Microcontrollers fo

… Arduino for the enlightened

Clemens Valens (Elektor France Editorial)

Apparently Arduino is an Italian name – but when you search on the Internet, you mainly find dozens of references relating to electronics and programming. What's more, these references are often in relation to art. Electronics and art – now there's an interesting subject that's worth delving into! So just what exactly is Arduino?

At first sight, Arduino [1] is a small microcontroller board with a USB port (**Figure 1**) that comes in several models. There are even 'daisy'-shaped boards (Lilypads) intended for wearable applications, i.e. to be incorporated into garments. The Arduino board is programmed in a language very similar to C using Open Source tools available for Windows, Mac, and Linux. The hardware is also Open and anyone can make their own Arduino – the circuit diagrams and PCB photo masks are available free over the Internet. Arduinos are used a great deal by artists who need electronics in their creations.

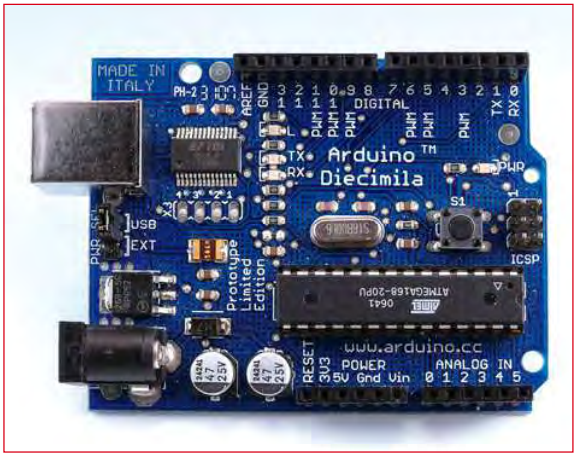


Figure 1. A Diecimila Arduino board. The new Duemilanove board is almost identical. These boards are cheap and easy to find.

When you look at it a bit more closely, an Arduino is not exactly a microcontroller board. In fact, an Arduino is quite simply an 8-bit microcontroller from Atmel – an ATmega8 for the earliest Arduinos and now more likely to be an ATmega168. This microcontroller is loaded with a 'bootloader' program that lets you load an application into the controller via a serial port, without overwriting the bootloader. Since modern computers no longer have serial ports, a USB port is often used. All this becomes an Arduino when you decide to dedicate certain pins of the controller to certain functions, since this allows the Arduino development environment to be used for writing and compiling application before loading them onto the controller.
The applications, called 'sketches', are written in a language that closely resembles C. Hardly surprising – it *is* C, but with some additional functions. All of the functions presented as the language for Arduino form a Hardware Abstraction Layer that lets you program the controller without needing to delve into the innards of the processor. The language has everything you need for most applications. Broadly speaking, there are functions for digital and analogue inputs/outputs, a few basic mathematical functions, time management functions (delays) and a few function for serial port communication – asynchronous (UART) and synchronous (SPI).

The digital I/O functions let you manipulate the logic levels of the pins, to read and write them. There is also a special function that makes it possible to measure the duration of a pulse. Using the analogue I/O functions, it is possible to read voltages and generate PWM signals. Lots of applications don't require anything more than this, and this is exactly where Arduino's strength lies. There's no need to go ferreting around in the registers and the controller data sheet to make a PWM output or a counter work — the 'dirty work' has already been done.

If these functions aren't enough, it's perfectly possible to program on a lower level and, just as in standard C, you can also add libraries with their own functions. But do watch out – if you go off into the darker depths of the Arduino programming language, you risk losing compatibility with the rest of Arduino community.

The Arduino community? Already, Arduino is a microcontroller, as well as a development environment and a programming language – now it's a community too? Yes! In fact, Arduino is more of a philosophy, the aim of which is to popularize technology to make it accessible to artists. Arduino is a logical sequel to Processing [2] and Wiring [3] projects. Processing is a multimedia programming language and Wiring is a development environment for artistic electronics But now we're starting to get away from our original

r Dummies ...

point; refer to the box about the origins of Arduino if you want to find out more.

