Microcontroller Systems and Interfacing

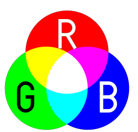
week 9 experimental lab

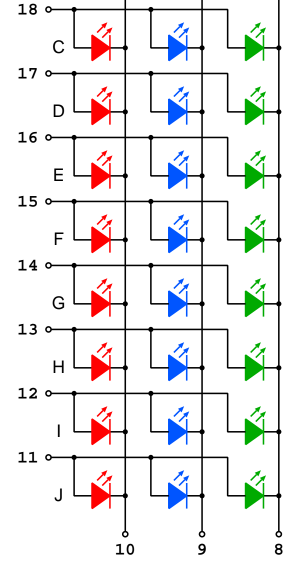
1 RGB LED arrays

We saw last week that LEDs arranged in a line are available packaged as a single 'bar graph' device for displaying information visually. The simplest bar graphs contain ten LEDs, each of which is a single colour (and often all are the same colour). A more versatile LED 'bar graph' uses multi-coloured 'RGB' LEDs.

An RGB LED is a single device containing three LEDs: one red, one green, and one blue (the primary colours). Each LED can be illuminated independently. By selectively turning the LEDs on or off, eight colour combinations are possible: black (all LEDs off), the three primary colours (one of the LEDs on), the three secondary colours (two of the LEDs on), and white (all three LEDs on).

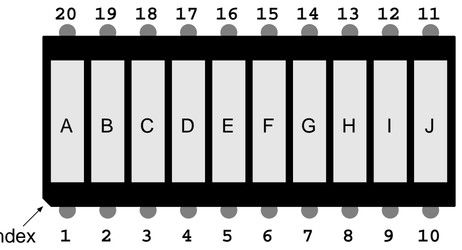
colour:	black	red	green	yellow	blue	magenta	cyan	white
red:	off	on	off	on	off	on	off	on
green:	off	off	on	on	off	off	on	on
blue:	off	off	off	off	on	on	on	on





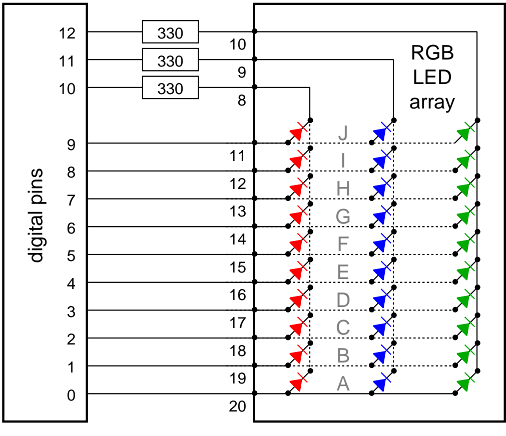
If each LED is independently controlled by PWM then an almost unlimited number of colours can be produced by mixing different levels of each of the three primary colours. (This is exactly how your computer's colour LCD panel works.)

RGB LED arrays contain multiple RGB LEDs in a single package. A typical array contains ten RGB LED segments with ten anode connections and three cathode connections. Each anode connection is common to single segment, and each cathode connection is common to all LEDs of the same colour. Each LED can therefore be illuminated independently by setting the corresponding anode pin positive and cathode pin negative.



For example, in the device we are using, the ten anode pins (11 through 20) can be kept normally LOW and the three cathode pins (1, 2, and 3) can be kept normally HIGH. In that state, none of the LEDs will be illuminated (because all of them will be reverse biased and therefore not conducting). To make the first segment ('A') light up red we can:

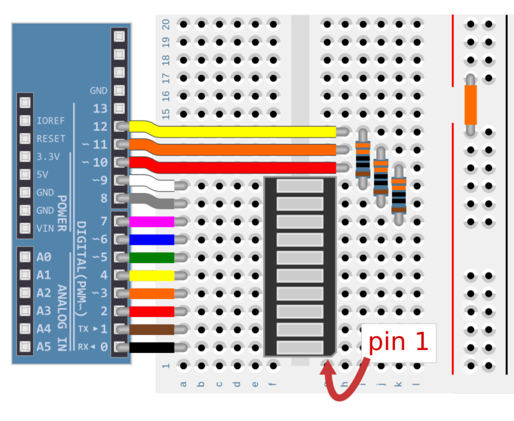
- Set pin 20 of the array HIGH. All LEDs in segment A now have the reverse bias removed, but they are still 'off' because their anodes and cathodes are at the same (positive) voltage. All LEDs in segments B – J remain reverse biased and therefore 'off'.
- Set pin 1 (or pin 10) of the array LOW. All red LEDs now have their cathodes at 0 V. The red LED in segment A is now forward biased and will conduct, turning it 'on'. The red LEDs in segments B – J have their reverse bias removed, but they are still 'off' because their anodes and cathodes are at the same (0 V) voltage.



The single LED at the 'intersection' of pins 1 and 20 (the red LED in segment A) will therefore turn 'on', while all other LEDs will remain 'off'.

Each LED is electrically identical to the other LEDs we have been using. Series current-limiting resistors are therefore necessary. These can be connected either to the anode pins or the cathode pins. (We will connect them to the three cathode pins for two reasons. First, it uses fewer resistors. Second, it lets us choose one of the ten segments and then illuminate any combination of its three LEDs simultaneously at full brightness.)

► Attach the RGB LED array and its three current-limiting resistors to the microcontroller.



The RGB array has a black face and is marked 'OSX10201'. Note that the orientation is different to the LED arrays we used previously (because the anode pins are on the opposite side of the package).

2 Programming the RGB LEDs

Let's configure the output pins and then turn on some of the LEDs.

▶ Configure the output pins.

Digital pins 0 through 12 must be configured as OUTPUTs. You can call pinMode() 13 times if you like, but a while() or for() loop will make the program more compact.

```
void setup(void) {
  for (int i= 0;  i <= 12;  i += 1)
    pinMode(i, OUTPUT);
}
```

▶ Turn on the one of the RGB LEDs.

Setting pin 0 to HIGH for half a second and then back to LOW for half a second will flash the LEDs in segment A of the array.

```
void loop() {
  digitalWrite(0, HIGH);  delay(500);
  digitalWrite(0,  LOW);  delay(500);
}
```

? Why is segment A flashing white?

The anodes of the LED array are being 'driven' by digital pins 0 through 9. Each pin connects to the anodes of the three LEDs (red, green, blue) in one of the array's segments. Setting one of these pins HIGH will place 5 V on all three of those anodes, potentially turning on all three of the LEDs.

The cathodes of the LED array are being 'driven' by digital pins 10 through 12. Each pin connects to the cathodes of ten LEDs, one per segment, that have the same colour. By default an output pin is LOW (0 V) until changed by digitalWrite().

Initially, therefore, setting one of the anode pins (0 through 9) HIGH will turn on all three LEDs in the corresponding segment. (This is because the cathodes of all the LEDs are connected to 0 V via pins 10 through 12, which are all LOW, and the series resistors.) Because of the way the human eye works, mixing red, green, and blue light in equal proportions makes us see what appears to be white light.

▶ Make segment A flash red instead of white.

To make the segment flash red we have to turn off the green and blue LEDs. The cathodes of all green LEDs are connected (through a series resistor) to digital pin 12, and the cathodes of all blue LEDs are connected (through a series resistor) to digital pin 11. Setting these two pins HIGH places 5 V on the cathodes of the green and blue LEDs. Under these conditions those LEDs cannot turn on since it is impossible to make their anodes 2 V more positive than the cathodes.

```
void setup(void) {
  for (int i= 0;  i <= 12;  i += 1)
    pinMode(i, OUTPUT);
  // turn off all blue and green LEDs
  digitalWrite(11, HIGH);
  digitalWrite(12, HIGH);
}
```

▶ Make segment A flash green instead of red.

Hint: which of pins 10 through 12 should *not* be set to HIGH in setup()?

▶ Make segment A flash blue instead of green.

Hint: same as above.

