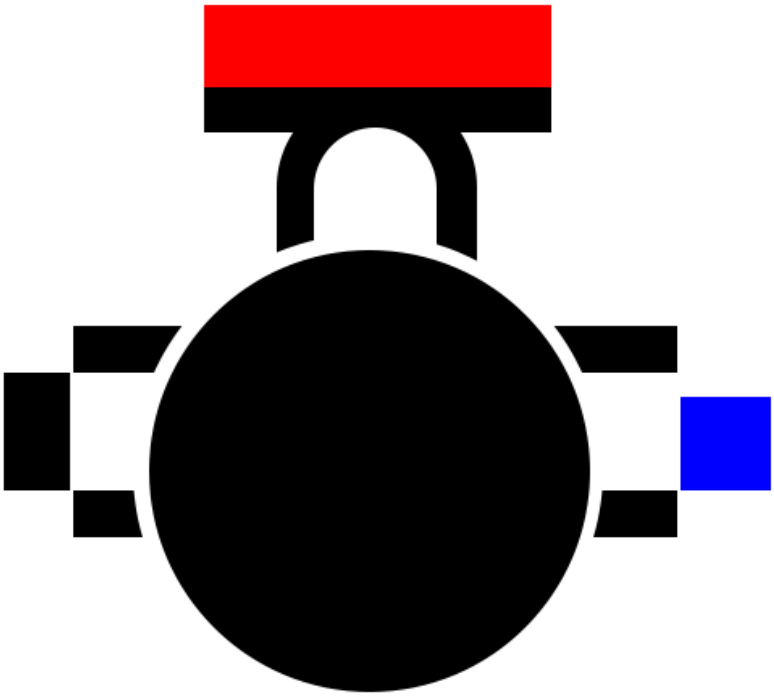
Build a Robot in a Day



Rob Miles

Version 2.1

Table of Contents

Introduction

Welcome to the world of robotics. If you've ever wondered what it takes to build a robot you can find out right here. In this book we are going to build a Hull Pixelbot. This is a little robot from Hull that is designed to be cheap and easy to make, infinitely extensible and great fun to play with.

All the software and the robot designs are free and open source. You can find them here:

```
github.com/HullPixelbot
```

You can find out the latest HullPixelbot news here:

```
www.hullpixelbot.com
```

You can download the sample code here:

```
github.com/HullPixelbot/Learning
```

Why do I need to build a robot?

Actually, you don't. The Hull Pixelbot is great fun to play with, but it won't help you wash the dishes or clean the car. It's only very little and its powers are comparatively weak, so it is no good for world domination either.

But the skills that you are going to get from building the robot will enable you to make some really useful devices. You'll find out how to create programs that can control physical hardware. The Hull Pixelbot is actually a bunch of active components on the end of some software. Just like every other robot that's ever been made.

So, you can take your skills and use them anywhere you like. If you think your life (or perhaps that of your cat) would be improved by a remote controlled, networked cat-flap, then building a Hull Pixelbot will give you the skills to realise this dream.

How much will this cost me?

One of the wonderful things about the world today is just how easy and cheap it is to get powerful hardware. For less than the price of a burger you can pick up a tiny computer that can be connected to the internet and used to read sensors and control devices.

We will be building our robot at "pocket money" prices. The aim is to get you an effective robotic platform for less than the price of a video game. And we will be building the robot in stages, so you can pick up the parts as we go along. At the start of each chapter we'll have a "shopping list" of things you'll need to complete this stage of the robot development.

The motor chassis is designed to be 3D printed. If you have a friend who owns a 3D printer you can give them a chance to show off their printing skills by making you the parts that you need. Printing an entire chassis takes around 8 hours or so of printing time, and it uses around 10 metres of filament, which should cost less than five pounds. If you are having problems finding someone to print your robot, I'd strongly advise you to seek out a local "hackspace". These are clubs run with the aim of sharing hardware and expertise. If you go along with the robot designs they might even print some robots of their own.

What will I learn?

You'll learn how to create programs and run them. You'll also discover how programs can be made to talk to hardware, and the fundamentals of the electronics that makes the hardware work. Specifically, the book will cover:

- Creating and deploying programs written in the C++ language. You'll be deploying programs to Arduino devices. It's not a programming book as such, many of the programs that you are using have already been written, but you will pick up some programming skills as you read the text and make changes to the code. You don't need to be able to program at the start, but you'll understand a lot more about programming at the end.
- Building hardware. This is the fun bit. You'll discover how to fit components together and make them work. Hopefully you'll also gain the confidence to take your own ideas and translate them into hardware.
- Simple electronics. You'll learn the difference between analogue and digital systems and some design principles which will mean you can create devices that work and don't disappear in a puff of smoke when you turn the power on.
- Simple networking. Making a robot is very well, but a device like a Hull Pixelbot only becomes really useful when it is linked to a network. You'll explore how simple networking technologies work and how you can use them to make properly connected devices.

How does the book fit together?

Each chapter will explore a particular feature of the robot. At the beginning of each chapter I'll describe how to construct the hardware, then we'll take a look at the way this hardware is used from the software. Finally, we'll finish with some things you might like to try. During the text I've got some conventions

This indicates an activity you should perform in at this point in the text. You may be given precise instructions, or you may have to work something out for yourself.

This indicates something that you may want to think about later.

This indicates a warning to be careful about this bit.

Is it all really easy?

No. There are some bits that are a bit tricky. When you build an electronic device that is controlled by software you get a "double whammy" of problem potential. When it doesn't work, you have to figure out whether the hardware is faulty or your program has a bug in it. This is difficult even for the most experienced engineers. But during this text you'll learn techniques that will help you deal with these problems.

As you struggle to get your robot to do what you want you'll gain a new respect for the people that take technology like this and package it into your phone, TV or washing machine. But you'll also gain the skills that make it possible for you to think about making the next generation of products like these.

Is there any sample code?

Yes. Within each chapter there'll be references to sample Arduino projects that you can run. You can download the sample code from github:

```
https://github.com/HullPixelbot/Learning
```

What do I need to get started?

If you want to build a robot you'll need some hardware tools and some software tools. These aren't expensive, and they'll serve you in good stead during your robotics career. You'll need hardware tools to build the robot, some software tools to create the program that controls it, and some components to actually fit together. Then you'll need the components to actually build the robot from.

Tools

Figure 1 These are the tools that I've used to make all my robots.

On the left I have my indispensable wire cutters and strippers. These let me cut wire to length and also remove the insulation from the metal core, so I can make connections. Next along you have some "snipe nose plyers". These are good for holding things and squeezing things together (which has the professional name of "crimping".

Next along are a pair of screwdrivers. One has a "slotted" end which is flat. I can use this for tightening screws that have a slot in the top. The next one along has a "Phillips" or "cross-head" tip, which I can use for cross head screws like the ones in the terminal blocks that we are going to be using.

The last two tools I'm rather proud of. They are "2mm hex drivers". I use these to tighten the bolts that hold the HullPixelbot together. You can use ordinary headed bolts (with a slot or a cross head), but I've found that bolts that you can tighten with these drivers are much easier to work with. There's no chance of the tool slipping out of the head of the bolt when you tighten it. When we look at the parts that you buy, I'll tell you where to get these bolts. They aren't much more expensive, and they are a lot easier to work with.

If you look at my toolkit you'll notice that every tool is from a different supplier. I've just picked them up as I've gone along, and replaced ones that I've lost or broken over the years. You don't need to buy an expensive toolkit full of shiny things you'll never use, I've found that you can get along just fine with the above.

It is really nice if you can set up a little place to work on your robot. Assembling robots on the sofa in front of the TV is possible (I've done this) but it is very frustrating when you drop a screw down the side of one of the cushions and then have to spend ages searching for it. If you can find a reasonable area of desk space that is well lit, you'll find it much easier to work there.

Software

The software tools that you want are all free, which is nice. To start with you'll not be writing programs from scratch, instead you'll be modifying ones that I've written. The tool that you will be using is the Arduino IDE. You can download this for free from:

```
https://www.arduino.cc/en/Main/Software
```

There are versions of this software for Windows PC, Mac and Linux computers. You can download and install the software now if you like. We'll investigate how to use it later in the text.

Components

There are two kinds of hardware components you'll need. You'll need the electronic components that power the robot and make it work and you'll also need the robot chassis, which is made up of 3D printed elements.

Robot Chassis

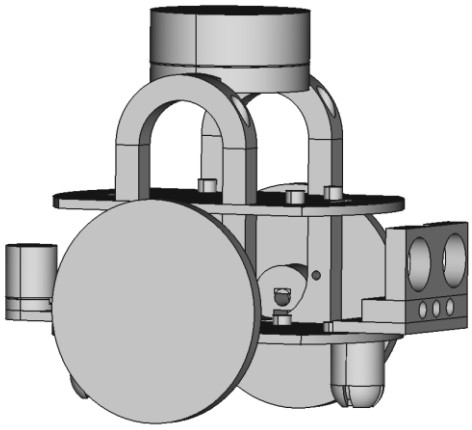


Figure 2 These are the Computer Aided Design (CAD) designs for the Hull Pixelbot

Above you can see the “skeleton” of a complete Hull Pixelbot. It comprises a number of elements that you bolt together. At the start of each chapter I’ll identify the elements that you need to print to complete that phase of the robot development.

Building the robot chassis

What you will do in this chapter

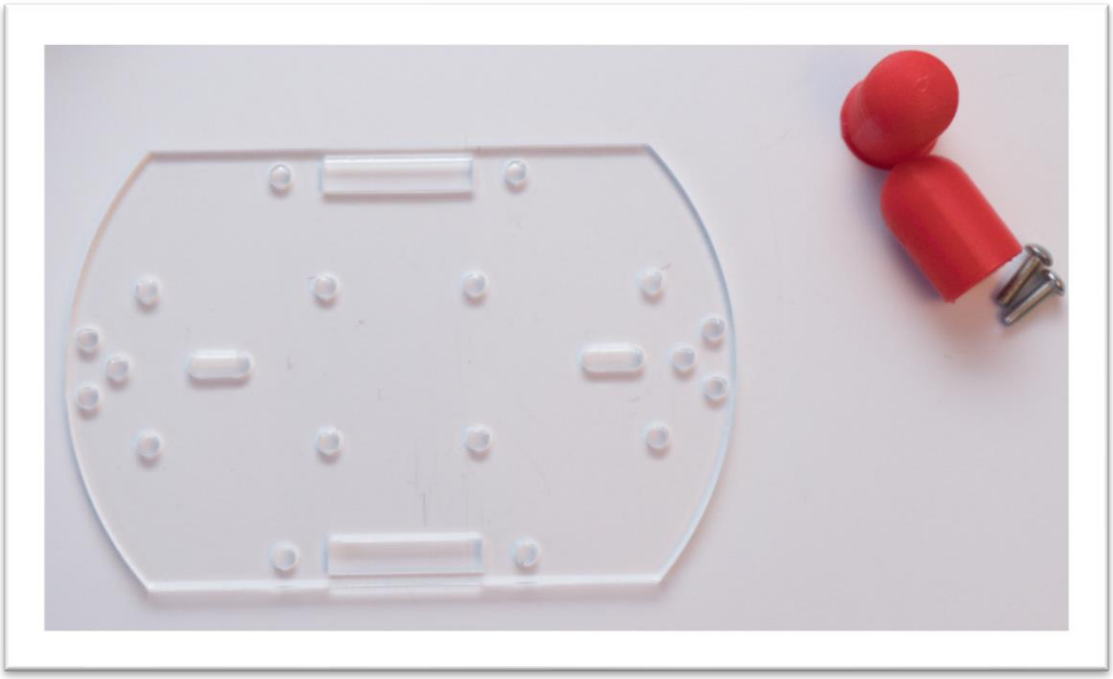
In this chapter you are going to build the moving platform for your robot. You'll fit the battery holder, processor and motors to provide the base for all future work. This is the most hardware intensive part, so be prepared for a bit of frustration as you fit everything together. In my experience, the best way to survive hardware projects is to pace yourself and have regular stops for coffee and refreshment. Actually, that's my approach for pretty much every project, but for hardware I find taking breaks especially important.

