Living with the Lab
Fall 2011
v: October 31, 2011

What's this `void loop` thing?

Gerald Recktenwald
gerry@me.pdx.edu

1 Overview

This document aims to explain two kinds of loops: the `loop` function that is a required component of all Arduino sketches, and the standard `for` and `while` loops that are used whenever iteration within a function is needed.

Beginning Arduino programmers can be confused by the semantics and logic of the "`void loop`" function that is required by all Arduino sketches. The `loop` function is repeated continuously because it is called by a hidden loop that controls the execution of the Arduino program. In other words, the `loop` function is the *body* of a loop that is running in a master (or *main*) program. The master program is added automatically to the compiled version of the user sketch before the compiled code is downloaded to the microcontroller.

It's possible, and often necessary, to include `for` loops and `while` loops in the body of the `loop` function. The beginning programmer can get twisted around by the idea that there are loops running inside a function called *loop*. These are not hard concepts to grasp once you understand how each part of the program is being repeated, and how often each repetition is performed. The goal of this document is to expose and clarify those concepts.

We begin with a review of the basic program structure of a sketch. We show that the `loop` and `setup` functions are just ordinary Arduino functions that happen to have special names. Next we review the syntax of `for` and `while` loops. The paper ends with a series of example programs that demonstrate the interaction between the `loop` function and loops iterating inside the `loop` function. The body of the paper covers three main concepts of Arduino programming:

- Basic structure of an Arduino sketch.
- Review of the syntax and use of Arduino functions, with special attention to the `setup` and `loop` functions.
- Syntax and use of `for` loops and `while` loops.

The final section of the paper uses sample programs to demonstrate how the repetition of the `loop` function interacts with `for` and `while` loops inside the `loop` function.

2 Basic Program Structure

All Arduino sketches *must have* two functions: `setup` and `loop`. The `setup` function is executed only once when the Arduino board is first turned on, or when the reset button is pressed. The `loop` function is executed repeatedly (and indefinitely) after the `setup` function is finished.

2.1 Required and Optional Parts of Sketch

Figure 1 is an annotated version of the basic `blink` sketch[1]. The sketch has the required `setup` and `loop` functions, along with an optional header at the top of the file. Most substantial sketches have a header that consists of comment statements and the definitions of global variables.

[1] This is a slightly modified version of the `blink.pde` that is distributed with the Arduino IDE.

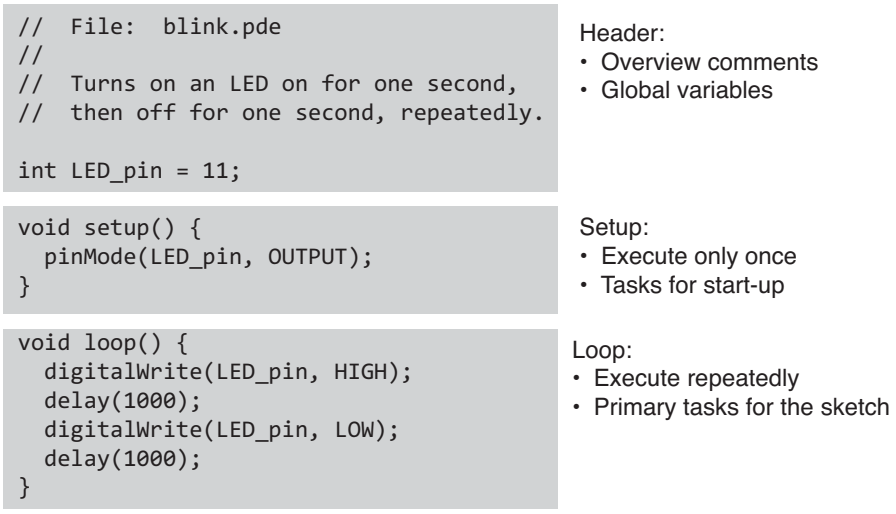

Figure 1: The header, `setup`, and `loop` components of a basic Arduino sketch.

Global variables defined in the header are shared by all functions in the sketch. The `blink` sketch in Figure 1 has one global variable, `LED_pin`. The value of `LED_pin` is available to all functions in the `blink.pde` file. A value is assigned to `LED_pin` in the header (the value `11`). The `LED_pin` variable is used once in `setup` and twice in `loop`.

