Introduction to Design (2)

Microcontrollers and Interfacing

Week 09

RGB LED arrays
Concurrency and timer interrupts

京都先端科学大学
KYOTO UNIVERSITY of ADVANCED SCIENCE
Department of Mechanical and Electrical System Engineering

this week

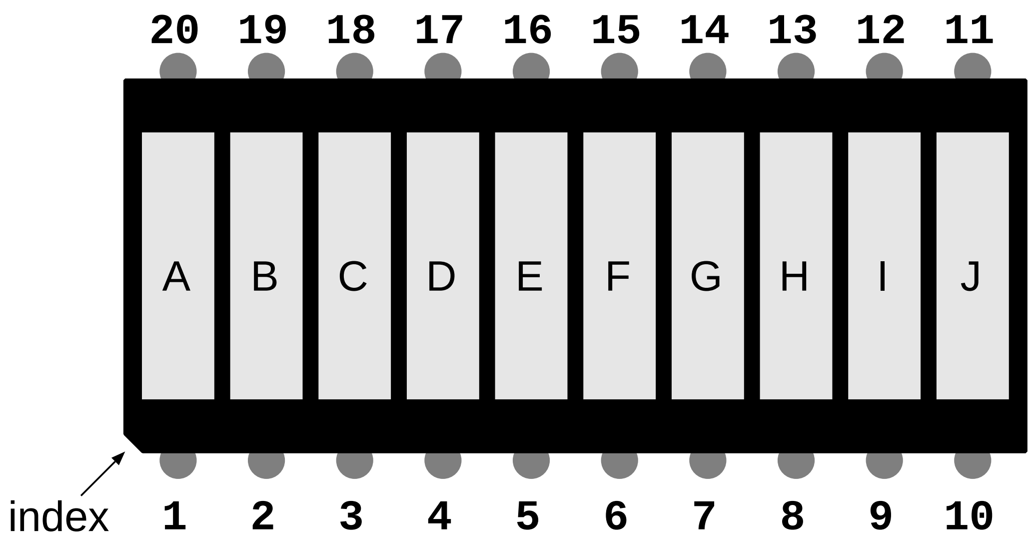
RGB LED arrays

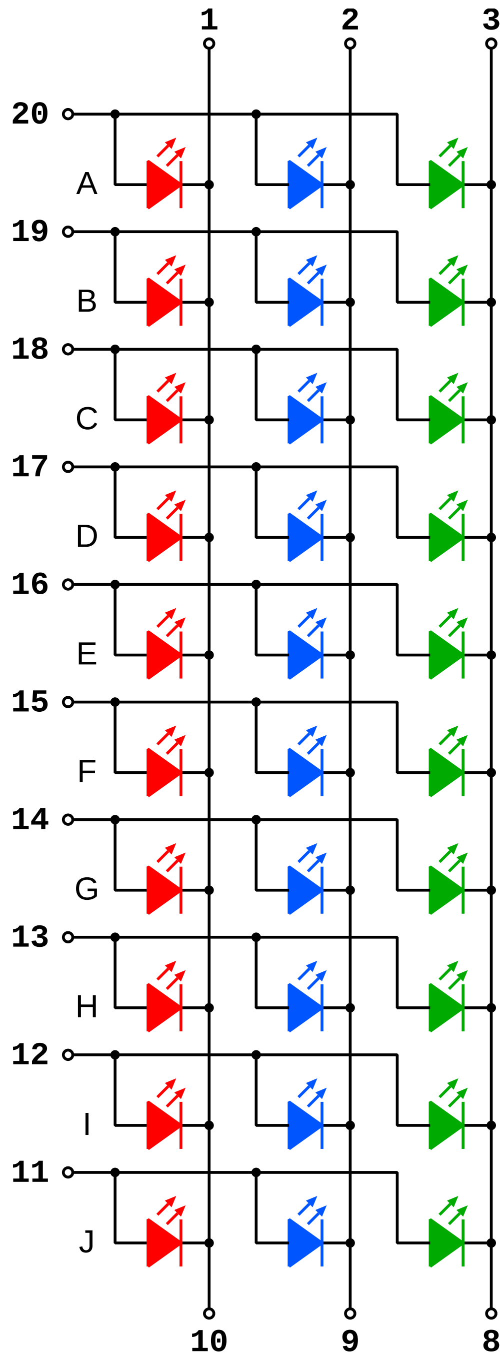
- connecting 30 LEDs using 13 pins
- software techniques
- time-division multiplexing

concurrency

- background tasks
- timer interrupts

RGB LED arrays





LED array with multiple colours

- each segment has red, green, and blue LEDs

anode and cathode pins create X-Y coordinate system

to switch on blue LED of segment F:

