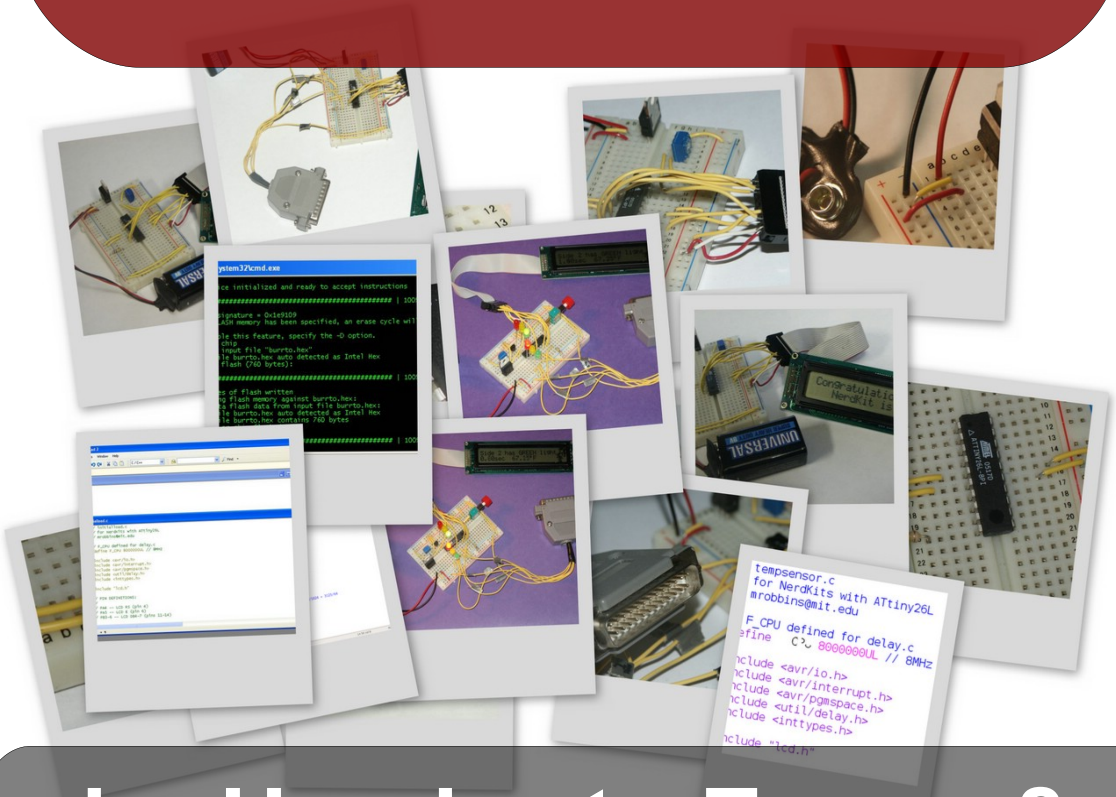
The NerdKits Guide



by Humberto Evans & Michael F. Robbins

The USB NerdKits Guide
by Humberto Evans & Michael F. Robbins
rev 836-10590nt2mc9f3rb8p7jzs

NerdKits

Congratulations on purchasing your Nerd Kit! This is a starter set into the wonderful world of Electrical Engineering. Every great engineer and scientist has started in front of a set of parts with not much more than the desire to build something. Remember, the point is to have fun and explore, but always keep safety first. Read all the warnings carefully, and enjoy...

Keep In Mind

We all had to start somewhere, and the real key to learning new things is to not be afraid. Be creative! Explore! The only places you can hurt yourself are during soldering (which is not needed in this kit) or by poking yourself with the sharp pins of the chips. If you make a mistake, the worst that can happen is you break a component, but they're all easily replaceable. Feel free to take chances, and if you discover anything good, please make sure to let us know.

Introduction to Electricity

Before we can begin experimenting with electricity we need to understand a little bit more about what it is, and how it behaves. The physics of electricity and magnetism is actually really complicated, and is very hard to understand without doing a lot of math. Luckily engineers like to do things the easy way when possible, so we have developed many abstractions and approximations that should make things easier.

Before we can think about how it works, lets spend a little bit of time talking about what it is. Electricity is really just charged particles. The universe is made up of atoms, which are in turn made out of electrons, protons, and neutrons. The protons are positively charged (+) and the electrons are negatively charged (-). Usually, they hang around together and cancel out the charge. However, in some materials (called conductors) like metals the electrons are free to move. Whenever you have moving electrons, you have a current, which means you have electricity!

So then, what makes electrons move? If you have ever heard that opposites attract, it's true in electricity. The reason behind it requires some physics to understand and has to do with lowest energy states, but all you really need to know is that opposite charges attract each other. Electrons move because we can create voltages that have either a positive or negative charge. Since electrons are negatively charged (-) they will flow towards a positive voltage (+).

Now that you know there are electrons that move around in conductors, and that you can make them move by putting a voltage across them, you have the basic knowledge of how all electronics work. We have batteries that create voltages, and we put these voltages across wires to make electrons move. This creates the currents that power our devices.

Unfortunately it's not quite that easy, and you will spend the rest of your electrical engineering life refining your understanding of electricity. One thing young engineers like you often get confused with is that voltages are relative. It is fairly obvious that if an electron is sitting between a +5V and a -5V it will flow towards the +5V. If an electron is sitting between a -5V and and -10V it will flow towards the -5V volts. This is because even though it is negative (with respect to some arbitrary ground), it is less negative than the alternative. When electrons are sitting between a -5V and -5V they do not move at all because neither side is more positive. In electricity all that matters is relative voltage!

Electricity in Practice

So far we have learned about current and how we can cause currents in wires with voltages. The more voltage we have, the harder the electrons are pushed and accelerated in the wire, leading to a higher current.. Here we introduce one more simple element that we deal with in electricity: resistance.

If you think about the statement above, you realize it doesn't make too much sense. It says that if you put a voltage across a wire all the electrons will immediately go the positive end, creating an infinite current. This is obviously not true in real life, which means that there has to be something slowing the electrons down. We call this a resistance. It is just something that resists a current coming across it. Good conductors, like wires, have very little resistance. Materials that don't conduct electricity have a very high resistance, meaning they resist current very well, so electrons don't flow through them. We even have special components that have resistances of different values called resistors.

Now we know enough to make sense of one of the fundamental laws in electricity, Ohm's law. It basically says that the more voltage you have, the higher current you have, but current will be decreased by any resistances.

R stands for the resistance, usually measures in ohms.

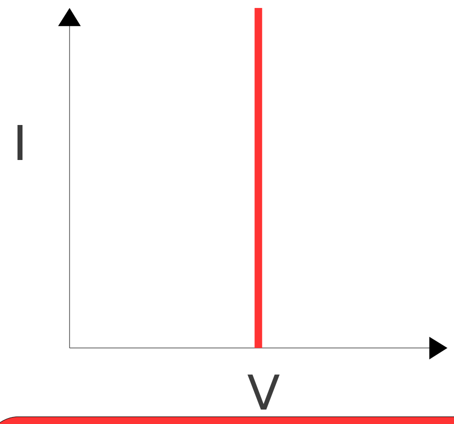
V stands for voltage

$$V=IR$$

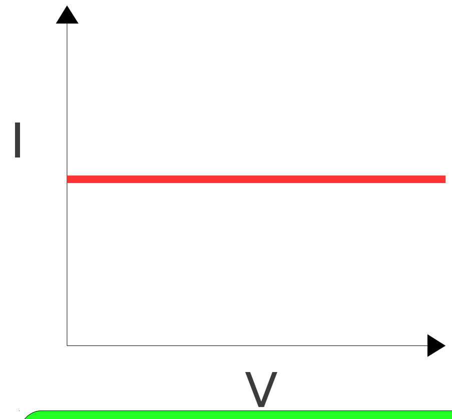
I stands for current, measured in Amps.

Circuit Elements - I-V Plots

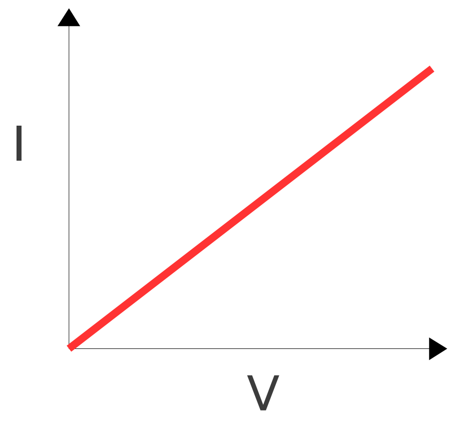
Now we can start thinking about electrical devices, and how they behave in terms of voltages and currents. Normal devices, like batteries, resistors, and light bulbs are two terminal devices. This essentially means there are two leads that come out of the device which you will connect to other things. As we have learned, the relationship between current and voltage is very important in electrical engineering. It is often very useful to characterize a device using what is called an I-V plot. It is simply a graph of the relationship between current and voltage across the terminals of a device. Below are three commonly found examples of I-V plots.



This represents a voltage source. Notice how it will have the same voltage across it no matter how much current you put across it. You are very used to this kind of device. It is a battery.

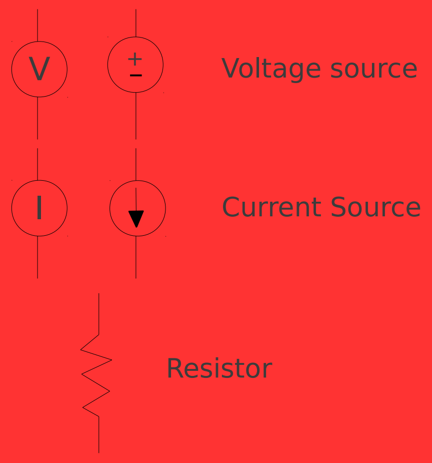


This represents a current source, notice how it always keeps the same current no matter how many volts are put across it.



This is another common circuit element that we already discussed. Can you remember the element with two terminals whose current increases as you put a bigger and bigger voltage across it?

When we draw circuit diagrams we use symbols to represent the elements:



Resistance is Futile

A resistor, just like its name suggests, is a device that has a certain resistance. These devices are very common in circuits and have many uses which you will explore as you play more with electronics.

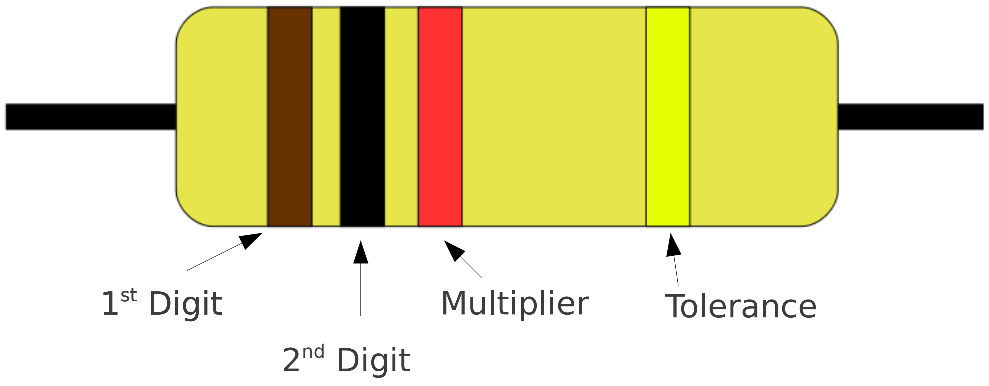
The resistance of a device is measured in Ohms, and every resistor has its resistance written on it. Unfortunately for beginners, the value of a resistor is not printed on using numbers. Instead they use a color code!

Decoding resistor values is not that hard once you know how, and it takes a little practice to get quick at it. There are plenty of online calculators that let you input the color sequence to give you the resistance, but you will never get good at it if you don't decode them yourself.

We use colors to represents different digits:

0	1	2	3	4	5	6	7	8	9
Black	Brown	Red	Orange	Yellow	Green	Blue	Violet	Grey	White

The resistance value is printed on the resistor using the following scheme:



The first three color bands represent the resistance value of the resistor. The first two you read off just like normal digits. In the example above Brown = 1, Black = 0 -> 10. The third band is the multiplier, this band multiplies the other two color bands by 10^{color}. So in this example it multiplies $10*10^2$ = 1000 Ohms.

Review: Let's say we had a resistor with color bands Orange, Orange, Black. The first two are the digits Orange = 3, Orange = 3 -> 33. The multiplier is Black = 0, so $33*10^0$ = 33 Ohms.

Resistance – Extra Tidbits

Thinking $*10^{color}$ can seem like daunting mental computation, but in reality all this does is add 'color' amount of zeros to the number, so if you have a multiplier of 2, it adds two zeros, a multiplier of 0 adds no zeros!

Conspicuously absent from the previous page is the explanation of the fourth color band. This color band represents the tolerance of the resistor. Most of the time, you don't really worry about this color band, but we are going to explain it for your educational enrichment. It turns out that it is really hard to manufacture resistors to an exact resistance, so we have to live with approximate numbers. The fourth band tells you how close you can expect the given resistance to be to the actual resistance. In the above example the Gold band means 5% tolerance which means the actual resistance is 1000 ohms + or – 5%, so somewhere between 950 and 1050 ohms.

The other thing you might be asking yourself, is why go through all the trouble of a complicated color code, why not print the number on the resistors! Well, printing color bands on tiny cylinders is a lot easier than printing numbers, and reading little color bands is a lot easier than trying to read tiny numbers (unless you are color blind, which is a major drawback of the color code system).

Although they are rare, there exist some resistors with 5 bands. Three digits, a multiplier and a tolerance. These resistors are just a little more specific about their resistance value.

There exists another convention for printing values on components which is very similar to the color codes. Instead of using three color bands they will actually print three numbers on the component. So a resistor with a value 102 written on it would be a 10 with a multiplier of 2, or 1000 ohms. This is more common in the smaller surface mount resistors, and on capacitors, whose value is written with this three number convention in picofarads.

Kirchoff's Laws

In most cases, you are either given the I-V plot, or you have to figure it out yourself. Once you have I-V relationships for all of your circuit elements, you have to figure out how a whole circuit works, which inevitably involves some connection of multiple circuit elements. Generally, you'll have a voltage source (such as a battery), and several elements which hook up in some arbitrary way. To figure out how you can use the I-V relationships of each element together, we have to use two laws, known as Kirchoff's Voltage Law (KVL) and Kirchoff's Current Law (KCL), and refer to a drawing of how the elements are connected, called a schematic.

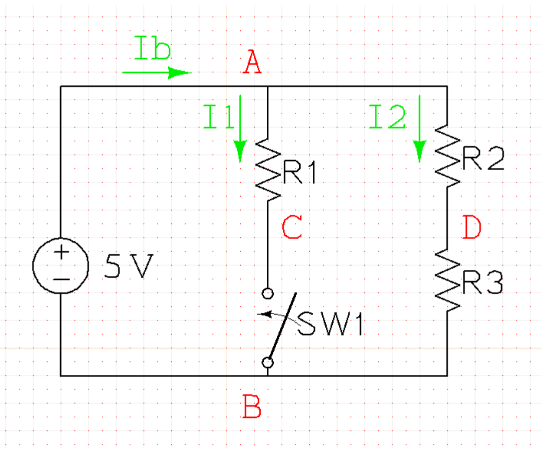
But first, there are two important concepts you need: the idea of a node, and the idea of a ground. A node is any point in the circuit where circuit elements connect. These nodes are typically wires and points on your breadboard, and that entire collection of wires and points is considered to be one node until you hit another circuit element.

Ground is just a particular node we choose to be the reference for all of the other nodes. (As we discussed before, the only thing that matters in analyzing a circuit is relative voltages. The concept of absolute voltage has no real meaning -- it must be defined with respect to some reference.) In general, for digital circuits, we choose ground to be at the negative terminal of the battery.

The KVL says that every node has an associated voltage (with respect to ground), and that these voltage drops add together as you go around the loops. If we pick two nodes, the voltage difference across them will be the same regardless of which path between the nodes we pick. For example, consider the circuit shown here, with a battery (voltage source), 3 resistors, and a switch. The KVL tells us that the voltage from A to B is the same all three ways:

$$V_{AB} = 5 \text{ volts}$$
$$V_{AB} = V_{R1} + V_{SW1}$$
$$V_{AB} = V_{R2} + V_{R3}$$

The subscripts mean “voltage from A to B”, or “voltage across R1”. (In the future, we may just say “voltage at node A”, and it's assumed that we're talking about it with respect to ground.)



switch open: $I_1 = 0$

switch closed: $V_{SW1} = 0$

The KCL says that at every node, the sum of the currents going in must equal the sum of currents going out. This is because you can't have electrons piling up at one place or another. (This is basically because like charges repel each other very strongly, so they won't want to pile up at any given point.) For this circuit, we can apply the KCL at node A or B, but in either case we would find:

$$I_b = I_1 + I_2$$

Once you've written out all these equations, plus the equations you had before from the I-V relationships for all of the elements, you can figure out the voltage of every node and the current going through every element. It's just an algebra problem.

If all this seems hard, it's because it is. This is the foundation of electrical engineering so don't be discouraged if you don't understand it all. Just try to keep some of it in your head as you move on with the NerdKits projects and hopefully you will begin building the intuition about how circuits work. As always, if you have questions email support@nerdkits.com and we will do our best to help you.

Three Terminal Devices: The MOSFET

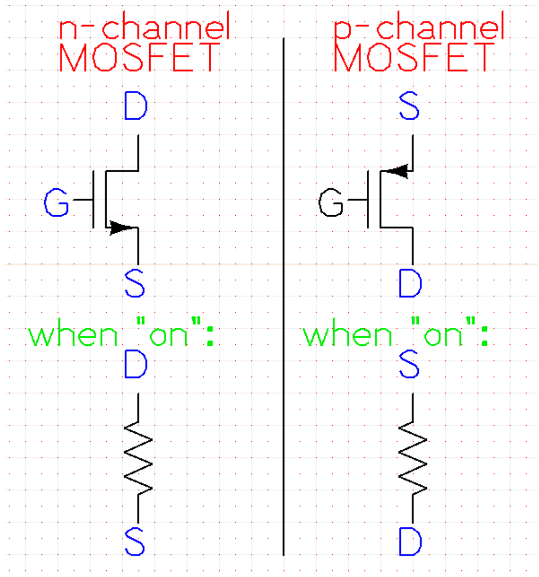
Until now, we've focused on two-terminal devices: they have only two nodes, so there is only one voltage and one current to worry about. However, by introducing the switch on the previous page, we hinted at another kind of device: one where the I-V relationship of two nodes is controlled by a third terminal.

Transistors are circuit elements which use one node to control the behavior at another node. For digital circuits, the most important kind of transistor is a MOSFET, which stands for Metal Oxide Semiconductor Field Effect Transistor. That's a lot of terminology to handle at once, but the basic idea is that a voltage between two of the nodes can switch the device on and off. This is an extremely powerful idea, and MOSFETs have enabled the modern digital world, because we can shrink them and put millions or even billions into a single chip.

There are two kinds of MOSFETs: n-channel, and p-channel. Each has three terminals: a Gate (G), a Source (S), and a Drain (D).

When the voltage between gate and source exceeds a certain level, the the drain and source behave like they're connected by a very small resistance (as shown). When the voltage between gate and source is small, there is no current allowed to flow between the drain and source.

Essentially, a MOSFET looks like a mechanical switch, having a very small resistance when ON and a very high resistance when OFF. But instead of being controlled by a mechanical lever, it's controlled by a voltage.



The difference between n-channel and p-channel devices is the polarity of the switching action:
For an n-channel MOSFET, the switch is ON when the gate is more **positive** than the source.
For an p-channel MOSFET, the switch is ON when the gate is more **negative** than the source.
In reality, there's a certain “threshold voltage” which determines where the on-off switching happens.

There are a few other things you may want to know about, but don't really need for now. First, MOSFETs aren't so simple in reality: we've told you about the “triode” mode, but there is another called “saturation” which is much more important in analog circuits. Second, there are other kinds of transistors. In particular, Bipolar Junction Transistors (BJTs) are very common in analog circuits and tend to have much higher amplification than MOSFETs, but have some disadvantages as well.