Make sure that you follow the steps in order. If you get the sequence wrong you might find that some pieces cannot be attached because of the way that the robot fits together.

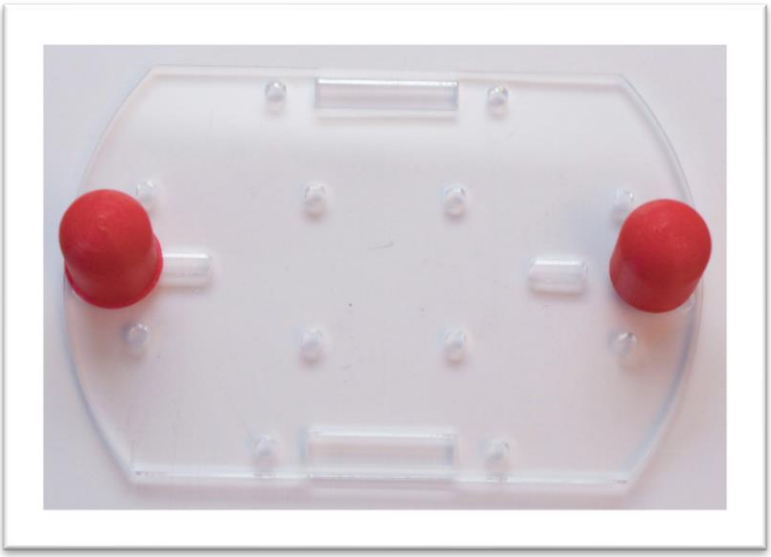
I've provided a handy checklist you can use to check off each item as you add it to the robot. You should print off a copy of this and then tick each step as you perform it. This makes it much easier to take a break, as it's easy to tell how far you've got.

1. Fit the bumpers

We are going to start by attaching the bumpers that the robot rests on. These are fitted at each end of the robot. You'll need the chassis, the bumpers and two 10mm bolts.

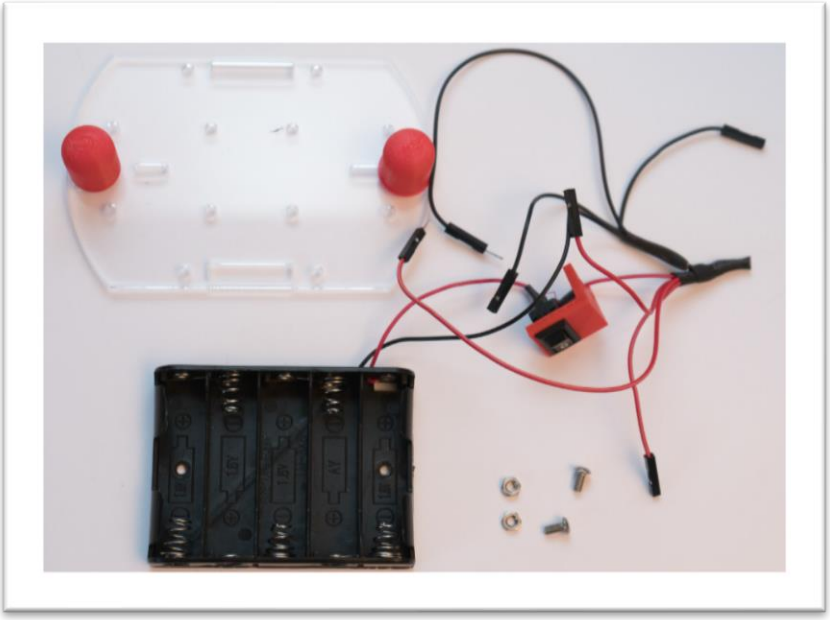


Attach each skid to the base using two 10mm bolts, screwing the bolts into the holes on each skid. Remember that to tighten the bolts you will turn them clockwise.



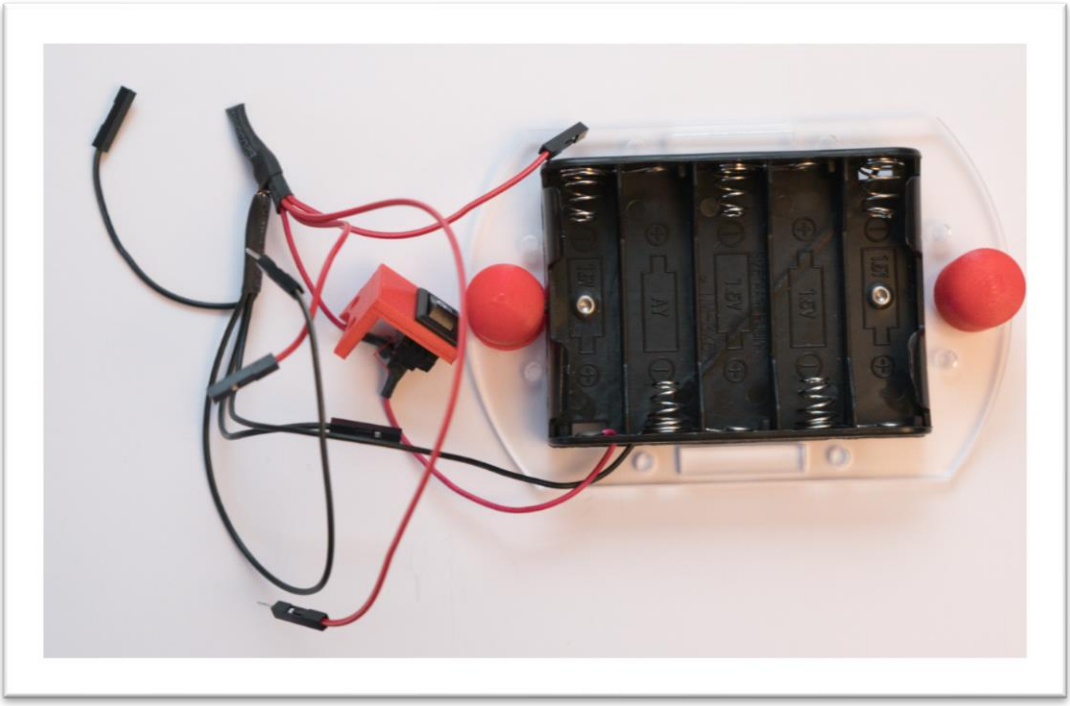
2. Attach the battery holder and cables

The battery holder fits underneath the robot. It's connected to the power switch and provides connections to all the different elements in the robot. It's provided as a complete assembly. We're going to bolt it underneath the robot.



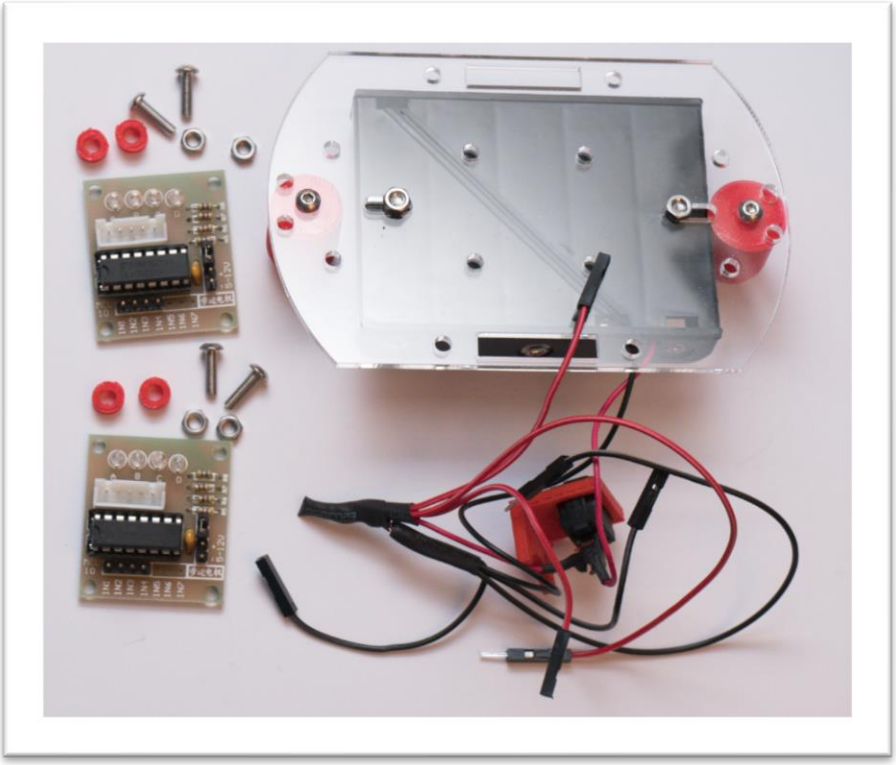
These are the components you'll need. You'll be using two 6mm bolts and nuts to suspend the battery box underneath the robot.

Fit the battery box under the robot. Push the bolt into the hole in the battery box and then through the slot in the chassis. Put the nut on the end of the bot to hold the battery box in place. Then repeat with the other bolt.

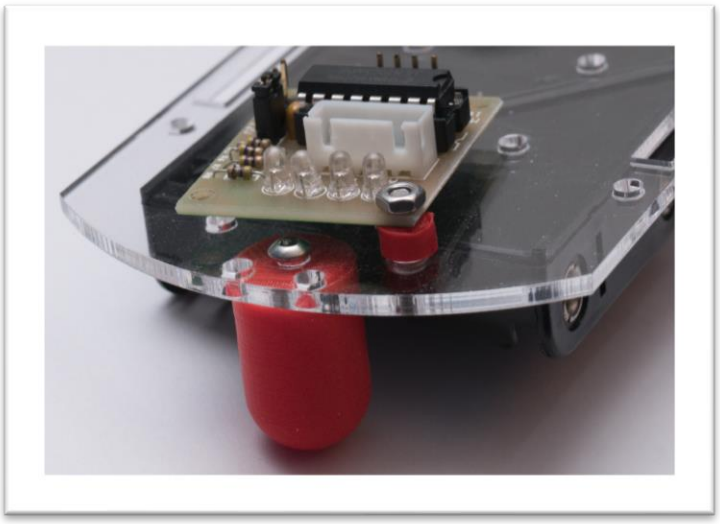


3. Fit the motor driver boards

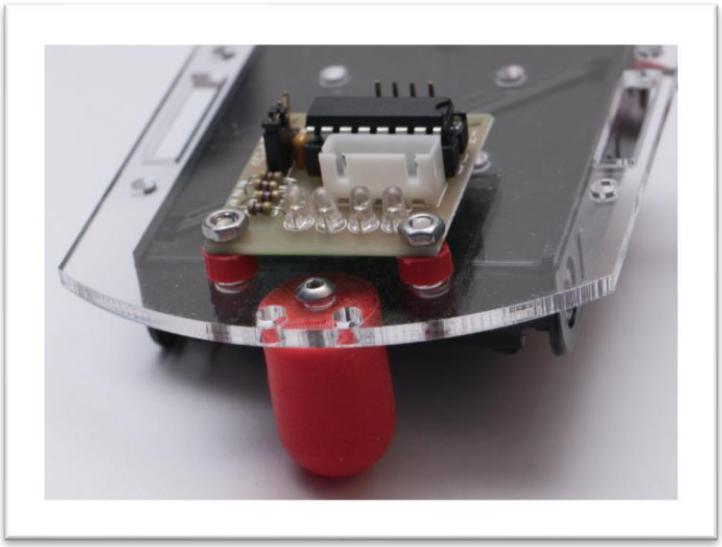
The motor drivers contain amplifiers that take the low-level control inputs from the Arduino computer and output signals strong enough to control the coils in the motors that will drive the robot around. Each board controls the coils in one motor.



You will need to fit a board for each motor. We are only using two bolts for each motor, but this is perfectly strong enough to hold each board in place.

