UNIT-3

Prototying Embedded Device with ARDUINO

ELECTRONICS

Before we get stuck into the ins and outs of microcontroller and embedded computer boards, let's address some of the electronics components that you might want to connect to them.

Don't worry if you're scared of things such as having to learn soldering. You are unlikely to need it for your initial experiments. Most of the prototyping can be done on what are called solderless breadboards. They enable you to build components together into a circuit with just a push-fit connection, which also means you can experiment with different options quickly and easily

When it comes to thinking about the electronics, it's useful to split them into two main categories:

Sensors: Sensors are the ways of getting information into your device, finding out things about your surroundings.

Actuators: Actuators are the outputs for the device—the motors, lights, and so on, which let your device do something to the outside world.

Within both categories, the electronic components can talk to the computer in a number of ways. The simplest is through digital I/O, which has only two states: a button can either be pressed or not; or an LED can be on or off. These states are usually connected via general-purpose input/output (GPIO) pins and map a digital 0 in the processor to 0 volts in the circuit and the digital 1 to a set voltage, usually the voltage that the processor is using to run (commonly 5V or 3.3V).

If you want a more nuanced connection than just on/off, you need an analogue signal. For example, if you wire up a potentiometer to let you read in the position of a rotary knob, you will get a varying voltage, depending on the knob's location. Similarly, if you want to run a motor at a speed other than off or full-speed, you need to feed it with a voltage somewhere between 0V and its maximum rating.

Because computers are purely digital devices, you need a way to translate between the analogue voltages in the real world and the digital of the computer.

An analogue-to-digital converter (ADC) lets you measure varying voltages. Microcontrollers often have a number of these converters built in. They will convert the voltage level between 0V and a predefined maximum (often the same 5V or 3.3V the processor is running at, but sometimes a fixed value such as 1V) into a number, depending on the accuracy of the ADC. The Arduino has 10-bit ADCs, which by default measure voltages between 0 and 5V. A voltage of 0 will give a reading of 0; a voltage of 5V would read 1023 (the maximum value that can be stored in a 10-bits); and voltages in between result in readings relative to the voltage. 1V would map to 205; a reading of 512 would mean the voltage was 2.5V; and so on.

The flipside of an ADC is a DAC, or digital-to-analogue converter. DACs let you generate varying voltages from a digital value but are less common as a standard feature of microcontrollers. This is due to a technique called pulse-width modulation (PWM), which gives an approximation to a DAC by

rapidly turning a digital signal on and off so that the average value is the level you desire. PWM requires simpler circuitry, and for certain applications, such as fading an LED, it is actually the preferred option.

For more complicated sensors and modules, there are interfaces such as Serial Peripheral Interface (SPI) bus and Inter-Integrated Circuit (I2C). These standardised mechanisms allow modules to communicate, so sensors or things such as Ethernet modules or SD cards can interface to the microcontroller.

Naturally, we can’t cover all the possible sensors and actuators available, but we list some of the more common ones here to give a flavour of what is possible.

SENSORS

Pushbuttons and switches, which are probably the simplest sensors, allow some user input. Potentiometers (both rotary and linear) and rotary encoders enable you to measure movement.

Sensing the environment is another easy option. Light-dependent resistors (LDRs) allow measurement of ambient light levels, thermistors and other temperature sensors allow you to know how warm it is, and sensors to measure humidity or moisture levels are easy to build.

Microphones obviously let you monitor sounds and audio, but piezo elements (used in certain types of microphones) can also be used to respond to vibration.

Distance-sensing modules, which work by bouncing either an infrared or ultrasonic signal off objects, are readily available and as easy to interface to as a potentiometer.

ACTUATORS

One of the simplest and yet most useful actuators is light, because it is easy to create electronically and gives an obvious output. Light-emitting diodes (LEDs) typically come in red and green but also white and other colours. RGB LEDs have a more complicated setup but allow you to mix the levels of red, green, and blue to make whatever colour of light you want. More complicated visual outputs also are available, such as LCD screens to display text or even simple graphics.

Piezo elements, as well as responding to vibration, can be used to create it, so you can use a piezo buzzer to create simple sounds and music. Alternatively, you can wire up outputs to speakers to create more complicated synthesised sounds.

Of course, for many tasks, you might also want to use components that move things in the real world. Solenoids can by used to create a single, sharp pushing motion, which could be useful for pushing a ball off a ledge or tapping a surface to make a musical sound.

More complicated again are motors. Stepper motors can be moved in steps, as the name implies. Usually, a fixed number of steps perform a full rotation. DC motors simply move at a given speed when told to. Both types of motor can be one-directional or move in both directions. Alternatively, if you want a motor that will turn to a given angle, you would need a servo. Although a servo is more controllable, it tends to have a shorter range of motion, often 180 or fewer degrees (whereas steppers and DC motors turn indefinitely). For all the kinds of motors that we’ve mentioned, you typically want to connect the motors to gears to alter the range of motion or convert circular movement to linear, and so on.

If you want to dig further into the ways of interfacing your computer or microcontroller with the real world, the “Interfacing with Hardware” page on the Arduino Playground website (http://playground.arduino.cc//Main/InterfacingWithHardware) is a good place to start. Although Arduino-focused, most of the suggestions will translate to other platforms with minimal changes. For a more in-depth introduction to electronics, we recommend Electronics For Dummies (Wiley, 2009).

SCALING UP THE ELECTRONICS

From the perspective of the electronics, the starting point for prototyping is usually a “breadboard”. This lets you push-fit components and wires to make up circuits without requiring any soldering and therefore makes experimentation easy. When you’re happy with how things are wired up, it’s common to solder the components onto some protoboard, which may be sufficient to make the circuit more permanent and prevent wires from going astray.