Why Digital?

The world around us is analog: it's not ones and zeros, or black and white, but infinite shades of gray. Why, then, do we have digital electronics? The answer is simple: NOISE.

Every circuit element has noise, whether it's because of electrons randomly colliding with atoms in a resistor (thermal noise), or because every current is really made up of a finite number of discrete particles (shot noise). Additionally, there can be other undesired influences like crosstalk or pickup – where one voltage in a circuit unintentionally and undesirably changes when a nearby voltage does. At some level, all wires are antennas, and our signals pick up and transmit radio interference.

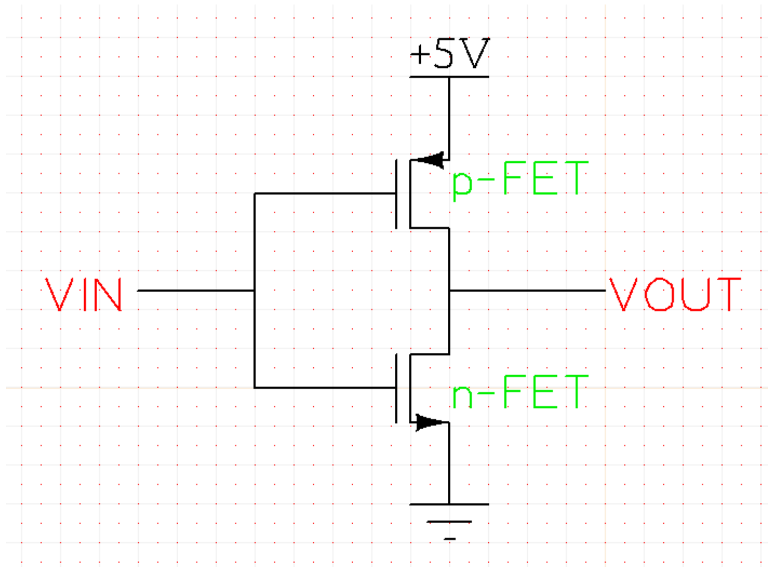
By making everything digital, we can remove this noise (as long as it's below a certain level). For example, when we transmit bits (1s and 0s) as voltages, we might decide that 0 volts means logic zero, and +5 volts means logic one.

If we're transmitting a logic one, and we get some noise added in, we may end up with +4.5 or +5.5 volts instead of the +5.0 that we desired. But, since this is digital logic, we can regenerate the original signal, since even with the noise we know it's supposed to represent a logic one.

For any digital logic, you can find the input and output voltage levels, as well as something called the "noise margins". This is the level of noise which the circuit can tolerate without mistaking ones for zeros and vice-versa.

The CMOS Inverter

Using real devices like the n-channel and p-channel MOSFETs we discussed earlier, we can build logic devices. The simplest logic device is an inverter, sometimes called "NOT", whose output is the opposite of the input.



Remember the model of the MOSFET we discussed earlier? When there's a large V_{GS} voltage between gate and source, the drain and source look like a closed switch, while when V_{GS} is near zero, the switch is open. (Remember that the source is the terminal with the arrow on it in the schematic.)

Therefore, when VIN=0, the p-FET is closed and the n-FET is open, so VOUT=+5V. And when VIN=+5V, the n-FET is closed while the p-FET is open, so VOUT=0.

That's why this circuit is an inverter.

IN	OUT
0	1
1	0

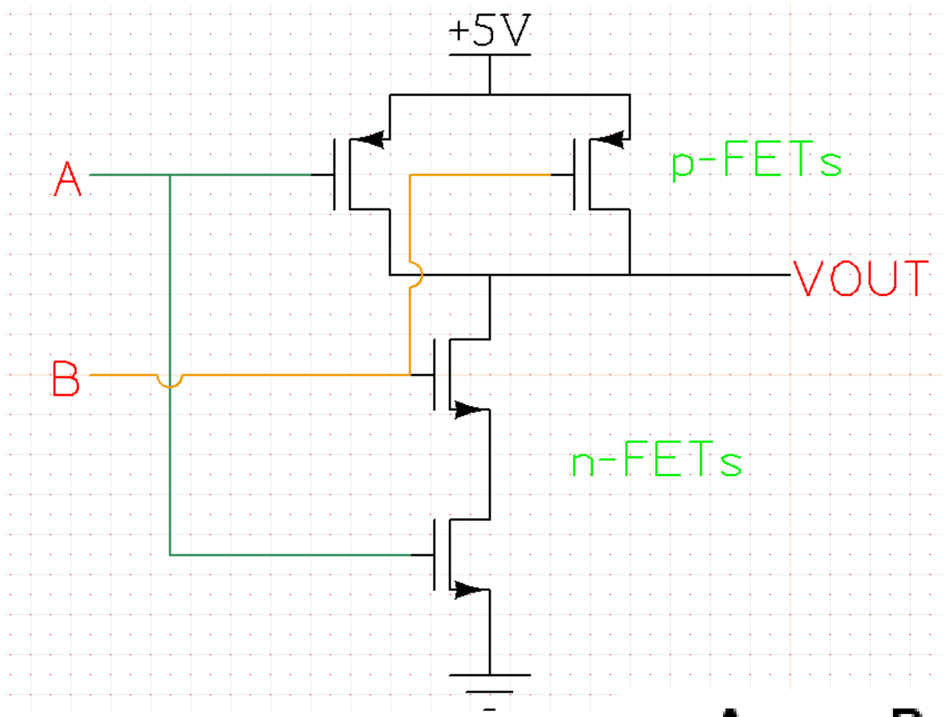
Don't get confused! A CLOSED switch is one that allows current to flow (because its contacts are closed against each other), while an OPEN switch does not allow any current to flow (because its contacts are open and unconnected).

This chart is called a "logic table". It describes the input-output relationship, not in terms of voltages, but in terms in logical ones and zeros.

This is the simplest logic table, and others will include multiple input and/or multiple outputs.

The CMOS NAND Gate

If we go beyond the simple case which had just one input and one output, we can make much more interesting logic gates. This one uses four transistors and is called a "NAND" gate, or "NOT AND".



If you go through the same analysis, considering the voltage for each MOSFET's gate to source, you'll see that the path from VOUT to GROUND is only closed when both A and B are logic one. Otherwise, the VOUT output node is connected to the +5 supply voltage so we get a logic 1.

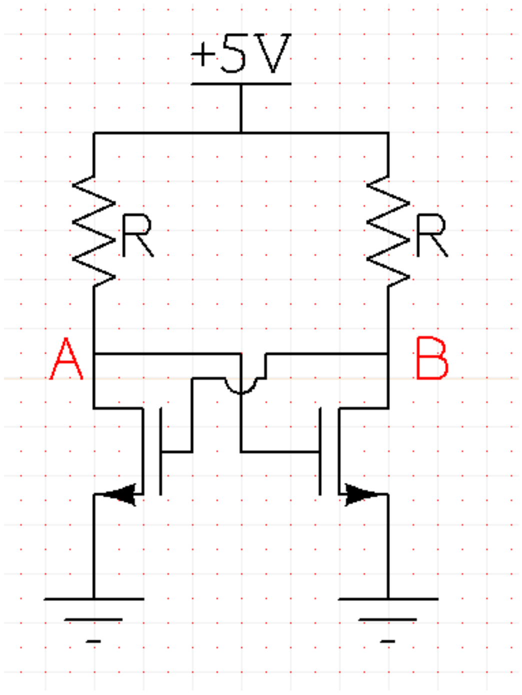
A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

This logic table has an extra column to show the extra input, and it describes the output for all possible combinations of the inputs. Can you see how it matches up to both the description and the circuit itself?

One cool result is that using only two-input NAND gates, we can build any possible logic function by combining them in the appropriate manner. Theoretically, this means the NAND gate is "universal". In fact, the entire Apollo Guidance Computer – the first computer ever to use integrated circuits – which kept astronauts on course toward the moon and back was built using only NOR gates (NOT OR).

Circuits with Memory

So far, we've only considered circuits where the output is purely a function of the input. But to build a computer, we need memory – the ability to remember whether we had a 1 or a 0. The circuit below shows one of the simplest examples of a single-bit memory cell.



While it's not clear where the input and output is, the essential point of this circuit is that there are two stable operating points:

A can be 1, and B is therefore 0, or

A can be 0, and B is therefore 1.

Because there are TWO, and not just ONE point where the circuit can be, this acts as a memory cell!

There are some missing details, like how to set the memory cell's value, but basically this demonstrates that a circuit can hold a bit. This is an example of positive feedback: it's basically two inverters, back to back, so any noise quickly gets removed and the signal is restored.

Binary!

Now we have a way of storing 1s and 0s arbitrarily, but have gained a lot more than just that. We now have a reliable way to represent data, store it, manipulate it, and send it! More specifically we have a way to represent numbers, and manipulate numbers. All we have to do is get used to representing the numbers we normally use using just 1s and 0s. We call this numbering system binary numbers.

The way we do this is very similar to the way we already think about numbers. The only difference is that binary is in base 2 and ordinary numbers are base 10. This means that there are only 2 digits instead of the 10 you are used to. However the way of thinking about the numbers is the same. To you, every place in a number represents a different power of 10. For example, a 1 in the first digit means 1, a one in the second digit means 10, a 1 in the third digit mens 100. In binary, since we only have two digits to work with, we have to roll over to the next digit every two numbers. So in binary, 0 still means 0, 1 still means 1, but then we have to roll over to the next digit, so a two is represented with a 1 in the second digit; 10. If we keep counting up, three is 11, then to get to four, we need to roll over both digits and add another one; 100. Can you figure out how you would represent the number 5? What about 10?

This concept is not very easy to understand but it is necessary in computer science because it is the way we represent everything. So it is definitely worth thinking about it if you don't get it right away. As always if you don't get something, we are here to help email support@nerdkits.com and we will try our best to help you.

Once you know binary representation of numbers, we can do the same operations we are used to doing. For example, adding two plus three:

```
         10
       + 11
  --------------------
        101
```

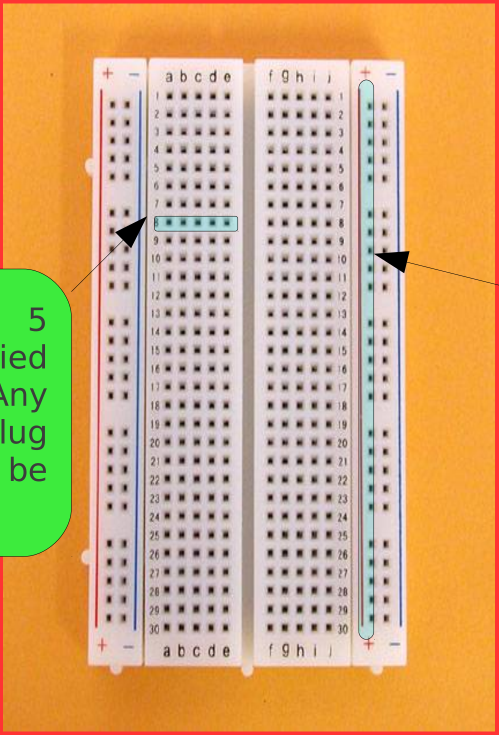
As expected it adds up to the binary representation of five. Notice how addition works the same as with base ten numbers: if a column fills up you carry it over to the next one.

This concept can be tough for people not used to thinking in binary. For an extra tutorial, check out our Bits, Bytes and Binary video tutorial on our website.

The Breadboard

The breadboard is the place where we build our circuits. The best thing is that you can add and remove components easily, so you are free to explore. All components in your Nerd Kit (unless we say otherwise) can be added to the breadboard by just pushing them into the tie points (the little holes). You shouldn't have to use too much force – be gentle and wiggle a bit if you're having trouble.

The purpose of a breadboard is to connect things without having to solder wires together. A breadboard has rows of "nodes" that are connected together under the plastic.



Each middle row has 5 interconnects that are tied together as one node. Any two (or more) things you plug into this row will be electrically tied together.

The breadboard comes with two long vertical nodes on either end usually referred to as the rails. Each of these vertical columns is connected as one node. These are typically used to bring power to the rest of the board. They are usually connected to the positive and negative ends of your power source, matching the "+" and "–" headings on the columns.

There are a few common methods for physically building circuits. The one you're probably most familiar with from looking inside computers and other commercial devices is called "printed circuit board" (PCB) construction, which consists of copper traces (wires) attached to a fiberglass board. This is a reliable way of building circuits, and PCBs are relatively cheap to mass produce.

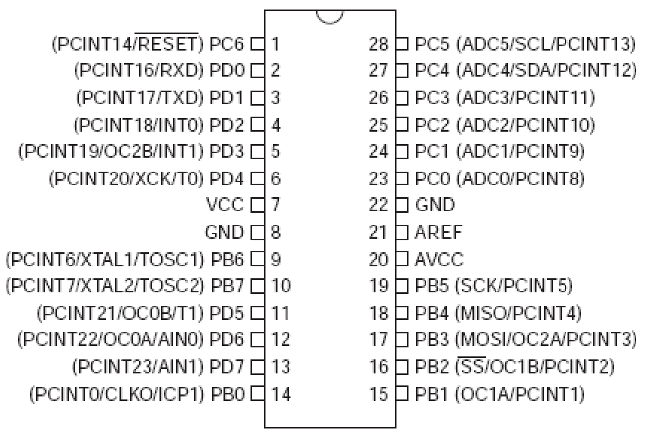
However, for experimentation, solderless breadboards are much more popular. While more expensive, they make it very easy to change a circuit. There are two disadvantages: first, that wires can wiggle loose and become disconnected, and second, that there are undesirable electrical properties. These properties are called "parasitics", and in particular, there is a "parasitic capacitance" between breadboard rows. This can be a big problem for high-frequency circuits, because that capacitance has to be charged and discharged very quickly, even though that's usually not desired.

The Microcontroller (MCU)

The microcontroller is the brain of the kit. It's basically a tiny computer. It has a processor, memory, inputs and outputs. It will take the instructions you give it (programming) and execute them, one by one, in less than a millionth of a second each. Much like a computer it has pins on which it can read voltages as inputs, and pins on which it can set voltages as outputs.

The flexibility you get from a device like this is at the core of embedded devices. Just imagine what you can do when you can control most other devices in any way you want, and re-program it when you need it to do something different. Microcontrollers like the one in your kit are what power most modern digital devices like cell phones, MP3 players, digital cameras, and video game systems.

From this point forward we might refer to the microcontroller as the MCU if it is more convenient.

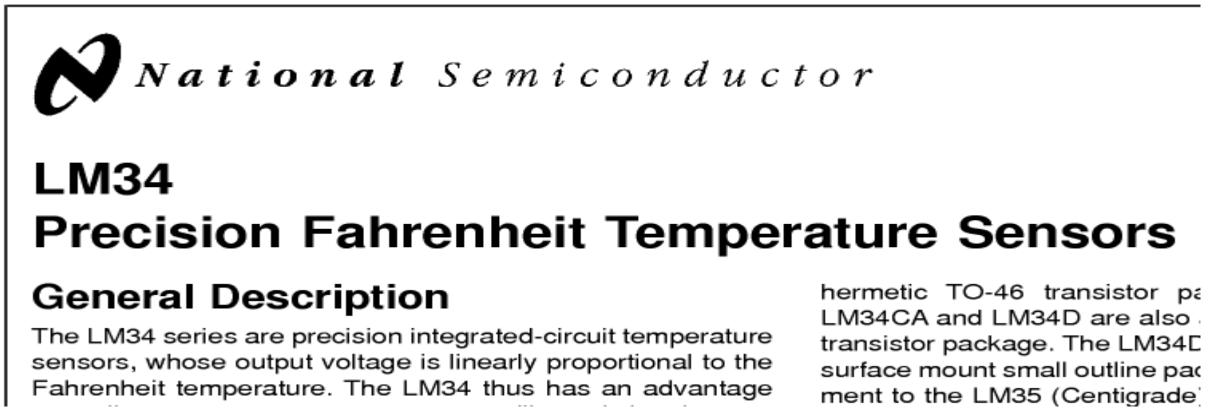


This is a pin-out of the microcontroller in your kit, just like the one you would normally see on the official data sheet. Don't worry about all of the labels – we will walk you through everything soon. However, you might be able to recognize some names like GND (ground) and VCC (power supply).

Your microcontroller is an ATmega168 manufactured by Atmel. For more information you can check out the full datasheet linked from the downloads section on our website. (You can get to your downloads from the members area.)

The Temperature Sensor

The electrical engineering field is full of sensors that you can use to get information about the outside world. One useful piece of information you can get is the temperature. This sensor provides a voltage on one of its pins that goes up linearly with temperature. You can use this voltage as an input to the microcontroller, as we will in this kit. However such sensors can also be useful in other places, like in analog circuits.


National Semiconductor

LM34
Precision Fahrenheit Temperature Sensors

General Description

The LM34 series are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Fahrenheit temperature. The LM34 thus has an advantage

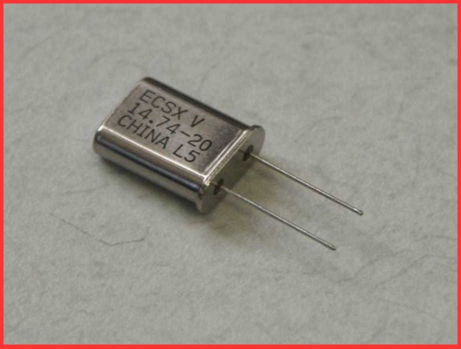
hermetic TO-46 transistor pa
LM34CA and LM34D are also
transistor package. The LM34D
surface mount small outline pac
ment to the LM35 (Centigrade)

The data sheet of a component is an engineer's best friend. It gives all the information about what the part is intended to be used for, and how it behaves under different conditions. All good engineers keep the data sheet handy when working with any integrated circuit. Luckily, the data sheets to most components can be found on the internet. Usually, a web search for the part number will point you in the right place.

The data sheet for the temperature sensor is included in your downloads section. You can always just Google for “LM34 datasheet” (it's what we do).

The Crystal Oscillator

Every computer has a clock. Your computer at home probably runs at something like 2 Gigahertz. The crystal oscillator for your NerdKit is a 14.7456 Megahertz crystal. That means that there are 14,745,600 instructions executed every second! That is not quite the 2 billion your PC can do, but its still enough to do interesting stuff.



Inside the metal can is a little thin piece of quartz grown so that it vibrates at exactly 14.7465 MHz. Just like a tuning fork is cut just right to oscillate at a certain frequency, this produces a very reliable clock signal that the chip follows when executing instructions.

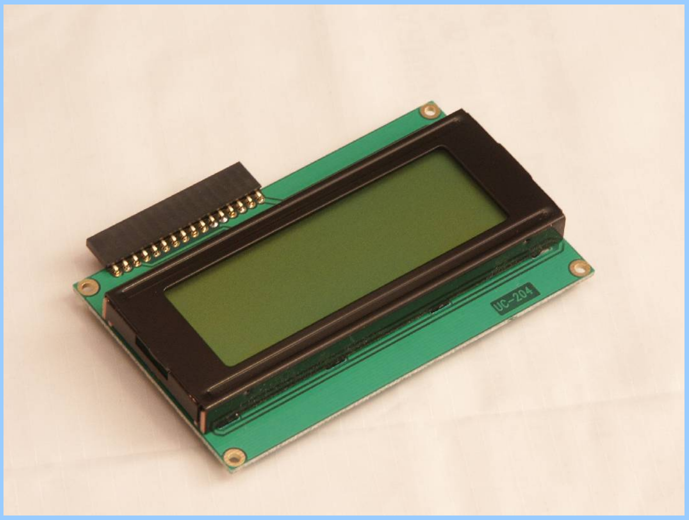
The reason we need a very precise clock signal is because many communication interfaces including ethernet, USB, and serial rely on very accurate timing to agree on when one bit ends and when the next bit begins.

Note: The crystal oscillator can come in a slightly different looking package. It is still a metallic can, it's just shorter! This crystal is exactly the same functionally, but the manufacturer just makes slightly different packages.

The LCD

The LCD in your NerdKit is a four line tall, 20 character wide LCD. It is the most fragile part in your kit, so you want to take the most care of it.

