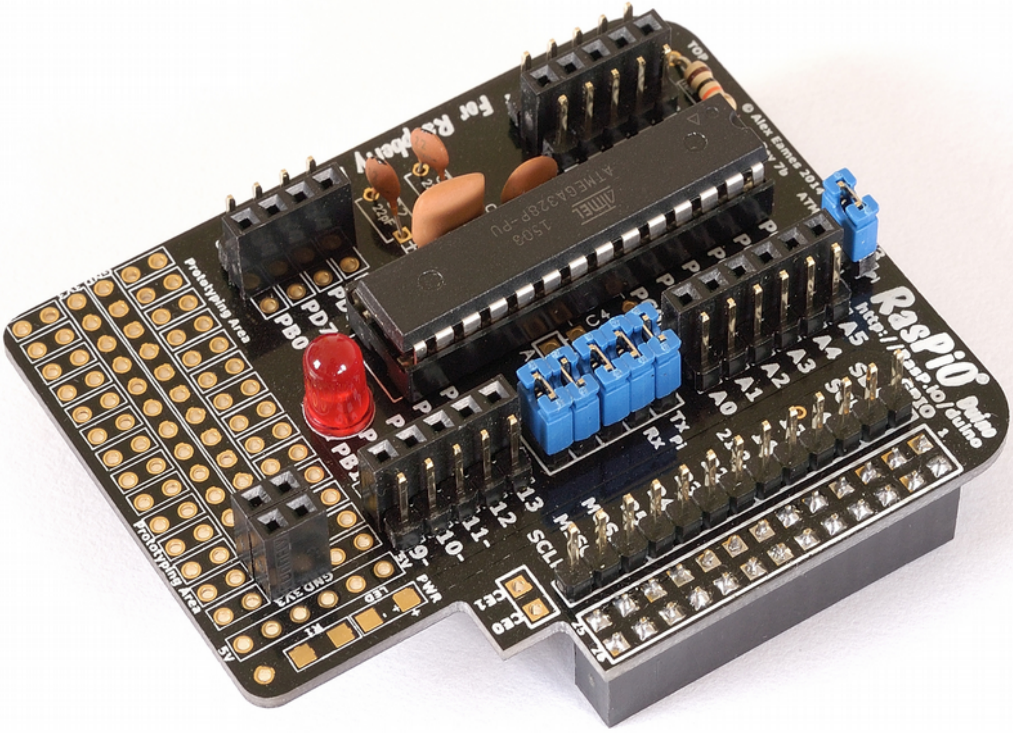
Learning Arduino Programming With



RasPiO® Duino

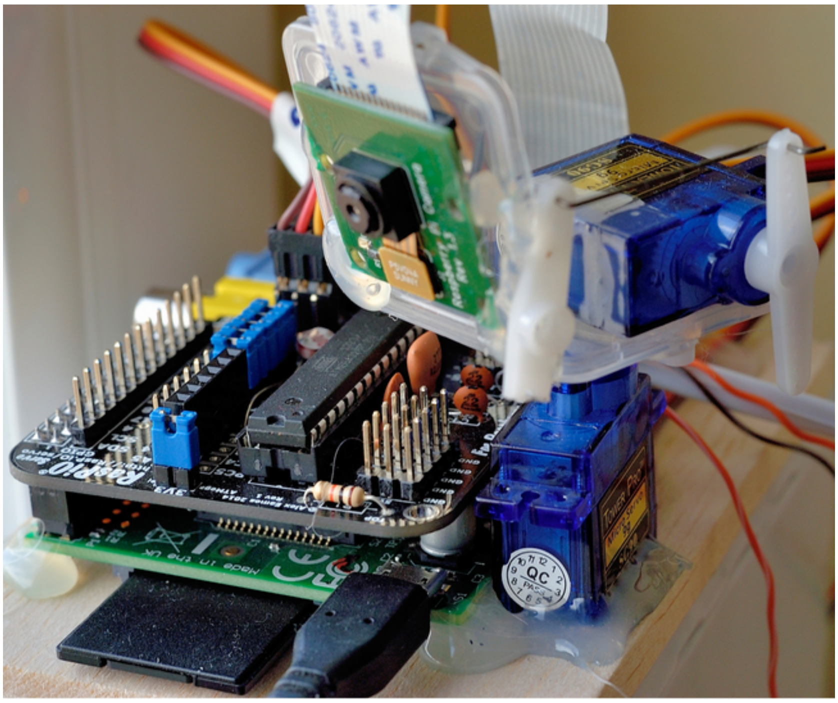
by Alex Eames

Introduction

The RasPiO®[1] Duino was developed out of the perceived need for a small, inexpensive Raspberry Pi® "Arduino-like" add-on board. There were already at least five other Arduino-like add-ons for the Pi out there, but they were either too expensive, too complicated, too big, or without adequate instructions.

With some, "RasPi.TV-style" instructions, I wanted to be able to help people get into Arduino programming. Also, by keeping the board inexpensive, I hope it's realistic for hardware hackers to be able to buy one for a specific project, solder things to it, program it, and let it get on with its job.

If you have any "unloved" older Raspberry Pis, you may find that you could give them a new lease of life as a Duino programmer, or use them in conjunction with a RasPiO Duino. I recently had great fun developing a twitter-controlled, pan-and-tilt, streaming, recording (with DropBox upload) tweeting, temperature and light reporting, security camera. This was based on a Raspberry Pi model B, and an earlier variant of the Duino board.

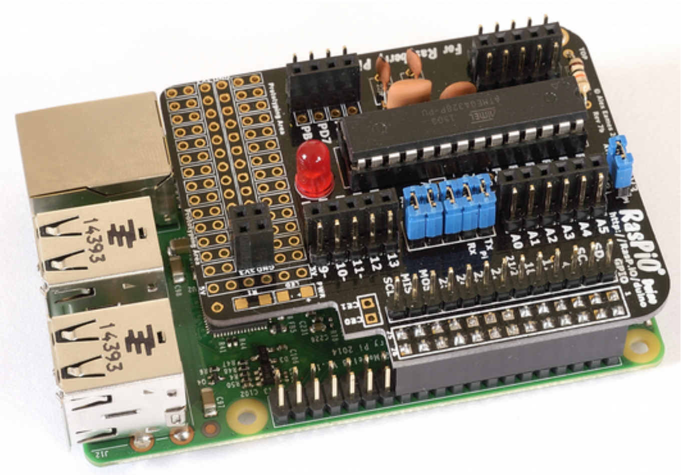


I hope you'll have as much fun with your RasPiO Duino(s) as I've had with this project.

[1] RasPiO is a trademark of Alex Eames. Raspberry Pi is a trademark of the Raspberry Pi Foundation

RasPiO Duino Instructions

Hardware Technical Overview



This page is mainly for the technically minded. If you just want to get on with building and programming your RasPiO Duino, you can skip to the next section. But do take notice of the bolded section about JP1.

The RasPiO Duino board runs on 3V3 at 12 MHz. For programming it uses the Gertboard version of the Arduino Integrated Development Environment (IDE) modified by Gordon Henderson.

Programming is via the SPI pins on the Pi, which are connected by 3 jumpers (labelled SCLK, MISO, MOSI). The reset pin on the ATMEGA328P-PU chip is hardwired to the Pi's CE0 pin and pulled up to 3v3.

JP1 at the top of the board connects the Pi's 3v3 rail with the ATMEGA. The board will not be powered unless you connect this jumper (or another power source) to the RasPiO Duino's 3v3 rail.

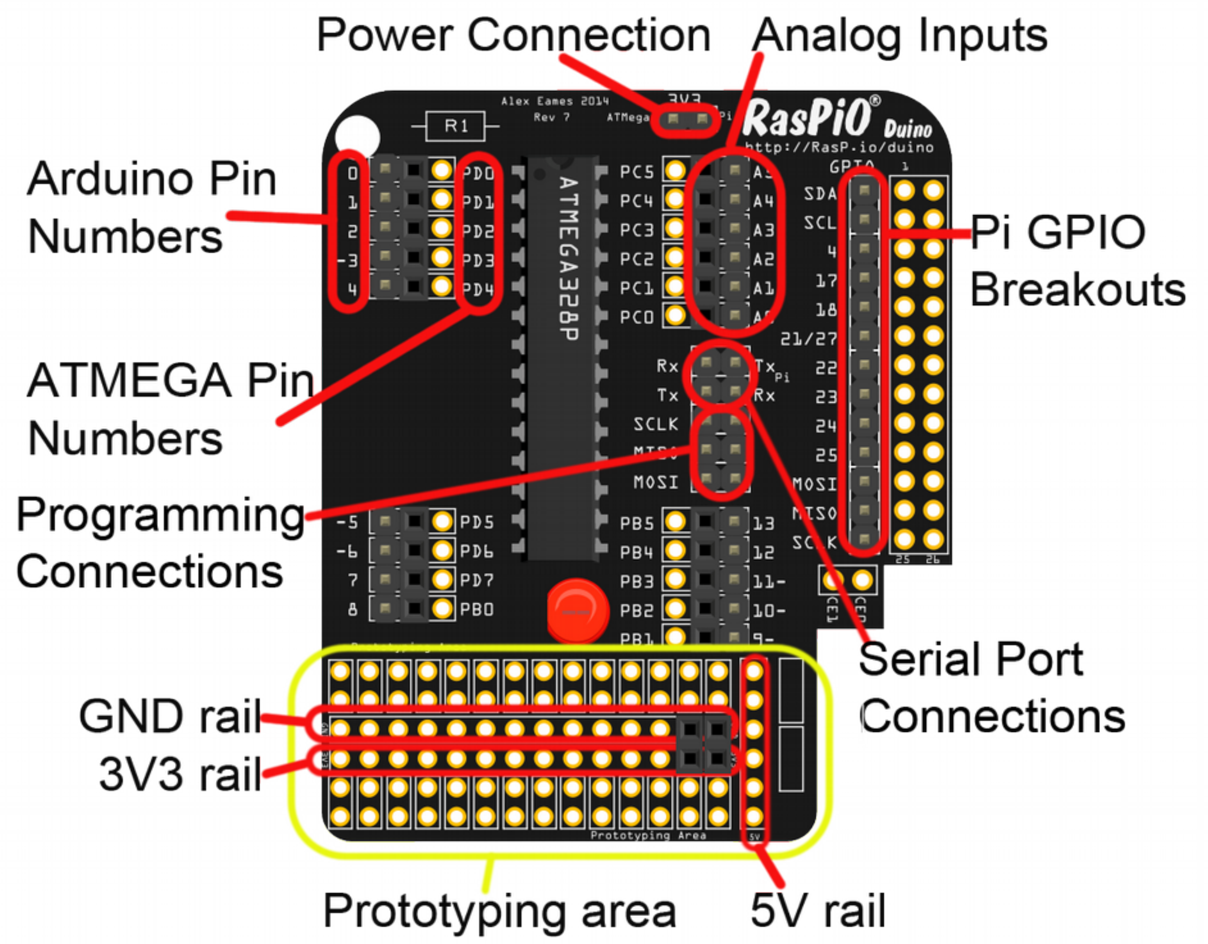
The external analog reference (A_{ref}) pin is hard-wired to 3v3 by default. If you need/want to change from the default (V_{cc}) analog reference to use either the internal 1V or an external reference, a track needs cutting on the back of the board to sever this 3v3 connection (**instructions for that are here**).

Everything else, hardware-wise, should be fairly straightforward. Just follow the build instructions or copy the photos. It's a through-hole only kit, although there are a few pads for optional SMT components (not supplied).

Know Your RasPiO Duino

The RasPiO Duino has been designed to fit any consumer model of Raspberry Pi and make it as easy as possible for people to get into Arduino programming on the Pi.

All of the useful ATMEGA pins are broken out three times, to male and female headers and a spare connection for user-determined purposes.



RasPiO Duino Board Layout

At the top there is a 2 pin header labelled 3V3. This connects the Pi's 3V3 rail to power the ATMEGA. It is also connected to the 3V3 rail in the prototyping area. If you don't want to connect your Duino to the Pi's 3V3 rail, you can leave this jumper off, but you will have to power it separately.

The pins are labelled both with their Arduino numbers (towards the outside of the board) and also the ATMEGA pin names.

Pins 0-13 are digital input/output pins. Those with a - (-3, -5, -6, 9-, 10-, 11-) are also able to be used as PWM pins.

The programming connections must be 'jumpered' to program the Duino. The serial connections need to be 'jumpered' to use serial communications between Duino and Pi.

Assembling the Board

There is a web page dedicated to the assembly of RasPiO Duino...

http://rasp.io/duino/duino-assembly/

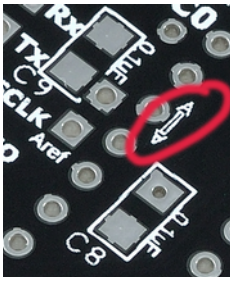
...on it you will find a comprehensive assembly video **(https://youtu.be/oiSV5wUjads)** as well as a full sequence of photographs, which can be clicked on to zoom in. But if you just want a text-based guide, you can find that below...

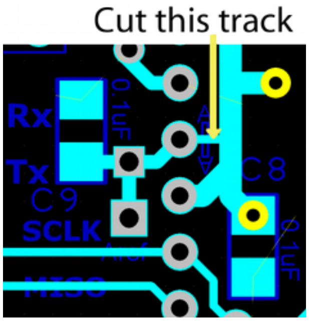
Verbal Assembly Instructions

1. Turn the board upside down and solder in the 330R resistor (Orange, Orange, Brown, Gold) in position R3. This is the only component underneath (in the standard configuration) apart from the 26-way header connecting the Pi. Once soldered, trim the protruding wires at both ends close to the PCB so as not to foul the ATMEGA and LED later.
2. With the board the right way up now, solder in the 10k resistor (Brown, Black, Orange, Gold) in position R1. (Either way round.)
3. Insert and solder the 28-way chip socket with the 'dimple' at the top end. It doesn't matter if you get it wrong, but it's there to help you get the chip the right way round.
4. Solder the 3 pin 12 MHz ceramic resonator. (Either way round.)
5. Solder a 100 nF capacitor (marked 104) into position C3. It doesn't matter which way round it goes. (A spare 100 nF is provided but not needed unless you are doing the Aref modification.)
6. Solder the two 22 pF capacitors (marked 22) in positions C1 & C2 (Either way round.)
7. Solder the 5x2 male header (which spans from Rx to MOSI) and the short single row male headers. There should be a 6, two 5s, a 4, and a 2 at JP1. JP1 is for a jumper to connect the Pi's 3V3 to the ATMEGA.
8. Solder the single female headers. There should be a 6, two 5s, a 4 and two 2s.
9. Solder LED1, paying attention to get it the right way round. It should line up with the silk-screen printing outline.
10. Solder the 26-way female header **ON THE UNDERSIDE OF THE PCB**.
11. Solder the 13-way single male header for the Pi GPIO breakout
12. Install the ATMEGA 328P-PU chip in the socket with its 'dimple' at the top. You will need to bend the pins slightly. The easiest way to do this without a special tool is to gently 'roll' the chip on a flat surface, so the pins of one side are slightly bent. Do it a little on each side until the pins line up nicely with the socket. Then press the chip into place.
13. Push jumpers onto JP1, SCLK, MISO & MOSI. JP1 provides power to the chip. The other three are needed for programming the ATMEGA. Tx and Rx are not needed unless you are using serial communication with the Pi.

Optional – if using A_{ref} Only

The A_{ref} pin is tied to 3V3 by default. If you want to use the chip's internal voltage reference or your own external source to set the A_{ref} voltage (most people won't need either) you can cut a track on the reverse of the board next to the A_{ref} pin. The place to cut is marked with <==> (cut inside the == signs). Then solder a pin or wire your voltage reference to the square pad marked A_{ref}. If you connect your own A_{ref}, install the capacitor C4, otherwise it's not needed.

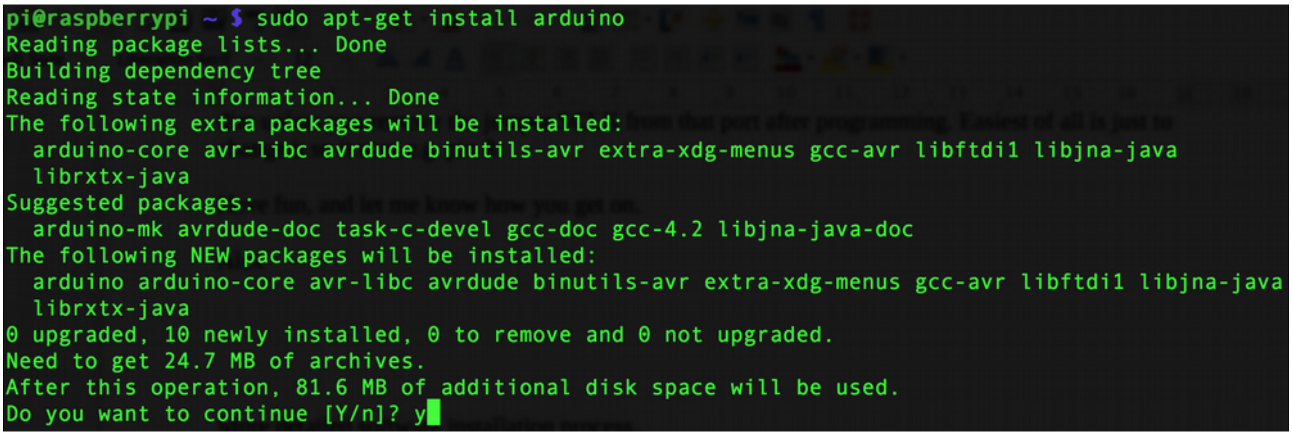




Software Installation

Before we can start using the RasPiO Duino, we need to install some software. First we'll install the Arduino IDE, then Gordon's modifications. From the command line...

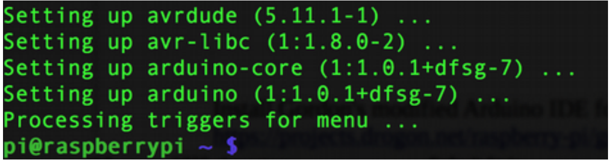
sudo apt-get update
sudo apt-get install arduino



```
pi@raspberrypi ~ $ sudo apt-get install arduino
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  arduino-core avr-libc avrdude binutils-avr extra-xdg-menus gcc-avr libftdi1 libjna-java
  librxtx-java
Suggested packages:
  arduino-mk avrdude-doc task-c-devel gcc-doc gcc-4.2 libjna-java-doc
The following NEW packages will be installed:
  arduino arduino-core avr-libc avrdude binutils-avr extra-xdg-menus gcc-avr libftdi1 libjna-java
  librxtx-java
0 upgraded, 10 newly installed, 0 to remove and 0 not upgraded.
Need to get 24.7 MB of archives.
After this operation, 81.6 MB of additional disk space will be used.
Do you want to continue [Y/n]? y
```

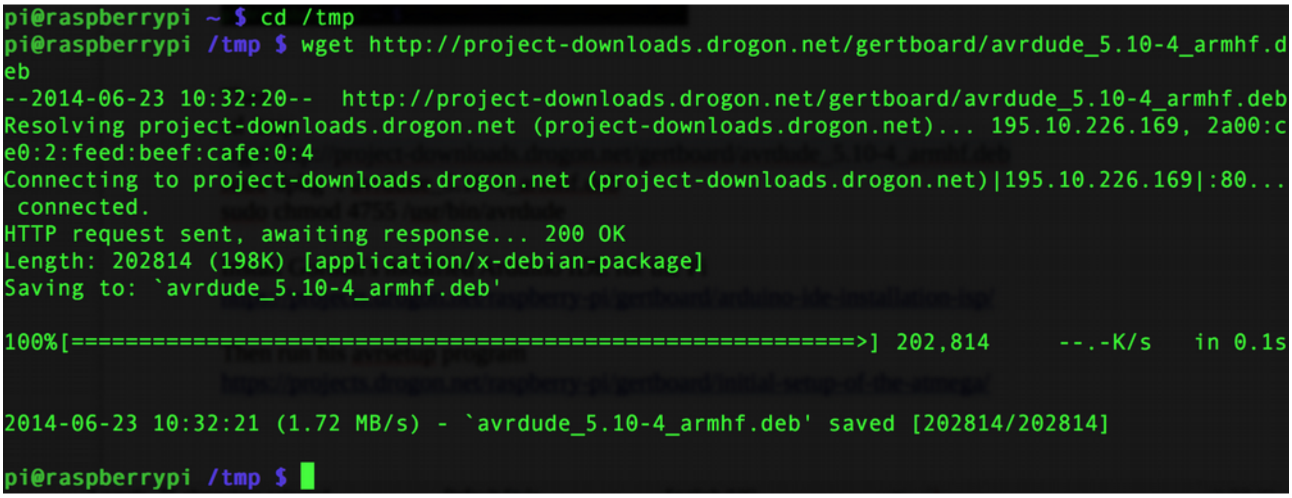
type y and press Enter

Then lots of things will install. It takes a while (several minutes) to download and install about 76 Mb of files. When finished, the last few messages should look like this...



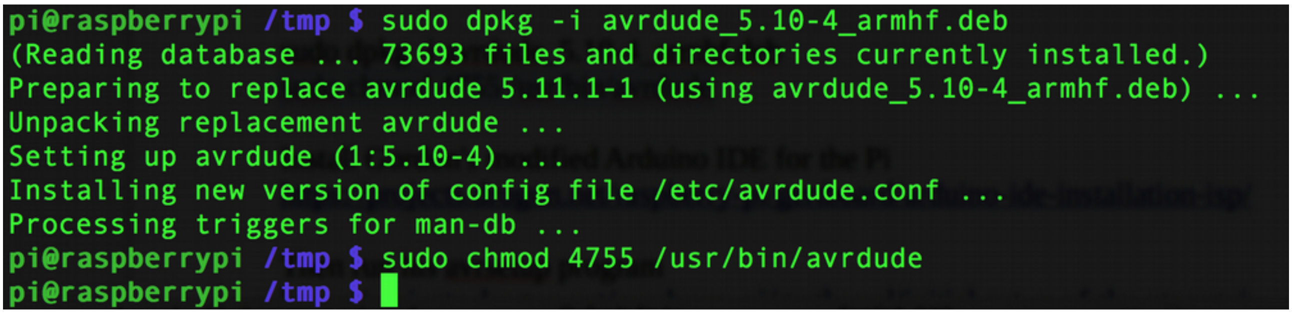
```
Setting up avrdude (5.11.1-1) ...
Setting up avr-libc (1:1.8.0-2) ...
Setting up arduino-core (1:1.0.1+dfsg-7) ...
Setting up arduino (1:1.0.1+dfsg-7) ...
Processing triggers for menu ...
pi@raspberrypi ~ $
```

Now we're ready to apply Gordon's modified avrdude...
cd /tmp
wget http://project-downloads.drogon.net/gertboard/avrdude_5.10-4_armhf.deb



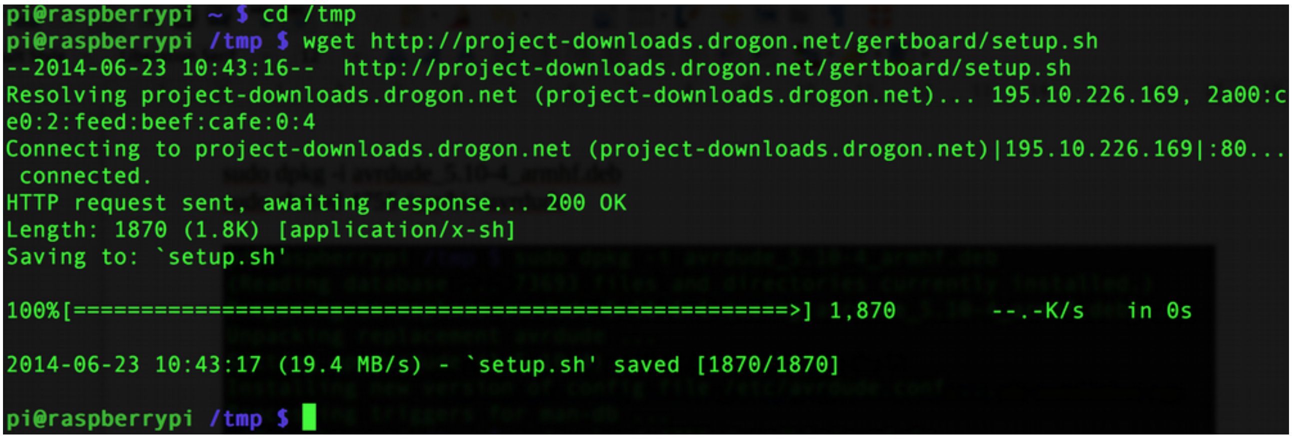
```
pi@raspberrypi ~ $ cd /tmp
pi@raspberrypi /tmp $ wget http://project-downloads.drogon.net/gertboard/avrdude_5.10-4_armhf.d
eb
--2014-06-23 10:32:20--  http://project-downloads.drogon.net/gertboard/avrdude_5.10-4_armhf.deb
Resolving project-downloads.drogon.net (project-downloads.drogon.net)... 195.10.226.169, 2a00:c
e0:2:feed:beef:cafe:0:4
Connecting to project-downloads.drogon.net (project-downloads.drogon.net)|195.10.226.169|:80...
 connected.
HTTP request sent, awaiting response... 200 OK
Length: 202814 (198K) [application/x-debian-package]
Saving to: `avrdude_5.10-4_armhf.deb'

100%[=========================================================>] 202,814     --.-K/s   in 0.1s

2014-06-23 10:32:21 (1.72 MB/s) - `avrdude_5.10-4_armhf.deb' saved [202814/202814]

pi@raspberrypi /tmp $
```

sudo dpkg -i avrdude_5.10-4_armhf.deb
sudo chmod 4755 /usr/bin/avrdude



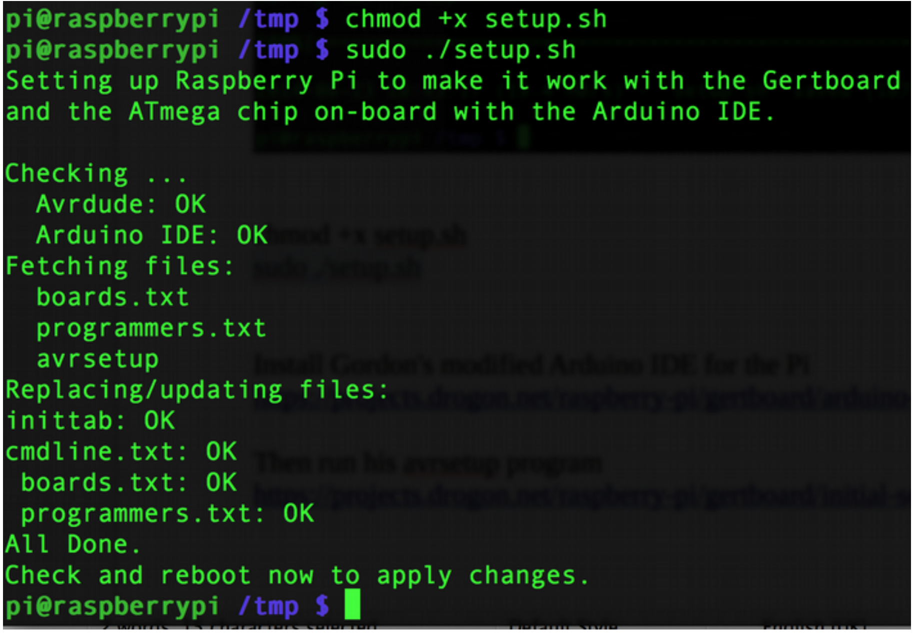
```
pi@raspberrypi /tmp $ sudo dpkg -i avrdude_5.10-4_armhf.deb
(Reading database ... 73693 files and directories currently installed.)
Preparing to replace avrdude 5.11.1-1 (using avrdude_5.10-4_armhf.deb) ...
Unpacking replacement avrdude ...
Setting up avrdude (1:5.10-4) ...
Installing new version of config file /etc/avrdude.conf ...
Processing triggers for man-db ...
pi@raspberrypi /tmp $ sudo chmod 4755 /usr/bin/avrdude
pi@raspberrypi /tmp $ 
```

