Chapter 22

ELECTRONICS: INTERACTING WITH THE PHYSICAL WORLD

Many scientific computing tasks require more than reformatting and analyzing data. The data have to get into your computer in the first place, usually through various sorts of automated sensors operating out in the physical world. Although some types of sensors are extremely sophisticated and must be purchased as part of expensive instruments, some of the most common data acquisition tasks, such as monitoring temperature and other environmental variables, can be accomplished with simple systems that you build yourself—and that is the focus of this chapter. The ability to create your own sensor systems and custom electronic devices to interact with the physical world can be one of the most rewarding aspects of scientific computing. It is often far less expensive to do it yourself than to purchase off-the-shelf systems; moreover, you can customize your equipment in ways that would never be possible with prefabricated systems. Lastly, since so many devices communicate with serial ports, we provide background on enabling and troubleshooting serial communication.

Custom electronics in biology

Typical scenarios for custom electronics in biology

There are many situations where building your own electronic devices can greatly facilitate your biological research. For example:

- You are concerned that the temperature of your freezer is unstable, but don't want purchase an overpriced $400 Internet-enabled temperature monitor. You put a temperature probe in the freezer and wire it to a custom circuit attached to a computer adjacent to the freezer. With a few lines of computer code, you program the circuit to send the temperature of the freezer to the computer every minute. You then program the

computer to save these data to a database and send a message to your mobile phone via email if the temperature goes outside a specified range.

- You are column-purifying DNA extractions, which requires switching collection tubes every twenty drops. You don't have a fraction collector, and with more than a minute between each drop you would be watching dripping columns all day. You arrange a light-emitting diode and phototransistor so that each drop interrupts the path of light between them, and wire them to a counting circuit. Now you can leave the setup unattended for long intervals of time.
- Your experiments require monitoring solar radiation and oxygen levels during the course of an experiment at a remote field site. You wire several sensors to a self-contained circuit that can convert the sensor data to numbers and store them on a memory card.
- You want to monitor depth and orientation of a marine turtle as it forages, so you create a circuit to log data from a pressure sensor and from an electronic compass, then embed the device into epoxy so it can be attached to the turtle's shell. To download the data afterward, you use a wireless transmitter built into the circuit.
- You need a programmable agitator that can tip a rack of tubes at different intervals at different times of the day. You attach a $10 hobby servo (a special type of motor) to the rack, wire it to an actuator circuit, and program it to turn the servo at the appropriate time. The whole system costs $40, and once you're done using it, all the parts can be used for other things.

We won't cover all the skills needed for each of these examples, but we will get you pointed towards the basic tools. In particular we will focus on general-use microcontroller circuit boards.

Simple circuits with complex microcontrollers

Building custom electronic systems without extensive formal engineering experience is much easier now than it was in preceding decades. Forty years ago anyone building a circuit had to work with basic components such as resistors, transistors, capacitors, inductors, and diodes. This required a detailed knowledge of electrical engineering theory, extensive time commitment, and the experience to assemble tens or hundreds of components. Even then, the overall complexity of the circuit was by necessity limited, since it was built from the ground up. Over the last thirty years, however, a wide variety of microchips has become available to the general public. These microchips contain prefabricated circuits with tens to billions of components, and by containing all the components in one package, they greatly facilitate the construction of complex circuits.

Many microchips are designed to perform a specific task, but the last decade has seen the proliferation of microcontrollers, entire general-purpose computers on a

microchip. They are too slow and underpowered for complex data analysis tasks, and most don't even have enough computational power to interface with a screen and keyboard. However, they are very cheap (most are less than $10), use little electricity, are relatively simple to incorporate into circuits with other components and computers, and are designed to be flexible and easy to program. They can be connected to many types of actuators (such as motors and servos) and sensors. Microcontrollers are replacing specialized custom-built circuitry all around you, in washing machines, thermostats, electronic micropipettors, data loggers, flashlights, and just about anything with a battery or plug.

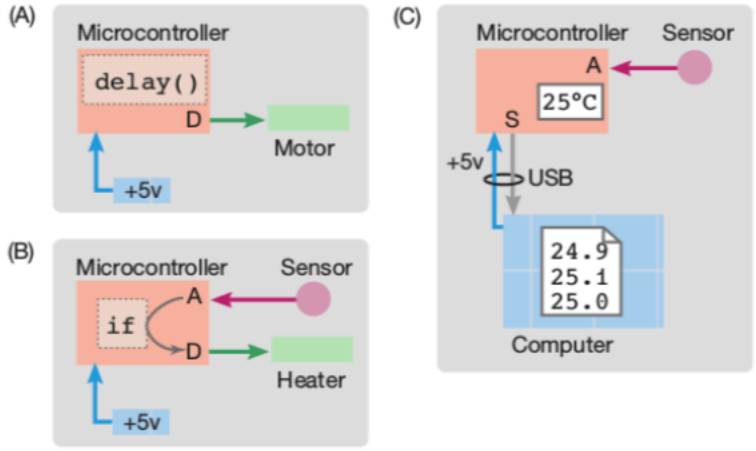


FIGURE 22.1 Common uses for microcontrollers in biological applications On the microcontroller block (orange) "D" represents digital output, "A" indicates an analog port, and "S" indicates a serial port. (A) This stand-alone circuit turns a motor a prescribed number of degrees after delays of specified intervals. No sensor is involved. (B) A stand-alone circuit monitors a temperature probe through an analog input, and uses the microcontroller's own program to determine when to turn on a heating element using the digital output when the temperature falls below a certain threshold. (C) A microcontroller as an external sensor for a computer. The microcontroller receives temperature information from a probe, converts this to numeric values, and sends the data via the USB port to the computer, where it is logged in a file. The USB connector also provides power to the microcontroller.

Microcontrollers fit right in with the other technologies we have chosen to include in this book—they are flexible tools that allow you to tackle a variety of challenges (Figure 22.1), and at the same time, working with them builds general skills. They can serve as interfaces to the physical world for your laptop or desktop computer, digitizing sensor data, relaying it to the computer (Figure 22.1C), and controlling actuators when instructed to do so. In such cases, they function as fully customizable computer peripherals. Alternatively, they can also be incorporated into stand-alone devices. In that role, they can monitor sensors and record data (e.g., a humidity logger), perform simple tasks based on sensor data (e.g., a drop counter or thermostat, as shown in Figure 22.1B), or serve as controllers for physical devices that need no sensory input (e.g., a tube rocker, as shown in Figure 22.1A).

Building a custom microcontroller-based instrument requires several steps:

- First, envisioning and designing the means of connecting the instrument to the physical world—identifying a sensor and how to connect with it.
- Second, wiring up the microcontroller chip with other components to build the sensor/actuator circuit.

- Third, connecting the microcontroller to a computer and programming it. Since the microcontroller has no screen or keyboard, all the software must be written on another computer and then transferred to the microcontroller memory.
- Fourth, for those situations where the instrument is to function as a computer peripheral, programming the attached computer to read and analyze the data.

