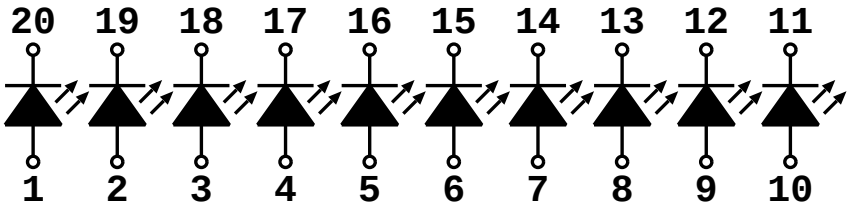
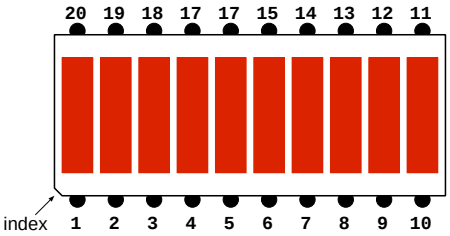
Microcontroller Systems and Interfacing

week 8 experimental lab

1 Using LED 'bar graph' arrays

We saw last week that LEDs arranged in a line are convenient for displaying information visually. So useful, in fact, that linear arrays of LEDs are available packaged as a single device. The simplest array of LEDs is the *bar graph* which (usually) contains a line of ten identical LEDs.

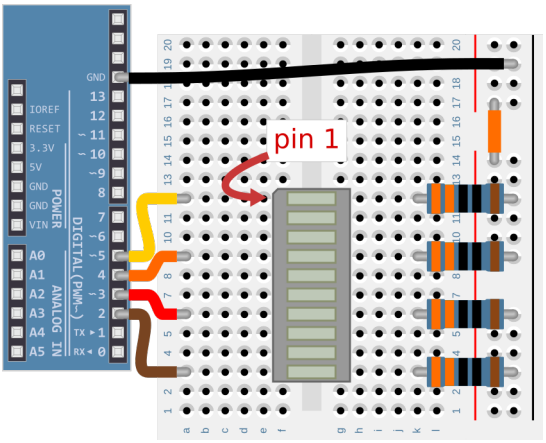




Each LED is independent of the others and has its own anode and cathode connections on opposite sides of the package. For an array of 10 LEDs there are therefore 20 connections on the package.

► Replace the individual LEDs in last week's experiment with a single LED array.

!!! NOTE: you have three LED arrays in your parts kit. The correct ones for this experiment are identical, have a grey front face, and are marked '`KINGBRIGHT DC-10GWA`'. (The other LED array, with a black face and marked '`OSX10201`', is *not* correct for this experiment!)

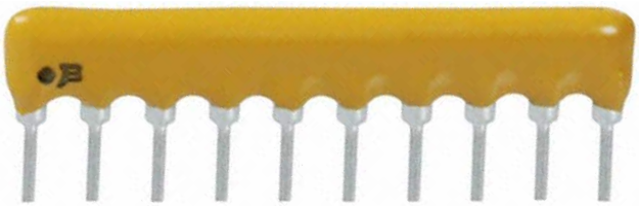


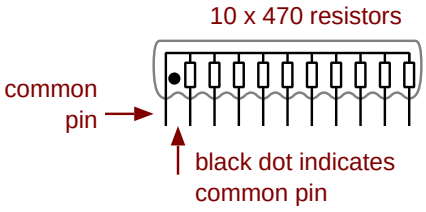
Because LEDs are polarised it is important to connect them the correct way round. Pin 1 is usually indicated by a flattened corner on the package. (This can be difficult to see, but it *is* there on the devices in your parts kit.)

The individual LEDs in the device are electrically identical to the ones you have been using. The same 330 Ω series current-limiting resistors will work just fine. Check that your program from the previous experiment works properly, lighting up four of the LEDs in the array.

► Replace resistors with resistor network.

