Binary Numbers

Normally we think of digits as being one of 0 through 9, leaving ten possible single numbers. Once you get to 10, you need to use 2 digits to represent your count. This numbering scheme, which lines up nicely with the number of 'digits' we have on two hands, is called Base 10 or Decimal, decimus being 10 in latin and all.

But Base 10 is not the only way to count. You could, for example, use base 5, where 4 was the largest single digit, and a value of 5 would require a second digit, or place. This would line up nicely for those who can only count on one hand. In this numbering system '10' would represent five items, '11' would equate to six, and ten items would look like: '20'. Confusing? Sorta, because you have spent your whole life with Base 10. But pretty simple once you know how it works.

As previously outlined, digital computers work by managing either the presence or absence of a charge, and assigning the two values 1 and 0 to those states. The smallest possible usable base is Base 2 — at least two different single digit combinations are required to do any real math. Fortunately that's exactly what we have with a digital computer. And so Base 2 is the numbering scheme that a computer uses.

In Base 2, '10' represents a value of two, '11' is three, and we need to add another place to get to four — '100'. This does mean that binary numbers are somewhat long, compared to their Base 10 version, but there are some interesting things we can do with them.

In Base 10, every new place is a multiple of ten, so we get 10, 100, 1000, etc. In Base 2, every place is a multiple of two, so we get 2, 4, 8, 16, 32, 64, etc. This explains why computers went from 8 bits to 16, then to 32 and now 64, instead of from 8 to 10, or 32 to 40 or some such — it just wouldn't make a lot of sense given how they represent numbers and do math internally.

This is also why your hard drive size doesn't always seem to match what is printed on the box. Terms like kilo, mega, and giga are latin derived SI units for a thousand, million, and one billion — i.e. 1000, 1000^2, & 1000^3. When combined with the term 'byte' to describe digital storage these should be written using the lower case abbreviations: kb, mb, gb. Their Base 2 relatives should be written as: KB, MB, & GB (or sometimes GiB). to define a kibibyte, mebibyte, gibibyte — i.e. 2^{10}, 2^{20}, & 2^{30}.

A single binary digit is called a **bit**. Group 8 of them together as a single value and it's called a **byte**. Half of a Byte, 4 bits, is typically called a **nibble**. There's another common term, **Word**, which is less clearly defined, but that's where the 16 vs 32 vs 64 bit architecture stuff comes in… that number defines the size of the Word on that processor, it's the natural default amount of bits that the processor will work with at a single time. So when someone tells you about their new 128 bit computer, you can just wisely reply: "Word" (as they really mean there computer has a CPU with a 128 bit word size.)

While it can be a little tedious to evaluate large binary numbers in your head, single byte values are pretty simple. A helpful tool is to make a value-column chart with the first 8 powers of 2, then fit your binary number in it underneath them. Then all you need to do is add up the values which have a '1' in their column.

Here are a few examples:

128	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	1	=255

128	64	32	16	8	4	2	1	
0	0	1	1	0	0	1	0	=50

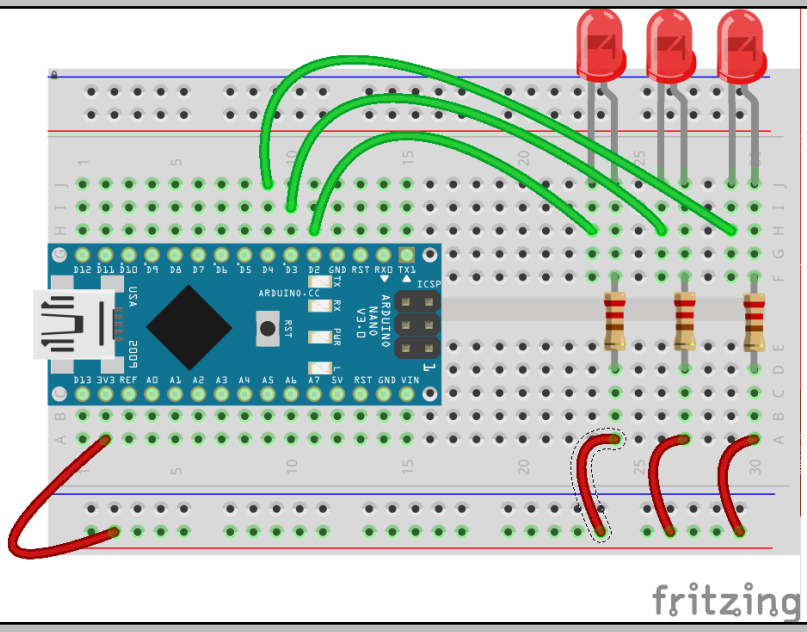
128	64	32	16	8	4	2	1	
0	1	0	0	0	0	1	1	=67

Converting a decimal number to binary is simply the reverse of the procedure. Find the largest bit column value that is less than or equal to (<=) the decimal number, add a one to that column, subtract that amount from the target, and repeat until done.

There are various operators that can be used to isolate, compare, shift and set individual bits. We will only look at a few of those in this class, but such direct operations can be very useful when working close to the hardware, or when absolute speed and size are vital.

In this experiment we will use LEDs to represent individual bits, with a lit LED representing a 1 value. We'll count up from 0 displaying the value on the LEDs until the maximum possible value is displayed, then loop back to 0.

© 2016-17 Dataseam

Components	Wiring Diagram
✓Microcontroller ✓Breadboard ✓3 LEDs ✓current limiting resistors	

Connection Instructions

Use 3 small LEDs, connect each Anode through a resistor to +3.3V. One potential arrangement is shown in the diagram, though there are many options. There should be 3 identical resistors in your kit, but use any you want, knowing that larger values (but not necessarily physically larger resistors) may make for dimmer LEDs.

Then connect the right hand side LED's Cathode to pin D2, The center LED to D3, & the left hand side to D4.

Sketch(es)	binCounter.ino

Analysis Questions

How many bits would you need for a binary clock?
What arrangement would make the most sense for such a clock?
Is there another base that would be better suited for timekeeping?

Programming Tasks

The LEDs represent a 3 digit binary number. Have the LEDs light up to represent each binary number 0 - 7 in sequence with a short pause after each number have it continue back at 0 once it's reached 7.

Notes

For this experiment you will want to use the ***modulo*** operator (%). It is often referred to as modulus or just mod as well. The mod operator returns the remainder component of simple division. For example:

7 % 4 will return 3, because 7 divided by 4 is 1 with a remainder of 3, while 8 % 4 would return 0 as 4 goes into 8 evenly, exactly 2 times, leaving no remainder.

Anything mod 1 is, of course, always 0, and something mod 0 is undefined and each different compiler may treat it differently; returning 0, 1, or an error.

Something mod 2 will be true (1) if the value was odd and false (0) if the value was even.

© 2016-17 Dataseam