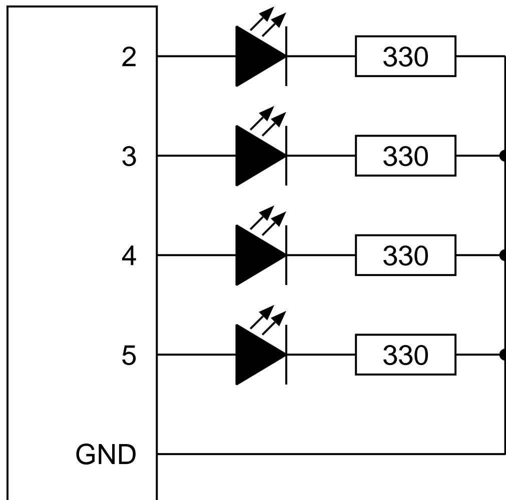
Microcontrollers Systems and Interfacing

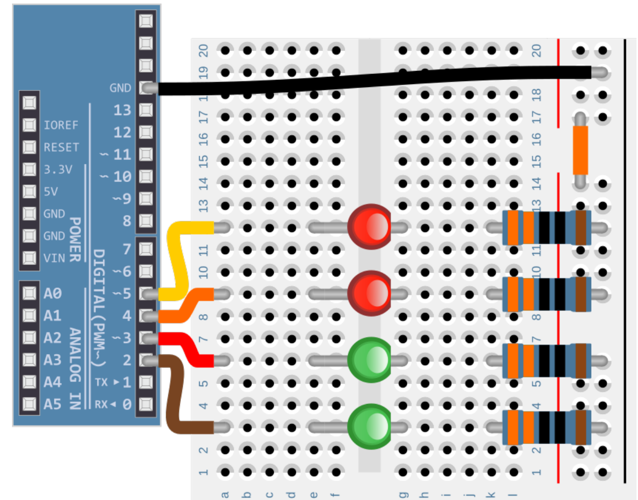
week 7 experimental lab

1 Using external LEDs

Let's connect several LEDs to the digital outputs and investigate how to manage them in software.

► Attach some LEDs to digital pins 2, 3, 4, and 5. Use an appropriate series current-limiting resistor for each LED.





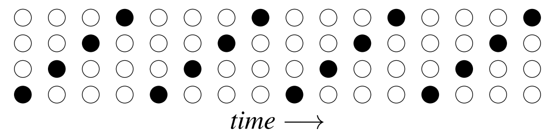
For each LED that you attach:

1. Add a line in the `setup()` function that sets the corresponding `pinMode` to `OUTPUT`.
2. Add a line to the `loop()` function that uses `digitalWrite()` to turn the LED on and off.
3. Run the program to verify that the LED is working.

```
void setup() {
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  ...etc...
}

void loop() {
  digitalWrite(2, HIGH);  delay(100);  digitalWrite(2, LOW);
  digitalWrite(3, HIGH);  delay(100);  digitalWrite(3, LOW);
  ...etc...
}
```

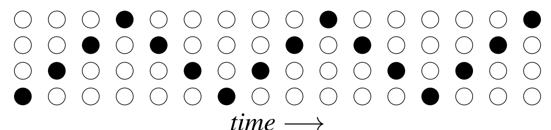
When you are finished you should have four (or more) LEDs attached and your program should be flashing all the LEDs one after the other, in sequence.



1.1 Making patterns on the LEDs

► Modify your program to 'bounce' the light back and forth along the line of LEDs.

Hint: You should be able to do this by duplicating the two lines of program code that flash the middle two LEDs, and then reversing their order.



1.2 Displaying information on the LEDs

You have already created circuits and software to read analogue values, for example, from a potentiometer or a light-dependent resistor. Instead of displaying the values on the serial monitor or plotter, let's display them on the LEDs.

► Obtain an analogue input on A0.

This could be from a potentiometer, a light-dependent resistor, or a temperature sensor. Create the sensor circuit at the opposite end of the breadboard (as far away from the LEDs as you can). Print the value of `analogRead(A0)` on the serial monitor to make sure your analogue input is working.

▶ Display the analogue value on the LEDs instead of the serial monitor.