Elektorino

Simple, free programming is something we're interested in. What's more, the electronics involved seem to be simple – so what could be more logical than to produce our own Arduino-compatible system? Well, that's just what we're going to do!

Our starting point is the basic Arduino Serial board. The office computer I use all the time still has a serial port, but for the unlucky owners of a computer that doesn't, we're going to use the USB-TTL cable [4]. In any event, we're going to be needing a TTL interface of some kind, as our own Arduino will only have a TTL serial port.

Our processor is going to be an ATmega168, which we'll be running at 16 MHz, to avoid getting caught out. For even though the controller is perfectly capable of operating at up to 20 MHz, the standard bootloader assumes a speed of 16 MHz. This can of course be modified if you are prepared to go delving into the bootloader – but for the moment we just want an Arduino board that works.

To finish off our Arduino, all we need do is add an LED, a reset push-button, a few resistors and capacitors, and two connectors: one for the serial port and the other for programming the bootloader. We need the latter to be able to program our Arduino for the first time, when the controller is still blank. Later, when the application is finished and the bootloader is no longer needed, this connector can be used for programming the controller directly from the application, which saves memory.

The LED has several functions. Given that the LED is present on several types of Arduino, lots of sketches use it. So does the bootloader, which flashes it at start-up.

Here's the circuit diagram of our finished Arduino (**Figure 2**). Thanks to the simplicity of the circuit, we can build it on prototyping board.

Unfortunately, we are not allowed to call our fine project Arduino – only boards approved by the Arduino community have the right to that name. This is why a second movement Freeduino [5] was created, which allows free use of the name Freeduino for home-made Arduino boards. But after all, it's only a name, so let's call ours Elektorino.

Implementation

Before you can load a sketch into the Elektorino, you need to load the bootloader. This is where things get more complicated, as there are two official Arduino bootloaders, the only apparent difference between them being the way the sketch is run after loading. This is achieved by way of a controller reset, which the Arduino environment can handle if the board has been equipped for it, and if it has the right

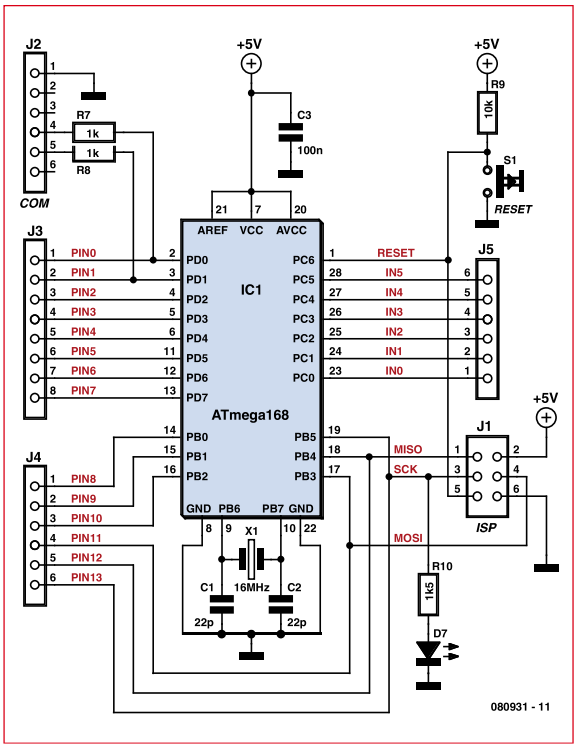


Figure 2. The circuit of the Elektorino. Not at all complicated. The connectors for the pins are not necessary, they are mainly used as a reference.