And now we need to download and run a setup script...
cd /tmp
wget http://project-downloads.drogon.net/gertboard/setup.sh



```
pi@raspberrypi ~ $ cd /tmp
pi@raspberrypi /tmp $ wget http://project-downloads.drogon.net/gertboard/setup.sh
--2014-06-23 10:43:16--  http://project-downloads.drogon.net/gertboard/setup.sh
Resolving project-downloads.drogon.net (project-downloads.drogon.net)... 195.10.226.169, 2a00:c
e0:2:feed:beef:cafe:0:4
Connecting to project-downloads.drogon.net (project-downloads.drogon.net)|195.10.226.169|:80...
 connected.
HTTP request sent, awaiting response... 200 OK
Length: 1870 (1.8K) [application/x-sh]
Saving to: `setup.sh'

100%[======================================================>] 1,870       --.-K/s   in 0s

2014-06-23 10:43:17 (19.4 MB/s) - `setup.sh' saved [1870/1870]

pi@raspberrypi /tmp $ 
```

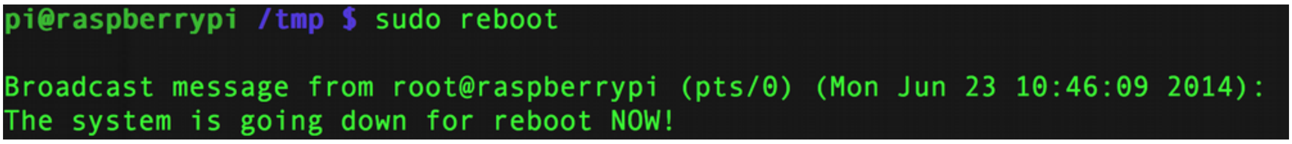
chmod +x setup.sh
sudo ./setup.sh



```
pi@raspberrypi /tmp $ chmod +x setup.sh
pi@raspberrypi /tmp $ sudo ./setup.sh
Setting up Raspberry Pi to make it work with the Gertboard
and the ATmega chip on-board with the Arduino IDE.

Checking ...
  Avrdude: OK
  Arduino IDE: OK
Fetching files:
  boards.txt
  programmers.txt
  avrsetup
Replacing/updating files:
inittab: OK
cmdline.txt: OK
 boards.txt: OK
 programmers.txt: OK
All Done.
Check and reboot now to apply changes.
pi@raspberrypi /tmp $ 
```

Then reboot...

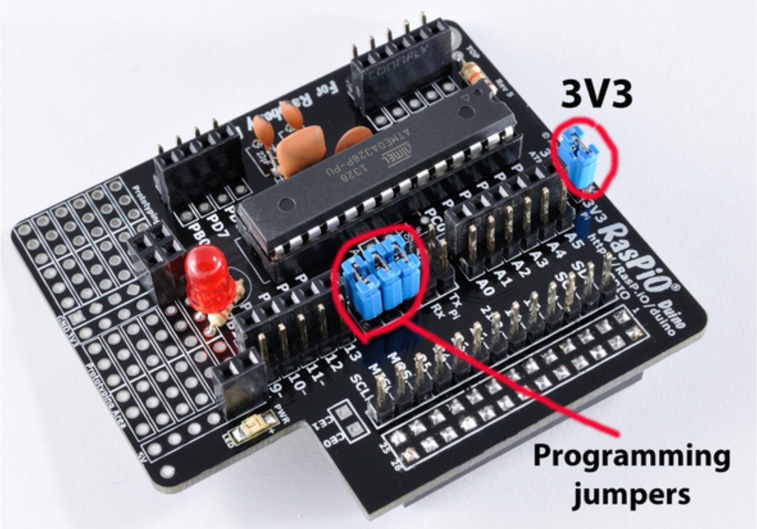
`sudo reboot`



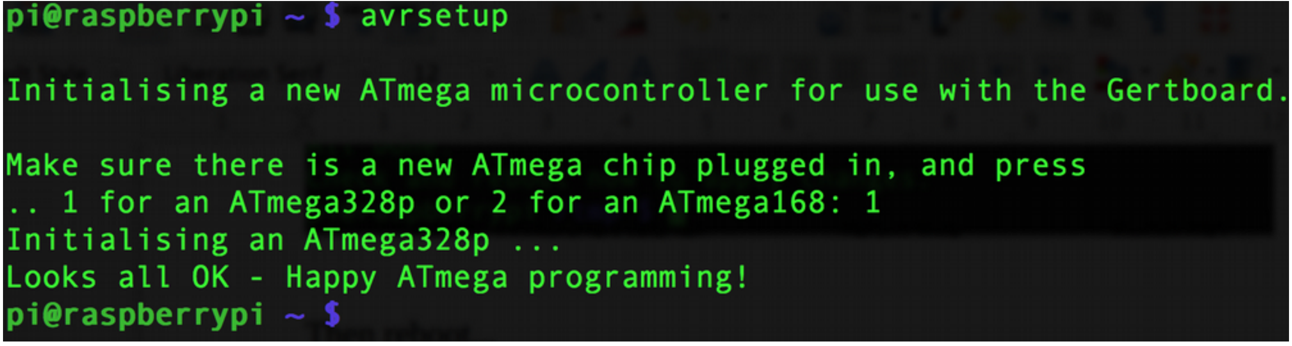
```
pi@raspberrypi /tmp $ sudo reboot

Broadcast message from root@raspberrypi (pts/0) (Mon Jun 23 10:46:09 2014):
The system is going down for reboot NOW!
```

Setting the Fuses (initialising your ATMega 328).

Now we're going to set the fuses on your ATMega328P-PU chip to configure the chip to work the way we want it to. Ensure that the lower three jumpers are in place SCLK, MISO, MOSI. Also the 3v3 jumper at JP1 should be in place (if you have not decided to power your duino separately).



Then type: `avrsetup`
then choose `1` for ATmega328p and press Enter. If it all works, this is what you should see...

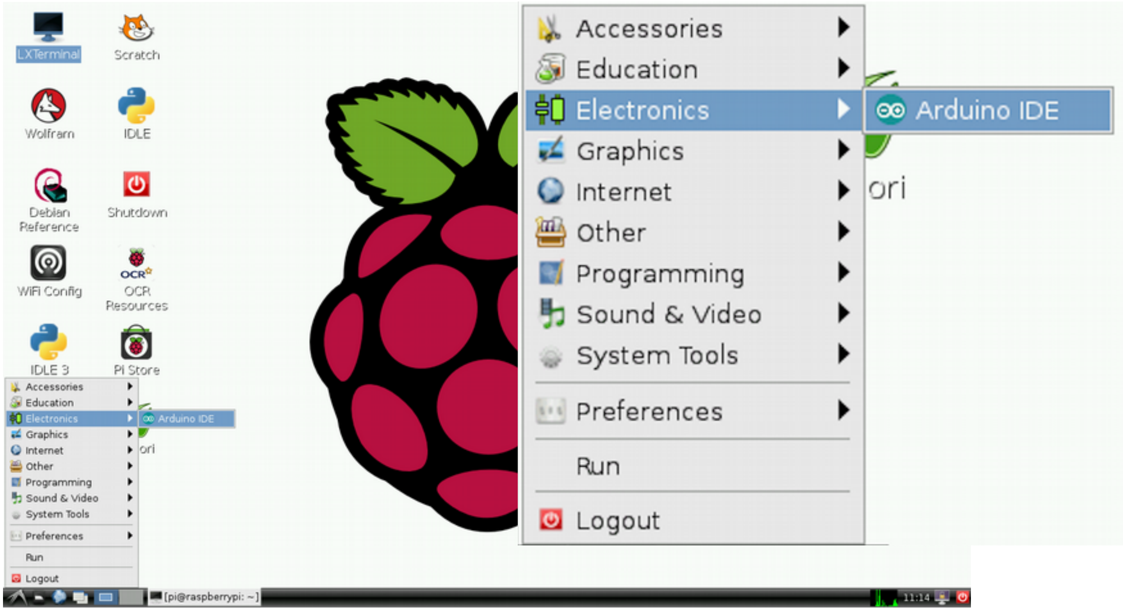


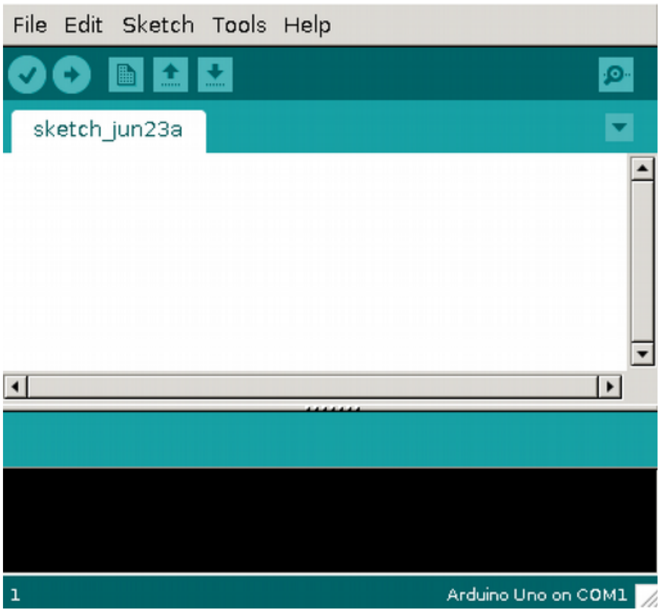
```
pi@raspberrypi ~ $ avrsetup

Initialising a new ATmega microcontroller for use with the Gertboard.

Make sure there is a new ATmega chip plugged in, and press
.. 1 for an ATmega328p or 2 for an ATmega168: 1
Initialising an ATmega328p ...
Looks all OK - Happy ATmega programming!
pi@raspberrypi ~ $
```

If, for some reason, it fails, try `sudo avrsetup`
You shouldn't really need *sudo* to run this, but one of the Beta testers found that it failed unless he used *sudo*. (It may have been something to do with his specific setup.)

Now go into the LXDE desktop environment
`startx`

then click on the icon and choose *Electronics > Arduino IDE*

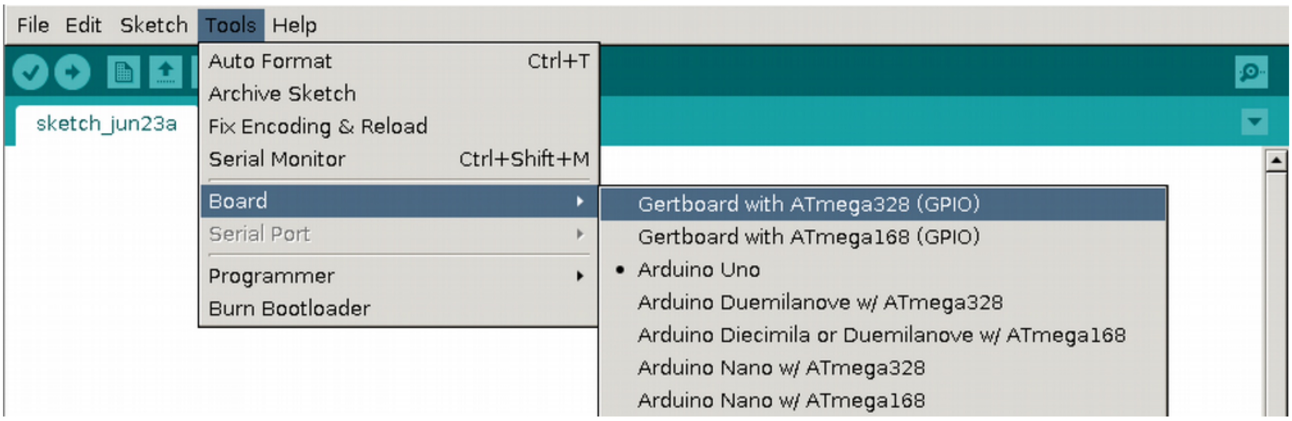




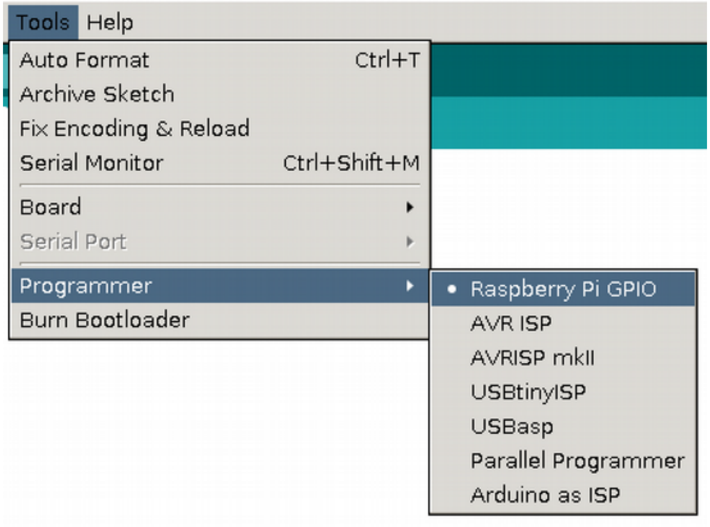
It takes a minute or so to start up. Then you should see this...

Now we need to ensure that the Arduino IDE is set to **Gertboard with ATMega 328 (GPIO)** because it defaults to Arduino Uno.

Tools > Board > Gertboard with ATMega 328 (GPIO)


File Edit Sketch Tools Help
sketch_jun23a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Programmer
Burn Bootloader
Gertboard with ATmega328 (GPIO)
Gertboard with ATmega168 (GPIO)
• Arduino Uno
Arduino Duemilanove w/ ATmega328
Arduino Diecimila or Duemilanove w/ ATmega168
Arduino Nano w/ ATmega328
Arduino Nano w/ ATmega168

Then we need to tell the IDE to use Raspberry Pi GPIO as the programmer... *Tools > Programmer > Raspberry Pi GPIO*



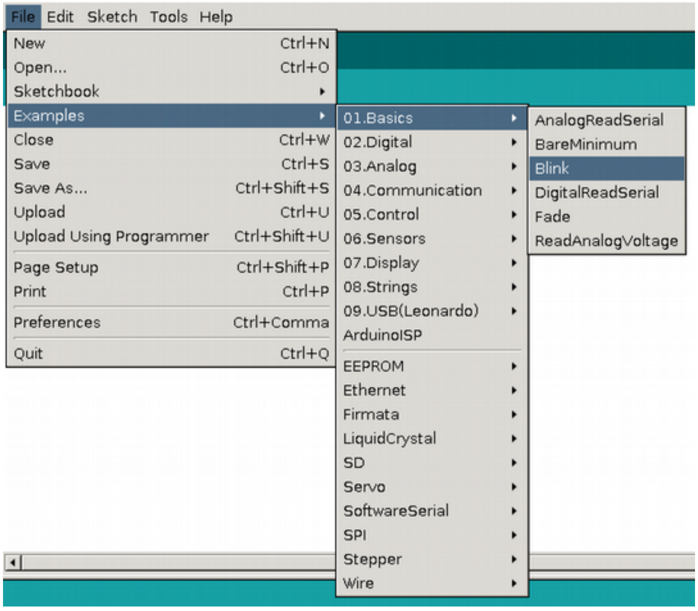
Now you're all set up and ready to try out some arduino sketches.

Walkthrough of the Blink Sketch.

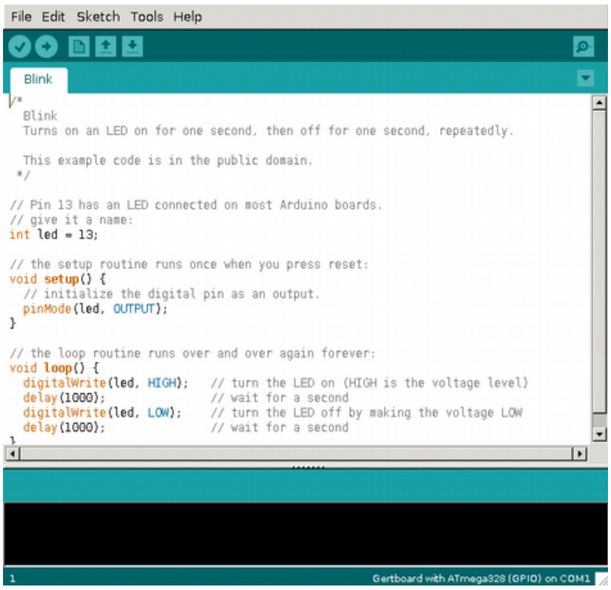
The blink sketch is the “hello world” of Arduino programming. It's usually the first thing people try out because there's an LED on pin 13 of most Arduino boards. This is useful because it flashes when the board is being programmed, so you know when that's happening. But it can also be used and controlled by your ATMega328 chip.

It's also built into the Arduino IDE. Here's how to find and upload it to your duino board...

File > Examples > Basics > Blink



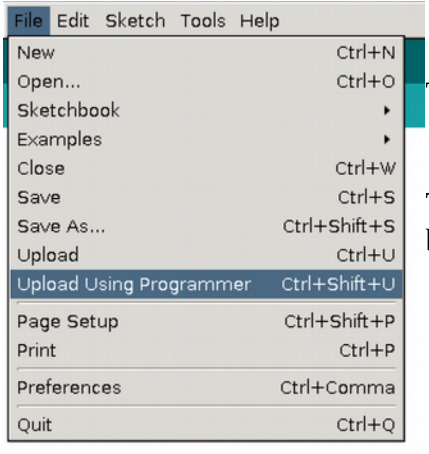
Then a new window containing the blink sketch should open..



```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// 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 (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

Then all you need to do is ensure that the SCLK, MISO & MOSI jumpers are in place, and...

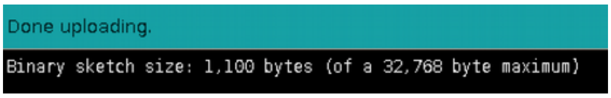
File > Upload Using Programmer (or CTRL + SHIFT + U)



Then the sketch will compile for about a minute...

Compiling sketch...

Then the LED should flash rapidly to show that the ATMega328 is being programmed. And you should see this...

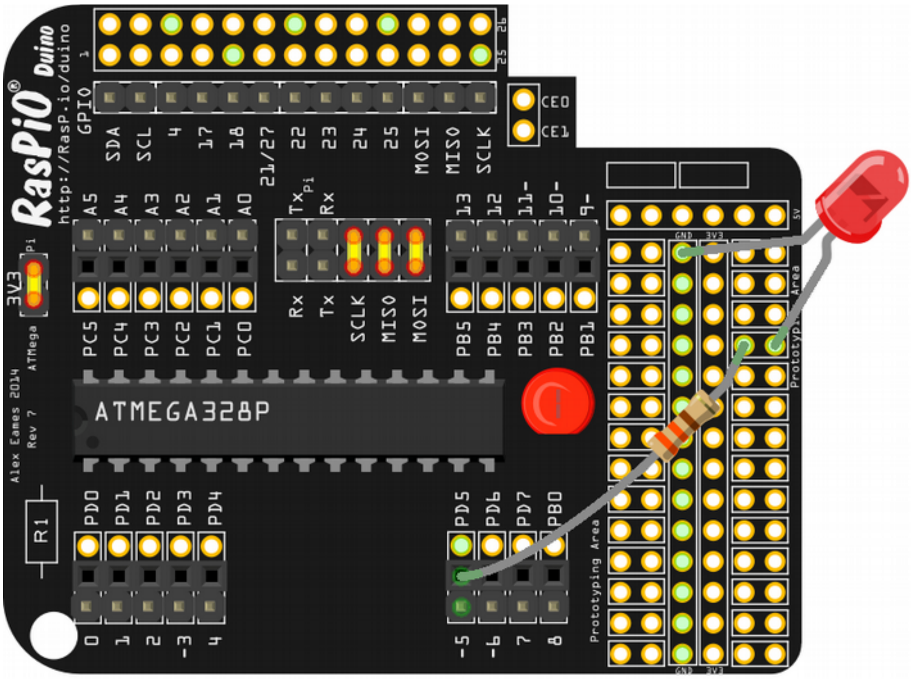


```
Done uploading.
Binary sketch size: 1,100 bytes (of a 32,768 byte maximum)
```

When that's finished, your LED should be flashing at one-second intervals. It may be a bit dim because of a small bug in the programming code (outside our control). If you remove the SCLK jumper it should be a bit brighter. But don't forget to put the jumper back in place before you try to reprogram the ATMega328 or you'll get error messages.

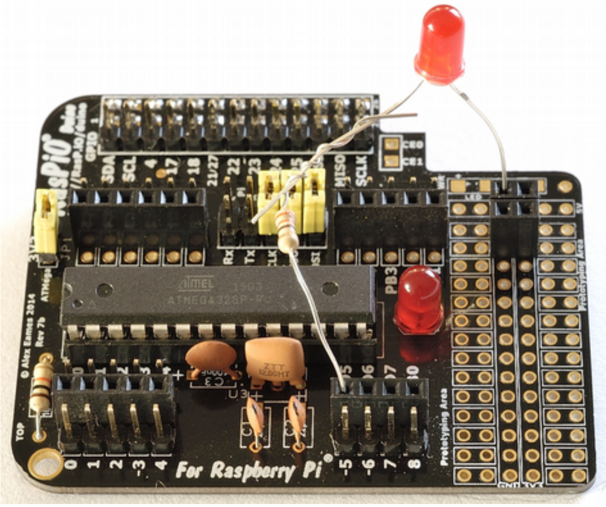
Now Use A Different Pin

You could easily side-step that issue by using pin 5 (`int led = 5;`) and adding an LED and 330 Ohm (Orange, Orange, Brown, Gold) resistor from pin 5 to LED to GND. In fact, we're going to do that next to test all 14 digital I/O ports and build towards our next experiment. Here's the circuit...

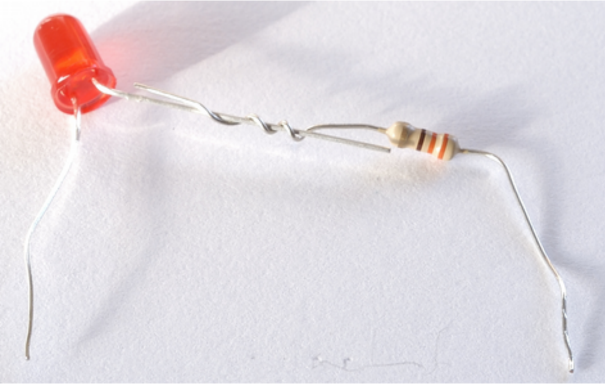


Adding a second LED and 330 Ohm resistor

In practice, you can easily twist the resistor wire round the LED's positive (long) leg, so you won't need to solder them to the board. The negative (short leg) end of the LED connects to GND (0V). It's recommended that you use the 2-way sockets for GND/3V3. If the connection is loose, you can bend the end of the LED wire a little bit so it makes a better connection. You can also do the same with the thin resistor wire at the other end.



LED long leg through 330R to pin 5. Short leg to GND.



Kinked ends of resistor & LED wires for better connection

So now we have an LED on digital pin 5, let's adjust the blink sketch to use that pin instead...

In line 10 of the blink sketch the output pin is defined...