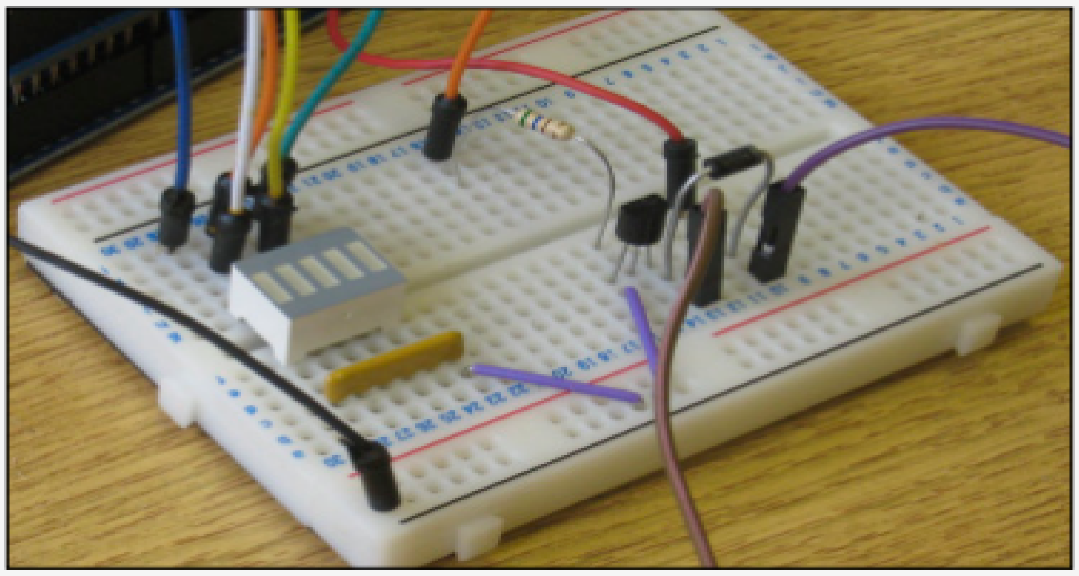
Moving beyond the protoboard option tends to involve learning how to lay out a PCB. This task isn’t as difficult as it sounds, for simple circuits at least,and mainly involves learning how to use a new piece of software and understanding some new terminology.

For small production runs, you'll likely use through-hole components, so called because the legs of the component go through holes in the PCB and tend to be soldered by hand. You will often create your designs as companion boards to an existing microcontroller platform—generally called shields in the Arduino community. This approach lets you bootstrap production without worrying about designing the entire system from scratch.

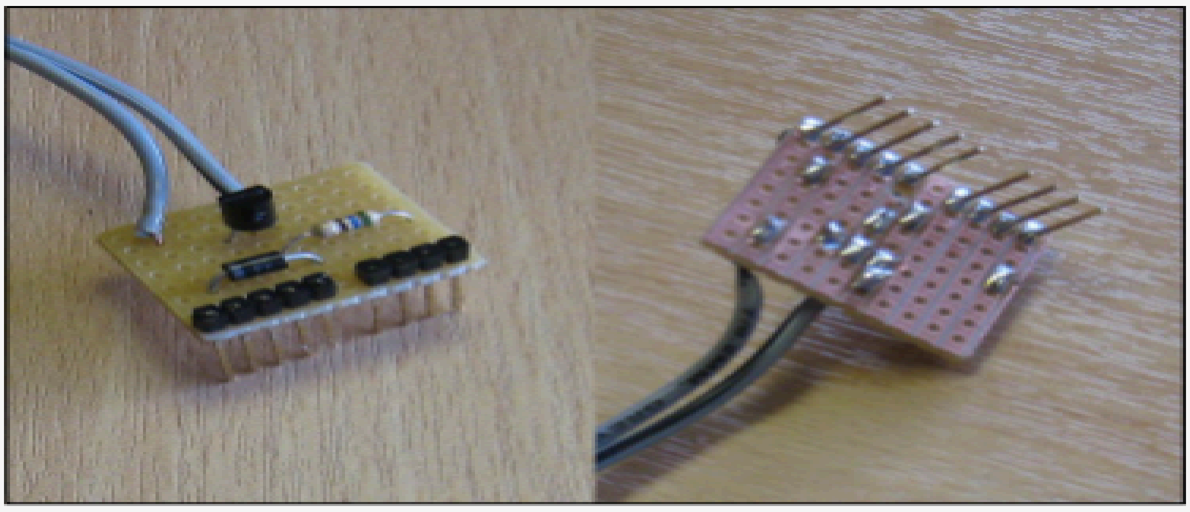
Journey to a Circuit Board

Let's look at the evolution of part of the Bubblino circuitry, from initial testing, through prototype, to finished PCB:

1. The first step in creating your circuit is generally to build it up on a breadboard. This way, you can easily reconfigure things as you decide exactly how it should be laid out.


The breadboard.

2. When you are happy with how the circuit works, soldering it onto a stripboard will make the layout permanent. This means you can stop worrying about one of the wires coming loose, and if you're going to make only one copy of the circuit, that might be as far as you need take things.



The stripboard.

3. If you need to make many copies of the circuit, or if you want a professional finish, you can turn your circuit into a PCB. This makes it easier to build up the circuit because the position of each component will be labelled, there will be holes only where the components go, and there will be less chance of short circuits because the tracks between components will be protected by the solder resist.

EMBEDDED COMPUTING BASICS

The rest of this chapter examines a number of different embedded computing platforms, so it makes sense to first cover some of the concepts and terms that you will encounter along the way.

Providing background is especially important because many of you may have little or no idea about what a microcontroller is. Although we've been talking about computing power getting cheaper and more powerful, you cannot just throw a bunch of PC components into something and call it an Internet of Things product. If you've ever opened up a desktop PC, you've seen that it's a collection of discrete modules to provide different aspects of functionality. It has a main motherboard with its processor, one or two smaller circuit boards providing the RAM, and a hard disk to provide the long-term storage. So, it has a lot of components, which provide a variety of general-purpose functionality and which all take up a corresponding chunk of physical space.

MICROCONTROLLERS

Internet of Things devices take advantage of more tightly integrated and miniaturised solutions—from the most basic level of microcontrollers to more powerful system-on-chip (SoC) modules. These systems combine the processor, RAM, and storage onto a single chip, which means they are much more specialised, smaller than their PC equivalents, and also easier to build into a custom design.

These microcontrollers are the engines of countless sensors and automated factory machinery. They are the last bastions of 8-bit computing in a world that's long since moved to 32-bit and beyond. Microcontrollers are very limited in their capabilities—which is why 8-bit microcontrollers are still in use, although the price of 32-bit microcontrollers is now dropping to the level where they're starting to be edged out. Usually, they offer RAM

capabilities measured in kilobytes and storage in the tens of kilobytes. However, they can still achieve a lot despite their limitations.