In much the same way that software has become more usable and accessible to a wider audience as it has grown more sophisticated, complex microcontrollers are much easier for non-specialists to use than their simpler predecessors. This is because they contain advanced internal circuitry that reduces the complexity of the external circuits that need to be built around them. In the simplest cases, microcontrollers need only a battery and one or two other electronic components to work. In practice, they are most often used with a few more components than that, including components to regulate the power supply and protect input and output pins, a button to reset the software if needed, indicator lights, and connectors to interface with sensors and actuators.

The electronics community has designed simple circuit boards that contain microcontrollers and these other basic components. In less than five minutes such a board can be taken out of its package, connected to a computer, and set up to collect data, without even needing to warm up a soldering iron. The most popular of these standardized boards is the **Arduino** (`www.arduino.cc`), which comes in several sizes and shapes and with different sets of built-in components. The Atmega microcontrollers on Arduino circuit boards come pre-loaded with software that simplifies the programming process. Arduino is "open-source hardware." Like open-source software, all the designs are freely available, and the systems are built with the intention that people will modify them and use them in novel ways. Several manufacturers build boards that comply with the Arduino standards, and all the needed software for programming and using them can be freely downloaded.

Basic electronics

We don't aim to teach you electronics within this one chapter; we only present a few bare-bones essentials necessary to build the most basic microcontroller-based instruments. We strongly suggest that you consult the guides listed at the end of the chapter if you are interested in pursuing electronics and microcontrollers further.

Electricity

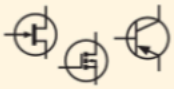
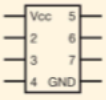
Unlike the Internet, an electronic circuit can be usefully thought of as a series of tubes through which fluid is flowing. This "fluid" is electrons, and some of the same intuitive concepts that apply to water in a pipe apply to electrons in a wire. The rate of electron flow in a circuit is called the **current** and is measured in amps (or often in

milliamps). The **potential** at any point in a circuit is analogous to pressure in a water pipe; it is measured in **volts**, and this is why potential is often called **voltage**. Power is a measure of the rate of energy flow. It is simply the product of the current times the voltage, and is measured in **watts** (or frequently in milliwatts).

Basic components

Understanding the nature of basic electronic components, and getting a sense of the vocabulary used to describe them, will help you get started not only with building your own circuits, but with interpreting circuits you find online and elsewhere. Table 22.1 introduces some of the more common components that you will see in schematics.

TABLE 22.1 Common electronic components

Symbol	Name	Properties and use
	Resistor	Restricts flow of electricity along a path
	Capacitor	Temporarily stores charge; dampens changes in voltage
	Diode	Allows current to flow in one direction; protects circuits; used to "rectify" alternating current to direct current; also used in voltage dividers
	LED	Diode that emits light
	Battery	Power supply to the circuit
	Ground	Common reference point for circuit potential voltage; equivalent to the negative terminal
	Transistor	Electronically switches or amplifies a signal
	Variable resistor	Restricts the flow of electricity in a flexible manner; can be controlled either mechanically with a knob or based on a signal like light or temperature
	Relay	Electromagnetic switch; can use a lower-power circuit to switch on much higher-powered circuits
	IC	Integrated circuit; packaged components to provide any of several functions, including timers, logic, and processing

A resistor is one of the most important electronic components. As its name implies, it resists the flow of electrons. It is analogous to a narrow section of a pipe. A higher voltage difference passed across the ends of a resistor (like a high pressure difference on either side of a pipe constriction) will lead to higher current through it, with the exact relationship depending on how much resistance is encountered. The unit of resistance is an **ohm**.

Encoding information with electric signals

Sensors are devices that convert a physical property into information encoded in an electronic signal. The properties that can be measured are nearly unlimited, and include temperature, force, pressure, light, orientation, conductivity, and chemical cues such as pH and oxygen concentration. Broadly speaking, the encoding of information into an electronic signal can be **analog** or **digital**. We will explain these terms shortly.

The specifications of data-acquisition devices like microcontrollers include descriptions of how many I/O (input/output) connections are available for each type of encoding. Often the same pin (i.e., connection) can be programmed to serve different functions. For example, a particular digital I/O pin could be configured to support simple ON/OFF input and output, to send text-formatted data via serial communication, or to produce variable output signals through pulse–width modulation.

Analog encoding

Analog signals are the most common starting point for simple sensors. Analog signals represent changes in a physical state as an electric property that varies continuously within a particular range (Figure 22.2A), usually current or voltage. Many sensors are essentially variable resistors that are sensitive to a particular environmental stimulus: the changing resistance is measured as a changing voltage by combining the sensor with resistors of fixed value and passing current through them. For example, in the case of a light sensing circuit, you might detect 0 volts in total darkness, 1.3 volts when a light bulb turns on, and 5.0 volts in full room light. Most sensors will saturate at some point, so in our example, you might continue to receive only 5.0 volts even as the light increases to full sunlight. Several analog-to-digital converters (also called A-to-D converters) are built into most microcontrollers. These convert the continuous range of

FIGURE 22.2 Common types of electric signals used to encode information (A) Analog signals can vary continuously. (B) A digital signal can only be on or off. (C) Serial signals are a special type of digital signal, where a series of on and off pulses represents more complex data, such as letters and numbers.

voltages into a numeric representation that can then be analyzed with software. For example, integer values between 0–255 might have a linear correspondence to the 0–5.0 volt values produced by your circuit, and that value would correspond in turn to light levels received at the sensor.

Digitally encoded signals

Digital signals have discrete rather than continuous states (Figure 22.2B). Usually these states are binary, that is, either off (zero volts) or on (often 5 volts for many devices). Just as there are many ways to represent information with an analog signal, there are a variety of approaches to encoding information with a digital signal. A few of these are described here.

Toggling When a digital signal is either on or off, it can be used directly to control simple devices. A digital output could be used to turn a single indicator light on or off, close a valve to stop the flow of gas, or open a door to introduce an animal into the experimental arena. Similarly, many types of input only have two states and are therefore readily encoded as a digital electronic signal. Such discrete events might include detecting when a firefly is flashing, whether or not a drop is passing by a sensor, or simply whether a button has been pushed.

All of these events can be controlled or measured with a single digital pin from a microcontroller. The microcontroller can be programmed to monitor when that pin is on or off, or to turn the pin on or off at particular times. Most microcontrollers will have a certain number of pins that can be used for digital I/O (input/output) of this sort. However, while dedicating an entire I/O pin to a particular piece of information works fine for simple applications, it is not practical for controlling devices that need to communicate large volumes of information about many different things. Such tasks are handled by the digital communications protocols described in the next two sections.