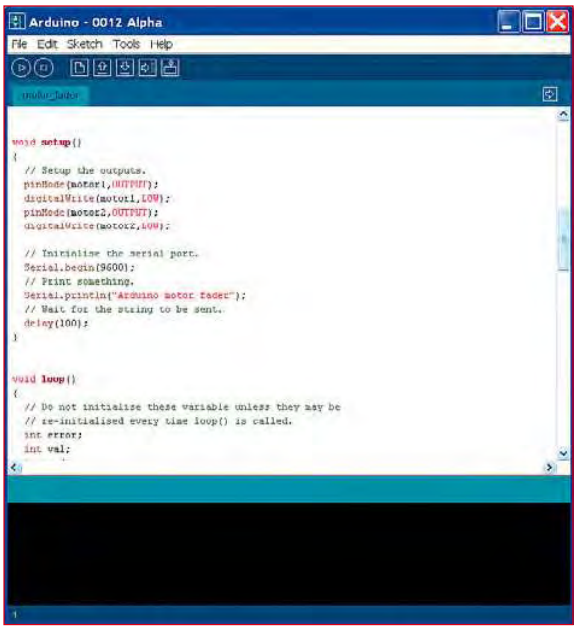


Figure 3.
The Arduino development environment.

bootloader. We haven't made any provision for this, and so we need the basic bootloader for the so-called NG (New Generation) boards.

But there is also another option: a third bootloader called ADABOOT [6]. This bootloader, an improved version of the official bootloaders, handles the reset and run delays differently. Initially, I worked for a bit with the NG bootloader, before replacing it with ADABOOT. Both worked perfectly well, but in the end I adopted ADABOOT, because it flashes the LED while the sketch is loading and because it is more convenient.
See the box to find out how to load the bootloader into the controller.

Hello world!

Once you have succeeded in loading the bootloader, it's time to see if Elektorino manages to communicate with the Arduino environment and if it is possible to load a sketch. So let's install Arduino. I've only done it under Windows XP, and that was extremely easy. All I had to do was download a large zipped file and unzip it somewhere onto the hard drive.

After running the Arduino environment (arduino.exe), you find yourself with a window like the one shown in **Figure 3**. Go into the *Tools* menu, then *Board*, and select the Arduino board being used. Of course, our one isn't listed, but an NG board using an ATmega168 will do.
You also have to select the serial port to be used for programming the microcontroller. Go into the *Tools* menu, then *Serial port* and select the correct port. If you want to use a USB serial port, check first that the drivers are present.
The Arduino environment comes with a small sketch, *Blink*, for checking that the board is working, and we can use this, since we have fitted the LED. The procedure is simple:

- Load the sketch; it's in *File* > *Sketchbook* > *Examples* > *Digital* > *Blink*.
- Compile the sketch by clicking the *Verify/Compile* button; this only takes a few seconds and (usually) ends with a success message.

- Load the sketch into Elektorino; first press the Reset button briefly, then click *Upload* to set the program loading. If all is well and if you are using ADABOOT, after a short delay you'll see the LED start to flash randomly — this is normal, it shows the transfer is taking place. After around five seconds (depending on the size of the sketch), the program is loaded and the controller is rebooted (the exact way in which the program is run depends on the bootloader). If the LED now flashes at a frequency of 1 Hz, everything has gone alright. Elektorino is working! If nothing happens, try resetting Elektorino.

Loading the bootloader...

...is not as hard as all that, as long as you have all the information you need. To save you hours on the Net, we've summed it up for you here in a few lines.

First of all, you need a programmer. There are several possibilities, for example, the one published in the 2008 double issue [7], or another 'SK200 compatible' programmer, easy to build using the circuit available on the PonyProg website [8]. On the Arduino website [1] yet another parallel port programmer is mentioned which is very simple and can be used directly from the Arduino environment. I tried it out, and managed to scramble one controller with it... so I went back to an SK200 one I already had.

Next, choose your bootloader. I recommend ADABOOT [6], but the NG version available on the Arduino website works perfectly well too.