```
int led = 13;
```

simply change `13` to `5` and then...

File > Save As

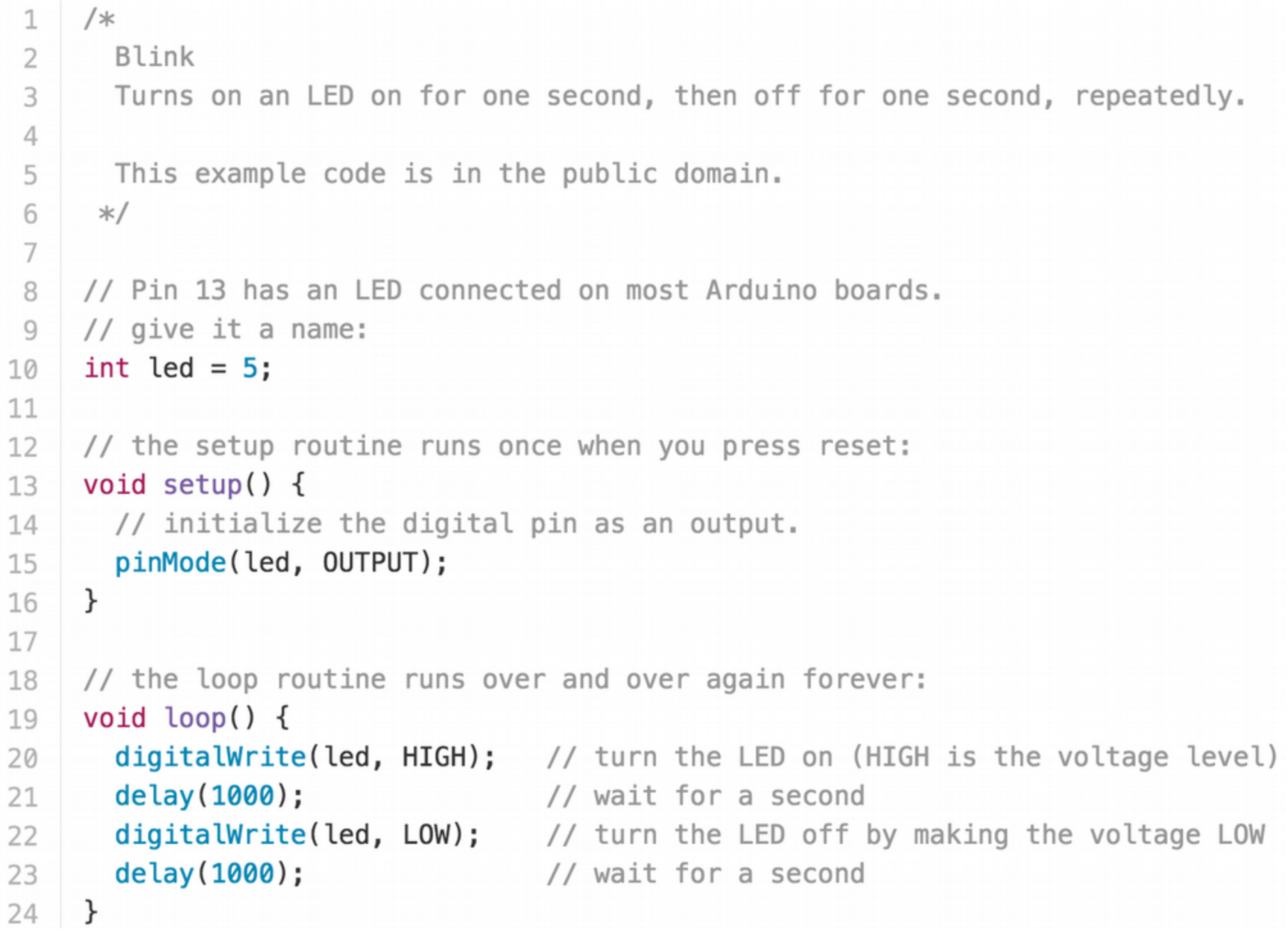
...give your file a name e.g. `blink2`. (It will create a sketchbook directory rather than overwriting the built-in blink sketch.) Then upload it to the RasPiO Duino with...

File > Upload Using Programmer (or CTRL + SHIFT + U)

Test All The Digital Pins

At this point, I'd recommend you test all the digital pins 0-13 on your Duino by setting the LED pin value to each one in turn, connecting the LED's resistor to that pin and then re-uploading the sketch. This will help you check that each port and header is soldered correctly. It will also give you some practice at tweaking and uploading sketches. Once you've done all that, you should now know how to 'flash' (upload) a sketch to your RasPiO Duino. You should also know if your digital input/output (i/o) ports are all working well.

Once you've finished testing all the ports, reattach your LED resistor to pin 5. Your sketch should now look like this (with only line 10 changed)...



```
 1  /*
 2    Blink
 3    Turns on an LED on for one second, then off for one second, repeatedly.
 4
 5    This example code is in the public domain.
 6   */
 7
 8  // Pin 13 has an LED connected on most Arduino boards.
 9  // give it a name:
10  int led = 5;
11
12  // the setup routine runs once when you press reset:
13  void setup() {
14    // initialize the digital pin as an output.
15    pinMode(led, OUTPUT);
16  }
17
18  // the loop routine runs over and over again forever:
19  void loop() {
20    digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
21    delay(1000);               // wait for a second
22    digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23    delay(1000);               // wait for a second
24  }
```

Code for blink2

Here's where you can find the code...

https://github.com/raspitv/raspio_duino/blob/master/blink2/blink2.ino

Blink Code Walk-through

Comments

Lines 1-9 of the blink sketch are comments. A comment is something written into the code for humans to read. It is completely ignored by the computer. In Arduino 'Wiring' language, which is a lot like C++, a multi-line comment starts with /* and ends with */. You can put any text you like in between and it makes no difference to the program. But it can be a good way to explain what you're doing.

You can also have inline comments, which you'll see start with //. For example, lines 8-9 and 20-23 of the Blink sketch all have inline comments.

Semi-colons

Something else you'll quickly notice is that most of the program lines end with a semi-colon ;

Ignoring comments, the only lines in our sketch which don't end with semi-colons are the ones with braces `{ and }`. Braces are used to denote a 'block' of code.

If you omit your semi-colons, your sketches will fail to compile[2] (ask me how I know).
Line 10 is the first actual line of functional code. 'Wiring' is quite a strict language. You need to declare what 'type' a variable is before you try to use it. We're using an integer (whole number) to tell the ATMEGA chip which port we want to use. So we need to declare it as an integer, using `int`...

```
int led = 5;
```

Further reading on variable types here...
http://www.arduino.cc/en/Reference/VariableDeclaration

Lines 13-16 are a program block (as denoted by the `{ }`) containing the setup() function. This function runs just once when the RasPiO Duino is powered up or reset. It's used to set up the pins/ports for how we want to use them. In our case, line 15...

```
pinMode(led, OUTPUT);
```

We already set our integer variable `led` to be 5, so this line will set up pin 5 to be an output pin. We're going to use it to switch our LED on and off.

So now we're all set up with the 'one-time' stuff, we'll go to the main program loop that will go round and round forever until we remove power, reset or reprogram the chip (or it crashes or breaks).

Lines 19-24 use braces `{ }` to denote a block of code...

```
void loop() {
   digitalWrite(led, HIGH);
   delay(1000);
   digitalWrite(led, LOW);
   delay(1000);
}
```

Within this block we are controlling our LED by 'writing' HIGH or LOW to our led pin, pin 5.
We're running the ATMEGA chip at 3.3V, so writing 'HIGH' sets the pin voltage at 3.3V (3V3).
Writing 'LOW' sets it at 0V (GND).

So in lines 20-21, we 'write' HIGH (connect 3.3V) to pin 5, switching on the LED. Then we wait 1000 ms (1 second) before doing anything else...

```
digitalWrite(led, HIGH);
delay(1000);
```

After that 1000 ms wait, we 'write' LOW (connect 0V) to pin 5, switching off the LED. Then we wait a further 1000 ms before the loop starts again and repeats lines 20-23...

```
digitalWrite(led, LOW);
delay(1000);
```

[2] Arduino sketches need to be compiled (converted into assembly language) before they can run. This is done for you by the Arduino IDE.

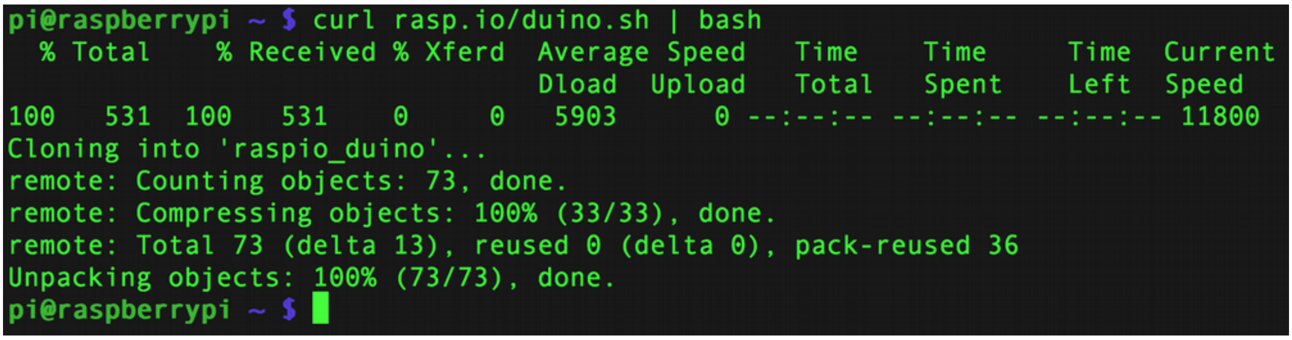
So as the sketch stands now, it will 'toggle' the LED on and off every second.

If you're curious about why the setup() and loop() both start with 'void' you can find that out here... **http://arduino.cc/en/Reference/Void**

Installing The Rest Of The Sketches

When going through the sketches in this booklet, links are provided to where you can copy and paste them into your Arduino IDE. If you prefer a quick and easy way to install all of the sketches right now, you can open an LXTerminal window (double-click LXTerminal icon) and run an install script with this command[3]...

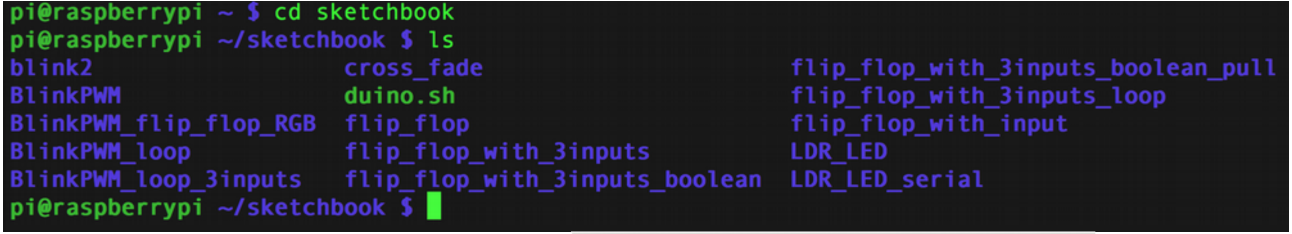
```
curl rasp.io/duino.sh | bash
```



```
pi@raspberrypi ~ $ curl rasp.io/duino.sh | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   531  100   531    0     0   5903      0 --:--:-- --:--:-- --:--:-- 11800
Cloning into 'raspio_duino'...
remote: Counting objects: 73, done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 73 (delta 13), reused 0 (delta 0), pack-reused 36
Unpacking objects: 100% (73/73), done.
pi@raspberrypi ~ $ 
```

This will clone the **raspio_duino Github repository** onto your Pi and then put the files in the right place for you to be able to use them from the Arduino IDE. It's better to do this after having set up the Arduino IDE, which is why we didn't do it before. If the Arduino IDE hasn't previously been run, our script will create a directory /home/pi/sketchbook. This is OK, but you will have to choose "sketchbook" as the directory to keep sketches in when you first start up the IDE. If the Arduino IDE has previously been run, the script will tell "mkdir: cannot create directory `sketchbook': File exists". That's fine. The rest of the script will still do its job.

If you then type cd sketchbook[4] you should see all the sketch directories...

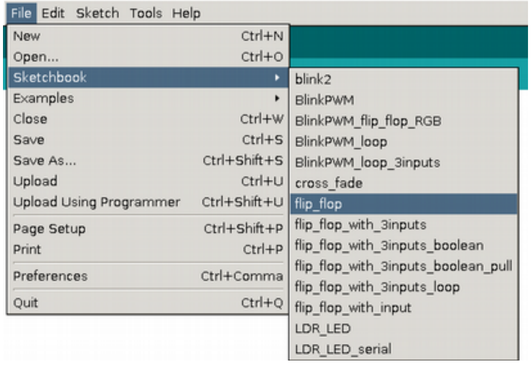


```
pi@raspberrypi ~ $ cd sketchbook
pi@raspberrypi ~/sketchbook $ ls
blink2                cross_fade                     flip_flop_with_3inputs_boolean_pull
BlinkPWM              duino.sh                       flip_flop_with_3inputs_loop
BlinkPWM_flip_flop_RGB flip_flop                     flip_flop_with_input
BlinkPWM_loop         flip_flop_with_3inputs         LDR_LED
BlinkPWM_loop_3inputs flip_flop_with_3inputs_boolean LDR_LED_serial
pi@raspberrypi ~/sketchbook $ 
```

To load any sketch, all you do is click

File > Sketchbook

...and then choose a sketch to load



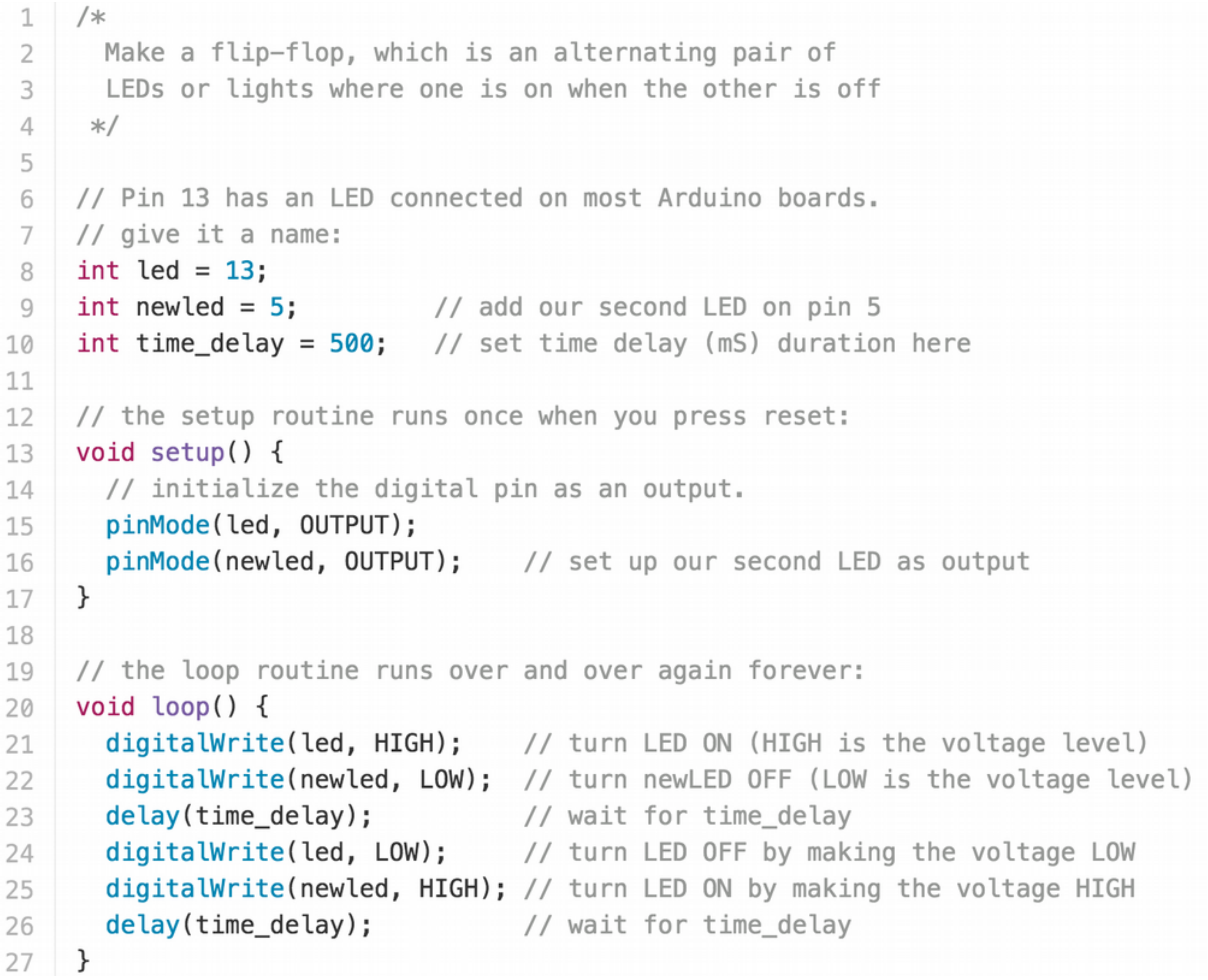
[3] If you want to check out this script before you run it, you can view it here http://rasp.io/duino.sh

[4] /home/pi/sketchbook if you're not in /home/pi already

Make a Flip-Flop

Now we're going to change the above sketch and hack it to 'make it our own'. We want to use both LEDs, so that one comes on when the other goes off and vice versa. We're going to add some new code and tweak the existing code a bit too...

First we've updated the comments (lines 1-4) at the start of the Blink sketch to show what we're now doing with our new sketch called flip_flop.



```
 1  /*
 2    Make a flip-flop, which is an alternating pair of
 3    LEDs or lights where one is on when the other is off
 4   */
 5
 6  // Pin 13 has an LED connected on most Arduino boards.
 7  // give it a name:
 8  int led = 13;
 9  int newled = 5;          // add our second LED on pin 5
10  int time_delay = 500;    // set time delay (mS) duration here
11
12  // the setup routine runs once when you press reset:
13  void setup() {
14    // initialize the digital pin as an output.
15    pinMode(led, OUTPUT);
16    pinMode(newled, OUTPUT);     // set up our second LED as output
17  }
18
19  // the loop routine runs over and over again forever:
20  void loop() {
21    digitalWrite(led, HIGH);     // turn LED ON (HIGH is the voltage level)
22    digitalWrite(newled, LOW);   // turn newLED OFF (LOW is the voltage level)
23    delay(time_delay);           // wait for time_delay
24    digitalWrite(led, LOW);      // turn LED OFF by making the voltage LOW
25    digitalWrite(newled, HIGH);  // turn LED ON by making the voltage HIGH
26    delay(time_delay);           // wait for time_delay
27  }
```

Code for flip_flop

Then in lines 8-10...

```
int led = 13;
int newled = 5;
int time_delay = 500;
```

...we change the value of `led` back to 13, and define two new integer variables...
`newled` for our new LED on pin 5 and
`time_delay` for our time delay, which is now set to 500 ms.

Storing the value of the time delay in a variable enables us to...

- have just one place where we set its value instead of two (lines 21 and 23 both set the delay value to 1000 in the Blink sketch)
- alter it within our program (we're going to do this later on)

We've also added a new line in our `setup()` block at line 16...

```
pinMode(newled, OUTPUT);
```

This sets up our new LED pin as an output. The variable name `newled` has been chosen to make it easy for us to see what it represents.

We've also made some changes in the main loop. Line 22 is new and line 23 has been changed to use our `time_delay` variable instead of the value 1000.

```
digitalWrite(newled, LOW);
delay(time_delay);
```

Notice that we're setting the new LED to LOW at the same time[5] as we're setting the original LED to HIGH. This is because we want one to be on when the other is off. Lines 21-23 switch `led` on and `newled` off, then waits for our `time_delay` period (whose value is set in line 10).

Lines 24-26 do the opposite. They switch `led` off and `newled` on, then wait for our `time_delay` period before the loop starts all over again. This will continue *ad infinitum*.

You can find the above code here...

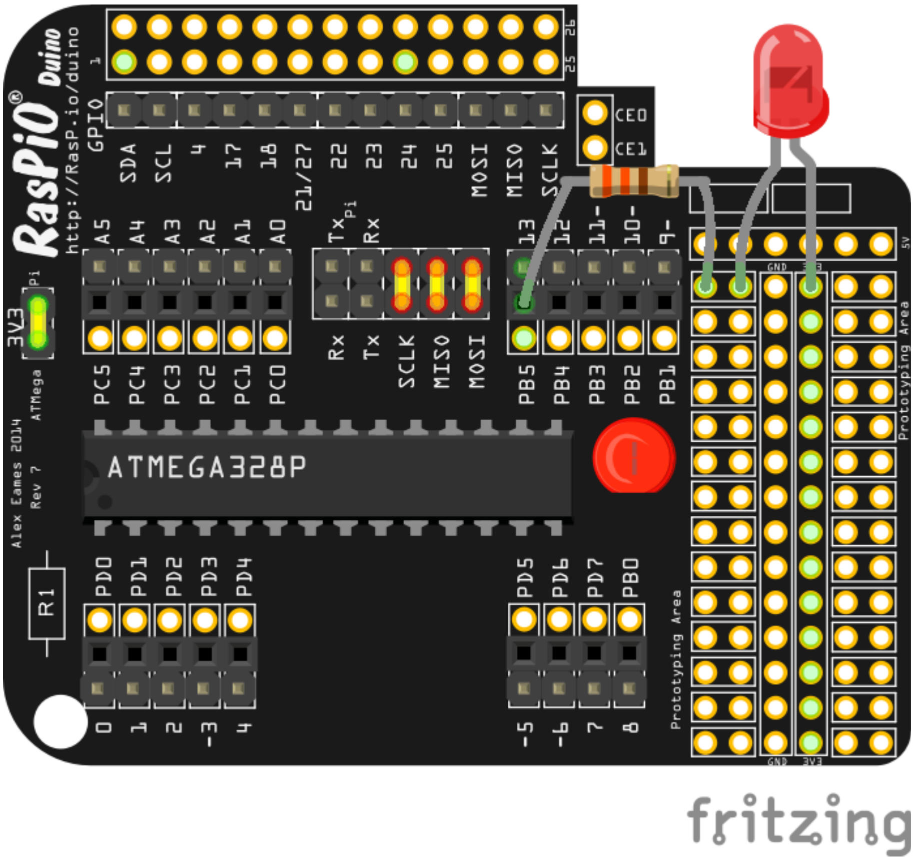
https://github.com/raspitv/raspio_duino/blob/master/flip_flop/flip_flop.ino

Now instead of changing the time delay in two places, if we want to speed up or slow down our flip-flop, we can just change the value of `time_delay` in line 10, or indeed elsewhere in the code. We'll make use of this a bit later on.

[5] Strictly speaking we're setting one just before the other, but it is so fast that you would need an oscilloscope to see any time difference. So it's effectively at the same time.

Single Pin Flip-flop

Because LEDs[6] are diodes, it is possible to make a flip-flop using just the unchanged Blink sketch with just one pin, if you connect a second LED 'backwards' and to 3V3 and both use pin 13. The following diagram shows the circuit...



Single port LED flip-flop circuit

Note that the short wire (-ve end) of the LED connects to pin 13 (via resistor) and the other end to 3V3. This means that when the pin is LOW/0V the extra LED will light. But when pin 13 is HIGH/3V3 the built-in LED will light.

It's a bit of a diversion from our progressive experiments, but I thought it was worth including, to show you the technique and help you 'think outside the box'. Now let's carry on building our flip-flop code and have a look at how to use inputs.

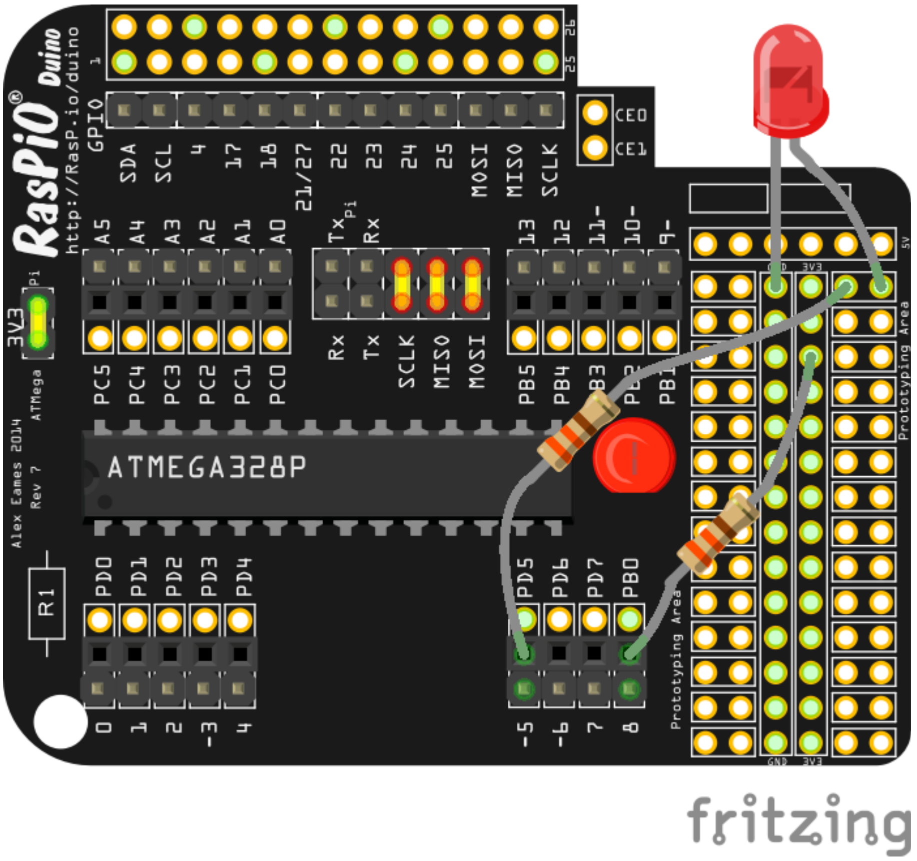
[6] LED stands for Light Emitting Diode. Current will only pass through a diode in one direction.

Flip-Flop Using Input For Frequency Control

Now we're going to modify our circuit and our code again slightly to demonstrate the use of inputs. Each pin 0-13 can be used as either an input or an output. When used as an output, the pins are set by our sketch to HIGH (3.3V, 1) or LOW (0V, 0).

When used as inputs, we can read whether the value of a pin is LOW (0) or HIGH (1). We can use this to make decisions in our program. So now we'll use one of our 330 Ohm resistors (Orange, Orange, Brown, Gold) to connect pin 8 to 3V3. If we set up pin 8 as an input, we can then read its value. When the other end of the resistor is connected to 3V3 the value will be HIGH (1). If we change it so that the other end of the resistor connects to GND (0V), the value we read will be LOW (0).

So we'll use this in our code to change the value of our `time_delay` variable, so that the flashing frequency will be faster when HIGH (1) and slower when LOW (0).



Flip-flop with input controlling frequency

So what have we changed from our basic flip-flop code to make this work? (**Full code on page 21**)

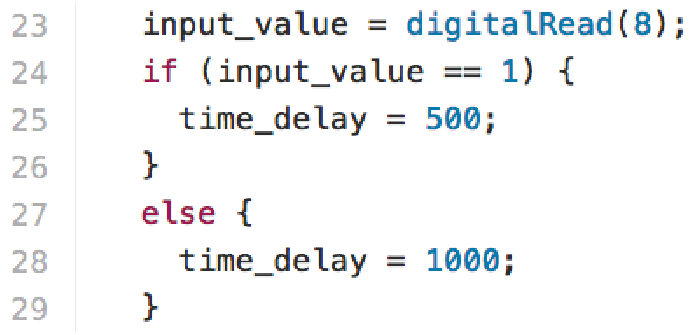
In line 11 we define an integer variable to store the value of our input, and set it to 1 initially...