The usual way to do this is to turn on a different number of LEDs depending on the input value. For example, assuming your analogue input value is between 0 and 100, then you might design your system such that:

input value	*LEDs lit*	
< 20	none	○○○○
< 40	2	●○○○
< 60	2, 3	●●○○
< 80	2, 3, 4	●●●○
≥ 80	all	●●●●

```
void loop() {
  int a0 = analogRead(A0);
  if (a0 >= 20) digitalWrite(2, LOW); else digitalWrite(2, HIGH);
  if (a0 >= 40) digitalWrite(3, LOW); else digitalWrite(3, HIGH);
  ...etc...
}
```

? Does your display system respond well to the full range of input values that you your circuit can generate? Does it work better if you make your program self-calibrating?

2 Managing LED states

Let's make some more ambitious patterns on the LEDs.

▶ Make a 'wave' pattern that turns on all LEDs, one at a time, then turns them off again, one at a time.

Modify your program to draw more interesting patterns on the LEDs. Two possibilities are shown on the right.

○○○●●●●●●●○○ ○○○●●●●○○○●●●●
○○●●●●○●●●●○ ○○●●●●○○○●●●●○
○●●●●○○○●●●● ○●●●●○○○●●●●○○
●●●●○○○○○●●● ●●●●○○○●●●●○○○

? Are these patterns starting to become difficult to program? Would you be happy making a pattern that repeated after, say, 64 different LED states? What is it about the way the LEDs are managed that makes the programming difficult, or tedious, or error-prone?

2.1 Representing the state of all four LEDs

pin number:	5	4	3	2
`LOW` *(off):*	0	0	0	0
`HIGH` *(on):*	1	1	1	1

Manipulating the individual state (on/off) of each LED as four separate, independent 'things' is not convenient. It would be more convenient to use a single value that represents the state of all four LEDs. For example, by representing the state of the LEDs as a single integer.

Because an LED can only be 'on' or 'off', we can represent the state of one LED using a single digit. Let's use '`0`' to represent 'off' and '`1`' to represent 'on'. Four of these digits, grouped together, represent the state of all four of our LEDs at once.

For example, all four LEDs 'off' would be represented as '`0000`', pin 2 LED 'on' would be '`0001`', pins 4 and 2 LEDs 'on' would be '`0101`', all four LEDs 'on' would be '`1111`', and so on.

There are sixteen such four-digit combinations that together describe all possible on/off states of four LEDs. Each of these states can be numbered with an integer in the range 0 to 15. The most sensible way to generate the numbering (from a mathematical point of view) is to consider each four-digit group as a binary (base 2) number.[1]

The table on the right shows how the individual LEDs are combined into a four-digit binary number representing a particular state of the LEDs. Each 'column' has a power-of-two weight (just as each column in a decimal number has a power-of-ten weight). For a particular state of the LEDs, multiplying each digit (`0` or `1`) by its column's weight (a power of two) and then adding them up gives an integer corresponding to that state.

situation	*state*				*decimal*
all LEDs 'off'	0	0	0	0	0
pin 2 LED 'on'	0	0	0	1	1
pin 3 LED 'on'	0	0	1	0	2
pin 4 LED 'on'	0	1	0	0	4
pin 5 LED 'on'	1	0	0	0	8
pins 5 and 2 LEDs 'on'	1	0	0	1	9
pins 5 and 4 LEDs 'on'	1	1	0	0	12
all LEDs 'on'	1	1	1	1	15
	×	×	×	×	
binary weight:	8	4	2	1	*sum* ↗
=	2^3	2^2	2^1	2^0	

The decimal values corresponding to the states are also shown in the table. However, you do not need to use decimal if you don't want to: when programming the LEDs you can work entirely in binary, if you prefer. (Once you get used to it, working in binary — or any other 'power-of-two' number base — is much better than working in decimal, because the microcontroller itself uses binary representation directly to store and manipulate numbers.)

2.2 Writing a super-useful utility function: `setLEDs()`

Let's use an integer to represent the state of all the LEDs. Having decided to do that, the next thing to do is write a function called `setLEDs()` that turns LEDs on or off according to the value of that integer.

[1] Binary (or 'base 2') because we are using only two digits, `0` and `1`. Compare with decimal (or 'base 10') which uses ten digits: `0`, `1`, ..., `8`, `9`.

▶ Create an empty 'skeleton' for your `setLEDs()` function.