You'd be forgiven if the mention of 8-bit computing and RAM measured in kilobytes gives you flashbacks to the early home computers of the 1980s such as the Commodore 64 or the Sinclair ZX Spectrum. The 8-bit microcontrollers have the same sort of internal workings and similar levels of memory to work with. There have been some improvements in the intervening years, though—the modern chips are much smaller, require less power, and run about five times faster than their 1980s counterparts.

Unlike the market for desktop computer processors, which is dominated by two manufacturers (Intel and AMD), the microcontroller market consists of many manufacturers. A better comparison is with the automotive market. In the same way that there are many different car manufacturers, each with a range of models for different uses, so there are lots of microcontroller manufacturers (Atmel, Microchip, NXP, Texas Instruments, to name a few), each with a range of chips for different applications.

The ubiquitous Arduino platform is based around Atmel's AVR ATmega family of microcontroller chips. The on-board inclusion of an assortment of GPIO pins and ADC circuitry means that microcontrollers are easy to wire up to all manner of sensors, lights, and motors. Because the devices using them are focused on performing one task, they can dispense with most of what we would term an operating system, resulting in a simpler and much slimmer code footprint than that of a SoC or PC solution.

In these systems, functions which require greater resource levels are usually provided by additional single-purpose chips which at times are more powerful than their controlling microcontroller. For example, the WizNet Ethernet chip used by the Arduino Ethernet has eight times more RAM than the Arduino itself.

SYSTEM-ON-CHIPS

In between the low-end microcontroller and a full-blown PC sits the SoC (for example, the BeagleBone or the Raspberry Pi). Like the microcontroller, these SoCs combine a processor and a number of peripherals onto a single chip but usually have more capabilities. The processors usually range from a few hundred megahertz, nudging into the gigahertz for top-end solutions, and include RAM measured in megabytes rather than kilobytes. Storage for SoC modules tends not to be included on the chip, with SD cards being a popular solution.

The greater capabilities of SoC mean that they need some sort of operating system to marshal their resources. A wide selection of embedded operating systems, both closed and open source, is available and from both specialised embedded providers and the big OS players, such as Microsoft and Linux. Again, as the price falls for increased computing power, the popularity and familiarity of options such as Linux are driving its wider adoption.

CHOOSING YOUR PLATFORM

How to choose the right platform for your Internet of Things device is as easy a question to answer as working out the meaning of life. This isn't to say that it's an impossible question—more that there are almost as many answers as there are possible devices. The platform you choose depends on the particular blend of price, performance, and capabilities that suit what you're trying to achieve. And just because you settle on one solution, that doesn't mean somebody else wouldn't have chosen a completely different set of options to solve the same problem.

Start by choosing a platform to prototype in. The following sections discuss some of the factors that you need to weigh—and possibly play off against each other—when deciding how to build your device.

We cover the decisions that you need to make when scaling up both later in this chapter and in Chapter 10.

Processor Speed

The processor speed, or clock speed, of your processor tells you how fast it can process the individual instructions in the machine code for the program it's running. Naturally, a faster processor speed means that it can execute instructions more quickly.

The clock speed is still the simplest proxy for raw computing power, but it isn’t the only one. You might also make a comparison based on millions of instructions per second (MIPS), depending on what numbers are being reported in the datasheet or specification for the platforms you are comparing.

Some processors may lack hardware support for floating-point calculations, so if the code involves a lot of complicated mathematics, a by-the-numbers slower processor with hardware floating-point support could be faster than a slightly higher performance processor without it.

Generally, you will use the processor speed as one of a number of factors when weighing up similar systems. Microcontrollers tend to be clocked at speeds in the tens of MHz, whereas SoCs run at hundreds of MHz or possibly low GHz.

If your project doesn’t require heavyweight processing—for example, if it needs only networking and fairly basic sensing—then some sort of microcontroller will be fast enough. If your device will be crunching lots of data—for example, processing video in real time—then you’ll be looking at a SoC platform.

RAM

provides the working memory for the system. If you have more RAM, you may be able to do more things or have more flexibility over your choice of coding algorithm. If you’re handling large datasets on the device, that could govern how much space you need. You can often find ways to work around memory limitations, either in code (see Chapter 8, “Techniques for Writing Embedded Code”) or by handing off processing to an online service (see Chapter 7, “Prototyping Online Components”).

It is difficult to give exact guidelines to the amount of RAM you will need, as it will vary from project to project. However, microcontrollers with less than 1KB of RAM are unlikely to be of interest, and if you want to run standard encryption protocols, you will need at least 4KB, and preferably more.

For SoC boards, particularly if you plan to run Linux as the operating system, we recommend at least 256MB.

Networking

How your device connects to the rest of the world is a key consideration for Internet of Things products. Wired Ethernet is often the simplest for the user—generally plug and play—and cheapest, but it requires a physical cable. Wireless solutions obviously avoid that requirement but introduce a more complicated configuration.

WiFi is the most widely deployed to provide an existing infrastructure for connections, but it can be more expensive and less optimized for power consumption than some of its competitors

Other short-range wireless can offer better power-consumption profiles or costs than WiFi but usually with the trade-off of lower bandwidth. ZigBee is 98 Designing the Internet of Things one such technology, aimed particularly at sensor networks and scenarios such as home automation. The recent Bluetooth LE protocol (also known as Bluetooth 4.0) has a very low power-consumption profile similar to ZigBee’s and could see more rapid adoption due to its inclusion into standard Bluetooth chips included in phones and laptops. There is, of course, the existing Bluetooth standard as another possible choice. And at the boringbut-very-cheap end of the market sit long-established options such as RFM12B which operate in the 434 MHz radio spectrum, rather than the 2.4 GHz range of the other options we’ve discussed.

For remote or outdoor deployment, little beats simply using the mobile phone networks. For low-bandwidth, higher-latency communication, you could use something as basic as SMS; for higher data rates, you will use the same data connections, like 3G, as a smartphone.

USB

If your device can rely on a more powerful computer being nearby, tethering to it via USB can be an easy way to provide both power and networking. You can buy some of the microcontrollers in versions which include support for USB, so choosing one of them reduces the need for an extra chip in your circuit.

Instead of the microcontroller presenting itself as a device, some can also act as the USB “host”. This configuration lets you connect items that would normally expect to be connected to a computer—devices such as phones, for example, using the Android ADK, additional storage capacity, or WiFi dongles.