```
int input_value = 1;
```

Next we set up pin 8 as an input in line 18...

```
pinMode(8, INPUT);
```

Then we need to add a few lines of new code (lines 23-29) into our main loop...



```
23     input_value = digitalRead(8);
24     if (input_value == 1) {
25       time_delay = 500;
26     }
27     else {
28       time_delay = 1000;
29     }
```

In line 23 we read the value of pin 8 using `digitalRead(8)` and assign it to the integer variable `input_value`

```
input_value = digitalRead(8);
```

In line 24 we are testing whether pin 8 is set HIGH or not (1 is HIGH, 0 is LOW).

```
if (input_value == 1) {
```

...runs the block of code between the braces {} if the condition inside the parentheses ( ) is met. Notice the use of a double equals sign ==. Double == is used for testing conditionals[7], single = is used for setting the value of variables. This is quite a common convention in several programming languages (including Python).

In our case, if our input pin 8 is at 3V3 (HIGH, 1) the time delay is set to 500 ms...

```
time_delay = 500;
```

...and then in line 26 we close this block of code with a }

But we also need to cater for the situation when our pin is at 0V (LOW, 0). The simplest way to do this is to use another block of code that is executed only if our conditional test `if (input_value == 1)` fails. So in line 27 we use...

```
else {
```

...to achieve this. The code in this `{}` block will only ever run if pin 8 is not 3V3/HIGH/1. In this case, we're setting the time delay to 1000 ms...

```
  time_delay = 1000;
}
```

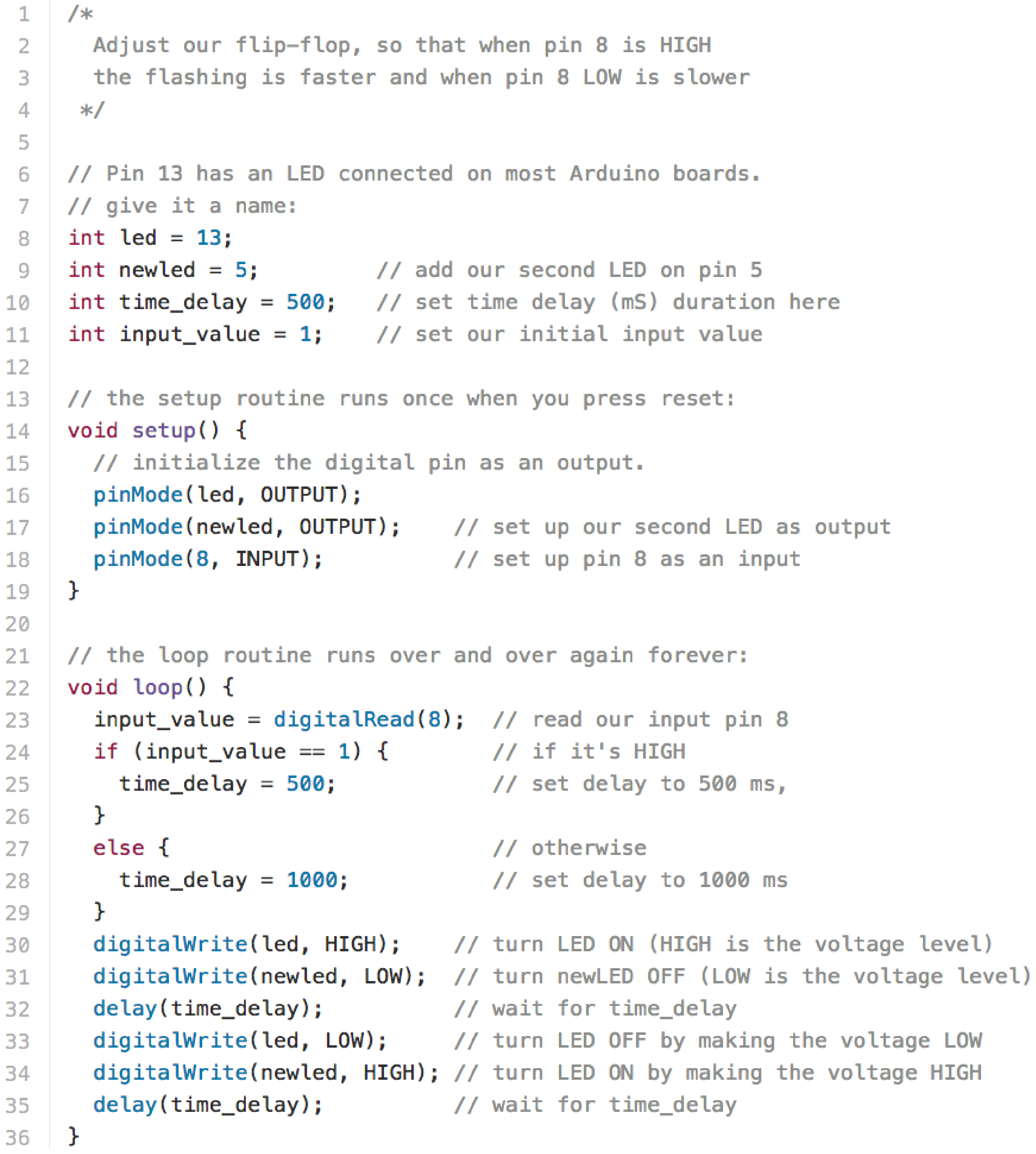
...and then closing the else block in line 29 with a }.

[7] A 'conditional' is a test to see if a certain condition is met e.g. *if x == 3*, or *if y > 0.* They are used to control the logical flow of a program

So lines 23-29 modify our basic flip-flop code to add a 'conditional' which can change the value of `time_delay` according to the status of pin 8.

What should now happen is, when pin 8 is connected to 3V3 (via 330 Ohm resistor) the LEDs should flash every 500 ms and when connected to 0V (GND) they will flash every 1000 ms.

Here's the full code...



```
 1  /*
 2    Adjust our flip-flop, so that when pin 8 is HIGH
 3    the flashing is faster and when pin 8 LOW is slower
 4   */
 5  
 6  // Pin 13 has an LED connected on most Arduino boards.
 7  // give it a name:
 8  int led = 13;
 9  int newled = 5;          // add our second LED on pin 5
10  int time_delay = 500;    // set time delay (mS) duration here
11  int input_value = 1;     // set our initial input value
12  
13  // the setup routine runs once when you press reset:
14  void setup() {
15    // initialize the digital pin as an output.
16    pinMode(led, OUTPUT);
17    pinMode(newled, OUTPUT);     // set up our second LED as output
18    pinMode(8, INPUT);           // set up pin 8 as an input
19  }
20  
21  // the loop routine runs over and over again forever:
22  void loop() {
23    input_value = digitalRead(8);  // read our input pin 8
24    if (input_value == 1) {        // if it's HIGH
25      time_delay = 500;            // set delay to 500 ms,
26    }
27    else {                         // otherwise
28      time_delay = 1000;           // set delay to 1000 ms
29    }
30    digitalWrite(led, HIGH);     // turn LED ON (HIGH is the voltage level)
31    digitalWrite(newled, LOW);   // turn newLED OFF (LOW is the voltage level)
32    delay(time_delay);           // wait for time_delay
33    digitalWrite(led, LOW);      // turn LED OFF by making the voltage LOW
34    digitalWrite(newled, HIGH); // turn LED ON by making the voltage HIGH
35    delay(time_delay);           // wait for time_delay
36  }
```

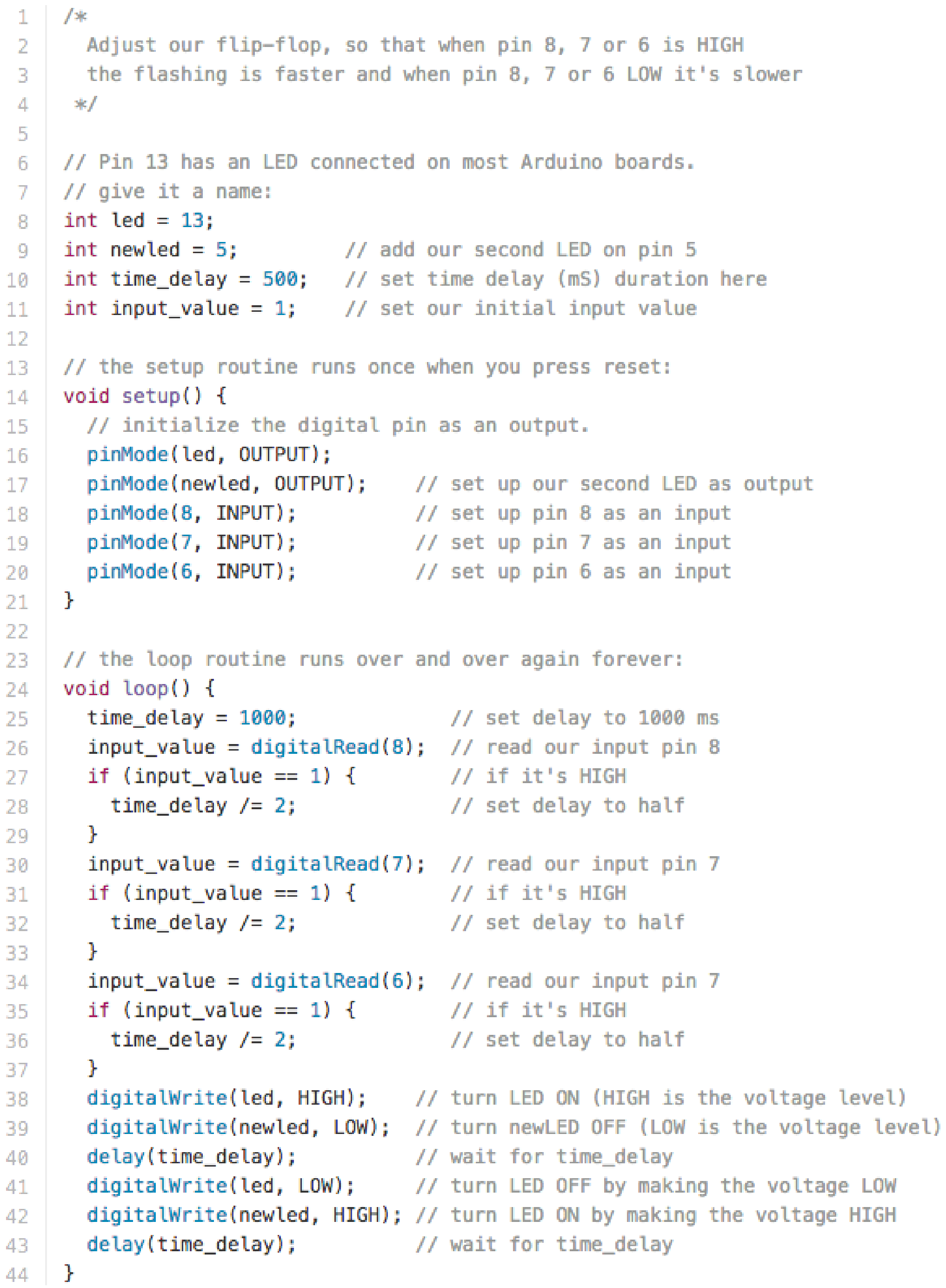
Code for Flip-flop with input controlling frequency

You can also find the above code here...

https://github.com/raspitv/raspio_duino/blob/master/flip_flop_with_input/flip_flop_with_input.ino

Flip-flop With 3 Inputs

Now we're going to use inputs on ports 7 and 6 as well and each pin which is HIGH will halve the `time_delay` so if all three ports are HIGH it will be only 125 ms.



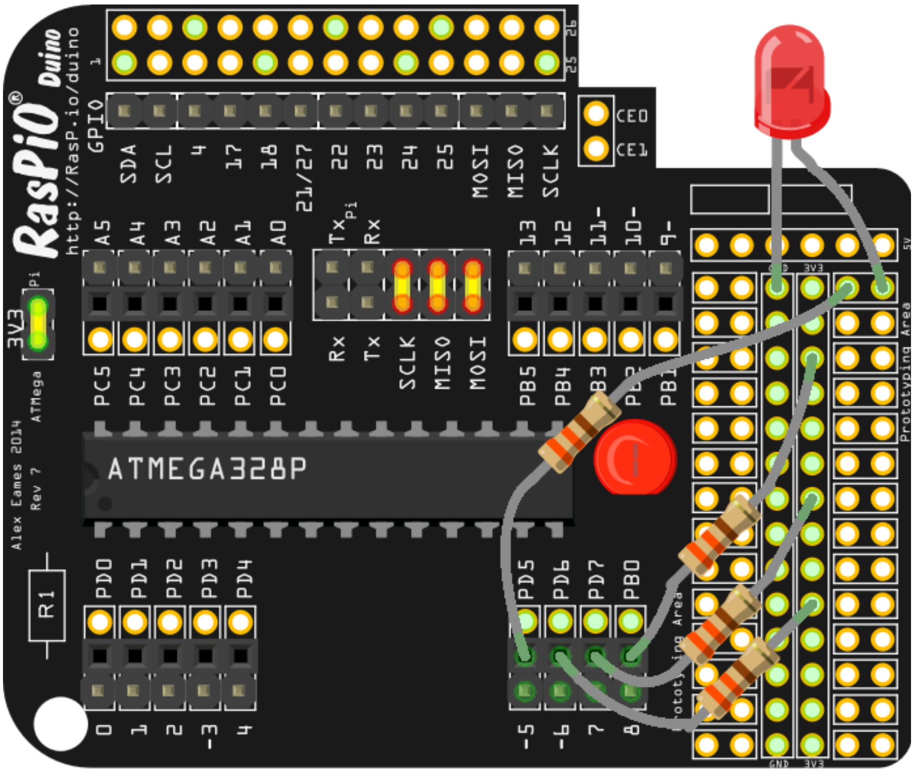
```
/*
  Adjust our flip-flop, so that when pin 8, 7 or 6 is HIGH
  the flashing is faster and when pin 8, 7 or 6 LOW it's slower
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
int newled = 5;         // add our second LED on pin 5
int time_delay = 500;   // set time delay (mS) duration here
int input_value = 1;    // set our initial input value

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
  pinMode(newled, OUTPUT);    // set up our second LED as output
  pinMode(8, INPUT);          // set up pin 8 as an input
  pinMode(7, INPUT);          // set up pin 7 as an input
  pinMode(6, INPUT);          // set up pin 6 as an input
}

// the loop routine runs over and over again forever:
void loop() {
  time_delay = 1000;             // set delay to 1000 ms
  input_value = digitalRead(8);  // read our input pin 8
  if (input_value == 1) {        // if it's HIGH
    time_delay /= 2;             // set delay to half
  }
  input_value = digitalRead(7);  // read our input pin 7
  if (input_value == 1) {        // if it's HIGH
    time_delay /= 2;             // set delay to half
  }
  input_value = digitalRead(6);  // read our input pin 7
  if (input_value == 1) {        // if it's HIGH
    time_delay /= 2;             // set delay to half
  }
  digitalWrite(led, HIGH);    // turn LED ON (HIGH is the voltage level)
  digitalWrite(newled, LOW);  // turn newLED OFF (LOW is the voltage level)
  delay(time_delay);          // wait for time_delay
  digitalWrite(led, LOW);     // turn LED OFF by making the voltage LOW
  digitalWrite(newled, HIGH); // turn LED ON by making the voltage HIGH
  delay(time_delay);          // wait for time_delay
}
```

Code for flip_flop_with_3inputs

Lines 19-20 are new, setting up pins 7 and 6 as inputs.
Line 25 is different too. Instead of using `else` three times I decided to set the `time_delay` to 1000 at the start of each loop and then divide it by two for each HIGH pin we find. This is what we're doing in lines 27-37. We're using the same variable `input_value` and reading each pin in turn. If an input pin is HIGH, `time_delay /= 2;` divides `time_delay` by 2.[8]

```
if (input_value == 1) {
  time_delay /= 2;
}
```

To play with this circuit, connect each port to 3V3 using the 330 Ohm (330R) resistors. You can also use jumper wires for this, but using resistors affords the ports a little protection (and I know you've got resistors because they were in the kit).



Flip-flop with 3 inputs

With all three input pins connected to 3V3, they will all read HIGH, so the 1000 ms delay time will be divided by 2 three times in quick succession, becoming 500, then 250, then 125 ms.

Observe what happens when you connect one or more to GND instead of 3V3. The flashing should get slower. When all three are connected to GND, the delay should be 1000 ms (1 s).

[8] A /= B divides A by B and assigns the result to variable A. Its the same for other compound operators *=, +=, -= **http://arduino.cc/en/Reference/IncrementCompound**

Get Rid Of Repeated Code Using Loops

You may have noticed in the *flip-flop with three inputs* sketch that we have almost the same code repeated three times from lines 18-20 and 26-37. Good programmers hate to see repeated code. It's inefficient. A more efficient way to do this would be to make a loop and use a variable for the pin number.

Lines 18-20 of our flip-flop with 3 inputs look like this...

```
pinMode(8, INPUT);
pinMode(7, INPUT);
pinMode(6, INPUT);
```

It's just about worth looping this for the three values here, but you'll see later on, where there are several lines of repeating code, it's definitely worthwhile. Here's how to do a loop...

```
for (initialisation; condition; increment){
   // code you want to repeat
}
```

`initialisation` is our starting point.
`condition` is the conditional test to carry on looping or stop.
`increment` sets the value by which we change our loop variable each time around (iteration)

...and here's what it looks like for our situation...

```
for (int thisPin = 6; thisPin < 9; thisPin++) {
   pinMode(thisPin, INPUT); // set pins 6-8 as inputs
}
```

This code shows we want to loop with the statement `for (),` then declares `thisPin` as an integer variable with a value of 6. That sets the starting value. Note the semi-colon ; which separates it from the end value.

`thisPin < 9;` sets the end value. The loop will execute while `thisPin` is less than 9.

`thisPin++) {` shows that we want to increment the value of `thisPin` by 1 each time round. The closing `)` and opening { show the end of the `for` statement and the start of the block of code we're going to be repeating (looping through).

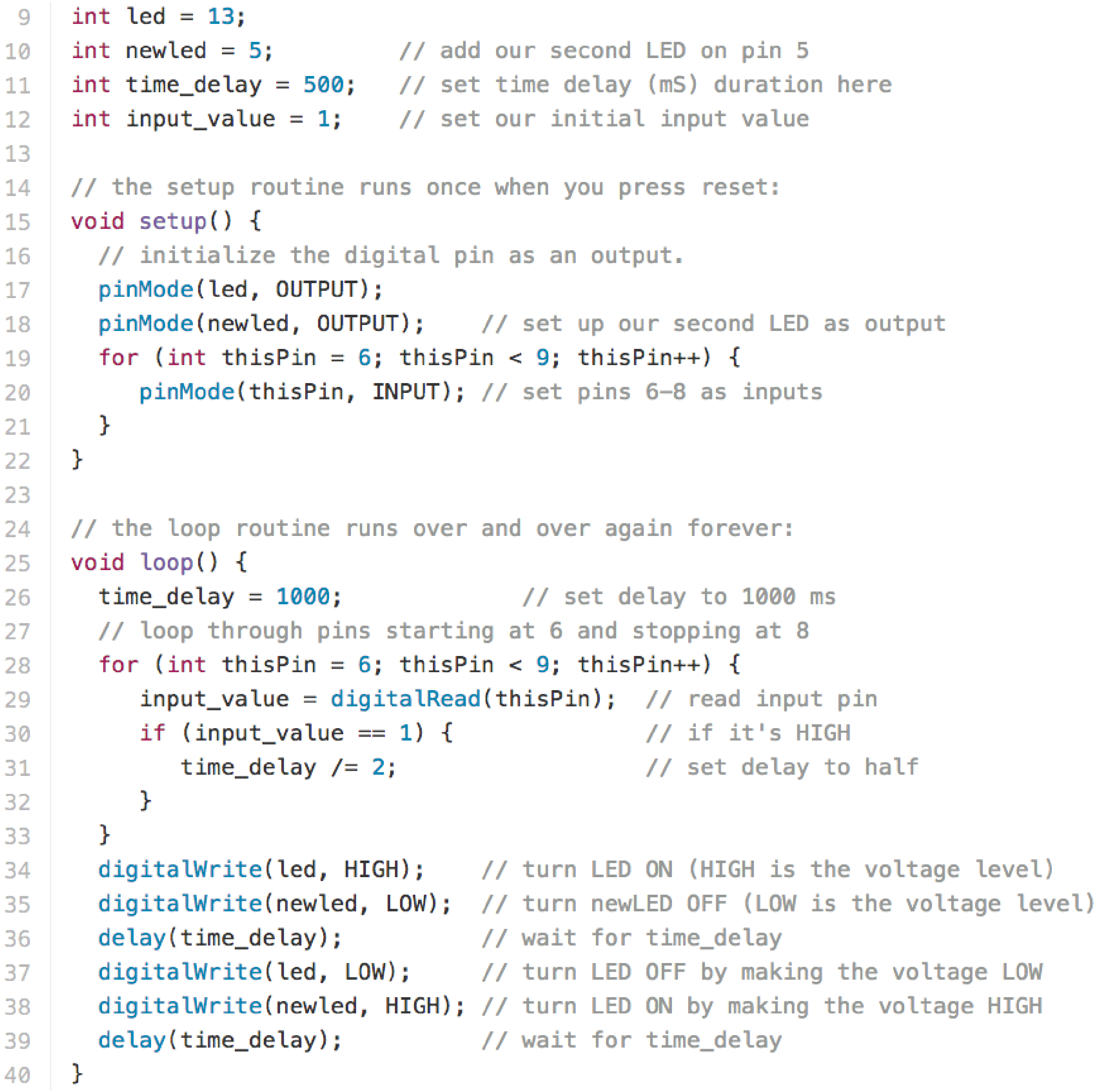
Further reading on for loops
http://arduino.cc/en/Reference/For

We can use exactly the same `for` loop method to condense lines 26-36 of our flip-flop with 3 inputs sketch to just 5 lines and eliminate lots of duplicated code in our main program loop

```
for (int thisPin = 6; thisPin < 9; thisPin++) {
   input_value = digitalRead(thisPin);
   if (input_value == 1) {
      time_delay /= 2;
   }
```

This doesn't change what the sketch does, but makes it work more efficiently. It's good to get into the habit of trying to write efficient code.

The first few lines (comments) have been removed in the code below, but you can see the full code here...
https://github.com/raspitv/raspio_duino/blob/master/flip_flop_with_3inputs_loop/flip_flop_with_3inputs_loop.ino



```
int led = 13;
int newled = 5;          // add our second LED on pin 5
int time_delay = 500;    // set time delay (mS) duration here
int input_value = 1;     // set our initial input value

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
  pinMode(newled, OUTPUT);     // set up our second LED as output
  for (int thisPin = 6; thisPin < 9; thisPin++) {
     pinMode(thisPin, INPUT); // set pins 6-8 as inputs
  }
}