2.1 Accessing every LED independently

Setting all three cathode pins 10, 11, and 12 to HIGH in setup() disables all LEDs. Setting all ten anode pins 0 through 9 to LOW (their default value) also disables all LEDs. From that situation, to turn an LED on requires both a cathode pin to be set to LOW and an anode pin to be set to HIGH. The cathode pin selects which of the three colours is 'enabled' and the anode pin selects which of the ten segments is 'enabled'. The corresponding LED will light up, and all others will remain off.

```
void setup(void) {
  for (int i= 0;  i <= 12;  i += 1)
    pinMode(i, OUTPUT);
  // turn off all red, blue and green LEDs
  digitalWrite(10, HIGH);
  digitalWrite(11, HIGH);
  digitalWrite(12, HIGH);
}
```

A useful way to think of this is like 'X-Y' addressing on a map or similar coordinate system. (The schematic diagram of the LED array shown at the top of the first page is laid out to suggest this kind of 'X-Y' addressing scheme.)

▶ Turn on all three colours, one at a time, in the A segment.

Hint: first select the A segment by setting the corresponding pin HIGH. Any colour LED that is now enabled, by setting a corresponding pin LOW, will immediately turn on in segment A. Select each colour in turn, using a loop, for example.

```
void loop(void) {
  digitalWrite(0, HIGH); // enable A segment
  // enable each colour, in turn
  for (int i= 10;  i <= 12;  i += 1) {
    digitalWrite(i,  LOW);  delay(500);
    digitalWrite(i, HIGH);
  }
  digitalWrite(0, LOW); // disable A segment
}
```

- Create a utility function to control the colour of a segment.

Let's make a function `setLED()` that turns on (or off) any combination of the three colour LEDs for any given segment.

`setLED()` will take two parameters: the segment number (0 through 9) and the colour, specified as three yes/no values for each of red, green, and blue. Similar to `setLEDs()` last week, the colour parameter will be an integer whose rightmost three bits represent the yes/no values for each colour. Red, green, and blue will therefore correspond to colours 1, 2, and 4, and white will be colour 7 (all three primary colours on at the same time).

colour parameter:	0	1	2	3	4	5	6	7
colour produced:	black	red	green	yellow	blue	magenta	cyan	white

To make the function more useful, setting the LED to colour to 0 will disable the segment in addition to turning off all three colours.

```
void setLED(int segment, int colour) {
  if (0 == colour)
    digitalWrite(segment, LOW);   // disable
  else
    digitalWrite(segment, HIGH); // enable

  if (0 == (colour & 1))
    digitalWrite(10, HIGH);       // red on
  else
    digitalWrite(10, LOW);        // red off

  if (0 == (colour & 2))
    digitalWrite(12, HIGH);       // green on
  else
    digitalWrite(12, LOW);        // green off

  if (0 == (colour & 4))
    digitalWrite(11, HIGH);       // blue on
  else
    digitalWrite(11, LOW);        // blue off
}
```

- Test your `setLED()` function.

The following `loop()` code exercises the `setLED()` function. For each of the 8 possible colours in turn, it switches on each each LED in turn from segment A to segment J.

```
void loop(void) {
  for (int colour = 0;  colour < 8;  colour += 1) {
    for (int segment = 0;  segment < 10;  segment += 1) {
      setLED(segment, colour);  // turn on the segment and the indicated colours
      delay(100);
      setLED(segment, 0);       // turn off the segment and all 3 colours
    }
  }
}
```

2.2 Displaying multiple colours

Because of the 'X-Y' addressing scheme for the LEDs, we cannot turn on two segments at the same time with a different colour in each segment. We can, however, exploit persistence of vision to achieve the same effect by turning each combination of 'segment × colour' on and off very rapidly.

- Display three colours on the array, *at the same time*.

Use `setLED()` to turn on the green LED in segments A to F (pins `0` to `5`), yellow (red + green) LED in segments G to I (pins `6` to `8`), and red LED in segment J (pin `9`).

```
void loop(void) {
  for (int segment = 0;  segment < 10;  segment += 1) {
    int colour = 2;  // green
    if (segment >= 6) colour= 3;  // yellow
    if (segment >= 9) colour= 1;  // red
    setLED(segment, colour);      // ON
    delay(1);                     // wait a little
    setLED(segment, 0);           // OFF
  }
}
```

- Make different combinations of colours on the LEDs.

Experiment with the `if()` statements in the previous goal's solution, to move the colours around. For example, a light (or temperature) display might want the first two segments to be blue (for 'cold' or 'dark'), the middle segments green, and the last two segments red (for 'hot' or 'bright').