The `setup` function in the `blink` sketch performs only one task, configuring the digital pin for output. The `loop` function turns the LED on, waits one second, turns the LED off, and waits one more second. That pattern is repeated indefinitely, causing the LED to blink. Note that the blinking is achieved without the need to write an explicit loop inside the body of the `loop` function. That's because the `loop` function is called by the invisible loop in the invisible main program that is added after the sketch is compiled and before it is downloaded to the AVR microcontroller on the Arduino board.

2.2 Structure of an Arduino Function

An Arduino sketch can contain user-defined functions designed to perform a specific sub-task of a more complex program. Programming statements are *encapsulated* in the code block that forms the body of the function. This code block is isolated from other code blocks (i.e., from the code in other functions), and this isolation is an important mechanism for keeping code blocks from adversely interacting with each other.

Of course, programs also needs a mechanism to get data into a function and to obtain a result from a function. Input and output from a function is implemented with *input parameters* and *return values*. The programmer decides what values are needed (what inputs) to perform a computation. The programmer also decides what results (what outputs) need to be returned to the part of the program that invoked or *called* the function.

Functions allow programs to build reusable chunks of code for tasks that appear more than once in a sketch. Functions also provide a way to use the same code in other sketches. For example, the sidebar on the next page shows a function that computes the tangent of an angle. This is a contrived example because the Arduino IDE provides a `tan` function for computing the tangent. One could imagine other computations that are less common, for example to read 20 values from a photoresistor and return the average reading. Such a `read_photoresistor` function would have

a more complex function body, but could share the input and output structure of the `tangent` function listed in the sidebar.

A complete exposition on the design and use of functions is beyond the scope of this document[2]. For our immediate purposes, it is important to recognize that `setup` and `loop` are ordinary Arduino functions. The names `setup` and `loop` are special because they are required by other parts of the Arduino software architecture.

All Arduino functions have the potential for multiple input parameters and a single return value. The inputs and return values are specified in the one-line function definition statement that is required at the start of any function.

Figure 2 shows a skeletal `loop` function. The function definition statement begins with `void`, which declares that the `loop` function is *not* going to return a value. The `void` keyword is not optional: all functions must indicate the kind of value that they return to the calling program. Some other types of return values are `int`, `float`, `double` and `char`. More generally, return values can be any type of Arduino variable[3]. To indicate that the function returns no value, the function is declared to have a `void` return value.

A function with an input and an output

In contrast to the `loop` function, consider the function defined below that returns the tangent of an input value.

```
double tangent( double x ) {
  double t = sin(x)/cos(x);
  return(t);
}
```

The function definition for `tanget` indicates that a value of type `double` is returned to the calling function when `tangent` terminates. The function definition also specifies that an input value of type `double` must be supplied by the calling function. Inside the body of the `tangent` function, the input value is stored in the variable `x`.

As shown in Figure 2,the `loop` function has no input parameters. The empty parentheses are required: you cannot simply drop the parentheses for a function that has no input our output.

The `setup` function has a different purpose from the `loop` function. However, it is similar to `loop` in that it does not return a value, and it has no input parameters. In summary,

- `setup` and `loop` have no return values. Hence the function definitions begin with `void setup` and `void loop`.
- `setup` and `loop` have no input parameters. Hence the function definitions end with empty input parameter lists, i.e., `()`.

.

[2]For more information, see `http://www.arduino.cc/en/Reference/FunctionDeclaration`
[3]See, e.g., `http://www.arduino.cc/en/Reference/HomePage`

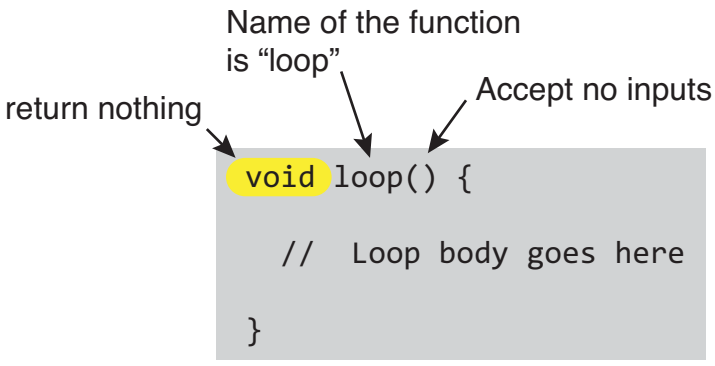

Figure 2: The `loop` function accepts no inputs and returns no values.