// the loop routine runs over and over again forever:
void loop() {
  time_delay = 1000;               // set delay to 1000 ms
  // loop through pins starting at 6 and stopping at 8
  for (int thisPin = 6; thisPin < 9; thisPin++) {
     input_value = digitalRead(thisPin);  // read input pin
     if (input_value == 1) {              // if it's HIGH
        time_delay /= 2;                  // set delay to half
     }
  }
  digitalWrite(led, HIGH);     // turn LED ON (HIGH is the voltage level)
  digitalWrite(newled, LOW);   // turn newLED OFF (LOW is the voltage level)
  delay(time_delay);           // wait for time_delay
  digitalWrite(led, LOW);      // turn LED OFF by making the voltage LOW
  digitalWrite(newled, HIGH); // turn LED ON by making the voltage HIGH
  delay(time_delay);           // wait for time_delay
}
```

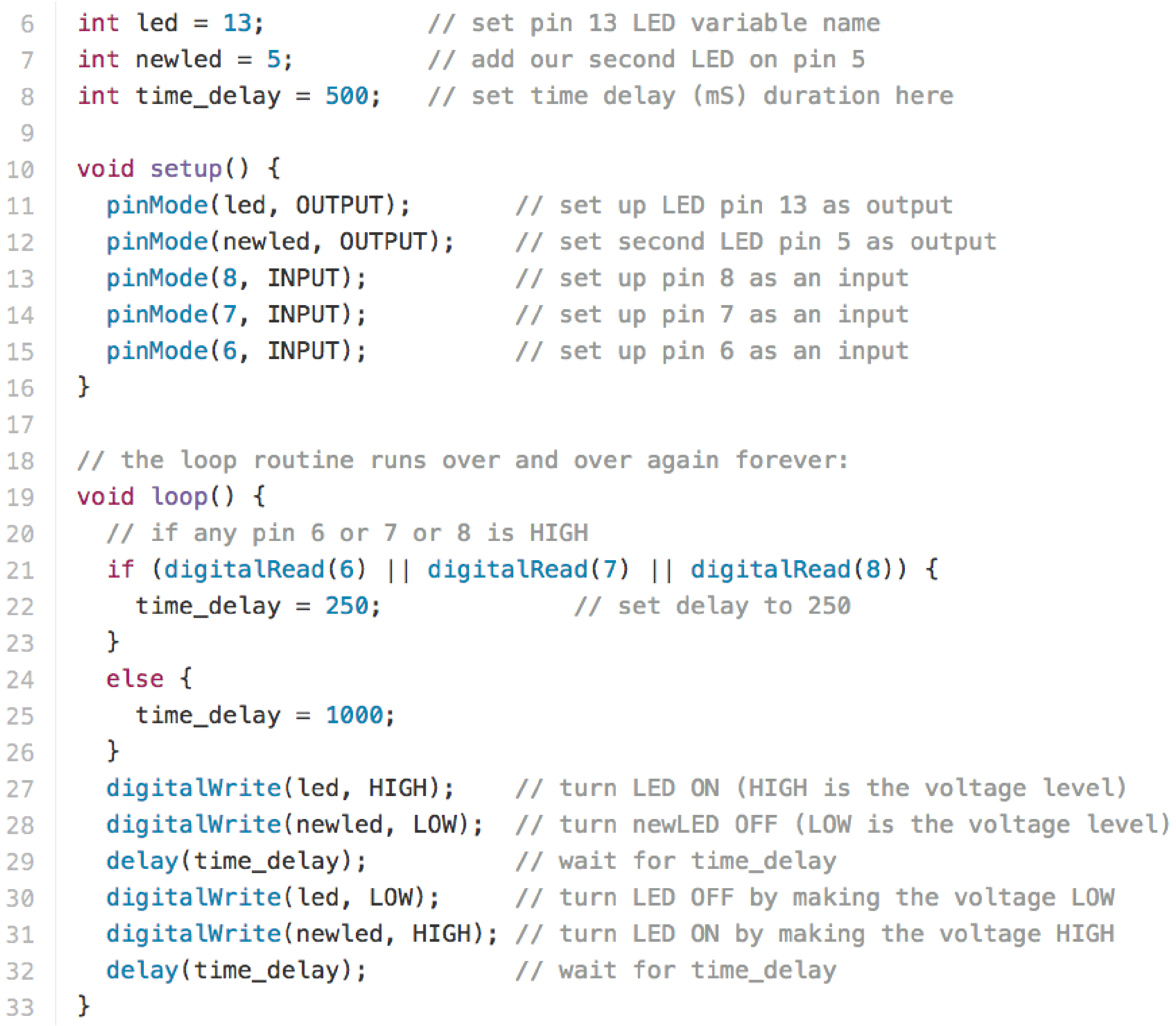
Code for flip_flop_with_3inputs_loop

Or || And &&   Not ! Boolean Operators

While we have the “flip-flop with 3 inputs” circuit made up, it's a good opportunity to show another way to make decisions. This time I want to show you how you can make decisions in your programs by combining inputs.

This time, we'll change our code so that the flip-flop LEDs will swap every 1000 ms if none of the three inputs are HIGH and every 250 ms if any of the three inputs are HIGH.

I've cut the initial comments out, you can see the full code here...
https://github.com/raspitv/raspio_duino/blob/master/flip_flop_with_3inputs_boolean/flip_flop_with_3inputs_boolean.ino



```
int led = 13;           // set pin 13 LED variable name
int newled = 5;         // add our second LED on pin 5
int time_delay = 500;   // set time delay (mS) duration here

void setup() {
  pinMode(led, OUTPUT);       // set up LED pin 13 as output
  pinMode(newled, OUTPUT);    // set second LED pin 5 as output
  pinMode(8, INPUT);          // set up pin 8 as an input
  pinMode(7, INPUT);          // set up pin 7 as an input
  pinMode(6, INPUT);          // set up pin 6 as an input
}

// the loop routine runs over and over again forever:
void loop() {
  // if any pin 6 or 7 or 8 is HIGH
  if (digitalRead(6) || digitalRead(7) || digitalRead(8)) {
    time_delay = 250;             // set delay to 250
  }
  else {
    time_delay = 1000;
  }
  digitalWrite(led, HIGH);    // turn LED ON (HIGH is the voltage level)
  digitalWrite(newled, LOW);  // turn newLED OFF (LOW is the voltage level)
  delay(time_delay);          // wait for time_delay
  digitalWrite(led, LOW);     // turn LED OFF by making the voltage LOW
  digitalWrite(newled, HIGH); // turn LED ON by making the voltage HIGH
  delay(time_delay);          // wait for time_delay
}
```

Code for flip_flop_with_3inputs_boolean

Comparing our new sketch with the simple **flip_flop_with_3inputs** sketch, this time we haven't defined an `input_value` variable at the top. The reason for this is that were going to read the inputs directly and make immediate decisions based on their values without using a variable.

How do we do this? It's very simple and similar to Python. We use the code (line 21, simplified)...

```
if (digitalRead(6)) {}
```

This works effectively the same as what we did before...

```
input_value = digitalRead(6);
if (input_value == 1) {}
```

...but because we don't need to store the value of the input for later use in the program, we can eliminate the variable `input_value` and streamline the code a bit.

`if (digitalRead(6)) {}` is how you'd do it for just one input, but we're using three. We want to speed up the flip-flop if any of input 6 or input 7 or input 8 is HIGH (==1). We can use the Boolean operator for OR, which is `||`

```
if (digitalRead(6) || digitalRead(7) || digitalRead(8)) {
  time_delay = 250;
}
```

This literally tests "if 6 is HIGH OR 7 is high OR 8 is HIGH". So if any of the three input ports reads HIGH, the code in the `{}` block will execute and the time delay will be shortened to 250 ms.

As well as OR `||`, there are two other **Boolean operators** we can use in our sketches to make decisions and determine the flow of our code.

AND is `&&`
NOT is `!`

So if we wanted to, we could easily change line 21 so that the time delay would only be 250 ms if *all three input pins* were HIGH by simply changing the two `||` to `&&`.

```
if (digitalRead(6) && digitalRead(7) && digitalRead(8)) {
  time_delay = 250;
}
```

This literally tests "if 6 is HIGH AND 7 is HIGH AND 8 is HIGH".

Not `!`, is used to invert our logic. Supposing we wanted to test for the situation where 6 and 8 are HIGH but 7 must be LOW. It's easy. We just invert `digitalRead(7)` by putting a `!` in front of it. If it's NOT HIGH, it must be LOW.

```
if (digitalRead(6) && ! digitalRead(7) && digitalRead(8)) {
  time_delay = 250;
}
```

Note that it is important to use `||` and `&&` for logical OR and AND, since single `|` and `&` are used for bitwise OR and AND, which is beyond the scope of this text.

Further info on bitwise operators
http://www.arduino.cc/en/Reference/BitwiseAnd

Eliminating Floating Signals

When playing around with the "3-inputs" sketches you may have noticed erratic behaviour. Sometimes the flip-flop might not change exactly when you think it should. The reason for this is that input ports (pins) require a very small amount of current to change their state from LOW to HIGH. Sometimes, a wire connected to a port can act as an antenna for stray radio signals or even static charge from your hand, which can be enough to flip the state of a port. Clearly this is not ideal. We want to be in control. If we're using input ports to decide when things happen, we don't want them changing state at some random times determined by spurious radio signals or hand movements.

So the way this is achieved is by using what's called a pull-up resistor. This "holds" the port (pin) HIGH, unless a definite GND (LOW, 0V) connection is made. The good news is that the ATMEGA328P-PU chip has pull-up resistors built-in, so all we have to do is activate them, and that's quite easy to do. We just use...

```
pinMode(8, INPUT_PULLUP);
```

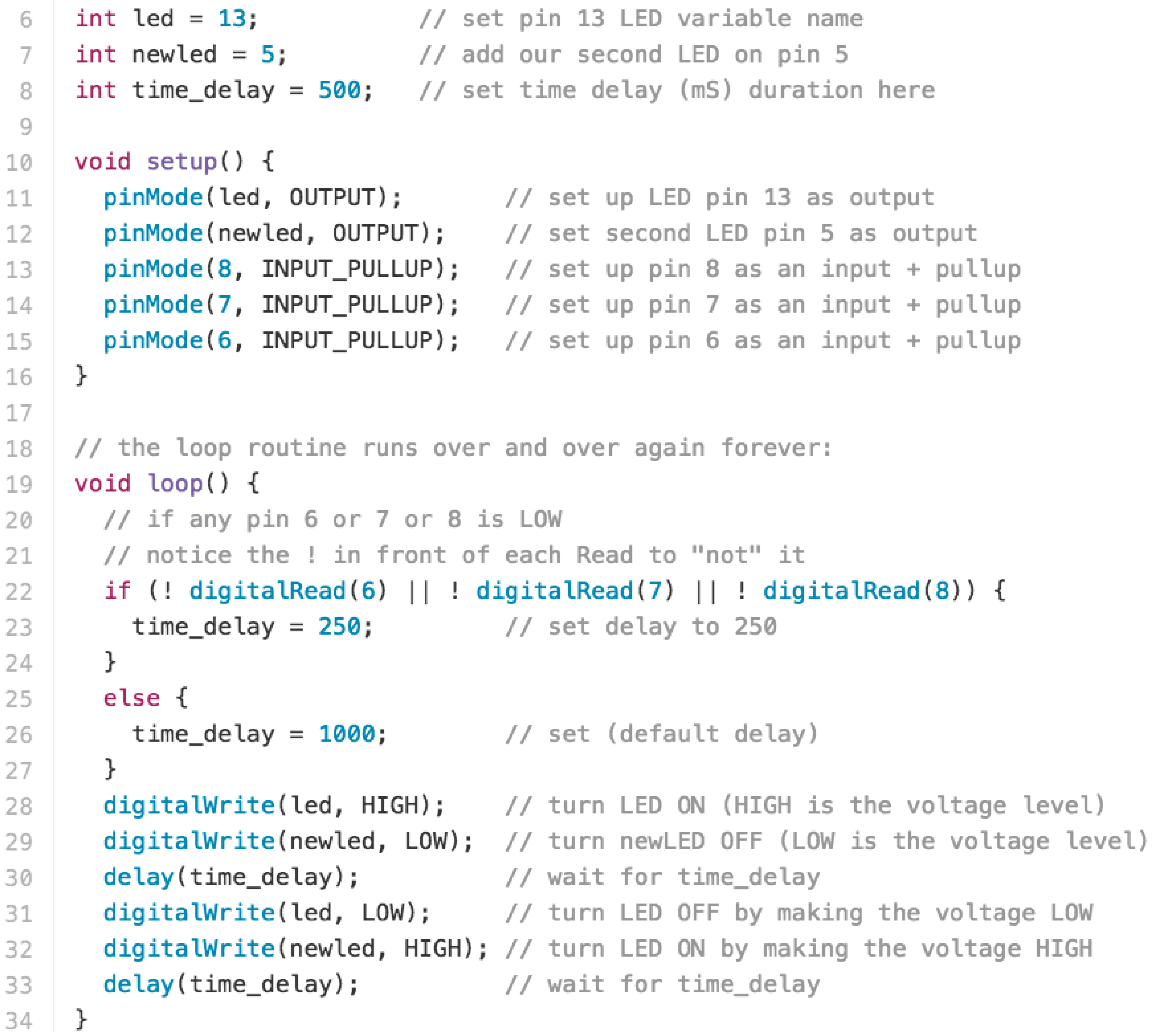
...instead of...

```
pinMode(8,INPUT);
```

...when we're setting up the pins.

In our case we will have to change the logic of our sketch around too because we've been testing to see if the pins were HIGH. When they are pulled up, they will be HIGH by default unless connected to GND (LOW). So we'll need to change the logic of our sketch to reflect this, and now we should be in control *and* immune from stray signals.

The modified code minus the initial comments follows on page 29. Additionally, you can see the full code here...
https://github.com/raspitv/raspio_duino/blob/master/flip_flop_with_3inputs_boolean_pull/flip_flop_with_3inputs_boolean_pull.ino



```
 6  int led = 13;             // set pin 13 LED variable name
 7  int newled = 5;           // add our second LED on pin 5
 8  int time_delay = 500;     // set time delay (mS) duration here
 9
10  void setup() {
11    pinMode(led, OUTPUT);        // set up LED pin 13 as output
12    pinMode(newled, OUTPUT);     // set second LED pin 5 as output
13    pinMode(8, INPUT_PULLUP);    // set up pin 8 as an input + pullup
14    pinMode(7, INPUT_PULLUP);    // set up pin 7 as an input + pullup
15    pinMode(6, INPUT_PULLUP);    // set up pin 6 as an input + pullup
16  }
17
18  // the loop routine runs over and over again forever:
19  void loop() {
20    // if any pin 6 or 7 or 8 is LOW
21    // notice the ! in front of each Read to "not" it
22    if (! digitalRead(6) || ! digitalRead(7) || ! digitalRead(8)) {
23      time_delay = 250;          // set delay to 250
24    }
25    else {
26      time_delay = 1000;         // set (default delay)
27    }
28    digitalWrite(led, HIGH);     // turn LED ON (HIGH is the voltage level)
29    digitalWrite(newled, LOW);   // turn newLED OFF (LOW is the voltage level)
30    delay(time_delay);           // wait for time_delay
31    digitalWrite(led, LOW);      // turn LED OFF by making the voltage LOW
32    digitalWrite(newled, HIGH);  // turn LED ON by making the voltage HIGH
33    delay(time_delay);           // wait for time_delay
34  }
```

Code for flip_flop_with_3inputs_boolean_pull

In lines 13-15 we're setting up the ports with pullup enabled.

```
pinMode(8, INPUT_PULLUP);
```

This holds the ports HIGH unless they are connected to GND.

In line 22, we've inverted the logic using three !

```
if (! digitalRead(6) || ! digitalRead(7) || ! digitalRead(8)) {}
```

We had to do this because we're using pullups to keep the pins HIGH unless connected to GND.

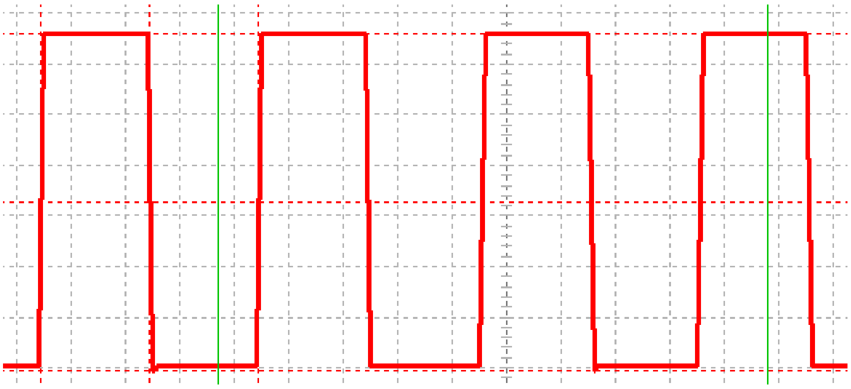
Further information on Boolean operators here...
http://arduino.cc/en/Reference/Boolean

Some PWM Experiments

PWM stands for pulse-width modulation. Put simply, this is a signal that is switched between on and off, usually rather quickly.

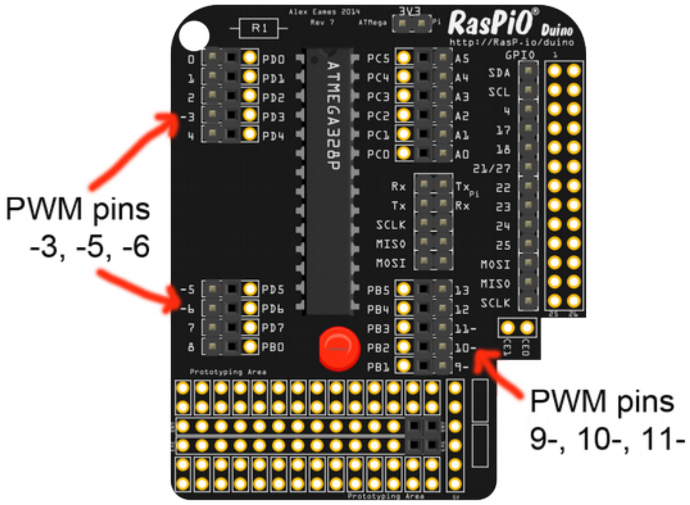
Do you remember when you were young, how you sometimes flicked the light switch on and off repeatedly, really quickly, to see what would happen? And you were told not to do it in case something bad happened (and it never did)? That was crude PWM.

It's a "square" waveform (on or off most of the time) and looks something like this…



A typical PWM "square" waveform

PWM is used for controlling a variety of devices – motor speed, servo positions and many other things. You can even use PWM to vary the brightness of leds by switching them on and off at varying frequency and duration.



PWM pins on RasPiO Duino

Since we have LEDs in the kit, we're going to demonstrate the use of the ATMEGA's PWM pins using those.

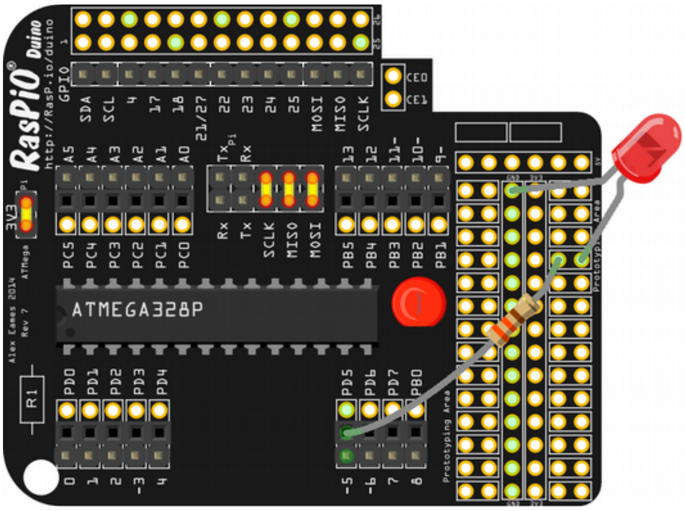
RasPiO Duino has 6 pins which can be used for PWM.

They are all labelled with a - to identify them (-3, -5, -6, 9-, 10-, 11-).

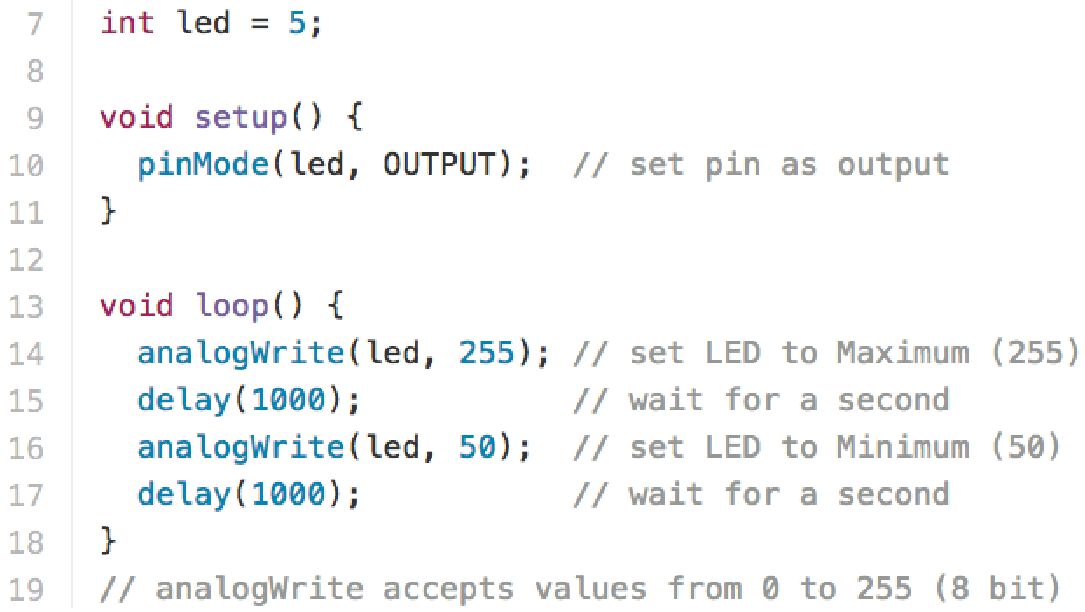
BlinkPWM

To use PWM you set the pin up as an output and then use `analogWrite(pin, value);` where `pin` is the PWM pin number (no minus signs, just positive integer) and `value` is an integer from 0 (fully off) to 255 (fully on).

So let's make a modified version of the blink sketch, using pin 5 with PWM and a red LED and resistor, like this...



This code will toggle the LED between low brightness and full brightness...



```
 7  int led = 5;
 8
 9  void setup() {
10    pinMode(led, OUTPUT);   // set pin as output
11  }
12
13  void loop() {
14    analogWrite(led, 255); // set LED to Maximum (255)
15    delay(1000);           // wait for a second
16    analogWrite(led, 50);  // set LED to Minimum (50)
17    delay(1000);           // wait for a second
18  }
19  // analogWrite accepts values from 0 to 255 (8 bit)
```

Code for BlinkPWM

Full code is found here...

https://github.com/raspitv/raspio_duino/blob/master/BlinkPWM/BlinkPWM.ino

Notice in line 10 we don't do anything different to set up the PWM pin. We just set it up as an output.

```
pinMode(led, OUTPUT);
```

Then in lines 14-17 we just alternate between full brightness...

```
analogWrite(led, 255);
```

...and minimal (but not zero) brightness...

```
analogWrite(led, 50);
```

...with a 1 second (1000 ms) delay. This keeps going *ad infinitum*.

Have a go at changing the PWM values and see what happens. Anything from 0 to 255 will work.

That's the basics of PWM, but let's take it a little further and do something a bit more interesting.

BlinkPWM_loop

In the previous sketch we alternately PWMed the values 50 and 255. That just flipped the LED between low and high brightness states. Now we're going to use a couple of `for` loops to vary the PWM value from 0 to 255 and back to 0 again, very quickly, in steps of 1. This will give us a continuous variation in brightness, almost like "breathing".

The circuit is the same as before, but we're adding some new code (**see page 33**) to handle the fading...

https://github.com/raspitv/raspio_duino/blob/master/BlinkPWM_loop/BlinkPWM_loop.ino

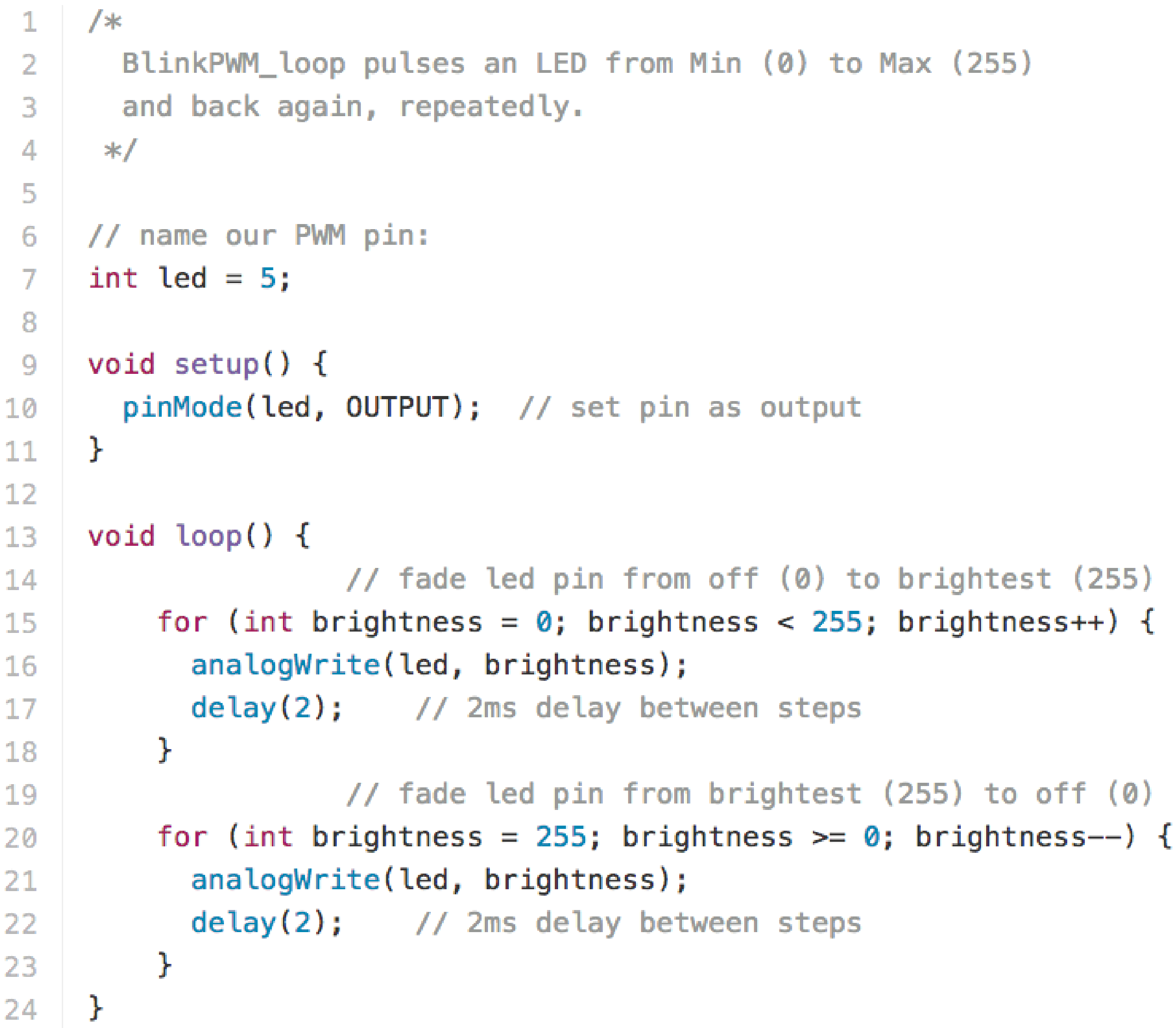
In lines 15-18 we vary the PWM value from 0 to 255 in steps of 1, with a 2 ms pause at each value. So it should take a little over half a second to transition from fully off to fully on. Notice the use of `brightness++` in line 15 to increment in steps of 1...

