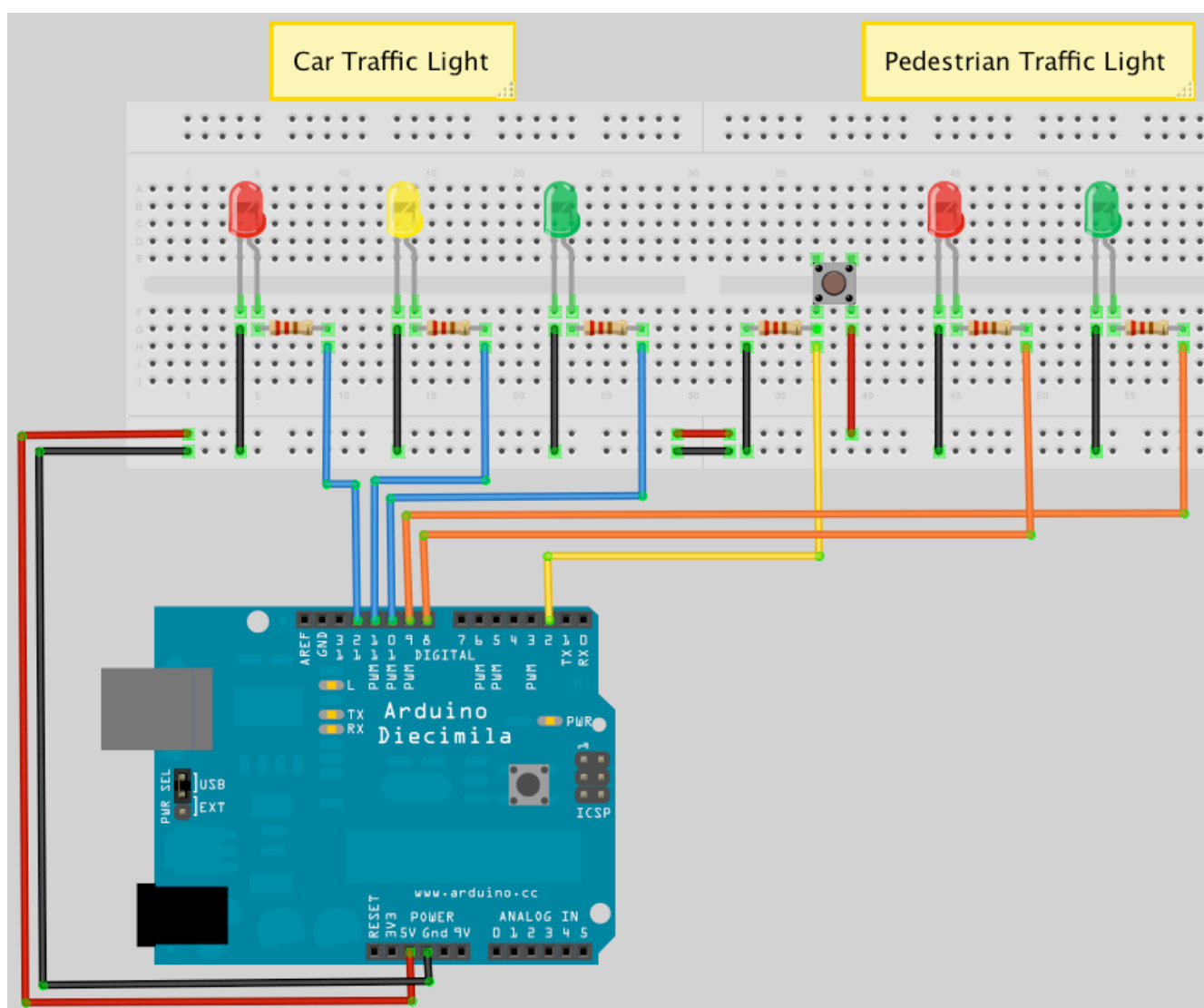


Project 4

Interactive Traffic Lights

Project 4 – Interactive Traffic Lights








This time we are going to extend the previous project to include a set of pedestrian lights and a pedestrian push button to request to cross the road. The Arduino will react when the button is pressed by changing the state of the lights to make the cars stop and allow the pedestrian to cross safely.