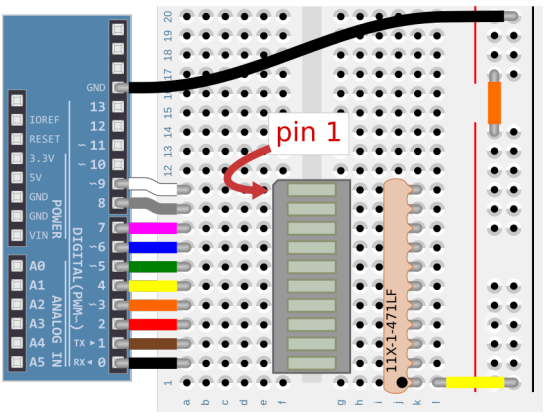
Individual current-limiting series resistors are required for this kind of display, but they are also very tedious to work with. Because of this, resistor networks are available which contain multiple, parallel resistors in a single package. You can replace ten individual resistors with a single resistor network that performs the same function.





The device has 11 pins. One pin connects to all of the resistors, giving them their common ground connection. The other ten pins of the device are connected to the other end of each of the resistors; in our circuit these pins connect to the cathode terminals of the LEDs in the array.

To leave a convenient space for later experiments it is also recommended to move the LED array down one row (so that it begins on the second row from the edge of the breadboard) and to use the last row as the common ground connection for the resistor network, as shown in the diagram. Be sure to place the common pin of the resistor network, indicated with a black dot, on the bottom row.



You can also connect up all ten LEDs. For this experiment we will also use digital pins 0 and 1, so connect pins 0 through 9 to the LED array. The program from the previous experiment should still work, unmodified, illuminating the third through sixth LEDs in the array.

1.1 Displaying information on the LED array

For this section you will have to modify `setup()` to configure pins `0` through `9` as `OUTPUT`s. You will also have to extend your `setLEDs()` function to control ten LEDs, starting from pin 0, using the rightmost 10 binary digits of an integer.

► Use the LED array to display ambient light level from A0 input.

If you like, you can revisit the earlier experiment that displayed the ambient light level on four LEDs. To adapt it for 10 LEDs you will have to change the thresholds from their earlier values of 20%, 40%, 60%, and 80%.

? What threshold values do you have to use? Are they the values that initially seem obvious? If you have N LEDs, what are the threshold values you would have to use (expressed in terms of N)?

1.2 Make a 'USS Enterprise turbolift display simulator'

The walls of the turbolift in the USS Enterprise have big light indicators that move up or down to indicate the motion of the lift.

► Modify your 'moving dot' program from the last lab to cover all 10 LEDs.

If you did not revisit the ambient light display for 10 LEDs then this is the perfect test of your `setup()` and `setLEDs()` functions.

See Challenge 4.2 for a suggestion of how to make the effect more realistic.

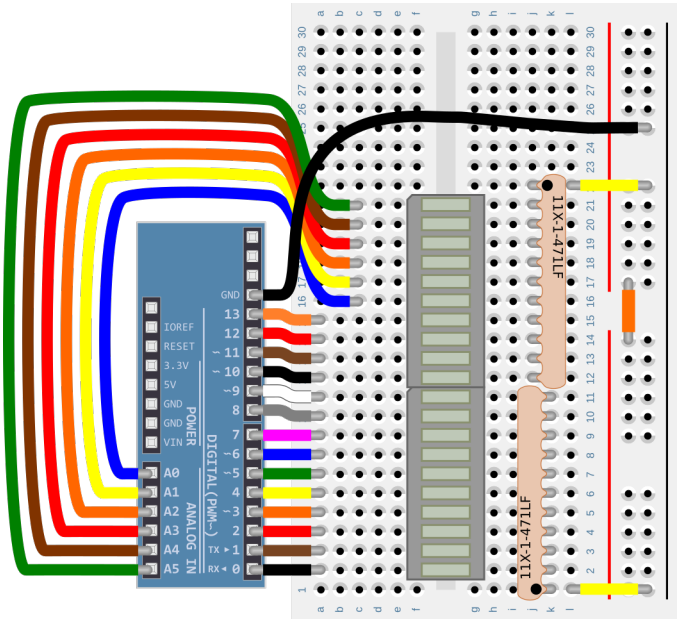
2 Using all available digital outputs

For this section you will have to modify `setup()` to configure pins `0` through `19` as `OUTPUT`s. You will also have to extend your `setLEDs()` function to control twenty LEDs, starting from pin 0, using the rightmost 20 binary digits of an integer.

► Connect a second LED array and resistor network.

