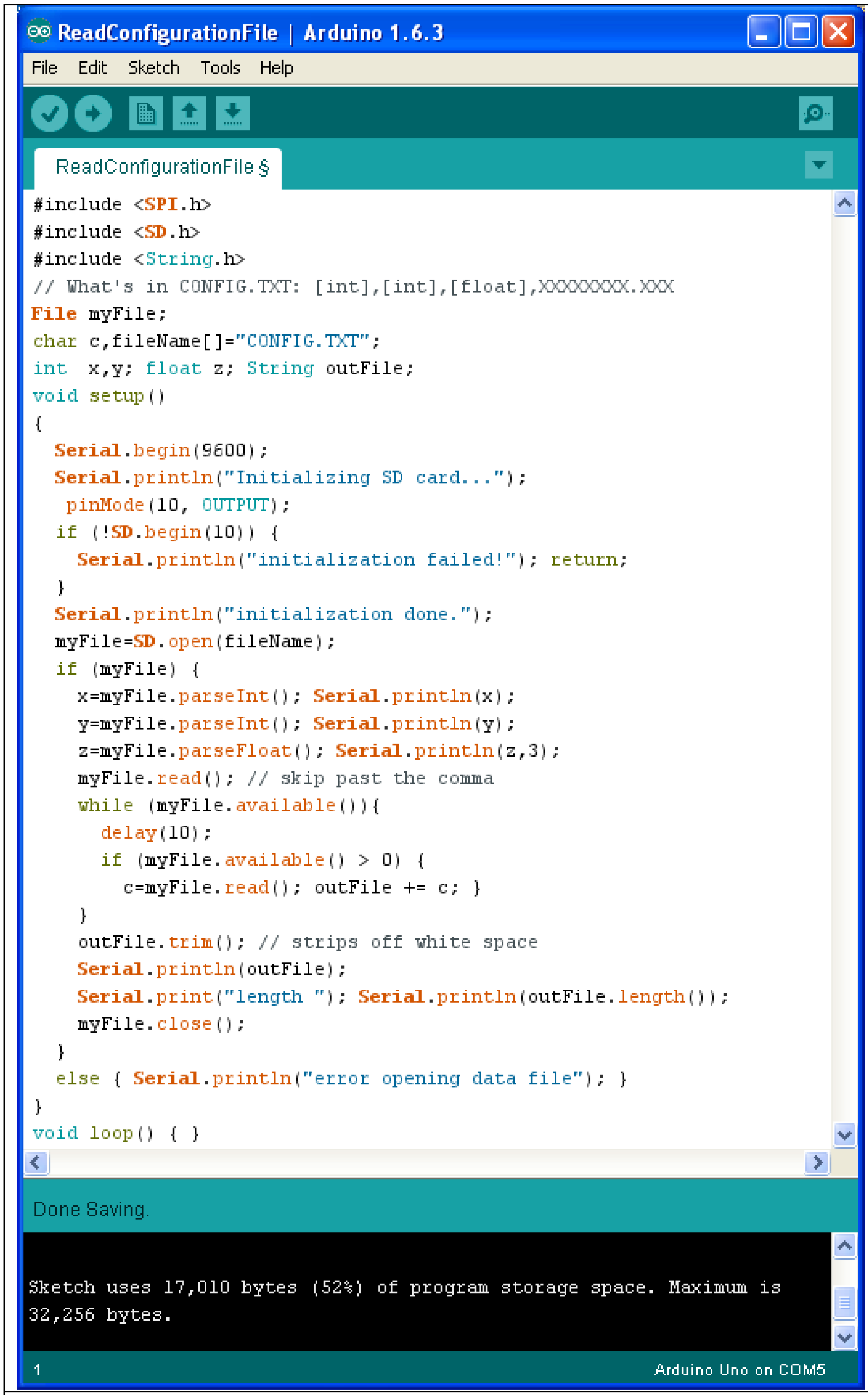
```
while (Serial.peek()<0) {
}
```

waits until the user types something into the box at the top of the serial port monitor window and presses the [Enter] key. If the user types `3,9600,3.3,LOGGER10.CSV[`Enter], the code extracts these four values – two integers, a floating point number, and a string. If you make a mistake, press the reset button on your Arduino board and start over again.

It is interesting to note that the `delay(10);` statements in line 15 and 17 turn out to be *required* for this sketch to work. This 10-ms delay is apparently necessary to "slow down" the code long enough to process data coming from the serial port buffer – an important and hard-earned lesson about a problem which took a *lot* of trial-and-error debugging to find and fix!

Both Sketch 20 and 21 assume a specific format for the characters saved in a text file or typed into the serial port buffer. In both cases, the format is integer, integer, floating point number, string – separated by commas. A space before the numerical values is OK, but not between the last comma and the start of the text in Sketch 21; without the `String` object, there is no `trim()` method to strip off white space. If the format is changed from what is shown, then the code must be changed accordingly. The code is not "smart enough" to figure out what you mean if you enter something unexpected!

Note that the code to read configuration values from the serial port (Sketch 21) is 5150 bytes, while the code in Sketch 20, with the `String` and `SD` libraries, takes almost 18,000 bytes of the 32,256 bytes available for code. On a "real" computer, this wouldn't be an issue, but sketch size can definitely be an issue for microcontroller programming.



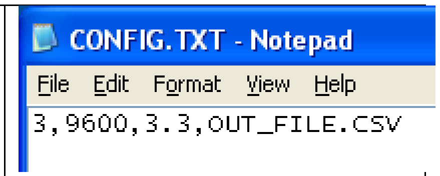
ReadConfigurationFile | Arduino 1.6.3

```
#include <SPI.h>
#include <SD.h>
#include <String.h>
// What's in CONFIG.TXT: [int],[int],[float],XXXXXXXX.XXX
File myFile;
char c,fileName[]="CONFIG.TXT";
int  x,y; float z; String outFile;
void setup()
{
  Serial.begin(9600);
  Serial.println("Initializing SD card...");
   pinMode(10, OUTPUT);
  if (!SD.begin(10)) {
    Serial.println("initialization failed!"); return;
  }
  Serial.println("initialization done.");
  myFile=SD.open(fileName);
  if (myFile) {
    x=myFile.parseInt(); Serial.println(x);
    y=myFile.parseInt(); Serial.println(y);
    z=myFile.parseFloat(); Serial.println(z,3);
    myFile.read(); // skip past the comma
    while (myFile.available()){
      delay(10);
      if (myFile.available() > 0) {
        c=myFile.read(); outFile += c; }
    }
    outFile.trim(); // strips off white space
    Serial.println(outFile);
    Serial.print("length "); Serial.println(outFile.length());
    myFile.close();
  }
  else { Serial.println("error opening data file"); }
}
void loop() { }
```

Done Saving.

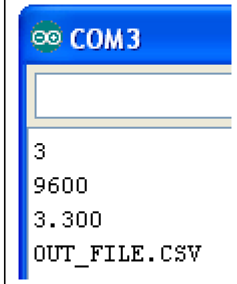
```
Sketch uses 17,010 bytes (52%) of program storage space. Maximum is
32,256 bytes.
```

Arduino Uno on COM5



CONFIG.TXT - Notepad

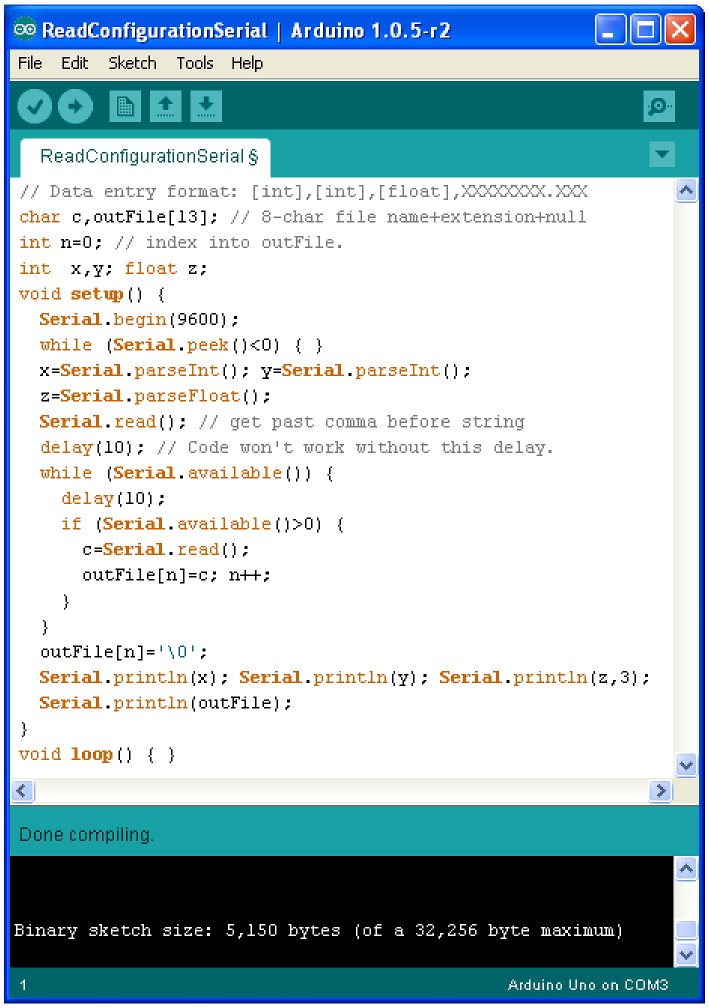
```
3,9600,3.3,OUT_FILE.CSV
```



COM3

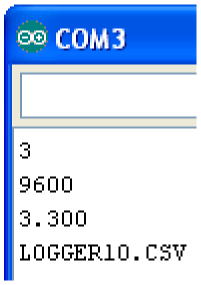
```
3
9600
3.300
OUT_FILE.CSV
```

Sketch 20. Read values from a text file stored on SD card.