Loading the bootloader into the controller can be done, for example, using AVRDUDE [9], supplied with the Arduino environment. AVRDUDE is a typical UNIX tool – it's basically a FreeBSD tool – with lots of incomprehensible options. Because it's very easy to make a mistake, here are the commands that work well (copy the bootloader into the directory that contains the AVRDUDE executable):

```
avrdude -p m168 -c pony-stk200 -V -e -U lock:w:0x3F:m -U hfuse:w:0xDF:m -U lfuse:w:0xFF:m -U efuse:w:0x0:m
avrdude -p m168 -c pony-stk200 -V -D -U flash:w:ATmegaBOOT_168_ng.hex
avrdude -p m168 -c pony-stk200 -V -U lock:w:0x0F:m
```

If you use another programmer, replace pony-stk200 with the appropriate value. Also check the name of your bootloader.

There are three commands that, broadly speaking, unlock the memory, load the program, set the fuses, and finally lock the memory. Refer to the AVRDUDE instructions if you want to know exactly what is going on (sensitive souls are advised to refrain!). Locking the memory is used to avoid overwriting the bootloader accidentally when loading a sketch.

A good website about the AVR is called Lady Ada [10].

A real application

It's all very well to have an Arduino development environment that works wonderfully well, but without a real application, it's not very interesting. I already had ten motorized slider pots, and it was high time to put them to good use. Why not with Elektorino? Elektorino has analogue inputs and PWM outputs — everything we need to drive a motor. So I'm going to suggest a driver for motorized faders. Note that this circuit can be used with any ATmega168-based Arduino board.

The fader in question (**Figure 4**) consists of a slider pot, a small motor, and an assembly of a few rollers, springs, and a piece of cord that enables the motor to move the slider in both directions. This assembly allows the motor to freewheel when the slider is unable to move – at each end of its travel, for example. Apart from the *10K B* marked on the fader, I didn't have any technical data on it, but a few experiments showed that the motor turned at a suitable speed when powered from 12 V. In this case, its consumption was around 200 mA.

The *B* marked on the fader might lead us to think it's a log model (as is often the case), but after checking, my faders turned out to be linear ones.

As a motor driver, I chose a cleverly-modified double H bridge with just two control lines and three states: anticlockwise, clockwise, and braking – just what we need (**Figure 5**). Usually, two controls allow four states, but in this circuit, states 00 and 11 are the same. A 5 V regulator has been slipped into the circuit so as to be able to power the whole controller assembly and motor from 12 V. The transistor are all NPN types, and those forming the bridge must be capable of carrying 200 mA happily. In my prototype, I used BD139s.

The potentiometer is wired as a simple potential divider. By measuring the voltage on the wiper, we know where it is (just so long as it's a linear pot).

The motor driver controls must be connected to digital outputs capable of supplying a PWM signal. An Arduino based around an ATmega168 has six, an ATmega8 only three. The pot wiper itself can be connected to any of the analogue inputs – in our case, in0.

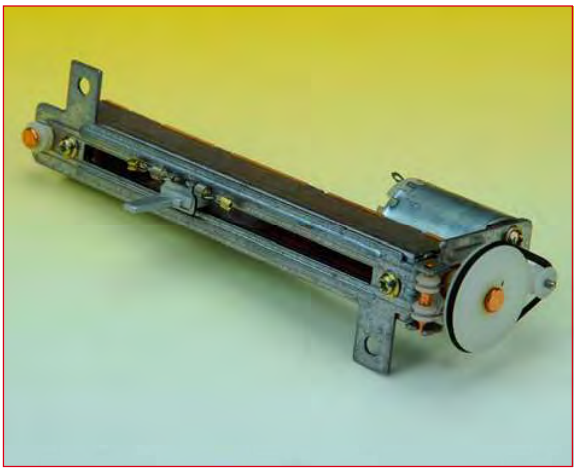


Figure 4.
A motorized fader of unknown make.