Analogue pins A0 to A5 are also digital pins 14 to 19, giving you a total of 20 digital pins. You can set their pin mode to `OUTPUT` and set their values using `digitalWrite()`, just like the other 14 digital pins.

You can plug your second LED array to the breadboard next to the first one, without a gap. Make sure pin 1 of the LED array is on the same side as your first LED array. You also have a second resistor network to use for the required series, current-limiting resistors. You will have to orient the resistor network in the opposite direction (so the two black dots on the two networks are facing away from each other). Connect the new LED array to digital pins 11, 12, 13 and to analogue pins A0 through A5.



Modify your `setup()` function so that pins 0 to 19 are `OUTPUT`s. Modify your `setLEDs()` function so that it control pins 0 to 19 using the rightmost 20 binary digits of its `leds` parameter. Modify your program to display a moving dot on all 20 LEDs. Check that it works.

? Does your moving dot program work? Can you figure out why not?

2.1 Avoiding the limit of 16-bit `ints`

Your `setLEDs()` function broke. The problem is that an `int` can only store 16 binary digits (bits). You will have to use a larger integer that can store up to 32 bits instead. These larger integers are called `long`s, and to use them just change `int` to `long` wherever you need to use more than 16 bits.

► Fix `setLEDs()` so that it can handle 20 pins (20-bit integers).

Changing the definition of `setLEDs()` from '`void setLEDs(int leds)`' to '`void setLEDs(long leds)`' should do the trick. Verify that your program now works properly with 20 LEDs.

► Create a 20-bit counter and display its value using setLEDs.

Create a variable outside `setup()` and `loop()` (for example, between them, just before the definition of `loop()`). Call is something imaginative, such as '`counter`'. Initialise it to `0`.

Inside `loop()`, display your `counter` on the LEDs then add `1` to it. Maybe also put a `delay(100)` at the end of `loop()` too. Check that your LEDs appear to be showing small binary numbers counting upwards.

Remove the `delay()`. Check that your counter eventually lights up all the LEDs.

► Figure out the speed of counter.

How can you do that? Hint: one way is to find the LED that is flashing approximately once per second. Then count how many LEDs are flashing faster than that one, lets call that number N. The number of times counter is being incremented per second is just 2^N.

► Make your counter faster.

The next section explains how.

3 Direct access to the microcontroller's I/O pins

Physically, the I/O pins are grouped into three sets of 8 pins. Each group of 8 pins is known by a letter (`B`, `D`, and `C`) and each pin within the group is known by a number (between 0 and 7).

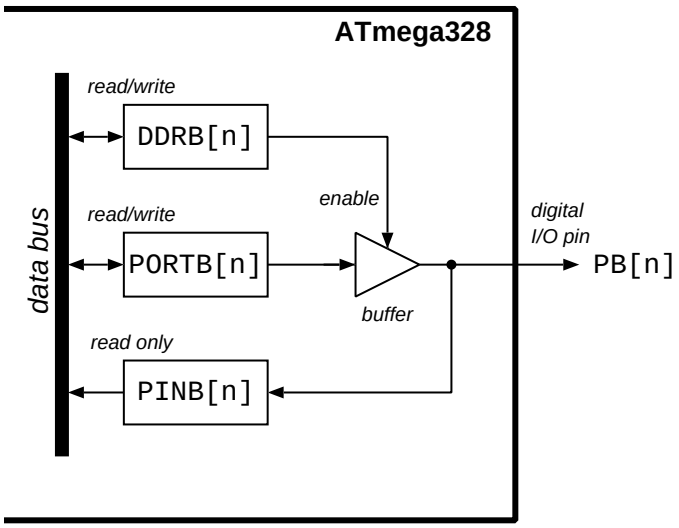
The pins `D0`–`D7` correspond to digital pins marked 0–7 on the microcontroller connectors.

The pins `B0`–`B5` correspond to digital pins marked 8–13.

The pins `C0`–`C5` correspond to digital pins marked 14–19 (also known as `A0`–`A5`).

3.1 Configuring digital pins in parallel

Each group of pins has two 'registers' associated with it, called `PORT` and `DDR` (which stands for 'data direction register'). Therefore we have three `PORT` registers called `PORTD`, `PORTB`, and `PORTC` and three more `DDR` registers called `DDRD`, `DDRB`, and `DDRC`.



