Projeto 9:

RGB controlados pelo Arduino
Prof. David Mendez Soares

(Beginning Arduino
Copyright © 2010 by Michael McRoberts)

Aula retirada da referencia acima.

Para o projeto 9 voce usará o hardware do projeto 8. Mandaremos comandos, via PC para o ARDUINO usando o Serial Monitor. Aprenderemos a manipular textos (string characters).

```
// Project 10 - Serial controlled mood lamp
char buffer[18];
int red, green, blue;
int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;
void setup()
{
Serial.begin(9600);
Serial.flush();
pinMode(RedPin, OUTPUT);
pinMode(GreenPin, OUTPUT);
pinMode(BluePin, OUTPUT);
}
void loop()
{
if (Serial.available() > 0) {
int index=0;
delay(100); // let the buffer fill up
int numChar = Serial.available();
if (numChar>15) {
numChar=15;
}
while (numChar--) {
buffer[index++] = Serial.read();
}
splitString(buffer);
}
}
void splitString(char* data) {
Serial.print("Data entered: ");
Serial.println(data);
char* parameter;
parameter = strtok (data, " ,");
while (parameter != NULL) {
```



```
setLED(parameter);
parameter = strtok (NULL, " ,");
}
// Clear the text and serial buffers
for (int x=0; x<16; x++) {
buffer[x]='\0';
}
Serial.flush();
}
void setLED(char* data) {
if ((data[0] == 'r') || (data[0] == 'R')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(RedPin, Ans);
Serial.print("Red is set to: ");
```

```
Serial.println(Ans);
}
if ((data[0] == 'g') || (data[0] == 'G')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(GreenPin, Ans);
Serial.print("Green is set to: ");
Serial.println(Ans);
}
if ((data[0] == 'b') || (data[0] == 'B')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(BluePin, Ans);
Serial.print("Blue is set to: ");
Serial.println(Ans);
}
}
```

The input text is designed to accept a lowercase or uppercase R, G, and B and then a value from 0 to 255. Any value over 255 will be dropped down to 255 by default. You can enter a comma or a space between parameters and you can enter one, two, or three LED values at any once; for example:

r255 b100
r127 b127 g127
G255, B0
B127, R0, G255

Code Overview

This project introduces a several new concepts, including serial communication, pointers, and string manipulation. Hold on to your hat; this will take a lot of explaining. First, you set up an array of char (characters) to hold your text string that is 18 characters long, which is longer than the maximum of 16 allowed to ensure that you don't get "buffer overflow" errors. **char buffer[18];**
You then set up the integers to hold the red, green, and blue values as well as the values for the digital pins:

```
int red, green, blue;
int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;
```

In your setup function, you set the three digital pins to be outputs. But, before that, you have the Serial.begin command:

```
void setup()
{
Serial.begin(9600);
Serial.flush();
pinMode(RedPin, OUTPUT);
pinMode(GreenPin, OUTPUT);
pinMode(BluePin, OUTPUT);
}
```

Serial.begin tells the Arduino to start serial communications; the number within the parenthesis (in this case, 9600) sets the baud rate (symbols or pulses per second) at which the serial line will communicate. The Serial.flush command will flush out any characters that happen to be in the serial line so that it is empty and ready for input/output. The serial communications line is simply a way for the Arduino to

communicate with the outside world, which, in this case, is to and from the PC and the Arduino IDE's Serial Monitor. In the main loop, you have an **if** statement

```
if (Serial.available() > 0) {
```

that is using the Serial.available command to check to see if any characters have been sent down the serial line. If any characters have been received, the condition is met and the code within the **if** statements code block is executed:

```
if (Serial.available() > 0) {
int index=0;
delay(100); // let the buffer fill up
int numChar = Serial.available();
if (numChar>15) {
numChar=15;
}
while (numChar--) {
buffer[index++] = Serial.read();
}
splitString(buffer);
}
```