- Move the LED code out of `loop()` to make way for some application code.

Another useful utility function might update the LEDs to display a value in the range 0 to 10. Maybe `displayValue()` would be a good name for the function. We might communicate the value to be displayed using a variable shared by `displayValue()` and `loop()`, imaginatively called '`value`', for example.

Note the '`return`' statement that is executed when the segment number reaches `value`. This causes the function to finish early, returning to the place in the program where it was used.

```
int value = 8;

void displayValue(void) {
  for (int segment = 0;  segment < 10;  ++segment) {
    if (value <= segment) return;
    int colour = 2;  // green
    if (segment >= 6) colour= 3;  // yellow
    if (segment >= 9) colour= 1;  // red
    setLED(segment, colour);
    delay(1);
    setLED(segment, 0);
  }
}
```

You can test the `displayValue()` function by writing a `loop()` that does nothing but call `displayValue()`.

```
void loop(void) {
  displayValue();
}
```

▶ Add some application code to `loop()` that updates `value`.

This can be anything you like. For example you might read `A0` to get a potentiometer value, or make a self-calibrating light meter, or a temperature meter, etc.

```
int Vmin = 1024, Vmax = 0;
void loop(void) {
  int a0 = analogRead(A0);
  Vmin = min(Vmin, a0);
  Vmax = max(Vmax, a0);
  value = map(a0, Vmin, Vmax, 0, 11);
  displayValue();
}
```

▶ Display a value that rises steadily from 0 to 10, then falls back to 0, in a repeating cycle.

This one should now be super easy for you! It is very similar to the programs that made patterns on the LED bar graph. For extra style, use two loops to update `value`. Pause for 50 ms with each setting of `value` so that the entire cycles takes one second.

```
void loop(void) {
  for (value= 0;  value <= 10;  value += 1) {
    displayValue();
    delay(50);
  }
  for (value= 9;  value >= 1;  value -= 1) {
    displayValue();
    delay(50);
  }
}
```

? Does your LED display look *fantastically cool*?

No? Why not? What's wrong with it? Can you figure out (without reading any more of this document) why it is behaving like that?

2.3 The problem with single-threaded programs

Our program is single-threaded. That means a single 'thread' of control moves through the program, with only one (very predictable) thing executing at any time, roughly like this:

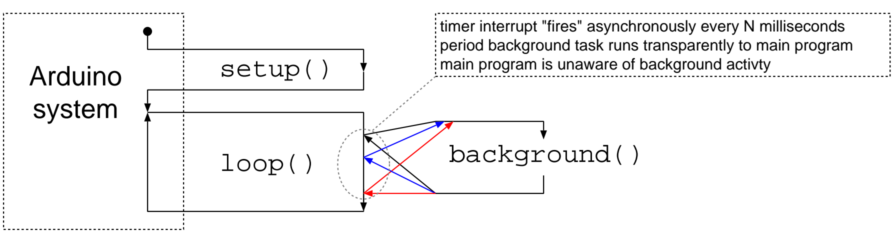
1. Run `setup()`.
2. Run `loop()`, which does the following 20 times:
   (a) Update `value`.
   (b) Call `displayValue()` to flash the LEDs appropriately.
   (c) Wait for 50 ms before continuing. ← During this time the LEDs are all *off*!
3. Go back to step 2.

How to fix this? It would be nice to update the LEDs in the 'background', independently of `loop()`. For example, every few milliseconds we would like to `displayValue()` 'automatically' regardless of what `loop()` is doing (including `delay()`ing). Fortunately we *can* do this, by periodically *interrupting* the program and asking it to `displayValue()`. This can be made completely 'transparent' to the application code in `loop()`. In other words, it can be done without involving `loop()` at all. Our program will behave as if we have two `loop()`s running *at the same time*. One of them will perform the application code, the other will perform the `displayValue()` update of the LEDs.

3 Interrupts

An interrupt is a way of diverting the microcontroller from its normal task and asking it to do something different for a while. Compare it to answering the telephone while you are in the middle of reading a book. Reading the book is the normal activity, the telephone ringing is the *interrupt*, and then answering the telephone and talking with the other person for a while is called *servicing* the interrupt. When interrupt servicing is finished the telephone call ends and you go back to reading your book from the point you were interrupted.