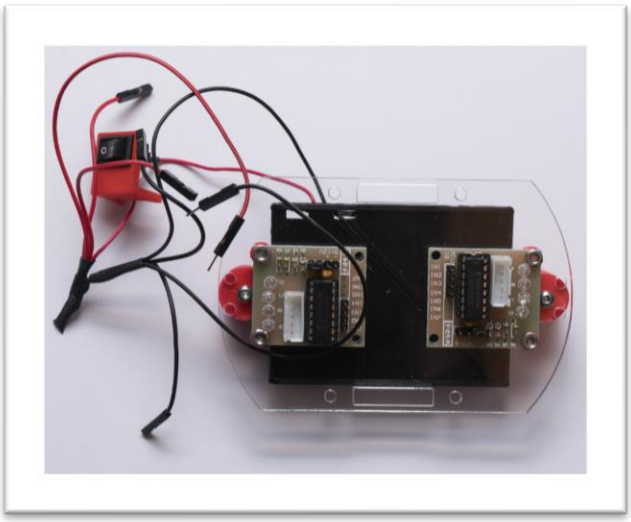


Fit the driver boards. I've found the best way to fit these is to push the bolt up from underneath the bottom plate, put the washer on the bolt, then the circuit board and then attach the nut.



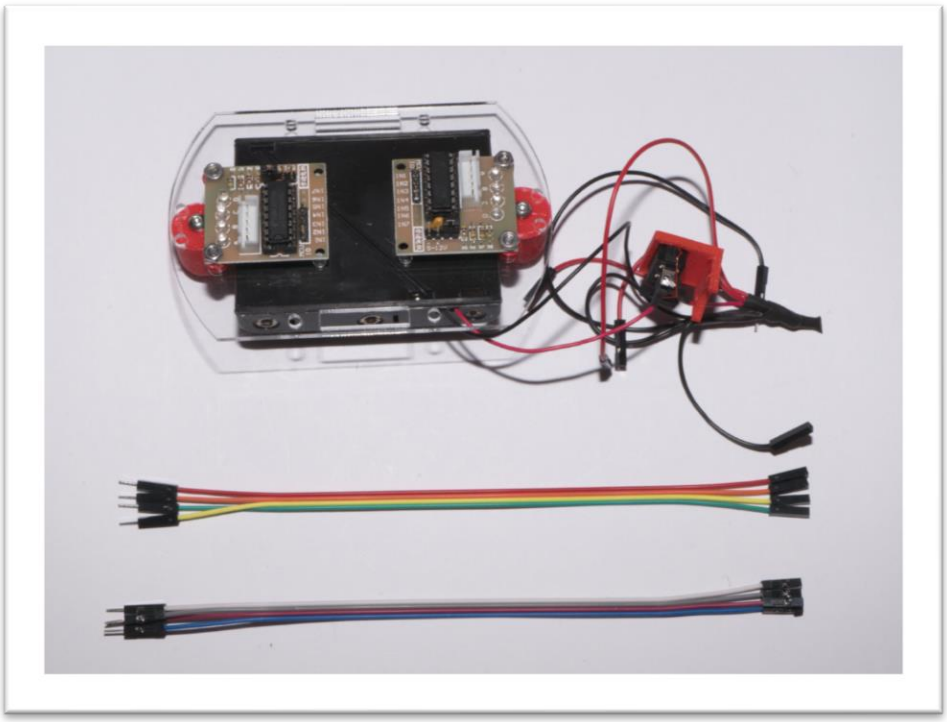
Three things to remember here:

1. Don’t fully tighten up one bolt until you've fitted both. That way you can move the board to one side to fit the washer onto the other bolt.
2. The bottom of the motor driver board tends to have some rather sharp bits sticking out of it. Be careful that you don’t scratch yourself when fitting the board.
3. Each board has a row of lights on one edge (these are the round transparent things). Each light shows when that particular motor drive coil is active. Put the lights towards the outside of the robot, so you can see them flicker when the robot is moving.



4. Attach the driver signal cables

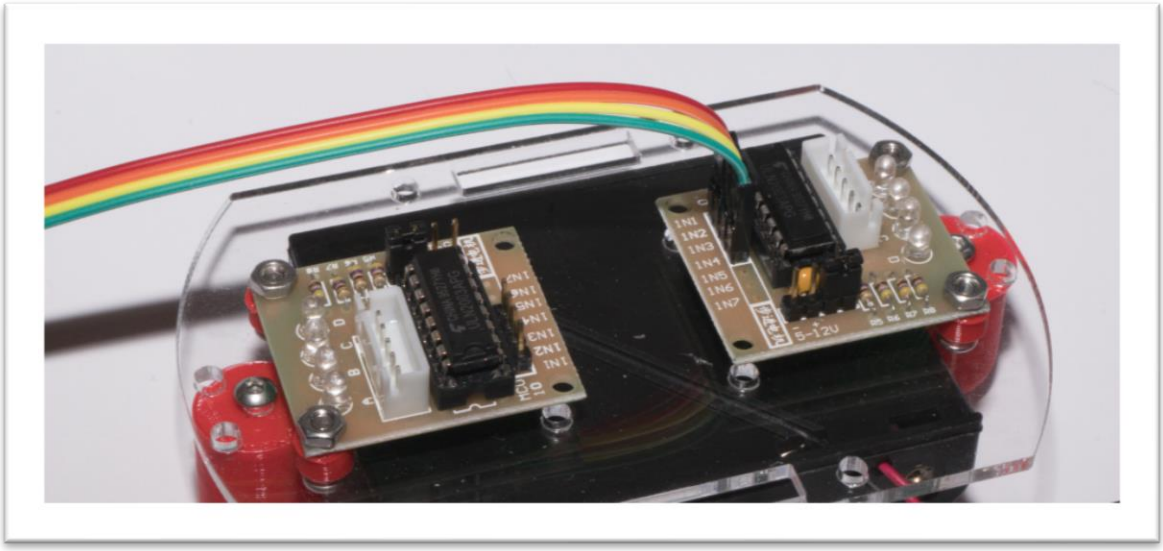
The Arduino computer will send signals to the motor driver board to turn on different coils in the motor and make the robot move. We need to connect cables between the motor control board and the Arduino to send these signals. To do this we will need two cables, each of which can send four signals.



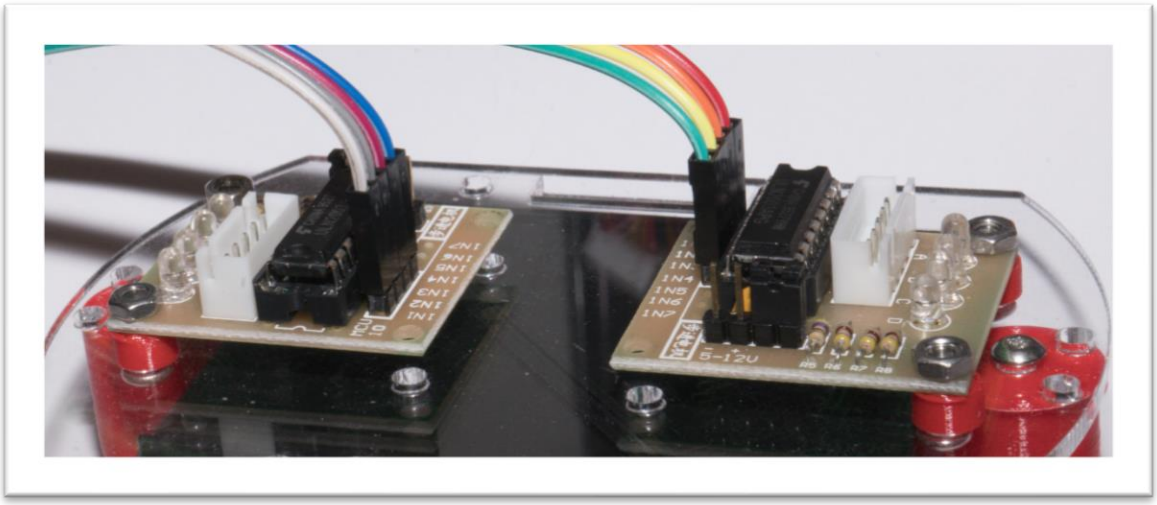
I'm using the colours red, orange, yellow, green for the **right** motor and white, grey, purple, blue for the **left** motor.

Hold your robot so that the wires coming out of the battery box point towards you. (as in the picture above) Then connect the sockets in the right cable (red, orange, yellow, green) to the

IN1, IN2, IN3 and IN4 pins on the *right hand* motor control board as you can see above. Make sure that the colours stay in sequence.



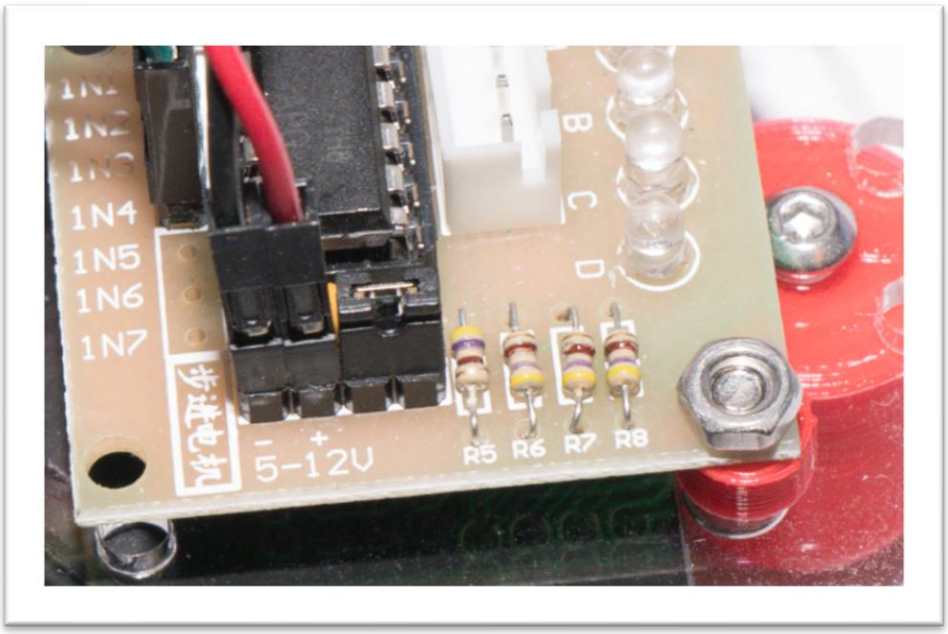
Make sure that there is a jumper (a little black plug that spans two pins) across the other two pins in that bank of four pins on the motor driver board.



Now connect the second cable to the other motor board. Connect white to IN1, grey to IN2, purple to IN3 and blue to IN4.

5. Connect the power to the driver boards

Now we need to connect the wiring harness to the motor driver boards. On each motor driver board there are two pins labelled + and -. These are the power connections for the driver boards. These pins are part of a bank of four pins.

