Functions

Lesson 13

We are now about to learn one of the most useful things about any programming language. We are going to learn about Functions. Learning functions is important because they allow you re-use and organize your code in a very flexible way.

What is a function?

Before we get into the details of how to write functions, lets discuss what they do and how they work.

A function is basically a collection of code that can be triggered to run by a single line of code from anywhere else in your program. The code inside a function can be really simple, or it can be very complex.

Functions are an excellent way to write code that you are likely to use several times in your program.

Let's imagine you are writing code to control a large robot. You may occasionally want to go measure temperatures on all the motors and actuators on the robot to make sure he isn't going to over heat. Doing this on a complex robot may require 10 or 20 lines of code. Now imagine you want to run this code each time you make the motors move. You could write these 10 or 20 lines of code every time you wanted to move the motors, but that wouldn't be very convenient. It would clutter up your code, it would take up more space in the robot's memory, and most important - if you wanted to change how the sensors are read, you may have to change it in many different places in your code and you'll no doubt miss a few. This could cause all kinds of problems.

A much better solution is to write the 10 or 20 lines of code in a single place inside a Function. You could then simply "call" this function anywhere else in your code where you wanted to measure the motor temperatures. If you ever needed to change how this process worked, you could change it in a single place inside your one single function.

You have actually been using Functions all this time throughout our lessons and didn't even realize it. All those times you used a word or set of words followed by parenthesis, you were actually "calling" a function. For example, motors() is a function, eyesOff() is a function, analogRead() is a function, digitalWrite() is a function, and even loop() is a function.

What you've done up to this point is "calling" functions. That basically means you're telling your program to go off and execute the code in the function, then return back to your main program.

We already know how to call functions, because we've been doing so ever since the very beginning with all the 'eyes' functions. In this lesson we're finally going to learn how to write our own functions to do whatever we want.

Let's look at the actual motors() function on the next page...

You have been using the motors() function to control Wink's motors. Below is the actual code that is run each time you call the motors() function.

```
void motors(int LeftMotorSpeed, int RightMotorSpeed){

  if(LeftMotorSpeed>MOTOR_MAX)
    LeftMotorSpeed=MOTOR_MAX;
  if(LeftMotorSpeed<-MOTOR_MAX)
    LeftMotorSpeed=-MOTOR_MAX;
  if(RightMotorSpeed>MOTOR_MAX)
    RightMotorSpeed=MOTOR_MAX;
  if(RightMotorSpeed<-MOTOR_MAX)
    RightMotorSpeed=-MOTOR_MAX;

  if(LeftMotorSpeed<0){
    digitalWrite(MotorDirection_Left,0);
  }
  else{
    digitalWrite(MotorDirection_Left,1);
  }
  if(RightMotorSpeed<0){
    digitalWrite(MotorDirection_Right,0);
  }
  else{
    digitalWrite(MotorDirection_Right,1);
  }
  analogWrite(MotorDrive_Left,abs(LeftMotorSpeed));
  analogWrite(MotorDrive_Right,abs(RightMotorSpeed));

  presentSpeedRight = RightMotorSpeed;
  presentSpeedLeft = LeftMotorSpeed;
}
```

} Variables passed to the function

If you try to give motors a number that is either larger than or smaller than maximum speed of the motors (255 in our case), this code constrains the speed to between -255 and +255. We could have also used the constrain() function here.

Set motor direction pins. If you passed the function a negative number, the motor directions are set to 0 which is reverse. If you passed the function a positive number, the motor directions are set to 1 which is forward.

} This begins sending the actual drive signal to the motor driver, which delivers power to the motors.

} Did you know you a pair of variables named "presentSpeedRight" and "presentSpeedLeft" will always tell you what speed the motors are running? This is where those values are set.

I bet you never realized all the above code is being run each time you call motors() to change the motor speed. You can find this function written in the WinkHardware tab along the top of your Arduino window if you're interested.