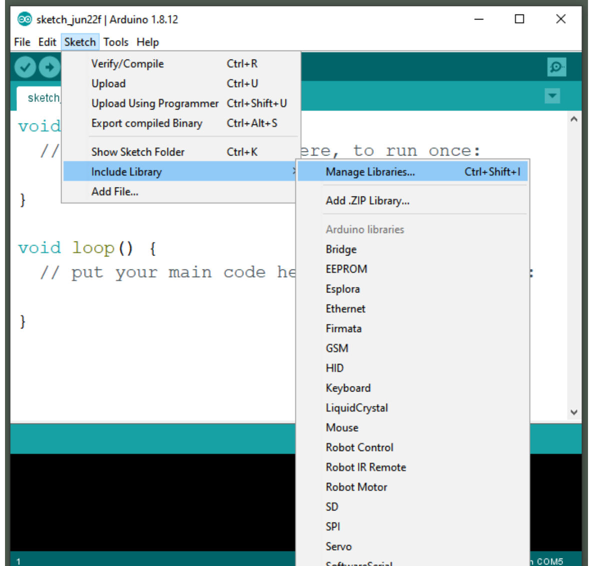
The microcontroller has several sources of interrupts. One source of interrupts are the built-in timers that can generate a *timer interrupt* at regular intervals. That is the kind of interrupt that we need in order to run our second, background '`loop()`' function to update the LEDs.

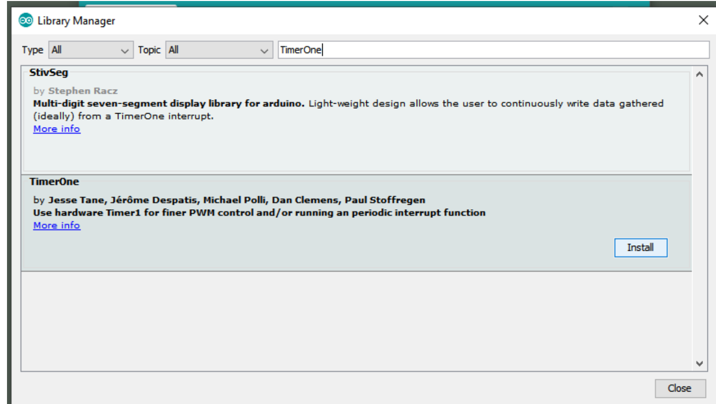


3.1 The `TimerOne` interrupt library

A user-friendly interface to timer interrupts is available using the `TimerOne` library. Before you can use the interface you will have to install the library.

Open the 'Sketch' menu and then select 'Include Library' followed by 'Manage Libraries...'. A new window will open. In the search box, type "TimerOne". The display will update to show the libraries that match. Select 'TimerOne' and then click the 'Install' button that appears. When the progress bar completes, you can close the window.

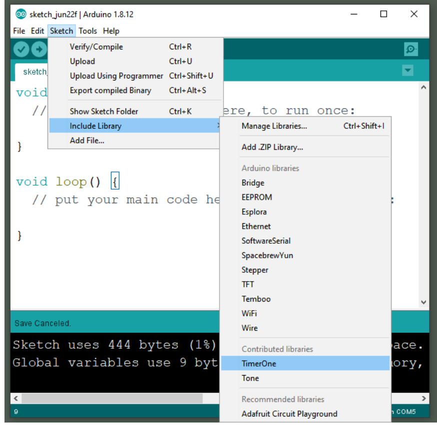


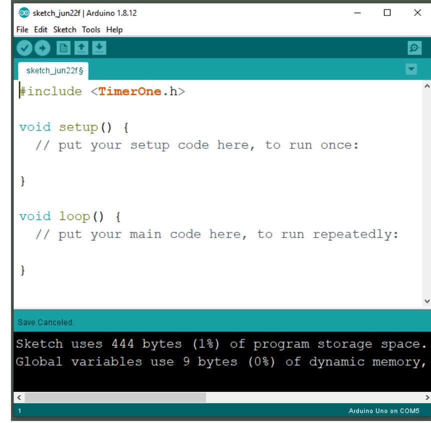


To use the library in your program, choose the 'Sketch' menu again and select 'Include Library', and near the bottom select 'TimerOne'. In the main window you should see the line

```
#include <TimerOne.h>
```

appear at the top of your program. (If you prefer, you can also just type that line into your program; the end result will be identical.)