ReadConfigurationSerial | Arduino 1.0.5-r2
File Edit Sketch Tools Help
ReadConfigurationSerial §

```
// Data entry format: [int],[int],[float],XXXXXXXX.XXX
char c,outFile[13]; // 8-char file name+extension+null
int n=0; // index into outFile.
int  x,y; float z;
void setup() {
  Serial.begin(9600);
  while (Serial.peek()<0) { }
  x=Serial.parseInt(); y=Serial.parseInt();
  z=Serial.parseFloat();
  Serial.read(); // get past comma before string
  delay(10); // Code won't work without this delay.
  while (Serial.available()) {
    delay(10);
    if (Serial.available()>0) {
      c=Serial.read();
      outFile[n]=c; n++;
    }
  }
  outFile[n]='\0';
  Serial.println(x); Serial.println(y); Serial.println(z,3);
  Serial.println(outFile);
}
void loop() { }
```

```
Done compiling.

Binary sketch size: 5,150 bytes (of a 32,256 byte maximum)
```



```
3
9600
3.300
LOGGER10.CSV
```

Sketch 21. Read values from serial port.

3.1.5 Putting it all together with complete data logging applications

In this section, two sketches will be presented which use the data logging shield with temperature and light sensors. Both sketches require that configuration information be supplied as input in the serial port window. (Based on the discussion in *3.1.3*, this seems like the easiest way to get configuration parameters.) Both sketches include pre-compile directives, `ECHO_TO_FILE` and `ECHO_TO_SERIAL`, that direct output to an SD card or to the serial port. In either case, there is some output to the serial port that displays the selected configuration parameters before reading values. The pre-compile directive values must be set manually (with 1 for "on" or 0 for "off") before the sketch is uploaded.

Sketch 22 logs values at prescribed intervals, expressed in seconds or minutes – typing

```
5,m,logfile1.csv
```

in the serial port window records values every 5 minutes and writes them in `logfile1.csv`, assuming that the `ECHO_TO_FILE` directive is set to 1; if it is set to 0 and `ECHO_TO_SERIAL` is set to 1, then the values are displayed in the serial port window and the file name is ignored.

The data collection always starts at a time that is an even multiple of the sampling interval. That is, for 2-second sampling, the samples are recorded at 0, 2, 4,… seconds, but not at 1, 3, 5,… seconds. For the example shown in Figure 8, the sampling started at 40 seconds, but it wouldn't have started at 39 seconds. For this code, the shortest sampling interval is 2 seconds and the longest is one hour. For one-hour sampling enter

```
0,m,…
```

because the minutes value returned by the clock is 0 at the start of each hour.

Some output from 2-second sampling written to the serial port is shown in Figure 9, with some data sampled in my office at 2-minute intervals, written to a `.csv` file, and opened in Excel. The temperature graph is a good illustration of the limitations of the Arduino's built-in 10-bit A/D resolution – the temperature resolution is about 0.3ºC. Although this may seem coarse, typical accuracy for the TMP36 sensor is only ±1ºC around room temperature and ±2ºC over its -40ºC to +125ºC range (http://www.analog.com/en/mems-sensors/digital-temperature-sensors/tmp36/products/product.html). The appropriateness of analog-to-digital conversions must, as always, be judged based on the inherent accuracy of the measurement providing the analog signal.

```
logging interval: 2s
Log to file: xxxxxxxx,xxx
Write to serial port.
year,month,day,hour,minute,second,day_frac,light,T_C
2014/7/23 15:18:40 23.63796 715 28.31
2014/7/23 15:18:42 23.63799 680 27.99
2014/7/23 15:18:44 23.63801 677 27.99
2014/7/23 15:18:46 23.63803 680 28.31
2014/7/23 15:18:48 23.63806 681 27.99
2014/7/23 15:18:50 23.63808 684 27.34
2014/7/23 15:18:52 23.63810 851 28.96
2014/7/23 15:18:54 23.63813 913 28.63
2014/7/23 15:18:56 23.63815 915 28.63
2014/7/23 15:18:58 23.63817 927 28.31
```

Figure 8. Sample output from Sketch 22.

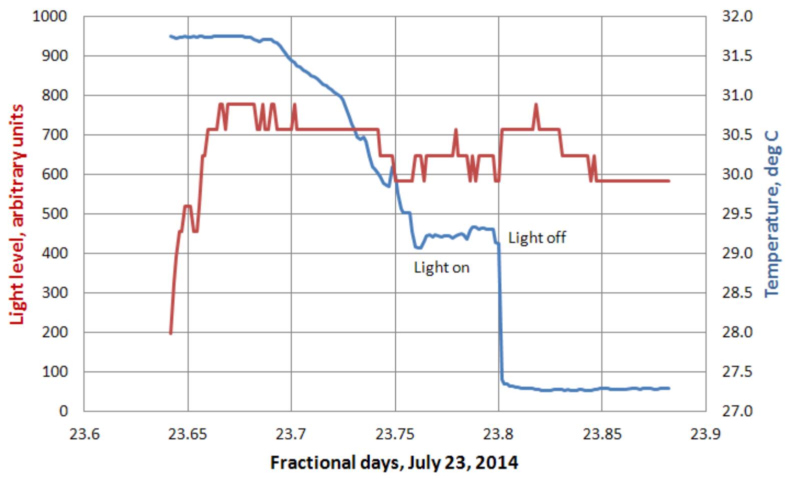


Figure 9. Recorded output from Sketch 22, 2-m intervals.

The light intensity is in arbitrary units. The logger was placed on my indoor office window ledge in the afternoon (it is rather warm there in the afternoon sunlight) and it is easy to see when, as the light faded in the early evening, I turned on the office light and then, later, turned it off when I left the office.

Sketch 22 is listed here in its entirety.

Sketch 22.

```
// MyLightTempLoggerB.ino
#include <SD.h>
#include <Wire.h>
```

```
#include <RTClib.h>
#define ECHO_TO_FILE 0
#define ECHO_TO_SERIAL 1
// input
// [int]dt,[char]m or c,[12-char file name]xxxxxxxx.xxx
// example: 5,s,logfile1.csv
// Minimum sampling interval, 2s
File logfile; // the logging file
int Second,delay_t=1000,dt;
char intervalType, outFile[13];
int tempReading, photocellReading,Year,Month,Day,Hour,Minute;
float temperatureC;
const float aref_voltage=3.3;
const int photocellPin=0,tempPin=1,chipSelect=10;
RTC_DS1307 RTC; // Define real time clock object.
void getConfiguration() {
  char c; int n=0;
  while (Serial.peek()<0) {}
  dt=Serial.parseInt(); delay(10); Serial.read();
  intervalType=Serial.read(); delay(10); Serial.read();
  while (Serial.available()) {
    delay(10); if (Serial.available()>0) {
      c=Serial.read(); outFile[n]=c; n++;
    }
  }
  outFile[n]='\0';
}
void dataOutput() {
  photocellReading=analogRead(photocellPin); delay(10);
  tempReading=analogRead(tempPin); delay(10);
  temperatureC = (tempReading*aref_voltage/1024 - 0.5)*100;
  #if ECHO_TO_SERIAL
    Serial.print(Year); Serial.print('/'); Serial.print(Month);
    Serial.print('/'); Serial.print(Day); Serial.print(' ');
    Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
    Serial.print(':'); Serial.print(Second);
    Serial.print(' ');
    Serial.print(Day+Hour/24.+Minute/1440.+Second/86400.,5);
    Serial.print(' '); Serial.print(photocellReading);
    Serial.print(' '); Serial.print(temperatureC,2);
    Serial.println();
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    logfile.print(Year); logfile.print(',');
    logfile.print(Month); logfile.print(',');
    logfile.print(Day); logfile.print(',');
    logfile.print(Hour); logfile.print(',');
    logfile.print(Minute); logfile.print(',');
    logfile.print(Second); logfile.print(',');
    logfile.print(Day+Hour/24.+Minute/1440.+Second/86400.,5);
    logfile.print(','); logfile.print(photocellReading);
    logfile.print(','); logfile.print(temperatureC,2);
    logfile.println();
    logfile.flush(); // write to file
  #endif // ECHO_TO_FILE
}
void setup() {
  Serial.begin(9600);
  getConfiguration();
  Serial.print("logging interval: ");Serial.print(dt);
  Serial.println(intervalType);
  if ((intervalType=='s') && (dt<2))
    Serial.print("Configuration error. Restart!");
  Serial.print("Log to file: "); Serial.println(outFile);
  pinMode(10,OUTPUT);
  #if ECHO_TO_SERIAL
    Serial.println("Write to serial port.");
    Serial.println("year,month,day,hour,minute,second,day_frac,light,T_C");
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    Serial.print("Initializing SD card...");
    if (!SD.begin(chipSelect)) {
      Serial.println("Card failed, or not present"); return; }
    else {
      Serial.println("card initialized."); }
    logfile=SD.open(outFile,FILE_WRITE);
    if (!logfile) {Serial.println("Could not create file."); return; }
    else {Serial.print("Logging to: "); Serial.println(outFile); }
    logfile.println("year,month,day,hour,minute,second,day_frac,light,T_C");
  #endif // ECHO_TO_FILE
  Wire.begin(); RTC.begin(); analogReference(EXTERNAL);
}
void loop() {
  DateTime now=RTC.now();
  Year=now.year(); Month=now.month(); Day=now.day();
  Hour=now.hour(); Minute=now.minute(); Second=now.second();
  if (intervalType=='s') {
    if ((Second%dt)==0) dataOutput();
  }
  if (intervalType=='m') {
    if ((Minute%dt==0) && (Second==0)) dataOutput();
  }
  delay(delay_t); // Don't process the same second twice!
}
```