Instead of writing all this code each time you want to change the speed of the motors, we've included it all in a single function named "motors". You can now make all this code run from anywhere in your program by simply using a single line of code like "motors(100,100);"

Passing values to and from a function...

Depending on what your function needs to do, you may need to provide it with some starting values. In other cases you may want the function to provide an answer or result back to you.

Let's consider how the motors functions work. When you call the motors() function, you need to include two values. You need to tell the function which speed to make the left and the right motors move. When you actually call the motors() function, you include a pair of speed numbers with it, like "motors(150,150)" to make both motors run forward at speed 150. If you don't include the numbers inside the parenthesis, the function has no way to know what new speed you want.

Remember from one of our first lessons on using the motors, you can call the function beStill() to make Wink stop moving. The beStill() function does not require any extra values or variables to run. This is because beStill() always sets the motors to speed 0. This is an example of a function that always does the same thing each time it is run, and has no need for any extra information or variables to accomplish the task it was written for.

There are other examples when you may want a function to return some value back to your main program. For example, the function buttonPressed() returns a value. The function returns a 1 if the button is being pressed, and returns a 0 if the button is not being pressed.

Let's look at how a function is written. Let's pretend we're going to write a function named "makeWinkGo", and we want that function to automatically make Wink's motors go at 100 speed. The function is written below with the important parts circled. We'll discuss on the next page.

```
void makeWinkGo(void){
  motors(100,100);
}
```

} This is the name of the function. You can name a function almost anything you want. The naming rules are the same as variables.

```
void makeWinkGo(void){
  motors(100,100);
}
```

} This is the code inside the function. This is what will run when the function is called.

```
void makeWinkGo(void){
  motors(100,100);
}
```

} This is where you specify any values or "arguments" you will be passing to the function when it is called.

```
void makeWinkGo(void){
  motors(100,100);
}
```

} This is where you specify what kind of value the function will RETURN back to your program.

Void...

So what's with the word "void" you keep seeing all over our example code? I'm sure you've been curious about this. Maybe the people who designed the C language should have chosen the word "nothing" instead. I suppose that "nothing" is a longer word to type over and over, so maybe they settled on the word "void" instead. At any rate, the word "void" in your program basically means "nothing".

When you see the word "void" inside the parenthesis of a function, like the makeWinkGo(void) example above, it means "nothing" is passed into the function when it is called. In a similar way, when you see the word "void" at the start of a function, it means that function does not return any values back to your program after it runs.

Parts of a Function...

You see the parts of the function on the previous page. The function begins with the type of data that is returned by the function. If the function doesn't return anything, we put "void" in this spot.

The next part is the name of the function. This is what you will use to "call" the function from other places in your program. Functions have the same naming rules as variables. It is customary to start a function name with a lower case letter. If your function needs muliple words, it is customary to lower case the firs letter, then upper case the first letter of each additional word - the same way variables are commonly named.

The next part is the parenthesis where you will indicate what values will be passed into the function, and in what order those values will be passed into the function. These values are also called "arguments" and sometimes they are called "parameters". It all means pretty much the same thing. If no arguments are to be passed into the function, write "void" inside the parenthesis.

The function then has an opening curley and a closing curley. All the code that will be run by the function goes inside those curley braces. The curley braces for functions work just like the curley braces in control structures like "if", "for", and "while".

A working example...

The best way to learn is by actually doing something, so let's start working with some real examples. We are going to use our makeWinkGo example from above, but we're going to make it more useful. Instead of always making Wink go at a certain speed, we'll make the function able to accept a value from your program, so the speed can be easily changed.

```
void loop(){
  makeWinkGo(150);  //call function makeWinkGo, pass value "150"
}

void makeWinkGo(int goSpeed){    //define function "makeWinkGo"
  motors(goSpeed,goSpeed);       //make both motors go
}
```

} This is how we call our new makeWinkGo function from inside our normal loop().