- make pin 15 2 V more +ve than pin 9

to ensure no other LEDs on:

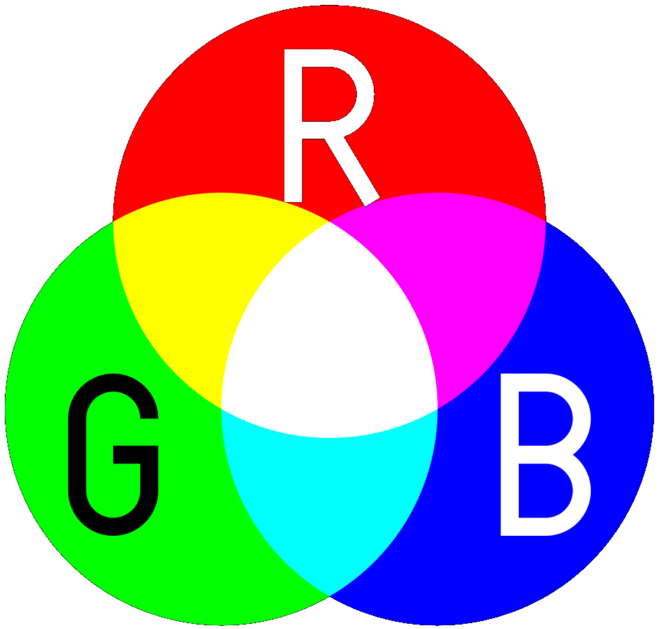
- make other anode pins 0 V (pins 11–14, 16–20)
- make other cathode pins 5 V (pins 8, 10)

why RGB?

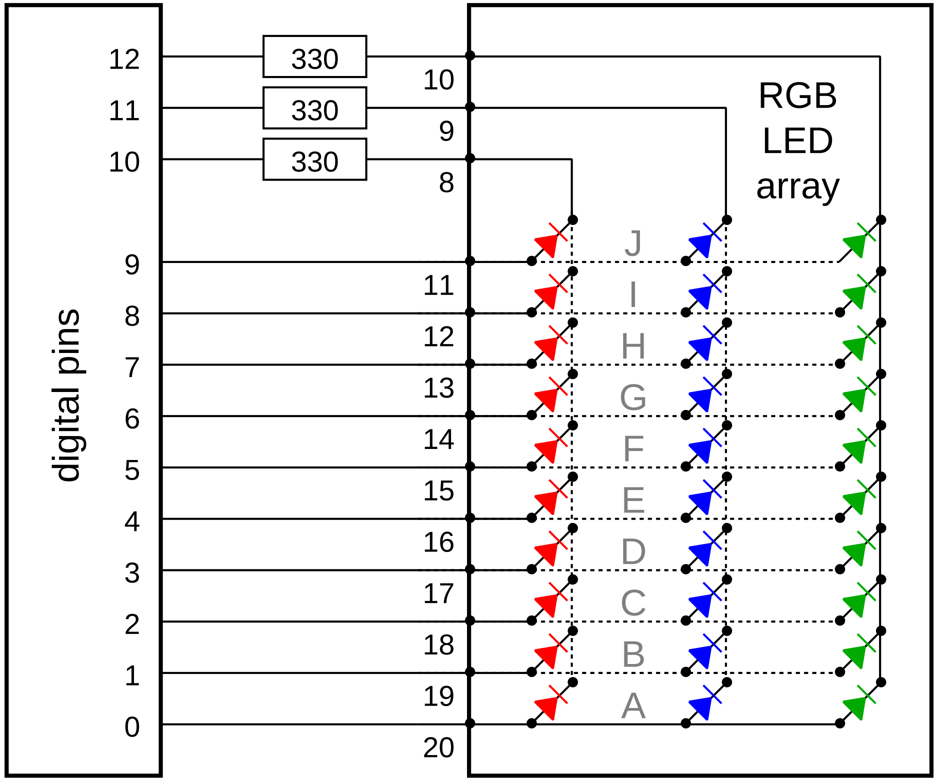
turning on different combinations of R, G, and B yields different colours