We will be using the microcontroller to drive a signal into the LCD and tell it what to display. The actual communication is complicated, but in the end, it's all voltages being driven onto wires. Once you understand how they communicate, you can make the LCD display anything you want!

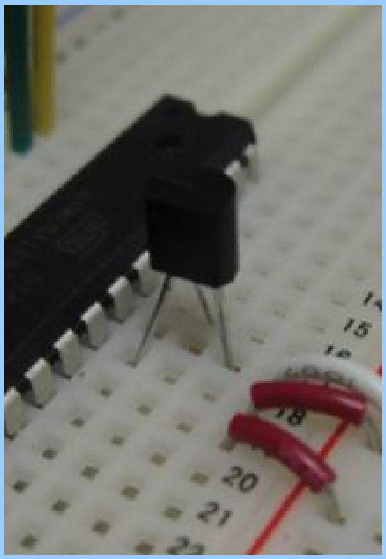


Almost all of the character LCD displays available use the same protocol, and this is typically referred to as the HD44780 protocol. This defines the commands and the timing used to control the display. In our projects, we've done most of the work implementing this LCD protocol in the lcd.c file, which you'll get to use very shortly. However, if you're curious about the protocol, look in the source code, or look on the internet for information about the HD44780.

Your First Project: Temperature Monitor

Now that you are familiar with most of the parts in your kit, it's time to begin your first project. You'll assemble the basic kit, and will then program it to read an analog voltage from the temperature sensor, and display the result on the LCD.

Remember, the point of these instructions is to give you an idea of the types of things you can do with circuits and microcontrollers. It is a very basic project, and with it you should be able to gain the knowledge you need to undertake your own projects. Think of this as a building block toward better things, and always remember that the most interesting projects are those that you have yet to imagine!



Parts List

Your kit includes extra parts that you will not need for this project. These are there to let you explore beyond the temperature sensor project. The parts you will need for the temperature display are listed here.

- Breadboard
- Microcontroller
- LCD
- Temperature Sensor (LM34)
- Voltage Regulator (7805)
- Crystal Oscillator
- SPDT Switch
- Wires
- 0.1uF capacitor
- 1K Ohm resistor

The basic setup of your breadboard, microcontroller, and LCD that will follow is not specific to the temperature display project. It's recommended that you keep this wiring in place as you develop other projects.

Quick Tutorial – Stripping Wire

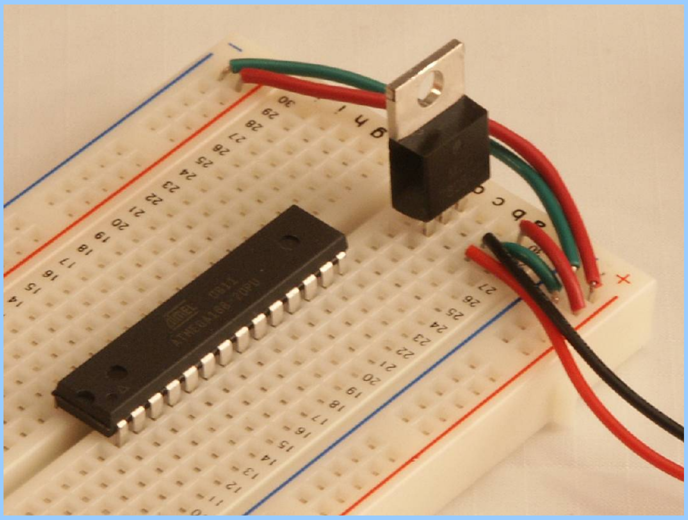
One thing you have to learn to do well is strip wire. Wires are really just a piece of metal covered in a thin coat of plastic. The insulation is there so that the electricity running through the wire doesn't shock you if you accidentally touch it, but more importantly so that it doesn't accidentally connect to other places in your circuit that it shouldn't. To connect things together, you need to remove a little bit of the plastic at the end to expose the metal. Usually you only want to take off enough to be able to stick the exposed part into the breadboard – about 3/8” or less.

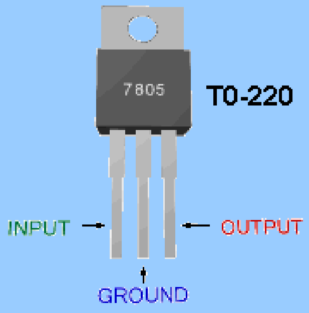
It takes practice to learn how to do it quickly. Using your wire strippers, squeeze the sharp part where you want to cut the plastic, twist a little, then pull out along the wire to separate the plastic and slide it out. The real trick is doing this quickly without cutting the metal inside the plastic. Adjusting the wire strippers to the right width is also key -- check out or stripping wire tutorial on our website.

In a pinch, you could do this without wire strippers. A simple knife will do the trick, but it is a lot easier and less dangerous with the right tool.

Step 1 – Set up Power to the Breadboard

1. Connect the left and right rails of the breadboard with two pieces of wire, connecting red to red and blue to blue. We recommend you do this across the bottom of the breadboard to give you more space later.
2. Plug the voltage regulator (7805) into the breadboard on the left side. Plug the input pin into row 28, the ground pin into row 29 and the output pin into row 30.
3. Make sure this is correct, or else it won't work!
4. Then using wire (which you'll have to strip) connect row 29 of the breadboard to the blue (ground, "-", GND) rail.
5. Now connect the red wire of the battery clip into row 28.
6. Connect row 30 of the breadboard to the red rail.
7. Connect the black wire of the battery clip to the blue rail.
8. ***Do not connect the battery yet!***



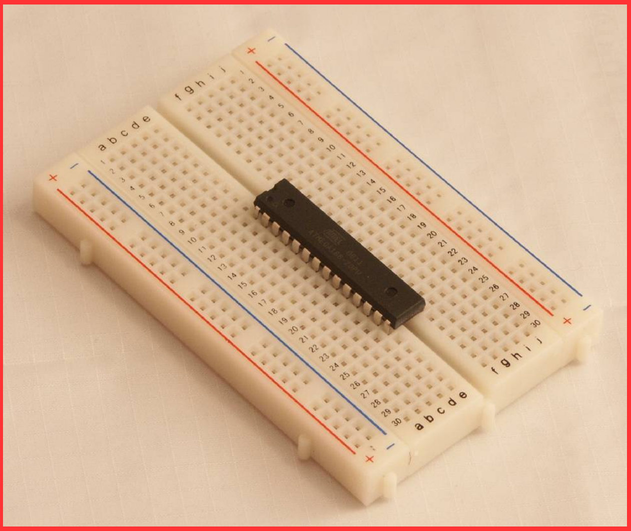


Now that you connected things to the breadboard, we think it's useful to know why you connected things the way you did. First, we bridged the left and right rails of the breadboard with wire. This is because we want both sides of the board to be connected to the same power supply. The 9 volt battery will be connected to the battery clip, and that will be your power source. Notice that we are passing the battery through the voltage regulator and then to the positive (red) rail. This is because most of the electronics in the kit need 5V to operate, and you have a 9V battery. The regulator takes any voltage above 5 volts and converts it down to a very steady 5. So, after you are done with this step, you will be passing 9V into the regulator and getting out 5, which you have connected to the red (positive) rail, and connected the ground pins to the blue (GND) rail. Now you have a clean 5V running down both sides of the board that we will use to power the circuit. (If you later decide to use a DC wall transformer for your NerdKit, it's important that you supply at least 7 volts DC to the input of the voltage regulator. See the 7805 datasheet.)

You might be curious as to what impact the voltage regulator has on power... Can you figure out what happens to the voltage that's dropped between the input and the output?

Step 2 – Plug in the Microcontroller

The microcontroller should already be plugged into your breadboard. Make sure it is plugged in starting at row 11 bridging across the two sides of the breadboard.

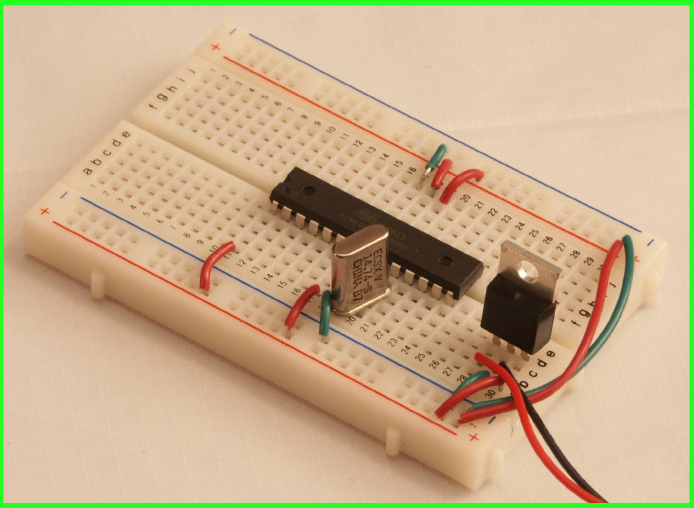


From now on, we will be referring to the pins on the microcontroller by their number using the standard way of numbering pins on integrated circuits. If you look down at the chip with the little half-circle indentation facing up, then pin 1 is on the top left. Then, we count up counterclockwise all the way around. This means pin 2 is just below pin 1, and it continues around to pin 28 on the top right. You can refer to the microcontroller page earlier in the instructions for a picture.

It is very important that you get the orientation of the microcontroller correct on the breadboard. The little indentation at the top is there to help you know which side pin 1 is. You have to put the microcontroller in bridging the center gap, because the five interconnects on each row are connected together as we mentioned before. If you didn't put it across the middle, you would be connecting two pins to each other, and you usually don't want that. Now that the microcontroller is in place, you can connect things to it by plugging them into the same row on the breadboard, or use wires to connect different rows together.

Step 3a – Wire up the Microcontroller

1. You will need to cut and strip 6 short wires for this step.
2. Using wires, connect pin 7 of the microcontroller to the red (+5V) rail of the breadboard.
3. Connect pin 8 of the microcontroller to the blue (GND) rail.
4. Connect pins 20 and 21 of the microcontroller to the +5 rail (this is on the right side of the breadboard).
5. Connect pin 22 of the microcontroller to the GND rail.
6. Connect the RESET pin (pin 1) to the +5V rail.
7. Connect the crystal oscillator to pins 9 and 10 on the MCU. You may want to trim the leads on the oscillator so that it sits closer to the breadboard when you plug it in.



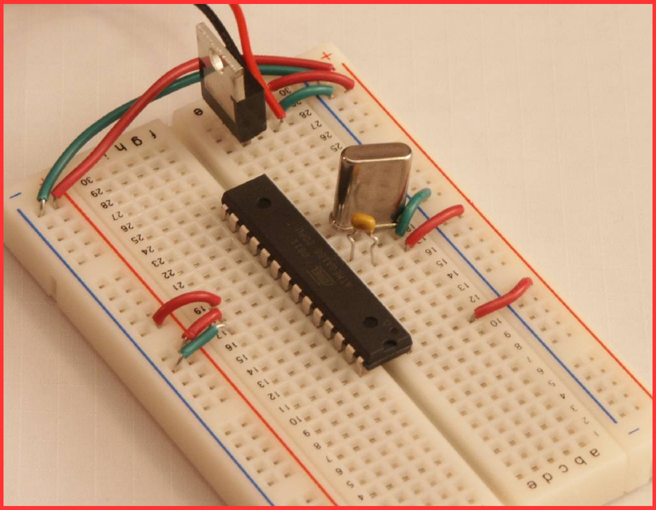
Remember when you bridged the two rails across the breadboard with wire, you connected them together, so you should have a +5 (red) and a GND (blue) on each side. You can choose to connect the microcontroller pins to either side you want, but the closest one is usually a good idea. So the best thing to do would be to connect pins 7 and 8 to the left side rails, and 20, 21 and 22 to the right side rails.

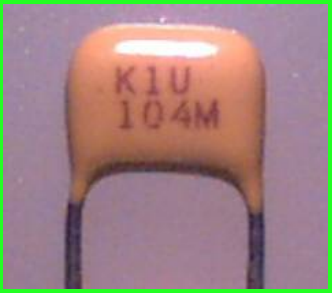
If you refer to the microcontroller pin names earlier in this guide, it should be obvious why we connected VCC and GND as we did. However, you'll notice that pin 20 is actually named “AVCC”. This refers to the power for the Analog to Digital Converter (ADC) inside the microcontroller. Since the ADC deals with analog voltages, it is particularly sensitive to noise. Therefore, in some applications, it will have a specially regulated power supply, so a seperate AVCC pin is present. For our purposes, we'll just connect it to VCC as usual.

Step 3b – Wire up the Microcontroller

Inside your parts bag, you should see three yellow capacitors. The largest one is 0.1uF, or 0.1 microFarads.

Insert the 0.1uF capacitor between pins 7 and 8 of the microcontroller (+5V and GND). You may want to trim the leads of the capacitor so that it sits closer to the breadboard when you plug it in.



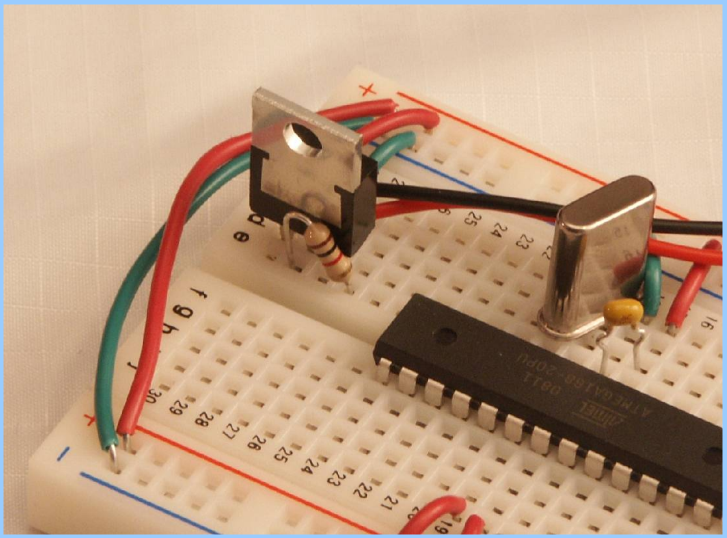


A capacitor is a device that stores energy in an electric field. It stores opposing electric charges on its terminals, which maintains a voltage across it. We're using it here as a "bypass capacitor," to help smooth out the power supply voltage to the microcontroller. This helps keeps the power supply voltage steady even when there are sudden changes in the demand for current.

The microcontroller, like any digital circuit, operates in discrete clock cycles, which generally means that there is a sudden demand for power at the beginning of every clock cycle, when all the bits are flipping. While on average the microcontroller doesn't use much power, these spikes in demand often can't be reliably provided by the battery and voltage regulator alone. Adding this capacitor ensures proper operation.

Step 4a – Wire up the LCD

1. In order to set the contrast of the LCD correctly, we must put a resistor between its contrast pin and ground. We have included a 1K ohm (1000 ohm) resistor in your kit for this purpose.
2. In order to save space on the breadboard we are going to pack the resistor in right next to the voltage regulator and use pin 27 of the breadboard to bring the node out to the LCD.
3. Grab your resistor and stick one leg in row 29 of the breadboard (the same row as the GND pin of the voltage regulator).
4. Stick the other end into row 27 of the breadboard. Connect a wire between row 27 of the breadboard to pin 3 of your LCD. (see the next page for LCD pin numbers, and the rest of the LCD connections)

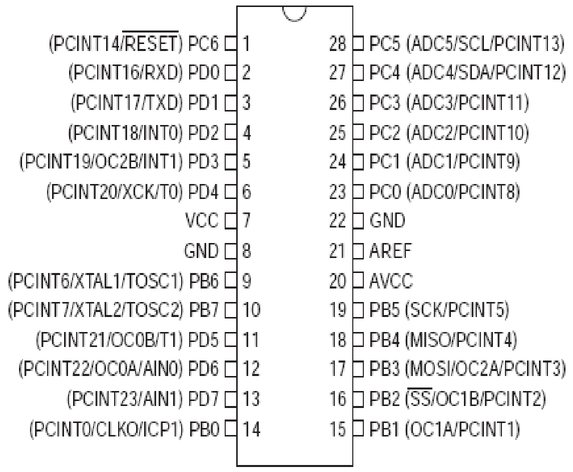


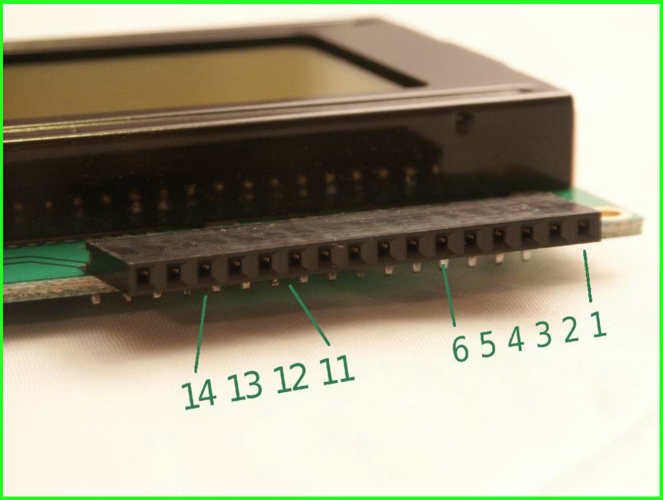
When you bend the resistor it is a good idea to make sure the lead that sticks up is the side connected to GND the way we did in the picture. This is because the metal tab of the voltage regulator chip's package is internally connected GND and we don't want it to short if it were to touch the other lead. This way it doesn't matter if the lead touches the metal because it is connected to GND anyway.

Be careful when picking out the resistor. There is a very small difference in the color bands between 1K ohm and 10K ohms. Make sure you refer back to the resistor color section or have a resistor color chart handy.

Step 4b – Wire up the LCD

1. Now its time to wire up the LCD. Make sure you double check all the connections against the LCD pin out, and the MCU pin out on this page.
2. Using wires make the following connections:
    1. Connect LCD pin 1 to GND (the blue rail)
    2. Connect LCD pin 2 to +5 (the red rail)
    3. Connect LCD pin 3 to the contrast resistor. See step 4a above.
    4. Connect LCD pin 4 to MCU pin 13
    5. Connect LCD pin 5 to GND
    6. Connect LCD pin 6 to MCU pin 12
    7. Connect LCD pin 11 to MCU pin 4
    8. Connect LCD pin 12 to MCU pin 5
    9. Connect LCD pin 13 to MCU pin 6
    10. Connect LCD pin 14 to MCU pin 11

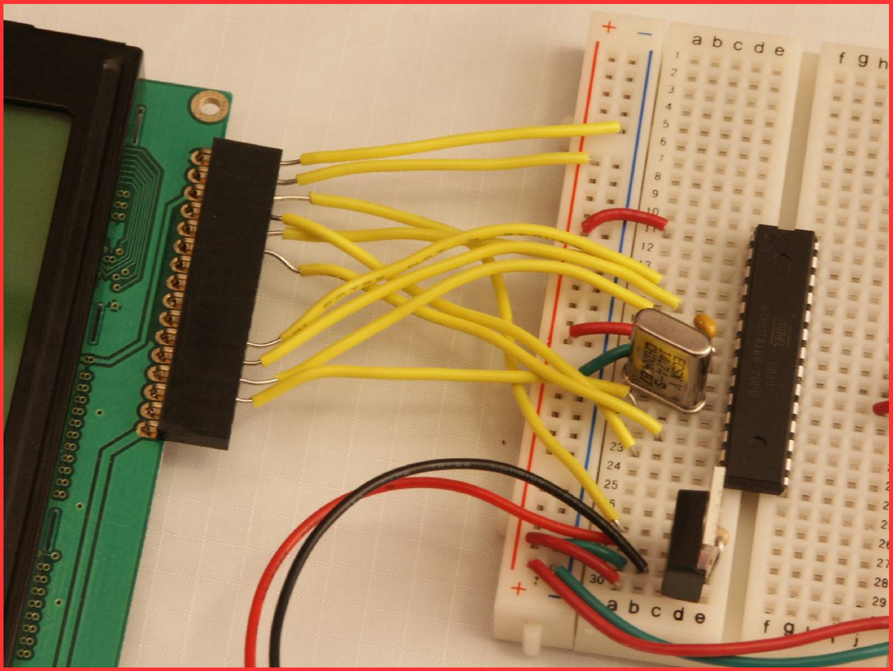




The resistor we put between the GND rail and pin 3 of the LCD is used to set the contrast of the LCD. We used a resistor because we are not very interested in varying the contrast. If you wanted to, you could hook up a potentiometer between the LCD pin and GND, which would give you the ability to modify the contrast on the LCD.