For the first time we are able to interact with the Arduino and cause it to do something when we change the state of a button that the Arduino is watching (i.e. Press it to change the state from open to closed). In this project we will also learn how to create our own functions.

From now on when connecting the components we will no longer list the breadboard and jumper wires. Just take it as read that you will always need both of those.

What you will need

| | |
|--------------------------|---|
| 2 x Red Diffused LED's |  |
| Yellow Diffused LED |  |
| 2 x Green Diffused LED's |  |
| 5 x 150Ω Resistors |  |
| Tactile Switch |  |

Connect it up

Connect the LED's and the switch up as in the diagram on the previous page. You will need to shuffle the wires along from pins 8, 9 and 10 in the previous project to pins 10, 11 and 12 to allow you to connect the pedestrian lights to pins 8 and 9.

Enter the code

Enter the code on the next page, verify and upload it.

When you run the program you will see that the car traffic light starts on green to allow cars to pass and the pedestrian light is on red.

When you press the button, the program checks that at least 5 seconds have gone by since the last time the lights were changed (to allow traffic to get moving),

and if so passes code execution to the function we have created called `changeLights()`. In this function the car lights go from green to amber then red, then the pedestrian lights go green. After a period of time set in the variable `crossTime` (time enough to allow the pedestrians to cross) the green pedestrian light will flash on and off as a warning to the pedestrians to get a hurry on as the lights are about to change back to red. Then the pedestrian light changes back to red and the vehicle lights go from red to amber to green and the traffic can resume.

The code in this project is similar to the previous project. However, there are a few new statements and concepts that have been introduced so let's take a look at those.

```
// Project 4 - Interactive Traffic Lights

int carRed = 12; // assign the car lights
int carYellow = 11;
int carGreen = 10;
int pedRed = 9; // assign the pedestrian lights
int pedGreen = 8;
int button = 2; // button pin
int crossTime = 5000; // time allowed to cross
unsigned long changeTime; // time since button pressed

void setup() {
  pinMode(carRed, OUTPUT);
  pinMode(carYellow, OUTPUT);
  pinMode(carGreen, OUTPUT);
  pinMode(pedRed, OUTPUT);
  pinMode(pedGreen, OUTPUT);
  pinMode(button, INPUT); // button on pin 2
  // turn on the green light
  digitalWrite(carGreen, HIGH);
  digitalWrite(pedRed, HIGH);
}

void loop() {
  int state = digitalRead(button);
  /* check if button is pressed and it is
  over 5 seconds since last button press */
  if (state == HIGH && (millis() - changeTime) > 5000) {
    // Call the function to change the lights
    changeLights();
  }
}

void changeLights() {
  digitalWrite(carGreen, LOW); // green off
  digitalWrite(carYellow, HIGH); // yellow on
  delay(2000); // wait 2 seconds

  digitalWrite(carYellow, LOW); // yellow off
  digitalWrite(carRed, HIGH); // red on
  delay(1000); // wait 1 second till its safe

  digitalWrite(pedRed, LOW); // ped red off
  digitalWrite(pedGreen, HIGH); // ped green on
  delay(crossTime); // wait for preset time period

  // flash the ped green
  for (int x=0; x<10; x++) {
    digitalWrite(pedGreen, HIGH);
    delay(250);
    digitalWrite(pedGreen, LOW);
    delay(250);
  }
  // turn ped red on
  digitalWrite(pedRed, HIGH);
  delay(500);

  digitalWrite(carYellow, HIGH); // yellow on
  digitalWrite(carRed, LOW); // red off
  delay(1000);
  digitalWrite(carGreen, HIGH);
  digitalWrite(carYellow, LOW); // yellow off

  // record the time since last change of lights
  changeTime = millis();
  // then return to the main program loop
}
```