The `DDR` registers control the direction of each pin, either `OUTPUT` (the pin writes values to an external device) or `INPUT` (the pin reads values from an external device). To configure a pin to be an `OUTPUT`, write a `1` to the corresponding bit in the `DDR`.

For example, pin 2 is the third bit in the `D` group (counting from the right, starting at `0`). To make pin 2 an output, set '`DDRD = 0b00000100;`' in your `setup()`.

As another example, pin 13 is the sixth bit in the `B` group (counting from the right, starting at `0`). To make pin 13 an output, set '`DDRB = 0b00100000;`' in your `setup()`.

You have probably already guessed that this is exactly what `pinMode()` is doing for you when you configure a pin as an `OUTPUT`.

The advantage of using the `DDR` registers instead of `pinMode()` is that you can configure up to 8 pins at the same time. For example, to make all 20 digital pins be `OUTPUT`s, all you need is to put these three lines in your `setup()`:

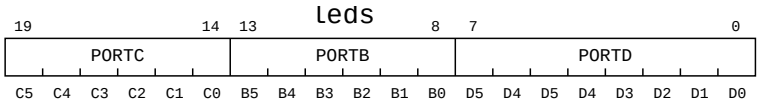
```
void setup() {
  DDRD = 0b11111111;  // pins 0 to 7 are all OUTPUTs
  DDRB = 0b00111111;  // pins 8 to 13 are all OUTPUTs
  DDRC = 0b00111111;  // pins 14 to 19 (A0 to A5) are all OUTPUTs
}
```

3.2 Writing to digital pins in parallel

The three `PORT` registers work exactly like the `DDR` registers except that you use them to write `HIGH` or `LOW` values to the digital pins. Writing `1` to a bit in a `PORT` register sets the corresponding pin to `HIGH`. Writing `0` to a bit in a `PORT` register sets the corresponding pin to `LOW`. For example:

```
PORTD = 0b11111111; //  pins 0 to 7 all HIGH
PORTD = 0b00000000; //  pins 0 to 7 all LOW
PORTD = 0b10000011; //  pins 0, 1, and 7 HIGH and pins 3 to 6 LOW
PORTB = 0b00111111; //  pins 8 to 13 all HIGH
PORTC = 0b00000000; //  pins 14 to 19 all LOW
```

When using a single integer to control many pins (as in your `setLEDs()` function) you will have to split that integer into several parts and then write each part to the appropriate pins. For example a 20-bit integer that control all the pins would have to be split into three parts like this:



To split this integer into three parts you can use two operators provided by the programming language:

`x >> y` gives an integer equal to `x` but with all the bits shifted right by `y` places
For example: `0b01110000 >> 2` gives you `0b00011100`

`x & y` gives an integer in which only bits set in both `x` and `y` are set
For example: `0b01111100 & 0b00111111` gives you `0b0011111100`

Using those operators you can extract the three parts of `leds`, corresponding to the three `PORT` registers, like this:

```
PORTD =  leds                    ;   // bits  0 to  7 already correct for PORTD
PORTB = (leds >>  8) & 0b00111111;   // bits  8 to 13 to the rightmost 6 bits of PORTB
PORTC = (leds >> 14) & 0b00111111;   // bits 14 to 19 to the rightmost 6 bits of PORTC
//              ^^      ^
//              ||      |
//              ||      +-- select only the 6 bits of the integer that we actually want
//              ||
//              ++-- shift the wanted 6 bits to the rightmost end of the integer
```

▶ Program `PORTD` directly.

Modify your `setup()` function, use `DDRD` to set all 8 digital pins from 0 to 7 as outputs. (Ignore other pins for now.) In your `setLEDs()` function, write the `leds` value directly to `PORTD`. (Ignore other pins for now.) In your `loop()` function, add a `delay()` 50 to slow down the program so that you can see the bits being displayed. Run the program and verify that the first 8 LEDs are counting in the way you expect.

▶ Program `PORTB` and `PORTC` directly.

Modify your `setup()` function to set all 20 digital pins as outputs (as explained above). Modify your `setLEDs()` function to write all 20 digital pins (as explained above). Modify your `loop()` function so the delay becomes lower, for example, `delay()` 1. Run the program and verify that the first 14 LEDs are counting in the way you expect.