Sketch 23 is similar to Sketch 22, but it accepts as input a sampling interval in seconds or minutes and then generates statistics over a longer interval, in minutes. For example, sampling every 10 seconds over a five-minute interval generates mean, maximum, minimum, and standard deviation of an input value based on 30 data samples (5 minutes = 300 seconds). With this code, it is not possible to generate statistics over an interval of less than 1 minute. It is up to the user to provide input that makes sense. Entering `10,s,5,logfile1.csv` is appropriate, but, for example, `30,m,2,logfile1.csv` makes absolutely no sense!

The statistics calculations are of interest. The mean of n samples is straightforward:

$$\overline{X} = \frac{\sum X}{n}$$

The standard deviation s of n samples taken from a normally distributed (Gaussian) population of values, the "sample standard deviation," is

$$s = \sqrt{\frac{\sum (X-\overline{X})^2}{(n-1)}}$$

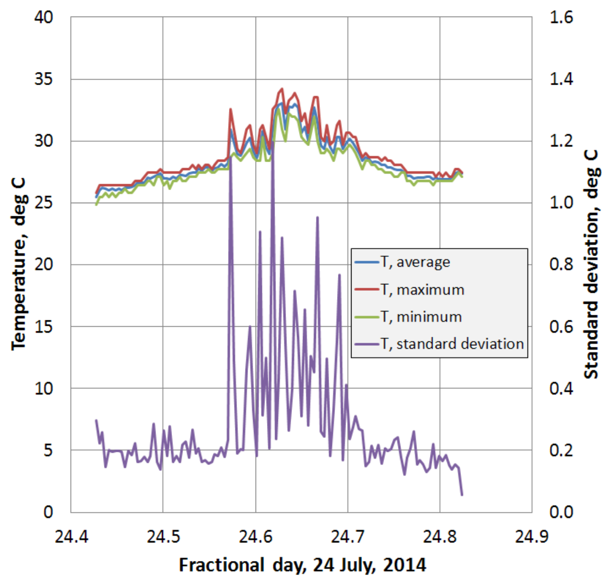
For computational purposes, the sample standard deviation is calculated by totaling the sum of the X's and the sum of the square of the X's as the X's are read one at a time.

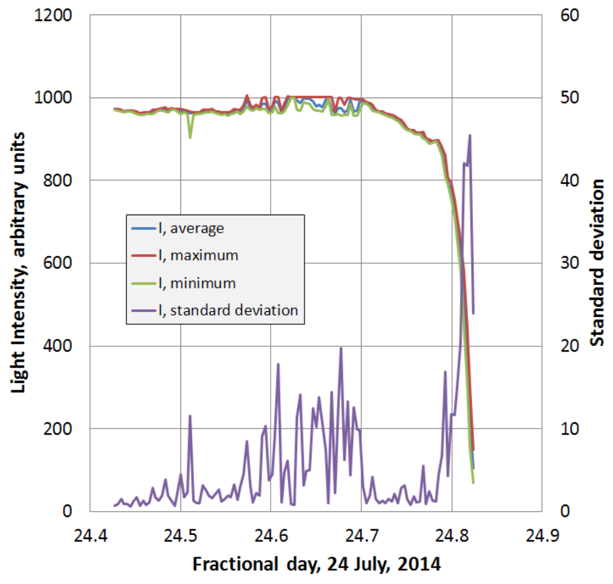
$$s = \sqrt{\frac{\sum X^2 - (\sum X)^2/n}{n-1}}$$

The standard deviation calculation can always be performed on any set of values, but it assumes a Gaussian distribution of the X values. Depending on what is being measured, this may or may not be true. For example, when a quantity is changing over time rather than fluctuating randomly, as light levels and temperatures are likely to do, a standard deviation calculated over some time interval may be interesting, but it is no longer a standard deviation in the statistical sense.

If values are constant over time, the standard deviation is 0 by definition. However, in that case, real number math to calculate the quantity under the square root sign *might* result in a very small negative number instead of 0, which can cause an error when the `sqrt()` function is used. To avoid this error, the value of the expression under the square root should be tested and the standard deviation assigned a value of 0 if that value is negative.

Figure 10 shows temperature and light intensity data recorded on the inside window ledge of my office, with 10-second sampling and statistics calculated over 5 minutes. The region with increased standard deviation for both temperature and light intensity corresponds to fluctuations due to changing cloud conditions. For data like these, the "standard deviation" is more properly interpreted just as a measure of the variability of the quantity being measured during a sampling interval. The light intensity appears to be "saturating" in the bright sunlight – that is, the resistance of the CdS photoresistor is no longer changing linearly with light intensity. One could experiment with different photoresistors to change this performance.





10-second sampling over 5-minutes.

Figure 10(a). Temperature, ºC. Figure 10(b). light intensity, arbitrary units.