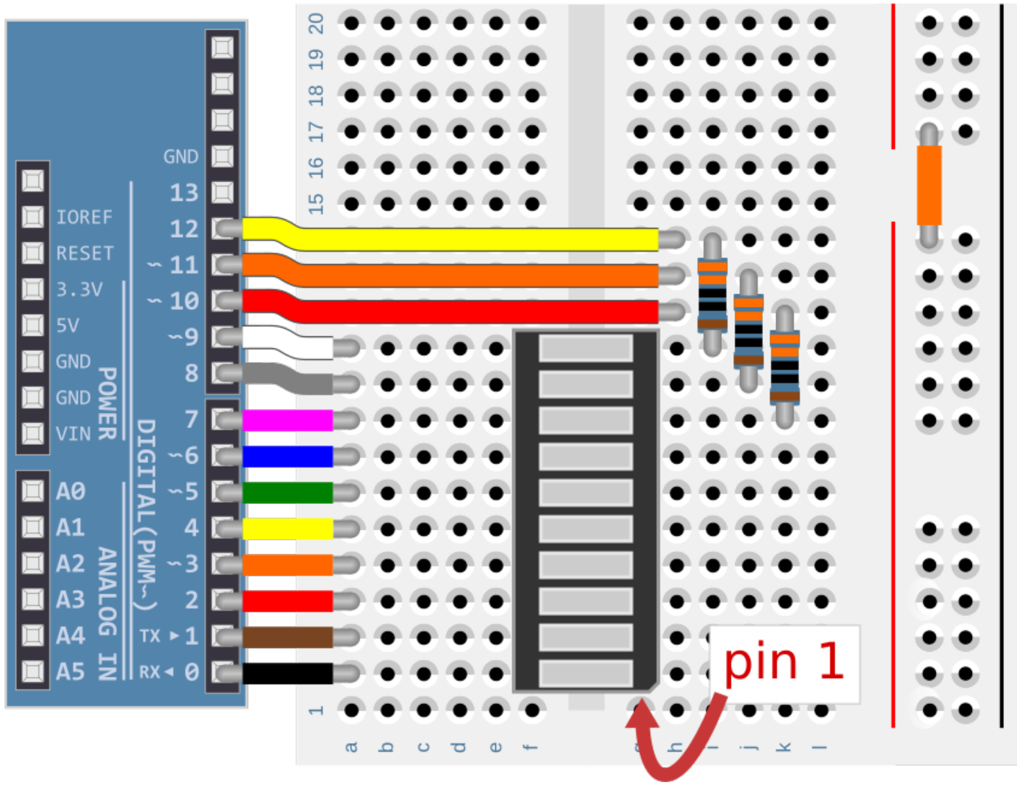
colour:	black	red	green	yellow	blue	magenta	cyan	white
red:	off	on	off	on	off	on	off	on
green:	off	off	on	on	off	off	on	on
blue:	off	off	off	off	on	on	on	on

additive colour synthesis:



interfacing with the RGB array





pins 0 to 9 drive anodes of each segment

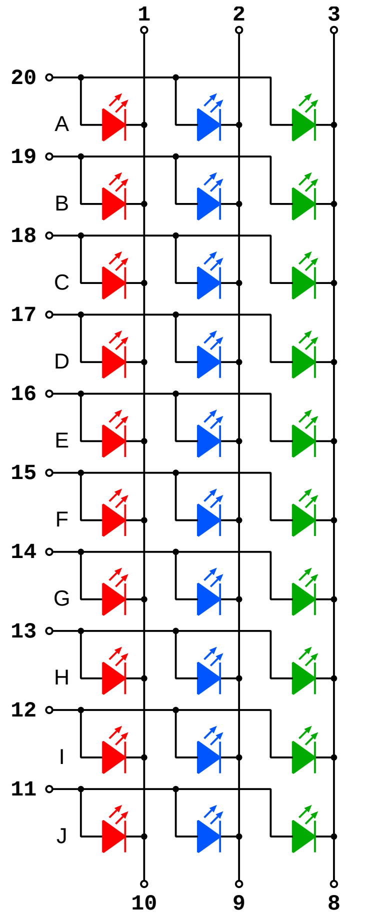
pins 10 to 12 drive cathodes of each colour

3 $\times$ series resistors limit current

- $\approx$ 10 mA per colour
- $\approx$ 30 mA total current with all LEDs lit

configuring LED pins

```
void setup() {
  // SEGMENTS                          disable
  pinMode( 0, OUTPUT);  digitalWrite( 0,  LOW);   // A
  pinMode( 1, OUTPUT);  digitalWrite( 1,  LOW);   // B
  pinMode( 2, OUTPUT);  digitalWrite( 2,  LOW);   // C
  pinMode( 3, OUTPUT);  digitalWrite( 3,  LOW);   // D
  pinMode( 4, OUTPUT);  digitalWrite( 4,  LOW);   // E
  pinMode( 5, OUTPUT);  digitalWrite( 5,  LOW);   // F
  pinMode( 6, OUTPUT);  digitalWrite( 6,  LOW);   // G
  pinMode( 7, OUTPUT);  digitalWrite( 7,  LOW);   // H
  pinMode( 8, OUTPUT);  digitalWrite( 8,  LOW);   // I
  pinMode( 9, OUTPUT);  digitalWrite( 9,  LOW);   // J
  // R/G/B                             disable
  pinMode(10, OUTPUT);  digitalWrite(10, HIGH);   // R
  pinMode(11, OUTPUT);  digitalWrite(11, HIGH);   // B
  pinMode(12, OUTPUT);  digitalWrite(12, HIGH);   // G
}
```