2.3 Who calls `loop`?

The `setup` and `loop` functions are ordinary Arduino functions that happen to accept no input parameters and return no values. But that begs two questions. Where would the input parameters come from if there were any? And where would the return value go if the was one? In other words, what (or who) calls `setup` and `loop`?

The Arduino Integrated Development Environment (IDE) performs some hidden (and routine) work that simplifies the writing of code[4]. One of those simplifications is the way that the code written in a sketch is *compiled* and then converted to a program that the AVR microcontroller on the Arduino board can run. When you click the compile button, the IDE first examines the syntax of your sketch to make sure you haven't made any obvious, language-violating errors. It then converts the C code that *you* can read into binary machine code that the *microcontroller* can read and execute. Next, the IDE combines the binary code generated from your sketch with additional binary code to form a single, executable program that is downloaded to the Arduino board. The net effect of this *build process* is that the IDE adds a hidden *master* or *main* program that calls your `setup` and `loop` functions.

The master program is hidden in the sense that you do not have to explicitly add it to your sketch or make any adjustments to it based on the features in your sketch. The hidden program expects to find the `setup` and `loop` functions defined in the appropriate way, i.e., as returning `void` and expecting no inputs.

The interaction of the master program with the `setup` and `loop` functions is depicted in Figure 3. The existence of the hidden master program helps to explain why the `loop` function is called "`loop`". The designers of the Arduino IDE could have called it `main_event_loop` or `user_function_that_repeats`. Instead they simply called it `loop`, a name that is descriptive of its operation, and that also reflects the truth that the `loop` function is called by a loop in the hidden master program.

The `loop` function is like the heartbeat of your program: it repeats indefinitely. During each

[4]See `http://arduino.cc/en/Hacking/BuildProcess`

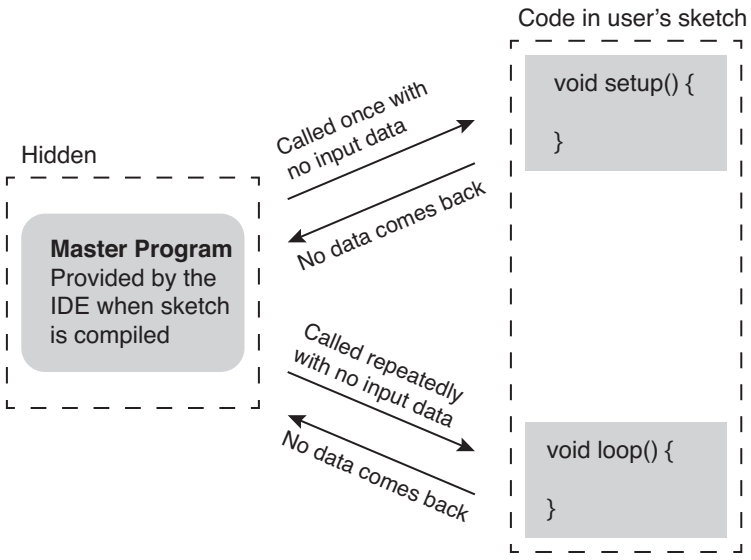

Figure 3: The hidden *master* function calls `setup` once and repeatedly calls `loop`.

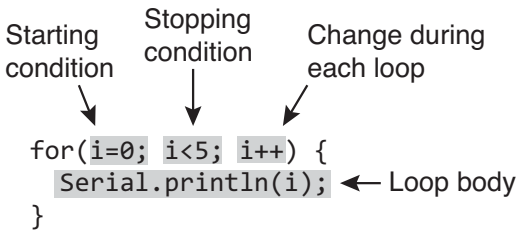

Figure 4: Components of the counter specification in a `for` loop.

heartbeat all the interesting work of your program occurs.

3 `for` and `while` Loops

Since the `loop` function is just an ordinary Arduino function, it can contain all the normal parts of the Arduino programming language, including other loops! In this section we describe the syntax of the two kinds of loop constructs: `for` loops and `while` loops. Your Arduino code can have as many (or as few) loops as you need. Just remember that the `loop` function is itself inside an invisible loop that is executing inside the invisible master program.