Project 4 – Code Overview

Most of the code in this project you will understand and recognise from previous projects. However, let us take a look at a few new keywords and concepts that have been introduced in this sketch.

```
unsigned long changeTime;
```

Here we have a new data type for a variable. Previously we have created integer data types, which can store a number between -32,768 and 32,767. This time we have created a data type of long, which can store a number from -2,147,483,648 to 2,147,483,647. However, we have specified an unsigned long, which means the variable cannot store negative numbers, which gives us a range from 0 to 4,294,967,295. If we were to use an integer to store the length of time since the last change of lights, we would only get a maximum time of 32 seconds before the integer variable reached a number higher than it could store.

As a pedestrian crossing is unlikely to be used every 32 seconds we don't want our program crashing due to our variable 'overflowing' when it tries to store a number too high for the variable data type. That is why we use an unsigned long data type as we now get a huge length of time in between button presses.

4294967295 * 1ms = 4294967 seconds
4294967 seconds = 71582 minutes
71582 minutes - 1193 hours
1193 hours - 49 days

As it is pretty inevitable that a pedestrian crossing will get it's button pressed at least once in 49 days we shouldn't have a problem with this data type.

You may well ask why we don't just have one data type that can store huge numbers all the time and be done with it. Well, the reason we don't do that is because variables take up space in memory and the larger the number the more memory is used up for storing variables. On your home PC or laptop you won't have to worry about that much at all, but on a small microcontroller like the Atmega328 that the Arduino uses it is essential that we use only the smallest variable data type necessary for our purpose.

There are various data types that we can use as our sketches and these are:-

| Data type | RAM | Number Range |
|-------------------------------|------------|---------------------------------|
| void keyword | N/A | N/A |
| boolean | 1 byte | 0 to 1 (True or False) |
| byte | 1 byte | 0 to 255 |
| char | 1 byte | -128 to 127 |
| unsigned char | 1 byte | 0 to 255 |
| int | 2 byte | -32,768 to 32,767 |
| unsigned int | 2 byte | 0 to 65,535 |
| word | 2 byte | 0 to 65,535 |
| long | 4 byte | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 byte | 0 to 4,294,967,295 |
| float | 4 byte | -3.4028235E+38 to 3.4028235E+38 |
| double | 4 byte | -3.4028235E+38 to 3.4028235E+38 |
| string | 1 byte + x | Arrays of chars |
| array | 1 byte + x | Collection of variables |

Each data type uses up a certain amount of memory on the Arduino as you can see on the chart above. Some variables use only 1 byte of memory and others use 4 or more (don't worry about what a byte is for now as we will discuss this later). You can not copy data from one data type to another, e.g. If x was an int and y was a string then x = y would not work as the two data types are different.

The Atmega168 has 1Kb (1000 bytes) and the Atmega328 has 2Kb (2000 bytes) of SRAM. This is not a lot and in large programs with lots of variables you could easily run out of memory if you do not optimise your usage of the correct data types. From the list above we can clearly see that our use of the int data type is wasteful as it uses up 2 bytes and can store a number up to 32,767. As we have used int to store the number of our digital pin, which will only go as high as 13 on our Arduino (and up to 54 on the Arduino Mega), we have used up more memory than was necessary. We could have saved memory by using the byte data type, which can store a number between 0 and 255, which is more than enough to store the number of an I/O pin.

Next we have

```
pinMode(button, INPUT);
```

This tells the Arduino that we want to use Digital Pin 2 (button = 2) as in INPUT. We are going to use pin 2 to listen for button presses so it's mode needs to be set to input.