all LEDs are initially *reverse biased*

to turn on a given LED both its anode *and* cathode voltages must be reversed

selecting individual LEDs

```
//                    0...9  0bxxxxxBGR
void setLED(int segment, int colour)
{
  // SEGMENT enabled when pin is HIGH
  if (0 == colour)                      // black
    digitalWrite(segment, LOW);   // => completely disable segment
  else
    digitalWrite(segment, HIGH); // enable segment

  // R/G/B enabled when pin is LOW
  if (0 == (colour & 0b001)) digitalWrite(10, HIGH);   // red off
  else                       digitalWrite(10,  LOW);   // red on

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

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

displaying different colours on different segments

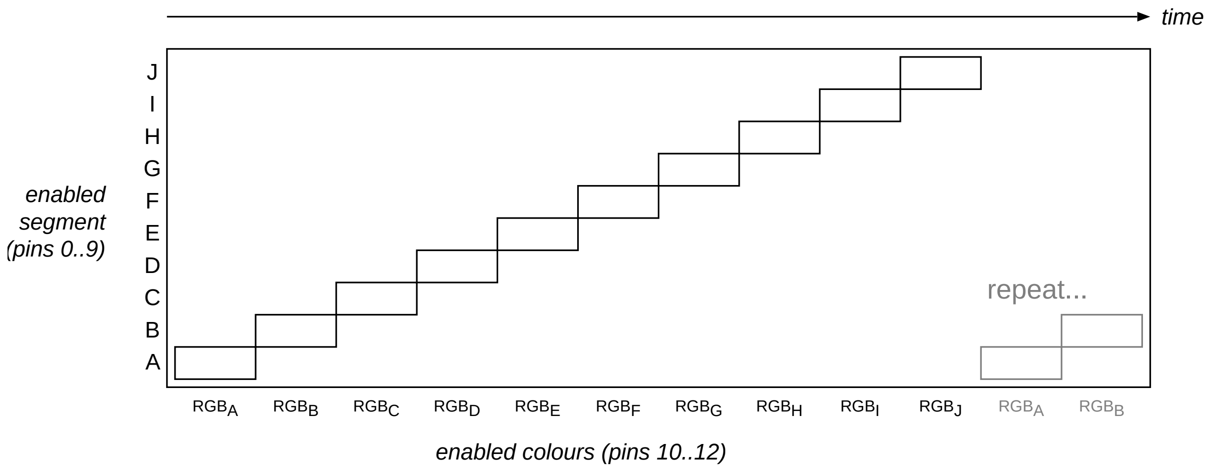
segments and colours ‘interfere’ with each other

- all colours that are enabled will affect all segments that are enabled

to display different colours on different segments

- enable one segment and enable just its colours
- repeat for all segments

allowing for all ten segments to have ten different colours leads to this pattern



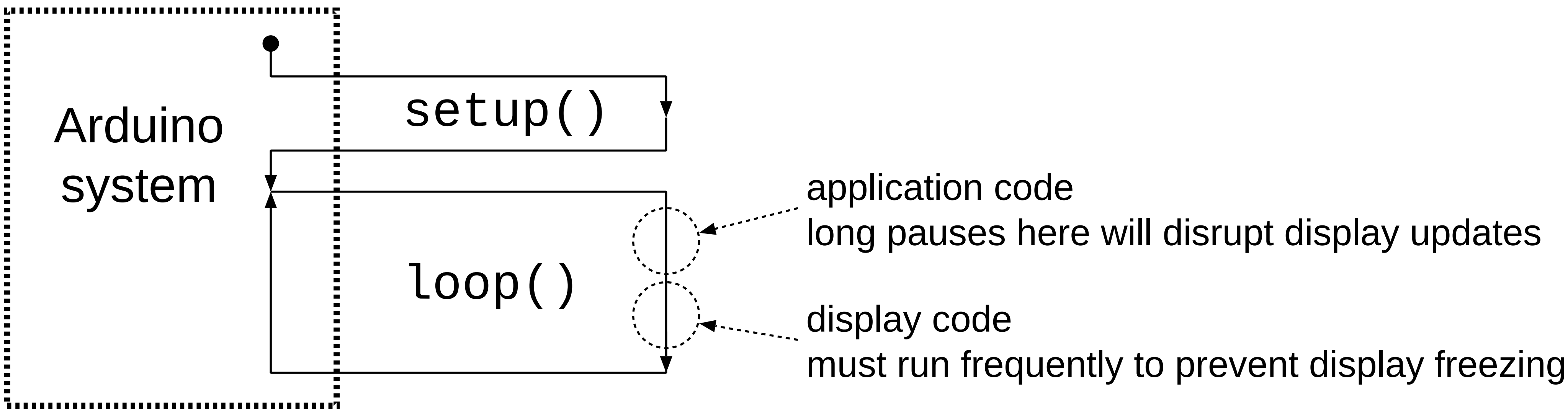
this is a very powerful technique in communication called *time-division multiplexing*

- 10 $\times$ different RGB_x values sent over one connection using 10 divisions of time