3.2 Programming with timer interrupts

Timer interrupts are controlled by several functions grouped together inside an object called `Timer1`. To set up a periodic timer, two of these functions must be called in `setup()`. The first, `Timer1.initialize()`, initialises the timer and tells it how often (in microseconds) it should call the 'background' function. The second, `Timer1.attachInterrupt()`, tells the timer which of our functions is the 'background' function.

The program on the right is a minimal example of using the timer interrupt to perform a background task on the RGB LED array. In `setup()` the pins connected to the first two segments and to the red LEDs are configured. All are initially `LOW`, which disables the two segments but enables the red LEDs. (Therefore to turn a segment on we need only set its pin `HIGH`.) The timer is then asked to generate 10 interrupts per second ($100,000\ \mu s$ interval) and to call the function `background()` to service the interrupt.

The `background()` function uses `digitalWrite()` to toggle segment B between 'on' and 'off' states.

The `loop()` function meanwhile toggles segment A every half a second.

The overall result is that segment A flashes once per second and segment B flashes five times per second.

```
#include <TimerOne.h>

void setup() {
  pinMode(0,  OUTPUT);
  pinMode(1,  OUTPUT);
  pinMode(10, OUTPUT);
  Timer1.initialize(100000);
  Timer1.attachInterrupt(background);
}

int value = 0;

void background(void) {
  digitalWrite(1, value);
  value = !value;          // not
}

void loop() {
  digitalWrite(0, HIGH);  delay(500);
  digitalWrite(0,  LOW);  delay(500);
}
```

4 A multi-colour bar-graph display with background updates

The complete colour bar graph program is shown on the right.

Two lines have been added to `setup()` that set up the timer interrupt to call `background()` every 1 ms.

The `background()` function is called 1,000 times per second to update the value displayed on the LEDs. It is almost the identical to the earlier `displayValue()` function. The difference is that the `for()` loop that iterated through all the segment has been removed. Instead the currently active segment is remembered in the variable `segment`. Each time `background()` runs it turns off the active segment, makes the next segment active, and then turns it with the appropriate colour provided `value()` is large enough. That segment then remains lit until the next time `background()` runs, and the process repeats. The result is that `background()` will cycle through all ten segments every second, illuminating only those segments permitted by the current `value`.

One new feature has been used in this program. The variable `value` has been marked '`volatile`'. This tells the compiler that it might be accessed at unpredictable times, for example, when `background` is called automatically every 1 ms. For that reason, the compiler takes extra care when updating `value` in `loop()` to make sure any unpredictable functions such as `background()` 'see' the correct value. A general 'rule of thumb' is that you should mark '`volatile`' any variable that can be accessed from both normal code (like `loop()`) and unpredictable asynchronous code (like `background()`).

```
#include <TimerOne.h>

void setup() {
  for (int pin = 0;  pin <= 12;  pin = pin + 1)
    pinMode(pin, OUTPUT);

  digitalWrite(10, HIGH);
  digitalWrite(11, HIGH);
  digitalWrite(12, HIGH);

  Timer1.attachInterrupt(background);
  Timer1.initialize(1000);
}

void setLED(int segment, int colour) {
  if (0 == colour)
    digitalWrite(segment, LOW);  // disable segment
  else
    digitalWrite(segment, HIGH); // enable segment

  if (0 == (colour & 0b001))
    digitalWrite(10, HIGH);      // red on
  else
    digitalWrite(10, LOW);       // red off

  if (0 == (colour & 0b010))
    digitalWrite(12, HIGH);      // green on
  else
    digitalWrite(12, LOW);       // green off

  if (0 == (colour & 0b100))
    digitalWrite(11, HIGH);      // blue on
  else
    digitalWrite(11, LOW);       // blue off
}

volatile int value = 0;  // volatile => 'unpredictable'
int segment = 0;         // currently active segment

void background(void) {
  setLED(segment, 0);           // turn off active segment
  segment += 1;
  if (segment > 9) segment = 0;
  if (value <= segment) return;
  int colour = 2;  // green
  if (segment >= 6) colour= 3;  // yellow
  if (segment >= 9) colour= 1;  // red
  setLED(segment, colour);      // turn on active segment
}

void loop(void) {
  // value = 0 1 2 ... 8 9 10
  for (value= 0;  value <= 10;  value += 1)  delay(50);
  // value = 9 8 7 ... 3 2 1
  for (value= 9;  value >=  1;  value -= 1)  delay(50);
}
```