Devices such as WiFi dongles often depend on additional software on the host system, such as networking stacks, and so are better suited to the more computer-like option of SoC.

Power Consumption Faster processors are often more power hungry than slower ones. For devices which might be portable or rely on an unconventional power supply (batteries, solar power) depending on where they are installed, power consumption may be an issue. Even with access to mains electricity, the power consumption may be something to consider because lower consumption may be a desirable feature. However,However, processors may have a minimal power-consumption sleep mode. This mode may allow you to use a faster processor to quickly performoperations and then return to low-power sleep. Therefore, a more powerful processor may not be a disadvantage even in a low-power embedded device.

Interfacingwith Sensorsand Other Circuitry

In addition to talking to the Internet, your device needs to interact with something else—either sensors to gather data about its environment; or motors, LEDs, screens, and so on, to provide output. You could connect to the circuitry through some sort of peripheral bus—SPI and I2C being common ones—or through ADC or DAC modules to read or write varying voltages; or through generic GPIO pins, which provide digital on/off inputs or outputs. Different microcontrollers or SoC solutions offer different mixtures of these interfaces in differing numbers.

Physical Size and Form Factor

The continual improvement in manufacturing techniques for silicon chips means that we’ve long passed the point where the limiting factor in the size of a chip is the amount of space required for all the transistors and other components that make up the circuitry on the silicon. Nowadays, the size is governed by the number of connections it needs to make to the surrounding components on the PCB.

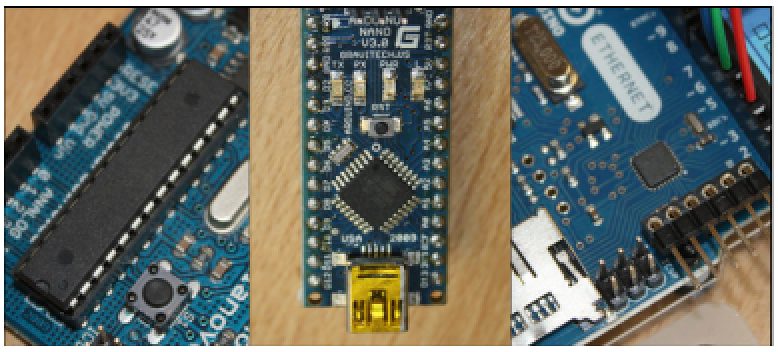
With the traditional through-hole design, most commonly used for homemade circuits, the legs of the chip are usually spaced at 0.1" intervals. Even if your chip has relatively few connections to the surrounding circuit—16 pins is nothing for such a chip—you will end up with over 1.5" (~4cm) for the perimeter of your chip. More complex chips can easily run to over a hundred connections; finding room for a chip with a 10" (25cm) perimeter might be a bit tricky!

You can pack the legs closer together with surface-mount technology because it doesn’t require holes to be drilled in the board for connections. Combining that with the trick of hiding some of the connections on the underside of the chip means that it is possible to use the complex designs without resorting to PCBs the size of a table.

The limit to the size that each connection can be reduced to is then governed by the capabilities and tolerances of your manufacturing process. Some surface-mount designs are big enough for home-etched PCBs and can be hand-soldered. Others require professionally produced PCBs and accurate pick-and-place machines to locate them correctly

Due to these trade-offs in size versus manufacturing complexity, many chip designs are available in a number of different form factors, known as packages. This lets the circuit designer choose the form that best suits his particular application.

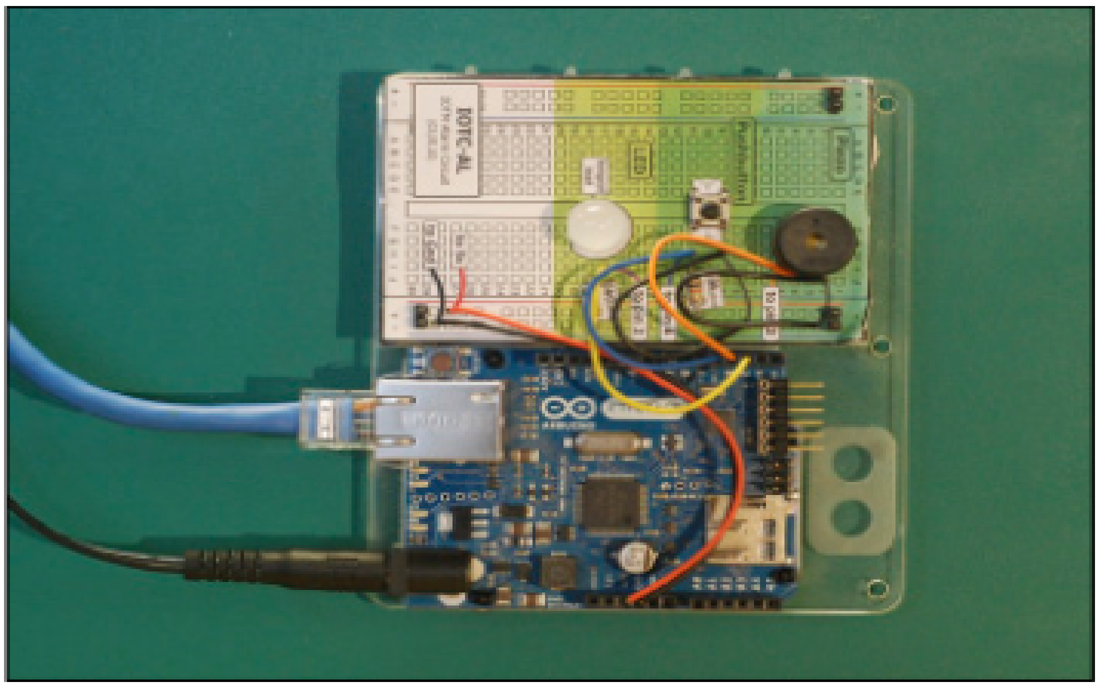
All three chips pictured in the following figure provide identical functionality because they are all AVR ATmega328 microcontrollers. The one on the left is the through-hole package, mounted here in a socket so that it can be swapped out without soldering. The two others are surface mount, in two different packages, showing the reduction in size but at the expense of ease of soldering.