```
for (int brightness = 0; brightness < 255; brightness++) {
```

In lines 20-23 we do the opposite and go back down from 255 to 0 using `brightness--` to decrement in steps of 1...

```
for (int brightness = 255; brightness >= 0; brightness--) {
```

And once that's happened, the main loop (lines 9-24) repeats.



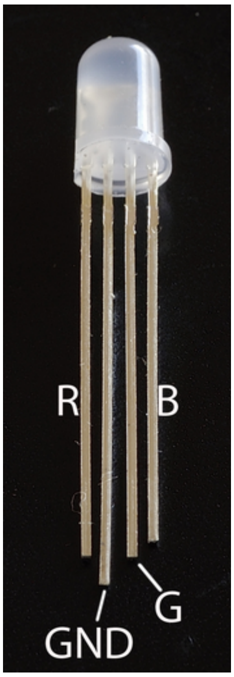
```
/*
  BlinkPWM_loop pulses an LED from Min (0) to Max (255)
  and back again, repeatedly.
 */

// name our PWM pin:
int led = 5;

void setup() {
  pinMode(led, OUTPUT);  // set pin as output
}

void loop() {
               // fade led pin from off (0) to brightest (255)
    for (int brightness = 0; brightness < 255; brightness++) {
      analogWrite(led, brightness);
      delay(2);    // 2ms delay between steps
    }
               // fade led pin from brightest (255) to off (0)
    for (int brightness = 255; brightness >= 0; brightness--) {
      analogWrite(led, brightness);
      delay(2);    // 2ms delay between steps
    }
}
```

Code for BlinkPWM_loop

Challenge

At this point, if you fancy a challenge, you could modify the above code to add three inputs, as we did before with the **flip_flop_with3inputs**, to control the speed.
(If you get stuck, there is an example in the Github repository, but you'll have to find it yourself, I'd rather you worked it out than just crib the answer.)

PWM + RGB LED



RGB LED

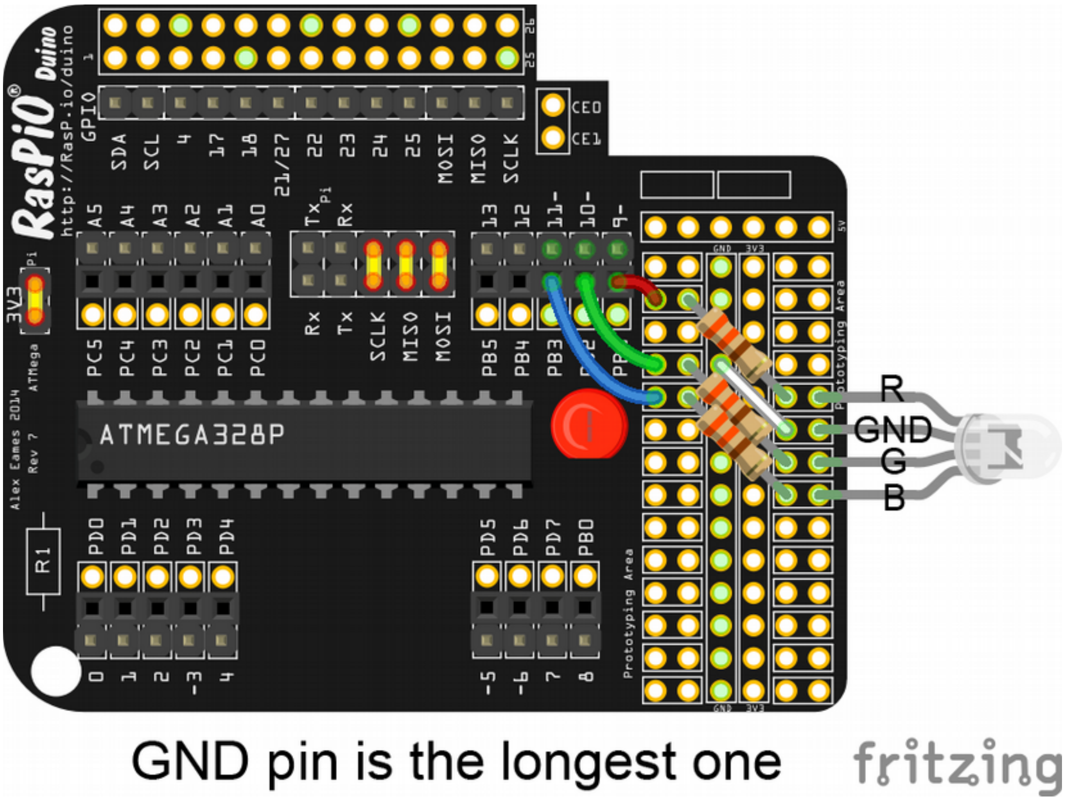
So now it's time for us to break out the 3 colour LED. Most computer screens these days have millions of red, green and blue (RGB) pixels. Mixing these three colours of emitted light, it's possible to make virtually any colour.

We can do a similar thing with our RGB LED. If we use three PWM pins, we can make 256 x 256 x 256 different possible combinations. Not all of these will be noticeably different from each other, but that's how we get smooth transitions.

An RGB LED has four pins. The longest one (in our case) is GND. We're using a common cathode[9] LED.

Red +ve is the pin on its own next to to the long GND pin. Blue +ve is the other outer pin and Green +ve is between GND and Blue +ve.

Let's build our circuit...

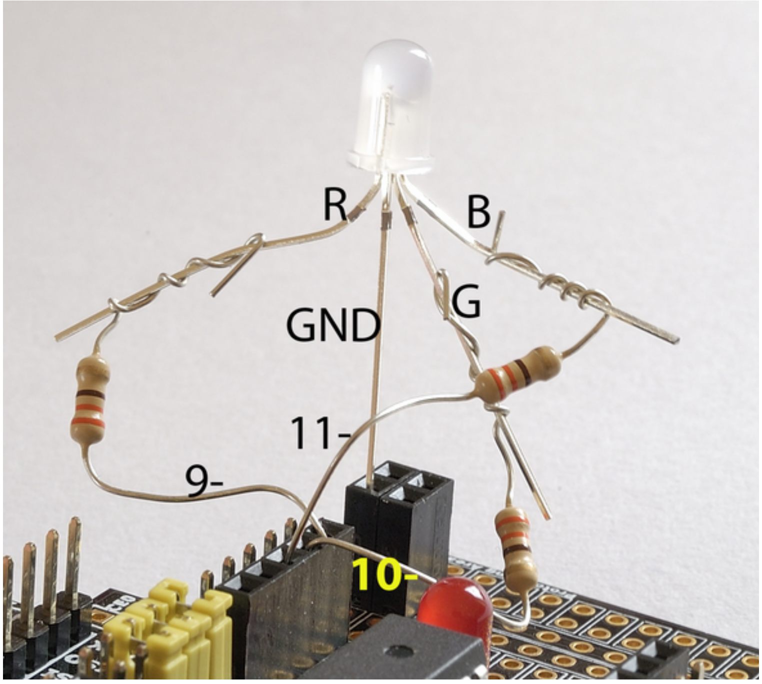


Circuit for BlinkPWM_flip_flop_RGB

As before, you can join the 330 Ω resistors to the LED by twisting them carefully around the LED legs, ensuring that none are touching each other. It looks a bit messy, but it works if you don't want to solder everything permanently to the board. You could also use a breadboard and jumper wires if

[9] You can also get common anode LEDs with opposite polarity.

you have them and prefer that approach.



Connecting RGB LED without soldering

The code for this is at...
https://github.com/raspitv/raspio_duino/blob/master/BlinkPWM_flip_flop_RGB/BlinkPWM_flip_flop_RGB.ino

The code is also on the next page. Let's have a look at the important parts...

In line 6 we define a variable for the `time_delay` and assign it a value of 2

```
int time_delay = 2;
```

In lines 9-11 we loop through pins 9, 10, 11 setting each one as an output

```
for (int rgbPin = 9; rgbPin < 12; rgbPin++) {
  pinMode(rgbPin, OUTPUT);
}
```

In lines 15-26 we've essentially taken the main loop code from the previous sketch BlinkPWM_loop and wrapped an extra loop around it so that it cycles through pins 9, 10 and 11 in turn instead of just PWMing one pin...

```
for (int rgbPin = 9; rgbPin < 12; rgbPin++) {
```

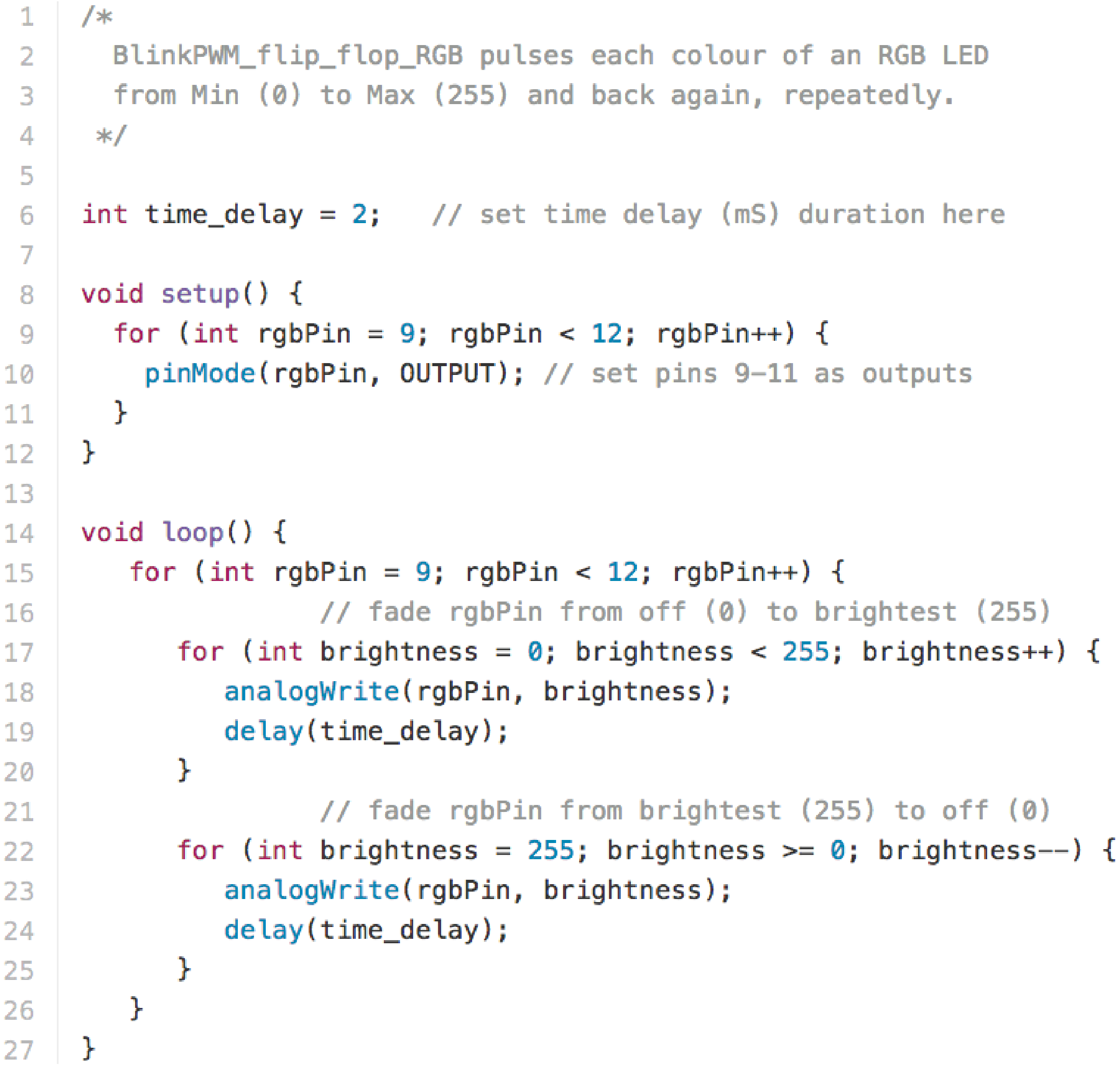
Main loop code from previous sketch (more or less)

```
}
```

The only other thing we did differently was to use a variable for the time delay.

```
delay(time_delay);
```

Here's the full code...

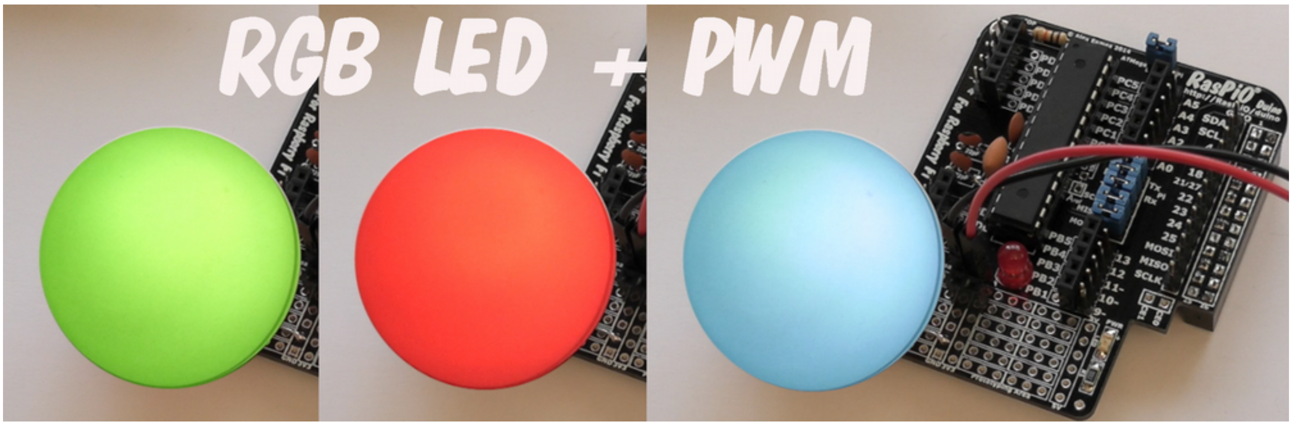


```
/*
  BlinkPWM_flip_flop_RGB pulses each colour of an RGB LED
  from Min (0) to Max (255) and back again, repeatedly.
 */

int time_delay = 2;    // set time delay (mS) duration here

void setup() {
  for (int rgbPin = 9; rgbPin < 12; rgbPin++) {
    pinMode(rgbPin, OUTPUT); // set pins 9-11 as outputs
  }
}

void loop() {
   for (int rgbPin = 9; rgbPin < 12; rgbPin++) {
               // fade rgbPin from off (0) to brightest (255)
      for (int brightness = 0; brightness < 255; brightness++) {
         analogWrite(rgbPin, brightness);
         delay(time_delay);
      }
               // fade rgbPin from brightest (255) to off (0)
      for (int brightness = 255; brightness >= 0; brightness--) {
         analogWrite(rgbPin, brightness);
         delay(time_delay);
      }
   }
}
```

Code for BlinkPWM_flip_flop_RGB

RGB LED Cross-fade

You may remember the colour-changing ball in the KickStarter video?



Well here's where I show you how I did it. This is a great lesson in itself. One of the great strengths of the Arduino ecosystem is that, pretty much whatever you want to do, someone has done it before and shared their code (a bit like I often do with Python code on **RasPi.TV**). In this case I wanted to do a colour-fading RGB LED. So I googled “colour fading RGB LED arduino” and the second link took me to **http://www.arduino.cc/en/Tutorial/ColorCrossfader**

I had a little look at the code, then copied and pasted it into the Arduino IDE to try it out. I needed to modify it slightly to use the pins that my RGB LED was connected to and I also created a couple of custom colours. But other than that, it was a “copy and paste job”. I'll also be honest here and tell you that I haven't fully taken the time to completely understand the maths of exactly how it works.

You won't always need to fully understand how it works to be able to use it. The fact is, it does exactly what I wanted, it was only a quick Google away, and it only required minor tweaks to make it my own. I also added a ping-pong ball on top of the RGB LED to act as a diffuser.

Fading is quite easy. We've already done it. Where it gets tricky is when you want to cross-fade with more than one pin changing at once. But there's no need to “reinvent the wheel” if someone has already done it for you. By all means look at the code and tweak it, change it, customise it and try to understand it. As long as you know enough to check that it's not doing anything malicious, don't be afraid to experiment (hack) with other people's code.

You can find my slightly modified version of the cross-fade code here...

https://github.com/raspitv/raspio_duino/blob/master/cross_fade/cross_fade.ino

I recommend you experiment with it. Try making a custom colour or changing the colour sequence to “make it your own”. Have fun! The main downside of RGB LEDs is that the red component is often a bit weak. You could try playing about with a lower value resistor to boost the red, but don't go below about 90 Ω or you run the risk of stressing the ATMEGA chip.

Using the Analog Inputs

Although I write in predominantly British English, I favour the use of "analog" rather than "analogue", simply because that's the way it's used in Arduino programming. So to avoid confusion, I shall use the US spelling here.

We live in an analog world, but computers can only "speak" digital information. For a computer to process information, it has to be converted into 0s and 1s, (On/Off, HIGH/LOW, 0V/3V3 etc).

So we need a mechanism of converting the analog information around us into digital form so that we can do something with it. To do this, we use a device called an analog to digital converter (also known as ADC or AD).

Your eyes, with their vast numbers of rods and cones are an incredible example of an analog to digital converter. Even the most modern camera sensor with all its millions of pixels cannot simultaneously determine as many colours, shades and intensities as your eyes.

Analog information is continuous, whereas digital information comes in steps. If you have enough steps, you can get a usefully high degree of measurement precision. The ATMEGA328P-PU has a 10-bit ADC on it. This means that it has 2^{10} = 1024 steps. This is referred to as "resolution". In practice, the ADC output values are 0-1023.

ADCs are used to measure the voltage of a signal. Most sensors are designed to output a voltage which is proportional to what they are measuring. So this gives us an elegant way of getting information from the analog world into the ATMEGA microcontroller.

The RasPiO Duino is run at 3V3 from the Raspberry Pi, so if the measured analog input signal is 3.3V, the ADC will output 1023. If the input signal is 0V, the ADC output will be 0.

If we divide 3.3V by 1023, we get the resolution of the device. 3.3V / 1023 = 0.00322 V/step That's 3 mV.

If we want to know what a sensor's voltage reading is we do the following calculation...

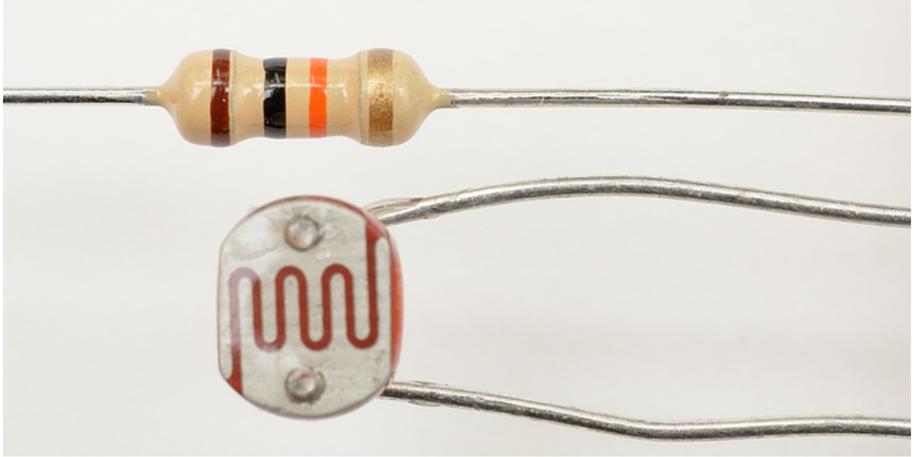
ADC reading / 1023 * 3.3 V = Sensor Voltage

From the sensor voltage, we can usually calculate temperature/pressure or whatever our sensor is measuring.

(There are ways to change the analog reference voltage from 3V3, but if you're going to do that, please **see the note on page 5** about cutting the A_{ref} track.)

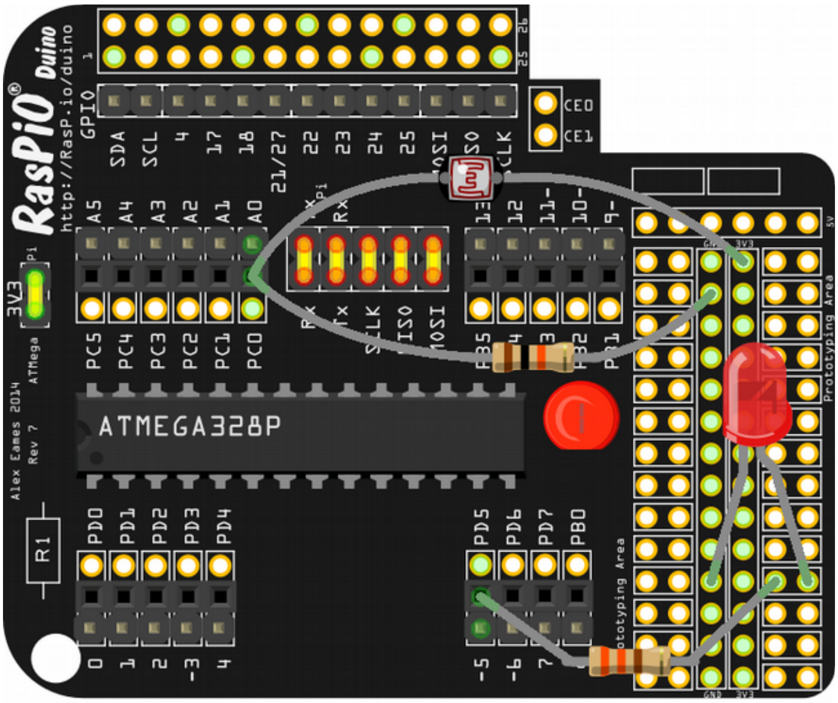
The RasPiO Duino has 6 analog input pins (A0-A5), so you can measure up to 6 sensors or voltages at once. This can be very useful indeed. (e.g. It's more than enough for a weather station.)

There's a light sensor in the RasPiO Duino kit so that we can experiment with the analog pins. It's called a light dependent resistor (LDR). When exposed to bright light it will offer almost no resistance, when in total darkness it will have a resistance of a couple of Mega-Ohms (MΩ).



LDR and 10kΩ resistor

But we want to measure voltage, not resistance, so we need an extra 10kΩ resistor (acting as one half of a voltage divider[10]) to read it using a circuit like this...



Circuit for LDR_LED

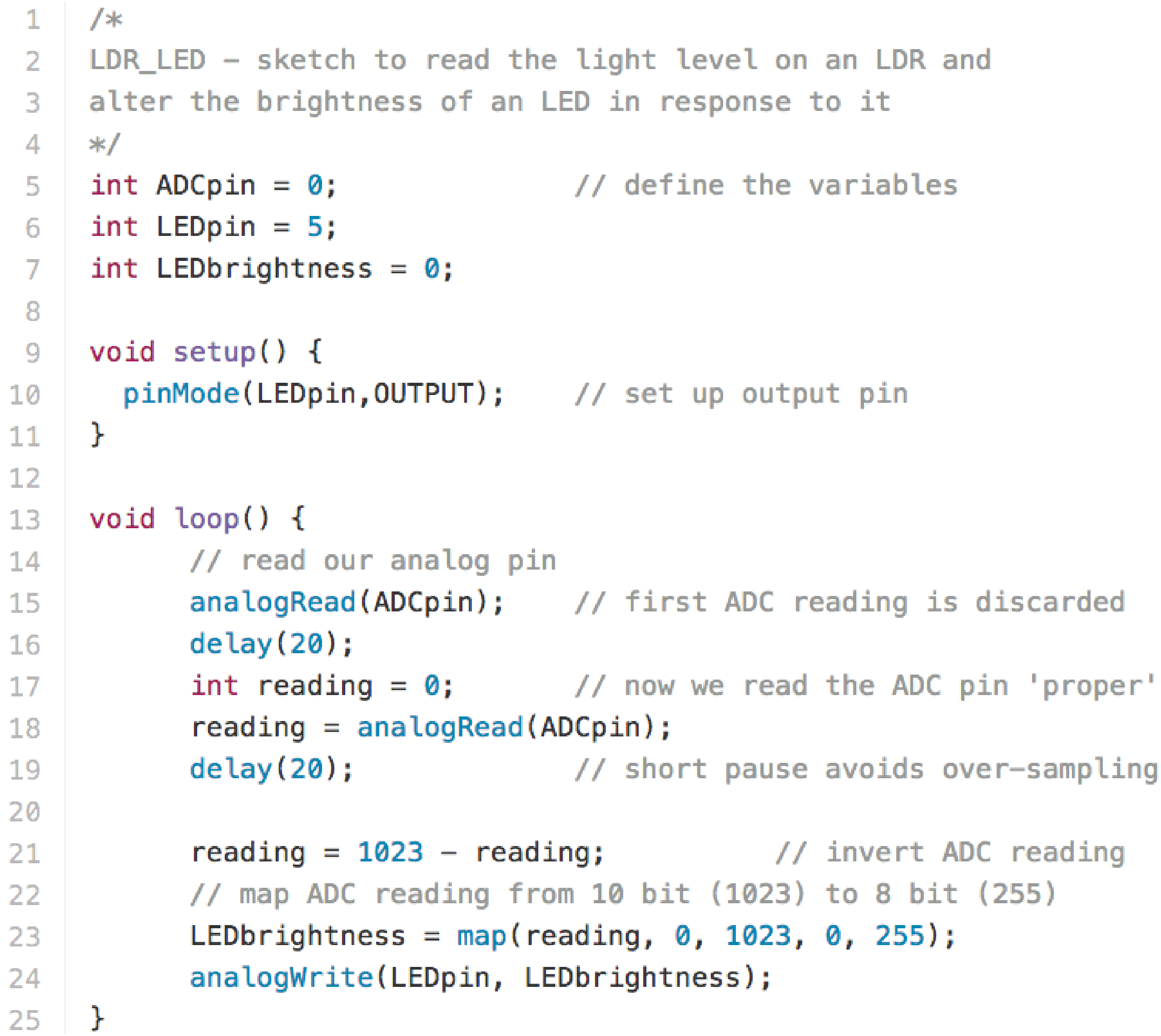
One end of the LDR is connected to 3V3 and the other to analog input A0. The 10kΩ resistor connects A0 to GND. (You can twist the A0 end of the resistor wire round the A0 end of the LDR wire to make a connection. They don't both have to fit in the female header socket.)

In darkness, the resistance of the LDR is very high and we get a voltage reading of ~0 V.
Under bright light the LDR resistance is very small and we get a reading of ~3.3 V.

[10] See **http://raspi.tv/2015/raspio-duino-as-a-lipo-monitor#divider** for more on voltage dividers.

LDR_LED

When there's a lot of light, the ADC reading from our LDR is high. When it's dark, the reading is low. So let's do a simple experiment where we vary the brightness of an LED according to how much light there is. When it's dark, the LED will be fully bright and when there's a lot of light the LED will be off.