► Display the light level (self-calibrated) on the RGB LED array.

This should be just as super-easy for you and, this time, the end result really will look *fantastically cool*!

5 Challenge

The colours (green, yellow, red) of the ten segments are fixed by the `background()` function. Remove this limitation by making a flexible RGB LED system. Here are the steps, and parts, that you might need to achieve that. (If you succeed then you will have a very useful and general facility that you can reuse in many of your own projects whenever a colourful bar graph display is needed.)

5.1 Use an array to specify the segment colours

An *array* is a collection of identical variables that are accessed by a numeric index (instead of giving each one of them a unique name). For example, if you have three integer variables

```
int x, y, z;
```

then you could replace them with a single array called 'a' holding three integers

```
int a[3];     // element-type array-name [ number-of-elements ]
```

and then make the following substitutions in your program: x → a[0], y → a[1], z → a[2]. Your program's behaviour will be *exactly the same*. The advantage of the individual int variables is that they can each be given a meaningful name, which makes your program easier to read and understand (and type, probably). The advantage of the array is that a particular int element of interest within the array a can be specified using another variable; for example, if i is an integer between 0 and 2, then 'a[i]' refers to the ith element of the array a. (This is called *indirect addressing* and is an important and fundamental idea in programming.)

To describe the colours in our ten-segment bar graph we could use a ten-element array of ints.

```
int ledColours[10];
```

Just like a normal variable, an array can be initialised when it is created. Follow the name with an assignment '=' and then a list of initial values between curly braces and separated by commas. Like this:

```
int ledColours[10] = { 2, 2, 2, 2, 2, 2, 3, 3, 3, 1 }; // six greens, three yellows, one red
```

If it is not immediately obvious what is going on here, compare the colour numbers in this array with the calculation of colour in the background function:

```
int colour = 2;               // green   for segments 0, 1, 2, 3, 4, 5
if (segment >= 6) colour= 3;  // yellow  for segments 6, 7, 8
if (segment >= 9) colour= 1;  // red     for segment  9
```

Note also that the indices in the ledColours array correspond exactly to segment numbers. In other words, these three lines in background can be replaced by a single line that fetches the required colour out of the array.

```
int colour = ledColours[segment];
```

► Modify your program as described above and verify that it works correctly.

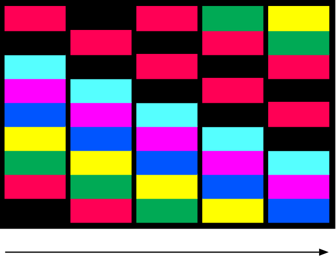
5.2 Marching colours

The following loop sets up all eight possible colours on the ten segments (with black and red repeated, because colours 8 and 9 are the same as colours 0 and 1 when you are only looking at the rightmost three bits).

```
void loop(void) {
  for (int i= 0;  i <= 9;  ++i)
  ledColours[i] = i;  // set the colours of each segment to be the same as its index
}
```

► Modify this loop() so that it displays 'marching colours'.

'Marching colours' are colours that appear to move along the entire length of the bar graph, stepping to the next segment every (say) 500 ms.



5.3 Improve performance

Instead of digitalWrite(), use the PORTD and PORTB registers to control pins 0 to 12.

5.4 [DIFFICULT] Mix the primary colours in different ratios

Use pulse-width modulation (PWM) to adjust the brightness of the red, green, and blue LEDs. For example, with 16 levels (4 bits) of PWM brightness for each individual LED, each segment could display 2048 (12 bits) different colours.