Sketch 23.

```
// MyLightTempLoggerC.ino
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <RTClib.h>
#define ECHO_TO_FILE 1
#define ECHO_TO_SERIAL 0
// input [int]dt,[char]m or s,[int]dtSave,XXXXXXXX.CSV
// example: 10,s,5,logfile1.csv
// global variables
float sumX=0,sumY=0,sumXX=0,sumYY=0;
float maxTemperature=-
100,minTemperature=150,maxPhotocell=0,minPhotocell=1023;
int N,KNT=0;
File logfile; // the logging file
int Second,delay_t=1000,dt,dtSave;
char intervalType,outFile[13];
int tempReading, photocellReading,Year,Month,Day,Hour,Minute;
float temperatureC;
const float aref_voltage=3.3;
const int photocellPin=0,tempPin=1,chipSelect=10;
RTC_DS1307 RTC; // Define real time clock object.
// Read configuration parameters from serial port.
void getConfiguration() {
  char c; int n=0;
  while (Serial.peek()<0) {}
  dt=Serial.parseInt(); delay(10); Serial.read();
  delay(10);
  intervalType=Serial.read(); delay(10); Serial.read();
  delay(10);
  dtSave=Serial.parseInt(); delay(10); Serial.read();
  while (Serial.available()) {
    delay(10); if (Serial.available()>0) {
      c=Serial.read(); outFile[n]=c; n++;
    }
  }
  outFile[n]='\0'; N=dtSave*60/dt;
}
//--------------------
// Get data and display or log it.
void dataOutput(int N) {
  float std_devX,meanX,std_devY,meanY,day_frac,a;
  photocellReading=analogRead(photocellPin); delay(10);
  tempReading=analogRead(tempPin); delay(10);
  temperatureC = (tempReading*aref_voltage/1023 - 0.5)*100;
  sumX+=temperatureC; sumXX+=temperatureC*temperatureC;
  // (float) the square of the photocell reading to avoid integer overflow.
  sumY+=photocellReading; sumYY+=photocellReading*(float)photocellReading;
  KNT++;
  if (temperatureC>maxTemperature) maxTemperature=temperatureC;
  if (temperatureC<minTemperature) minTemperature=temperatureC;
  if (photocellReading>maxPhotocell) maxPhotocell=photocellReading;
  if (photocellReading<minPhotocell) minPhotocell=photocellReading;
  day_frac=Day+Hour/24.+Minute/1440.+Second/86400.;
  #if ECHO_TO_SERIAL
    Serial.print(Year); Serial.print('/'); Serial.print(Month);
    Serial.print('/'); Serial.print(Day); Serial.print(' ');
    Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
    Serial.print(':'); Serial.print(Second);
    Serial.print(' '); Serial.print(day_frac,5);
    Serial.print(' '); Serial.print(temperatureC,4);
    Serial.print(' '); Serial.print(photocellReading);
    Serial.println();
  #endif // ECHO_TO_SERIAL
    if (KNT==N) {
      a=sumXX-sumX*sumX/N; if (a<0) a=0;
      std_devX=sqrt(a/(N-1));
      meanX=sumX/N;
      a=sumYY-sumY*sumY/N; if (a<0) a=0;
      std_devY=sqrt(a/(N-1));
      meanY=sumY/N;
      KNT=0; sumX=0; sumXX=0; sumY=0; sumYY=0;
      #if ECHO_TO_SERIAL
        Serial.print(Year); Serial.print('/'); Serial.print(Month);
        Serial.print('/'); Serial.print(Day); Serial.print(' ');
        Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
        Serial.print(':'); Serial.print(Second
);
        Serial.print(' '); Serial.print(day_frac,5); Serial.print(' ');
        Serial.print(meanX,4); Serial.print(',');
        Serial.print(maxTemperature,4);
        Serial.print(','); Serial.print(minTemperature,4);
        Serial.print(','); Serial.print(std_devX,4); Serial.print(',');
        Serial.print(meanY,4); Serial.print(','); Serial.print(maxPhotocell);
        Serial.print(','); Serial.print(minPhotocell);
        Serial.print(','); Serial.print(std_devY,4); Serial.println();
      #endif // ECHO_TO_SERIAL

      #if ECHO_TO_FILE
        logfile.print(Year); logfile.print(','); logfile.print(Month);
        logfile.print(',');
        logfile.print(Day); logfile.print(','); logfile.print(Hour);
        logfile.print(',');
        logfile.print(Minute); logfile.print(','); logfile.print(Second);
        logfile.print(',');
        logfile.print(day_frac,5); logfile.print(',');
        logfile.print(meanX,4); logfile.print(',');
        logfile.print(maxTemperature,4);
        logfile.print(','); logfile.print(minTemperature,4);
        logfile.print(','); logfile.print(std_devX,4); logfile.print(',');
        logfile.print(meanY,4); logfile.print(',');
        logfile.print(maxPhotocell);
        logfile.print(','); logfile.print(minPhotocell);
        logfile.print(','); logfile.print(std_devY,4); logfile.println();
        logfile.flush(); // write to file
      #endif // ECHO_TO_FILE
      maxTemperature=-
100,minTemperature=150,maxPhotocell=0,minPhotocell=1023;
    }
}
//--------------------------------
void setup() {
  Serial.begin(9600);
  getConfiguration();
  Serial.print("sampling interval: "); Serial.print(dt);
Serial.println(intervalType);
  Serial.print("logging interval: ");
  Serial.print(dtSave); Serial.println(" minutes");
  if ((intervalType=='s') && (dt<2))
    Serial.print("Configuration error. Restart!");
  Serial.print("Log to file: "); Serial.println(outFile);
  pinMode(10,OUTPUT);
  #if ECHO_TO_SERIAL
    Serial.println("Write to serial port.");
    Serial.println("year,month,day,hour,minute,second,day_frac,T_mean,
      T_max,T_min,T_stdDev,Light_mean,PC_max,PC_min,PC_stdDev");
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    Serial.print("Initializing SD card...");
    if (!SD.begin(chipSelect)) {
      Serial.println("Card failed, or not present"); return; }
    else {
      Serial.println("card initialized."); }
    //char filename[]=outFile;
    logfile=SD.open(outFile,FILE_WRITE);
    if (!logfile) {Serial.println("Could not create file."); return; }
    else {Serial.print("Logging to: "); Serial.println(outFile); }
    logfile.println("year,month,day,hour,minute,second,day_frac,T_mean,
     T_max,T_min,T_stdDev,PC_mean,PC_max,PC_min,PC_stdDev");
  #endif // ECHO_TO_FILE
  Wire.begin();
  RTC.begin();
  analogReference(EXTERNAL);
  do {
    DateTime now=RTC.now();
    Minute=now.minute(); Second=now.second();
    delay(10);
  } while ((Minute*60+Second)%(dtSave*60) != 0);
}
void loop() {
  DateTime now=RTC.now();
  Year=now.year(); Month=now.month(); Day=now.day();
  Hour=now.hour(); Minute=now.minute(); Second=now.second();
  if(Second%dt==0) dataOutput(N);
  delay(delay_t);
}
```

3.2 A High-Resolution Datalogger

3.2.1 Hardware

Although the code development in *3.1.4* stands on its own, for many purposes an Arduino-based datalogger is useful only if it is possible to attain a digital resolution much better than what is available with the built-in 10-bit Arduino A/D conversion. Fortunately, the four-channel 16-bit ADS1115 board mentioned at the beginning of this document (https://learn.adafruit.com/adafruit-4-channel-adc-breakouts) makes this very easy!

First, download and install the required `Adafruit_ADS1015.cpp` and `Adafruit_ADS1015.h` files from https://github.com/adafruit/Adafruit_ADS1X15 and install them in their own folder, `\libraries\Adafruit_ADS1015`. Even though the names of these files imply that they are for the 12-bit ADS1015 board, they also contain code for the 16-bit ADS1115 board. The web page source for the library files includes an example sketch for testing the ADS1115 board and there is no reason to duplicate it here.

The Adafruit data logging shield will still be used for its real time clock and SD card interface, but unlike the applications discussed in Section 3.1, the voltage outputs from sensors are connected to the ADS1115 inputs rather than directly to the Arduino's analog input pins. This project will use single-ended inputs for four channels rather than differential inputs for two channels. The Arduino handles all the communications with the ADS1115 and its four input channels through the SCL (clock) and SDA (data) pins. It is simply a matter of "polling" these four input channels one at a time and converting the integer value into an analog voltage. Figure 11(a) shows the connections needed for the ADS1115 board to communicate with the Arduino board.

Figures 11(b) and (c) show a complete datalogger based on the Sparkfun 5-V Arduino Pro microcontroller. Table 5 gives the components list. The Arduino Pro board has been chosen over the Arduino Uno R3 because of its lower power consumption. The plug-in FTDI connector can be switched between multiple devices, so only one is needed regardless of how many loggers you build. The prices shown are for single units. Quantity discounts may be available.

The Arduino Pro is pin-compatible with the Arduino Uno R3 shown in Figure 1. The FTDI board requires a USB mini-B connector rather than the Standard-B connector on the Uno R3. Standard-B connectors are commonly used on printers, for example. Mini-B connectors are used on some digital cameras such as those in the Canon PowerShot series. The ADS1115 power (5V) and ground connections are visible coming from the upper right hand corner of the of the board. The other connections (refer to the ADS1115 documentation) are made on the back side of the datalogger shield. The screw terminals are, from right to left in this image, GND, GND, A0, A1, A2, A3.

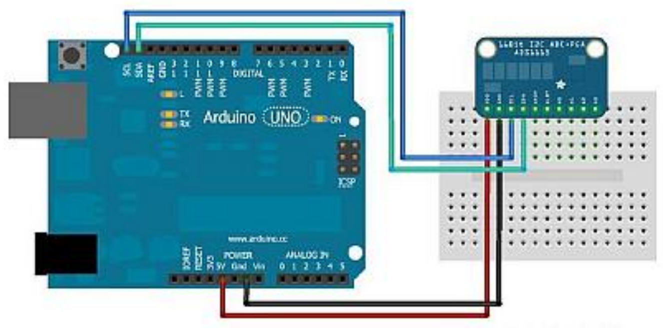


Figure 11(a). Connection the ADS1115 board (see https://learn.adafruit.com/downloads/pdf/adafruit-4-channel-adc-breakouts.pdf).

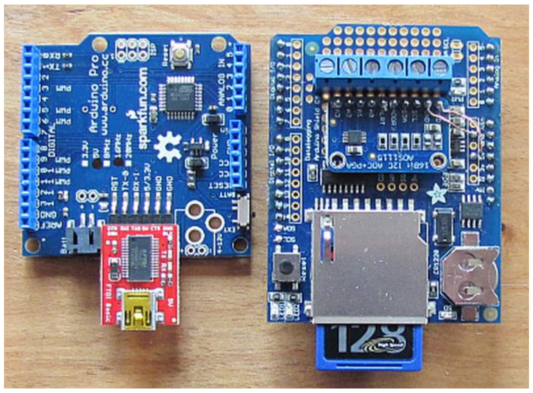


Figure 11(b). 5-V ADS1115 A/D board with Sparkfun Arduino Pro and FTDI/USB board with Adafruit data logging shield.