Through-hole versus surface-mount ATmega328 chips

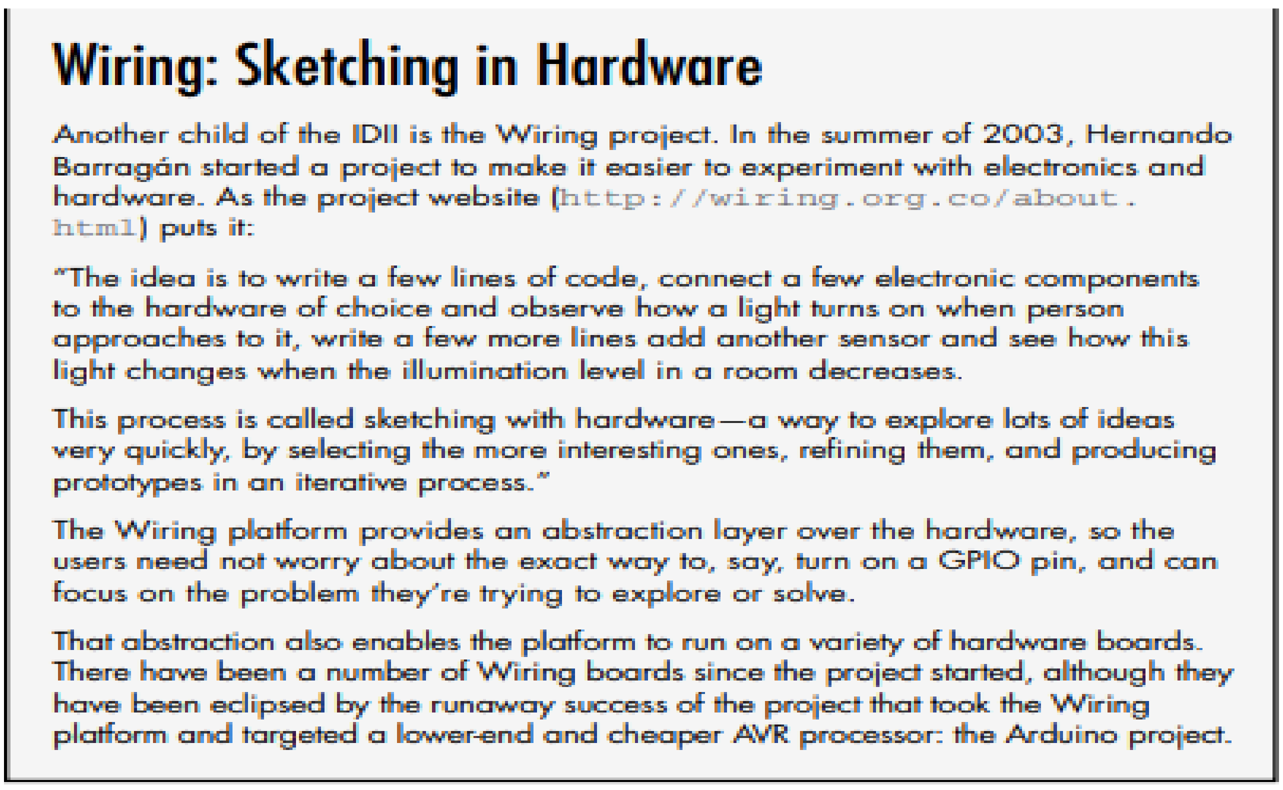
ARDUINO

Without a doubt, the poster child for the Internet of Things, and physical computing in general, is the Arduino.

These days the Arduino project covers a number of microcontroller boards, but its birth was in Ivrea in Northern Italy in 2005. A group from the Interaction Design Institute Ivrea (IDII) wanted a board for its design students to use to build interactive projects. An assortment of boards was around at that time, but they tended to be expensive, hard to use, or both.


An Arduino Ethernet board, plugged in, wired up to a circuit and ready for use.

So, the team put together a board which was cheap to buy—around £20— and included an onboard serial connection to allow it to be easily programmed. Combined with an extension of the Wiring software environment, it made a huge impact on the world of physical computing.



Wiring: Sketching in Hardware

Another child of the IDII is the Wiring project. In the summer of 2003, Hernando Barragán started a project to make it easier to experiment with electronics and hardware. As the project website (`http://wiring.org.co/about.html`) puts it:

"The idea is to write a few lines of code, connect a few electronic components to the hardware of choice and observe how a light turns on when person approaches to it, write a few more lines add another sensor and see how this light changes when the illumination level in a room decreases.

This process is called sketching with hardware—a way to explore lots of ideas very quickly, by selecting the more interesting ones, refining them, and producing prototypes in an iterative process."

The Wiring platform provides an abstraction layer over the hardware, so the users need not worry about the exact way to, say, turn on a GPIO pin, and can focus on the problem they're trying to explore or solve.

That abstraction also enables the platform to run on a variety of hardware boards. There have been a number of Wiring boards since the project started, although they have been eclipsed by the runaway success of the project that took the Wiring platform and targeted a lower-end and cheaper AVR processor: the Arduino project.

A decision early on to make the code and schematics open source meant that the Arduino board could outlive the demise of the IDII and flourish. It also meant that people could adapt and extend the platform to suit their own needs.

As a result, an entire ecosystem of boards, add-ons, and related kits has flourished. The Arduino team's focus on simplicity rather than raw performance for the code has made the Arduino the board of choice in almost every beginner's physical computing project, and the open source ethos has encouraged the community to share circuit diagrams, parts lists, and source code. It's almost the case that whatever your project idea is, a quick search on Google for it, in combination with the word "Arduino", will throw up at least one project that can help bootstrap what you're trying to achieve. If you prefer learning from a book, we recommend picking up a copy of Arduino For Dummies, by John Nussey (Wiley, 2013).

The "standard" Arduino board has gone through a number of iterations: Arduino NG, Diecimila, Duemilanove, and Uno.

The Uno features an ATmega328 microcontroller and a USB socket for connection to a computer. It has 32KB of storage and 2KB of RAM, but don't let those meagre amounts of memory put you off; you can achieve a surprising amount despite the limitations.

The Uno also provides 14 GPIO pins (of which 6 can also provide PWM output) and 6 10-bit resolution ADC pins. The ATmega's serial port is made available through both the IO pins, and, via an additional chip, the USB connector.