Parallel data transmission Most of the data inside your computer is transmitted within and between microchips as **parallel** digital information. A series of wires encodes numbers in binary (see Appendix 6), with each wire specifying a particular digit (1, 2, 4, 8, etc.) and with on corresponding to 1 and off corresponding to 0. These numbers can have literal numeric interpretations (e.g., integers or floats, as described in the earlier programming chapters), or they can stand for other types of information, such as ASCII characters (again, see Appendix 6). The number of wires used to transmit a value depends on how many bits (binary digits) the microprocessor employs in its calculations. In computers, 32-bit and 64-bit processors are standard. Most microcontrollers use only 8 bits, which is plenty for simple control and monitoring circuits, but not as suitable for complicated computing.

Computers used to have parallel ports for connecting external devices, mainly printers. However, this method of connection required one wire for each bit, resulting in bulky sockets and thick cables. The resulting cost and inconvenience has

ended the days of parallel ports in modern computers. Thus, although parallel data transmission is regularly used within devices, it is now rarely used between them.

Serial data transmission Serial data transmission is a close relative of parallel data transmission. Information is transferred as binary numbers encoded with on and off signals. Rather than send the signal for each digit in a separate parallel wire, though, all the signals are sent together in the same wire; they are staggered through time as a series of sequential pulses which encode information (Figure 22.2C). Common serial data interfaces include USB (Universal Serial Bus), Ethernet, FireWire, SATA (Serial-ATA, used for hard drives), and RS-232. Wireless serial data protocols include WiFi and Bluetooth. Serial data transmission is everywhere.

Serial data transmission may sound a lot more complicated than parallel data transmission, and in some respects, it is. Data must be packaged into bursts of signals, fed through a single wire, and then unpacked again on the other end. This all requires background processing. In addition, since each digit is sent single-file rather than in parallel, data transmission takes longer than it would with a parallel port with more wires. But these inconveniences have been easily surmounted as electronics have become cheaper and faster, and the huge savings of doing away with large connectors and cables with dozens of wires more than makes up for the other costs.

Sensors and instruments frequently come with circuitry already built in to convert measurements into serial data. Many of these devices use the RS-232 serial data interface, which is old and relatively slow, but also widespread and reliable. Few computers come with RS-232 ports anymore, but cheap RS-232-to-USB adapters are widely available. Serial data from an external port can be received directly from a special kind of terminal window in your computer, and it can also be accessed from within Python and other programs using easily installed libraries and modules, as we will describe shortly.

Pulse-width modulation PWM is another common way of encoding information. PWM is simply a series of pulses that occur at regular intervals, but whose duration is variable. At any point in time the signal is either on or off, but information is encoded as the width of the pulse rather than as binary numbers. If the interval between the pulses is short enough, PWM can be used to vary the average power to a device, just as with a continuous analog signal. For example, a light powered by a PWM circuit will appear brighter when the width of the pulse is longer. PWM is frequently used to control motors and other high-power actuators, since it is more efficient to rapidly turn the power on and off than it is to reduce the voltage when adjusting the power.

Building circuits

Schematics

The language used to communicate an electronic circuit is a two-dimensional diagram called a **schematic** (Figure 22.3A). This represents the wires and components and how they are interconnected. When you are building a circuit, you are translating the schematic diagram into actual connections with your physical components.

The schematic shown in Figure 22.3A is a very simple connection between a battery and a light-emitting diode (LED), including a resistor that is required to restrict the flow of current (without it, the LED would burn out). The schematic shows the three components of the system; in this case the components are labeled with words, but usually they will only be shown by symbols.

Many components have polarity, meaning that if they are put into a circuit backwards they won't work in the intended way. The symbols for polarized components are asymmetric, as is the structure or labeling of the components themselves. It is important to be sure that such components are oriented in the correct direction. LEDs, for instance, have a longer leg for the positive side of the circuit. Other components, such as resistors, have no polarity.

Breadboards

Once you have the design of a circuit, the next step is to translate it into an actual circuit and put together the electronic components. Some of the components you need may already be present on the microcontroller board, but in most cases you will need to supplement the board with at least a few more sensors and parts for your particular task. Electronic circuits are usually made by soldering together wires and components (soldering is an electronics version of welding); however, for testing a prototype and for debugging your circuits, you can use a device called a **breadboard**. This is a plastic rectangle filled with holes (Figure 22.3B,C). Behind the scenes, the holes are interconnected in such a way that wires and the legs of components inserted into them can be connected to other wires without soldering. There are several options for connecting your breadboard circuits to your microcontroller: (1) you can make connections between the microcontroller and breadboard with a few wires; (2) you can plug the entire microcontroller into the breadboard as if it were just another component (this works for small boards, like the Arduino Nano); or (3) you can get a breadboard "shield" that can piggy-back directly onto the microcontroller board.

In the breadboard depicted in Figure 22.3, the two rows of red and green holes along the top and bottom are connected along their length. A wire plugged into a red row will be connected to another wire plugged into any other red hole in that row, but not with the red row on the opposite side of the board. (The holes will

not appear colored on an actual breadboard—they are just shown that way in the illustration.) These rows are used to provide power and ground (that is, plus and minus from the battery or power supply) to all the places that power is needed in the board. Simply plug in a wire to one end of row and tap into it by plugging in the end of another wire anywhere else along that row.

The patches of holes near the center of the board are connected in columns, some of which are highlighted in blue in Figure 22.3. Any wire inserted into a hole in that section will be connected electrically to any other wire in that same column. Holes on opposite sides of the central divide are not connected to each other. These holes also allow you to connect to the wires of a chip (an integrated circuit) that is placed with two rows of pins straddling the central divider. Each pin of the chip becomes available to wires inserted in holes of the adjacent column.

Translating a schematic to a breadboard

Look at the breadboard circuit in Figure 22.3C and trace through the connections as they correspond to the schematic. The black wire coming from the negative terminal of the battery (outside the boundary of the image) is connected to a green hole, so anything plugged into the green row—such as the blue wire—is effec-