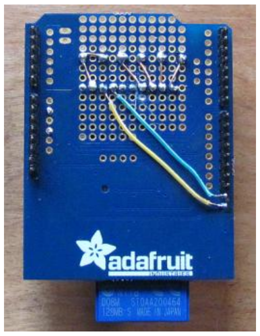


Figure 11(c). Back of data logging shield.

Table 5. Components for a high-resolution datalogger.

Component	**Source**	**Price (as of July 2014)**
Arduino Pro 5-V microcontroller, DEV-10915	www.sparkfun.com	$14.95
Female header pack, PRT-11269		$1.50
FTDI basic breakout board – 5V, DEV-09716		$14.95
Adafruit data logging Shield, PID 1141	www.adafruit.com	$19.95
ADS1115 16-bit 4-channel ADC, PID 1085		$14.95
3-position terminal block (2), ED2610-ND	www.digikey.com	$0.51 each
SD card	Various sources	~$5-$7
Total cost (approximate, not including USB cable and shipping)		**~$75**

3.2.2 Programming

Sketch 24 is a modification of Sketch 22 which uses the hardware shown in Figure 4. The logic for reading and recording data is the same as in Sketch 21, but the code for logging temperature and light data at 10-bit resolution is replaced by code for accessing input from up to four voltage-producing sensors connected to the ADS1115 board. Configuration parameters for controlling how the data are collected and logged are read from the serial port window when the sketch is uploaded:

- (integer) the sampling interval in minutes or seconds
- (character) m or s, identifies the sampling interval as minutes or seconds
- (integer) a value that sets the gain for the ADS1115 board
- (string of characters) name of the output file (12 total characters, including extension)

Each parameter is separated with a comma, with no spaces. For example, `5,s,1,logfile1.csv` will sample data at 5-second intervals, using a `GAIN_ONE` setting (see Table 6, below), and log the data to `logfile1.csv` if the `ECHO_TO_FILE` directive is set to 1.

The pre-compile directives should be changed manually before the code is uploaded to the microcontroller. The `ECHO_TO_FILE` directive turns data logging to an SD card on (1) or off (0) and the `ECHO_TO_SERIAL` directive turns serial port output on or off. In both cases, there is some initial serial port output to show the configuration parameters and to make sure that the SD card is working properly if `ECHO_TO_FILE` is turned on. The `ECHO_TO_SERIAL` directive is useful for checking the operation of the sketch, especially if you make changes to the code.

The ADS boards have six possible programmable gains, with resolutions as shown in Table 6 for both the ADS1015 (12-bit) and ADS1115 (16-bit). (See `ads1115.setGain(GAIN_ONE);` in Sketch 24.) The total (±) input range applies to differential operation. The ADS board returns 16-bit signed integers, so the effective digital-to-analog resolution for single-ended operation is only 15 bits. For example, for single-ended operation at a `GAIN_ONE` setting the resolution is $4.096•2/2^{15}$ = 8.192/32767=0.250 mV, and the conversion from integer values from 0 to 32767 returned when the inputs are polled is (Ax/32767)•4.096. You can test this calculation by connecting a 1.5V battery (or some other known voltage source) between one of the inputs and ground, recording some data, and checking the conversion from integer values to volts; use a voltmeter to monitor the actual voltage of the battery, which will be above 1.5V when it is new.

Although the default gain setting of `GAIN_TWOTHIRDS` has a stated range of ±6.144V, in fact the voltage applied to any input channel should never exceed the power supply voltage (5V for the hardware shown in Figure 5).

Table 6. Interpreting ADS gain settings.

Gain setting	**Gain setting code**	**Input range, differential/ single-ended**	**ADS1015/1115 resolution**	**ADS1015/1115 conversion to volts**
`GAIN_TWOTHIRDS` (default)	3	±6.144V/0–6.144V	3 mV/0.375 mV	(Ax/4095)•6.144 (Ax/32767)•6.144
`GAIN_ONE`	1	±4.096V/0–4.096V	2 mV/0.250 mV	(Ax/4095)•4.096 (Ax/32767)•4.096
`GAIN_TWO`	2	±2.048V/0–2.048V	1 mV/0.125 mV	(Ax/4095)•2.048 (Ax/32767)•2.048
`GAIN_FOUR`	4	±1.024V/0–1.024V	0.5 mV/0.0625 mV	(Ax/4095)•1.024 (Ax/32767)•1.024
`GAIN_EIGHT`	8	±0.512V/0–0.512V	0.25 mV/0.03125 mV	(Ax/4095)•0.512 (Ax/32767)•0.512
`GAIN_SIXTEEN`	16	±0.256V/0–0.256V	0.125mV/0.015625 mV	(Ax/4095)•0.256 (Ax/32767)•0.256

Input channels with no voltage source connected will produce spurious and meaningless values. They can be ignored, of course, and you don't even have to read values from unused channels, but it is simpler just to read all the channels. It is a good idea to connect unused inputs to ground.

Sketch 24.

```
// HiResDataLogger.ino
// Format for serial port window input:
// [int]dt,[char]m or s,[int]gain 1,2,3 (for 2/3),4,8, or 16,
[string]xxxxxxxx.xxx
#include <SD.h>
#include <Wire.h>
#include <RTClib.h>
#include <Adafruit_ADS1015.h>
#define ECHO_TO_FILE 0
#define ECHO_TO_SERIAL 1
Adafruit_ADS1115 ads1115;
File logfile; // the logging file
int Second,delay_t=1000,dt,chipSelect=10,gain;
char intervalType, outFile[13];
int Year,Month,Day,Hour,Minute;
float DtoA;
RTC_DS1307 RTC; // Define real time clock object.
void getConfiguration() {
  char c; int n=0;
  while (Serial.peek()<0) {}
  dt=Serial.parseInt();
  delay(10); Serial.read(); intervalType=Serial.read();
  delay(10); Serial.read(); gain=Serial.parseInt();
  delay(10); Serial.read();
  while (Serial.available()) {
    delay(10); if (Serial.available()>0) {
      c=Serial.read(); outFile[n]=c; n++;
    }
  }
  outFile[n]='\0';
}
void dataOutput() {
  int16_t adc0,adc1,adc2,adc3;
  adc0 = ads1115.readADC_SingleEnded(0);
  adc1 = ads1115.readADC_SingleEnded(1);
  adc2 = ads1115.readADC_SingleEnded(2);
  adc3 = ads1115.readADC_SingleEnded(3);
  #if ECHO_TO_SERIAL
    Serial.print(Year); Serial.print('/'); Serial.print(Month);
    Serial.print('/'); Serial.print(Day); Serial.print(' ');
    Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
    Serial.print(':'); Serial.print(Second);
    Serial.print(' ');
Serial.print(Day+Hour/24.+Minute/1440.+Second/86400.,5);
    Serial.print(' '); Serial.print(adc0);
    Serial.print(' '); Serial.print(adc1);
    Serial.print(' '); Serial.print(adc2);
    Serial.print(' '); Serial.print(adc3);
    Serial.println();
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    logfile.print(Year); logfile.print(',');
    logfile.print(Month); logfile.print(',');
    logfile.print(Day); logfile.print(',');
    logfile.print(Hour); logfile.print(',');
    logfile.print(Minute); logfile.print(',');
    logfile.print(Second); logfile.print(',');
    logfile.print(Day+Hour/24.+Minute/1440.+Second/86400.,5);
    logfile.print(','); logfile.print(adc0);
    logfile.print(','); logfile.print(adc1);
    logfile.print(','); logfile.print(adc2);
    logfile.print(','); logfile.print(adc3);
    logfile.println();
    logfile.flush(); // write to file
  #endif // ECHO_TO_FILE
}
void setup() {
  Serial.begin(9600);
  getConfiguration();
  Serial.print("logging interval: ");Serial.print(dt);
  Serial.println(intervalType);
  Serial.print("Log to file: "); Serial.println(outFile);
  pinMode(10,OUTPUT);
  #if ECHO_TO_SERIAL
    Serial.print("ADS gain setting = "); Serial.println(gain);
    Serial.println("Write to serial port.");
    Serial.println("year,month,day,hour,minute,second,day_frac,A0,A1,A2,A3");
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    Serial.print("Initializing SD card...");
    if (!SD.begin(chipSelect)) {
      Serial.println("Card failed, or not present"); return; }
    else {
      Serial.println("card initialized."); }
    logfile=SD.open(outFile,FILE_WRITE);
    if (!logfile) {Serial.println("Could not create file."); return; }
    else {Serial.print("Logging to: "); Serial.println(outFile); }
    logfile.println("year,month,day,hour,minute,second,day_frac,light,T_C");
  #endif // ECHO_TO_FILE
  Wire.begin(); RTC.begin(); ads1115.begin();
  Serial.print("Gain setting = ");
  switch(gain) {
    case 1: {ads1115.setGain(GAIN_ONE); DtoA=4.096/32768;
            Serial.println("GAIN_ONE"); break;}
    case 2: {ads1115.setGain(GAIN_TWO); DtoA=2.048/32768;
            Serial.println("GAIN_TWO"); break;}
    case 3: {ads1115.setGain(GAIN_TWOTHIRDS); DtoA=6.144/32768;
            Serial.println("GAIN_TWOTHIRDS"); break;}
    case 4: {ads1115.setGain(GAIN_FOUR); DtoA=1.024/32768;
            Serial.println("GAIN_FOUR"); break;}
    case 8: {ads1115.setGain(GAIN_EIGHT); DtoA=0.512/32768;
            Serial.println("GAIN_EIGHT"); break;}
    case 16: {ads1115.setGain(GAIN_SIXTEEN); DtoA=0.256/32768;
            Serial.println("GAIN_SIXTEEN"); break;}
    default: {Serial.println("Oops... no such gain setting!"); return; }
  }
}
void loop() {
  DateTime now=RTC.now();
  Year=now.year(); Month=now.month(); Day=now.day();
  Hour=now.hour(); Minute=now.minute(); Second=now.second();
  if (intervalType=='s') {
    if ((Second%dt)==0) dataOutput();
  }
  if (intervalType=='m') {
    if ((Minute%dt==0) && (Second==0)) dataOutput();
  }
  delay(delay_t);
}
```