If you need more space or a greater number of inputs or outputs, look at the Arduino Mega 2560. It marries a more powerful ATmega microcontroller to the same software environment, providing 256KB of Flash storage, 8KB of RAM, three more serial ports, a massive 54 GPIO pins (14 of those also capable of PWM) and 16 ADCs. Alternatively, the more recent Arduino Due has a 32-bit ARM core microcontroller and is the first of the Arduino boards to use this architecture. Its specs are similar to the Mega's, although it ups the RAM to 96KB.

DEVELOPING ON THE ARDUINO

More than just specs, the experience of working with a board may be the most important factor, at least at the prototyping stage. As previously mentioned, the Arduino is optimised for simplicity, and this is evident from the way it is packaged for use. Using a single USB cable, you can not onlypower the board but also push your code onto it, and (if needed) communicate with it—for example, for debugging or to use the computer to store data retrieved by the sensors connected to the Arduino.

Of course, although the Arduino was at the forefront of this drive for ease-ofuse, most of the microcontrollers we look at in this chapter attempt the same, some less successfully than others.

Integrated Development Environment

You usually develop against the Arduino using the integrated development environment (IDE) that the team supply at http://arduino.cc. Although this is a fully functional IDE, based on the one used for the Processing language (http://processing.org/), it is very simple to use. Most Arduino projects consist of a single file of code, so you can think of the IDE mostly as a simple file editor. The controls that you use the most are those to check the code (by compiling it) or to push code to the board.

Pushing Code

Connecting to the board should be relatively straightforward via a USB cable. Sometimes you might have issues with the drivers (especially on some versions of Windows) or with permissions on the USB port (some Linux packages for drivers don't add you to the dialout group), but they are usually swiftly resolved once and for good. After this, you need to choose the correct serial port (which you can discover from system logs or select by trial and error) and the board type (from the appropriate menus, you may need to look carefully at the labelling on your board and its CPU to determine which option to select).

When your setup is correct, the process of pushing code is generally simple: first, the code is checked and compiled, with any compilation errors reported to you. If the code compiles successfully, it gets transferred to the Arduino and stored in its flash memory. At this point, the Arduino reboots and starts running the new code.

Operating System

The Arduino doesn't, by default, run an OS as such, only the bootloader, which simplifies the code-pushing process described previously. When you switch on the board, it simply runs the code that you have compiled until the board is switched off again (or the code crashes).

It is, however, possible to upload an OS to the Arduino, usually a lightweight real-time operating system (RTOS) such as FreeRTOS/DuinOS. The main advantage of one of these operating systems is their built-in support for multitasking. However, for many purposes, you can achieve reasonable results with a simpler task-dispatching library.

If you dislike the simple life, it is even possible to compile code without using the IDE but by using the toolset for the Arduino's chip—for example, for all the boards until the recent ARM-based Due, the avr-gcc toolset.

The avr-gcc toolset (www.nongnu.org/avr-libc/) is the collection of programs that let you compile code to run on the AVR chips used by the rest of the Arduino boards and flash the resultant executable to the chip. It is used by the Arduino IDE behind the scenes but can be used directly, as well.

Language

The language usually used for Arduino is a slightly modified dialect of C++ derived from the Wiring platform. It includes some libraries used to read and write data from the I/O pins provided on the Arduino and to do some basic handling for "interrupts" (a way of doing multitasking, at a very low level). This variant of C++ tries to be forgiving about the ordering of code; for example, it allows you to call functions before they are defined. This alteration is just a nicety, but it is useful to be able to order things in a way that the code is easy to read and maintain, given that it tends to be written in a single file.

The code needs to provide only two routines:

- setup(): This routine is run once when the board first boots. You could use it to set the modes of I/O pins to input or output or to prepare a data structure which will be used throughout the program.

loop(): This routine is run repeatedly in a tight loop while the Arduino is switched on. Typically, you might check some input, do some calculation on it, and perhaps do some output in response.

To avoid getting into the details of programming languages in this chapter, we just compare a simple example across all the boards—blinking a single LED:

```
// Pin 13 has an LED connected on most Arduino boards.

// give it a name:

int led = 13;

// the setup routine runs once when you press reset:

void setup() {

// initialize the digital pin as an output.

pinMode(led, OUTPUT);

}

// the loop routine runs over and over again forever:

void loop()

 {

digitalWrite(led, HIGH); // turn the LED on

delay(1000); // wait for a second

digitalWrite(led, LOW); // turn the

LED off delay(1000); // wait for a second

}
```

Reading through this code, you'll see that the setup() function does very little; it just sets up that pin number 13 is the one we're going to control (because it is wired up to an LED).

Then, in loop(), the LED is turned on and then off, with a delay of a second between each flick of the (electronic) switch. With the way that the Arduino environment works, whenever it reaches the end of one cycle—on; wait a second; off; wait a second—and drops out of the loop() function, it simply calls loop() again to repeat the process.

Debugging

Because C++ is a compiled language, a fair number of errors, such as bad syntax or failure to declare variables, are caught at compilation time. Because this happens on your computer, you have ample opportunity to get detailed and possibly helpful information from the compiler about what the problem is.

Although you need some debugging experience to be able to identify certain compiler errors, others, like this one, are relatively easy to understand:

```
Blink.cpp: In function 'void loop()':Blink:21:

error:'digitalWritee' was not declared in this scope
```

On line 21, in the function loop(), we deliberately misspelled the call to digitalWrite.

When the code is pushed to the Arduino, the rules of the game change, however. Because the Arduino isn't generally connected to a screen, it is hard for it to tell you when something goes wrong. Even if the code compiled successfully, certain errors still happen. An error could be raised that can't be handled, such as a division by zero, or trying to access the tenth element of a 9-element list. Or perhaps your program leaks memory and eventually just stops working. Or (and worse) a programming error might make the code continue to work dutifully but give entirely the wrong results.

If Bubblino stops blowing bubbles, how can we distinguish between the following cases?