The purpose of all the wires coming out of the MCU is to tell the LCD what to display. Later, when you write the code to write things to the LCD, the pins you connected the LCD to will be used. They include four wires for data, plus a few signaling lines which tell the LCD when a new "nibble" of four bits is ready.

Step 4c – Wire up the LCD

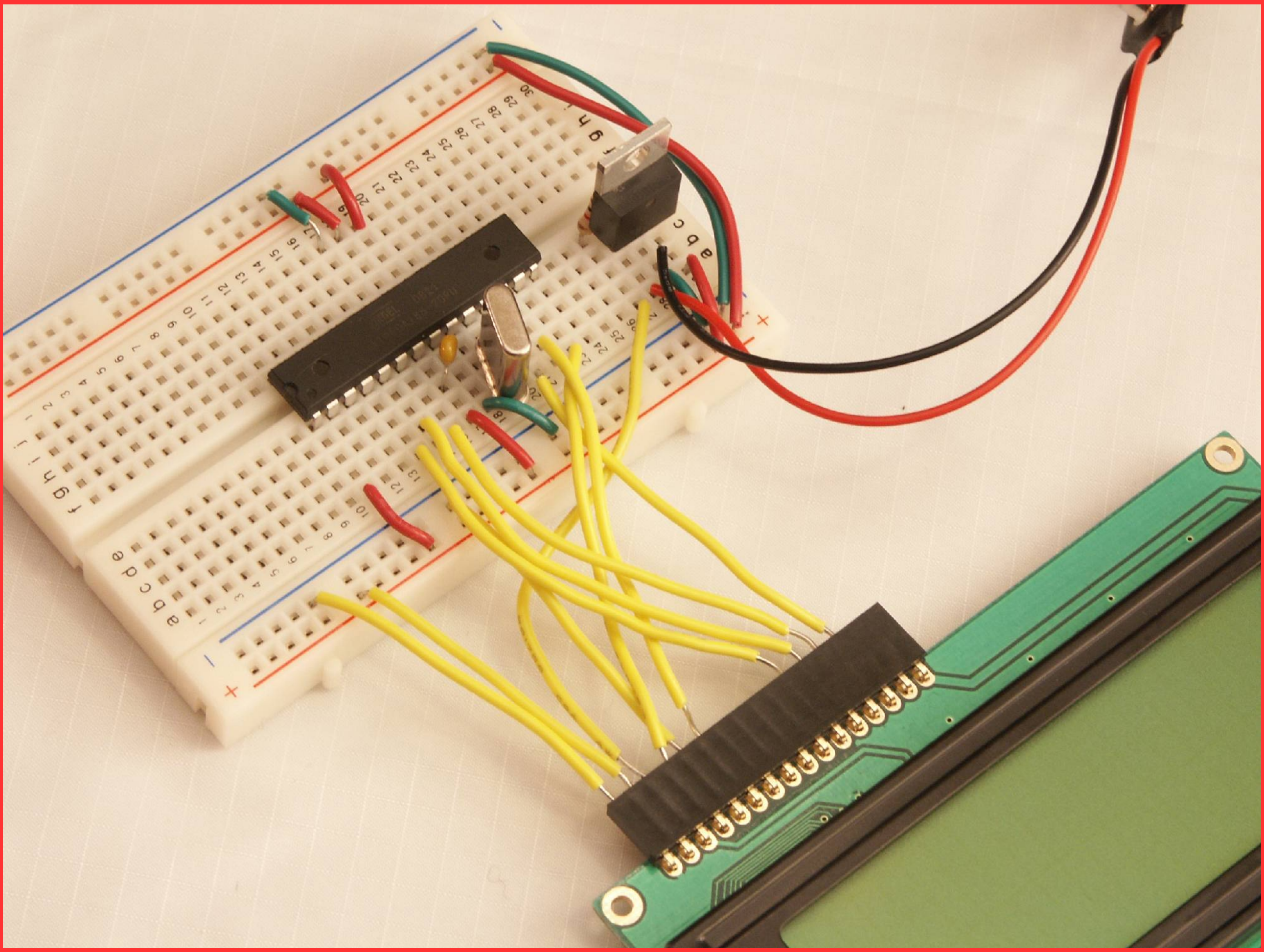


The LCD we include is of a common type called HD44780, which is a standard protocol for sending characters to the LCD. This protocol sends 4 bits of data to the LCD at a time. These data bits are carried on LCD pins 11, 12, 13, and 14. Since a character requires 8 bits to send, you need to send two of these 4 bit “nibbles.”

There are two more wires that go from the MCU to the LCD. On LCD pin 4 is a signal that determines whether you are sending data or a control command (e.g. clear screen, go to position X, etc). On LCD pin 6 is a signal to tell the LCD that a new nibble is ready to read.

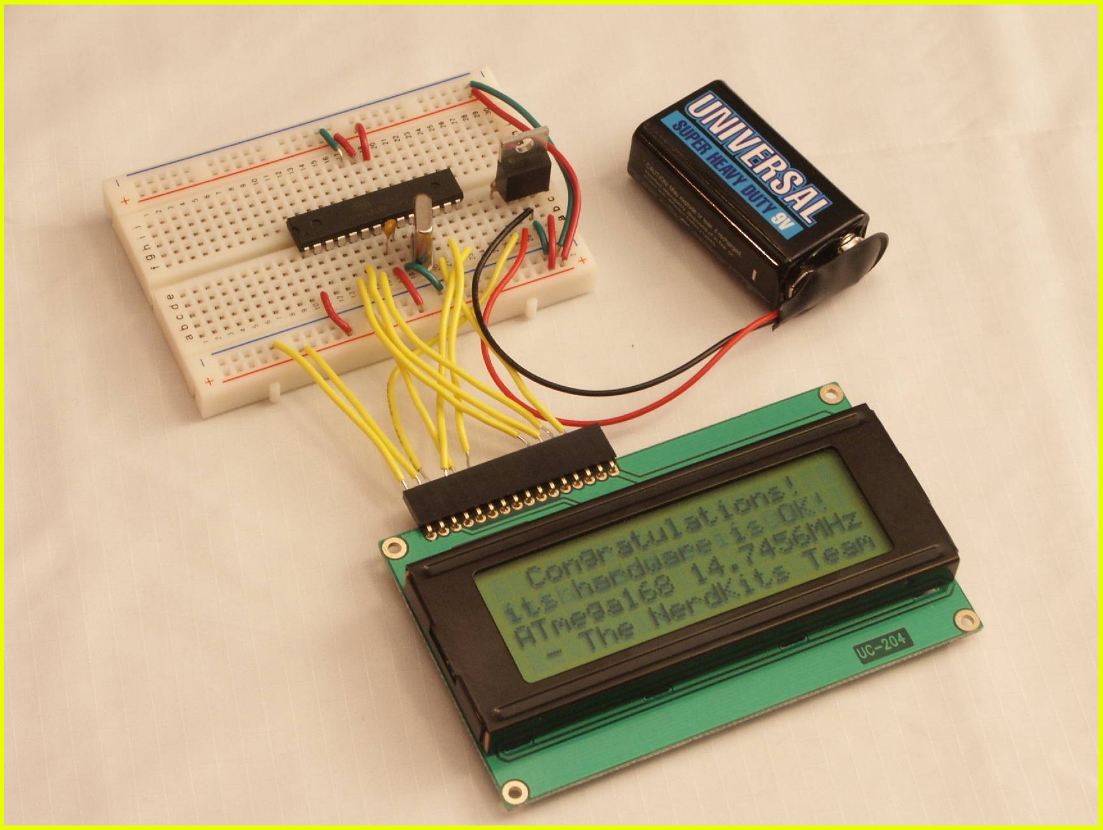
Step 5 – Verify all Connections

1. Make sure you have connected all the pins to the right places. It won't work if you made even one mistake. Go back and double check all the instructions and make sure it looks something like the photo below.
2. Triple-check any connections to VCC or GND.



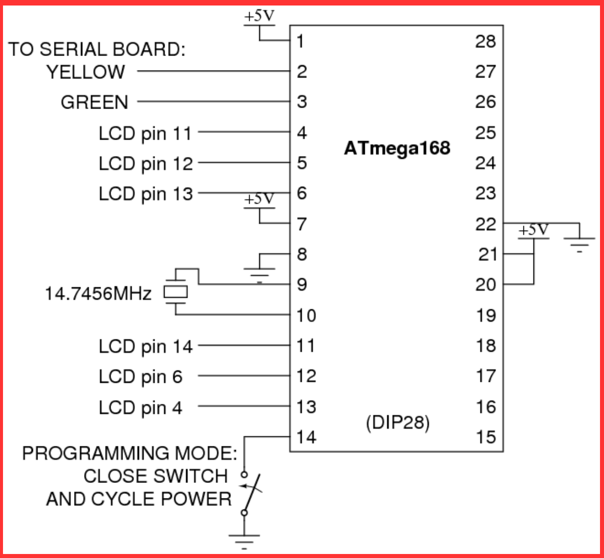
Step 6 – Try It!!!

Plug the battery into the battery clip. This gets power flowing to the chip, and it should turn on.



Congratulations! The first part of
your Nerd Kit hardware is now set up.

It doesn't work! - Troubleshooting



If you are not getting anything on the LCD there are a few very common mistakes.

Symptom: Voltage regulator gets hot. No activity on the LCD.
Diagnosis: You likely flipped VCC and ground at some point. Double check the pinout on this page against your circuit.

Symptom: First and third row of LCD turns on, but no characters are displayed.
Diagnosis: This means the LCD is getting power, but not getting any data it recognizes. One of the wires between the MCU and the LCD is loose or in the wrong place.

Symptom: Everything looks right, but nothing comes on on the LCD.
Diagnosis: It is possible you have the wrong contrast resistor for the LCD. You want a 1K ohm resistor, which means color bands brown, black, red. Remember brown=1, black=0, red=2, so $10\times10^2 = 1000$ ohms

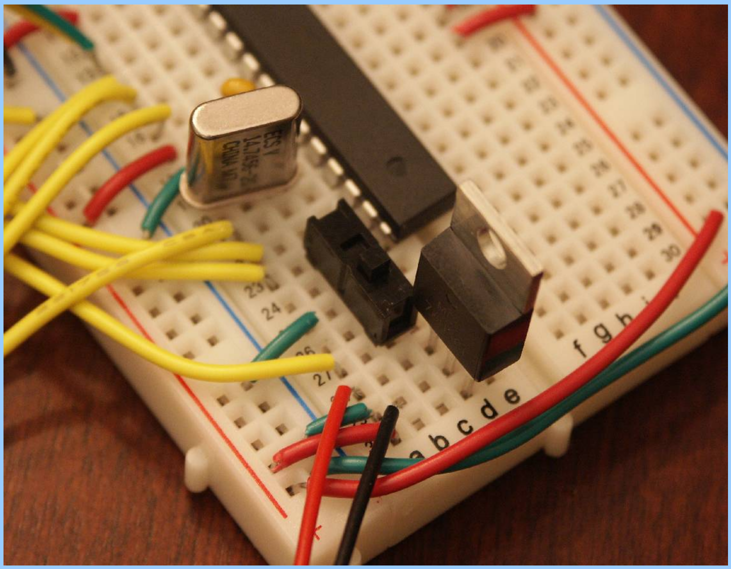
Symptom: Everything looks right, and you checked the contrast resistor but nothing comes on on the LCD.
Diagnosis: Although it is rare, your battery might be dead from the factory. Use a multimeter to check the voltage between the battery leads, which should be very close to 9V. Also check the voltage between your power rails, which should be very close to 5V. If you do not have a multimeter handy try a fresh battery.

If you have tried all these and still have no luck email support@nerdkits.com with a detailed description of your problem. If you can, please include a picture of your circuit so we can try to see what's wrong.

Step 7a – Wire up the Programming Header

To be able to program the chip, we need to be able to connect it to your computer. First we have to set up a switch that will tell the chip whether it should run the program that is already on the chip, or if you are trying to program new code onto it.

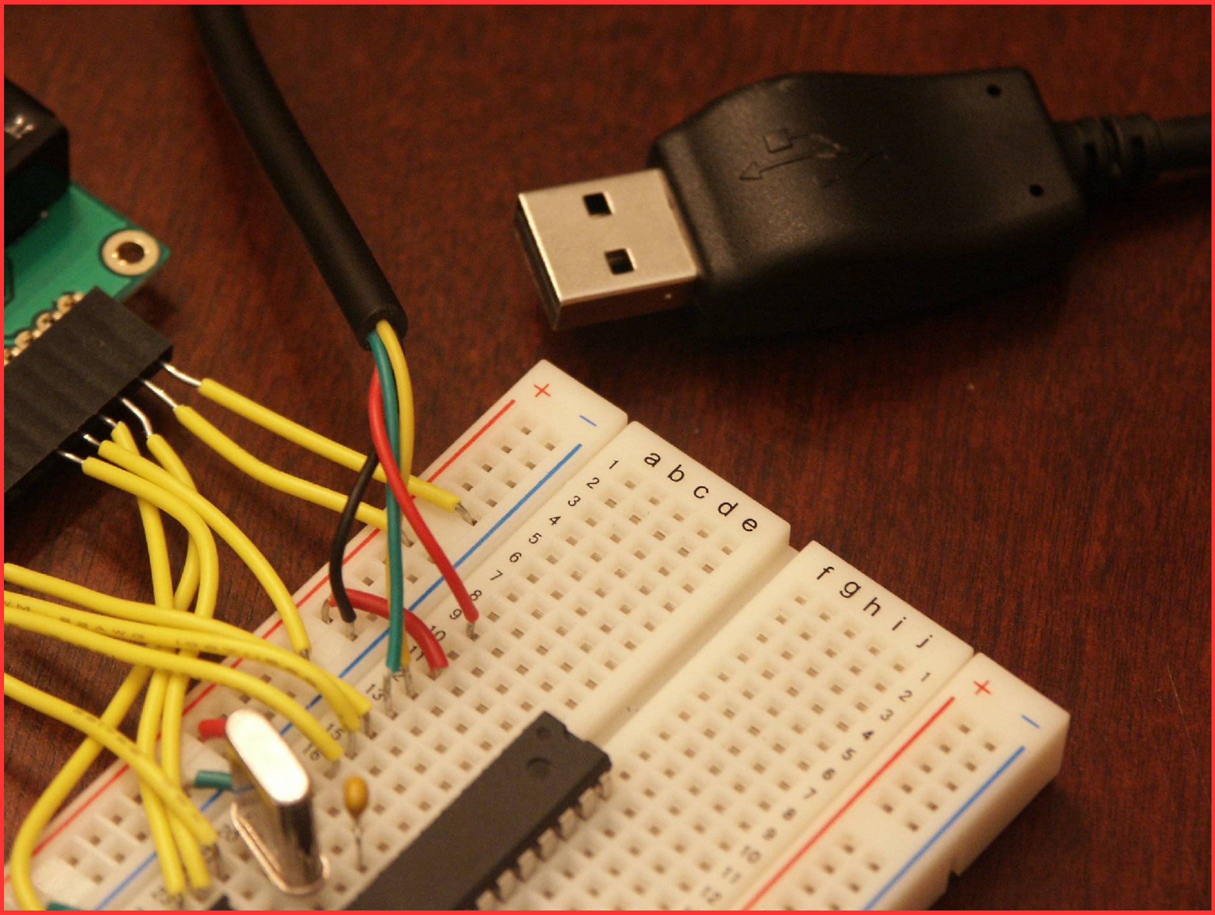
First, take a wire and connect row 25 on the left side of your breadboard to GND. Then take out one of the SPDT (single pole double throw) switches included in your kit. Plug the switch into rows 24, 25, and 26 of the breadboard. The switch will connect pin 14 of the MCU to GND when the switch is up and disconnects it when the switch is down. This will allow us to tell the MCU when we want it to boot into programming mode or to run the program already on the chip.



Step 7b – Wire up the NerdKits USB Cable

Included in your kit is the USB programming cable. Follow these steps to connect it to your NerdKit.

1. Connect the black wire to the blue GND rail.
2. Connect the red wire to row 9 of the breadboard (Row 9 is an empty row -- see appendix B for more about this wire).
3. Connect the yellow wire to MCU pin 2 (row 12 on the breadboard).
4. Connect the green wire to MCU pin 3 (row 13 on the breadboard).



These wires allow your computer to put the microcontroller into programming mode, and then to write your compiled code into the microcontroller's flash memory.

Quick Aside – The NerdKits USB Cable

In order for you to be able to program your microcontroller from your computer, we've included a USB cable that has a little bit of circuitry in it. The cable acts as a USB to serial adapter, which talks to the USB drivers on your operating system and lets you send and receive bytes at a known speed, with one wire for data transmitted from computer to microcontroller, and one wire for data going the other way.

This performs two important functions:

First, it converts from the USB protocol (which many computers have) to a serial port connection (which are becoming increasingly rare).

Second, it makes sure the voltage levels are in the right range to talk to a microcontroller, with +5V representing a digital 1, and 0V representing a digital 0.

Step 8 – Install the Cable Driver

Now we need to install the driver so that your computer recognizes the USB to Serial cable.

If you are on Windows you should be able to install the device using the drivers found on the NerdKits downloads page.

On Linux, the cable we are using has been supported since Kernel 2.6. You should be able to plug in the cable and it should work. Type 'dmesg' in the command line and check to make sure that it recognizes a PL2303.

For Mac OS X we have included the drivers in the Installers folder in the NerdKits download. Just download and double click the installer and follow the instructions.

Quick Command Line Tutorial

The command line can do a lot of things for you, and learning how to use it is key if you are a programmer.

If you are a Windows user, go to run Start -> Run, then type "cmd" in the box and press enter. This will open up the Windows command prompt. If you type "dir" it will tell you all the files in the current directory. You can type "cd <directory>" to change the current directory to <directory>. You can do this over and over to navigate to any folder you want. Typing "cd .." takes you one folder up.

If you want to know more about the Windows command line, try reading
http://www.animatedsoftware.com/faqs/learndos.htm
It explains all the commands you will ever need and then some.

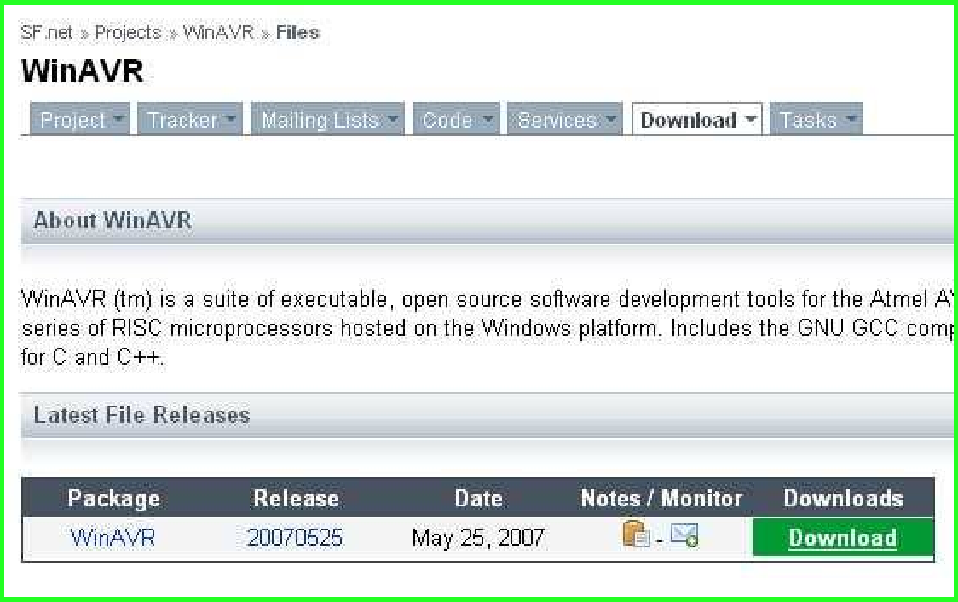
If you are a Linux or a Mac user you need to get to a command line, or Shell. This is usually straightforward on any Linux system. On Macs you can open a command line by opening the Finder, clicking on Applications, and double clicking on the Terminal program (it looks like a little black screen).