Sketch 25 is a modification of Sketch 23. It calculates statistics for each of the four input channels. For example, `10,s,5,logfile1.csv` samples data every 10 seconds and calculates statistics with thirty values over 5 minutes. Statistics cannot be calculated over an interval of less than 1 minute. The longest time period for calculating statistics is one hour (specify 0 minutes in the configuration input).

Sketch 25.

```
//HiResDataLoggerB
// Format for serial port window input:
// [int]dt,[char]m or s,[int]dtSave (1-60),
// [int]gain 1,2,3 (for 2/3),4,8, or 16,[string]xxxxxxxx.xxx
#include <SD.h>
#include <Wire.h>
#include <RTClib.h>
#include <Adafruit_ADS1015.h>
#define ECHO_TO_FILE 0
#define ECHO_TO_SERIAL 1
Adafruit_ADS1115 ads1115;
float max0=-100.,min0=100.,max1=-100.,min1=100.;
float max2=-100.,min2=100.,max3=-100.,min3=100.;
float sumX0=0.,sumXX0=0.,sumX1=0.,sumXX1=0.,sumX2=0.;
float sumXX2=0.,sumX3=0.,sumXX3=0.;
float std_dev0,mean0,std_dev1,mean1,std_dev2,mean2,std_dev3,mean3;
float adc0,adc1,adc2,adc3;
float DtoA,dayFrac;
File logfile; // the logging file
int N,KNT=0;
int delay_t=1000,dt,dtSave,chipSelect=10,gain;
int Year,Month,Day,Hour,Minute,Second;
char intervalType, outFile[13];
RTC_DS1307 RTC; // Define real time clock object.
void getConfiguration() {
  char c; int n=0;
  while (Serial.peek()<0) {}
  dt=Serial.parseInt();
  delay(10); Serial.read(); intervalType=Serial.read();
  delay(10); dtSave=Serial.parseInt();
  delay(10); Serial.read(); gain=Serial.parseInt();
  delay(10); Serial.read();
  while (Serial.available()) {
    delay(10); if (Serial.available()>0) {
      c=Serial.read(); outFile[n]=c; n++;
    }
  }
  outFile[n]='\0';
  if (intervalType=='s') N=dtSave*60/dt; else N=dtSave/dt;
}
void dataOutput() {
```

```
adc0 = DtoA*ads1115.readADC_SingleEnded(0);
sumX0+=adc0; sumXX0+=adc0*adc0;
adc1 = DtoA*ads1115.readADC_SingleEnded(1);
sumX1+=adc1; sumXX1+=adc1*adc1;
adc2 = DtoA*ads1115.readADC_SingleEnded(2);
sumX2+=adc2; sumXX2+=adc2*adc2;
adc3 = DtoA*ads1115.readADC_SingleEnded(3);
sumX3+=adc3; sumXX3+=adc3*adc3;
KNT++;
if (adc0>max0) max0=adc0; if (adc0<min0) min0=adc0;
if (adc1>max1) max1=adc1; if (adc1<min1) min1=adc1;
if (adc2>max2) max2=adc2; if (adc2<min2) min2=adc2;
if (adc3>max3) max3=adc3; if (adc3<min3) min3=adc3;
dayFrac=Day+Hour/24.+Minute/1440.+Second/86400.;
#if ECHO_TO_SERIAL
  Serial.print(Year); Serial.print('/'); Serial.print(Month);
  Serial.print('/'); Serial.print(Day); Serial.print(',');
  Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
  Serial.print(':'); Serial.print(Second); Serial.print(',');
  Serial.print(dayFrac,5); Serial.print(',');
  Serial.print(adc0,5); Serial.print(',');
  Serial.print(adc1,5); Serial.print(',');
  Serial.print(adc2,5); Serial.print(',');
  Serial.print(adc3,5); Serial.println();
#endif // ECHO_TO_SERIAL
if (KNT==N) {
    // Protect against small negative values
    // (possible with real number arithmetic
    // under some conditions when the values
    // don't change during sampling interval).

  std_dev0=sqrt(max(0,(sumXX0-sumX0*sumX0/N)/(N-1)));
  std_dev1=sqrt(max(0,(sumXX1-sumX1*sumX1/N)/(N-1)));
  std_dev2=sqrt(max(0,(sumXX2-sumX2*sumX2/N)/(N-1)));
  std_dev3=sqrt(max(0,(sumXX3-sumX3*sumX3/N)/(N-1)));
  mean0=sumX0/N; mean1=sumX1/N; mean2=sumX2/N; mean3=sumX3/N;
  #if ECHO_TO_SERIAL
    Serial.print(Year); Serial.print('/'); Serial.print(Month);
    Serial.print('/'); Serial.print(Day); Serial.print(',');
    Serial.print(Hour); Serial.print(':'); Serial.print(Minute);
    Serial.print(':'); Serial.print(Second); Serial.print(',');
    Serial.print(dayFrac,5); Serial.print(',');
    Serial.print(mean0,5); Serial.print(','); Serial.print(max0,5);
    Serial.print(',');
    Serial.print(min0,5); Serial.print(',');  Serial.print(std_dev0,8);
    Serial.print(',');
    Serial.print(mean1,5); Serial.print(','); Serial.print(max1,5);
    Serial.print(',');
    Serial.print(min1,5); Serial.print(',');  Serial.print(std_dev1,8);
```

```
      Serial.print(',');
      Serial.print(mean2,5); Serial.print(','); Serial.print(max2,5);
      Serial.print(',');
      Serial.print(min2,5); Serial.print(',');  Serial.print(std_dev2,8);
      Serial.print(',');
      Serial.print(mean3,5); Serial.print(','); Serial.print(max3,5);
      Serial.print(',');
      Serial.print(min3,5); Serial.print(',');  Serial.println(std_dev3,8);
    #endif //ECHO_TO_SERIAL
    #if ECHO_TO_FILE
      logfile.print(Year); logfile.print('/'); logfile.print(Month);
      logfile.print('/'); logfile.print(Day); logfile.print(',');
      logfile.print(Hour); logfile.print(':'); logfile.print(Minute);
      logfile.print(':'); logfile.print(Second); logfile.print(',');
      logfile.print(dayFrac,5); logfile.print(',');
      logfile.print(mean0,5); logfile.print(','); logfile.print(max0,5);
      logfile.print(',');
      logfile.print(min0,5); logfile.print(',');  logfile.print(std_dev0,8);
      logfile.print(',');
      logfile.print(mean1,5); logfile.print(','); logfile.print(max1,5);
      logfile.print(',');
      logfile.print(min1,5); logfile.print(',');  logfile.print(std_dev1,8);
      logfile.print(',');
      logfile.print(mean2,5); logfile.print(','); logfile.print(max2,5);
      logfile.print(',');
      logfile.print(min2,5); logfile.print(',');  logfile.print(std_dev2,8);
      logfile.print(',');
      logfile.print(mean3,5); logfile.print(','); logfile.print(max3,5);
      logfile.print(',');
      logfile.print(min3,5); logfile.print(',');
      logfile.println(std_dev3,8);
      logfile.flush();
    #endif //ECHO_TO_FILE
    KNT=0;
    max0=-100.,min0=100.,max1=-100.,min1=100.,max2=-100.;
    sumX0=0.,sumXX0=0.,sumX1=0.,sumXX1=0.,sumX2=0.;
    sumXX2=0.,sumX3=0.,sumXX3=0.;
  }
}
void setup() {
  Serial.begin(9600); getConfiguration();
  Serial.print("sampling interval, logging interval: ");Serial.print(dt);
  Serial.print(intervalType);
  Serial.print(", ");Serial.print(dtSave);Serial.println('m');
  Serial.print("statistics computed with ");
  Serial.print(N);Serial.println(" samples.");
  pinMode(10,OUTPUT);
  #if ECHO_TO_SERIAL
    Serial.print("ADS gain setting = "); Serial.println(gain);
    Serial.println("Write to serial port.");
// no line breaks allowed in code! Put print string all on one line.
   Serial.println("date,time,day_frac,A0_mean,A0_max,A0_min,A0_stddev,
    A1_mean,A1_max,A1_min,A1_stddev,A2_mean,A2_max,A2_min,A2_stddev,
    A3_mean,A3_max,A3_min,A3_stddev");
  #endif // ECHO_TO_SERIAL
  #if ECHO_TO_FILE
    Serial.print("Initializing SD card...");
    if (!SD.begin(chipSelect)) {
      Serial.println("Card failed, or not present"); return; }
    else {
      Serial.println("card initialized."); }
    logfile=SD.open(outFile,FILE_WRITE);
    if (!logfile) {Serial.println("Could not create file."); return; }
    else {Serial.print("Logging to: "); Serial.println(outFile); }
// no line breaks allowed in code! Put print string all on one line.
    logfile.println("date,time,day_frac,A0_mean,A0_max,A0_min,A0_stddev,
     A1_mean,A1_max,A1_min,A1_stddev,A2_mean,A2_max,A2_min,A2_stddev,
    A3_mean,A3_max,A3_min,A3_stddev");
  #endif // ECHO_TO_FILE
  Wire.begin(); RTC.begin(); ads1115.begin();
  Serial.print("Gain setting = ");
  switch(gain) {
    case 1: {ads1115.setGain(GAIN_ONE); DtoA=4.096/32768;
            Serial.println("GAIN_ONE"); break;}
    case 2: {ads1115.setGain(GAIN_TWO); DtoA=2.048/32768;
            Serial.println("GAIN_TWO"); break;}
    case 3: {ads1115.setGain(GAIN_TWOTHIRDS); DtoA=6.144/32768;
            Serial.println("GAIN_TWOTHIRDS"); break;}
    case 4: {ads1115.setGain(GAIN_FOUR); DtoA=1.024/32768;
            Serial.println("GAIN_FOUR"); break;}
    case 8: {ads1115.setGain(GAIN_EIGHT); DtoA=0.512/32768;
            Serial.println("GAIN_EIGHT"); break;}
    case 16: {ads1115.setGain(GAIN_SIXTEEN); DtoA=0.256/32768;
            Serial.println("GAIN_SIXTEEN"); break;}
    default: {Serial.println("Oops... no such gain setting!"); return; }
  }
  do { // wait until multiple of dtSave minutes, 0 sec
    DateTime now=RTC.now();
    Minute=now.minute(); Second=now.second();
  } while ((Minute%dtSave+Second) !=0);
}
void loop() {
  DateTime now=RTC.now();
  Year=now.year(); Month=now.month(); Day=now.day();
  Hour=now.hour(); Minute=now.minute(); Second=now.second();
  if (intervalType=='s') {
    if ((Second%dt)==0) dataOutput();
  }
  else {
    if ((Minute%dt)==0) dataOutput();
  }
  delay(delay_t); // Don't process the same second twice.
}
```