An integer called index is declared and initialized as zero. This integer will hold the position of a pointer to the characters within the char array. You then set a delay of 100. The purpose of this is to ensure that the serial buffer (the place in memory where the received serial data is stored prior to processing) is full before you process the data. If you don't do that, it's possible that the function will execute and start to process the text string before you have received all of the data. The serial communications line is very slow compared to the execution speed of the rest of the code. When you send a string of characters, the Serial.available function will immediately have a value higher than zero and the **if** function will start to execute. If you didn't have the **delay(100)** statement, it could start to execute the code within the **if** statement before all of the text string had been received, and the serial data might only be the first few characters of the line of text entered. After you have waited for 100ms for the serial buffer to fill up with the data sent, you then declare and initialize the numChar integer to be the number of characters within the text string. So, if we sent this text in the Serial Monitor

```
R255, G255, B255
```

the value of numChar would be 17. It is 17, and not 16, because at the end of each line of text there is an invisible character called a NULL character that tells the Arduino when it has reached the end of the line of text. The next **if** statement checks if the value of numChar is greater than 15; if so, it sets it to be 15. This ensures that you don't overflow the array char buffer[18]. Next is a **while** command. This is something you haven't come across before, so let me explain. You have already used the **for** loop, which will loop a set number of times. The **while** statement is also a loop, but one that executes only while a condition is true. The syntax is as follows:

```
while(expression) {
// statement(s)
}
```

In your code, the **while** loop is:

```
while (numChar--) {
buffer[index++] = Serial.read();
}
```

The condition it is checking is numChar. In other words, it is checking that the value stored in the integer numChar is not zero. Note that numChar has -- after it. This is a post-decrement: the value is decremented *after* it is used. If you had used –numChar, the value in numChar would be decremented (have one subtracted from it) before it was

evaluated. In your case, the **while** loop checks the value of numChar and then subtracts 1 from it. If the value of numChar was not zero before the decrement, it then carries out the code within its code block. numChar is set to the length of the text string that you have entered into the Serial Monitor window. So, the code within the **while** loop will execute that many times.The code within the **while** loop is

```
buffer[index++] = Serial.read();
```

and this sets each element of the buffer array to each character read in from the Serial line. In other
words, it fills up the buffer array with the letters you entered into the Serial Monitor's text window. The **Serial.read()** command reads incoming serial data, one byte at a time. So now that your character array has been filled with the characters you entered in the Serial Monitor, the **while** loop will end once numChar reaches zero (i.e. the length of the string). After the **while** loop you have

```
splitString(buffer);
```

which is a call to one of the two functions you created and called **splitString()**. The function looks likethis:

```
void splitString(char* data) {
Serial.print("Data entered: ");
Serial.println(data);
char* parameter;
parameter = strtok (data, " ,");
while (parameter != NULL) {
setLED(parameter);
parameter = strtok (NULL, " ,");
}
// Clear the text and serial buffers
for (int x=0; x<16; x++) {
buffer[x]='\0';
}
Serial.flush();
}
```

The function returns no data, hence its data type has been set to void. You pass the function one parameter, a char data type that you call data. However, in the C and C++ programming languages, you are not allowed to send a character array to a function. You get around that limitation by using a pointer. You know it's a pointer because an asterisk has been added to the variable name *data. Pointers are an advanced subject in C, so I won't go into too much detail about them. If you need to know more, refer to a book on programming in C. All you need to know for now is that by declaring data as a pointer, it becomes a variable that points to another variable. You can either point it to the address at which the variable is stored within memory by using the & symbol, or in your case, to the value stored at that memory address using the * symbol. You have used it to cheat the system, because, as mentioned, you aren't allowed to send a character array to a function. However, you are allowed to send a pointer to a character array to your function. So, you have declared a variable of data type Char and called it data, but the * symbol before it means that it is pointing to the value stored within the buffer variable. When you call **splitString()**, you sent it the contents of buffer (actually a pointer to it, as you saw above):

splitString(buffer);

So you have called the function and passed it the entire contents of the buffer character array. The first command is

Serial.print("Data entered: ");

and this is your way of sending data back from the Arduino to the PC. In this case, the print command sends whatever is within the parentheses to the PC, via the USB cable,

where you can read it in the Serial Monitor window. In this case, you have sent the words “Data entered: ”. Note that text must be enclosed within quotes. The next line is similar

Serial.println(data);

and again you have sent data back to the PC. This time, you send the char variable called data, which is a copy of the contents of the buffer character array that you passed to the function. So, if your text string entered is

R255 G127 B56

then the

Serial.println(data);

command will send that text string back to the PC and print it out in the Serial Monitor window. (Make sure you have enabled the Serial Monitor window first.) This time the print command has ln on the end to make it println. This simply means “print with a linefeed.” When you print using the print command, the cursor (the point at where the next symbol will appear) remains at the end of whatever you printed. When you use the println command, a linefeed command is issued, so the text prints and then the cursor drops down to the next line:

Serial.print("Data entered: ");
Serial.println(data);