The two main commands you will need are "ls" which lists all the files in a directory. And "cd <directory>" for change directory.

We expect Linux users to already familiar with the terminal. For Mac users, here is a good explanation of the command line http://www.macdevcenter.com/pub/a/mac/2005/05/20/terminal1.html

Step 9a – Set up the programming environment in Windows

1. One of the best things about this line of microcontrollers is that there is a powerful, cross platform C compiler available.
2. We have included the installer for WinAVR AVR-Studio in the downloads section, you can also download it at. http://winavr.sourceforge.net/.
3. Double click the Win-AVR installer and follow the on-screen instructions to install the software.



WinAVR (tm) is a suite of executable, open source software development tools for the Atmel A
series of RISC microprocessors hosted on the Windows platform. Includes the GNU GCC comp
for C and C++.

Latest File Releases

Package	Release	Date	Notes / Monitor	Downloads
WinAVR	20070525	May 25, 2007	-	Download

Win-AVR contains a suite of programs that make programming the Atmel chips easy. The installer will install gcc on your computer which we will use to compile the C code we write into instructions the chips understand. It also installs a text editor called Programmers Notepad in which we will write the code. It is very helpful for providing syntax highlighting and the ability to view multiple files at once.

Step 9b – Set up the programming environment on Linux and Mac

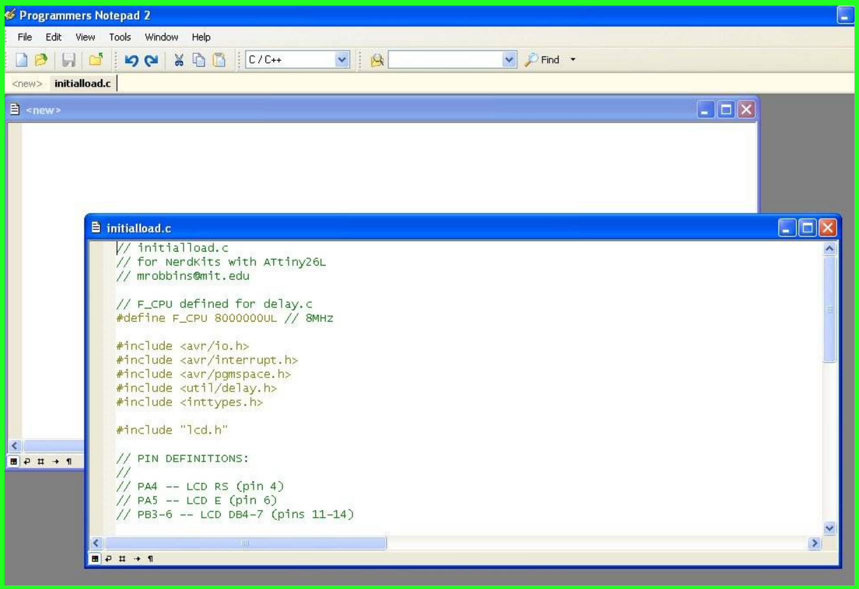
For Mac users, we have included the AVR MacPack on the NerdKits download section. Run the installer from the Installers folder and follow the instructions. Like WinAVR, AVR MacPack contains all the command line tools you need to be able to compile and install C code on the AVR microcontrollers.

IMPORTANT: After installing AVR MacPack go to a terminal and type "avr-gcc-select 4" to switch to the new compiler.

If you are a Linux user, this is a lot easier. Simply install avrdude, gcc-avr, and avr-libc. Use apt-get or your favorite package manager to install them. Use your favorite text editor to edit the code. Gedit will work just fine, but you might prefer something more powerful like Vim or Emacs.

Step 10a – Compile and Install your first program

1. Your NerdKit's microcontroller came pre-installed with a program that we wrote to display the previous message to LCD. We will now upload a different program to the chip to show you how to install a new program.
2. We have included the code for a new program in a folder called initialload. This code has not yet been compiled. Notice that the folder contains a Makefile. This is a special type of file that tells the gcc compiler which files to compile and in what order. In our case, it will also be calling the program that actually programs the chip.
3. Make sure you download the code for this kit from the downloads section (in the members area) of the NerdKits website.
4. Open up Programmers Notepad. Then from initialload folder open the file initialload.c. If you are on Mac or Linux, you don't have Programmers Notepad, but you can use any text editor you have to edit the file.
5. What you see is the code that will display a message to the LCD. You can look it over if you want, although you don't have to understand it since we are just going to compile and load it.



```
// initialload.c
// for NerdKits with ATtiny26L
// mrobbins@mit.edu

// F_CPU defined for delay.c
#define F_CPU 8000000UL // 8MHz

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <util/delay.h>
#include <inttypes.h>

#include "lcd.h"

// PIN DEFINITIONS:
//
// PA4 -- LCD RS (pin 4)
// PA5 -- LCD E (pin 6)
// PB3-6 -- LCD DB4-7 (pins 11-14)
```

The code we have included is not really that complicated, and has a lot of comments if you want to try to understand it. If you want a quick challenge, try to figure out where the text is that gets displayed on the LCD. If you think you know where it is try changing it and displaying a different message. We suggest making a copy of the original file before changing it...

Step 10b – Prepare for Programming

If you are on Windows you need to check what COM port the USB cable loaded as. This might change every time you connect it. Go to the Device Manger from Control Panel. Expand the Ports (COM & LPT) section and see what is next to the "Prolific USB-to-Serial" line. It should be COM5, COM6, or some other number. You need to open the Makefile in a text editor and edit the line that begins with AVRDUDEFLAGS. At the end you need to change "/dev/ttyUSB0" to "COM5" or whatever you saw in the previous steps.

There is a bug with the driver that causes the cable not work with any COM port larger than COM5, so if your computer assigns something larger to it, you have to manually change it.

Mac users need to check /dev for a device like /dev/cu.PL2303-0000211A Using the terminal, list the files in the /dev folder, you should see a file like cu.PL2303-0000211A the last 4 numbers will change depending on which port you plug it into. You need to open the Makefile in a text editor and edit the line that begins with AVRDUDEFLAGS. At the end you need to change "/dev/ttyUSB0" to "/dev/cu.PL2303-0000211A" or whatever you saw in the the /dev directory.

If you are on Mac you might have to give yourself write access to the Code folder on your hard drive:

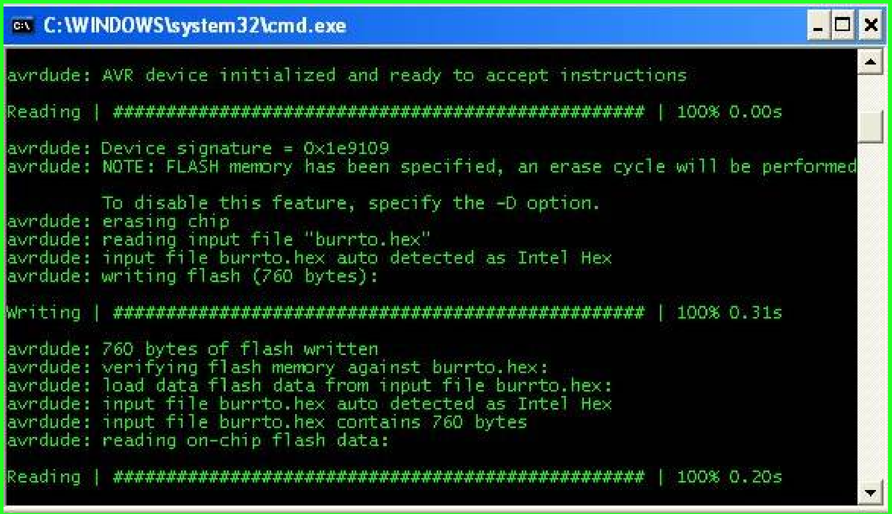
1. Select the Code folder in the Finder.
2. From the File menu, choose Show.
3. Choose Ownership and Permissions.
4. Change the permissions to give yourself read and write access.
5. Hit apply to enclosed items

For Linux users, "/dev/ttyUSB0" should work.

Make sure your NerdKit is connected to the cable correctly, and the programming switch is up. Disconnect the battery and reconnect it to boot into programming mode. When the kit is turned on with the switch in the UP position, it knows to listen for a new program. When the switch is down it just runs the program already on the chip.

Step 10c – Compile and Install your first program

1. Now that you have seen the code, we need to use the command line to program the chip.
2. Open up the command line and navigate to the folder with the code in it.
3. List the files in the directory and make sure there is a file called Makefile.
4. Type "make" and press enter to run the "make" command. This command will look for the Makefile and use it to compile all the code, and will then program the chip using the binary files. (If you want to learn more about Makefiles, go to http://www.eng.hawaii.edu/Tutor/Make/ .)
5. You should see an output on the command line telling you what the program is doing.
6. If all goes well, one of the last lines should read "avrdude: --- bytes of flash verified" where --- is some number. This means that avrdude has successfully written the program onto the chip, and read it back.



```
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e9109
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed

         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "burrto.hex"
avrdude: input file burrto.hex auto detected as Intel Hex
avrdude: writing flash (760 bytes):

Writing | ################################################## | 100% 0.31s

avrdude: 760 bytes of flash written
avrdude: verifying flash memory against burrto.hex:
avrdude: load data flash data from input file burrto.hex:
avrdude: input file burrto.hex auto detected as Intel Hex
avrdude: input file burrto.hex contains 760 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.20s
```

Congratulations!

You have now programmed your chip for the first time.

At this point you have enough knowledge about your NerdKit to begin developing your own projects. You know how to put components on the breadboard and where to write the code to program it. We are now going to add a temperature sensor and make a small thermometer that displays the temperature on the LCD, and learn some C programming on the way there.

It doesn't work - Troubleshooting

Symptom: You get an error that says

```
avrdude: ser_open(): can't open device...
```

Diagnosis: The COM port or Mac/Linux device address is incorrect in the Makefile.

Symptom: After typing make you get a message that says:

```
avrdude: AVR device not responding
avrdude: initialization failed, rc=-1
```

Diagnosis: This will happen if the NerdKit is not on, or one of the programming cable wires is loose.

Symptom: You get

```
avrdude: verification error; content mismatch
```

Diagnosis: This means an error occurred while transferring your program to the chip. Try running it again. If the problem persists try a new battery.

Symptom: When you try to program, it tries to write the program but stops before it finishes.
Diagnosis: This could indicate your battery is low. Try a fresh one.

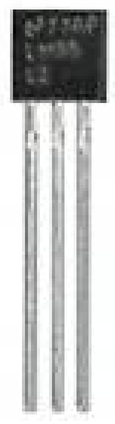
The Temperature Sensor

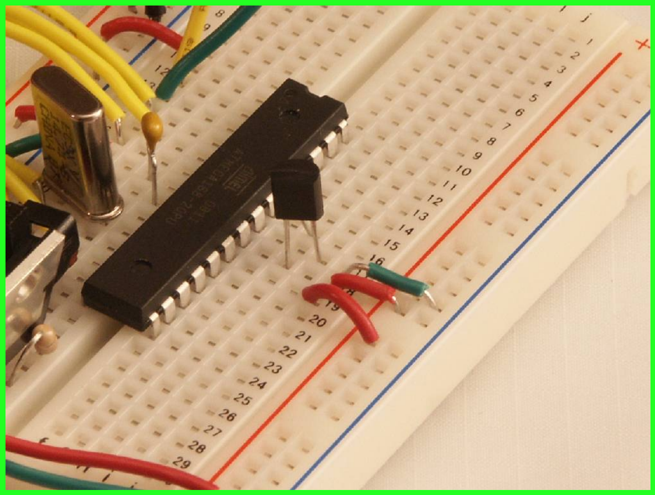
We are now going to guide you through your first functional NerdKits project! As you do the project, you should learn a lot about how to code for embedded digital devices.

The first thing we need to do is connect the temperature sensor to the chip so that it can read the temperature information. Find your temperature sensor: it's the chip that has LM34 written on the front, and looks like the picture below.

If you look at the temp sensor, you need to connect the right pin to MCU pin 22, which is in row 17 on the right side of the breadboard. The middle pin needs to be connected to MCU pin 23 (breadboard row 16). The left pin goes in MCU pin 21 (breadboard row 18). You're going to need to bend the temperature sensor pins a little to make them fit into the rows. (Left and right are relative to looking at the flat face of the chip, with the pins pointing down.)

When you connect the battery, the temperature sensor will output a voltage on the middle pin that is proportional to the temperature. We will write code that takes this voltage, converts it to a digital number, and then writes it to the screen.





Embedded Programming – Part 1

Programming for embedded devices like your Atmel chip is a little bit different than programming for your PC. We are lucky that this chip has C compiler for it, which means you can write in the C programming language and the compiler will turn it into instructions the chip understands. However, even if you know C there are a few quirks that you have to get use to.

Although we will try to explain most of the C concepts, if you have never written a computer program before we recommend you read one of the many online tutorials to get started first. We recommend
http://www.howstuffworks.com/c.htm
It has a good explanation without being too technical.

We are going to guide you through writing all the code for the temperature sensor. You might not understand all of it, but it is important that you try your best to understand it, and it is even more important that you physically type the code yourself. This will get you used to programming, and help you understand the code.

Take a look inside the tempsensor_edu folder that is on the NerdKits download (you should already have a copy on your hard drive). It contains several files, including the Makefile you should be a little familiar with by now. The file we will be editing is tempsensor.c. This file right now is just a stub where you will be writing your own code. In case you get stuck, we have also included the tempsensor folder which contains the full working code. Open up tempsensor.c in Programmers Notepad and take a look at it. You will be editing this file directly. Make sure you save often to prevent losing your work, and don't just copy and paste from this document. Type the code out yourself. It's the best way to learn!

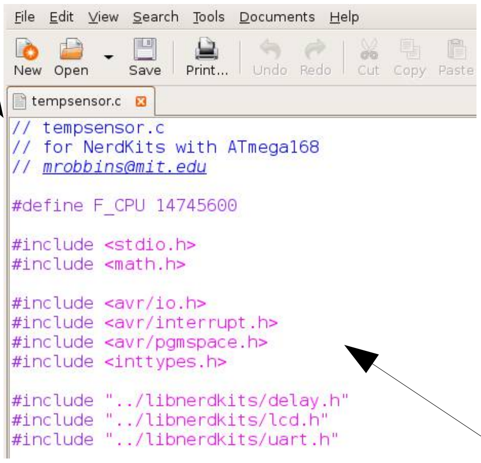
The first thing to learn about modern programming languages is that they are extremely high level. Computers operate in a world of ones and zeros, and use very complicated codes. If we were to try to talk directly to a CPU, it would be very difficult. Luckily, we have developed translators that take programming languages like C and turn them into things the computer understands. We call this process compiling the code. The compiler basically takes the C code you write and outputs a binary file that really is just ones and zeros that the computer understands.

Generally, we don't have to worry too much about the binary code that the compiler produces, and higher level languages usually abstract away the concept of a bit (a one or a zero the computer understands). However, for better or worse, when talking to small devices like your MCU, we sometimes have to worry about specific bits and what they do.

Embedded Programming – Part 2

Let's start at the top. Add the following lines to the code in the right places. Make sure you read the explanations and don't just copy the code.

In C, anything that comes on a line after // is a comment. The compiler completely ignores it. They are very useful for writing reminders to yourself, and for explaining the code in plain English. Good programmers will always comment their code well.


File Edit View Search Tools Documents Help
New Open Save Print... Undo Redo Cut Copy Paste
tempsensor.c

```
// tempsensor.c
// for NerdKits with ATmega168
// mrobbins@mit.edu

#define F_CPU 14745600

#include <stdio.h>
#include <math.h>

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <inttypes.h>

#include "../libnerdkits/delay.h"
#include "../libnerdkits/lcd.h"
#include "../libnerdkits/uart.h"
```

The #define statement is a special statement in C that is used to define a pre-defined constant. All this means is that after that line, you can type F_CPU and it will mean 14745600. This is merely the number 14,745,600.

Include statements are a very important part of the C language, especially when programming for embedded devices like your MCU. C, like many other languages, is made up of a basic set of things you can do. It's pretty much a combination of functions you can call to do things in your code. The language comes with lots of these sets of functions we call libraries, or modules. Now, when we write a program, we usually only want to include the things we need to do a certain task. To use anything in a library that is part of the C language, you first have to write an include statement. Its simple, you just type #include followed by the name of the module you are using enclosed in < >. You will get better at knowing which modules you need as you program more. The modules we have included do some pretty basic things, like input and output for avr chips <avr/io.h>, and functions for waiting specified amounts of time <util/delay.h>

Notice that there are two types of include statements. Modules that are part of the C programming language are enclosed in <>. You can also write modules yourself, and include them using “”. In this case, we have written lcd.h which contains useful functions for writing things to the LCD, and have included it using an #include statement. Since it's in the same directory as your main code, you use the “” type of include. All the libraries we have included for you are in the libnerdkits folder.

Quick Aside – Bits on a chip

The digital world uses bits to represent information. Explaining what a bit really means is complicated, and really unnecessary. (You might enjoy learning about what "information" really means, and how it relates to probabilities, and entropy, and compression.) For now, we are going to consider a bit as something that can be a 1 or 0 – on or off. Or better yet, something that can represent a 1 or 0. This is convenient in electrical engineering because we can use voltages to represent 1s and 0s. In fact we have developed many tools and common circuit elements that let us store, manipulate, and compare high and low voltages. Modern engineers use these tools to bridge the gap between the physical world of voltages, and the 1s and 0s they are representing.

Even if you are familiar with programming in C, chances are you are not too familiar with bit manipulation. It is a fairly hard concept to learn, but it is well worth it if you really want to understand how the digital world works. Digital components, your MCU included, use registers to store bits of information. These registers can be set to 1 or 0. This physically means that a register will have either a high or low voltage. Fortunately, the chip will hide all the voltage stuff away from us, and we can think of it as writing a 1 or a 0 into some slot on the chip. These registers and bits are used for different things. Sometimes they will be used to store values you can read back later, sometimes they will hold information you will want to read for some purpose, and sometimes you will want to set bits on registers to make the chip behave a certain way.

The datasheet for the chip has all the information about the chip. It explains what the different registers are called, what they do, and what setting their bits to different values does. As we walk you through the code we will explain how to set bits on the chip.

If you are not very familiar with the concept of a bit, or how we use operators to manipulate them, we suggest you read

http://www.cprogramming.com/tutorial/bitwise_operators.html

It is a quick tutorial on binary operators.

The keys are to understand the concepts of AND, OR, and NOT on an individual bit level – and they're concepts you already know. Once you do that, you then just have to realize that operations can be combined to make arbitrary relationships between inputs and outputs. And from the C programming perspective, that these operators ("&" for AND, "|" for OR, and "~" for NOT) operate on a whole byte at a time, but treat each bit independently.

Embedded Programming – Part 3a

This set of comments is just there to help you when you are writing the code. They are reminders of which pins mean what.

This is a good practice to start when you're working on your own projects.

This is the first really difficult part to understand. This is the stuff that makes microcontroller programming hard, so don't be discouraged if you don't get it right away.

The next thing we did was define a function. A function in C is just a set of instructions that are grouped together. You can later call the function, and these instructions will be executed in order. We declare a function by stating its return type – what it outputs – first. This particular function does not return anything, so we write “void”. Then we give the function a name, in this case adc_int(). While we could have called it anything we wanted, it is good coding practice to use useful names that mean something. In this case ADC stands for Analog to Digital Controller. Then we use { and } to enclose the instructions that are part of the function.

```
// PIN DEFINITIONS:
//
// PC0 -- temperature sensor analog input

void adc_init() {
  // set analog to digital converter
  // for external reference (5v), single ended input ADC0
  ADMUX = 0;

  // set analog to digital converter
  // to be enabled, with a clock prescale of 1/128
  // so that the ADC clock runs at 115.2kHz.
  ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);

  // fire a conversion just to get the ADC warmed up
  ADCSRA |= (1<<ADSC);
}
```

This line is simply setting one of the registers on the chip. ADMUX, ADCSRA, ADEN and all the rest of the variables used are predefined in on the include files. When we assign values, it's as if we are actually setting the values of a register called ADMUX. To figure out what this register does, you have to look in the data sheet. In fact, the data sheet has all the information about the chip you could ever need.