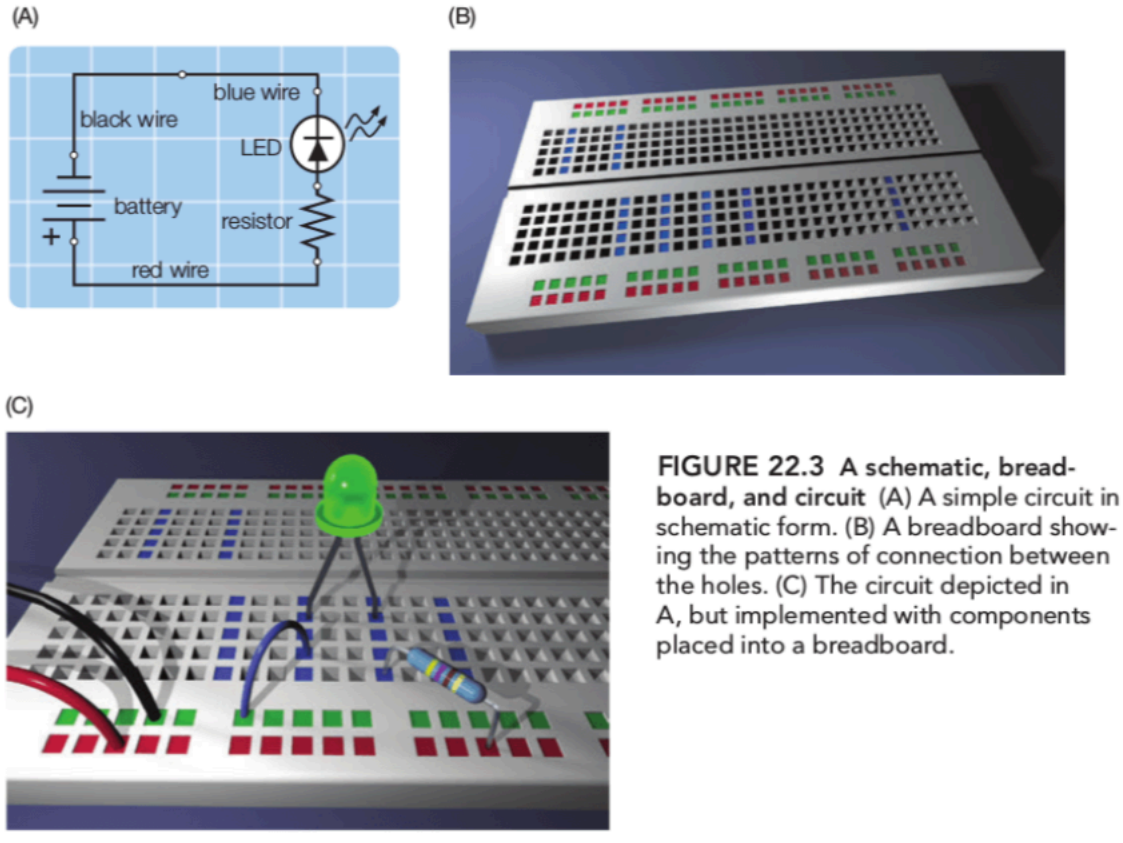


FIGURE 22.3 A schematic, breadboard, and circuit (A) A simple circuit in schematic form. (B) A breadboard showing the patterns of connection between the holes. (C) The circuit depicted in A, but implemented with components placed into a breadboard.

tively connected to the battery. The blue wire then connects to a column in the center of the board, and one leg of the LED is plugged into that same column. The other leg of the LED is inserted into the same row, but in a different column, so it is not directly connected to the blue wire. It is, however, connected to one end of the resistor inserted in the same column. The other end of the resistor is connected indirectly to the red wire (the positive battery terminal), because it is inserted into the same red row.

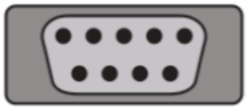
As shown, this circuit would cause the LED to light up continuously until the battery ran down. If you wanted to add a switch or control to the circuit, a good spot to do it would be in place of the blue wire. Breaking that link would cause the LED to turn off and no current to flow through the circuit.

Serial communication in practice

Unlike the stand-alone circuit above, your data-collection devices will probably be designed to interact with a microcontroller. In those cases, after you have built your electronic circuit and connected it to your microcontroller, you will need to connect the microcontroller to your computer's serial port. This is necessary to program it and may also be needed to operate it if you are building a device that passes data to the computer during normal operation. Some microcontroller boards have USB adapters built in, while others have RS-232 ports. Boards with USB ports can draw power from the same connection, obviating the need for a separate power supply when operating.

Serial communication is so widespread and critical to getting many types of physical devices to interact that here we provide a general background that extends beyond the topics needed to interface with microcontrollers. If you are fortunate, someone will have already built the exact instrument that you need for your research, and it will output your data in text format through a serial connection. With the right setup, you can connect a cable between your instrument and your computer, then use a variety of mechanisms and programs to capture the data stream to your computer for processing, storage, or display. (There are also options like Bluetooth, ZigBee, or XBee, which transmit serial data wirelessly.)

The traditional connector for serial communications to an instrument is a 9-pin RS-232 connector, shown at right, called DB-9 (not to be confused with an analog video connector, which looks similar but has three rows of pins instead of two). Of the nine connections, only three pins are required: two pins are used in communication, one for transmitting signals (`TX`) and one for receiving signals (`RX`), while the third pin is used for a common ground connection. On older PCs, these connectors are attached to serial port interfaces in the operating system called COM1 or COM2. Newer computers lack these ports, but USB-to-RS232 connectors make them available. When using one of these adapters, you will need to know the port address—that is, the place to send and receive data. With a USB-to-serial adapter installed, there will be what looks like a file in your `/dev` folder that acts as a socket for connecting to the serial port. On Linux or Mac



OS X, you can list the contents of your /dev folder and look for files named starting with tty.*:

Try /dev/tty*, /dev/ttyS*, or ttyusb0. Your device may be called ttyS0.

```
host:~ lucy$ ls /dev/tty.*
/dev/tty.Bluetooth-Modem    /dev/tty.usbserial-A70064y
/dev/tty.Keyserial1
```

The port for a serial adapter will only show up while it's plugged in, so you might not see any of these devices if you try this command now. You will use the appropriate path in programs or when trying to access your serial port through other data acquisition systems, as described shortly.

Baud rate and other settings

Serial communications protocols have a few options regarding what conventions to use when sending data back and forth. The most important option is the speed at which communication will occur, known as the **baud rate**. If data are being sent at a speed of 9,600 but the computer is set to receive at 19,200 baud, then you might get strange, unrecognizable characters on the screen. A good first step is to match the baud rate to the highest speed that is supported at both ends of the connection, and which can be transmitted the required distance without errors.

Other settings that affect serial communications include those which describe the nature of each data packet that is sent, including the number of bits, whether a parity bit is included (meaning one of the bits is used to detect errors in the other bits), and the number of stop bits (signals sent to mark the end of each series of bits). By far the most common setting for these is 8-N-1 (a standard abbreviation for the above settings), and, unless otherwise instructed, you should start with these options. **Handshaking**, the negotiation between devices about when and how serial data are transmitted, can be done either by the program (called software handshaking), by additional electronic lines (called hardware handshaking), or by built-in buffers on the devices (no handshaking). With this last option, one device sends the data and just assumes the other is listening, and the serial port itself stores as much data as it can until read by a program. Usually you can leave flow control off and rely on the buffers, but in the event that you need to turn it on, software flow control is often called XON/XOFF and hardware flow control is RTS/CTS. Although you will encounter a variety of baud rate settings, the data and flow values are almost always set to the defaults described here.