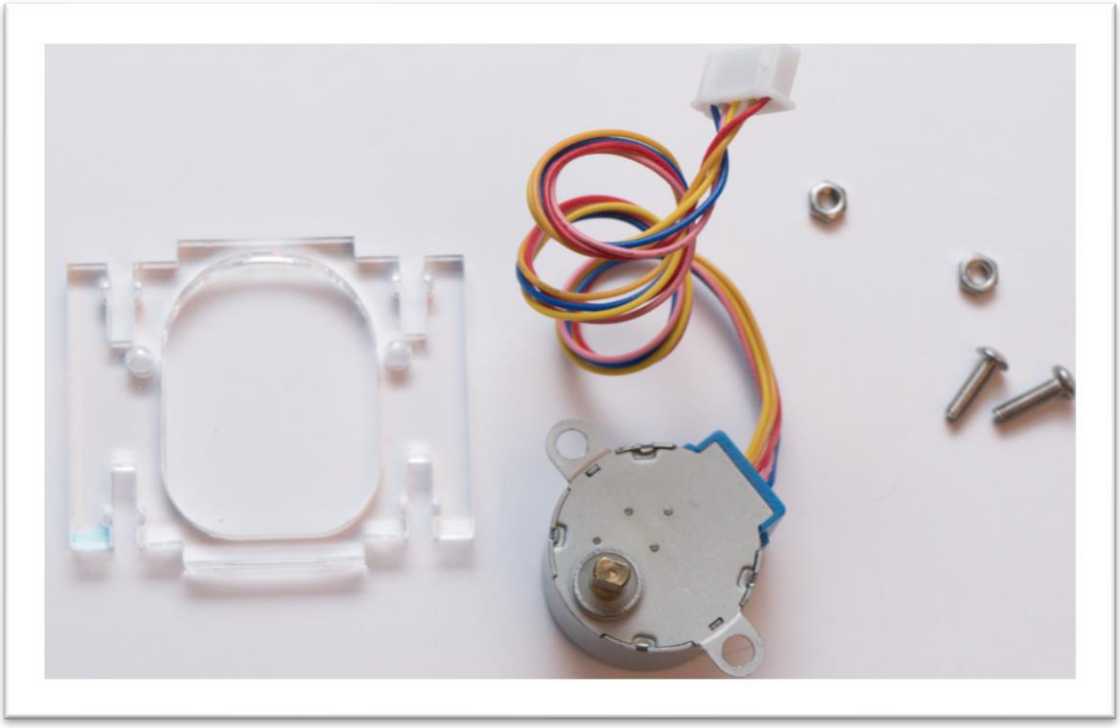


Connect a socket on the end of the short wires to each of the power pins on the motor board. Red wires go to the pins labelled + and black wires go to the pins labelled -.

Do this for both boards.

6. Attach the motors to the sides

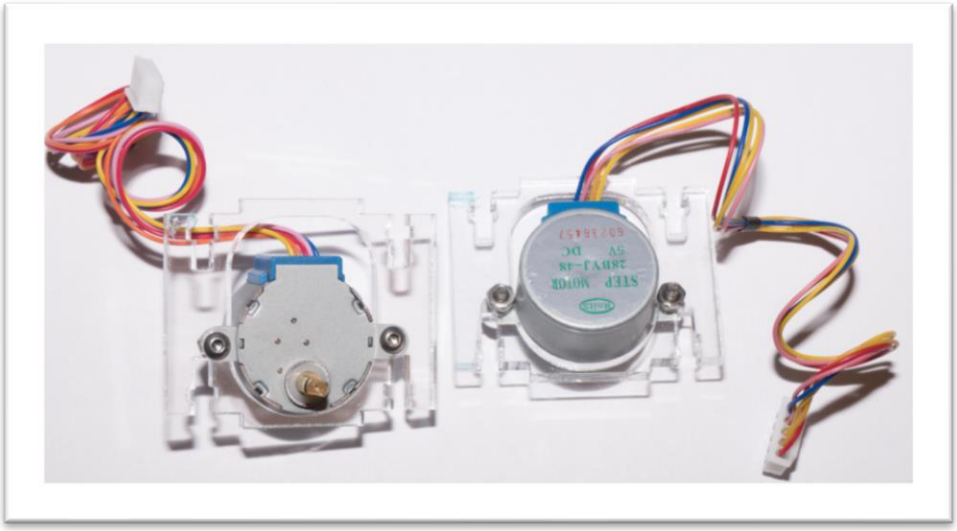
The motors are attached to the sides of the robot using two 10mm bolts and two nuts.



These are the components you'll need. You'll need two sets.

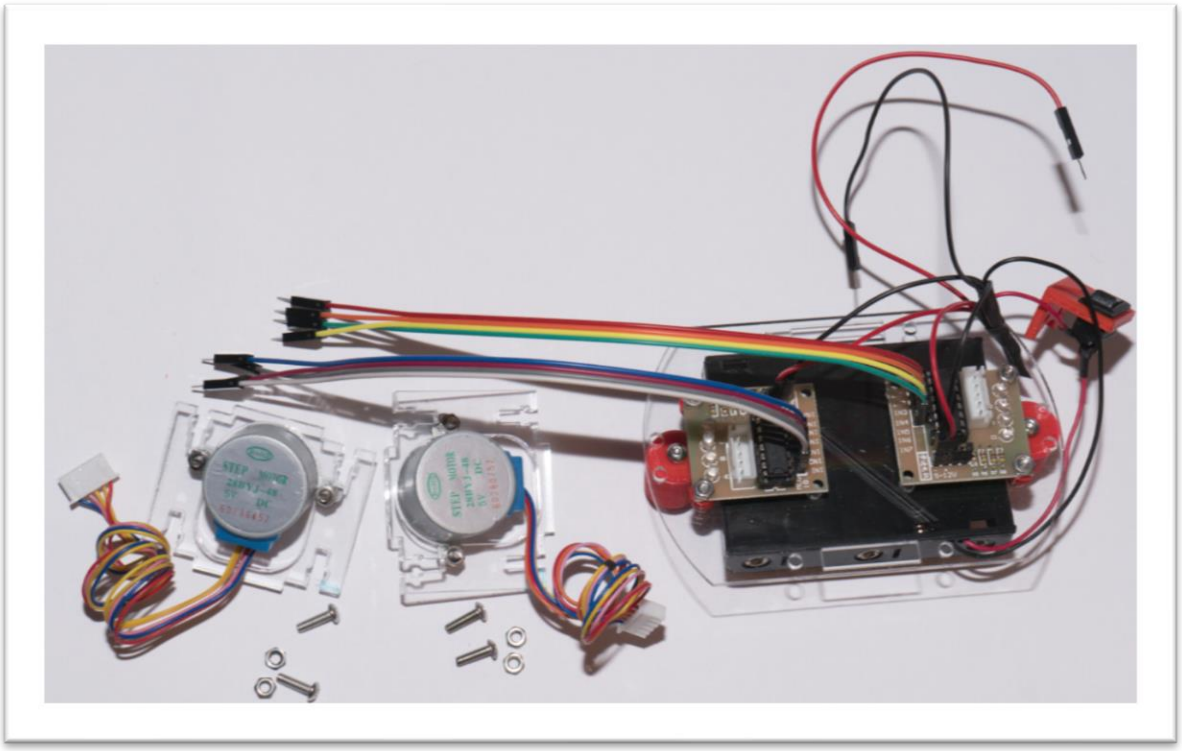
Thread the motor wire through the hole in the robot side and then push a bolt through the hole in the motor and then into a hole in the robot side. The motor only fits one way up, with the curve of the motor fitting into the curve in the robot side. Put a nut on the end of the bolt to hold the motor in place.

Repeat this with the other motor. Then, to ensure that we don't have too many dangling cables, take each motor cable and wind it around your finger and release it, to create a couple of coils.

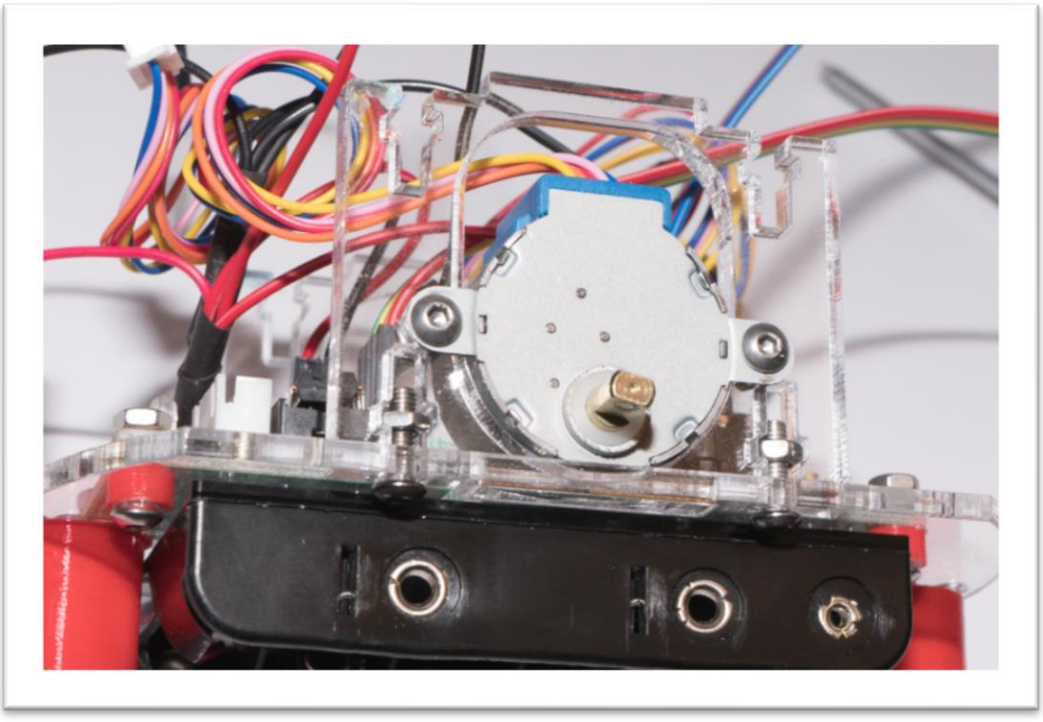


7. Attach the sides to the base

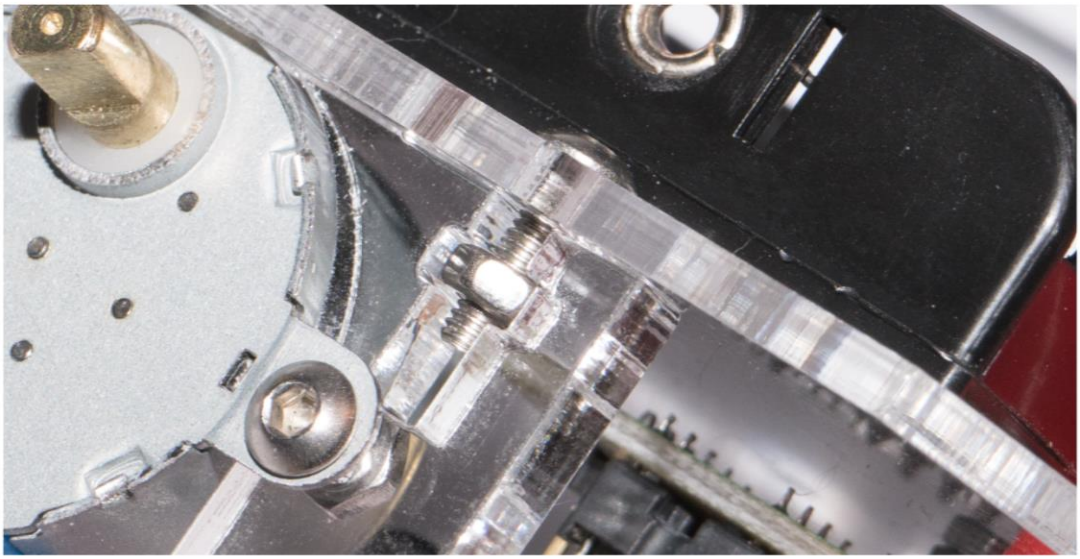
In the next step we'll fit the sides to the robot chassis. For this you'll need the two sides, the base, four 10mm bolts and four nuts.