```
 1  /*
 2  LDR_LED - sketch to read the light level on an LDR and
 3  alter the brightness of an LED in response to it
 4  */
 5  int ADCpin = 0;              // define the variables
 6  int LEDpin = 5;
 7  int LEDbrightness = 0;
 8
 9  void setup() {
10    pinMode(LEDpin,OUTPUT);    // set up output pin
11  }
12
13  void loop() {
14        // read our analog pin
15        analogRead(ADCpin);    // first ADC reading is discarded
16        delay(20);
17        int reading = 0;       // now we read the ADC pin 'proper'
18        reading = analogRead(ADCpin);
19        delay(20);             // short pause avoids over-sampling
20
21        reading = 1023 - reading;           // invert ADC reading
22        // map ADC reading from 10 bit (1023) to 8 bit (255)
23        LEDbrightness = map(reading, 0, 1023, 0, 255);
24        analogWrite(LEDpin, LEDbrightness);
25  }
```

Code for LDR_LED

In lines 5-7 we declare our integer variables.
In lines 9-11 we set up our output port for the LED
In line 15 we read our analog pin (A0) with `analogRead(ADCpin);`
Notice that we didn't need to set anything up in the setup function in order to do this.

Then we wait for 20 ms. We aren't going to do anything with this ADC reading. The reason for this is that, very often, your first ADC reading is 'spurious', so it can be good practice to just ignore it.

In lines 17-19 we're declaring a new integer variable to hold the new ADC reading, then reading A0 again. Then we wait 20ms again. If we didn't have any delays in our sketch, we'd be sampling the ADC faster than it could cope with. Our readings would be garbage.

In line 21 we invert our ADC reading value by subtracting it from 1023...

```
reading = 1023 - reading;
```

This allows us to make the LED bright when the light level is low, rather than high.

In line 23 we're using a built-in function called `map()`[11] which lets us 're-scale' our ADC reading from a number which is between 0 and 1023 to a number between 0 and 255. This is because our PWM `analogWrite()` function is 8 bit (0-255), whereas our ADC readings are 10-bit (0-1023).

Then in line 24 we write the re-scaled `LEDbrightness` value to the `LEDpin`. And the sketch loops round forever.

Once you've uploaded the sketch, you can play with it in four ways...

1. Shine a torch or bright light on the LDR – the LED should go dim
2. Cover the LDR with a dark pen lid, aluminium foil, or turn off the room lights if it's night time – the LED should be at full brightness
3. Disconnect the ends of the LDR and resistor from A0, and connect a wire or resistor from A0 to GND – the LED should be at full brightness
4. Disconnect the ends of the LDR and resistor from A0, and connect a wire or resistor from A0 to 3V3 – the LED should be fully off

LDR_LED_serial

You may have noticed in the **LDR_LED circuit diagram** that I sneaked in two extra connections unannounced. I'm talking about the Rx-Tx and Tx-Rx jumpers. You didn't actually need them for the previous sketch, but you will for this one because they connect the serial ports of the ATMEGA and the Pi so that they can communicate with each other.

So this time we're going to calculate the LDR's voltage level, using our ADC, and we're going to send that and the ADC value to the Pi through the serial port. On the Pi we'll look at that serial data stream in two different ways...

1. Using a terminal monitor program called minicom
2. Using a Python script

Both ways are good to know about and really useful. Minicom is great for observing the raw output from the RasPiO Duino's serial port. But if you ever want to actually use the data, you'll need a way of getting it into a program. In Python there is a module called pyserial pre-installed in Raspbian, so all we have to do is use it. Minicom will require installation. So let's install it now and then I'll show you the tweaks we need to make to the previous script to output data to the serial port.

```
sudo apt-get update              ...to update your package lists, then...
sudo apt-get install minicom     ...to install minicom.
```

[11] Further info on the map() function can be found here **http://www.arduino.cc/en/Reference/Map**

To run minicom, type...

minicom -b 9600 -o -D /dev/ttyAMA0 (the final character is a zero not a letter o)

...and it will open up a serial monitor on the screen. Let's have a look at that command...

minicom runs minicom

-b tells minicom we want to specify the baud rate (speed)

9600 is the baud rate we're using in our sketch (it needs to match)

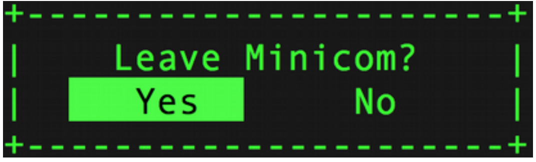
-o tells minicom not to initialise (useful for jumping in and out of minicom sessions)

-D allows us to specify the serial port

/dev/ttyAMA0 is the address of the Pi's serial port.

Once you're in minicom, you will see anything that gets written to your Pi's serial port on the screen. There won't be anything yet because we've not done our sketch. One more thing you need to know about minicom is how to get out of it.

To exit, you type CTRL+A and then X and then confirm YES by pressing ENTER.



```
+----------------------+
|    Leave Minicom?    |
|     Yes       No     |
+----------------------+
```

This will return you to your command line prompt.

So now we've got a way of monitoring the RasPiO Duino's serial output, let's create some serial output to monitor.

We're going to modify our previous sketch to...

Initialise the RasPiO Duino's serial port in the setup() function with...
Serial.begin(9600); (line 11)

Add one variable to store the ADC reading for later use int adc = reading; (line 20)

Calculate the voltage from our ADC reading (line 23)...
float voltage = reading * 3.3 / 1023.0;

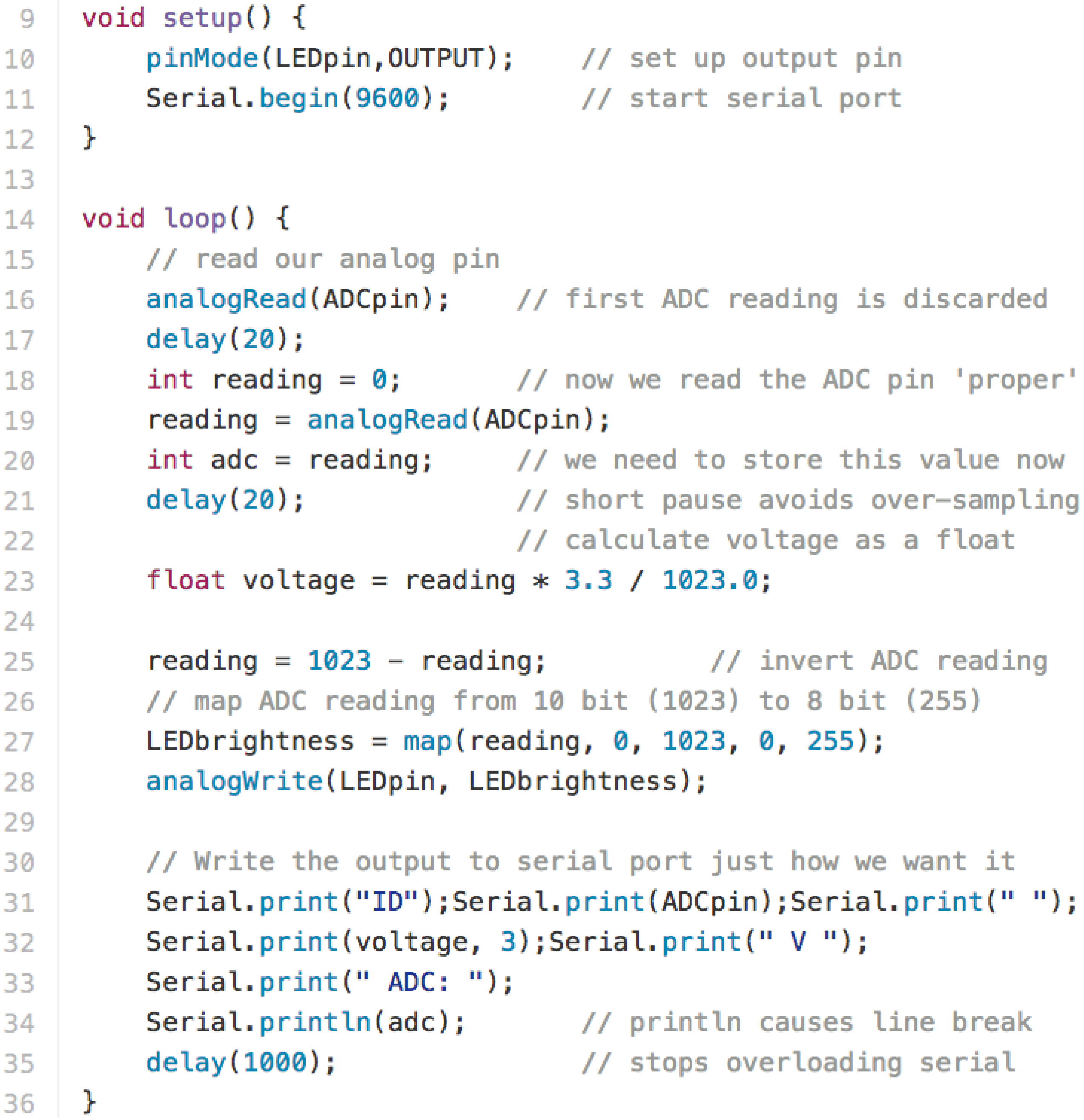
Notice that voltage is not an integer variable, but a floating point variable. This allows us to perform decimal operations on it, which you can't do on integers.

Then lines 31-34 use the Serial.print() function to "print" our output to the serial port.
Serial.print("ID");Serial.print(ADCpin);Serial.print(" ");
Line 31 prints ID followed by 0 followed by a space. This 'tags' our output and allows our Python program to identify it. Sometimes you get garbage output on the serial port, which can cause your

Python programs to crash out unexpectedly. So if you start each line with an identifying 'tag' you can get your program to ignore anything without the tag. In situations where you're using more than one analog pin, it also allows you to tell your Python program which ADC pin you're reporting. So it's a very useful trick.

Line 34 uses the `Serial.println()` function to “print to the serial port with a line break at the end” this lets the Pi know we've finished a line. Always ensure you finish your serial output with a line break. It avoids problems with pyserial.

Line 35 causes a 1 second delay (needed for the Python script to work properly).



```
 9  void setup() {
10      pinMode(LEDpin,OUTPUT);    // set up output pin
11      Serial.begin(9600);        // start serial port
12  }
13
14  void loop() {
15      // read our analog pin
16      analogRead(ADCpin);    // first ADC reading is discarded
17      delay(20);
18      int reading = 0;       // now we read the ADC pin 'proper'
19      reading = analogRead(ADCpin);
20      int adc = reading;     // we need to store this value now
21      delay(20);             // short pause avoids over-sampling
22                             // calculate voltage as a float
23      float voltage = reading * 3.3 / 1023.0;
24
25      reading = 1023 - reading;          // invert ADC reading
26      // map ADC reading from 10 bit (1023) to 8 bit (255)
27      LEDbrightness = map(reading, 0, 1023, 0, 255);
28      analogWrite(LEDpin, LEDbrightness);
29
30      // Write the output to serial port just how we want it
31      Serial.print("ID");Serial.print(ADCpin);Serial.print(" ");
32      Serial.print(voltage, 3);Serial.print(" V ");
33      Serial.print(" ADC: ");
34      Serial.println(adc);       // println causes line break
35      delay(1000);               // stops overloading serial
36  }
```

Code for LDR_LED_serial (first 8 lines same as LDR_LED)

Full code at...
https://github.com/raspitv/raspio_duino/blob/master/LDR_LED_serial/LDR_LED_serial.ino

Monitoring With Minicom

Let's upload this sketch to the RasPiO Duino and start minicom so we can look at the output on our screen...

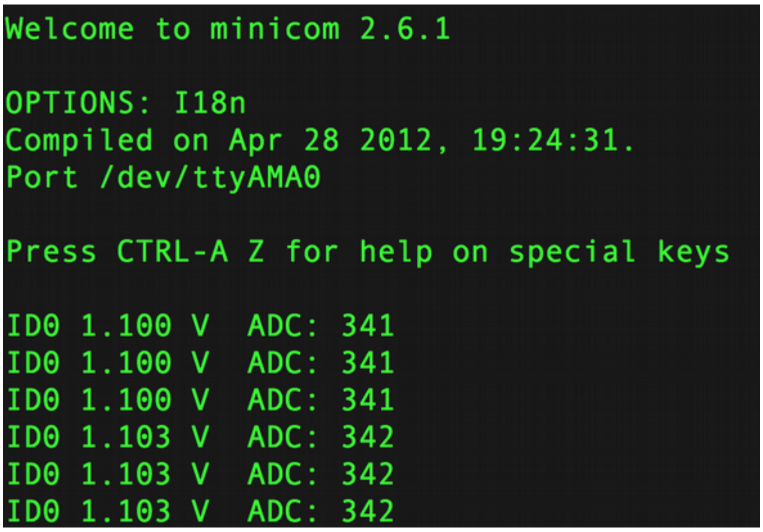
```
minicom -b 9600 -o -D /dev/ttyAMA0
```

Your output should look something like this.

Then if you shine a light or torch on your LDR, both the voltage and ADC readings should increase (unless you're already in bright light).

If you cover the LDR with something dark and opaque, the readings should decrease.

Readings should update on the screen once every second.



```
Welcome to minicom 2.6.1

OPTIONS: I18n
Compiled on Apr 28 2012, 19:24:31.
Port /dev/ttyAMA0

Press CTRL-A Z for help on special keys

ID0 1.100 V  ADC: 341
ID0 1.100 V  ADC: 341
ID0 1.100 V  ADC: 341
ID0 1.103 V  ADC: 342
ID0 1.103 V  ADC: 342
ID0 1.103 V  ADC: 342
```

Monitoring serial port with minicom

Monitoring Serial With A Python Script

Now we're going to use a Python script with pyserial to read the Pi's serial port instead of using minicom. The reason I'm showing you both methods is that minicom is brilliant for debugging. It shows you exactly what's being written to the Pi's serial port in real time. But minicom doesn't “do” anything with the data.

Python can be a bit 'finicky', particularly with regard to timings. I had initially written the previous sketch with a 500 ms delay instead of 1000 ms and the Python program could not read the serial port. It was only when I changed it to 1000 ms that it would work. Minicom worked fine with the 500 ms delay. It was useful to be able to see that the sketch was doing its job, but it still took me quite a long time to trace the issue.

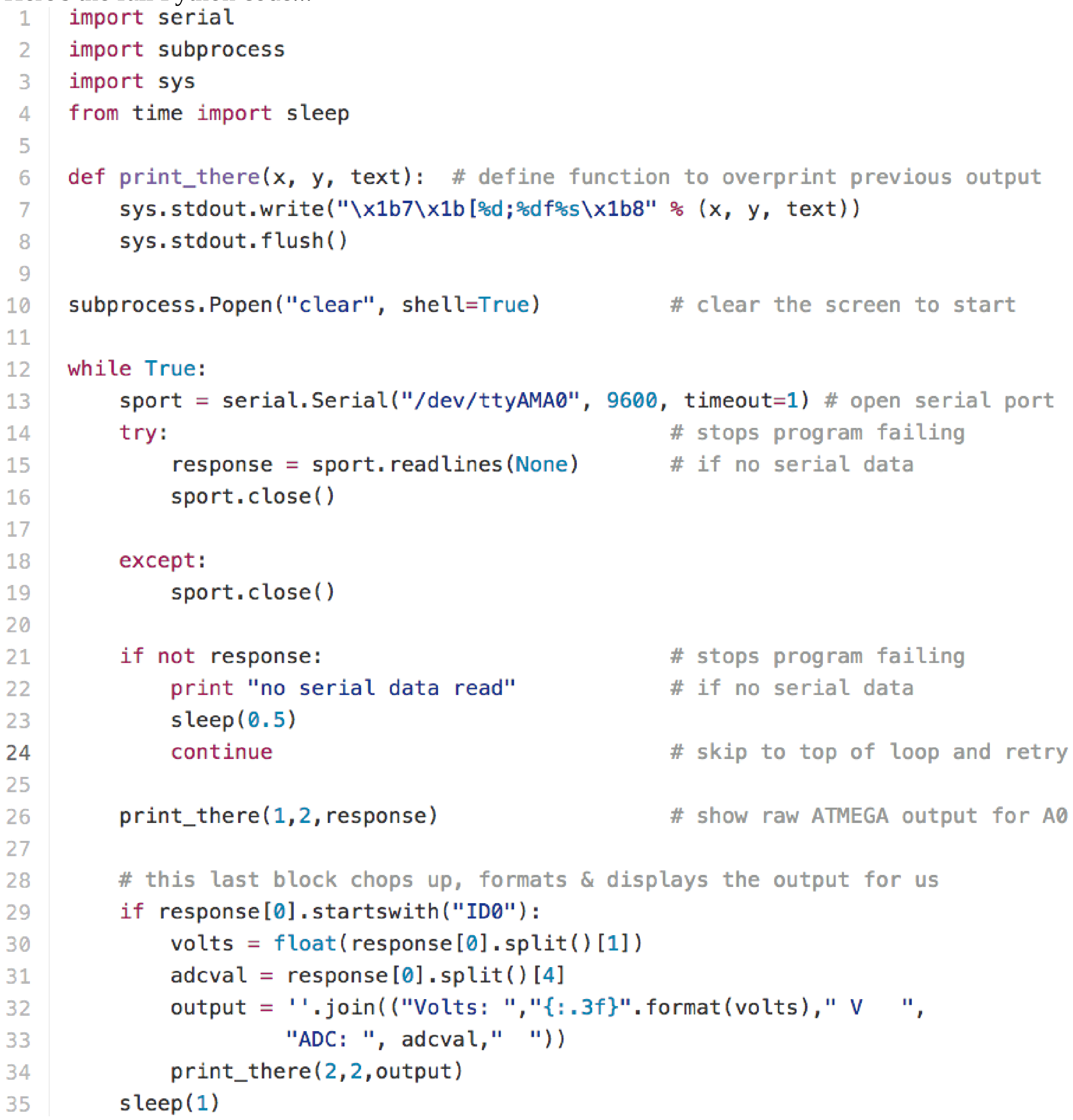
If you want to be able to do anything with your serial data, you need to be able to capture it in a program. For example if you wanted to read temperature and pressure sensors and log the data, you'd need to be able to read that data, rather than just displaying it in Minicom.

The following Python script `duino-testserial.py` reads the Pi's serial port and does two things with the data. It...

1. Displays the raw serial output from the RasPiO Duino in line 1 on the screen
2. Reads in the data, processes and formats it into something more user-friendly, and then displays it on the second line of on-screen output

Instead of scrolling the output, like minicom does, we've made the Python script “overwrite” each new reading on top of the previous one.

Here's the full Python code...



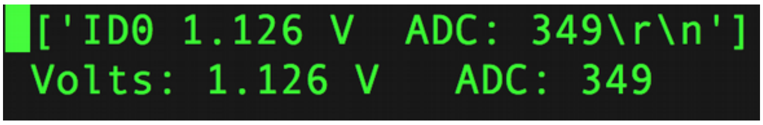
```python
import serial
import subprocess
import sys
from time import sleep

def print_there(x, y, text):  # define function to overprint previous output
    sys.stdout.write("\x1b7\x1b[%d;%df%s\x1b8" % (x, y, text))
    sys.stdout.flush()

subprocess.Popen("clear", shell=True)          # clear the screen to start

while True:
    sport = serial.Serial("/dev/ttyAMA0", 9600, timeout=1) # open serial port
    try:                                       # stops program failing
        response = sport.readlines(None)       # if no serial data
        sport.close()

    except:
        sport.close()

    if not response:                           # stops program failing
        print "no serial data read"            # if no serial data
        sleep(0.5)
        continue                               # skip to top of loop and retry

    print_there(1,2,response)                  # show raw ATMEGA output for A0

    # this last block chops up, formats & displays the output for us
    if response[0].startswith("ID0"):
        volts = float(response[0].split()[1])
        adcval = response[0].split()[4]
        output = ''.join(("Volts: ","{:.3f}".format(volts)," V   ",
                 "ADC: ", adcval,"  "))
        print_there(2,2,output)
    sleep(1)
```

You can download it from here...
https://github.com/raspitv/raspio_duino/blob/master/duino-testserial.py

You can run it from the command line by typing...

python duino-testserial.py

The output should look something like this.



```
['ID0 1.126 V  ADC: 349\r\n']
 Volts: 1.126 V   ADC: 349
```

Output from duino-testserial.py Python script

Code Walk-through of the Python.

Lines 1-4 imports various libraries we need to use in our code. Pyserial is serial, subprocess is used when we clear the screen, sys is used when we overwrite our screen output, and sleep is used for time delays.

Lines 6-8 define a function called `print_there()` which we use to position our on-screen output exactly where we want it.

Line 10 clears the screen in the same way as if you typed “clear” on the command line.

Line 12 starts our main loop, which goes on forever.

Line 13 opens the serial port at 9600 baud and with a timeout of 1 second

```
sport =serial.Serial("/dev/ttyAMA0", 9600, timeout=1)
```

Lines 14-19 enable us to handle the situation where there is no readable serial port data available. The use of a `try: except:` block allows us to handle any failure to read the serial port without making the program crash out with an “exception”.

Line 15 tries to read the serial port and put whatever it reads into a list variable called `response.` If it succeeds, line 16 is executed, the serial port is closed and lines 26 onwards are run. If it fails, line 19 is run. This also closes the serial port, but in the situation where nothing was read from it.

In line 21 we test to see if the variable `response` contains any data. If it doesn't, we tell the users “no serial data read” and wait half a second before `continue` sends us back up to line 12 to try again.

If data was successfully read from the serial port, line 26 prints the raw serial output (as a Python list variable, hence the ["])

Then in lines 29-34 we're identifying and splitting out the data so that we can do something with it. In fact we're not doing a lot with it apart from displaying it in the way we want, but this section should show you how to extract the parts of the data that you want to use and make them into the data types you need them to be. All serial input data is a string by default. In order to do calculations using, say, the voltage, we need to convert it into a float variable (line 30).

Line 29 takes the first line of data `response[0]` that we read from the serial port and tests if it begins with “ID0”. If it doesn't, we ignore it. If it does, lines 30-34 are executed.

Line 30 takes the second item `[1]` from the first line of serial data `response[0]` and converts it into a float variable called `volts.` The `split()` function splits a string up into items delimited by spaces (by default). So ID0 is index [0], 1.132 is [1], V is [2] etc.

```
volts =float(response[0].split()[1])
```

It can be a bit hard to get your head round it, so here's what each part represents...