In the main program loop we check the state of digital pin 2 with this statement:-

```
int state = digitalRead(button);
```

This initialises an integer(yes it's wasteful and we should use a boolean) called 'state' and then sets the value of state to be the value of the digital pin 2. The `digitalRead` statement reads the state of the digital pin within the parenthesis and returns it to the integer we have assigned it to. We can then check the value in state to see if the button has been pressed or not.

```
if (state == HIGH && (millis() - changeTime) > 5000) {
    // Call the function to change the lights
    changeLights();
}
```

The `if` statement is an example of a control structure and it's purpose is to check if a certain condition has been met or not and if so to execute the code within it's code block. For example, if we wanted to turn an LED on if a variable called x rose above the value of 500 we could write

```
if (x>500) {digitalWrite(ledPin, HIGH);
```

When we read a digital pin using the `digitalRead` command, the state of the pin will either be `HIGH` or `LOW`. So the `if` command in our sketch looks like this

```
if (state == HIGH && (millis() - changeTime) > 5000)
```

What we are doing here is checking that two conditions have been met. The first is that the variable called state is high. If the button has been pressed state will be high as we have already set it to be the value read in from digital pin 2. We are also checking that the value of `millis()-changeTime` is greater than 5000 (using the logical AND command `&&`). The `millis()` function is one built into the Arduino language and it returns the number of milliseconds since the Arduino started to run the current program. Our `changeTime` variable will initially hold no value, but after the `changeLights` function has ran we set it at the end of that function to the current `millis()` value.

By subtracting the value in the `changeTime` variable from the current `millis()` value we can check if 5 seconds have passed since `changeTime` was last set. The calculation of `millis()-changeTime` is put inside it's own set of parenthesis to ensure that we compare the value of state and the result of this calculation and not the value of `millis()` on its own.

The symbol '`&&`' in between

```
state == HIGH
```

and the calculation is an example of a Boolean Operator. In this case it means AND. To see what we mean by that, let's take a look at all of the Boolean Operators.

| | |
|-------------------------|-------------|
| <code>&&</code> | Logical AND |
| <code> </code> | Logical OR |
| <code>!</code> | NOT |

These are logic statements and can be used to test various conditions in if statements.

`&&` means true if both operands are true, e.g. :

```
if (x==5 && y==10) {....
```

This `if` statement will run it's code only if x is 5 and also y is 10.

`||` means true if either operand is true, e.g. :

```
if (x==5 || y==10) {.....
```

This will run if x is 5 or if y is 10.

The `!` or NOT statement means true if the operand is false, e.g. :

```
if (!x) {.....
```

Will run if x is false, i.e. equals zero.

You can also 'nest' conditions with parenthesis, for example

```
if (x==5 && (y==10 || z==25)) {.....
```

In this case, the conditions within the parenthesis are processed separately and treated as a single condition and then compared with the second condition. So, if we draw a simple truth table for this statement we can see how it works.

| x | y | z | True/False? |
|---|----|----|-------------|
| 4 | 9 | 25 | FALSE |
| 5 | 10 | 24 | TRUE |
| 7 | 10 | 25 | FALSE |
| 5 | 10 | 25 | TRUE |

The command within the `if` statement is

```
changeLights();
```

and this is an example of a function call. A function is simply a separate code block that has been given a name. However, functions can be passed parameters and/or return data too. In this case we have not passed any data to the function nor have we had the function return any data. We will go into more detail later on about passing parameters and returning data from functions.

When `changeLights();` is called, the code execution jumps from the current line to the function, executes the code within that function and then returns to the point in the code after where the function was called.

So, in this case, if the conditions in the `if` statement are met, then the program executes the code within the function and then returns to the next line after `changeLights();` in the `if` statement.

The code within the function simply changes the vehicles lights to red, via amber, then turns on the green pedestrian light. After a period of time set by the variable `crossTime` the light flashes a few time to warn the pedestrian that his time is about to run out, then the pedestrian light goes red and the vehicle light goes from red to green, via amber and returns to it's normal state.

The main program loop simply checks continuously if the pedestrian button has been pressed or not and if it has, and (&&) the time since the lights were last changed is greater than 5 seconds, it calls the `changeLights()` function again.

In this program there was no benefit from putting the code into it's own function apart from making the code look cleaner. It is only when a function is passed parameters and/or returns data that their true benefits come to light and we will take a look at that later on.

Next, we are going to use a lot more LED's as we make a 'Knight Rider' style LED chase effect.