The `setLEDs()` function needs one parameter to tell it what do do: an `integer` whose digits (in binary) represent whether each LED should be on or off. Since `setLEDs()` does not calculate any useful result to use in an expression, it should be treated as a command (like `digitalWrite()`) rather than a function (like `analogRead()`). The template therefore should look just like `setup()` and `loop()` (which do not return any result) except that it needs one parameter, whose type ('`int`') and name ('`pattern`') are written inside the '`(`'...'`)`' parentheses.

```
void setLEDs(int pattern) {
  // turn LEDs on or off
  // according to the
  // value of PATTERN
}
```

To perform its task, `setLEDs()` needs to test individual digits in the binary representation of `pattern`. Luckily the programming language that we are using has two features that make this very easy:

1. Integers can be written directly in binary. Just write a '`B`' (or '`0b`') followed by the digits ('`0`' or '`1`') of the number.
2. The numerical operator '`&`' tells us whether two numbers have a '`1`' in the same column of their binary representations.

Together these let us test each digit in `pattern` to see if it is `0` or `1`. Let's explore each of these features briefly before using them to implement `setLEDs()`.

2.2.1 Exploring binary constants

To investigate the correspondence between decimal and binary numbers...

▶ Make `setLEDs()` print the numeric value of the `patterns` parameter

First, if necessary, add a statement to initialise the serial monitor in `setup()`.

Then add a statement to the body of `setLEDs()` that prints its parameter '`pattern`' on the serial monitor.

Finally, in the body of `loop()`, use the function `setLEDs()` several times with some binary constants to see their value in decimal.

? What should the decimal values of each of these binary numbers be? Try to calculate the values by yourself, but also refer to the table on the previous page if you are in any doubt.

Upload the program to the microcontroller, open the serial monitor, and check that the numbers printed correspond to your expectations.

```
void setup() {
  // after other initialisation, add...
  Serial.begin(115200);
}
void setLEDs(int pattern) {
  Serial.println(pattern);
}
void loop() {
  setLEDs(0b0000); // binary
  setLEDs(0b0001);
  setLEDs(0b0010);
  setLEDs(0b0100);
  setLEDs(0b1000);
  setLEDs(0b1001);
  setLEDs(0b1100);
  setLEDs(0b1111);
  delay(1000); // <-- decimal ;-)
}
```

▶ Try changing the numbers in `loop()` to different binary numbers. Try slightly longer ones. Can you reliably predict the decimal value of each number you try?

▶ Print the numbers in binary.

Modify `setLEDs()` so that it prints `pattern` in binary. To do this, add a second parameter to `Serial.println()` whose value is '`BIN`'. This tells `Serial.println()` to use binary when printing `pattern`. Upload the program and check that the output on the serial monitor corresponds to the values you have written inside the `loop()` function.

```
void setLEDs(int pattern) {
  Serial.println(pattern, BIN);
}
```

If you are curious, you can now try printing some decimal numbers in binary. Just replace some of the constants in `loop()` with decimal numbers (without the '`0b`' prefix). Check that you can predict the binary representation for each of the decimal numbers that you try.[1]

2.2.2 Exploring the digits of a binary number

Let's now use the '`&`' operator to test each individual digit of `pattern`.

▶ Print whether each digit of `pattern` is a '`0`' or a '`1`'.

The '`&`' operator (pronounced 'and') compares two binary number operands, let's call them '`A`' and '`B`', digit by digit. The result is a binary number which has the digit '`1`' in all the places where both `A` *and* `B` have a '`1`', and the digit '`0`' everywhere else. For example, '`0b1100 & 0b0110`' = '`0b0100`' because only the third digit (from the right) is a '`1`' in both input operands. (You can prove this by putting '`setLEDs(0b1100 & 0b0110)`' in your `loop()` function and checking the result in the serial monitor.) The table shows how `&` operates for one pair of corresponding digits.

a	*b*	*a* & *b*
0	0	0
0	1	0
1	0	0
1	1	1

[1] The trick is to figure out how to make the number by adding up powers of two. For example, 42 is made by adding up $32+8+2$, which is $2^5+2^3+2^1$, and so its binary representation must be '`101010`'.

To check if a specific digit of a number is `0` or `1`, just '`&`' it with the power of two corresponding to that digit position. (For example, the table on the right shows how to check the digits of the number `0b1100`.)