Null modem

The port on a computer's RS-232 connection will likely be set to use pin 3 of the connector to transmit data (TX) and pin 2 to receive (RX). In contrast, when an instrument is set to talk to a computer, its connector will have the pins in the opposite order: pin 3 will receive data and pin 2 will transmit. This way, all traffic from the computer goes out along pin 3 and all data from the instrument goes out on pin

2 (Figure 22.4A). If you try to connect two devices that both consider themselves to be master devices (or both slave devices), they will clash, with each transmit pin trying to push data to the other (Figure 22.4B). To solve this, you insert a **null modem** connecter between the two devices. This swaps pins 2 and 3 as they pass so that the two computers can talk to each other (Figure 22.4C). This situation sometimes occurs when you create your own device designed to operate by a serial connection or when you try to connect a laboratory instrument to a circuit designed to monitor or control its output.

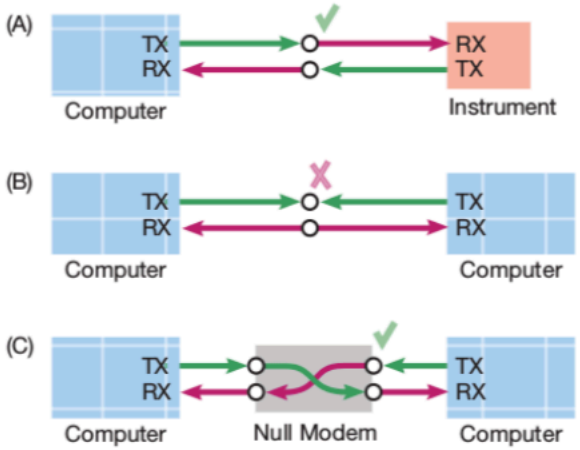


FIGURE 22.4 Serial connections and the use of a null modem connector

Software for serial communication

There are a few programs that will allow you to see serial data and send data to a serial device. On Windows, you can download the graphical applications PuTTY or Tera Term, or you can use Hyperterminal, which was installed as a utility prior to Windows 7. On Mac OS X, the most widely used graphical interface for serial communications is the venerable ZTerm. In a shell terminal window within OS X and Linux, you can use `kermit` or `minicom`, which may need to be installed.

Serial capabilities are also available in the shell environment of OS X via the built-in `screen` command. Here is an example session using the `screen` command in a terminal window; note that it only works if you have a serial-port adapter device plugged in:

```
host:~ lucy$ ls -a /dev/tty.*   ← Gives a list of devices
/dev/tty.Bluetooth-Modem     /dev/tty.Keyserial1
host:~ lucy$ screen /dev/tty.Keyserial1 19200 ← Begin communication at 19,200 baud
```

Now your screen will display any serial data coming through the serial adapter, and items that you type will be sent out through that serial port. (Often, instruments will take simple text commands to change their sampling frequency or other parameters.) To end the `screen` session, type ctrl A followed by ctrl \.

You can have a lot of fun with a cheap USB-to-RS-232 adapter and a cable, which will let you talk to instruments, sensors, and even freezers around the laboratory.

Serial comms through Python

Building a microcontroller-powered device is often only part of the mission. You may want to log information from your new device, or for that matter, from another sensor. To get more sophisticated with your serial communications, you can write Python programs to store and display serial data. To use serial communications in your programs, download and install the `pyserial` library from the

link at pyserial.sourceforge.net. Uncompress the archive and cd into the resultant directory in a terminal window. To install, type:

```
sudo python setup.py install    ← This will ask for your password
```

Note that the module you import into your program after installation is called serial, not pyserial. In a lot of ways, talking to a serial device is similar to opening a file for reading and then using the corresponding variable (below named Ser) as the source of data. Because a serial stream doesn't necessarily have a prescribed length, you will read from it with a while loop instead of a for loop, and program the loop to exit when a certain condition has been met or when a certain number of lines have been read. The serialtest.py program shown here, which is available in the pcfb/scripts example folder, reads data from the serial port and prints them to the screen:

```
#!/usr/bin/env python

# serialtest.py -- a demo of serial comms within python
# requires pyserial to be installed
import serial

# This address will not be the same
# find out your port name by using: ls /dev/tty.* and insert here
MyPort='/dev/tty.usbserial-08HB1916'

# be sure to put the timeout if you are using the readline() function,
#  in case the line is not terminated properly
# the other option is using ser.read(1), which reads one byte (=char)
# 19200 is the baud rate here.
Ser = serial.Serial(MyPort, 19200, timeout=1)
if Ser:
   i = 0
   # count lines and exit after 20
   while (i<20):
      Line = Ser.readline()   # read a '\n' terminated line
      print Line.strip()
      i += 1
   Ser.close
```

Arduino microcontroller boards in practice

Where to start

Arduinos are multipurpose microcontroller boards simple enough to be used in grade school projects, yet powerful enough to create complex and sophisticated ro-

bots, instruments, and devices. There are many excellent tutorials, including some that also teach basic principles of electronics. We recommend the official Arduino site `arduino.cc`, and `www.ladyada.net/learn/arduino`. Some books related to Arduino and electronics are also recommended at the end of this chapter.

The easiest way to begin with Arduino is to purchase a starter kit from Adafruit Industries (`www.adafruit.com`) or SparkFun Electronics (`www.sparkfun.com`). These kits include the microcontroller board, breadboards, wires, and sensors you will need to build simple projects. The supplied tutorials walk you through basic circuits and the software needed to run them. Once you have these example circuits running, you can start adding your own sensors and actuators and modifying and adding to the supplied software. This will allow you to build your skills incrementally.

Building circuits with Arduino

Arduino pins can be flexibly configured: Fourteen pins can perform digital input or output. Two of the digital pins can be used as `TX` and `RX` in serial communication, and six can generate pulse-width modulation output (simulating analog output). Six additional pins can be used for analog input, using 10 bits of information (see Appendix 6) to encode voltages from 0 to 5 volts as integer values from 0 to 1,023. The uses of these multipurpose boards are wide-ranging. Figure 22.5 shows a simple thermostat circuit, like that diagrammed in Figure 22.1B, which requires a single analog input for reading temperature and a digital output for toggling a relay that turns a heater on and off. Microcontrollers cannot directly supply enough electric power to devices that require large amounts of current, such as heaters or motors. Instead, they can control an intermediary device, such as a relay (see Table 22.1), to switch on and off the power supplied to the device.