Figure 12 shows data from one channel of an ADS1115 board, collected using Sketch 25. The data are from a pyranometer (http://www.instesre.org/construction/pyranometer/pyranometer.htm) which measures incoming solar radiation. Sampling starts at a multiple of 5 minutes, with sampling every 10 seconds thereafter and statistics generated every 5 minutes. Initially the pyranometer is in shadow. It emerges from the shadow at around 24.39 days. The sky was partly cloudy with scattered to broken cumulus, which explains the large swings between maximum and minimum values and the corresponding swings in standard deviation. (Recall the comments about standard deviation calculations in the discussion of Sketch 23.)

The gain was set to 16. Even though the voltages are small compared to the total range of 0.256V, the 16-bit resolution is still more than adequate for this measurement. For a calibrated pyranometer (which this one wasn't) the sketch could be modified to convert voltage to watts/m^2.

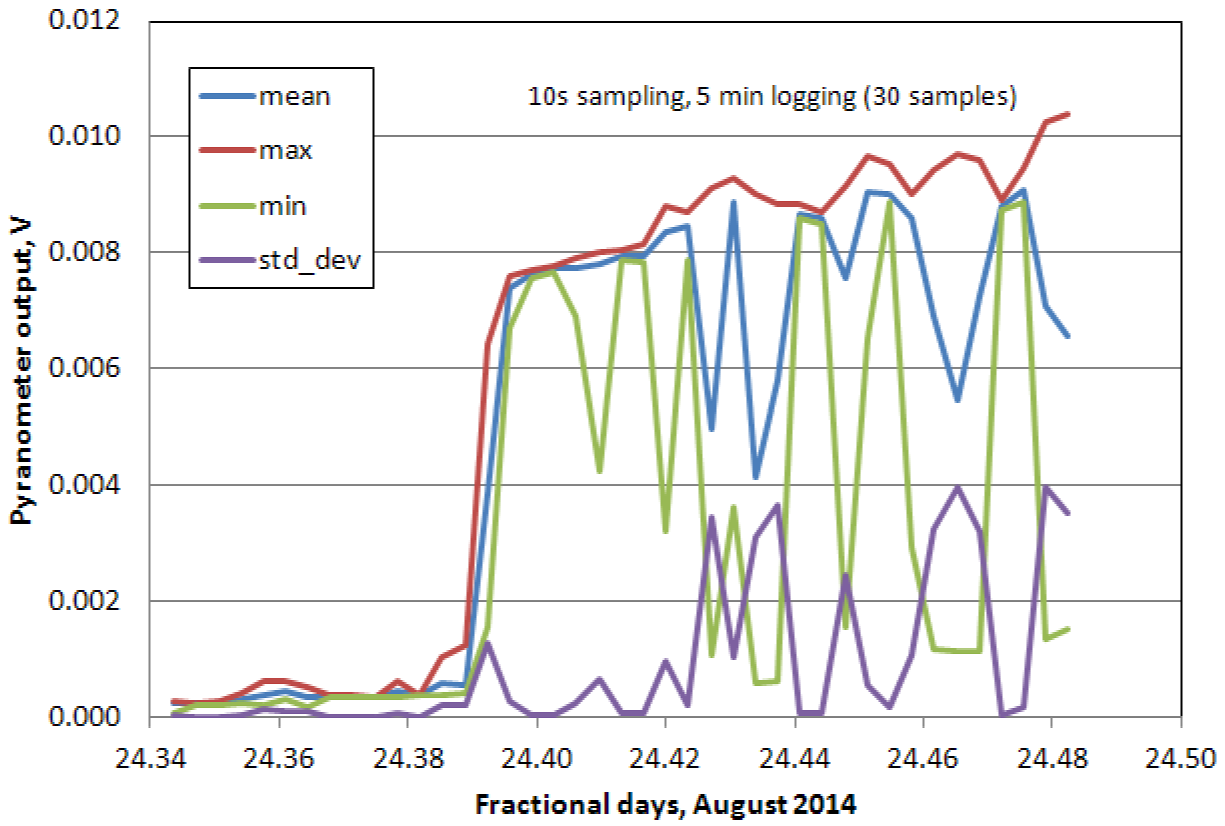


Figure 12. Sample statistics output from one channel of an ADS1115 board.

4. SOME ADDITIONAL CONSIDERATIONS

4.1 Expanding the Number of High-Resolution Channels

Because up to four different addresses can be assigned to each ADS1015 or 1115 board, up to four boards can be connected at the same time. For details, see https://learn.adafruit.com/adafruit-4-channel-adc-breakouts. Whether this is worthwhile or not depends on your application. Calculating statistics for each channel produces 4 values per channel (mean, maximum, minimum, and standard deviation), a total of 16 values for one board. This is potentially a *lot* of data to handle. On the plus side, you can assign different gain settings to each board.

4.2 Enclosures for Your Arduino Datalogger

Finding an appropriate enclosure for an Arduino project is not just a matter of appearances. Using any kind of exposed electronics in very humid environments or in a situation where moisture can condense on a pc board must be avoided, as this will almost certainly destroy the board. There are commercial enclosures for Arduinos, but they can be relatively expensive and they are certainly not necessarily weatherproof. (A simple plastic enclosure from Sparkfun, PRT-10088, cost $12 in July, 2014, compared to only $15 for an Arduino Pro board.) It is possible to make your own enclosures much more cheaply using, for example, standard plastic household electric outlet boxes and covers from your local home supply or hardware store. You will have to cut holes for the USB cable and power supply – a Dremel tool is useful for this, although it *can* be done with a drill and sharp knife. You could also use a "snap top" plastic sandwich container. In any case, for outdoor use, it is a good idea to fill any container with something which will absorb moisture, such as uncooked rice or cat litter.