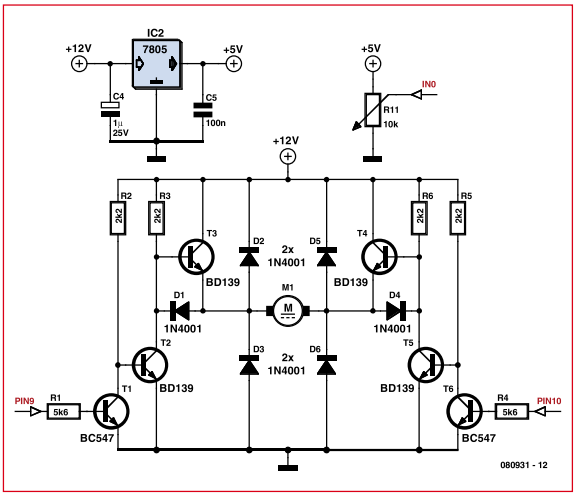


Figure 5.
The modified double H bridge and its three states. The labels refer to the pin designations, not the terminals on the controller.

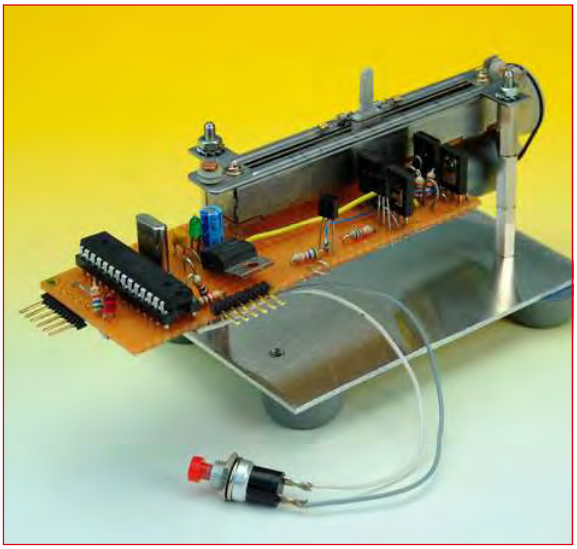


Figure 6.
The Elektorino prototype: the ATmega168 is on the left and the double H bridge to drive the motor on the right.

The sketch

Now that we have connected a motor driver to Elektorino (**Figure 6**), it's time to deal with the software. You'll see, the final sketch will be amazingly simple, thanks to the power of the Arduino.

A basic sketch consists of two functions: *setup()* and *loop()*, which are called by the layer of a lower level. In *setup*, called once at runtime, we put everything that relates to initializing the system – for example, the inputs/outputs and the serial port speed.

99.9% of embedded programs probably spend their whole lives in a loop. This is why in Arduino this loop is already implemented in the form of the *loop* function. This *loop* function is called periodically and may be regarded as Arduino's *main*. It's important to realize that, even though it looks like a special function, *loop* is just like any other function in C. So its local variables are reset each time it is called and variables that are required to keep their values between different occasions *loop* is called must be declared globally (or as *static*, for those familiar with C.)

The *setup* in our sketch doesn't contain anything very much. The Arduino pins are inputs by default, so only the two outputs need to be initialized. We're going to be using the serial port to drive our circuit, and for this it needs to be initialized. Thanks to the simplification offered by Arduino, all we have to do is enter the communication speed – in our case, 9,600 baud.

Processing, Wiring, and Arduino

Processing [2] is a language and an Open Source programming environment for programming images, animations, and interactions. The project, an initiative from Ben Fry and Casey Reas, is based on ideas developed by the Aesthetics and Computation Group of the MIT Media Lab. Processing was created in order to teach the fundamentals of programming in a visual context and to serve as a sketchbook or professional software production tool. Processing runs under GNU/Linux, Mac OS X, and Windows.

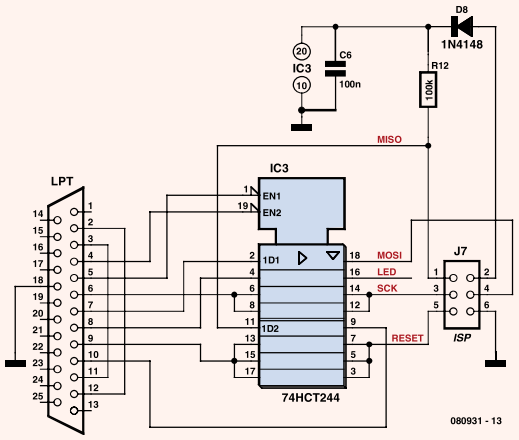
Several books have already been written on Processing.