There are a number of advanced sensors now available that enable surprisingly sophisticated yet simple-to-build projects. Types of sensors include temperature probes, accelerometers, gyroscopes, light sensors, strain gauges, proximity detectors, radiation sensors, magnetic field detectors, gas sensors, pressure sensors, flex

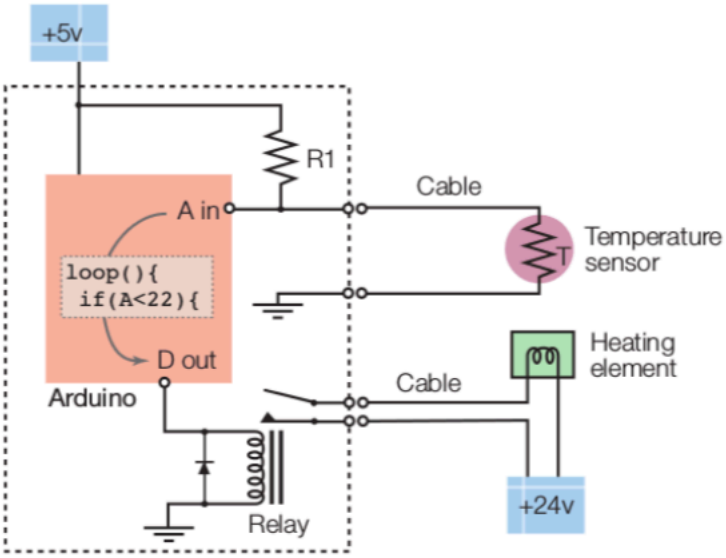


FIGURE 22.5 A simple thermostat circuit A sensor that changes resistance with temperature is combined with a resistor (R1) to convert this information to a changing voltage, which is then monitored with an analog input of the Arduino. A digital output controls a relay, which turns the heater on and off. The heater is supplied from a separate power supply that can handle higher voltage and current than the Arduino is capable of.

sensors, compasses, and microphones. Arduino software is available for most of these sensors, and hardware can be ordered from suppliers like SparkFun Electronics, so building projects with them can be just about plug-and-play.

Some Arduino microcontroller boards, including the Arduino Uno (`arduino.cc/en/Main/ArduinoBoardUno`), are designed to work with shields—premade circuit boards that piggyback on the microcontroller board. The breadboard shield for simple prototyping has already been mentioned. Other shields add functionality for connecting to the Internet via an Ethernet adapter, connecting to other Arduinos wirelessly, logging data to a flash memory card, motor and servo control, self-location with the Global Positioning System (GPS), and other tasks. These shields make a good starting point for many scientific tasks. For compact projects, the Arduino Nano is a fully functional unit with a reduced footprint that plugs directly into a breadboard.

Programming Arduino

An open-source Arduino programming tool (Figure 22.6) is available from the Arduino website (`arduino.cc/en/Main/Software`) for OS X, Linux, and Windows operating systems. This software package is used to compose programs, compile them, and transfer them to the Arduino, as well as monitor traffic through the Arduino's serial port. One of the nice things about Arduino, in addition to broad online support, is that there are many example scripts linked from within the program menus, so you are never far from seeing a working code sample.

The Arduino programming language is based on the language C, but it is not difficult to perform most operations that you have already seen demonstrated within Python. Being aware of a few structural differences will help you get up to speed more quickly. Arduino programs have two main sections: `setup()` and `loop()`. The `setup()` function is where you put commands that you want to run a single time when the board is powered up. Here you will configure how you want to use each of the pins, as well as set them to their initial states. You also need to set up serial communication if you would like to use that in the program:

```
int RedPin = 10;
int GrnPin = 8;
int InputPin = 14;
long previousMillis;

void setup()
{
  pinMode(RedPin, OUTPUT);    // set pin 10 as digital output
  pinMode(GrnPin, OUTPUT);    // set pin  8 as digital output
  pinMode(InputPin,INPUT);    // set pin 14 as digital input

  Serial.begin(19200);  // ...set up the serial output
  Serial.println("Starting...");
```

```
  digitalWrite(RedPin, LOW);   // LOW is predefined constant = 0
  digitalWrite(GrnPin, HIGH); // HIGH is predefined 1
  previousMillis = millis();   // Store current time
}
// continue with loop() below ...
```

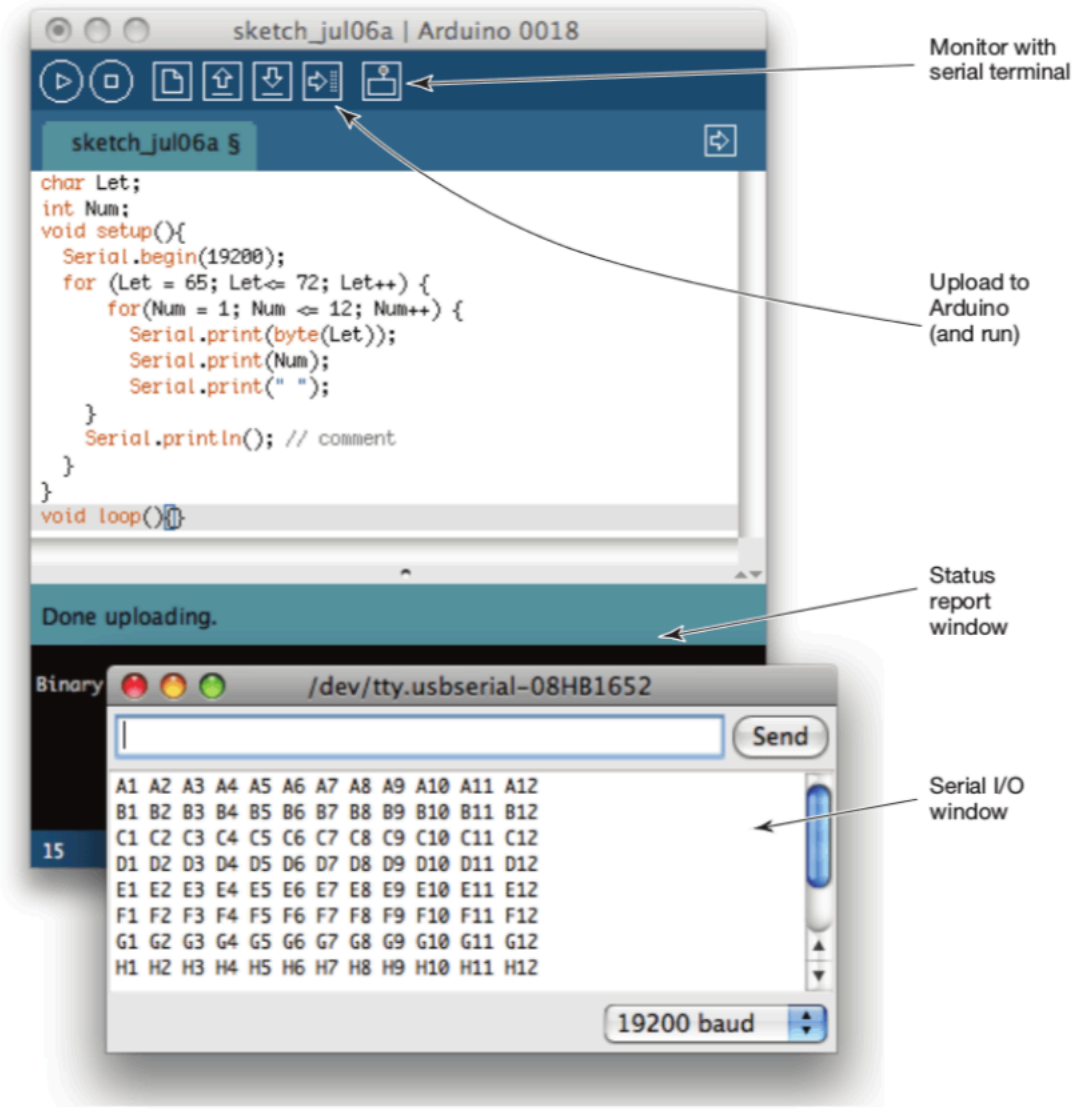


FIGURE 22.6 The Arduino programming interface showing an example program

Some things to notice about the Arduino language: Most lines need to end in a semicolon to mark the end of the command. Comments are indicated by // instead of #. Blocks for functions, loops, or logical expressions are bounded by curly brackets {} before and after the commands. Indentation does not matter, but it is still a good idea to use it to mark off sections of your program. Unlike in Python, you need to tell the system up front what kind of variable each name corresponds to; at that point you can also assign an initial value. If you are using an integer value, for example, you might say:

```
int X = 0;
```