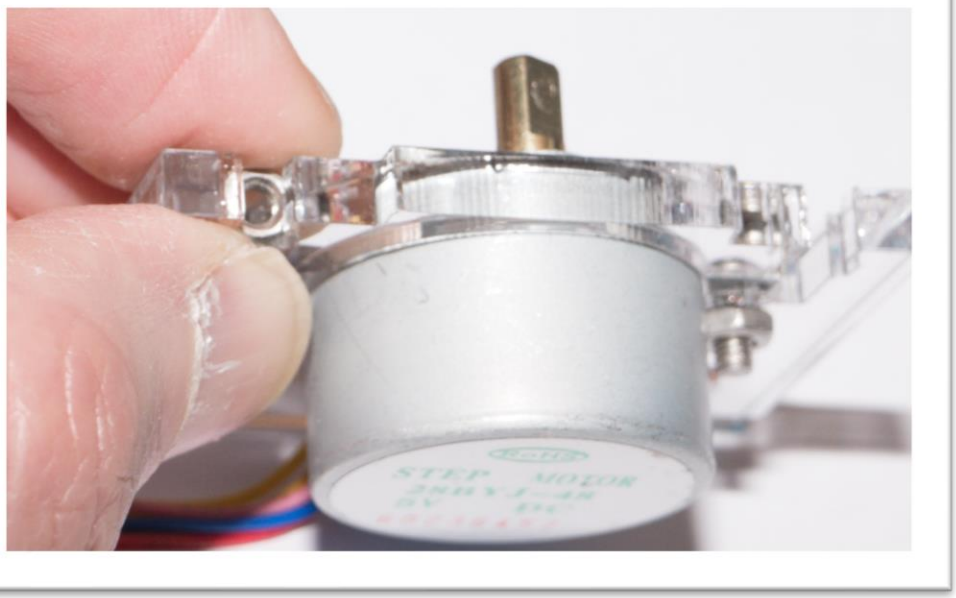
The sides fit into slots in the base and are held in place by bolts which fit into slots in the tower.



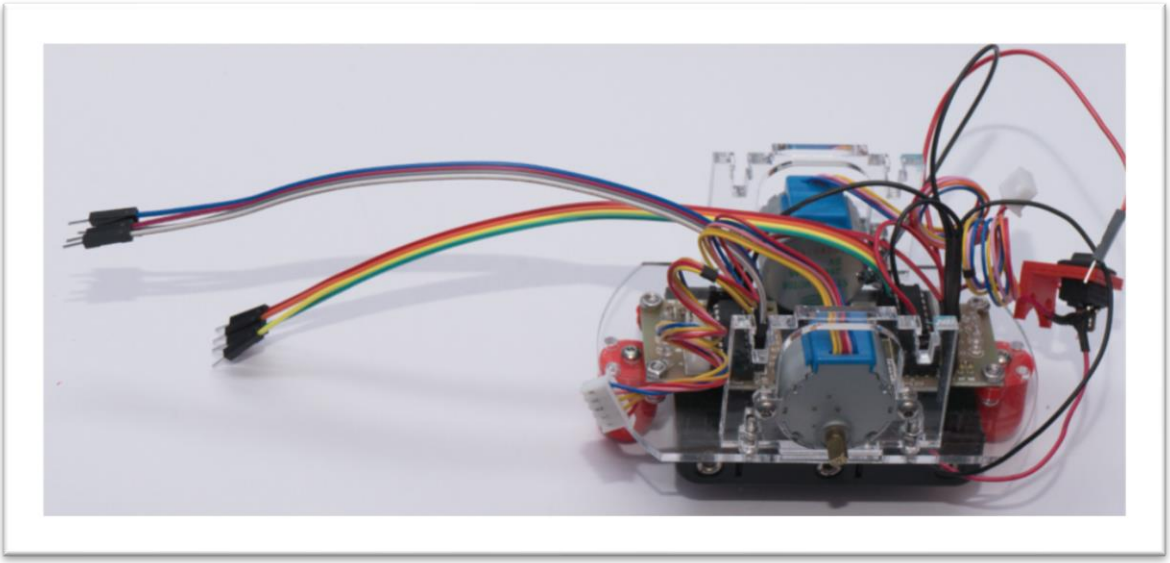
Below you can see how the bolts work.



Attach the two sides to the robot. Make sure that the bronze output shaft from the motor is at the **bottom** of the robot, near the battery holder, as you can see in the pictures above.



This is perhaps the most difficult part of the assembly, so take it slowly. The best way is to start by holding the tower vertical so you can slide the nut into the slot, and then hold this in place with one hand while you line up the base and the bolt, as shown above.

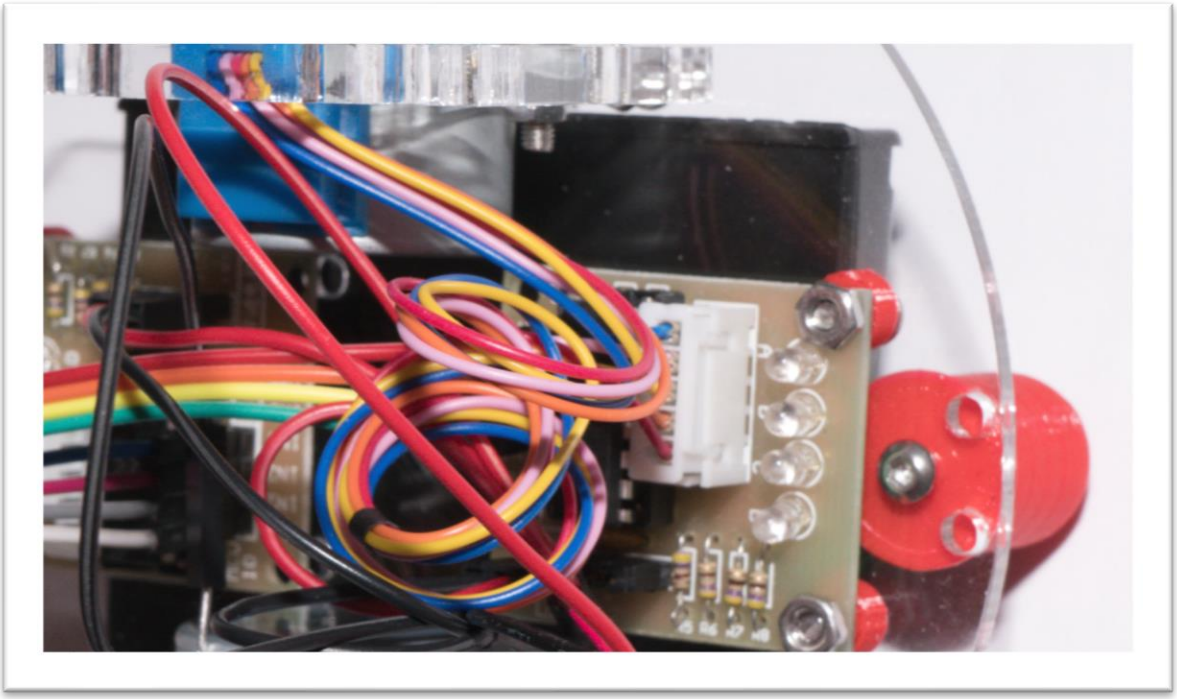


8. Plug the motors into the motor driver boards

Now that you have the motors fitted onto the robot, the next step is to plug them into the driver boards.

Hold the robot chassis so that the battery holder wire is pointing towards you (as in the picture above) and plug the cable from the **top** motor into the **right hand** motor driver board.

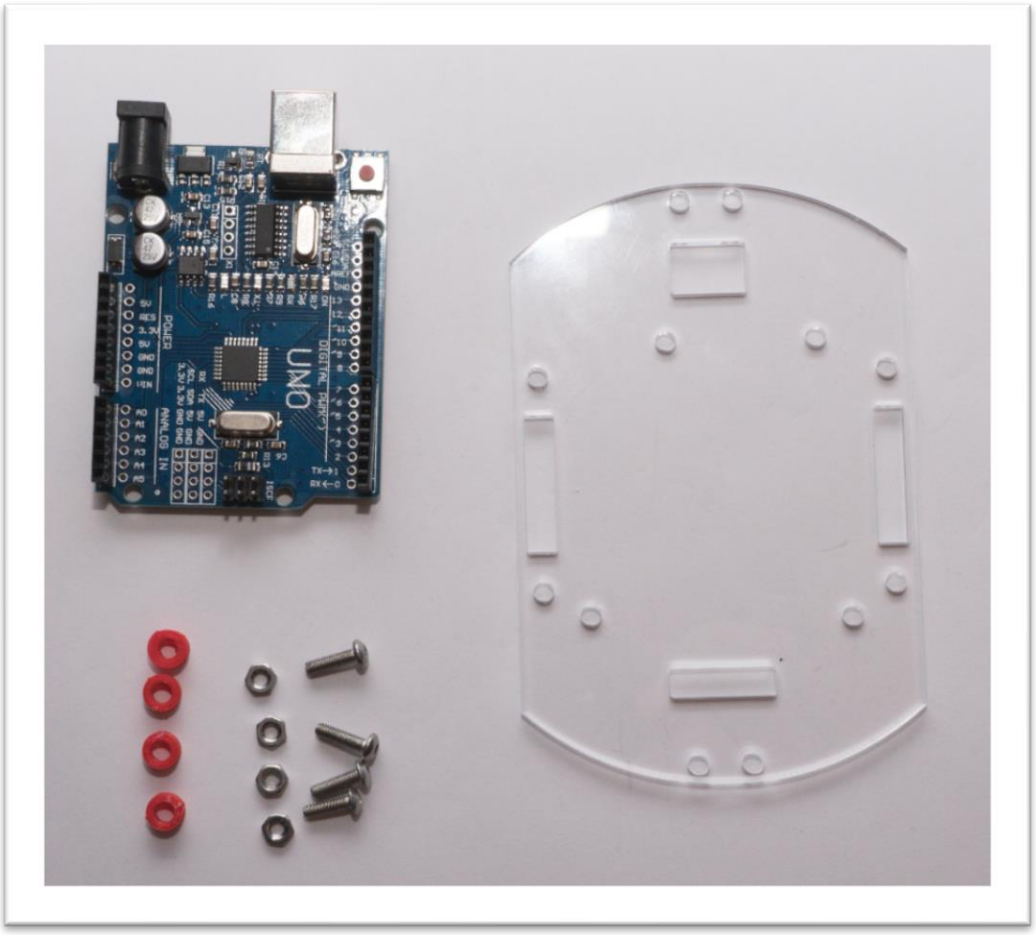
The plug will only go in one-way round. Repeat the with the **bottom** motor, plugging this into the **left** driver board.



Congratulations. That's the base of the chassis complete. Now would be a good time for a break. And now you can start calling yourself a "Robot Engineer".

9. Fit the Arduino Uno to the robot top

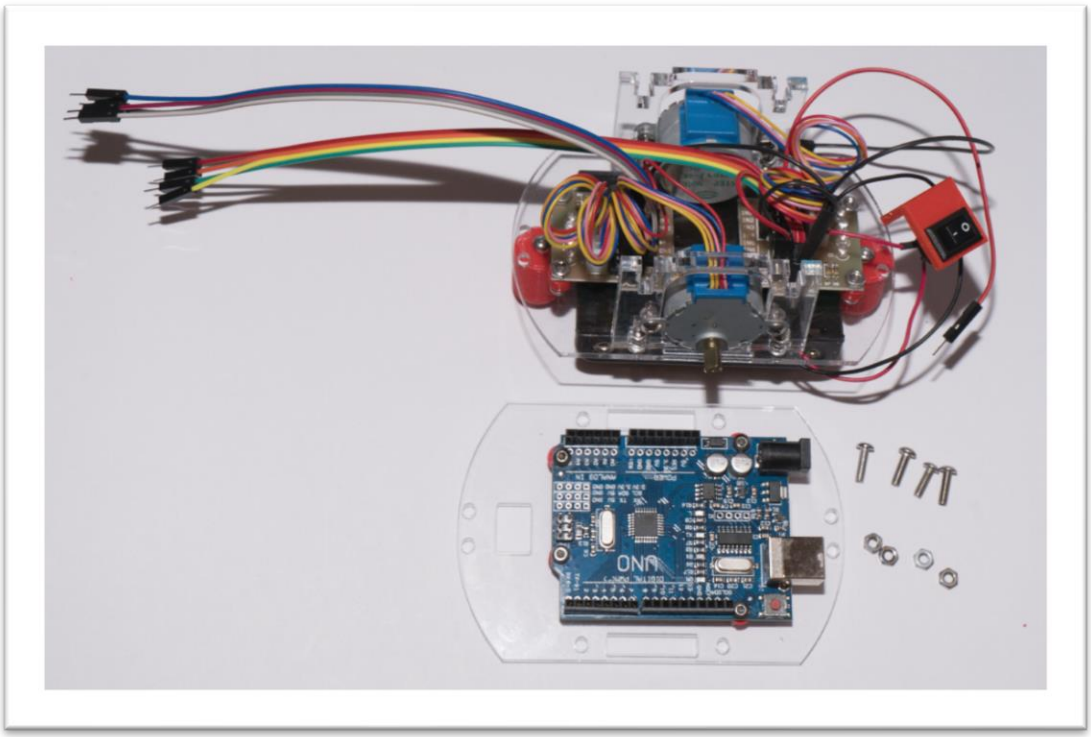
The Arduino Uno computer provides the “brains” of our robot. We attach it to the top of the robot using four bolts. For this step you'll need the Arduino, the four 10mm bolts, four spacers and four nuts.