} This is the actual makeWinkGo function. Sometimes we say this is where a function is "defined".

Wink_Ch13Functions_Ex01

In this example, we see how the new function makeWinkGo is defined. We also see how you can call this function from within your main loop() code.

Open up this example from the Companion Code archive that goes along with these lesson plans to see how the whole thing works together. You can change the value passed into makeWinkGo() to change how fast he will go.

Returning a value...

A function can "return" a value back to your program. Let's write a simple program that adds two numbers together, then returns the result. We're going to introduce a few important ideas in this example, so follow along carefully with the explanation below.

```
int result;       //declare "result" as a global variable

void loop(){
  result = addMe(3,2); //call addMe, store returned value in
                       //the 'result' variable.
  Serial.println(result);  //print to serial monitor
  delay(2000);
}

int addMe(int a, int b){    //define the function "addMe"
  int answer;      //declare local variable "answer"
  answer = a+b;    //add the values passed, store in "answer"
  return answer;   //return back with the value in "answer"
}
```

} Declare 'result' as a "global" variable.

} The value of "answer" below is returned back and stored in "result".

} Define our new "addMe" function
} Declare 'answer' as a "local" variable
} Do the math, put result in "answer"
} Return the value in "answer"

Wink_Ch13Functions_Ex02

First lets look at the function we've created named "addMe". You can see that the function will return an "int" data type. The function is named "addMe", and it accepts two arguments inside the parenthesis. The arguments are separated by commas. You have to tell the function what data types will be passed into the function so it knows how to handle them. In this case the data type for both arguments is "int".

We also have to name each incoming argument something so we can reference the passed values inside the function. In this case, we've named the first argument "a" and the second argument is named "b". Once inside the function we can now reference these just like normal variables.

We'll use the variable "answer" to hold the result of the math. We need to declare it first, which we have done inside the function. The last line of code inside the function is the "return". The "return" statement causes the function to stop running and go back to the main program with whatever value specified in the statement. In this case we return the value in the "answer" variable.

This "returned" value is then stored in the variable "result" in the main program where addMe was first called. Expirement passing different values to addMe to prove it's working correctly.

"Global" and "Local" variables...

We introduced another new concept in this example, which becomes important once you start using functions. The concept is "global" variables and "local" variables.

A "global" variable is any variable that is declared outside of a function. A "local" variable is any variable that is declared inside a function. Let's have another look at the same example to discuss this idea.

```
int result;       //declare "result" as a global variable

void loop(){
  result = addMe(3,2); //call addMe, store returned value in
                       //the 'result' variable.
  Serial.println(result);  //print to serial monitor
  delay(2000);
}

int addMe(int a, int b){    //define the function "addMe"
  int answer;      //declare local variable "answer"
  answer = a+b;    //add the values passed, store in "answer"
  return answer;   //return back with the value in "answer"
}
```

} The variable "result" is a global variable because it is declared outside of any function. It is not inside setup() or loop() or any other function.

} The variable "answer" is a local variable because it is declared inside a function. The variable "answer" can only be used within this function.

Wink_Ch13Functions_Ex02

If a variable is "global", that variable can be referenced from anywhere in any part of your program, as well as any functions you may write. You can read it, or change it from anywhere.

If a variable is "local", that variable can only be referenced from within the function that declared it. In this case, we can use the variable "answer" as much as we want inside the addMe function. But if we then try to use the variable "answer" inside the loop() function, the program will not compile because the loop() function doesn't know that the variable "answer" ever existed.

Why would we do this? It seems rather limiting. Why wouldn't we just declare all variables in the entire program as global variables?

Though you could certainly do this, let's discuss what actually happens in the processor's memory when variables are declared. When a "global" variable is declared, the processor picks a spot in its memory to store the value. That spot is then always reserved for the value of that specific variable. No other part of your program can re-use that same spot in memory to store a different variable because it has been reserved.