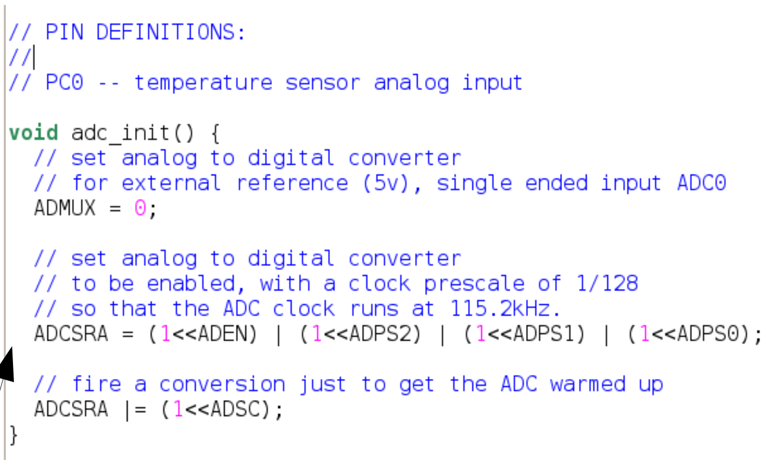
The first line is simply setting ADMUX to be 0 this is simply telling the chip which reference voltage and which input we want to use. This is important when using Analog to Digital converters. For now its not too important you understand what this means.

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The data sheet for this chip can be found in the downloads section of the NerdKits website., or just Google for ATmega168 datasheet.

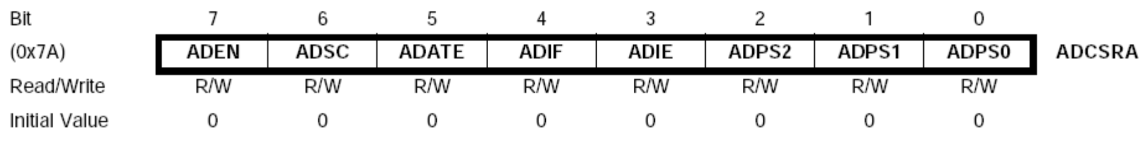
Embedded Programming – Part 3b

The entire point of this function is to initialize the ADC (Analog to Digital Converter). This will help us take the analog voltage of the temperature sensor and turn it into digital information that we can work with. If you look at the data sheet and the comments in the code, you can see that it is merely setting some registers on the chip such that things work correctly later.



```
// PIN DEFINITIONS:
//
// PC0 -- temperature sensor analog input

void adc_init() {
  // set analog to digital converter
  // for external reference (5v), single ended input ADC0
  ADMUX = 0;

  // set analog to digital converter
  // to be enabled, with a clock prescale of 1/128
  // so that the ADC clock runs at 115.2kHz.
  ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);

  // fire a conversion just to get the ADC warmed up
  ADCSRA |= (1<<ADSC);
}
```

This line is more complicated than the other ones. This is because for the ADCSRA register we need to set more than one bit to 1. In this case, we need to set the ADEN, ADPS2 and ADPS1, ADPS0 bits to 1. If you want you can read the data sheet and find out why we want to set them to one. Since ADEN is bit #7, ADPS2 is bit #2, ADPS1 is bit #1, ADPS0 is bit #0 we want to set ADCSRA equal to 10000111 in binary. This is a number that is all zeros with ones in the places corresponding to ADEN, ADPS2, ADPS1, and ADPS0. The easiest way to construct such a number is using the method above. What this line does is creates three separate numbers. (1<<ADEN) creates 10000000, (1<<ADPS2) creates 00000100, (1<<ADPS1) creates 00000010, and (1<<ADPS0) creates 00000001. Then we OR these numbers together to create number we want. In C the | operator means OR.



Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The last line does something similar to the other two. You should read the data sheet and find out what happens when you set ADSC bit of ADCSRA to 1.

We are now done with our first function. Now that it is written, we can call it later in the code to initialize the analog to digital converter

Embedded Programming – Part 4a

Now we are going to write another function called adc_read(). This function will be used to read a voltage from the analog to digital converter. Again, fill in your file to make it look like the one below, but try to understand why you are doing each line while you type it.

This function will return a value when it is called, so we have to tell the compiler what kind of value it will be returning. In this case it returns something of type uint16_t. In C this means an unsigned integer that is 16 bits long. It's unsigned because it can't be negative, and it's 16 bits long because the ADC can measure 1024 steps (10 bits), so the information can't be held in an 8-bit integer.

You should already know what this line does. If you don't remember, try figuring it out by looking at the data sheet. (Hint: look up what the ADSC bit does.)

```
uint16_t adc_read() {
  // read from ADC, waiting for conversion to finish
  // (assumes someone else asked for a conversion.)
  // wait for it to be cleared
  while(ADCSRA & (1<<ADSC)) {
    // do nothing... just hold your breath.
  }
  // bit is cleared, so we have a result.

  // read from the ADCL/ADCH registers, and combine the result
  // Note: ADCL must be read first (datasheet pp. 259)
  uint16_t result = ADCL;
  uint16_t temp = ADCH;
  result = result + (temp<<8);

  // set ADSC bit to get the *next* conversion started
  ADCSRA |= (1<<ADSC);

  return result;
}
```

This while loop is here to make the chip wait until the ADC is done computing the result. If you read the data sheet you will see that you set the ADSC bit high when you want the chip to start computing. The chip then sets this bit low to let you know it is done. This while loop continuously checks the ADSC bit of ADCSRA and does nothing while it is still high. To understand how this block of code does this you need to know how a while loop works, and what bitwise AND (&) does. If this is shaky, review the C tutorial and bit operators references we suggested earlier.

Once the ADC is done computing a value, we can read the result it has stored for us. As we see on the data sheet, the ADC stores the result from its conversion in two 8-bit registers, named ADCL and ADCH (for low and high). It splits it up because there is not enough room in just one byte to represent such a large number. Because of this, we have to store the results in separate variables, and then stick them back together. Since ADCH is the high order bits, we have to shift it up by 8, then add it to the ADCL to get the full 16 bit number.

This last line is the return statement. It tells the function to return a value to whoever called the function. We simply return the result from the ADC.

The ADC on the chip plays a guessing game with the analog voltage, starting in the middle of the voltage range and asking if its guess is under or over. It repeats this ten times to get a full 10-bit result. This technique is called a "successive approximation ADC."

On our chip, it takes about 0.1 milliseconds to make one reading. The adc_read() function would have to wait during all this time. However, in the code we begin a new read at the end so the rest of the code can run while the ADC is making a reading. An alternative method, not covered in this guide, is to use interrupts, which would let the microcontroller do other things while the ADC is working.

Embedded Programming – Part 4b

Next we define a function to make it easy to convert a sample we get from the ADC to a number in degrees Fahrenheit. The conversion at this point is not too complicated; it simply takes the reference voltage which is the +5 rail, divides it by the number of steps that the ADC uses (1024 because you can represent 1024 different numbers with 10 bits), and divides that by 10 because the temperature sensor changes by 10mV per degree Fahrenheit.

```
double sampleToFahrenheit(uint16_t sample) {
  // conversion ratio in DEGREES/STEP:
  // (5000 mV / 1024 steps) * (1 degree / 10mV)
  //    ^^^^^^^^^^^               ^^^^^^^^^^
  //     from ADC                  from LM34
  return sample * (5000.0 / 1024.0 / 10.0);
}
```

If you wanted to make a very accurate temperature sensor you would have to be a lot smarter about this conversion. Here we are assuming that the reference voltage is exactly 5V, which is not exactly true especially after your battery starts losing its charge. A way of calibrating the reference voltage, as well as other optimizations, would be necessary to make a very accurate temperature sensor.

Embedded Programming – Part 5

The next function we are going to write is the main() function. Every program you write in C has to have a main() function. This is the function which automatically runs when you start your microcontroller. From the main method you can do whatever you want, including calling other functions. From our main method we will be doing computations, using the methods we already wrote, and writing to the LCD.

```
int main() {
  // start up the LCD
  lcd_init();
  FILE lcd_stream = FDEV_SETUP_STREAM(lcd_putchar, 0, _FDEV_SETUP_WRITE);
  lcd_home();

  // start up the Analog to Digital Converter
  adc_init();

  // start up the serial port
  uart_init();
  FILE uart_stream = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);
  stdin = stdout = &uart_stream;

  // holder variables for temperature data
  uint16_t last_sample = 0;
  double this_temp;
  double temp_avg;
  uint8_t i;
```

lcd_init() and lcd_home() are functions that were included from lcd.h. These functions were written by us to make it easy for you to write things to the LCD. You should open up lcd.c and see what these functions do. The line between them is declaring a variable called lcd_stream. This sets the LCD up as a stream that you can write to in the same way you would write to a file.

The next line is a call to adc_init(), the function which you wrote earlier. The next three lines are just declaring variables which we will put values in later.

uart_init() is a function in uart.h. The next two lines set up another stream uart_stream and map stdin and stdout to that stream. These three lines together setup the connection over the serial cable to the computer. This is not necessary, since we are printing out the results of the temperature conversion to the LCD, but we are also going to be printing out the values to the serial connection so you can read them from your computer if you want to.

The last four lines simply declare four variables that we will use to hold data in the future.

It is important to notice that we have not closed the { at the beginning of the main function, so we are not done writing it yet. The instructions that are coming up are also going to be part of this function.

Embedded Programming – Part 6a

The next while loop is going to loop forever. This is because once we turn on the microcontroller we want it to run until it is turned off. In most cases where you are writing an application that stays on you will have an infinite loop like this one.

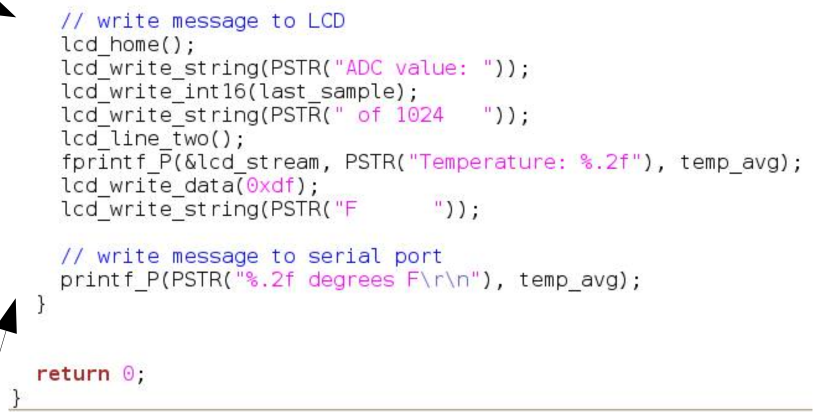
In fact there is no well defined behavior for what happens if your main method actually returns, so don't let your code finish.

```
// holder variables for temperature data
uint16_t last_sample = 0;
double this_temp;
double temp_avg;
uint8_t i;

while(1) {
  // take 100 samples and average them!
  temp_avg = 0.0;
  for(i=0; i<100; i++) {
    last_sample = adc_read();
    this_temp = sampleToFahrenheit(last_sample);

    // add this contribution to the average
    temp_avg = temp_avg + this_temp/100.0;
  }
```

Here we are using a for loop to average out 100 samples from the temperature sensor. The for loop loops 100 times, each time it takes a sample from the temperature sensor, divides it by 100 and adds to a running total of the average. At the end the average of the 100 samples are left in the variable temp_avg. Notice that we are using the function sampleToFahrenheight() you wrote earlier to convert the sample into degrees Fahrenheit.

Embedded Programming – Part 6b

After we have the number we want to display, it is just a matter of writing the information we want to the screen. All of these lines use functions in lcd.h to write characters to the screen where we want them. Make sure you understand what each of these lines does, and feel free to explore. Write interesting things to your LCD!


```
    // write message to LCD
    lcd_home();
    lcd_write_string(PSTR("ADC value: "));
    lcd_write_int16(last_sample);
    lcd_write_string(PSTR(" of 1024   "));
    lcd_line_two();
    fprintf_P(&lcd_stream, PSTR("Temperature: %.2f"), temp_avg);
    lcd_write_data(0xdf);
    lcd_write_string(PSTR("F      "));

    // write message to serial port
    printf_P(PSTR("%.2f degrees F\r\n"), temp_avg);
  }

  return 0;
}
```

Then make sure you write the } bracket. This closes the while loop that will keep going forever until the microcontroller is turned off. This loop will continuously poll the ADC to get the current temperature, do the averaging to reduce noise, convert the voltages to degrees, and then write the new message.

The last thing we write is the return statement. Because the while loop should be looping forever this function should never return in theory. That means the program should never get to executing the return line. We still have to put it in, or else the compiler gets confused. Make sure you save your file.

Notice how we have to use different functions when writing different things to the LCD. When we are writing numbers, like last_sample we use lcd_write_int16 or lcd_write_int16_centi because those numbers are 16 bit integers. To write words we use lcd_write_string(PSTR("your string here")). The PSTR() macro is a special compiler macro that lets us use the flash memory to store a string instead of using up our RAM. We have to use it because lcd_write_string() expects a string in the flash memory. To write special characters we need to use the lcd_write_data() function and give it the special ASCII code for the character we want. In this case 0xdf stands for the little degree sign. Again, none of these things are actually necessary. If you want to just view the temperature you could print out just that. If you want to write your own message, you can. We also used fprintf_P() to write data to the LCD. It is easier to do things this way when you are using floats.

The last line uses printf_P() to write a line to stdout, which you remember we mapped to the UART stream which writes data to the serial connection. If you wanted to you could connect your computer and read the temperature from your computer. The printf and fprintf functions are very powerful, and you can find information about in our printf/scanf tutorial on our website.

Embedded Programming – Part 7

The final step is to compile the code and program the chip. You should already be familiar with this process from step 8. We have already included a Makefile in the tempsensor folder to make it easy for you. All you have to do is connect the NerdKit to your USB Port, turn it on, navigate to the folder from the command line and type “make”. This will compile all the necessary files and then upload the code to the chip.

If it doesn't work, don't be discouraged. It never works for anybody the first time. Good engineers are just good at figuring out what they did wrong and fixing it. If the command line tells you there is a compilation error, you most likely misspelled something, or forgot a line. Usually, the compiler will give you hints toward where to look. Go back and check every line against the template to make sure you did not make any mistakes. If the code uploads successfully and it still doesn't work check all your connections again, as it is likely a wire just wiggled loose.

Once it works you should see the temperature and any message you chose to write. This means the code is running off the chip. You can go ahead and disconnect it from your computer, move it around and see the temperature change!

We are very interested to know what people are doing with their NerdKits. If you are stuck, have questions, comments, or you just want to show off your cool project, send us some pictures or write us an email at projects@nerdkits.com.

Congratulations! You have finished your first NerdKit Project!

Quick Aside: AND and/or OR

Before we dive directly into the next project, we are first going to cover the basics of the AND and OR operator again. There are two main reasons why you need to be VERY familiar with these operators. First, because we are working in a microcontroller we are limited by space and speed much more than we would be if we were programming in a normal computer. Because of this, many of the computations we do are lower level computations that directly involve bit manipulations. Second, as you are no doubt have realized if you have done the temperature sensor, the way we control the settings of the MCU and pins of the MCU is through setting and reading bits on different registers. In the tempsensor project, you notice we did bit manipulations using AND and OR to set the ADCSRA register. The Analog to Digital converter would behave differently depending on how we set the bits of that register.

Lets review what the AND operator does. The operator compares two bits and returns a 1 only if both of the bits are 1, otherwise it returns a 0. The & operator will perform the operation to two numbers in a bitwise fashion. Which means if you do 11011010 & 11111111 the result will be 11011010. Notice that it just ANDs each corresponding bit of the number.

The OR operator takes two bits and returns a 1 if either one, or both are a 1, and returns 0 if they are both 0. The | operator does the OR operation, so doing 11011010 | 11111111 would return 11111111. One interesting to note is that ANDing a number with all 1s will result in the exact same number as before. What will ORing a number with all 0 do?

The last very important piece to the AND and OR puzzle is the ~ (NOT) operator. This operator just flips whatever bit you pass it. So calling ~(11011010) will return 00100101.

Extra Practice: AND and/or OR

AND and OR is used a lot in embedded programming, so we are including some extra problems for you to work through. As you are working them out try to notice the special properties of AND, OR, and NOT.

11111111 & 00001111 = ______________

11111111 | 00001111 = _______________

11011010 | 00000100 = ______________

11011010 | 10000100 = ______________

11011010 & 10000100 = ______________

11011010 & ~(00001000) = ___________

11011010 & ~(00001001) = ___________

11011010 & ~(1<<3) = _______________

11011010 | (1<<2) = _________________

Suppose ADCRA is an 8 bit register...

I put the value 11011010 into ADCSRA.
Then I do:

ADCSRA &= ~(1<<3)
What is ADCSRA now? _____________

Answers are on the next page...

Extra Practice: AND and/or OR

Here are the answers to the questions in the last page. As always, it pays to think about why we chose the particular problems and not just about what the answer is. Pay special attention to the bold problems.

11111111 & 00001111 = 00001111

11111111 | 00001111 = 111111111

11011010 | 00000100 = 11011110

11011010 | 10000100 = 11011110

11011010 & 10000100 = 10000000

11011010 & ~(00001000) = 11010010

11011010 & ~(00001001) = 11010010

11011010 & ~(1<<3) = 11010010

11011010 | (1<<2) = 11011110

Suppose ADCRA is an 8 bit register...

What is ADCSRA now? 11010010

Notice that if we take a binary number, and OR it with a number that only has 1 bit on, the resulting number will be exactly the same as the first number, except that it now has a 1 in the bit position where the second number had a 1. In the second bolded example we notice that we end up with a 0 in the place where put a 1, and the rest of the number is unchanged. Think about why this is useful!

AND, OR, and bit masking

So, why so much emphasis on bits, AND and OR? Well, as you no doubt have see by now most of the interesting stuff we do with our microcontroller requires setting and reading registers. This means that there exists physical spaces on the MCU that hold 8 bit numbers. These registers control what the MCU does. The pins themselves are controlled by 8 bit registers, and you have to set bits to turn them on or off. Setting the PWM, and ADC, and even enabling the functionality of your chip involves setting individual bits of certain registers. Luckily, using AND and OR, and all our newfound knowledge of bit manipulation setting and reading bits on registers is as easy as π.

```
ADCSRA |= (1<<ADSC);
```

Lets take a look at a quick example from the tempsensor. Recall that to start a conversion you need to set the ADSC bit of the ADCSRA register to 1. This line will set the ADSC bit to 1, without affecting the rest of the bits of the ADCSRA bit! This is useful because the other bits control how the ADC behaves so we don't want to alter them. That is the power of OR.

Setting the ADCS bit starts a conversion on the ADC. When the MCU is done, it sets this bit back to 0, so we need to check this bit to see if it is still set or not. To do this we look at the value of this expression:

```
ADCSRA & (1<<ADSC)
```

Using your knowledge of the & operator, you can see how this works. Shifting a 1 over by ADSC bits will put a 1 at that bit and 0s everywhere else. This will effectively mask out all the other bits, and leave a 1 in the ADSC spot only if the bit is still set! Otherwise the expression evaluates to 0.

One last thing that is worth mentioning is that the way we create these numbers can vary greatly. For example if I write ADCSRA |= (1<<2), it will have the same effect as writing ADSCRA |= 4, or ADSCRA |= 0x04. I can choose to write the number in decimal, or hex, or out of a sequence of shifts, but as long as the binary representation of the number is right, the code will work. So when trying to understand other peoples code be ready to do some binary conversions. Sometimes our engineers here at NerdKits use particularly convoluted methods just to keep you on your toes!

