Action movie bomb

Building an action movie like detonator circuit countdown timer.

Enable native scrolling

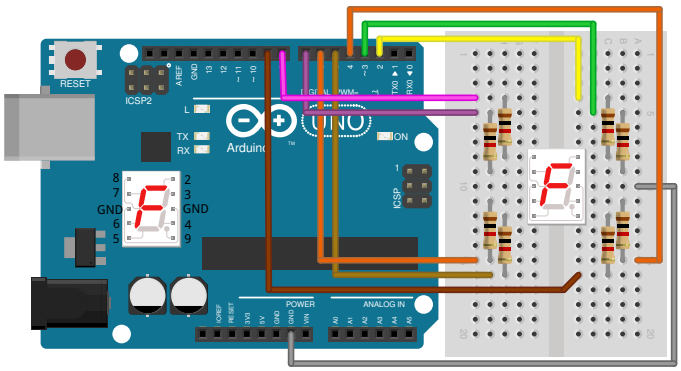
The concept

Connect a seven-segment display to your Arduino.

Use binary operations to encode symbols.

Display a hexadecimal countdown.

Connecting the display



Use 1kΩ resistors to limit the LED currents.

Make sure the pin numbers match the ones in the diagram.

Slicing bits

```
1 // 170 in binary:
2 int binary_number= 0b10101010;
3
4 void loop() {
5   for (int i=0; i<8; i++) {
6     digitalWrite(led_segments[i],
7                  (binary_number /*TODO*/ (1<<i))!=0);
8   }
9 }
```

...

Replace `/*TODO*/` with an operator so that the binary representation of `binary_number` is displayed on the LEDs.

Observation:

The & (binary and) operator can be used to apply a bit mask.

```
  0b01010101
& 0b00001111
= 0b00000101

  0x55
& 0xf0
= 0x50
```


Displaying Symbols

Edit the constant `binary_number` in the previous program so that the number `1` is displayed on the seven-segment display.

The final countdown

On the following slides we will build a hexadecimal countdown timer using the segment encoding developed earlier.

```
uint8_t hex_to_segments[]= {
  //.GFEDCBA
  0b00111111, // 0
  ...
  0b01110001, // F
};
```

...

The final countdown

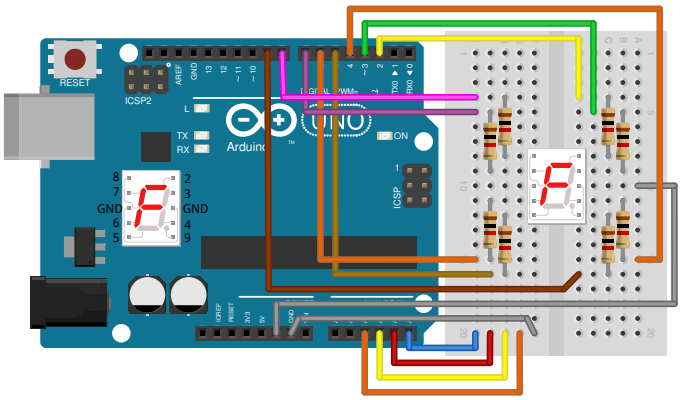
Download the skeleton program on the previous slide.

Replace `/* TODO */` with code that displays a hex countdown on the display and stops at zero.

`F E D C A 9 8 … 2 1 0` … *BOOM?*

Optional: when the countdown is done display the following number sequence in hexadecimal: `13, 14, 10, 13`.

Wirecutter



Connect pins `A2` to `A5` to GND.

Wirecutter

```
1 if ( digitalRead(wire_pins[0]) &&
2      digitalRead(wire_pins[1]) &&
3      digitalRead(wire_pins[2]) &&
4      digitalRead(wire_pins[3])) {
5   sucess_animation();
6 }
```

...

Edit the program above according to the comment in the loop function.

Use the reset button on the arduino to restart the program.

Working example

Extra task

Wirecutter 1337

```
1 if (wires & 10) fail_animation();
2 if (!(wires ^ 5)) sucess_animation();
```

...

Try beating the program above *without dying*.

Appendix

The following slides are annotations that are intended to help you write better C code.

They are not part of the tutorial or the lecture. You are not required to read or understand them.

Please direct any questions regarding them directly to the autor of the slides.

Appendix the `var␣=` pattern

In the lecture you have learned, that the statement `i=i+1` can also be written as `i++`.

You have also learned, that the statement `i=i+2` can be also written as `i+=2`.

The second case is actually only a special case of a more generic pattern as all operations that take two arguments (+,-,/,*,%,&,|,^,<<,>>) can be written in the shorter `var␣=` format.

```
a&= 1;  ⇔ a= a & 1;  // Binary and (clear all but the first bit)
a^= 1;  ⇔ a= a ^ 1;  // Binary xor (toggle the first bit)
a|= ~1; ⇔ a= a | ~1; // Binary or  (clear the first bit)
a<<= 1; ⇔ a= a << 1; // Shift left (multiply by two)
```


Appendix stdint

Observation: the C integer types do not have the same length on different architectures.

```
1 // Wait one second.
2 for (int i=0; i<60000; i++) {
3   delay(1);
4 }
```

On a modern computer, where an `int` is 32 or more bits long, the code above would wait for one second.

On an Arduino it would loop forever as `i` is only 16 bit wide and can never be greater than 32767.

This is *inconsistent*.

Appendix stdint

To avoid these inconsitencies you should, in most cases use the integer types defined in `stdint.h`.

In the Arduino environment this file is included by default. When writing C in different environments you can include it using `#include <stdint.h>`.

Appendix stdint

`stdint.h` defines, among others, the following types:

```
int8_t  - a signed integer that is guaranteed to be 8 bits wide
uint8_t - a unsigned integer that is guaranteed to be 8 bits wide

int16_t, uint16_t - integers that are guaranteed to be 16 bits wide
int32_t, uint32_t - integers that are guaranteed to be 32 bits wide
int64_t, uint64_t - integers that are guaranteed to be 64 bits wide
```

These types should be used when your code depends on the size of a type.

The following statement is *true* on all platforms:

```
uint8_t i= 0x80;
if ((i<<1) == 0) {
```

Appendix stdint

When you are only interested in the minimum size a type has you can use the following types:

```
int_fast8_t, int_fast16_t, int_fast32_t, int_fast64_t     - signed
uint_fast8_t, uint_fast16_t, uint_fast32_t, uint_fast64_t - unsigned
```

These types are guaranteed to be at least the specified number of bits wide but may be wider.

The following statement *could* be *false* on some platforms:

```
uint_fast8_t i= 0x80;
if ((i<<1) == 0) {
```


Appendix size_t

When dealing with memory and pointers (not yet discussed in the lecture) you often have to pass around the size of an object or a the number of objects in an array.

In these cases you should use the `size_t` type as it is guaranteed to be able to hold the size of any object that fits into the memory.

```
1 void bad_example(struct example *elem, int num_elem)
2
3 void good_example(struct example *elem, size_t num_elem)
```

When you are doing calculations on sizes (dangerous) and the result may be negative (very dangerous) you should use the signed `ssize_t` type.