But when you declare a "local" variable, at the time the function runs, an spot in memory which is not presently being used for anything is selected to hold the value of this "local" variable. This spot in memory is used and reserved only as long as the function is running. As soon as the function completes, this spot in memory is no longer reserved and another function could then use the same spot in memory to hold its own local variables. This is often called "dynamic" memory.

By using dynamic memory, less memory is used overall. If a function is only running for a brief time, and only needs that space in memory temporarily while it is running, there is no reason we can't let another function eventually use the same location in memory to store its own temporary values.

If we declared everything as global variables, we would waste a lot of space in memory that could be more efficiently used by other functions.

You would normally declare global variables if you want the value to be more permanent and you may want to reference that variable from many different places or functions of your program.

For example, you may have a global variable called presentTemperature. You may write a function that turns on a temperature sensor, reads the sensor, does some math on the reading from the sensor, then puts the final result into the presentTemperature variable. You could then reference the presentTemperature variable from anywhere else in your program. You could use it in "if" statements, you could use it to populate other variables, whatever.

In general, if you are writing a function, and the function needs to store a value, but only temporarily while the function is running, you should declare this variable inside the function itself so it becomes a local variable. This is what we have done with the "answer" variable, because the "answer" variable is not needed any longer after the addMe function completes.

Passed arguments automatically become local variables...

We should also point out that the arguments passed into the function also automatically become their own local variables. In our addMe example, when the function is called, a local variable called "a" and a local variable called "b" are created. These local variables are populated with the first and second values you included in the parenthesis when you called the addMe function from your main program.

Writing a function for barrier detection...

Remember back to lesson 10 where we learned about barrier detection using Wink's IR headlight. In that lesson, we used a process of reading Wink's front ambient sensor with the headlight on, then off, then subtracting the two values. Lesson 10 Example 5 shows this process working. Remember we had to repeat a block of code that was about seven lines long. We had to do this each time we wanted to determine how much light was being reflected from a barrier.

This is an excellent example of where a function would have worked much better. Let's write our own function to accomplish this process. We'll name our new function "getReflectedLight". That pretty well describes what the function will be doing. We don't need to pass any values into the function to make it work, but we do need the function to return a value back to our main program.

Consider how we can improve Lesson 10 Example 5 by turning the process of reading the reflected light into its own function, then calling the function each time we want to read the reflected light, rather than repeating our seven lines of code each time.

This is a good challenge to try on your own. I've posted my example of the improved code on the next page.

Cleaned up barrier detect example. Now using a function to read reflected light.

```
#include “WinkHardware.h”

int reflectedLight;
int baseline,threshold;

void setup(){
  hardwareBegin();
  playStartChirp();

  delay(2000);   //wait 2 seconds
  baseline = getReflectedLight();
  threshold = baseline + 10;
}

void loop(){
 reflectedLight = getReflectedLight();

 if (reflectedLight < threshold)  //threshold from above
  {
    motors(100,100);               //drive forward
  }
  else                             //otherwise...
  {
    motors(0,0);                   //be still
  }
} //end of loop()

int getReflectedLight(void){
 int lightOff, lightOn, diff;      //declare as local variables
 digitalWrite(Headlight, HIGH);    //turn on IR Headlight
 delay(1);                         //delay 1 millisecond
 lightOn = analogRead(AmbientSenseCenter);  //read sensor
 digitalWrite(Headlight, LOW);     //turn off IR Headlight
 delay(1);                         //delay 1 millisecond
 lightOff = analogRead(AmbientSenseCenter);  //read sensor
 diff = lightOn - lightOff;
 return diff;
} //end of getReflectedLight function
```

Wink_Ch13Functions_Ex03

} Declare global variables

} Normal setup functions

} Call getReflectedLight function (below)

} Call getReflectedLight function (below)

} This is our new getReflectedLight function. It doesn’t require any input arguments, so we have “void” in the parenthesis. It does return an int type value, so we specify “int” as the return type in the opening line.