Just like Arduino, **Wiring** [3] is a programming environment with microcontroller board for exploring electronic arts, teaching programming, and quick prototyping. Wiring, programmed in Processing, is an initiative by Hernando Barragán and was designed at the Interaction Design Institute Ivrea (IDII) in Italy.

Arduino [1] is a fast, Open Source electronic prototyping platform. Arduino is aimed at artists, stylists, enthusiasts, and anyone interested in creating objects or interactive environments. Created by Massimo Banzi, Gianluca Martino, David Cuartielles, and David Mellis, Arduino uses a programming language based on 'Processing'. Arduino may be regarded as a simplification of 'Wiring'.

Moving the pot wiper is done in *loop*. The principle is very simple: if the voltage measured on the input pin is different from the voltage required, the slider must be moved in the direction which will reduce this difference. In real life, it's a bit more complicated than that. To start with, there's the problem of direction, but more significant still is the problem of inertia. Once the slider is moving, it takes a little time for it to come to a complete halt. So it's easy to overshoot the required position if braking occurs too late. In the event of an overshoot, the slider has to be brought back, with the same risk of overshooting again, and so on. The system may even begin to oscillate.

The STK200 compatible programmer used by the author. There are also simpler programmers – it's a matter of personal preference.



To avoid these problems, we have used a Proportional Differential (*P-D*) regulator. In this type of regulator, the system reaches its final value without overshoot by continuously adjusting the correction signal according to the difference remaining to be corrected. So at the start of an adjustment, when the error is greatest, a strong correction signal is applied. Then, once the difference starts to reduce, the correction signal reduces too and the system slows down.

The correction signal consists of two parts: a signal proportional to the error (*P*) and a signal proportional to the error reduction (*D*). With a properly adjusted system of this sort, the slider can be moved quickly without overshooting the target value.

In the sketch (**Listing 1**) we can see the *P-D* regulator in the *loop* function. First, we measure the voltage at the wiper. The target value is subtracted from the measured value to obtain the error to be corrected. From this value, we calculate the two components *P* and *D* of the correction signal. The *P* component is the error multiplied by the constant K_p; the *D* component is obtained by multiplying the difference between the current value and the previous error by the constant K_d. The values for these constants were determined by experimentation, and you can modify them to see how the affect the adjustment. It's highly instructive.

The two components P and *D* are combined and the result is adapted to the range of usable values. The pot slider doesn't move for values below 50, and the maximum value for the PWM signal is 255.

Then we look to see if the error is small enough for us to be able to stop the motor. This comparison has to be performed for both slider directions. We leave a small margin for error, since perfection is perhaps a little over-ambitious.

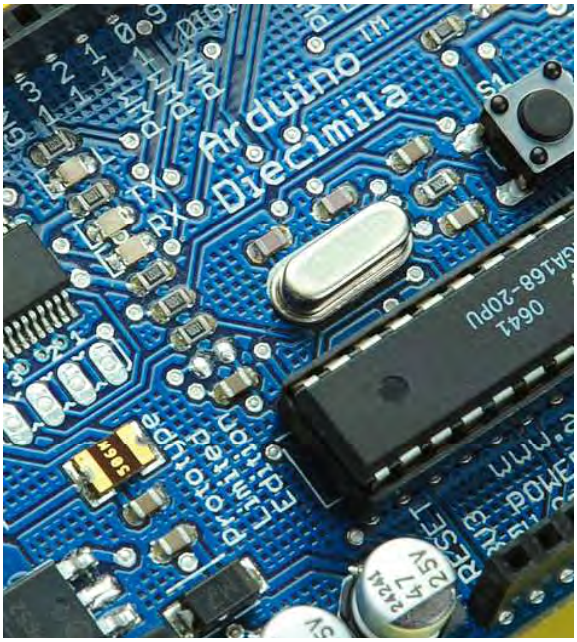
When the error is small enough, we prevent further corrections so as to free up the slider; we make the assumption that the system is never going to overshoot the target value.