```
['ID0 1.132 V  ADC: 351\r\n']          response
ID0 1.132 V  ADC: 351\r\n              response[0]
1.132                                  response[0].split()[1]
```

Line 31 extracts the ADC value 351 using `response[0].split()[4]` and assigns it to variable `adcval`. If we wanted to do any maths with it, we'd need to make it an integer. As it stands, it's a string.

Lines 32-33 build up and format our `output` string in the way we want it to be displayed using the `join()` function. We're also using the `format()` function to ensure that our floating point `volts` are displayed to the required number of decimal places (3).

Line 35 gives us a 1 second pause before we repeat the whole loop.

Have Fun!

Hopefully this booklet has given you a flavour of the basic functionality of the RasPiO Duino. There are plenty of more advanced elements of Arduino programming that have not been touched upon here. I hope to be able to add some more advanced examples as time progresses. I can heartily recommend the arduino.cc website. There is a full programming reference there...

http://www.arduino.cc/en/Reference/HomePage

The RasPiO Duino Github repository containing code for all of the sketches in this booklet can be found here...

https://github.com/raspitv/raspio_duino

The RasPiO Duino model used in this booklet was produced by the guys at Fritzing. We were pleased to be able to support Fritzing in this way. If you have Fritzing 0.9.2b or newer, the RasPiO Duino model is included as part of the software. This is great because it means that you can share your circuits and sketches with other people really easily. If you haven't used Fritzing before, I highly recommend it. It's an excellent free and open source software package for creating and sharing circuit diagrams and PCB designs...

http://fritzing.org/

If you've just downloaded this booklet to check the RasPiO Duino, you can get hold of the hardware kit here...

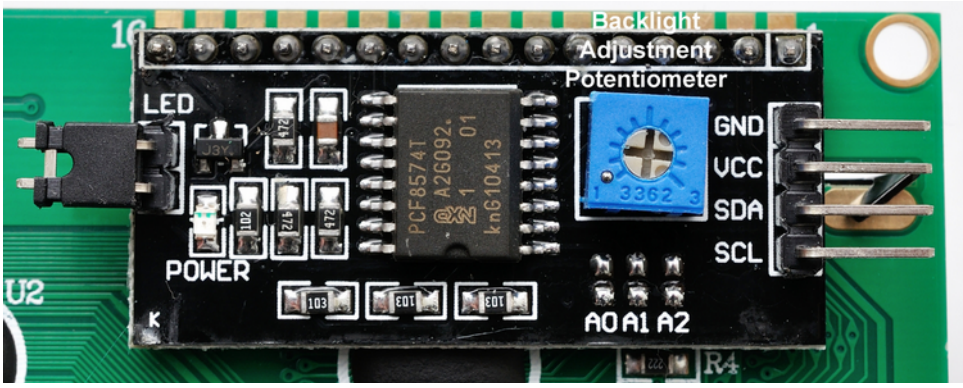
http://rasp.io/duino

RasPiO Duino is also available from ThePihut, Pimoroni, Ryanteck, Pi-Supply and CPC.

Using a 20x4 i^{2}c Character LCD with RasPiO Duino

i2c LCD 20x4 characters

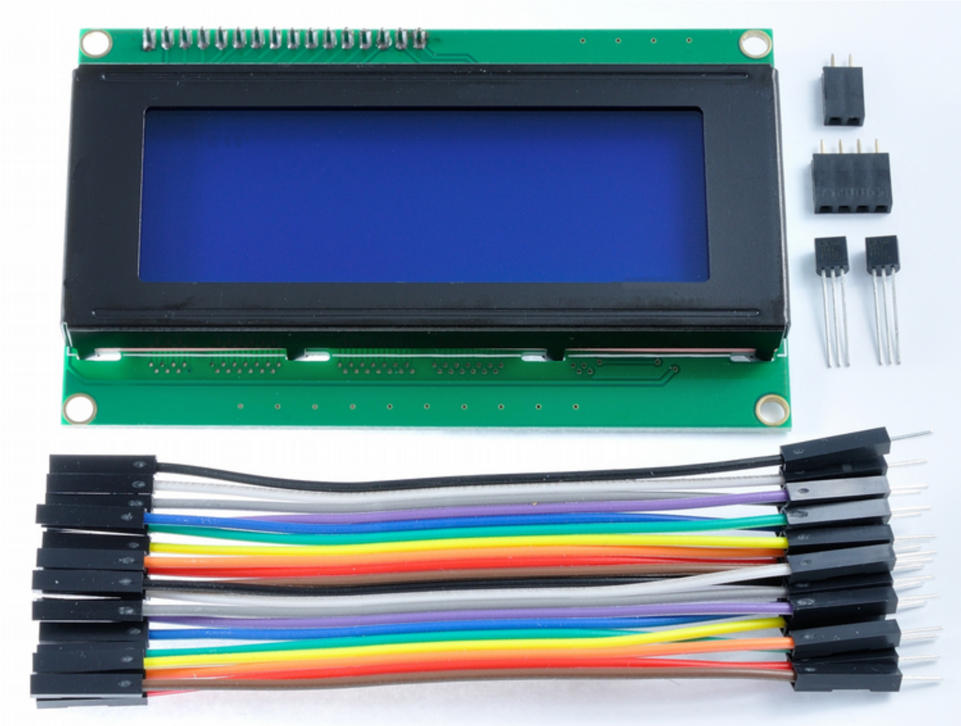
Character LCDs are a great way of showing output from the RasPiO Duino. You can get both bare and i^{2}c displays. i^{2}c displays only require 4 wires, using 2 ports on the Duino. Whereas the bare displays require 11 or more wires to function, using up most of the I/O ports. The displays are the same, but the i^{2}c ones have a little “backpack” that handles the conversion between the Hitachi HD44780 interface and i^{2}c for you. The jumper enables/disables the backlight and the blue potentiometer is for backlight contrast adjustment.



i2c backpack on LCD

The RasPiO LCD20x4 kit has been put together with the RasPiO Duino in mind, although it will work directly from the Raspberry Pi's i^{2}c ports as well. You can find the kit here...

http://rasp.io/lcd20



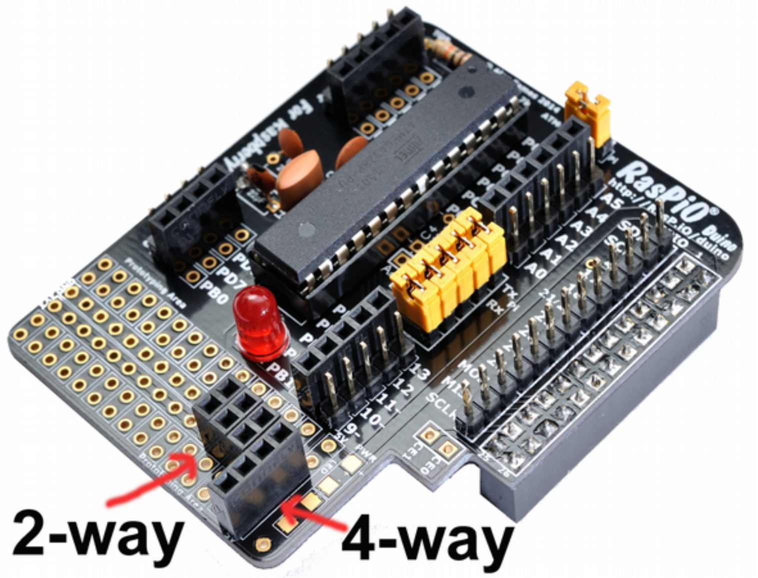
RasPiO LCD20x4 kit contents

The kit contains...

- a lovely 20x4 LCD with blue backlight and pre-soldered i^{2}c backpack
- 2 x TMP-36 analog temperature sensors
- 1 two-way female header
- 1 four-way female header
- 20 M-F jumper wires (10cm)

Soldering the Headers

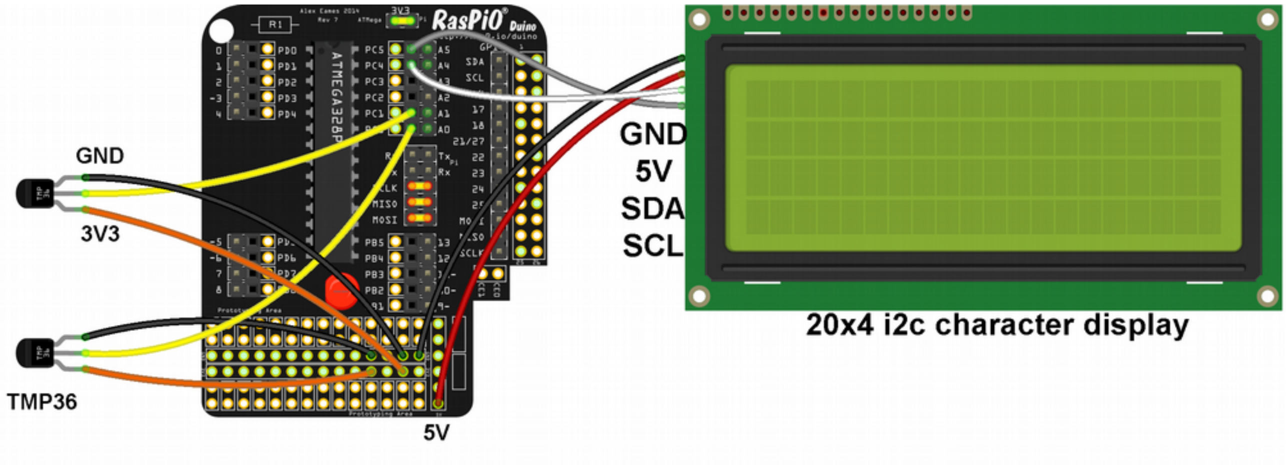
You will probably need to add some extra headers on 5V, 3V3 & GND rails. There is a female 4-way header provided for use on the 5V rail. The additional 2-way female header is to add an extra socket on each of the 3V3 and GND rails. This is what the RasPiO Duino should look like once they're added...



RasPiO Duino with extra headers soldered on

Wiring the Duino to the Display

A5/PC5 on duino	to SCL on display (grey wire)
A4/PC4 on duino	to SDA on display (white wire)
5V on duino	to VCC on display (red wire)
GND on duino	to GND on display (black wire)

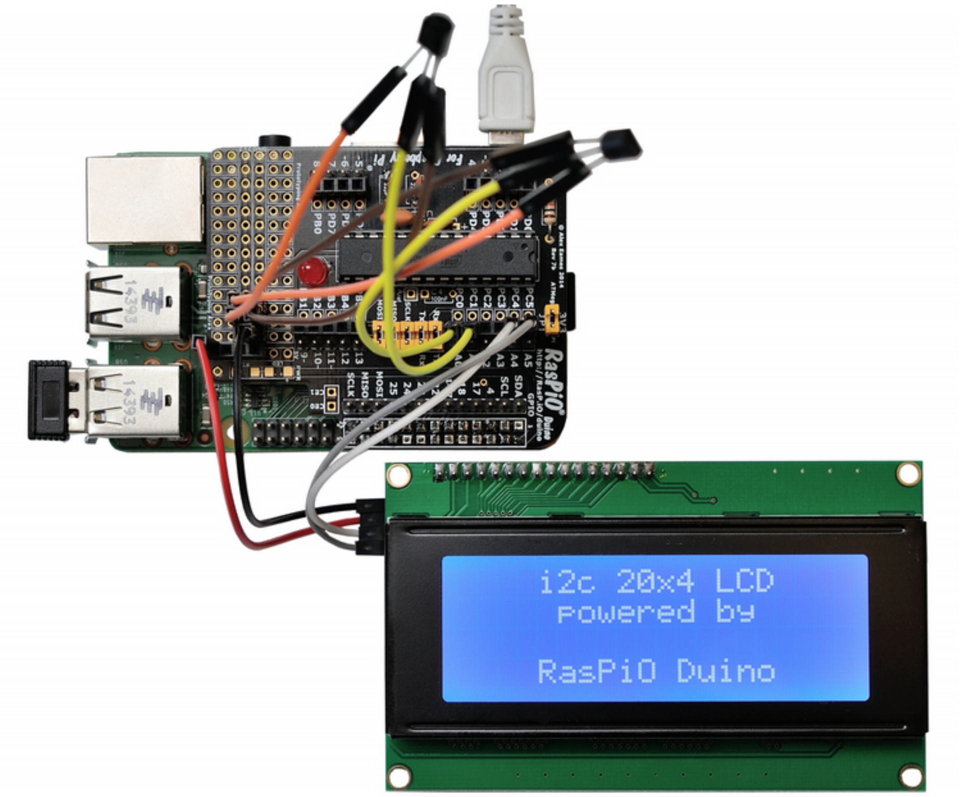


Wiring a 20x4 i2c display and TMP36 sensors

Wiring the Temperature Sensors to the Duino

Pin 1 to Duino 3V3 (orange wires)
Pin 2 to Duino A0 or A1 analog port (yellow wires)
Pin 3 to Duino GND (black wires)

This is what it looks like all wired up using the kit components...



RasPiO LCD20x4 in use

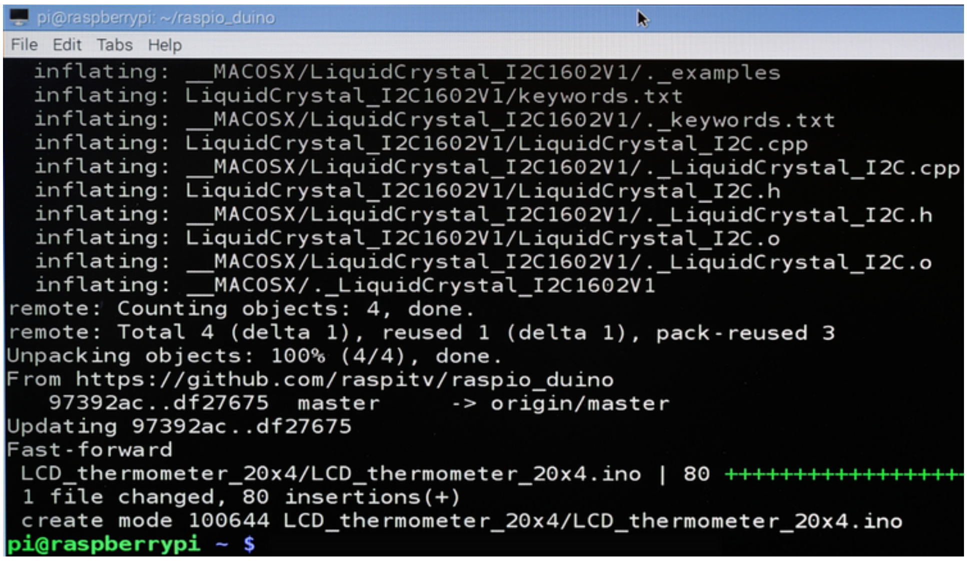
Installing the Software

We're using an open source library called `LiquidCrystal_I2C1602V1` to drive the i2c LCD, so we'll install that next. From the command line, type...

```
curl rasp.io/duinolcd.sh | bash
```

If you want to check out the script before you run it, you can find it here...
https://github.com/raspitv/raspio_duino/blob/master/duinolcd.sh

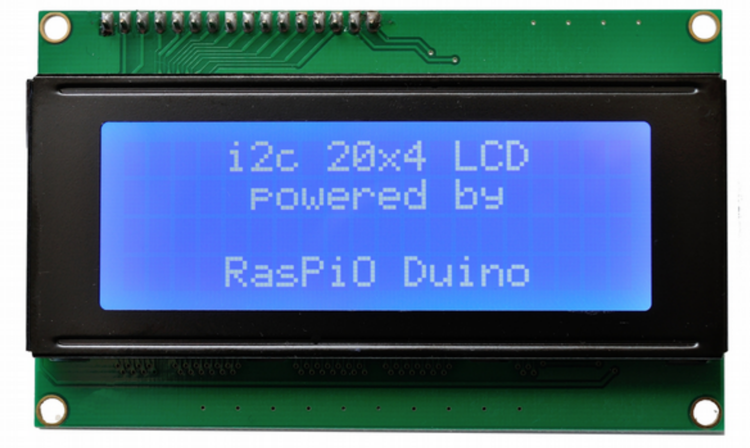
The script will download a zip file, unzip it and place the `LiquidCrystal_I2C1602V1` library files in `/usr/share/arduino/libraries`
Then it refreshes your `/home/pi/raspio_duino` directory to ensure it has the latest files. Then the example sketch `LCD_thermometer_20x4.ino` is placed in `/home/pi/sketchbook` and the script cleans up after itself by deleting all the temporary files it created. When run, your on-screen output should look something like this...



```
   inflating: __MACOSX/LiquidCrystal_I2C1602V1/._examples
   inflating: LiquidCrystal_I2C1602V1/keywords.txt
   inflating: __MACOSX/LiquidCrystal_I2C1602V1/._keywords.txt
   inflating: LiquidCrystal_I2C1602V1/LiquidCrystal_I2C.cpp
   inflating: __MACOSX/LiquidCrystal_I2C1602V1/._LiquidCrystal_I2C.cpp
   inflating: LiquidCrystal_I2C1602V1/LiquidCrystal_I2C.h
   inflating: __MACOSX/LiquidCrystal_I2C1602V1/._LiquidCrystal_I2C.h
   inflating: LiquidCrystal_I2C1602V1/LiquidCrystal_I2C.o
   inflating: __MACOSX/LiquidCrystal_I2C1602V1/._LiquidCrystal_I2C.o
   inflating: __MACOSX/._LiquidCrystal_I2C1602V1
remote: Counting objects: 4, done.
remote: Total 4 (delta 1), reused 1 (delta 1), pack-reused 3
Unpacking objects: 100% (4/4), done.
From https://github.com/raspitv/raspio_duino
   97392ac..df27675  master     -> origin/master
Updating 97392ac..df27675
Fast-forward
 LCD_thermometer_20x4/LCD_thermometer_20x4.ino | 80 +++++++++++++++++
 1 file changed, 80 insertions(+)
 create mode 100644 LCD_thermometer_20x4/LCD_thermometer_20x4.ino
pi@raspberrypi ~ $
```

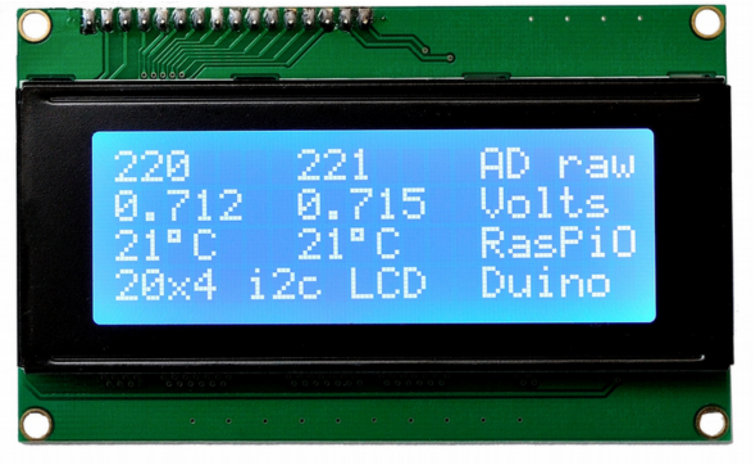
On-screen output during software installation

Once you've wired your display, you can upload the sketch `LCD_thermometer_20x4.ino` to the RasPiO Duino (remember to Upload Using Programmer) and it should give you 5 seconds of this startup screen...



```
    i2c 20x4 LCD
     powered by

    RasPiO Duino
```

...followed by a display of the output from both temperature sensors...



```
220    221    AD raw
0.712  0.715  Volts
21°C   21°C   RasPiO
20x4 i2c LCD  Duino
```

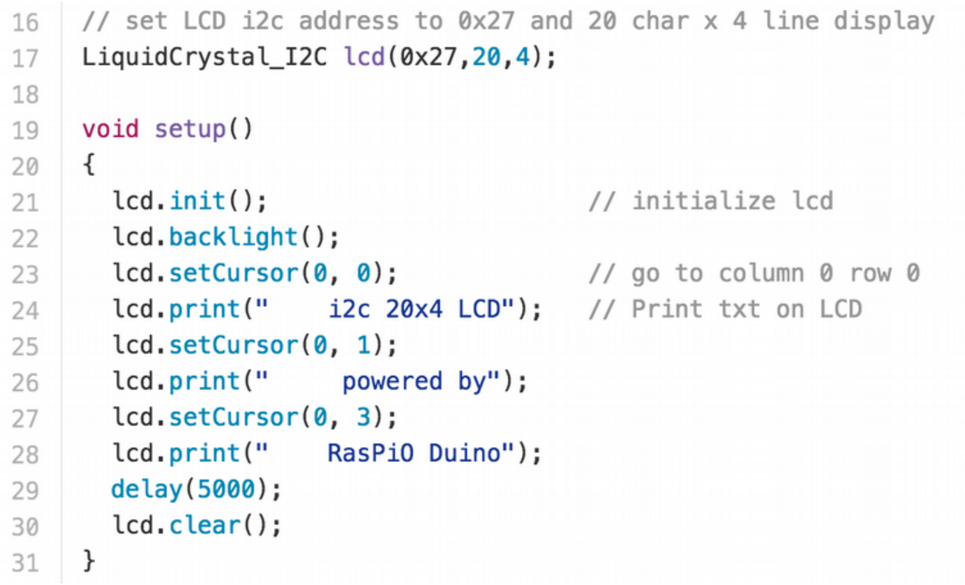
Thermometer sketch output

`LCD_thermometer_20x4.ino` Sketch Walk-through

This sketch is quite long, but most of it deals with what is being 'printed' on the LCD screen. The full sketch is found here. As usual, the first few lines are comment, describing the sketch. After that we have lines 7 & 8...

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

...which ensure that the required libraries are loaded when we compile the sketch. In lines 9-15 we're just declaring variables (nothing new there). In 16 onwards we start doing new things, so let's have a closer look at those parts...



```
16  // set LCD i2c address to 0x27 and 20 char x 4 line display
17  LiquidCrystal_I2C lcd(0x27,20,4);
18
19  void setup()
20  {
21    lcd.init();                        // initialize lcd
22    lcd.backlight();
23    lcd.setCursor(0, 0);               // go to column 0 row 0
24    lcd.print("    i2c 20x4 LCD");     // Print txt on LCD
25    lcd.setCursor(0, 1);
26    lcd.print("     powered by");
27    lcd.setCursor(0, 3);
28    lcd.print("    RasPiO Duino");
29    delay(5000);
30    lcd.clear();
31  }
```

LCD_thermometer_20x4.ino lines 16-31

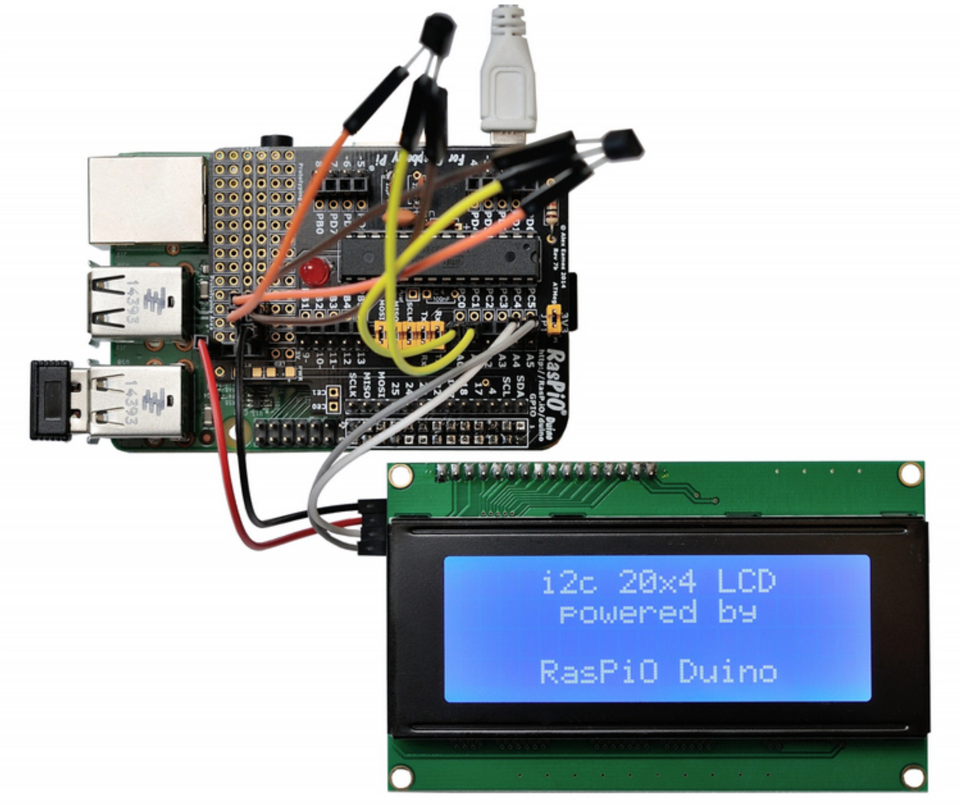
In line 17 we're setting the i^2c address (0x27) of the LCD's i^2c backpack. The 20 and the 4 set the number of columns and rows on the screen.

In 19-31 we write our setup function.
21 `lcd.init();` intialises the LCD.
22 `lcd.backlight();` turns on the backlight (you can turn it off with `lcd.noBacklight();` if you ever need to do that).
23 `lcd.setCursor(0, 0);` sets the current position to column 0, row 0 (top left) of the LCD
24 `lcd.print(" i2c 20x4 LCD");` prints the supplied text to the LCD. You have to be careful that your text will fit the number of characters available if your output is to make sense.
25-28 deal with more of the same – displaying text.
29 `delay(5000);` causes a 5 second wait, so that you can appreciate the full glory of the “Splash screen”.
30 `lcd.clear();` clears the LCD ready for us to start showing our ADC output, voltages and temperatures.

The rest of the sketch uses procedures we've already learned about in the analog section.
In lines 35-56 we're reading the two analog channels 0 and 1, and displaying their raw values on the top row (0) of the LCD.
Lines 58-65 we convert those ADC values into voltages and display those on the second row (1) of the LCD.
Lines 67-75 we calculate temperature values from the measured voltages and display those temperature values on the third row (2) of the LCD.
Lines 77-78 we just print some constant text on the fourth row (3) of the LCD.
Line 79 we wait half a second before starting the main loop again.

So that's a brief introduction to using the RasPiO LCD20x4 kit with RasPiO Duino. I hope you have fun using it...

http://rasp.io/lcd20



RasPiO LCD20x4 running on RasPiO Duino