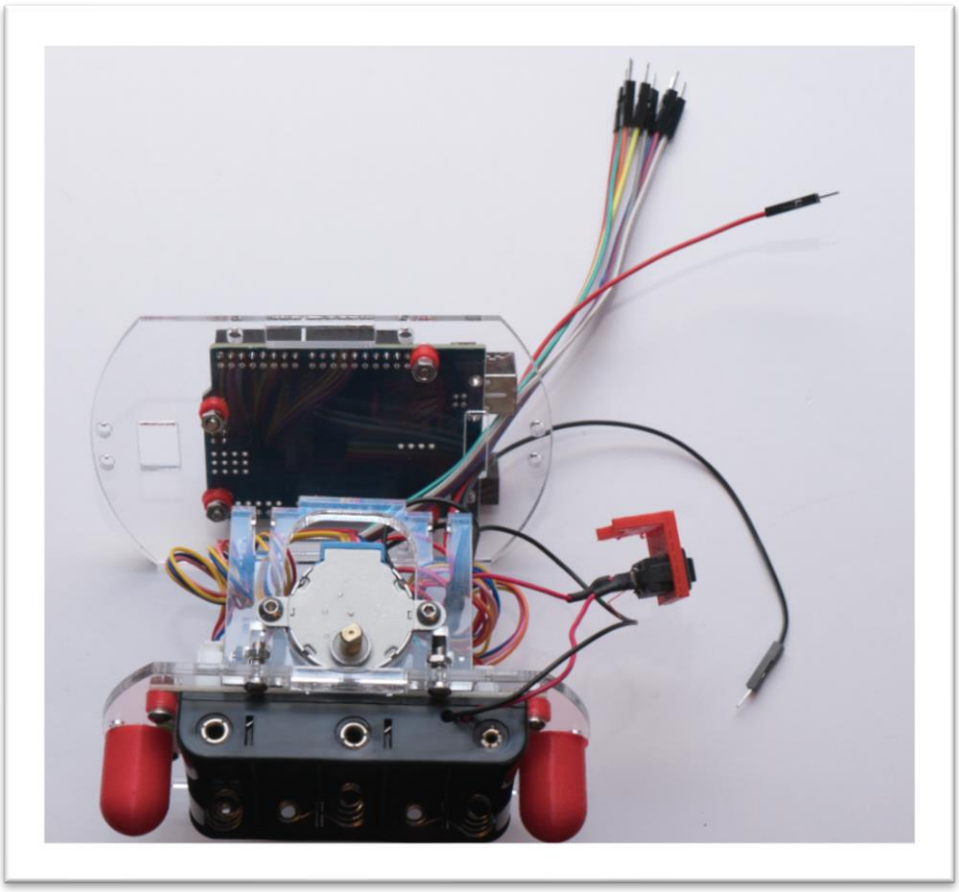
Secure the Arduino to the top plate using four 10mm bolts. The best way to do this is to push the bolts through the holes in the Arduino, push the washers onto the ends of the bolts and then put the top plate on top of this. The washers will hold the bolts in place while you line them up with the holes in the top plate. Note that the Arduino only fits on one way.

10. Fit the robot top

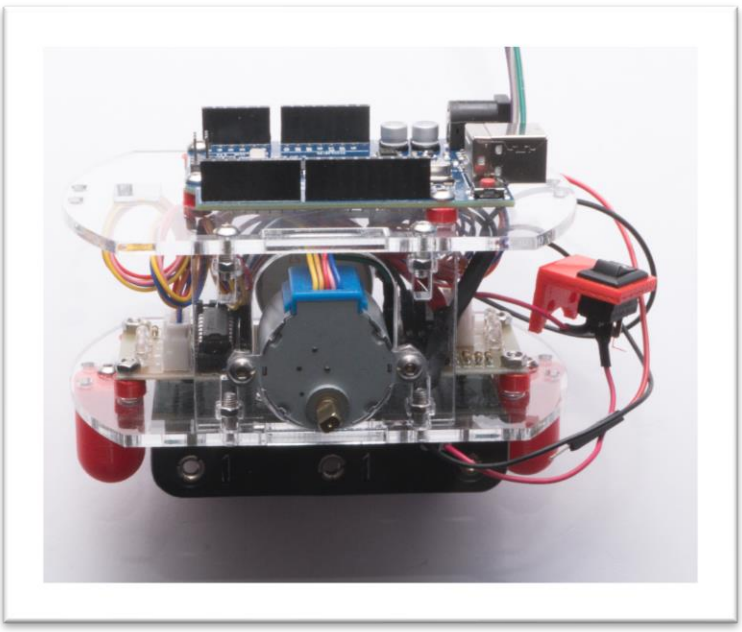
Now we fit the robot top. This is a bit fiddly, but just take it slowly and you'll be fine. You'll need the robot top, the robot chassis, four 10mm bolts and four nuts.



Start by lining up the top and bottom as shown in the picture above. Then fold the motor control cables over and push them through the holes in the robot top, as shown below. Then push the red and black power wires through the hole as well.

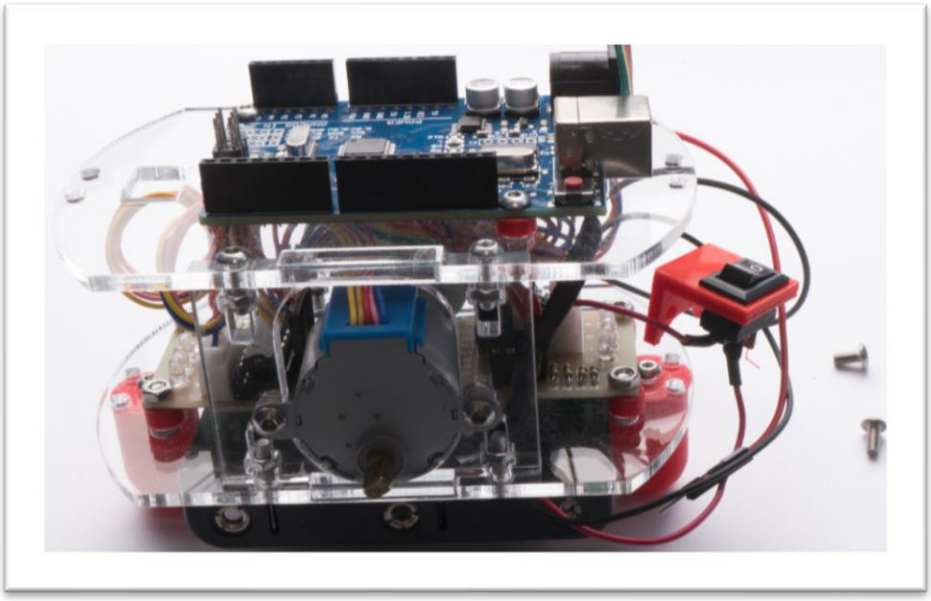


Now fit the top of the robot onto the robot sides in the same way that you fitted the sides to the bottom of the robot.

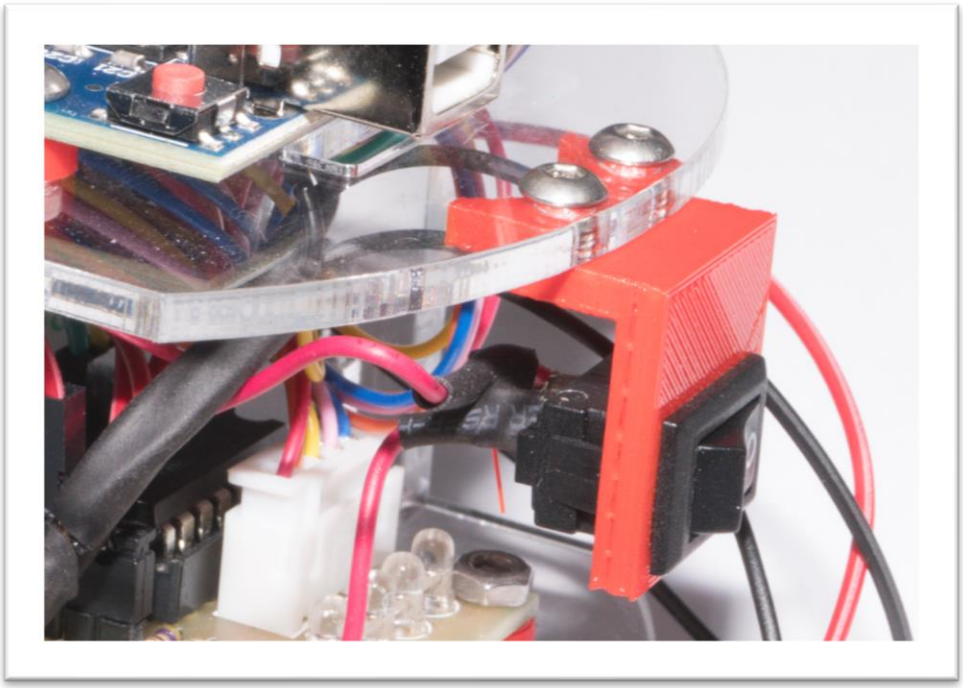


11. Fit the power switch

The power switch is already fitted into a bracket which you screw onto the robot top. You'll need two 6mm bolts to do this.

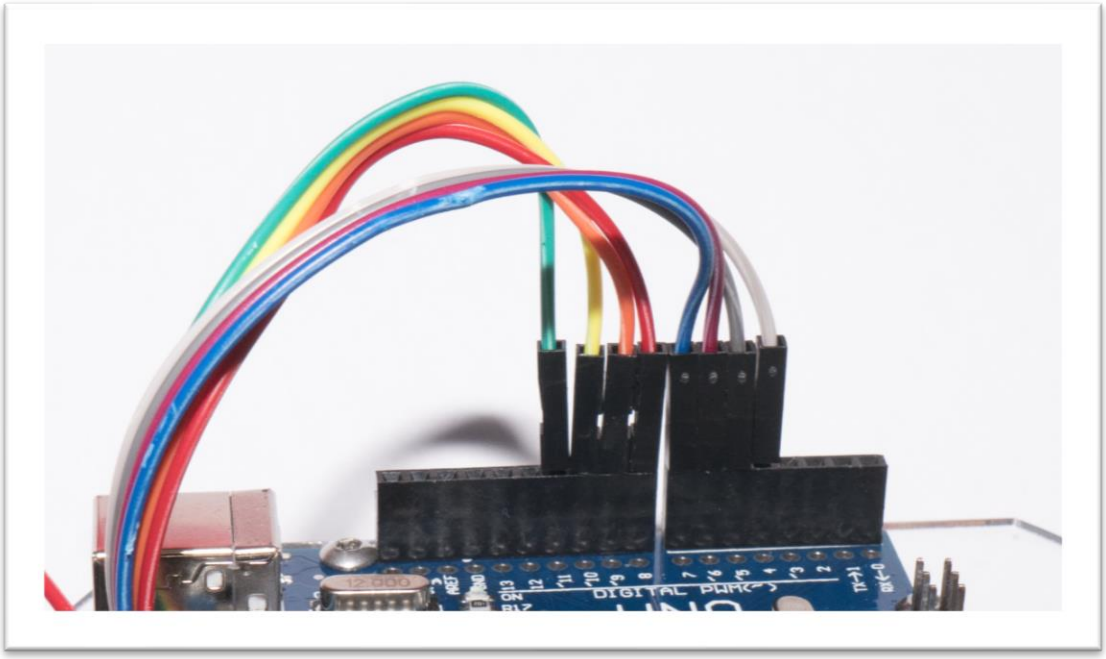


Put the bolts into the holes in the top of the robot and then tighten them into the bracket that holds the switch.