3.1 `for` loop

A `for` loop is an iteration structure that is most often used when there is a known number of repetitions to be performed. For example, to add up a list of numbers when the length of the list is known, use a `for` loop. The generic structure of a for loop is

```
for ( Starting condition; Stopping condition; increment/decrement ) {
    Loop body
}
```

The *`Starting condition`*, *`Stopping condition`* and *`increment/decrement`* rule are expressions the control the changes to the *loop counter* on each pass through the loop. A loop counter, which can be an Arduino variable. Often, the loop counter is an integer variable, and often it is a single letter label like `i`, `j`, or `k`.

Figure 4 shows a `for` loop that prints out the first five integers, beginning with zero.

The starting condition is an assignment of an initial value to a loop counter. The stopping condition is a logical statement that involves a comparison. The increment/decrement rule is the formula for changing the loop counter on each pass through the loop. Table 1 gives some examples of common combinations of starting conditions, stopping conditions, and increment/decrement rules.

Example: Compute $\sum_{i=1}^{10} i$

```
int i, sum;

sum = 0;
for ( i=1; i<=10; i++) {
  sum = sum + i;
}
```

Table 1: Examples of rules for counters in a `for` loop. There are many possible different combinations of starting conditions, stopping conditions, and increment/decrement rules. The expressions in this table are common patterns.

Starting condition	Stopping condition	Increment decrement rule	Description
`i = 0;`	`i < 5;`	`i++`	Start with `i=0`. Continue as long as `i` is less than 5. Increment `i` by one at the end of each pass through the loop.
`i = 1;`	`i <= n;`	`i+=1`	Start with `i=1`. Continue as long as `i` is less than or equal to the value stored in the variable `n`. Increment `i` by one at the end of each pass through the loop.
`i = 10;`	`i > 0;`	`i--`	Start with `i=10`. Continue as long as `i` is greater than 0. Decrement `i` by one at the end of each pass through the loop.
`i = 0;`	`i < 8;`	`i+=2`	Start with `i=0`. Continue as long as `i` is less than 8. Increment `i` by two at the end of each pass through the loop.

3.2 `while` loop

A `while` loop is an iteration structure that is most often used when there is a unknown number of repetitions to be performed. A `while` loop may have, but does not need, a loop counter. A `while` loop is a natural way to repeat an operation until a condition is met, when that condition is not directly or obviously dependent on the number of repetitions of the loop. The generic structure of a while loop is

```
while ( Continuation condition  ) {
    Loop body
}
```

Figure 5 shows an annotated `while` loop that waits for a random number larger than 5 to be generated by the built-in `random` function. Note that this loop does not have a specified number of iterations to be executed. Because the sequence of numbers returned by a sequence of calls to `random` is unpredictable, the number of repetitions of the loop will vary as the code is repeated.

The behavior of a `while` loop is controlled by the continuation condition. Typically the continuation condition is a simple logical expression. It would seem, therefore, that the `while` loop does

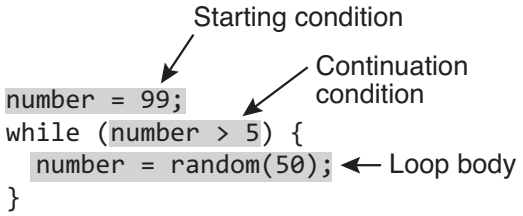

Figure 5: Components of the counter specification of a `while` loop.

its job with less need for code than a corresponding `for` loop. In fact, there is no real code savings in using a `while` loop because important statements that affect the continuation condition are not inside the parenthesis immediately following the `while` statement. The code in Figure 5 shows that the `while` loop has start and stop conditions analogous to the start/stop conditions of a `form`.

`while` loops can be used instead of `for` loops with an appropriate change of the continuation criteria. For example, the following `while` loop sums the numbers from 1 to 10, just as the `for` loop in the preceding example.

Example: Compute $\sum_{i=1}^{10} i$

```
int i, sum;

i = 0;
sum = 0;
while ( i<=10 ) {
  i = i + 1;
  sum = sum + i;
}
```

The preceding computation is more naturally expressed with a `for` loop.