Because of the amount of memory devoted to storing its value, an integer variable of type `int` in an Arduino program can only range from -32,768 to +32,767. If you add 1 to a value of 32,767, it will jump to the value of -32,768. To avoid problems like this, you can use an integer variable of type `long` instead, and this will let your values range above two billion. In Arduino operations, you often will need to monitor time in milliseconds. A regular integer would barely allow you to keep track of time for one minute, so `long` numbers become necessary even for relatively short intervals.

After the initial `setup()` function comes the main program `loop()`. The commands in this block are carried out repeatedly, until the Arduino is powered off. This is very different from the relatively linear Python programs you have been using, which may contain loops, but exit when the commands are completed. This intrinsic looping property makes Arduinos well suited to operations involving monitoring and sensing.

Many operations designed for use in the `loop()` portion of an Arduino program perform an operation once within a defined period (e.g., check the temperature once per 10 seconds). Instead of using a `delay()` function, which halts operation of the program until a prescribed interval has passed, it is better to allow the loop to run free, but check the passage of time and perform an operation when the interval has been exceeded. This basic convention is used in many loop-based systems, including LabVIEW, which we will introduce shortly.

Here is a fragment of code to achieve a timed loop:

```
int Interval = 500;  // time between operations -  doesn't change
long PreviousMillis = 0;  // current milliseconds - use long int

void setup(){
  //use the built-in millis() function to get the system time
  PreviousMillis = millis();
}
void loop()
```

```
{
   //check if an interval has passed. if not, continue looping
  if (millis() - PreviousMillis > Interval) {
    //we have passed the interval. do an operation here
    PreviousMillis = millis(); // redefine the reference time
    // continue the loop
}
```

If you would like to connect your Arduino creation to a graphical interface or a more complex data acquisition program than a simple serial monitor, you have several choices. You can connect to a programming environment called Processing (`processing.org`) that has high-level graphing capabilities; you can use some of the Python interface protocols (`arduino.cc/playground/Interfacing/Python`); or you can connect to just about any other data acquisition environment through a serial connection, supported by most Arduino devices.

Other options for data acquisition

MATLAB has built-in support for collecting data from a serial port (type `doc serial` from the MATLAB prompt), so it is possible to write your data capture, analysis, and presentation pipeline all within that environment.

LabVIEW by National Instruments (`ni.com`) is a powerful but expensive option for data acquisition. NI, as the company is known, sells many cards and interfaces for doing analog, digital, serial, and other types of data acquisition. Instead of using text like most languages, LabVIEW uses diagrams to wire together programming elements, as shown in Figure 22.7. Each variable (for example user input, output, a measured value, or a predefined constant) is represented by a box, and

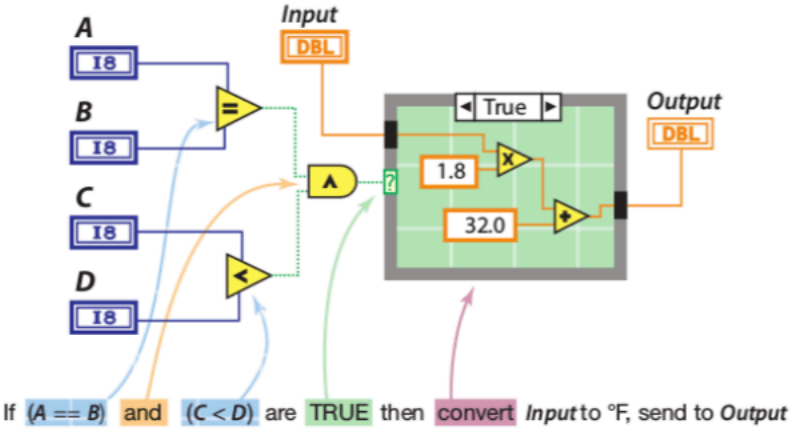


FIGURE 22.7 A fragment of LabVIEW code with an accompanying translation into more familiar programming terms

wires are used to represent connections between the variables, mathematical and logical operators, loops, and `if` statements.

The LabVIEW programming environment is a unique and capable way to rapidly develop a graphical interface for your instrument, including built-in modules for creating knobs and buttons, sliders, graphs, and indicators (Figure 22.8).

There are a few key concepts that differ for developing LabVIEW programs. As with the Arduino, most programs run continuously within a loop. In LabVIEW, loops are represented by a box, and everything in the box is executed repeatedly. LabVIEW itself figures out the sequence of events for your wired diagram, so you don't have to plan the order of operations. Values needed for later calculations are automatically obtained in the proper sequence. Global variables are also not usually used: instead, special operators called shift registers are placed at the left and right edges of the `while` loop. Variables wired to a shift register on the right side are passed back to the corresponding representation on the left, thus persisting through the next iteration of the loop.

Although LabVIEW hardware and the basic software are expensive (more than $1,000), there are academic licenses available to students and teachers which make the software available for less than $100. Because LabVIEW programs are inherently graphical in nature, this is a relatively easy way to incorporate a user-friendly interface into your data acquisition and control tasks, if that is a priority. Many