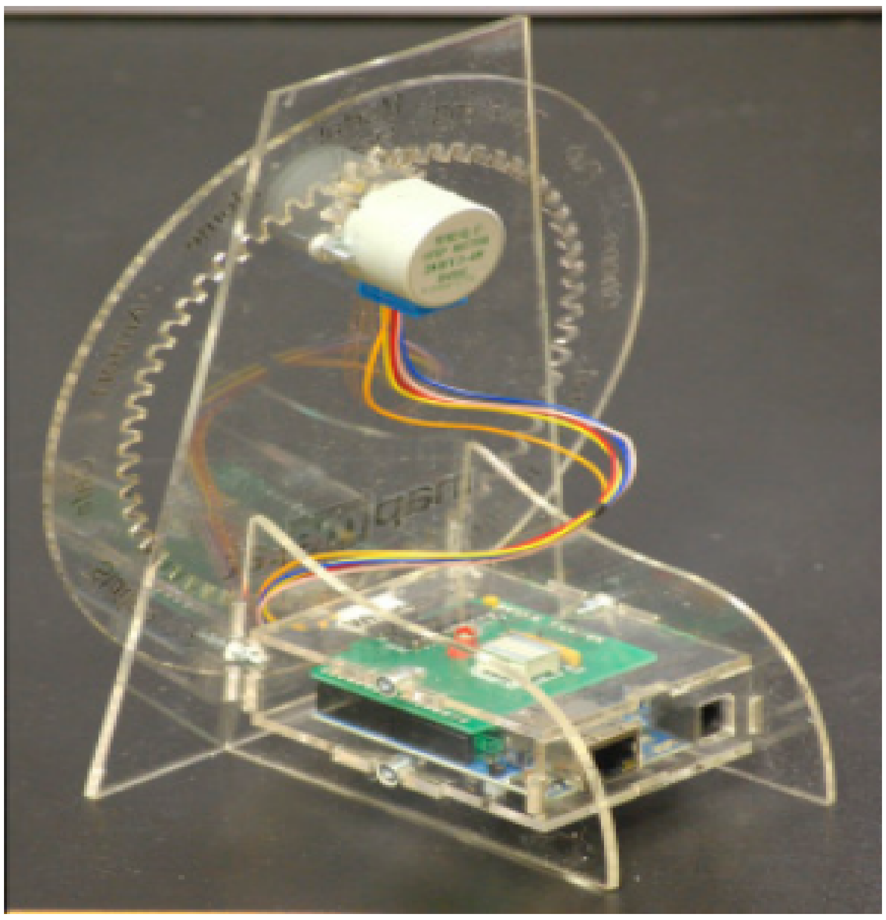
- Nobody has mentioned us on Twitter.
- The Twitter search API has stopped working.
- Bubblino can't connect to the Internet.
- Bubblino has crashed due to a programming error.
- Bubblino is working, but the motor of the bubble machine has failed.
- Bubblino is powered off.

Adrian likes to joke that he can debug many problems by looking at the flashing lights at Bubblino's Ethernet port, which flashes while Bubblino connects to DNS and again when it connects to Twitter's search API, and so on. (He also jokes that we can discount the "programming error" option and that the main reason the motor would fail is that Hakim has poured bubble mix into the wrong hole. Again.) But while this approach might help distinguish two of the preceding cases, it doesn't help with the others and isn't useful if you are releasing the product into a mass market!

The first commercially available version of the WhereDial has a bank of half a dozen LEDs specifically for consumer-level debugging. In the case of an error, the pattern of lights showing may help customers fix their problem or help flesh out details for a support request.

Runtime programming errors may be tricky to trap because although the C++ language has exception handling, the avr-gcc compiler doesn't support it (probably due to the relatively high memory "cost" of handling exceptions); so the Arduino platform doesn't let you use the usual try... catch... logic.

Effectively, this means that you need to check your data before using it: if a number might conceivably be zero, check that before trying to divide by it. Test that your indexes are within bounds. To avoid memory leaks, look at the tips on writing code for embedded devices in Chapter 8, "Techniques for Writing Embedded Code"



Rear view of a transparent WhereDial. The bank of LEDs can be seen in the middle of the green board, next to the red “error” LED.

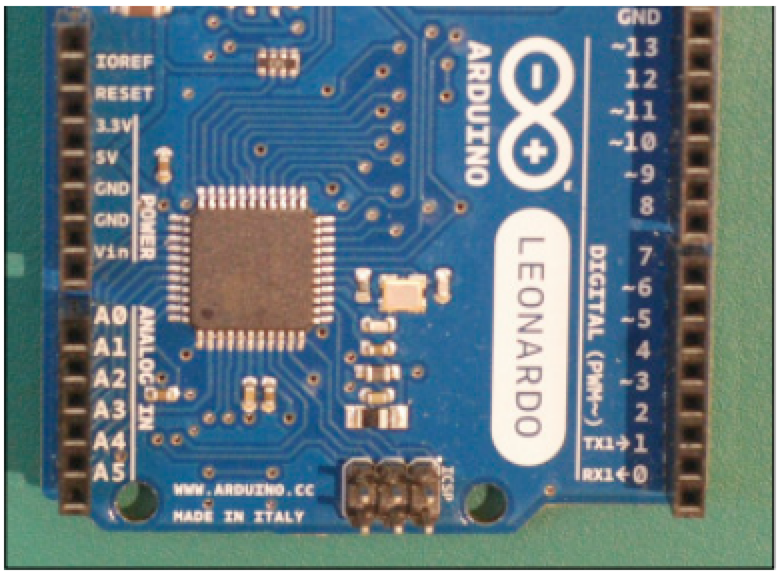
But code isn’t, in general, created perfect: in the meantime you need ways to identify where the errors are occurring so that you can bullet-proof them for next time. In the absence of a screen, the Arduino allows you to write information over the USB cable using Serial.write(). Although you can use the facility to communicate all kinds of data, debugging information can be particularly useful. The Arduino IDE provides a serial monitor which echoes the data that the Arduino has sent over the USB cable. This could include any textual information, such as logging information, comments, and details about the data that the Arduino is receiving and processing (to double-check that your calculations are doing the right thing).

SOME NOTES ON THE HARDWARE

The Arduino exposes a number of GPIO pins and is usually supplied with “headers” (plastic strips that sit on the pin holes, that provide a convenient

solderless connection for wires, especially with a “jumper” connection). The headers are optimised for prototyping and for being able to change the purpose of the Arduino easily

Each pin is clearly labelled on the controller board. The details of pins vary from the smaller boards such as the Nano, the classic form factor of the Uno, and the larger boards such as the Mega or the Due. In general, you have power outputs such as 5 volts or 3.3 volts (usually labelled 5V and 3V3, or perhaps just 3V), one or more electric ground connections (GND), numbered digital pins, and numbered analogue pins prefixed with an A.