The result can be used as the condition in an `if` statement. If the tested bit in the number is `1` then the result is non-zero and is considered 'true' by the `if` statement. If the tested bit in the number is `0` then the result is zero and is considered 'false' by the `if` statement. The programming pattern for testing digits a binary number (for example, our `pattern` parameter) therefore looks like this:

digit to test		*number tested*		*result*		*logical value*
`0b1000`	&	`0b1100`	=	`0b1000`	⇒	`true`
`0b0100`	&	`0b1100`	=	`0b0100`	⇒	`true`
`0b0010`	&	`0b1100`	=	`0b0000`	⇒	`false`
`0b0001`	&	`0b1100`	=	`0b0000`	⇒	`false`

```
  if (0b0001 & pattern) Serial.println("digit 0 is 1"); else Serial.println("digit 0 is 0");
  if (0b0010 & pattern) Serial.println("digit 1 is 1"); else Serial.println("digit 1 is 0");
  if (0b0100 & pattern) Serial.println("digit 2 is 1"); else Serial.println("digit 2 is 0");
  if (0b1000 & pattern) Serial.println("digit 3 is 1"); else Serial.println("digit 3 is 0");
```

► Print the digit values for various numbers.

Modify `setLEDs()` so that the body of the function contains the four lines shown above. (To keep the output tidy, you might want to add an additional `Serial.println()`, with no parameter, immediately after those four lines.) Check the output on the serial monitor. Verify that the output in the serial monitor corresponds properly to the digits of the binary numbers that you are using in `loop()`.

2.2.3 Implementing `setLEDs()`

You now have everything you need to implement `setLEDs()`. By replacing

`Serial.println("digit 0 is 1")` with `digitalWrite(2, HIGH)` and
`Serial.println("digit 0 is 0")` with `digitalWrite(2, LOW)`

(and similarly for the other three `if` statements in `setLEDs()`), your `setLEDs()` function will turn the LEDs on or off depending on whether the corresponding digits in `pattern` are `1` or `0`. Your `setLEDs()` function should now look like this:

```
void setLEDs(int pattern) {
  if (0b0001 & pattern) digitalWrite(2, HIGH); else digitalWrite(2, LOW);
  if (0b0010 & pattern) digitalWrite(3, HIGH); else digitalWrite(3, LOW);
  // insert similar code for pins 4 and 5 here
}
```

You should now be able to write this week's very first program in a much more compact way. Test your `setLEDs()` function with the following `loop()` code

```
void loop() {
  setLEDs(0b0001);   delay(100);
  setLEDs(0b0010);   delay(100);
  setLEDs(0b0100);   delay(100);
  setLEDs(0b1000);   delay(100);
}
```

which should produce exactly the same 'moving dot' pattern on the LEDs.

► Try other patterns.

Now that you have a working `setLEDs()` function, try using it to generate some of the other patterns that you made. Are those patterns easier to make using `setLEDs()`?

3 Review

This week went deeper into programming theory than previous weeks. The skills that you have learned are very powerful. Here is a brief summary of the most important points:

- Numbers can be used to represent information that is not numeric. This is a huge 'design idea'.
- Each binary digit ('bit') of an integer can be used to independently store a yes/no, true/false, `1`/`0`, value.
- Integer constants can be written in other bases, for example, using a '`0b`' prefix to write binary constants.
- In some situations, decimal is not the most convenient number base to use.
- The `&` (and) operator can be used to test whether a specific digit within an integer is `1` or `0`.
- The result of the `&` operation can be used in an `if` statement to control what the program does.

The techniques we looked at this week are used by professional embedded programmers every day.

4 Challenges

4.1 Dot mode vs. bar mode

When we displayed an analogue voltage on the LEDs we lit up one, two, three, or all four LEDs. The LEDs therefore created a solid ‘bar’ of light. An alternative display would light up only one LED according to the analogue voltage. Modify your analogue voltage display program to light only one LED to indicate the level.

	LEDs lit			
input value	*‘bar’ mode*		*‘dot’ mode*	
< 20	none	○○○○	none	○○○○
< 40	2	●○○○	2	●○○○
< 60	2, 3	●●○○	3	○●○○
< 80	2, 3, 4	●●●○	4	○○●○
≥ 80	all	●●●●	all	○○○●

4.2 Make a binary counter

Use `setLEDs()` to make the LEDs count in binary from `0000` to `1111`. Do not call `setLEDs()` sixteen times. Instead call `setLEDs()` just once, from inside a loop that uses a variable to count from 0 to 15.

? What happens if you let your variable count from 0 to 31? Why?

4.3 Independent PWM

Use pulse width modulation to illuminate the first LED at 50% brightness, the second at 25%, the third at 12.5%, and the fourth at 6.25%.