Remove the delay entirely from your `loop()` function. Verify that all 20 LEDs are counting in the way you expect.

? How fast is your counter counting? (Use the same procedure as described earlier.) How much faster is this version of `setLEDs()`, compared to the one that used lots of `digitalWrite()`s?

3.3 Warning!

You can rely on `pinMode()` and `digitalWrite()` working properly on almost any Arduino-compatible microcontroller system. If you use those functions in your programs then your programs will probably work 'everywhere'. The same is not true when accessing hardware registers directly. As soon as your program starts to use `DDR` and `PORT` registers directly it may not be portable, and will only work on the exact same make and model of microcontroller as you used to develop it.

Is the loss of portability worth the gain in performance? That is an engineering decision that only you can make.

4 Challenges

4.1 Faster counting

Try to make the counter display even faster. You can do this by never returning from the `loop()` function. (Every time you return from that function the microcontroller does some housekeeping work to support things like the serial monitor. Since we are not using the serial monitor, there is no real need to return from `loop()` at all.)

Use a `while(1)` statement to add an infinite loop to your loop() function. Place all of the counter program inside the loop. Look at the LEDs and figure out how many times the counter is being incremented every second.

? How much faster is this than the version that returns from `loop()`? How much time do you save by not returning from `loop()`? How much time is 'wasted' each time you return from `loop()`?

4.1.1 Obtain the ultimate counting speed

The shifting of bits in `setLEDs()` is wasting a little time. Let's avoid that.

In your `loop()` function, remove the counter variable and the code that increments it. Replace that with a single `for()` loop:

```
for (int i = 0;  i <= 0b11111111;  i = i + 1) {
  PORTD = i;
}
```

This loop displays a count on the first 8 LEDs (pins 0 through 7) from `00000000` to `11111111` *as fast as possible*. It even avoids the overhead of using the function `setLEDs()`. Let's call this the 'i-loop'.

The next 6 LEDs (on pins 8 through 13) are connected to `PORTB`. They should count up by 1 each time the first 8 LEDs count through a *complete* cycle of 256 values (in the 'i-loop'). To arrange for this you can 'nest' the i-loop inside another loop, like this:

```
for (int j = 0;  i <= 0b00111111;  j = j + 1) {
  PORTB = j;
  // the original i-loop goes here, inside the new loop:
  for (int i = 0;  i <= 0b11111111;  i = i + 1) {  // i-loop
    PORTD = i;                                     // i-loop
  }                                                // i-loop
}
```

Let's call that the 'j-loop'. Note the change of variable name (to prevent the j-loop from interfering with the i-loop), the different range of values in the loop (because the j-loop controls only 6 LEDs, not 8), and (of course!) the use of `PORTB` instead of `PORTD` because the j-loop controls a different set of LEDs.

You now have two nested loops that count the first 14 LEDs from `00000000000000` to `11111111111111` *as fast as possible*. To complete the 'fast-as-possible' counting program you need to control the final 6 LEDs, connected to `PORTC`. To count them properly you need to nest the j-loop inside one more loop. That final outer loop will look just like the j-loop except that it uses a different variable name (to avoid interfering with either the j- or i-loops) and it uses `PORTC`.

Implement the final outer loop and check that all 20 LEDs appear to be counting the way you expect (you will only be able to see the last 6 LEDs or so changing because the others are too fast).

? How much faster is this version of the counter? [VERY DIFFICULT] Can you think of any way to make it even faster?

4.2 [DIFFICULT] A more realistic turbolift

The real (haha! ☺) turbolifts can move in different directions. Modify your turbolift simulator so that the movement of the dot is controlled by a potentiometer. In the *centre* of its travel (so the `analogueRead()` value will be about `511`) the turbolift should be stationary: the dot should be still, not moving at all through the LEDs. If you turn the potentiometer towards maximum the dot should start to move, faster and faster as the potentiometer is turned further. If you turn the potentiometer in the other direction, towards minimum, the dot starts move in the other direction, faster and faster as the potentiometer is turned further.

If you make this work, post a video of the circuit working to the Files section of the Teams channel. Just like Kirk and Spock in the famous malfunctioning turbolift incident, you have achieved 'manual control'!