This function will measure, calculate, and return the amount of light reflected from a barrier. It can now be called with a single line of code from anywhere in our program.

This new and improved example does the same thing as Lesson 10 Example 5, except it is now much easier to read and understand.

We've done a few things in the above example to improve Lesson 10 Example 5.

We have added the code required to read and calculate the reflected light into a separate function. Because most of the variables we were using before are now local variables inside the function, I've removed some of the global variables that were in the original example.

I've also changed the order the sensor is now read. This time I'm reading it "on" first, then "off", which makes more sense to me. I've also changed the name of the original centerLightOnly variable to "reflectedLight" because I think that better represents the value we're using it for.

Returning more than one value...

Sooner or later, you are going to want a function to return more than one value. Our new function getReflectedLight measures the light reflected back to only the center ambient sensor. But Wink has three ambient sensors - one in the center facing forward, and one facing to either side. Wouldn't it be nice if we could measure all three sensors at the same time and return all three values?

That would be nice. However, unfortunately, the people who wrote the C language either hadn't thought of this, or they thought they had a good reason to prevent this. The short answer is that you cannot return more than one value from a function. (And for those of you that are more advanced, be it known that you can't return an array either, so that idea is out).

But there is a work around.

The easiest way to return multiple values is to create global variables for each of the values. Then you can assign new values to these global variables inside your function, and after the function completes, the variables will still hold their new values. Let's modify our getReflectedLight function to do this. (Note, the example in the companion code is much easier to read, please check it).

```
int getReflectedLight(void){
  int leftOff, rightOff, centerOff;  //declare as local variables
  int leftOn, rightOn, centerOn;
  digitalWrite(Headlight, HIGH);    //turn on IR Headlight
  delay(1);                         //delay 1 millisecond
  leftOn = analogRead(AmbientSenseLeft);      //read left sensor
  centerOn = analogRead(AmbientSenseCenter); //read center sensor
  rightOn = analogRead(AmbientSenseRight);   //read right sensor
  digitalWrite(Headlight, LOW);     //turn off IR Headlight
  delay(1);                         //delay 1 millisecond
  leftOff = analogRead(AmbientSenseLeft);      //read left sensor
  centerOff = analogRead(AmbientSenseCenter); //read center sensor
  rightOff = analogRead(AmbientSenseRight);   //read right sensor
  leftReflected = leftOn - leftOff;           //calculate result
  rightReflected = rightOn - rightOff;        //calculate result
  centerReflected = centerOn - centerOff;     //calculate result
  return centerReflected;                     //return center value
} //end of getReflectedLight function
```

} Local variables

} Read all sensors with headlight on

} Read all sensors with headlight off

} Do calculations, store results in global variables

} Return centerReflected value

Wink_Ch13Functions_Ex04

Have a look at Wink_Ch13Functions_Ex04 in the companion code archive for an easier time reading the code.

In this new example, the only change necessary to the rest of our code is the global variable declarations at the very top. Because `leftReflected`, `rightReflected`, and `centerReflected` are all global variables now, they need to be declared outside of any functions.

```
int reflectedLight;
int leftReflected,rightReflected,centerReflected;
int baseline,threshold;
```

} New global variables declared at top of program.

You will also notice that even though centerReflected is now a global variable, and we could reference that value from within the rest of our code, we are still going to return centerReflected anyway. Why would we do this?

The main reason is for “backward compatibility”. Because our existing program is already expecting a result from the getReflectedLight function, and we are still calculating this same value, we may as well return it as we have been in the past. This way, any code that already works with the old function will still work with the new function. Any time you modify a function that is already being used somewhere else in your code, it is worth considering how you can keep the new function backward compatible so you don’t accidentally break the rest of your code that was working okay up to that point.