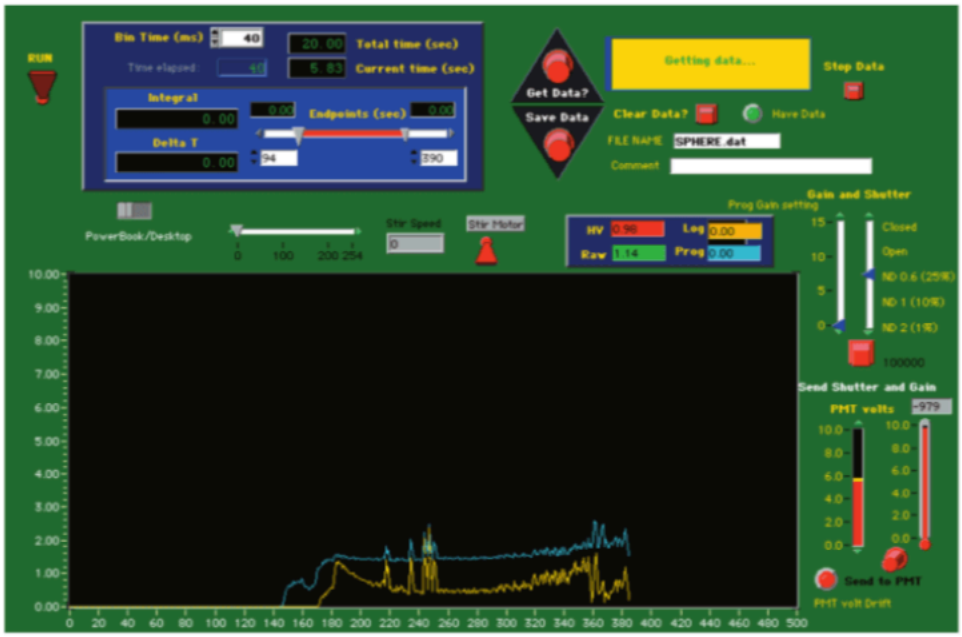


FIGURE 22.8 The graphical interface of a custom designed LabVIEW virtual instrument Each item on the front panel can be dragged from a palette, and then the elements are wired up on a diagram to provide input and output variables.

example programs (called VIs, or virtual instruments) can be downloaded from the online support community at `forums.ni.com`.

Common sources of confusion

When you work through an electronics book or tutorial, you may encounter a few common sources of confusion.

Measuring voltage

Like difference in pressures, voltage must be measured relative to a standard. The standard is usually called the **ground**, after the historical practice of using the Earth as a reference point for electric potential. Within a circuit, the ground, indicated diagrammatically by the symbol ⏚, is your reference point—the electronic equivalent of sea level. It corresponds to the negative terminal of a battery or power supply. The convention is that current flows in electrical circuits from the positive terminal to ground (but see the next section for the physical actuality that this convention simplifies).

You will often see apparently dead-end connections to the ground symbol all over the place in a circuit schematic. This doesn't mean those wires or terminals are unconnected at one end, or that there are several different grounds; it is just that they are all connected back to the negative power side. Showing all these wires would complicate the presentation of the schematic.

Current flow versus electron flow

It is sometimes confusing to think about the way that current flows through a circuit. In an electronics sense, it is said that positive charge flows from plus to minus—a convention dating back at least to Benjamin Franklin. However, in the physics sense, current is actually made up of electrons flowing from minus to plus, with positive "holes" (a lack of negative charges, like a double negative) flowing as a counter-current. As long as current is described in a manner which is internally consistent, this won't affect your circuit diagrams. The practical result, though, is that arrows in the schematic symbols for diodes, transistors, and other components may sometimes look like they are pointing in a direction opposite to the way you imagine current to be flowing (i.e., from positive to ground).

Pull-up and pull-down resistors

When you create a circuit, for example with an Arduino, you might assume that an unconnected input, since it has no voltage applied to it, would be off and have a logical value of zero (`LOW`). However, if you leave an input floating like this, it actually has an undetermined state. It will certainly go on (`HIGH`) when you apply a positive voltage, but it might also be high in this undetermined state.

To avoid this problem, you can connect the input to either +5 volts or the ground through a large resistance (typically 4,700 to 10,000 ohms), so that very little current flows through this part of the circuit. If you use a **pull-up resistor** (Figure 22.9A),

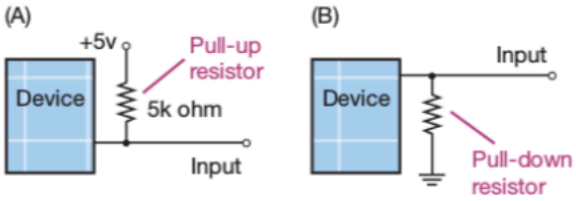


FIGURE 22.9 Pull-up and pull-down resistors

the input will be normally high, and you will have to apply a low signal (connect to ground) to switch the input. If instead you tie the input to ground through the resistor, (Figure 22.9B) it is known as a **pull-down resistor**, and the input will be held low until a sufficient positive signal is applied. In either case, because the resistance is so large, the signal from the pull-up or pull-down line will be overwhelmed by the sensor voltage when it is applied, and very little current will be drained from your circuit through this path.

SUMMARY

You have learned:

- How physical properties can be measured through the use of electronic sensors
- How information can be encoded as electric signals
- How a breadboard works to prototype a circuit
- The basic principles of serial communication
- That microcontroller boards like the Arduino can be used to build scientific instruments
- That LabVIEW is a powerful but expensive approach to data gathering
- How to avoid some common sources of confusion when starting out with electronics

Moving forward

- Get yourself an Arduino board from Adafruit Industries (`www.adafruit.com`) or SparkFun Electronics (`www.sparkfun.com`) and work through the tutorials online.
- Read about XBee and other wireless communication systems that can provide a wireless serial connection, sometimes over distances of more than a kilometer.
- Compare the specifications and capabilities of other data-acquisition systems, such as the SerIO board and PICaxe microcontrollers. These could be more suited to the demands of your project, such as having the ability to conserve power by sleeping and waking when needed.

- To test your circuits, get a multimeter, which measures resistance, voltage, and current. An oscilloscope, either as a stand-alone instrument or a USB-attached computer peripheral, will let you see plots of voltage over time with resolution of microseconds. This is very useful for determining what is going on behind the scenes in your circuits.
- Think about what measurements or experiments you perform that could be automated, or what sensors would make your field work easier. Work with a technician or try to come up with an electronics solution to the problem.

Recommended reading

References are listed with general texts at the top, and more specialized books at the bottom.

Mims, Forrest M. *Getting Started in Electronics*. Master Publishing, 2003.
A classic text that provides a friendly introduction to electronics, components, and circuits.

Mims, Forrest M. *Science and Communication Circuits & Projects*. Master Publishing, 2004.

Mims, Forrest M. *Electronic Sensor Circuits & Projects*. Master Publishing, 2004.
Both pamphlets by Forrest Mims include creative solutions to designing sensors and environmental monitors. Although these are now a bit old, they have clear explanations and still contain a great deal of useful information and clever ideas.

Scherz, Paul. *Practical Electronics for Inventors*. McGraw-Hill, 2006.
Provides a more in-depth and comprehensive treatments of electronics. (We did not just select this because we felt kinship with the title.) Be sure to get at least the second edition (2006) or later, as the first edition is reported to have a fair number of errors.

Margolis, Michael. *Arduino Cookbook*. O'Reilly Media, 2010.
This book contains a wealth of useful examples presented in Problem:Solution format. Each solution includes circuit diagrams and an example program, so that it can be rapidly implemented.

Igoe, Tom, and Dan O'Sullivan. *Physical Computing: Sensing and Controlling the Physical World with Computers*. Course Technology PTR, 2004.
Although this book does not talk about Arduino in particular, it covers a lot of the principles and concepts used when developing sensor devices and networks, and when working with microcontrollers.