AVR ISP via USB

If you use the FT232R chip as a USB interface and if you have access to all its pins, it is possible to use this chip to load the bootloader into the microcontroller without even needing a special ISP programmer! The FT232R chip has a bus called a CBUS with a bit function that lets you manipulate the associated pins individually. A certain Mr Suz from Japan has written a small piece of software that exploits this possibility and which can be downloaded free. Its avrdude-serjtag tool only works under Windows and its website is unfortunately in Japanese (suz-avr.sblo.jp/article/4438871.html). However, on his site one of his compatriots kindly explains in detail in English how to program an Arduino using this tool. See reference [11] for details.

In this way, it's possible to move the slider manually, without the system's trying to move it back into place. (Who's the strongest?)
Once the slider has been released, the system starts to output the slider position periodically (10 Hz) via the serial port. The serial port input is also scanned and as soon as four characters have been received, they are transferred as a target value for the slider and the PD regulator is re-activated to move it to its new position. No format checking is performed for the value received, the system requires an ASCII four-digit value between 0000 and 1023. To minimize errors, the target value obtained is limited between 3 and 1020, which minimizes problems of continuous activation at the ends of the travel.
The serial port is not used while the motor is operating, as this might produce interference, resulting in inaccurate positions or even oscillation. I've not taken the trouble to find out why: I'll leave that for you to do!

(080931-I)



References and Resources

[1] http://arduino.cc
[2] www.processing.org
[3] http://wiring.org.co
[4] www.elektor.fr/usb-ttl
[5] www.freeduino.org
[6] http://nearspacevermont.org/TheShoppe/freeduino/ADA-BOOT.shtml
[7] SimpleProg – ISP for AVR, Elektor, July/August 2008
[8] www.lancos.com/prog.html
[9] www.bsdhome.com/avrdude
[10] www.ladyada.net/learn/avr/index.html
[11] www.geocities.jp/arduino_diecimila/bootloader/index_en.html
Getting Started with Arduino, Banzi, Massimo, O'Reilly, 2008
Making Things Talk, Igoe, Tom, O'Reilly, 2007
The Duemilanove Arduino board is available from several sources including FunGizmos (US), LittleBird (Australia), SKPang (UK), Tinker (Italy), Make Magazine (Makershed.com).

Listing 1 – Easy-peasy!

```
void loop()
{
  int error;
  int val;
  int spd;
  float spd_p, spd_d;

  // read wiper voltage.
  val = analogRead(slider);

  // Calculate error.
  error = val - target;

  // Calculate proportional component P.
  // Two directions - so use absolute value.
  spd_p = abs(error)*Kp;

  // Calculate differential component D.
  spd_d = (last_error-error)*Kd;
  last_error = error;

  // Now mix P and D.
  spd = int(spd_p+spd_d);

  // Do not exceed limits.
  spd = constrain(spd,0,255);
  // Compensate friction.
  if (spd<50) spd += 50;

  if (error<-1 && stop==0)
  {
    // To maximum value ("left").
    digitalWrite(motor2,LOW);
    analogWrite(motor1,spd);
  }
  else if (error>1 && stop==0)
  {
    // To minimum value ("right").
    digitalWrite(motor1,LOW);
    analogWrite(motor2,spd);
  }
  else
  {
    // Shut down motor
    digitalWrite(motor1,LOW);
    digitalWrite(motor2,LOW);
    stop = 1;

    // Transmit cursor position.
    Serial.println(val);
    delay(100);

    // 4 characters form a new target value.
    if (Serial.available()>=4)
    {
      target = Serial.read()
   - '0'; // Thousand.
      target = Serial.read() - '0'
   + target*10; // Hundred.
      target = Serial.read() -
   '0' + target*10; // Ten.
      target = Serial.read() -
   '0' + target*10; // One.
      constrain(target,1,1022);
      // Start motor
      stop = 0;
    }
  }
}
```