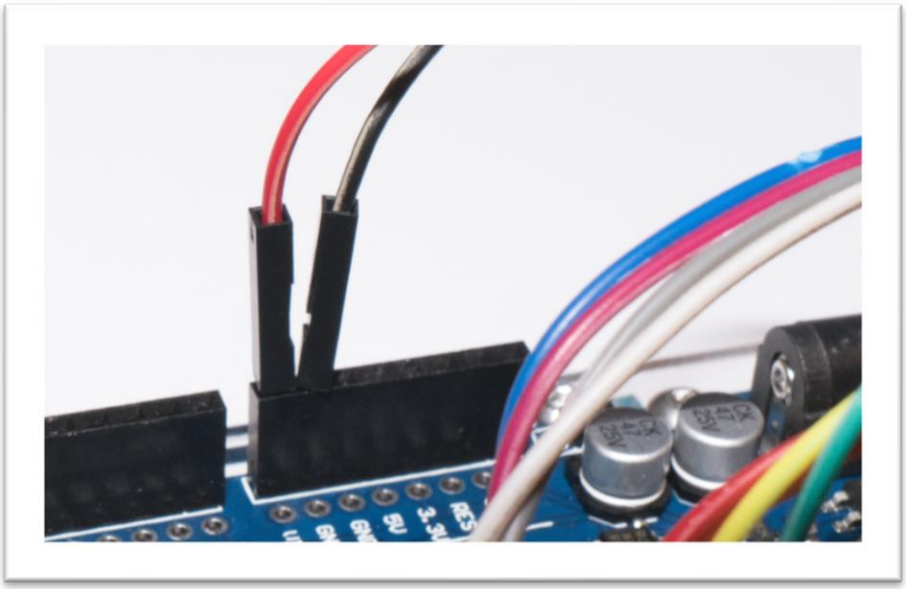


12. Connect the Arduino

Now we need to connect the Arduino to the motor driver boards and the power.



Connect each of the driver cables as shown above. They should be connected to the **digital** pins.



Now connect the two power cables to the other side of the board. The black cable should go to a socket marked **GND** (it doesn't matter which one) and the red should go to a socket marked **VIN**.

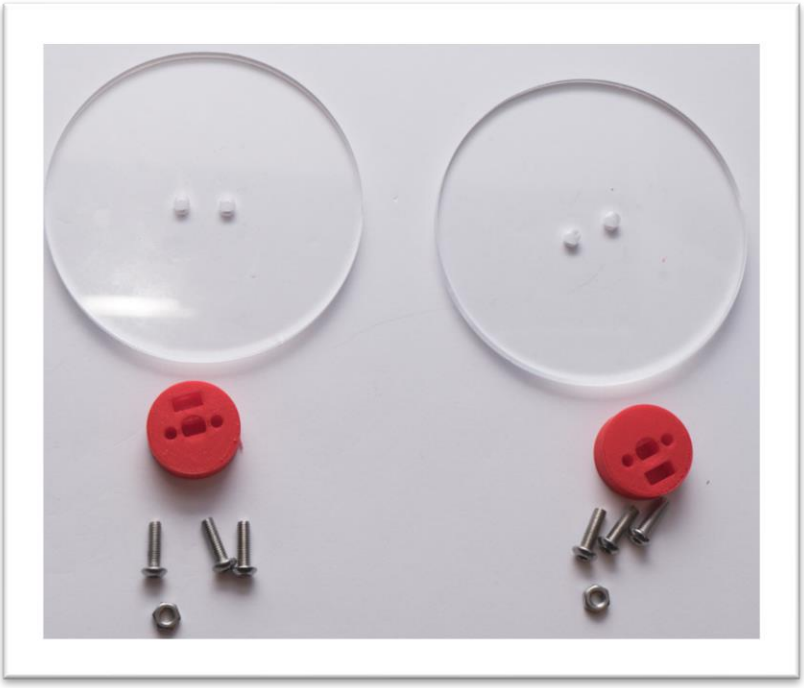
These are the complete connections:

Cable	Arduino Connection
Power Ground (black cable in twisted pair)	GND
Power Live (red cable in twisted pair)	VIN
Left Motor White	D4
Left Motor Grey	D5
Left Motor Purple	D6
Left Motor Blue	D7
Right Motor Red	D8
Right Motor Orange	D9
Right Motor Yellow	D10
Right Motor Green	D11

Connect the cables as shown above. Be careful as you plug each one in. The plugs go in the same sequence as the colours in the cable, but I've found that the cable has a nasty habit of twisting and swapping some of the plugs.

13. Add the wheels

Finally we can add the wheels, to make our robot truly mobile. You'll need two wheels, along with six 10mm bolts and two nuts.

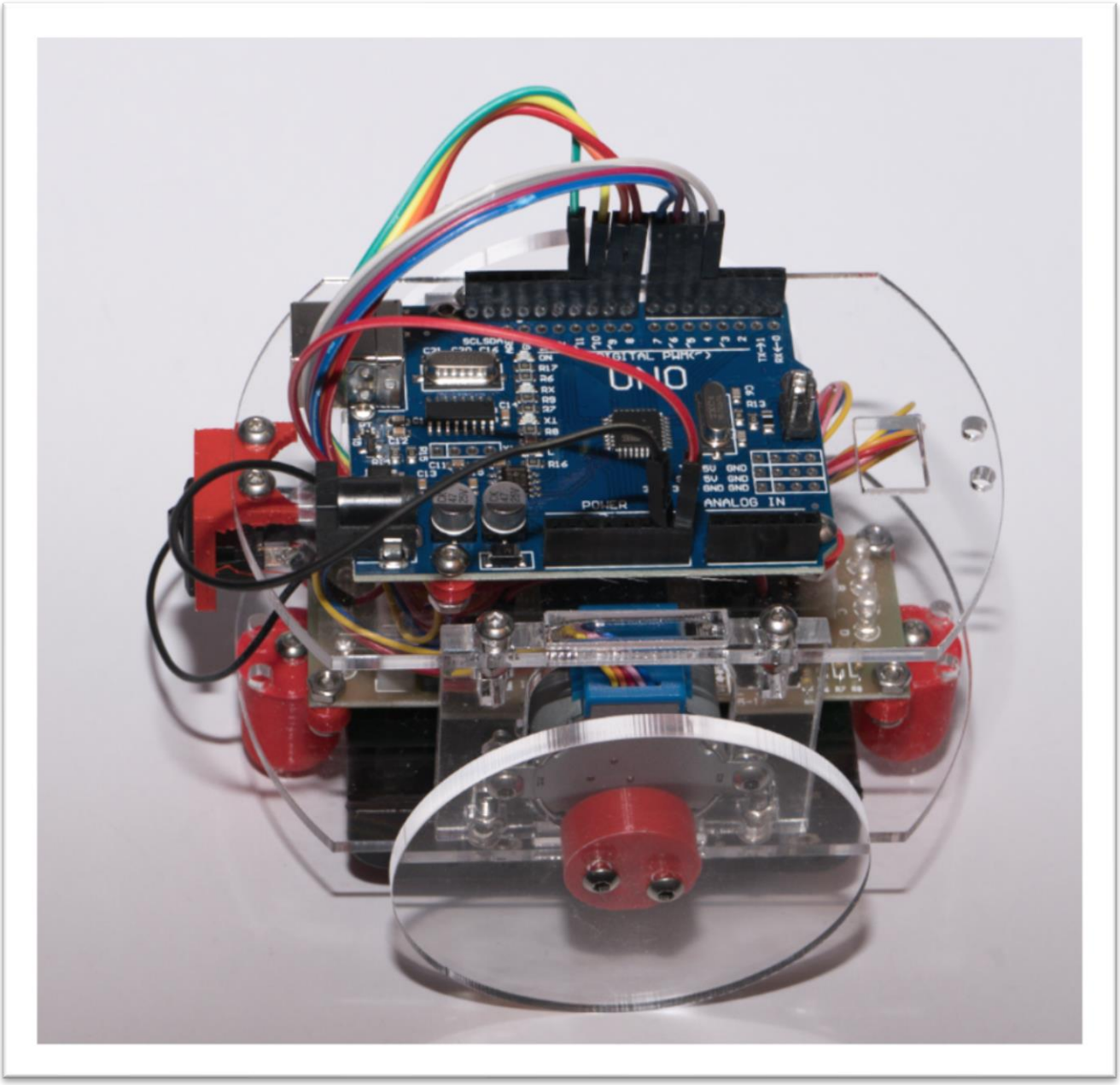


The hubs are bolted onto the wheels. Each hub uses a bolt to clamp down on the motor shaft. The bolt tightens agains a nut that is held in the hub.

Drop the nut into the slot on the back of the bolt, and then screw the bolt into the side of the hub, as shown above.

Now put two bolts through the wheel and screw it onto the hub. For a finishing touch you can put an elastic band around each wheel to serve as a tyre (although this is really fiddly).

Don’t go mad with the tightening of the wheel bolts. If you over tighten them you might find that they force the wheel away from the axle and bend the wheel hub, which is not good.



Congratulations! You've made a complete robot.

Taking our first steps in robotics

What you will do in this chapter

In this chapter you are going to learn the core skills that you'll need to create and deploy programs on the Arduino device. You'll also learn how programs running on the Arduino can send signals to devices connected to it. On the way you'll pick up some C++ programming techniques and even a little bit of physics and electronics.

You'll finish off with some challenges which will let you explore how digital computers can produce "analogue like" signals by a technique known as "pulse width modulation". If that doesn't sound impressive, I don't know what is.

Getting the Arduino software development kit

You might think that Arduino is a brand of computer, but actually it is a standard design which was created by the Arduino organisation. You can buy Arduino boards from Arduino, but you can also get compatible ones from other suppliers, often at a much lower price.

The Arduino organisation has created designs for lots of different computers, the one we are going to use for our robot is called the “Arduino Uno”. This computer fits on a single board and can be powered from a battery or mains adapter. The Arduino Uno has digital ports (signals that can be either on or off) and analogue ports (signals that support a number of different signal levels). The robot will use the digital signals to drive the motor coils (which are either on or off) and read analogue signals from the light sensor.

We will create the programs for the Arduino using a desktop PC. We can use a free program editor that works with either Windows, Mac or Linux machines. When we have written our program, we use the editor to transfer the program into the Arduino where it is stored in special memory that retains data even when the Arduino is turned off. When we turn the Arduino on, it runs the program automatically. Once we have built our robot we’ll find out how to write programs and send them to the Arduino.

The Arduino software development kit (SDK) is a complicated piece of software that contains a number of elements that, when used together, let you create programs for your Arduino. To be precise, the Arduino SDK contains the following things:

- A *text editor* that you can use to write the program text
- A *compiler* to convert the program text into instructions that the Arduino processor can understand.
- A *programmer* that will take the compiled program and load it into the Arduino device.
- A *terminal* that you can use to hold a conversation with the program that you have just written

You will learn to use each of these elements over the course of the next few chapters.