We can call the function and directly assign the return value to a variable like this...

```
reflectedLight = getReflectedLight();
```

} Automatically assign centerReflected value to the reflectedLight variable

We can also call the function by itself. If you call the function by itself, after the function completes running, our global variables for left, right, and center reflected will all be updated to the new values. Like this...

```
getReflectedLight();
reflectedLight = centerReflected;
```

} This code would do the same thing as the above example

So now that we have the left, right, and center reflected values being easily calculated for us by our new getReflectedLight function, can we think of a way to improve Wink’s behavior? Right now he just drives straight until he’s close to a barrier. When he gets close to the barrier, he stops. Instead of stopping, could we make him turn to one side or the other to try and drive around it?

Try that on your own then check out the example on the next page.

```
void loop(){
   reflectedLight = getReflectedLight();

   if (reflectedLight < threshold)         //threshold from above
      {
        motors(100,100);                   //drive forward
      }
      else                                  //otherwise...
      {

        if (leftReflected > rightReflected){
          motors(80,-80);                  //rotate right
        }
        else{
          motors(-80,80);                  //rotate left
        }
        delay(100);     //give robot a chance to rotate a while

      }
} //end of loop()
```

Wink_Ch13Functions_Ex05

Instead of just running motors(0,0); to stop Wink, we have added this new section that rotates Wink based on which side measured more reflection. This should steer him away from barriers.

When you run this example, you may notice Wink always wants to steer a certain direction, even if this is turning toward the barrier. This can be caused if one of the ambient sensors tends to generally read a bit higher than the other. There are slight variences in manufacturing, and the IR headlight may be very slightly angled to one side or the other. If you open this example from the companion code archive, you will see a Serial.print line that is commented out. You can un-comment this line to see the sensor values in your serial monitor window.

Test the readings by placing Wink squarely facing directly into a sheet of white paper or other flat white object in front of his nose. Observe the values in your serial monitor. If you notice one side is generally higher than the other by a certain amount, you can add this amount to the calculation. I noticed on my Wink that the right side always measured about 7 counts higher than the left. So I edited the getReflectedLight function and just subtracted 7 from the line that calculates rightReflected. This resulted in Wink's turning movements being more balanced.

So what else can you do with functions? It's time to fly on your own for a while. Go back over some of the other lessons, or write your own behavior starting with just the BaseSketch. See if you can do something fun using functions. Now that you see how they work you'll find it's much easier to write code that does more interesting things.

A few more notes about Functions...

You could learn more about how functions work for days, but we've covered the basics here in this lesson. There are a few more things I'll point out before we finish up here.

Nesting Functions...

One function can call another function, and that function can call yet another function, and so on. This is sometimes called "nesting functions". For example, you may call a function that measures reflected light, and if the reflected light is above a certain limit, you may stop the motors by calling beStill(), which then calls motors(0,0). So your main program has paused and gone off to run your new reflected light function, which then pauses to go off and run beStill(), which then pauses to go off and run the motors() function.

Once motors() completes, it will go back to where it left off in beStill(). Once beStill() is complete, it will go back to where it left off in your reflected light function. Once that reflected light function is complete, it will finally go back to where it left off in your main program.

There is however a limit to how many layers deep your program can go. Each time a function is paused to go off and start another function, the processor needs to remember where it left off in the previous function before starting the next one. Almost like leaving a bookmark of where it left off.

This stack of bookmarks is often called "the stack" by advanced programmers. If you put too many of these bookmarks in the stack, then the stack will overflow. (Called, as you would guess, a "stack overflow") which is where that term comes from.

The limits of this stack can change depending on some other background tasks being run by Arduino. Generally, if you keep the number of nested functions to six or less, you should never have to worry about a stack overflow.

The stack limits of the processor used in Wink are a bit over my own head, so if you do understand this process better than I do, please contact me through the plumgeek.com website. I'd like to update this section if someone can help me better understand the exact limits involved.

Functions with lots of arguments...

In our examples we've only passed two arguments to our functions. But you can pass as many arguments as necessary. It is good practice to not pass more arguments than necessary to accomplish the task so the function is easier to use. Sometimes you may have a valid reason to need to pass six or even ten arguments. Keep in mind the arguments are passed in using the same order that you included them in your parenthesis.