4.3 [DIFFICULT] Avoiding ten series resistors

Can you think of some way to use just *one* series resistor for the *whole* array, *without* the brightness decreasing as the number of 'on' LEDs increases?

Hint: all LEDs still need to go through the single series resistor to protect the microcontroller from over-current (connect their cathodes together and then connect the cathodes to ground through the resistor). You cannot (must not) avoid that. But, maybe you can do something else that ensures no LED will interfere with any of the others?

Microcontroller Systems and Interfacing

week 8 reference material

A Mapping digital I/O pins to/from register bit positions

The D registers control the data direction (DDRD), allow for setting the logic level of output pins (PORTD), and provide for reading the logic level of input pins (PIND) for digital pins 0 through 7. The lowest six bits of the B registers do the same for digital pins 8 through 13.

	DDRB / PORTB / PINB								DDRD / PORTD / PIND							
data direction and I/O registers	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
'digital pin' number	-	-	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A.1 DDR, PORT, and PIN registers

In the DDRs (data direction registers), setting a bit to 1 makes the corresponding digital pin an output. Setting the bit to 0 (the default) makes it an input.

For pins configured as outputs, writing a 1 to the corresponding bit of the PORT register sets the logic level to HIGH. Writing a 0 to the bit sets the output level to LOW.

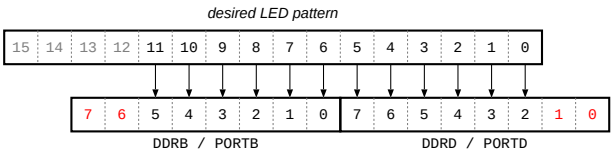
Reading a PIN register returns an 8-bit integer in which each bit set to 1 indicates that the corresponding digital pin is detecting a logic HIGH level.

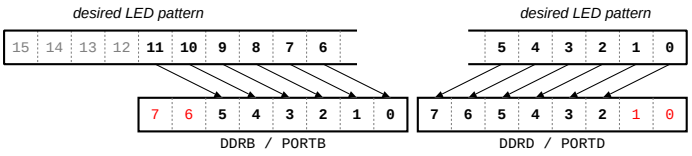
All 8 bits of the registers can be manipulated in parallel. For example, to set digital pins 2 through 13 as outputs, first bits 2 through 7 of DDRD would be set to 1, and then bits 0 through 5 of signal(DDRB) would be set to 1.

```
void setup(void) {   //  OUTPUTS
  DDRD = 0b11111100; // pins 2-7
  DDRB = 0b00111111; // pins 8-13
}
```

A.2 Controlling all outputs in parallel

To control all 12 of these bits in parallel, the lowest 12 bits of a 16-bit `int` can be used to store the desired `pattern` of 1s and 0s on the outputs. Bit 0 of `pattern` controls digital pin 2, which is bit 2 of PORTD. Bit 7 of PORTD would be taken from bit 5 of the `pattern`. Bits 0 through 5 of PORTB would be taken from bits 6 through 11 of the `pattern`.





Shifting the `pattern` left two bits and then masking (bit-wise 'and'ing) the result with `0b11111100` yields the correct bit pattern for PORTD. Shifting the `pattern` right six bits and then masking the result with `0b00111111` yields the correct bit pattern for PORTB.

```
void setLEDs(int pattern)
{
  PORTD = (pattern << 2) & 0b11111100;
  PORTB = (pattern >> 6) & 0b00111111;
}
```

Note that reading a PORT register returns the last value written to it, which is useful when manipulating bits using bit-wise logical operators.

B Bit-wise logical operators

Setting, clearing, and testing individual bits within an integer or hardware register can be done using 'bit-wise' operators ('and' `&`, 'or' `|`, 'exclusive or' `^`, and 'not' `~`) to modify only relevant bits.

`~x`	inverts each bit in `x`	`~0b00100000 == 0b11011111`
`x &= y`	clears bits in `x` where `y` has `0`s	`x &= 0b11011111`
		or `x &= ~0b00100000`
`x |= y`	sets bits in `x` where `y` has `1`s	`x |= 0b00100000`
`x ^= y`	inverts bits in `x` where `y` has `1`s	`x ^= 0b00100000`