- the segment pins (0..9) are a kind of ‘clock’ saying which division is being sent

interrupts

updating segments using time-division multiplexing relies on persistence of vision

- a long pause in the updates will ‘freeze’ the LEDs with one or zero segments lit
- the application code can introduce pauses unpredictably
  - by using `delay()`
  - by waiting for serial communication
  - etc.
- putting application code and LED updates in the same `loop()` is not reliable



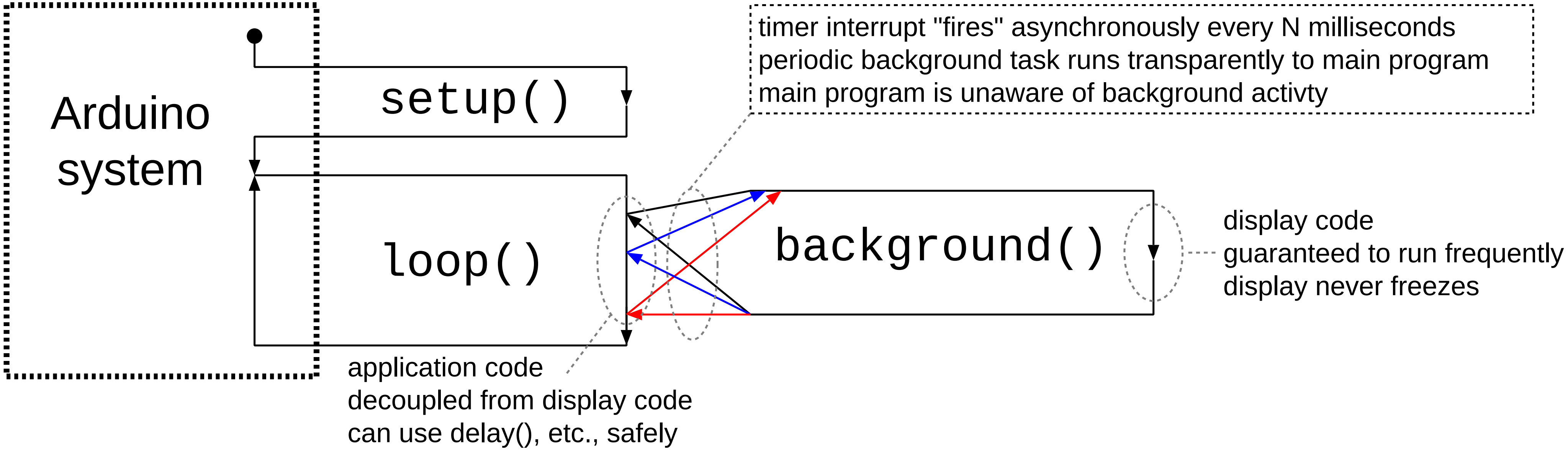
interrupts

an interrupt

- temporarily diverts the microcontroller from its normal task
- a 'background' task can then be performed
- when finished, the main task resumes as if nothing had happened

if the interrupt is attached to a regular timer

- the background task is performed regularly, decoupled from the normal task
- almost as if there were two `loop()`s running concurrently



using timer interrupts

```
#include <TimerOne.h>

void setup(void) {
  Timer1.initialize(period);            // in microseconds
  Timer1.attachInterrupt(handler);     // the name of a function
}

void handler(void) {
  // this function is executed asynchronously
  // every period microseconds
  // to perform background tasks
  // e.g., updating a multiplexed display
}

void loop(void) {
  // application code performs a foreground task
  // and can be written as if
  // the background task did not exist
}
```