So if you look at your two print commands, the first one prints out “Data entered: ” and then the cursor remains at the end of that text. The next print command will print data (which is the contents of the array called buffer) and then issue a linefeed, which drops the cursor down to the next line. If you issue another print or println statement after this, whatever is printed in the Serial Monitor window will appear on the next line. You then create a new char data type called parameter

Char* parameter;

and as you are using this variable to access elements of the data array, it must be the same type, hence the * symbol. You cannot pass data from one data type variable to another; the data must be converted first. This variable is another example of one that has *local scope*. It can be seen only by the code within this function. If you try to access the parameter variable outside of the **splitString()** function, you will get an error.
You then use a strtok command, which is a very useful command for manipulating text strings. Strtok gets its name from String and Token because its purpose is to split a string using tokens. In your case, the token it is looking for is a space or a comma; it’s being used to split text strings into smaller strings. You pass the data array to the strtok command as the first argument and the tokens (enclosed within quotes) as the second argument. Hence

parameter = strtok (data, " ,");

and it splits the string at that point, which is a space or a comma. So, if your text string is

R127 G56 B98

then after this statement the value of parameter will be

R127

because the strtok command splits the string up to the first occurrence of a space or a comma. After you have set the d variable parameter to the part of the text string you want to strip out (i.e. the bit up to the first space or comma), you then enter a while loop with the condition that the parameter is not empty (i.e. you haven’t reached the end of the string):

while (parameter != NULL) {
Within the loop we call our second function:
setLED(parameter);
(We will look at this one in detail later.) Then you set the variable parameter to the next part of the string up to the next space or comma. You do this by passing to strtok a NULL parameter, like so:
parameter = strtok (NULL, " ,");
This tells the strtok command to carry on where it last left off. So this whole part of the function

```
char* parameter;
parameter = strtok (data, " ,");
while (parameter != NULL) {
setLED(parameter);
parameter = strtok (NULL, " ,");
}
```

is simply stripping out each d part of the text string that is separated by spaces or commas and sending that part of the string to the next function called **setLED()**. The final part of this function simply fills the buffer array with NULL character, which is done with the /0 symbol, and then flushes the serial data out of the serial buffer so that it's ready for the next set of data to be entered:

```
// Clear the text and serial buffers
for (int x=0; x<16; x++) {
buffer[x]='\0';
}
Serial.flush();
```

The **setLED()** function is going to take each part of the text string and set the corresponding LED to the color you have chosen. So, if the text string you enter is
G125 B55
the **splitString()**function splits that into the two separate components
G125
B55
and send that shortened text string onto the **setLED()** function, which will read it, decide what LED you have chosen, and set it to the corresponding brightness value. Let's go back to the second function called **setLED()**:

```
void setLED(char* data) {
if ((data[0] == 'r') || (data[0] == 'R')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(RedPin, Ans);
Serial.print("Red is set to: ");
Serial.println(Ans);
}
if ((data[0] == 'g') || (data[0] == 'G')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(GreenPin, Ans);
Serial.print("Green is set to: ");
Serial.println(Ans);
}
if ((data[0] == 'b') || (data[0] == 'B')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(BluePin, Ans);
Serial.print("Blue is set to: ");
Serial.println(Ans);
```

```
}
}
```
This function contains three similar **if** statements, so let's pick one to examine:

```
if ((data[0] == 'r') || (data[0] == 'R')) {
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(RedPin, Ans);
Serial.print("Red is set to: ");
Serial.println(Ans);
}
```

The **if** statement checks that the first character in the string data[0] is either the letter r or R (upper case and lower case characters are totally different as far as C is concerned. You use the logical OR command (the symbol is ||) to check if the letter is an r OR an R, as either will do. If it is an r or an R, the **if** statement knows you wish to change the brightness of the red LED, and the code within executes. First, you declare an integer called Ans (which has scope local to the settled function only) and use the strtol (String to long integer) command to convert the characters after the letter R to an integer. The strtol command takes three parameters: the string you are passing it, a pointer to the character after the integer (which you won't use because you have already stripped the string using the strtok command and hence pass a NULL character), and the base (in your case, it's base 10 because you are using normal decimal numbers as opposed to binary, octal or hexadecimal, which would be base 2, 8 and 16 respectively). In summary, you declare an integer and set it to the value of the text string after the letter R (or the number bit). Next, you use the constrain command to make sure that Ans goes from 0 to 255 and no more. You then carry out an analogWrite command to the red pin and send it the value of Ans. The code then sends out "Red is set to: " followed by the value of Ans back to the Serial Monitor. The other two **if** statements do exactly the same but for the green and blue LEDs.