The Point (so far)

It might seem like all this talk about registers is a bit redundant, but it is the one thing that most people struggle to get their heads around. Most of the useful operations in a microcontroller are based around setting a register, or reading from a register. If you take a glance through the datasheet, most of the pages will have tables on them that simply explain what a certain register does. Want to use a pin as an output? You're gonna have to set a bit on register. Want to use a pin as an input? You're gonna have to set a bit on a register. Want to read whether something connected to a pin is high or low? You are going to have to read the value of one bit on a register. Do you remember how to do that?

But those things are easy... Want to use one of the timers on the chip? Page 102 of the Datasheet will tell you all about one of the timers. You are going to have to set some bits on TCCR0A and TCCR0B (these are the control registers for the timer). Then, as the timer is running you can read from TCNT0, which is going to hold the current timer value. You can also set OCR0A to do an output compare, and set TIMSK0 to make interrupts fire when certain conditions are met.

The point is, an MCU has a lot of power within it, and to use the power you need to set and read registers. Most of the time, learning how to do something with a microcontroller is a matter of reading the datasheet to find out which bits on which register you need to set. The really hard part is designing the system to use all the pieces together to do something cool.

Before moving on, make sure you know how to read a bit from a register, and set a certain bit of a register without affecting the other bits. More importantly, make sure you understand why it is important to be able to do so.

Quick Overview – Digital Input and Output

The microcontroller has 23 pins of digital inputs and outputs. Some of these are used for the LCD and programming cable already. But if you want to use them as general-purpose digital inputs and outputs, you should be aware of the following registers:

- DDRB, DDRC, and DDRD: set the direction of each pin (input or output)
- PORTB, PORTC, and PORTD: set the driver on each pin (output)
- PINB, PINC, and PIND: report the value measured on each pin (input)

The datasheet has much more information on how this all works.

Here's an example of turning a pin on and off (output). Suppose we want to control pin PC3:

```
DDRC |= (1<<PC3);              // set PC3 as output
while(1) {
        PORTC |= (1<<PC3);     // set PC3 high
        PORTC &= ~(1<<PC3);// set PC3 low
}
```

Here's an example of using a pin as an input.

```
DDRC &= ~(1<<PC3);          // set PC3 as input
while(1) {
        if(PINC & (1<<PC3)) {
                // do something if PC3 is high
        } else {
                // do something else if PC3 is low
        }
}
```

The microcontroller's pins can drive enough current to brightly light an LED, which is what we do in the Blinking an LED example. Additionally, don't forget that for a pin configured as an input, setting its corresponding PORT bit high will turn on an internal pull-up resistor. This is useful for connecting buttons and switches, so that the PIN value will be high when the switch is open, and low when the switch is closed to ground.

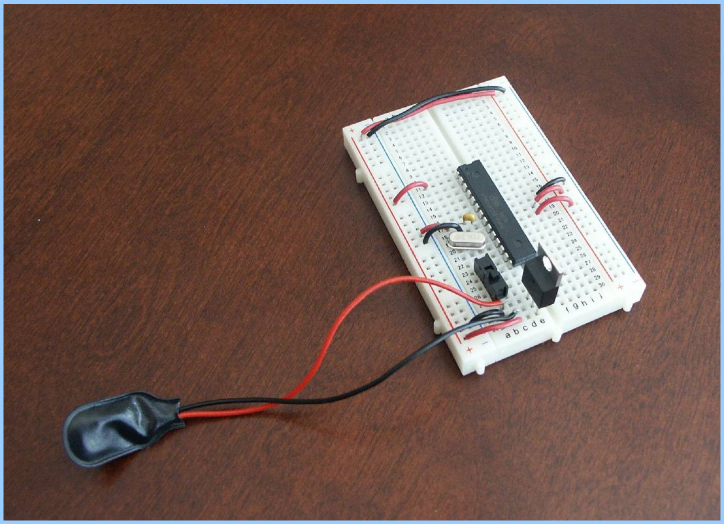
You will have more practice with these concepts in upcoming projects. But this page is here for reference.

Taking a step back – Blinking an LED

The temperature sensor project was meant to give you a peek into the wonderful amount of things that are possible once you learn to use microcontrollers to their full potential. Completing the first project also gets your NerdKit set up with the power regulator and crystal so you are ready to reprogram and start tinkering!

In this next mini project we are going to guide you through one of the simplest exercises in microcontroller programming – blinking an LED. In some ways, this might be considered the "Hello World" of microcontroller programming (though admittedly you already know all you need to write "Hello World" to the LCD). We will also be pointing out some of the simpler concepts of C programming along the way.

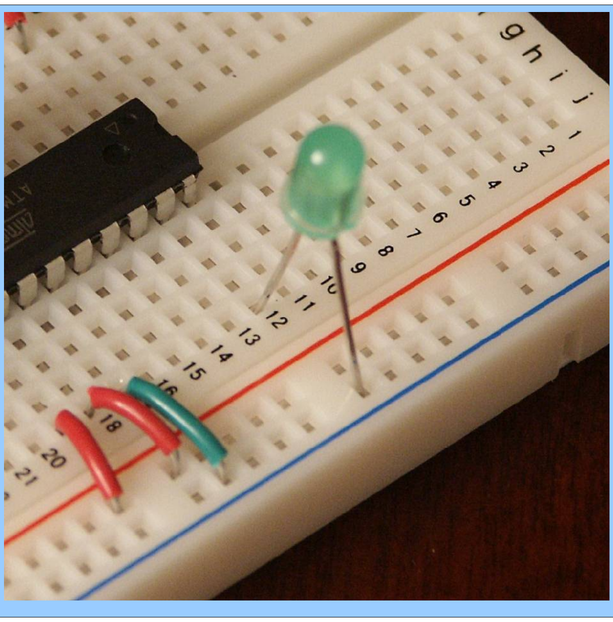
In order to get started with this project you need to remove the temperature sensor. Optionally, you can remove the LCD and all its wires too since they won't be required in this project. However, we recommend leaving them there as they do not get in the way, and the LCD can be used if desired.



Blinking an LED – Connect the LED

It is now time to connect the LED. Grab one of the LEDs included with your kit. When you hold it up, notice two things about it. One of the leads is shorter than the others. This lead is called the cathode. The cathode can also be identified because there is a flat dimple on the side of the LED on the cathode side. The other side is called the anode.

For an LED to light up you have to put a positive voltage across from the anode to the cathode. If you were to put a voltage across the other way, from cathode to anode, the LED would not light up. The exact workings of LEDs, including their voltage current relationship and how to modulate brightness are all lessons for another day (perusing the NerdKits forums will provide some insight), but for now it will suffice to say that the 5V provided by the MCU pins, combined with the effective resistance already present on the MCU pins will work just fine to light up an LED.



Take your LED and connect the anode to pin 27 of the MCU and the cathode to the GND rail. This way, when you use code to turn the pin high, the pin will put +5V at the anode, and the cathode will be at GND, putting +5V across the LED, making it light up! (In reality, the voltage at the pin will only get to around 2.5 volts, because the LED would require a tremendous amount of current to get to 5V, and the effective resistance of the MCU pin makes the pin voltage drop because of the current.)

Blinking an LED – The Code

The code for this quick project is presented in its entirety. As always try to look through the code and understand what is going on. The only thing you should be a little confused about is the concept of a register and how we flip bits on it. That will be covered further in the next project.

```c
// led_blink.c
// for NerdKits with ATmega168
// hevans@nerdkits.edu

#define F_CPU 14745600

#include <avr/io.h>
#include <inttypes.h>

#include "../libnerdkits/delay.h"
#include "../libnerdkits/lcd.h"

// PIN DEFINITIONS:
//
// PC4 -- LED anode

int main() {
  // LED as output
  DDRC |= (1<<PC4);

  // loop keeps looking forever
  while(1) {
    // turn on LED
    PORTC |= (1<<PC4);

    //delay for 500 milliseconds to let the light stay on
    delay_ms(500);

    // turn off LED
    PORTC &= ~(1<<PC4);

    //delay for 500 milliseconds to let the light stay off
    delay_ms(500);

  }

  return 0;
}
```

These are the include statements that we use to import some useful libraries. Without these we couldn't use the delay_ms() function.

Next comes the main() function. This is the function that the MCU automatically runs as soon as your program starts running. Notice that in this simple program, we only have one function, the main function.

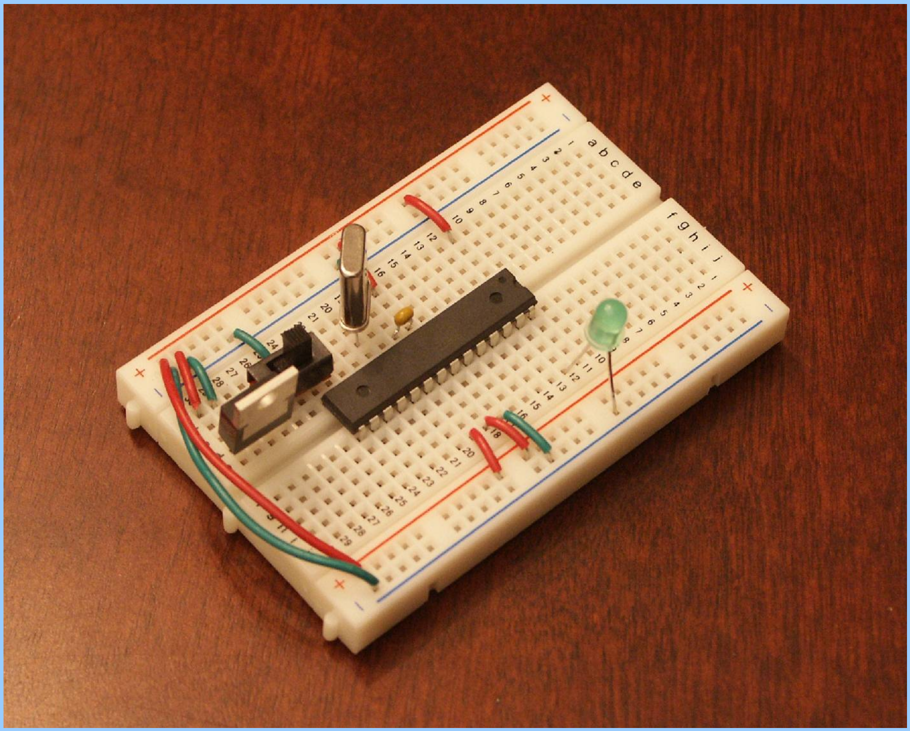
Next we make PC4 an output pin.

We then enter a while loop that will loop forever (as long as the MCU is on). The loop flips PC4 to high by setting the right bit on the PORTC register. It then does a delay of half a second during which the LED will stay on. Then it flips the pin back to off by setting the bit on the register back to 0. Since this is in the while loop it will keep happening over and over and over.

In this example we are only doing one thing in our while loop. If you feel up to it, modify the code to also write a message to LCD. Perhaps just a simple "hello", or even consider printing whether the LCD is on or off at that point in time. Also notice how easy it is to turn a pin to be a digital high or a digital low. It is a simple matter of setting a bit on a register!

Blinking an LED – Upload and Run

It is now time to run the code. Open up a command line and navigate to the led_blink folder. Notice the code is there along with its Makefile. Hook up your NerdKit to the computer via the serial port, put it in programming mode, and type make. It should compile and upload the code!



The thing to notice here is that you have individual control over every pin to make it a digital high (+5V) or a digital low (0V or GND). You can use this idea, along with accumulated knowledge of electronics to control other devices. In this case we are using the MCU to create a voltage difference across an LED. In the tempsensor project we used these voltages to send data over to the LCD to display characters. Using this very very basic idea of just turning a pin high or low, you can control most electronic components out there including motors, solenoids, and even LCDs. All of digital electronics is just sending data as electrical signals at the right time, and you now have the ability to do that using your MCU. The next step is to read digital signals on your MCU pins. We will tackle that in the bitwise arithmetic project.

In summary to individually control a microcontroller pin, you must first set it as an output, by setting the right bit on a DDR register. Then set it to be high or low, by setting the right bit on a PORT register. It's that easy! Take a moment to think about how amazingly powerful that is.

Blinking an LED – More Reflection

The other thing to take away from this project is an idea of how a very very basic program will work, and how you would go about starting your own project from scratch. The only absolutely required part is the main() function, which is the function that automatically runs on the MCU. You usually have some include statements as those are the ones that allow you to do interesting things, but in this case, not even those were really required. Inside the main() function, there is usually an infinite loop. This is because a microcontrollers program is meant to run as long as the chip is on, and start again once it is done. What happens when the MCU code gets to the end of the main function is actually undefined, so don't let it happen!

The other thing to notice about that very simple program is the comments! You should always comment your code, no matter how simple the program is. It helps you remember what your own code does, and it helps other understand what you were trying to do with it. The code you write, if well commented, could even help others learn about the finer aspects MCU programming.

Blinking an LED – The Makefile

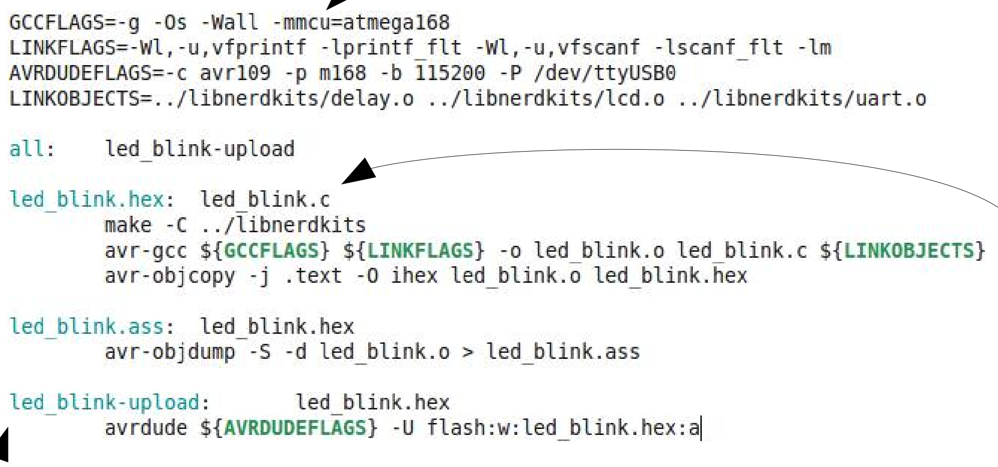
An extra advantage of the blinking LED project is that it gives us a chance to talk about the Makefile. Every time you want to make a new project, step one is to create a new folder inside the Code folder. Call it something related to what you are doing (not mandatory, but good practice). Inside that folder you need at least two files, the Makefile, and a C file with your main() method in it. We recommend copying the Makefile from a different project (intialload is a good one) and modifying to fit your new project. In practice this modification is just a replace-all from the old project to the new project, so if you compare the initialload makefile to the makefile for this led_blink project, you will see that the word initialload has been replaced with led_blink. The Makefile itself is a very powerful tool that can be used to compile even very large projects, however in the end it comes down to being just a list of commands that can be chained together using targets.

When you run 'make' from the command line, it automatically looks for the Makefile and runs the 'all' target

The 'all' target has one dependency. The makefile will make make sure that those dependencies are met before executing the commands that belong to that target. So since led_blink-upload is a dependency, it jumps to that target.

The led_blink-upload also has one dependency. It also has one command that belongs to it, but make knows that it can't execute that command until it goes through its dependencies. So, make jumps to the led_blink.hex target

The top defines variables, which are just strings of text that get replaced in later in the file. They are useful for defining flags that get passed to commands later in the Makefile.



```
GCCFLAGS=-g -Os -Wall -mmcu=atmega168
LINKFLAGS=-Wl,-u,vfprintf -lprintf_flt -Wl,-u,vfscanf -lscanf_flt -lm
AVRDUDEFLAGS=-c avr109 -p m168 -b 115200 -P /dev/ttyUSB0
LINKOBJECTS=../libnerdkits/delay.o ../libnerdkits/lcd.o ../libnerdkits/uart.o

all:	led_blink-upload

led_blink.hex:	led_blink.c
	make -C ../libnerdkits
	avr-gcc ${GCCFLAGS} ${LINKFLAGS} -o led_blink.o led_blink.c ${LINKOBJECTS}
	avr-objcopy -j .text -O ihex led_blink.o led_blink.hex

led_blink.ass:	led_blink.hex
	avr-objdump -S -d led_blink.o > led_blink.ass

led_blink-upload:	led_blink.hex
	avrdude ${AVRDUDEFLAGS} -U flash:w:led_blink.hex:a
```

The led_blink.hex target has one dependency led_blink.c, and if you wrote your led_blink.c file that file exists, so that dependency is met. Make then goes ahead and executes the commands for that target. These commands are regular command line commands you could run if you wanted to. Make is just a great way of automating all these command line tasks that need to be run together. This target first executes make for the libnerdkits libraries to make sure those are compiled and up to date. Then it calls avr-gcc to compile your code, and avr-objcopy to turn your code into led_blink.hex for the MCU. Notice that the “output” of this target makes sure that led_blink.hex exists. Once it is done with those three commands make backtracks and executes the target that it came from, in this case it is led_blink-upload. Note that its dependency was led_blink.hex, but we already took care of that, so now it is ready to upload. It executes the command which is a call to avrdude which uploads the .hex file to your MCU. Make then backtracks again and executes the all target now that its dependency is met, but since it has no commands under it, it just exits. Thus leaving your MCU with its shiny new code on it!

Blinking an LED – More about Make

As stated before, make (along with the Makefile) are just a very fancy way of grouping sets of command line commands together. More importantly though, it gives you ability to define dependencies between different parts of your compilation sequence. If you think about it, it takes a very neat “top down” approach to compiling your program. You attempt to run the last thing you will need first, in this case the upload target. Since uploading depends on the .hex file existing, it goes and takes care of that until all the dependencies are met. In fact, you can jump to a specific target in your Makefile without going through all of it by passing that argument to 'make'. For example, running 'make led_blink.hex' will run just that target, compile the program and just exit. Make is a great way of having all these tasks together and running parts or all of them when necessary. It is a great tool that developers use all the time, so it is never too soon to get comfortable with it.

Disclaimer: As stated before make and the Makefile are extremely powerful tools. This tutorial barely scratchers the surface of what make is capable of, and actually takes a rather simplistic view of it. Make can include if statements to compile differently based on certain parameters. Perhaps most importantly it takes time stamps into consideration when resolving dependencies so that it does not recompile a target, if none if its dependencies have changed. There is much more to learn about Makefiles, but for now this light introduction will do.

Continued Learning – Bitwise Arithmetic

Now that you have completed your two projects, we hope you feel a great sense of accomplishment, and that you are now hooked on digital electronics. Surely, by getting through the temperature sensor project you have gotten a feeling for how complicated these systems are. If you finished the temperature sensor, and you're still confused about a lot of things you did, don't worry. Now that you have an assembled NerdKit, you have in front of you a platform from which you can more easily learn the ins and outs of programming for embedded devices. Hopefully you at least have an idea of what questions you should be asking.

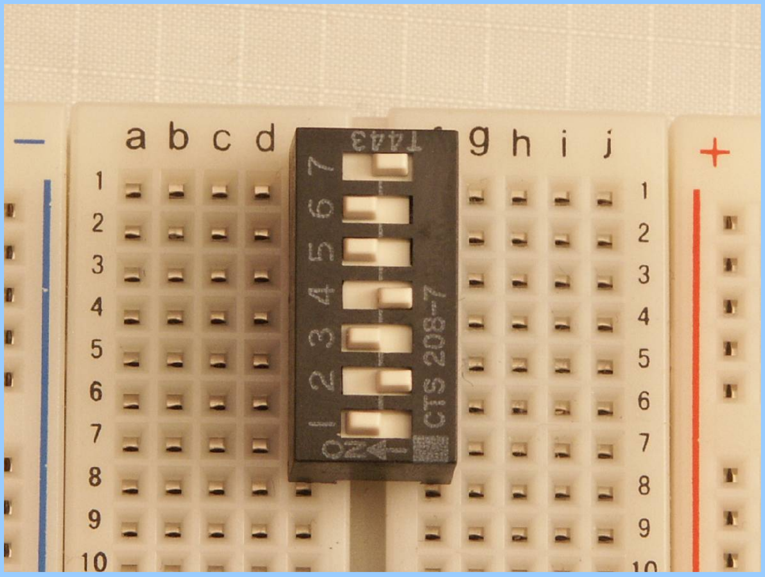
In the temperature sensor project, we taught you only what you needed to get things started and get a working system quickly. In the blink led project we explored using the MCUs pins as output pins. In this simpler project we are going to explore more of what you can do with the code and the MCU. We will also discuss the concept of the registers on the MCU.