4.4 Independent LED control

Write two functions similar to `setLEDs()` that work as follows:

`enableLEDs(int pattern)`
turns on every led corresponding to a `1` in the `pattern`. LEDs that are not indicated by a `1` in `pattern` are not affected (they remain as they were, either on or off).

`disableLEDs(int pattern)`
turns off every led corresponding to a `1` in the `pattern`. LEDs that are not indicated by a `1` in `pattern` are not affected (they remain as they were, either on or off).

Re-implement `setLEDs()` using `enableLEDs()` and `disableLEDs()`. (Do not use `digitalWrite()` anywhere in `setLEDs()`.) Hint: look up the ‘bitwise complement’ operator which is written ‘∼’ in the Arduino programming language.[2] (For example, ‘∼`x`’ is the bitwise complement of ‘`x`’.)

4.5 Use arithmetic operators on bit patterns

Try setting the LEDs to ‘`0b0001 << 0`’, then to ‘`0b0001 << 1`’, and then ‘`0b0001 << 2`’, and ‘`0b0001 << 3`’, and finally ‘`0b0001 << 4`’. What does the ‘<<’ operator do?

Try setting the LEDs to ‘`0b1000 >> 0`’, then to ‘`0b1000 >> 1`’, and then ‘`0b1000 >> 2`’, and ‘`0b1000 >> 3`’, and finally ‘`0b1000 >> 4`’. What does the ‘>>’ operator do?

[DIFFICULT] Use what you have just discovered to write the ‘back-and-forth bouncing dot’ pattern from Section 1.1 using only one call to `setLEDs()`. Do not use an explicit loop in your program.

Hint: instead of explicitly setting the LEDs to the desired patterns, store the initial pattern (e.g., `0b0001`) in a variable. In another ‘current direction’ variable remember whether the pattern is currently moving to the left or to the right. Use the << and >> operators to update the variable with the next pattern in the sequence. Use `&` on the pattern in combination with the ‘current direction’ variable to check if the pattern has ‘reached the end’ of the LEDs and should therefore start moving in the opposite direction.

4.6 The same, but in decimal

Of course, absolutely everything we just did could also be done (almost identically) using decimal numbers. However, since it takes $\log_2(10) \approx 3.32$ binary digits to represent one decimal digit everything would be 3.32 times less space efficient (not to mention the huge extra computational cost of extracting each digit) — and in embedded systems, efficiency is often critical.

[DIFFICULT] Rewrite `setLEDs()` to check the digits in the *decimal* representation of `pattern`. Check that it works by removing the ‘`0b`’s from the front of the numbers describing one of your patterns. Hint: to extract a single digit from a decimal number, divide by 10^n (a power of 10) and then take the remainder on division by 10 (using the modulus operator ‘`%`’).

[EXTREMELY DIFFICULT] Instead of ‘on’ and ‘off’, make the digits between `0` and `9` set the *brightness* of the corresponding LED between 0% and 100%.

[2] E.g: `https://en.wikipedia.org/wiki/Bitwise_operation#NOT`

Microcontrollers Systems and Interfacing

week 7 reference material

A Positional number systems

A *positional number system* represents numeric values as sequences of one or more *digits*. Each digit in the representation is weighted according to its position in the number. The weights increase towards the left by a constant factor called the *base* or *radix* (the Latin word for 'root'). The most familiar positional number system is decimal, in which the radix is 10.

$$
\begin{array}{rccccccl}
 & 10^2 & & 10^1 & & 10^0 & & \\
 & 100 & & 10 & & 1 & & \\
 & \times & & \times & & \times & & \\
365_{10} = & 3 & + & 6 & + & 5 & & \\
 & 300 & + & 60 & + & 5 & & = 365
\end{array}
$$

Computers represent numbers in hardware using the binary system, in which the radix is 2.

$$
\begin{array}{rcccccccl}
 & 2^3 & & 2^2 & & 2^1 & & 2^0 & \\
 & 8 & & 4 & & 2 & & 1 & \\
 & \times & & \times & & \times & & \times & \\
1101_2 = & 1 & + & 1 & + & 0 & + & 1 & \\
 & 8 & + & 4 & + & 0 & + & 1 & = 13
\end{array}
$$

A radix R representation uses R distinct digits, starting with '0'. Binary uses two digits, '0' and '1'. Decimal uses ten digits, '0' through '9'.

binary	*decimal*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15