You can power the Arduino using a USB connection from your computer. This capability is usually quite convenient during prototyping because you need the serial connection in any case to program the board. The Arduino also has a socket for an external power supply, which you might be more likely to use if you distribute the project. Either way should be capable of powering the microcontroller and the usual electronics that you might attach to it. (In the case of larger items, such as motors, you may have to attach external power and make that available selectively to the component using transistors.)

Outside of the standard boards, a number of them are focused on a particular niche application—for example, the Arduino Ethernet has an on-board Ethernet chip and trades the USB socket for an Ethernet one, making it

easier to hook up to the Internet. This is obviously a strong contender for a useful board for Internet of Things projects.

The LilyPad has an entirely different specialism, as it has a flattened form (shaped, as the name suggests, like a flower with the I/O capabilities exposed on its "petals") and is designed to make it easy to wire up with conductive thread, and so a boon for wearable technology projects.

Choosing one of the specialist boards isn't the only way to extend the capabilities of your Arduino. Most of the boards share the same layout of the assorted GPIO, ADC, and power pins, and you are able to piggyback an additional circuit board on top of the Arduino which can contain all manner of componentry to give the Arduino extra capabilities.

In the Arduino world, these add-on boards are called shields, perhaps because they cover the actual board as if protecting it.

Some shields provide networking capabilities—Ethernet, WiFi, or Zigbee wireless, for example. Motor shields make it simple to connect motors and servos; there are shields to hook up mobile phone LCD screens; others to provide capacitive sensing; others to play MP3 files or WAV files from an SD card; and all manner of other possibilities—so much so that an entire website, http://shieldlist.org/, is dedicated to comparing and documenting them.

In terms of functionality, a standard Arduino with an Ethernet shield is equivalent to an Arduino Ethernet. However, the latter is thinner (because it has all the components laid out on a single board) but loses the convenient USB connection. (You can still connect to it to push code or communicate over the serial connection by using a supplied adaptor.)

OPENNESS

The Arduino project is completely open hardware and an open hardware success story

The only part of the project protected is the Arduino trademark, so they can control the quality of any boards calling themselves an Arduino. In addition to the code being available to download freely, the circuit board schematics and even the EAGLE PCB design files are easily found on the Arduino website.

This culture of sharing has borne fruit in many derivative boards being produced by all manner of people. Some are merely minor variations on the main Arduino Uno, but many others introduce new features or form factors that the core Arduino team have overlooked.

In some cases, such as with the wireless-focused Arduino Fio board, what starts as a third-party board (it was originally the Funnel IO) is later adopted as an official Arduino-approved board.

Arduino Case Study:

The Good Night Lamp While at the IDII, Alexandra Deschamps-Sonsino came up with the idea of an Internet-connected table or bedside lamp. A simple, consumer device, this lamp would be paired with another lamp anywhere in the world, allowing it to switch the other lamp on and off, and vice versa. Because light is integrated into our daily routine, seeing when our loved ones turn, for example, their bedside lamp on or off gives us a calm and ambient view onto their lives.

This concept was ahead of its time in 2005, but the project has now been spun into its own company, the Good Night Lamp. The product consists of a "big lamp" which is paired with one or more "little lamps". The big lamp has its own switch and is designed to be used like a normal lamp. The little lamps, however, don't have switches but instead reflect the state of the big lamp.

Adrian was involved since the early stages as Chief Technology Officer. Adrian and the rest of the team's familiarity with Arduino led to it being an obvious choice as the prototyping platform. In addition, as the lamps are designed to be a consumer product rather than a technical product, and are targeted at a mass market, design, cost, and ease of use are also important. The Arduino platform is simple enough that it is possible to reduce costs and size substantially by choosing which components you need in the production version.

A key challenge in creating a mass-market connected device is finding a convenient way for consumers, some of whom are non-technical, to connect the device to the Internet. Even if the user has WiFi installed, entering authentication details for your home network on a device that has no keyboard or screen presents challenges. As well as looking into options for the best solution for this issue, the Good Night Lamp team are also building a version which connects over the mobile phone networks via GSM or 3G. This option fits in with the team's vision of connecting people via a "physical social network", even if they are not otherwise connected to the Internet.

Arduino Case Study: Botanicalls

Botanicalls (www.botanicalls.com/) is a collaboration between technologists and designers that consists of monitoring kits to place in plant pots. The Botanicalls kits then contact the owner if the plant's soil gets too dry. The project write-up humourously refers to this as "an effort to promote successful interspecies understanding" and as a way of translating between a plant's communication protocols (the colour and drooping of leaves) to human protocols, such as telephone, email, or Twitter.

The original project used stock Arduino controllers, although the kits available for sale today use the ATmega 168 microcontroller with a custom board, which remains Arduino-compatible, and the programming is all done using the Arduino IDE. To match the form factor of the leaf-shaped printed circuit board (PCB), the device uses a WizNet Ethernet chip instead of the larger Arduino Ethernet Shield. Future updates might well support WiFi instead.

Arduino Case Study: BakerTweet

The BakerTweet device (www.bakertweet.com/) is effectively a physical client for Twitter designed for use in a bakery. A baker may want to let customers know that a certain product has just come out of the ovens—fresh bread, hot muffins, cupcakes laden with icing—yet the environment he would want to tweet from contains hot ovens, flour dust, and sticky dough and batter, all of which would play havoc with the electronics, keyboard, and screen of a computer, tablet, or phone. Staff of design agency Poke in London wanted to know when their local bakery had just produced a fresh batch of their favourite bread and cake, so they designed a proof of concept to make it possible.

Because BakerTweet communicates using WiFi, bakeries, typically not built to accommodate Ethernet cables, can install it. BakerTweet exposes the functionality of Twitter in a "bakery-proof" box with more robust electronics than a general-purpose computer, and a simplified interface that can be used by fingers covered in flour and dough. It was designed with an Arduino, an Ethernet Shield, and a WiFi adapter. As well as the Arduino simply controlling a third-party service (Twitter), it is also hooked up to a custom service which allows the baker to configure the messages to be sent.