For the project we are going to build a simple bitwise adder using the DIP switches included in your kit. We are going to use 6 of the switches to represent 2 different 3 bit numbers. By flipping the switches you are going to be able to change the binary representation of the numbers. We will then write code to read these as inputs and print on the LCD the decimal representation of the numbers and the sum of the two numbers.

Bitwise Arithmetic – Step 1

If you have the circuit set up from the temperature sensor we need to remove a few components to make room for the DIP switch on the board and on the MCU. Remove the temperature sensor.

The DIP switch has two rows of pins on either side. Insert the part into the board so that it straddles the middle groove of the breadboard. Make sure you connect it along 7 rows that are not being used by anything else. It is recommended that you connect it towards the top of the breadboard with the “On” arrow pointing towards the left.



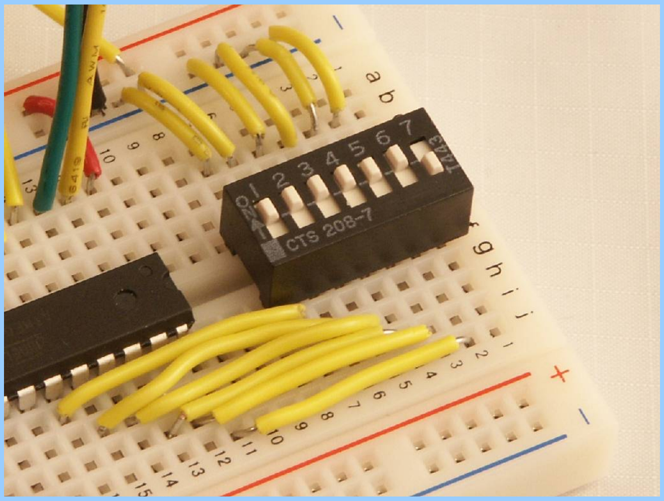
The DIP switch is a very simple device. Notice that there are seven little switches on the top, and seven pairs of pins along the bottom. When the switch is flipped up the two pins below that switch become electrically connected. By plugging in the switches so they bridge the left and right nodes on your breadboard, you have created a way to just flip a switch and connect the two nodes, then flip it back to disconnect them. This will be useful soon when we want to represent binary numbers as high and low voltages. We will use the dip switch to connect one of the pins to ground when we want the input to be low, and we will disconnect it when we want the microcontrollers built-in pullup resistor to pull the voltage up to +5.

Bitwise Arithmetic – Step 2

We are going to be representing two 3 bit numbers, so we need to connect 6 of the DIP switches to six empty pins on the microcontroller. From now on we are going to be referring to the DIP switches by their number: DP1 is the switch with the number 1 over it, DP2 is number 2 etc. You have to connect the "off" side of the switches to the microcontroller. That means the side that does not have the numbers on it has to be connected to the microcontroller. We chose to connect to the DIP switches in the following way.

DP1 -> MCU pin 23
DP2 -> MCU pin 24
DP3 -> MCU pin 25
DP4 -> MCU pin 26
DP5 -> MCU pin 27
DP6 -> MCU pin 28

All of these pins on the MCU should have been unused. Now connect the other side of each of the DIP switches to ground.



Let's think a little about why we just connected the circuit in this way. When you flip the switch "on" with the switch towards the numbers, it makes an electrical connection between its two pins. This makes a connection from the pin, all the way to ground, pulling the voltage on the pin down to 0. When you flip the switch to the "off" position, it breaks the connection to ground allowing the internal pull-up resistor to bring the voltage at the pin up to a high voltage.

Bitwise Arithmetic – Step 3

Now that we have the board set up, we have to write the code to interpret the values at the pins and do something with them. Working with microcontrollers is pretty low level, so in order to do this we are going to have to do some fairly tricky bitwise arithmetic. If you are not too familiar with the concept or bits and bytes, we recommend you watch the video “Bits Bytes and Binary” on the NerdKits web page.

The first thing we need to do is understand how to work directly with the pins of the microcontroller. This is explained on pages 73-76 of the datasheet of the MCU. Try reading it over and see how much you can understand. Reading the datasheet is always the first step in tackling a problem using a new electronic component.

The most important things about working with pins are as follows. Each individual pin on the microcontroller can be used for different things, so you have to use the code to tell the microcontroller how you want to use it. All of the communication with the microcontroller's logic is done through registers. Most registers, as far as your code is concerned, are simply variables that are 8 bit integers. For example, if you look on page 2 of the datasheet you will see that some of the pins on the right side are referred to as PC0-PC5. If you wanted to read the current values of these pins, that is see if they were high or low, you would want to read the value of the PINC register. PINC is a variable in your code that will be continuously updated to have the value of the pins. It's an 8 bit integer because it holds the value of 8 different pins. For example, if only PC0 was high the value of PINC would be 00000001 in binary, which translates to 1 in decimal. If PC3 and PC0 were the only ones high then the value of PINC would be 00001001 which translates to 9 in decimal.

Your MCU has a bunch of registers most of which you can either read or set to make your MCU behave in different ways, which is what makes it such a powerful and complicated chip. For now all you really need to know is that each set of pins is controlled by 3 registers: the DDRx, PORTx, and PINx where x can be either B, C, or D.

Bitwise Arithmetic – Step 4

The dip_arithmetic folder has the complete working code for this project. You should only use it if you are completely stuck. Instead work with the dip_arithmetic_edu folder. This folder contains a stub for dip_arithmetic.c that you can fill in with your code. Open up dip_arithmetic.c in Programmers Notepad, or your favorite text editor and get ready to code.

You should have seen these first two lines before. These two lines initialize the LCD and move the cursor to the first position.

```
int main() {
  // start up the LCD
  lcd_init();
  lcd_home();

  // Set the 6 pins to input mode - Two 3 bit numbers
  DDRC &= ~(1<<PC0); // set PC0 as input
```

The DDRC register tells the MCU if you are using the pins as inputs or outputs. Remember we are driving the pin high or low using the DIP switch, so we want to use the pin as an input to the MCU. Setting PCx of DDRC to 0 will configure that pin as an input pin. In this line we are setting the PC0 bit of DDRC to 0, which will set the PC0 pin as an input pin. You might think that we have done so in a very odd way. After all setting DDRC = 0 would set all the C pins to input. While this is true, we need to have a method for setting individual bits of a register to what we want, without changing the other ones. This line accomplishes exactly that. First notice that we have used the &= operator, this operator will set DDRC to be the bitwise AND of the old value and the value we are assigning it. Then we create the bit we want to set. First we left shift a 1 by PC0, this creates the number 00000001. If we had left shifted it by PC4 it would have created 00010000. It's just a 1 shifted over that many times. Then we use the NOT (~) operator to flip all the bits. Doing ~(1<<PC0) gives us 11111110, which is just the opposite of 00000001. Doing DDRC &= 11111110 will leave all the bits of DDRC unchanged except the zeroth bit which is set to 0. If you had done DDRC &= 11101111, it would leave all the bits unchanged except the fourth one, which would be set to 0.

This is a particularly hard concept to understand. You can think of registers as bins of information where 1 or 0 are stored that change the behavior of the chip. In the line above we wanted to set just one of the bins to 0, leaving the other ones unchanged. We had to use the &= operator, the NOT operator and the << (left shift) operator to be able to do that. If you do not understand how this works, ask! We are always here to help at support@nerdkits.com. Luckily you can follow this pattern whenever you want to set a bit to 0 without changing the other ones in a register. There is a different pattern we will follow when we want to set a bit to 1. Once you learn them you can just reuse the pattern.

In your code set PC0, PC1, PC2, PC3, PC4, and PC5 to 0 on the DDRC register., so that they are configured as input pins. Note that these are the pins to which you connected the DIP switches.

Bitwise Arithmetic – Step 5

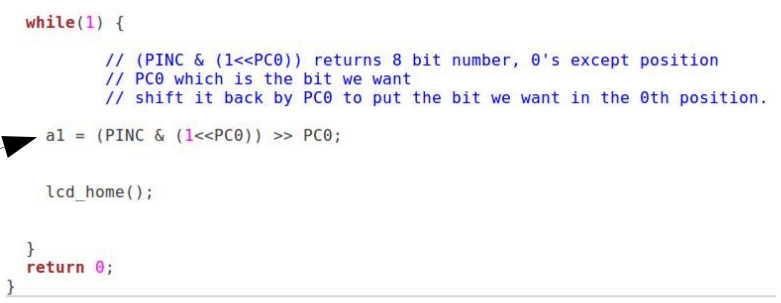
The PORTC register is used to turn the internal pull-up resistors on and off. (Note: this is only true if the pin is in input mode. When the pin is in output mode PORTC is used for something different.) We want to turn the internal resistors on for the pins we are using. If the PCx bit of PORTC is 1 then the pull-up resistor is turned on. This line is setting the PC0 bit of PORTC to 1 without changing the rest of the bits. This time we use the |= (OR EQUALS) operator. Shifting 1 by PC0 gives us 00000001. If your OR that with any 8 bit number it will leave all the bits unchanged except the first one. Use this pattern to set PC0,PC1,PC2, PC3, PC4, and PC5 of PORTC to 1.

```
int main() {
  // start up the LCD
  lcd_init();
  lcd_home();

  // Set the 6 pins to input mode - Two 3 bit numbers
  DDRC &= ~(1<<PC0); // set PC0 as input


  // turn on the internal resistors for the pins
  PORTC |= (1<<PC0); // turn on internal pull up resistor for PA0


  // declare the variables to represent each bit, of our two 3 bit numbers
  uint8_t a1;
```

Here we are declaring a variable that we will use later. It is declared as a uint8_t which means it is an unsigned 8 bit integer. I have called it a1 because it will be the first digit of one of my numbers. Declare 5 other variables of the same type, one for each digit of the numbers we are going to be working with.

Bitwise Arithmetic – Step 6

Next we start a while loop that will run forever. This is where we will continuously read the values coming in from the DIP switches, and then print stuff to the LCD.



```
while(1) {

        // (PINC & (1<<PC0)) returns 8 bit number, 0's except position
        // PC0 which is the bit we want
        // shift it back by PC0 to put the bit we want in the 0th position.

  a1 = (PINC & (1<<PC0)) >> PC0;

  lcd_home();

}
return 0;
}
```

On this line, we are grabbing the value for one of the bits. The PINC register contains the current value of each of the C pins. Reading a 1 on the PC0 bit will mean that the PC0 pin is high, and 0 will mean that it is low. In order to read a single bit we again have to do some bitwise arithmetic. First we create a number with a 1 in the place we want. 1<<PC0 creates 00000001. PINC & (1<<PC0) returns a number that is zero everywhere except the PC0 position. We then right shift that number back by PC0 so that it will be in the first spot in the number. This last step is unnecessary for PC0 because it is the first one, however it will be necessary for the other digits. For example, if you wanted to read the PC2 pin, you would first left shift a 1 by PC2; that would give you 00000100. When you & that with the PINC register it will leave all zeros except whatever the value of the pin PC2 is in that position; 00000x00 (where x is 1 or 0 depending on the value on the pin). You then need to right shift the number by PC2 to give you 0000000x. The result will be either the number 1 or the number 0 depending on the value of the pin

Set the other 5 variables you defined equal to the value of the other 5 pins. You now have 6 different variables each representing the value of 1 pin.

Now that you have the pin values you need to display that to the screen. Use what you have learned to print out the values of the different bits. First try displaying just one of the bits. See if you can flip one of the DIP switches and watch the display change. Then try representing three of the pins as one number. You will have to left shift and add to make this work. Finally try representing two different numbers and adding the results. The code provided represents two 3 bit numbers, prints them, then also displays the sum of the numbers. Take a look at it if you are stuck. Also remember you can email support@nerdkits.com for extra help or explanations. Hint: use lcd_write_int16(); it prints a number to the lcd.

The final result of this project does not seem that complicated. However once you have completed it you should have gained a lot of knowledge about working with microcontrollers. Setting and reading pins is the only way the MCU can communicate with outside components, but once you can do so you have the ability to change and react based on what is going on the outside world! This is the idea behind most electronic components. You now have the power to read individual pins and do something with that value. Once you understand this concept the possibilities are endless.

What's Next

Now that you have finished The NerdKits Guide here are some other things you might want to think about to further your journey in electronics.

Sensors:
- Photodectors
- Buttons
- Keypads
- Accelerometers and gyroscopes
- Magnetic sensors

Actuators:
- Motors
- Solenoids
- Servo Motors
- Relays
- Speakers and Buzzers
- Fans
- Electromagnets
- Thermoelectric heating and cooling elements

Parts Vendors:
- DigiKey - www.digikey.com
- Mouser - www.mouser.com (no minimum orders)
- Jameco - www.jameco.com
- Herbach and Rademan - www.herbach.com
- McMaster-Carr - www.mcmaster.com (mechanical parts)
- All Electronics - www.allelectronics.com
- Advanced Circuits - www.4pcb.com
- MPJA - www.mpja.com

Books and Further Reading
- “The Art of Electronics” by Horowitz and Hill
- MIT 6.002 (Circuits and Electronics) on MIT OCW
http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-002Spring-2007/VideoLectures/index.htm
- A brief overview of C from MIT 6.004 (Computation Structures)
http://6004.csail.mit.edu/currentsemester/handouts/c_overview.htm

Final Words

With what you have learned we hope you have enough knowledge to begin tackling your own projects. Feel free to explore, and make sure you send us pictures of anything cool you do to nerds@nerdkits.com. We are always here to help at support@nerdkits.com.

We hope NerdKits has given you enough to start thinking about new exciting projects you can undertake, but we also understand that this guide is only the first step.

We try to continuously add new educational materials, and new project ideas on our website. So check our website regularly, and make sure you look through our video tutorials http://www.nerdkits.com/videos/.

Appendix A: Using a DC Adapter with your NerdKit

Using the included 9V battery for your initial development work is highly recommended, because it provides a controlled environment for testing your kit. It also significantly limits the possible damage you can do to your components, or even your computer! However, if you'd like to switch out to a "wall wart" DC adapter, there are a few things you should keep in mind.

First, the input to the 7805 voltage regulator can be roughly in the range of 7V to 30V. Therefore, it's best if you have a DC adapter that outputs perhaps 9V or 12V.

Second, there is no protection for reverse voltage. Connecting your DC adapter backwards will probably destroy your NerdKits components. Also, some "wall wart" adapters actually have AC outputs, not DC. Check the labeling and confirm with a multimeter.

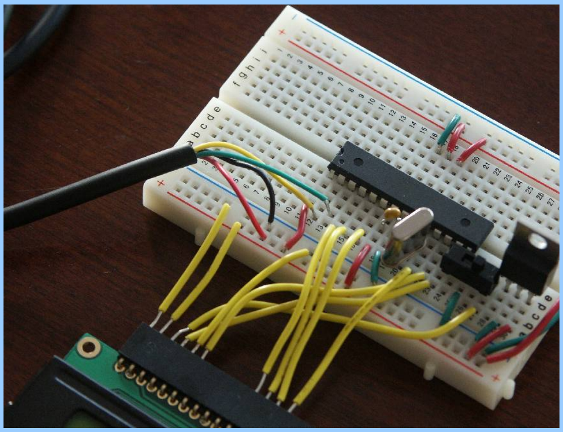
Third, you have to be mindful of isolation issues between different ground levels. Your computer has one GND voltage, and as long as your wall adapter provides "galvanic isolation" by using a transformer to allow for voltage offsets, this will be fine. But theoretically, due to bad design, or bad wiring in your home, the ground levels could be different -- leading to dramatic failures, and possibly damaging the kit or your computer. While we haven't personally experienced this, please note that it is possible and we can't take responsibility for it.

Fourth, you may need extra "bypass capacitance", as described in step 3b, because these power supplies can be more noisy.

Appendix B: Using USB Power

When connecting the USB programming cable, you connected the red wire to an empty row of the breadboard. The yellow and green wires are the data wires used to communicate with the chip. The black and red wires ahowever, are actually the USB power connections. The USB connector provides a 5V power supply that comes from your
computer. You can use this to power your NerdKit instead of the battery. To do this do the following:

1. Make sure the cable is disconnected from your computer.
2. Take the red wire of the programming cable, and connect it to the red rail.
3. Take the black wire and connect it to the blue rail.
4. As soon as you plug the programming cable in, your board will have power



Notice two things. First, you did not connect this new power source through the voltage regulator. This is because the USB power supply is 5V, and unlike the 9V of the 9V battery, it does not need to be regulated down to the 5V the MCU needs. Second, you are operating your circuit from your computers powers supply! This should be both intriguing, anda little unnerving. Although your operating system is probably smart enough to shut off your USB ports power supply if it is drawing too much current, there is no guarantee that it will do so. Take extra care when you are powering your devices off of your computer.

Appendix C: Bootloaders and More Microcontrollers

If you're ready to obtain more ATmega168-20PU microcontrollers so you can have multiple projects at once, be aware that we pre-program ours with a bootloader. This bootloader is a special piece of code that lets you use a simple method, like the USB to Serial cable, to upload your program to the chip.

The good news is that we include both the source code and compiled bootloader as part of the NerdKits Download. You can find these in the Code/bootloader directory.

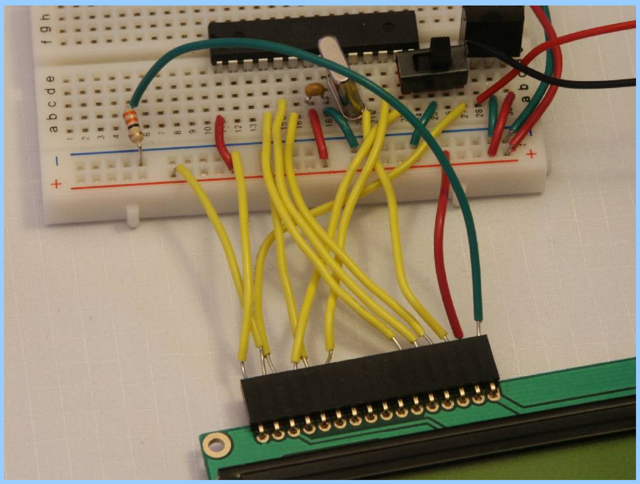
The bad news is that special programming devices are required to get the bootloader on in the first place. So, if you order additional ATmega168 chips from a normal electronics vendor, you'll also need to find a way to get the bootloader in place. If you have a parallel port in your computer, e-mail us and we might be able to provide a "hack" which would allow you to program them. If not, you'll have to consider third-party ISP programmers, such as the ATAVRISP2 from Atmel. If you have specific questions about this topic, feel free to e-mail us at support@nerdkits.com

At NerdKits, we can also offer you pre-programmed microcontrollers, tested LCDs, and other parts. Check out our store at http://www.nerdkits.com/store/. If you can't find what you are looking for, contact us and let us know what you're interested in.

Appendix D: The LCD Backlight

The LCD that comes with your kit has an LED backlight. Before using it, you have to make sure that you are running your kit off of on an AC adapter. The LED Backlight draws too much current for the battery.

In order to hook up the backlight, hook up LCD pin 15 to the +5 V rail. Then, using a 33 ohm resistor connect pin 16 of the LCD to GND in series with the 33 ohm resistor.



How did we pick 33 ohms? Our LCD manufacturer told us that the LED backlight operated at a forward voltage of 4.2V (probably with two LEDs in series), and could withstand a maximum current of 180mA. Since we know that V=IR, and we know what V is, we can choose R to set the correct I. To save power, we decided to operate at a lower brightness and a lower current. With a 33 ohm resistor, how much current flows through the backlight?

(5.0 - 4.2) / 33 = 0.0273 = 27.3 mA