4.3 Powering Your Arduino Datalogger

Even when they are not powering sensors, Arduinos used as dataloggers require a lot of power compared to something like the four-channel Onset Computer Corporation UX120-006M loggers, which will run continuously for months on two AAA batteries (www.onsetcomp.com/products/data-loggers/ux120-006m). This isn't a problem for indoor applications near a computer because any of the projects described in this document will happily run on power supplied through a USB port. But what about applications away from a power source?

Figure 13 shows voltage as a function of time, at room temperature, using six Duracell ProCell alkaline D cells in series (using a two-cell holder and a four-cell holder, www.allelectronics.com BH-143 and BH-141) epoxied together and wired in series). The initial voltage is a little less than 9V because the batteries were not brand new when I used them for this test. They are powering an Arduino Uno R3 board and Adafruit data logging shield with an ADS1115 board. Data from all four channels are logged to an SD card file every 30 seconds. As shown in Figure 13, battery voltage is decreasing by a few tenths of a volt per day, indoors at room temperature. Possibly the rate of decrease with an Arduino Pro board would be smaller.

Is this typical performance? It is hard to say. Battery discharge rates will depend on what you are asking your microcontroller to do, and on temperature. Outdoors in cold weather, battery life will be shortened, perhaps considerably. So, in any critical data logging operation, you should check the battery voltage regularly. Whenever the voltage reaches 7V, the batteries need to be replaced; at a lower voltage the microcontroller's on-board voltage regulator will stop functioning.

Is using batteries a reasonable solution? That is a financial question rather than a technical one. In July 2014, a 12-pack of D-cell Energizer alkalines cost $10.95 and a 72-pack was $59.70 from www.batterymart.com.

Rechargeable batteries are tempting, but expensive! From the same source, a single rechargeable D cell was $10.95 and a charger for up to four D cells was $27.95. A 9-V rechargeable battery would work for short-term testing – $6.75 for a NiMH rechargeable plus $8 for a two-battery charger from www.batterymart.com – but these batteries will not last long in the field..

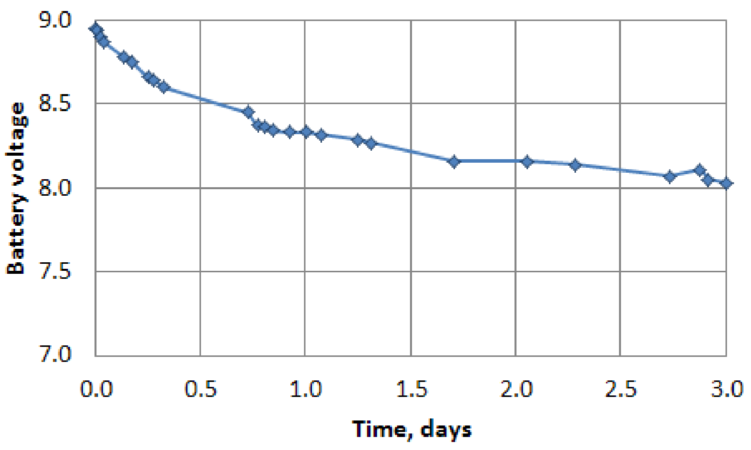


Figure 13. Battery voltage from 6 Energizer alkaline D cells, with Uno R3, Adafruit data logging shield and ADS1115 board, logging every 30 seconds.

It might be tempting to use 8 1.5-V batteries in series, for an initial voltage of a little more than 12 V when the batteries are new, so you wouldn't have to change batteries so often. But I would avoid this temptation. When you connect an external power supply to the Arduino Uno (using the 2.1 mm jack), that voltage is delivered to a 7805 5-V regulator on the Uno board. So-called linear regulators like the 7805 are not very efficient devices. Excess power ((input voltage – 5V) × current) must be dissipated as heat by the regulator and board. (That is why some versions of 780x regulators have a metal tab for connecting to an external heat sink – see the example shown below in Figure 14.) Even though the 7805 regulators on Arduino boards are rated for an input up to 18V, excess heat from an input greater than 12V may damage the regulator and the board. The 9(+) volts generated from 6 1.5-V batteries connected in series may generate some heat on the board, but it shouldn't cause problems.

For powering your Arduino away from your computer, but where you still have access to 110V power, you can use a "wall wart" power supply, but be careful! Consider a typical transformer-based wall wart rated 9V at 300 mA. This means that it will supply 9V to a circuit drawing 300 mA. However, for lower current loads, the voltage will be much higher – 14 or 15 volts or even higher. As most Arduino applications draw very little current, this means that the onboard regulator will be dissipating more heat than you think due to an input voltage that is much higher than 9V. A better solution is to use a regulated ("switching") supply such as the $7 ID 9V, 1A, supply from adafruit.com. Regulated supplies tend to be more expensive than transformer-based supplies, but they are by far a better choice for working with Arduinos.

For long-term continuous outdoor operation away from a power source, solar power is a reasonable approach, but you need to be careful. Typically, such a system consists of a 12-V lead-acid battery and a solar battery charger. One solution, from www.batterymart.com, is shown in Figure 14. This battery charger/maintainer will charge lead-acid batteries rated at up to 10 Ah. It has suction cups so you can mount it on a window; it will work reasonably well even if it is mounted indoors on a window (ideally south-facing). The battery in Figure 14 is 100 cm (4") long. The solar panel is 240 cm (9.5") square.

www.batterymart.com/p-12v-1_3ah-sealed-lead-acid-battery.html
SLA-12V1-3, $12.95 (July 2014)

www.batterymart.com/p-blsolar-2-12v-2_5w-solar-panel.html ACC-BLSOLAR2, $24.95 (July 2014)

Figure 14. Components for a solar-charged battery supply.

Figure 15 shows a lead acid battery and a 6 D-cell battery pack, with several interchangeable connectors.

A lead-acid battery can be charged to well over 12V and, as noted above, this can cause heat dissipation problems if connected directly to an Arduino board. My advice is to *not* connect a lead-acid battery directly to the power input on an Arduino board. To protect the onboard voltage regulator, you could connect an LM7809 9-V regulator between the battery and the board's power input, which will keep the heat generation to an acceptably low level on the board.

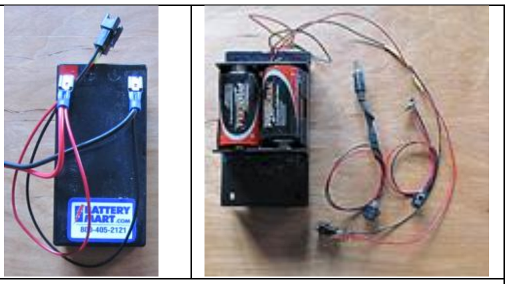


Figure 15. 12-V lead-acid battery and 9-V D-cell pack with interchangeable connectors.

This is a workable solution, but using two 780x voltage regulators – one onboard and another off-board, is not very efficient! It is much more efficient to connect the output from a solar-charged battery to a "synchronous buck voltage regulator" such as the D24V5F9 from Pololu Robotics & Electronics (https://www.pololu.com/product/2845) and connect the output of that device to your Uno either through the power input jack or the Vin pin. These buck down regulators, which are very efficient devices, are more expensive than LM780x devices, ~$5 rather than <$1, but they are worth the cost. Figure 16 shows one of these regulators next to an LM7809 regulator in a TO-220 package. Their pins are on 0.1" centers, just like the TO-220 pins, so they are easy to use with a standard breadboard. They do not need a tab for connection to a heat sink because they don't have to dissipate heat like linear regulators do.

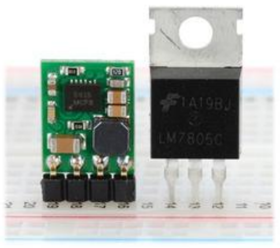


Figure 16. D24V5F9 buck down regulator next to a LM7809 regulator in a TO-220 package.

Although I have noted above that the power demands of an Arduino-based datalogger, even without the power requirements of sensors, are large relative to similar commercial data loggers, for many projects under reasonable sun conditions (even in winter?) those demands can be met by even a small 12-V 1.2Ah lead-acid battery such as the one shown in Figure 15. There is, of course, nothing wrong with using a larger battery for projects that require power support for sensors. The performance of lead-acid batteries is very temperature-dependent. They should be kept charged and not subjected to deep discharges – a condition which shouldn't be a problem in this application. Once in use, lead-acid batteries should not be removed from a charger for long periods of time. Treated properly, they will last for a long time.

Be careful about trying to use a solar panel voltage controller such as the SCN-3 from www.allelectronics.com ($29.95). These devices have one pair of input terminals for the solar panel and two pairs of output terminals – one for the battery and another for the load. The load terminals may automatically turn off if the battery falls below 12V. These devices should work OK if you make connections to the battery out terminals. The current draw even from a relatively inefficient 9-V regulator powering an Arduino project will often be small enough that it should not prevent the battery from charging in sunlight when it is connected to an Arduino board.