When you write your function, if the first argument is "int brightness", then the local variable "brightness" will be populated by the first value you include in the parenthesis when calling the function. Same with the second, third, and tenth argument.

To understand what arguments a function needs, have a look at the function itself and study the order and types of values the function is expecting. If it was written well, the names given to the local variables in the function definition will help you figure out what the values do.

Really Simple Functions...

Functions don't have to be complicated. Sometimes they can be used as simple shortcuts to shorten up longer lines of code. For example, when I am using my serial monitor to debug code, I often use the line "Serial.print("\t");" to create tabs between lines of output data. I always have the hardest time doing the quote-backslash-t-quote combination. It would be much eaiser for me to just type "tab();" Why not make a function for that?

```
void tab(void){
  Serial.print("\t");
}
```

} Simple shortcut function to make a tab appear in your serial monitor window.

You could do a similar thing to read a light sensor, because the analogRead along with the sensor name is rather long, you can make it super short and simple like this...

```
int slft(void){
  return analogRead(AmbientSenseLeft);
}
int srght(void){
  return analogRead(AmbientSenseRight);
}
int sctr(void){
  return analogRead(AmbientSenseCenter);
}

void loop(){
 leftSensor = slft();
}
```

} Simple shortcut to use instead of typing out the longer analogRead statement every time.

} Just use slft(); in place of the longer statement above.

One caution about using these super simplified functions, is that if you get them too short, they can start to make your code less readable. If I was reading your code and found a call to "sctr()", it probably wouldn't be obviuos to me what that did. And when you re-read it yourself eight months from now, it won't be obvious to you either.

"Wrapper" functions...

A "wrapper" is a function that basically just "wraps around" another function. It's an easy way to call a function by a different name. Maybe you created a function called "remoteControlBasicHandler" and it is now used all over your code within lots of other functions. What if you realize you should re-name it to something more simple like "irHandler"? If you don't want to go re-name everything, you could write a wrapper. Like this...

```
void irHandler(int carrierFrequency, int dataRate){
  remoteControlBasicHandler(carrierFrequency,dataRate);
}
```

} Wrapper function example

Using the Arduino tabs to store functions...

Notice there are tabs across the top of the Arduino window such as FunStuff and WinkHardware. There is also a copy of each tab with a ".h" behind it. Ignore that .h tab for now, but do have a look at the tab that does not have the .h in the name. These tabs are separate files that are stored in the same folder as your Arduino sketch. The code in these tabs is automatically included when your sketch is compiled.

We suggest using the FunStuff tab to write your own functions, especially functions that may be of general use for your programming with Wink. This would be a good place to put your shortcut functions like tab() or slft() from the example above. If you create a function you use often to read Wink's sensors, you can put that here also. Then when you write other examples, you can just copy and paste the function into your new project as needed.

You should also note that when Arduino compiles your code into machine language for Wink's brain, it will only compile code inside functions you have actually called in your program. For example, if you have a long function that measures all the sensors, creates averages, and solves some complicated movement problem, that code will only be loaded onto Wink if you actually call that function somewhere in your program.

For this reason, you don't waste any memory or code space inside the robot by leaving behind many different functions that aren't actually being called. Sometimes it is useful to have all these functions sitting in one of your tabs just in case they are needed, but know you won't waste space if a function isn't needed. This way you can have tab() sitting there all the time if you ever want to use it, but you won't waste space inside the robot if you happen to not use it in a given program.

As long as you're looking at the tabs... spend some time looking over WinkHardware. This is where we've stored all of Wink's background functions. Have a look at how they are written. The code does look a bit complicated but that's because it's more advanced. It'll be more readable once you learn some more. This is a good example of how we can hide away some of the more complicated looking code and make it all run by simply calling the function and letting the function perform its task.