As with any programming feature, it is possible to make errors in logic that produce intended effects. For example, consider two code snippets that show what happens when important statements are not included.

```
int i, sum;

sum = 0;
while ( i<=10 ) {
  sum = sum + i;
}
```

```
int i, sum;

sum = 0;
while ( i<=10 ) {
  i = i + 1;
  sum = sum + i;
}
```

The code on the left is an infinite loop – once started it never terminates because the value of `i` is never changed. The loop on the right is unreliable because the value of `i` is not specified before the loop begins. There is no guarantee that `i` = 0 at the start of the loop.

Therefore, proper use of a `while` loop requires the programmer to prepare for the first evaluation of the starting condition – just like the *starting condition* in a `for` loop. The programmer must write the continuation condition – the complement of the *stoppping condition* in a `for` loop. The programmer must provide some mechanism for changing the outcome of the continuation condition – just like the *increment/decrement* part of a `for` loop. Figure 5 shows how these three basic parts of loop management show up in a simple `while` loop.

Therefore, at a logical level, a `for` loop and a `while` loop are equivalent. At a practical level the choice of a `for` loop or `while` loop is best evaluated according to the programmers intention or reason for writing a loop. Use a `for` loop if you need to iterate a number of times that is known at the start of the loop. Use a `while` loop if you don't know how many iterations are necessary, and especially if the condition for stopping the iterations are determined by some external event, like the pressing of a button.

The example on the next page shows a more natural use for a `while` loop, namely, waiting in the `setup` function for a button to be pushed. That would be useful if you wanted to make your program wait for some condition to be manual established before the `loop` function is to begin.

A `while` loop can be used to wait for user input, as shown in the following example.

Example: Wait for button input

```
void setup() {
  int button_pin = 9;           //  Digital input on pin 9
  int button_pressed = FALSE;   //  Status of the button, initially not pressed
  pinMode( button_pin, INPUT);  //  Configure the input pin

  // Repeat until the user presses a button connected to button_pin
  while ( !button_pressed ) {
     button_pressed = digitalRead( button_pin );
  }
}
```

4 Exercises with the Loop Function

In this section a series of very simple Arduino sketches is discussed. Read the sketch and predict its behavior *before* you read the explanation provided here.

> **Note to instructors:** It would be good for students to figure this code out for themselves without the explanations given below. Each program is short enough that students could enter it manually. However, students would benefit by predicting the outcome after they have read and studied the code, but before they use the Arduino to run the code.
>
> Explanations are given here to offer pedagogical support. It would be best for students to study these codes on their own and before they use the Arduino to show how the code actually works.

Loop 1

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i = 0;
  i = i + 1;
  Serial.println(i);
}
```

What is the output of the code to the left?

This code always prints "1" to the Serial Monitor. That is not likely to be the intent of the code developer. The declaration `int i = 0` resets the value of `i` every time during the `loop`

Loop 2

```
int i = 0;

void setup() {
  Serial.begin(9600);
}

void loop() {
  i = i + 1;
  Serial.println(i);
}
```

What is the output of the code to the left?

This code prints the integers 0, 1, 2, ... and continues until the reset button in pushed or the Arduino is disconnected from its power supply. Each integer is on a separate line because the `println` method of the `Serial` object is used.

If the reset button is pushed, the code begins executing again by running `setup` once, and then calling `loop` indefinitely. This causes the integers to be printed again, starting with 0, 1, 2, etc.

If the power to the Arduino is disconnected, the program stops running. The next time the power is restored, the program resumes as if the reset button had been pushed.

Loop 3

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i;
  for ( i=0; i<5; i++ ) {
    Serial.println(i);
  }
}
```

What is the output of the code to the left?
The integers from 0 through 4 are printed in a repeating patter, i.e., 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4.... Each integer is printed on a separate line.

Loop 4

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i;
  for ( i=0; i<5; i+=2 ) {
    Serial.println(i);
  }
}
```

What is the output of the code to the left?

Loop 5

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i;
  for ( i=0; i<10; i++) {
    i = 5;
    Serial.println(i);
  }
  Serial.println("for loop over\n");
```

What is the output of the code to the left?

This code prints the number 5 indefinitely. Each copy of 5 is on its own line.

This code contains a programming bug: *never* change the loop index inside the *body* of a `for` loop.

Loop 6

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  int i;
  for ( i=0; i<10; i++) {
    Serial.println(i);
    delay(100);
  }
  Serial.println("for loop over\n");
}
```

What is the output of the code to the left?