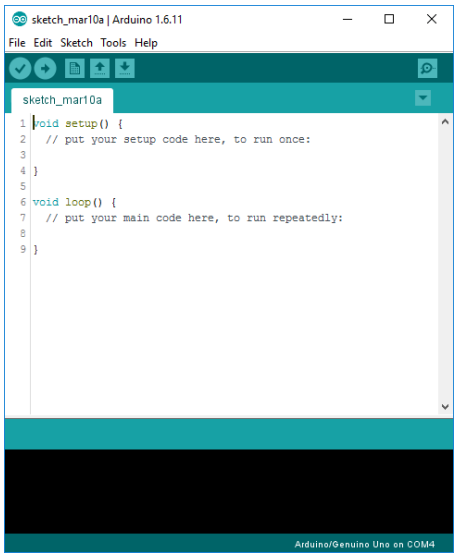
The first thing you need to do is get a copy of the Arduino SDK on your computer. Go to:

```
https://www.arduino.cc/en/Main/Software
```

There you can download versions of the Arduino SDK for Windows, Mac or Linux PC. The Arduino SDK works the same on all the different platforms. The only difference between the platforms is in how they manage the connection between the computer and the Arduino. All the connections use a USB cable that links a usb port on the computer the computer to the big socket on the Arduino board. However, the precise way that the usb port is managed differs for each of the computers.

Go to the web address above and download the software for your machine.

Follow the instructions to install the software and then start it up.



When the Arduino SDK starts running you should see a screen like the one above. The next thing you need to do is get this software connected to your Arduino device.

Connecting your computer to your Arduino device

The Arduino uses a *serial* connection to talk to the host computer over a USB connection. One end of the cable goes into a USB port on your PC, and the other end of the cable connects to a component in the Arduino that provides the serial port. You might need to install a driver that lets your PC talk to the Arduino serial hardware. Here are the options:

Windows PC connection

With a PC, you should find that when you plug the Arduino into the PC it is detected and any drivers that are required are loaded automatically.

Mac connection

In the case of the Mac, things are not quite so smooth. It might be that the Arduino will work, but you may need to download and install a driver to make it work. You can find the driver here:

```
http://www.wch.cn/download/CH341SER_MAC_ZIP.html
```

Don't worry that the site has a lot of Chinese on it, just download the driver and follow the instructions to install it. Then you may be asked to reboot your Mac (shut it down and restart it). After the reboot your Mac you should find that you can connect your computer to the Arduino.

Linux PC connection

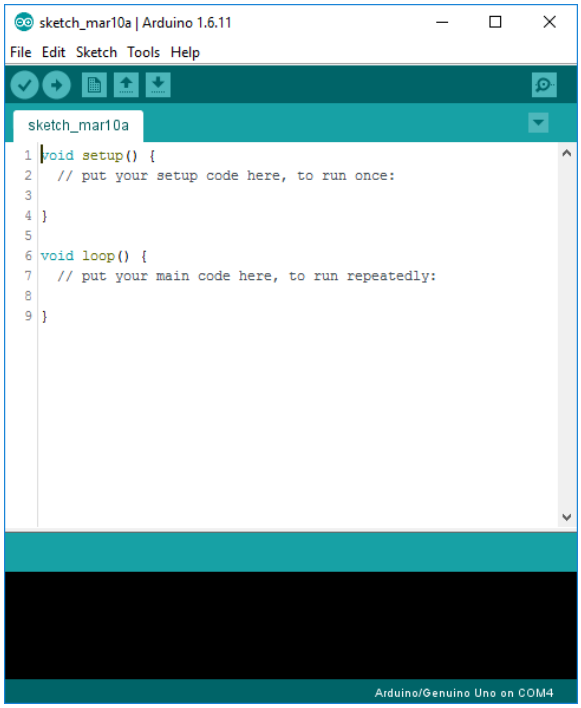
With a Linux PC it rather depends which particular version of Linux you are using. You can find some help here:

```
http://playground.arduino.cc/Learning/Linux
```

Writing our first Arduino program

There's an unwritten rule in programming that your first program involving hardware has to make some lights flash. So, that's what we are going to do. Then, when we get bored with that (after about five seconds in my case) we're going to get the robot motors to turn.

I'm going to be using the Windows PC version of the Arduino software. The screens look very similar for the other platforms. We can start by opening the Arduino software.



This is what the Arduino software looks like when you start it up for the first time. What you can see is an empty Arduino program. You can download it into your robot and it will run, but it won't do much. The text in the middle is the actual program code, written in the programming language C++.

The C++ programming language

C++ is a big, complicated programming language that is used all over the world. Open up a game on your Xbox and there's a very good chance it was created using the C++ programming language, as were large chunks of Microsoft Windows, the Apple Mac operating system and Linux. To name just a few trademarks.

We're not going to learn the whole of C++ in this book, we're going to learn just enough to get by. You'd use the same technique if you were going abroad. You'd learn just enough of the language to be able to get a room in a hotel, and order a drink (and perhaps ask the way to the beach). We're going to learn just enough to make our robot do what we want.

Having said that, this will serve as an introduction to programming, which you might find useful in the future.

What is a program?

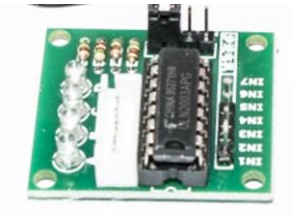
A program is a set of instructions you give the computer to tell it how do something. When we write programs to control an Arduino device the instructions are presented in the form of two *functions,* both written in C++. A function is a lump of program code that has been given a name. In this program one function is called `setup` and the other is called `loop`.

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

This is the text of the program that you can see in the Arduino editor above. The names of the functions actually tell us exactly what happens in them. The first one is where we set up the Arduino hardware. The second one is called repeatedly to provide control of whatever the Arduino is plugged into.

Flashing a light on the motor driver board

The motor driver boards for the stepper motors on the Hull Pixelbot have lights on them.



Above you can see a driver board, on the left you can see a vertical row of four light emitting diodes (leds) that indicate when their corresponding motor coil is turned on.

So, if we turn on one of the coils in the stepper motor we should see the light for that coil turn on. The lights themselves are a special kind of electronic component called a *light emitting diode* or LED.

You might be wondering why we have a “stepper motor driver board”. Why can’t we just connect the Arduino directly to the motor?

This all has to do with power. Power is the stuff that makes things happen. Your car can go faster than mine because the motor in your car produces more power. To control your car you press a pedal. This doesn’t actually make the wheels move (unless you are very young), instead it sends a signal that tells the motor to produce more power, causing the car to go faster.

You can think of the output pin from the Arduino as being a signal, which can tell another switch what to do. In the case of the stepper motor driver board above, the *chip* (the black component on the right hand side of the board) contains some transistors that are acting as

switches and can be turned on or off by the input signal. To help people understand when the motor is turned on, the driver board has the lights that signal when a particular output is turned on, and we are going to use those in this exercise.

The stepper motor driver board is fine for driving the small motors on our robot, if you want to control larger amounts of power you need bigger components, and of course a larger power supply.

Digital Signals

The Arduino is fitted with a number of connections that can work with *digital* signals. A digital signal can either be HIGH (sending out a voltage) or LOW (not sending out a voltage). In the case of the Arduino hardware the HIGH state means 5 volts, and the low state means 0 volts.

It's dangerous to assume that HIGH always means 5 volts. In some digital devices, the HIGH state means 3.3 volts. The difference in voltage doesn't sound that much, only 1.7 volts, but this is enough to cause damage, so you need to be careful when connecting your robot circuits to other devices.

Our light flashing program will use a digital signal to make the light flash. The digital pins on an Arduino can be either inputs (read the state of the digital pin) or outputs (set the state of the digital pin). When we turn the Arduino on, all the pins are set as inputs.

Using the `setup` function

The designers of the Arduino provide the `setup` function to set up the hardware ready for use by the rest of the program. In the light flashing program we are going to use this function to configure one of the signal pins on the Arduino to be an output. Then, in the `loop` function, we are going to this pin alternately HIGH and LOW. This should cause the led to appear to flash. We have one of the motor coils connected to digital pin number 4, so let's use pin 4 and set it to be an output. This is our `setup` function.

```
void setup() {
  pinMode(4,OUTPUT);
}
```

C++ programs can use *libraries* of readymade functions. The `pinMode` function you can see above does just what the name implies. It sets the mode of the given pin. We tell the function the number of the pin and the mode that we want it to use. If we wanted to set the mode of a different pin, we'd use a different number.

What do you think we would use if we wanted to set pin 5 as an input?
Yep – that's right:

```
pinMode(5,INPUT);
```

Using the `loop` function

The `loop` function is called repeatedly when the Arduino is running. If you think about it, this makes good sense. Programs that control devices are inherently "loopy", in that they will have a behaviour that needs to be repeated as long as the device is switched on. A digital thermostat created to control the heating in a house would continuously read the temperature and turn on the heater if the temperature gets too low.

In the case our light flashing program we want the `loop` function to turn the light on and then turn it off again. Then, when `loop` is called repeatedly we get our flashing light.

```
void loop() {
  digitalWrite(4,HIGH);
  digitalWrite(4,LOW);
}
```

The `loop` function above turns pin 4 high and turns it low, which will cause the corresponding led on the stepper motor driver board to go on and off. The `loop` function uses a library function called `digitalWrite` which is told the number of the pin to be written, and the state to write to that pin.

You might be wondering how the C++ language actually calls the `loop` function repeatedly. We will find this out later, when we write some loops of our own.

Running our first program

Below you can see the complete program that is going to flash the light for us.

```
void setup() {
  pinMode(4,OUTPUT);
}

void loop() {
  digitalWrite(4,HIGH);
  digitalWrite(4,LOW);
}
```

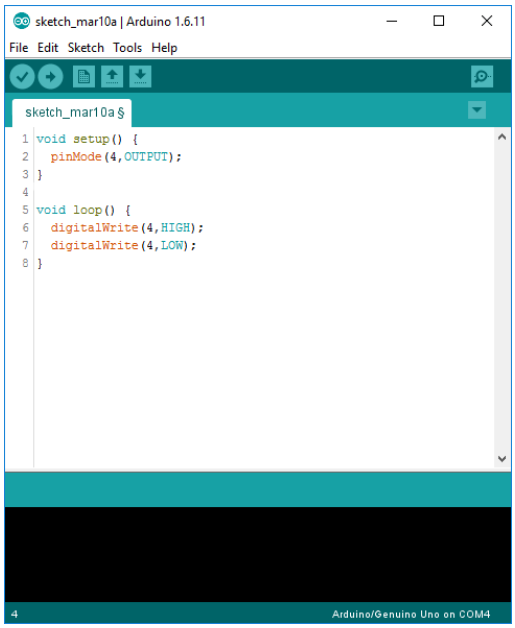
`Ex01 FlashLed`

You can find this code in the folder "`Ex01 FlashLed`", but typing it in is fun too.

Next, we need to get the program into the Arduino on the robot. We use the Arduino software to do this. Remember that this software doesn't actually run the program above, it lets us create the program and then send it to the Arduino on the robot. Once we've loaded the program into the robot it will start running when the robot power is turned on.

Start the Arduino software on your computer.

Type in the program above.



Now that we have the program typed in we can load it into the robot. The first stage is connecting the robot to your computer.

Connecting the robot to your computer

You'll need a USB cable to do this. Many Arduino kits are supplied with a cable, frequently one in a nice shade of blue. This will work fine, but you might find it a bit short if you're using a desktop PC. If you need to buy another one you need to search for a USB type B cable.

Plug your robot into the computer using the USB cable. If you are using a Windows PC you should hear the characteristic sound of the computer finding a new device. If this is the first time you've plugged the Arduino into your computer, you may have to wait a little while as the operating system locates drivers for it. For more details on how to do this, check the section *Connecting your computer to your Arduino* above.

There's no need to turn on the battery power for your robot when it is plugged into the computer. The way that we have wired up the robot means that power will be supplied from the computer to the Arduino and the robots when the robot is plugged into it. You can have the robot battery power turned on if you like, but this might use up a bit of the battery.

Configuring the Arduino device

Next, we need to tell the Arduino software which device it is using. The Arduino software can be used to build and deploy programs for lots of different computer boards, each of which is programmed in a slightly different way.

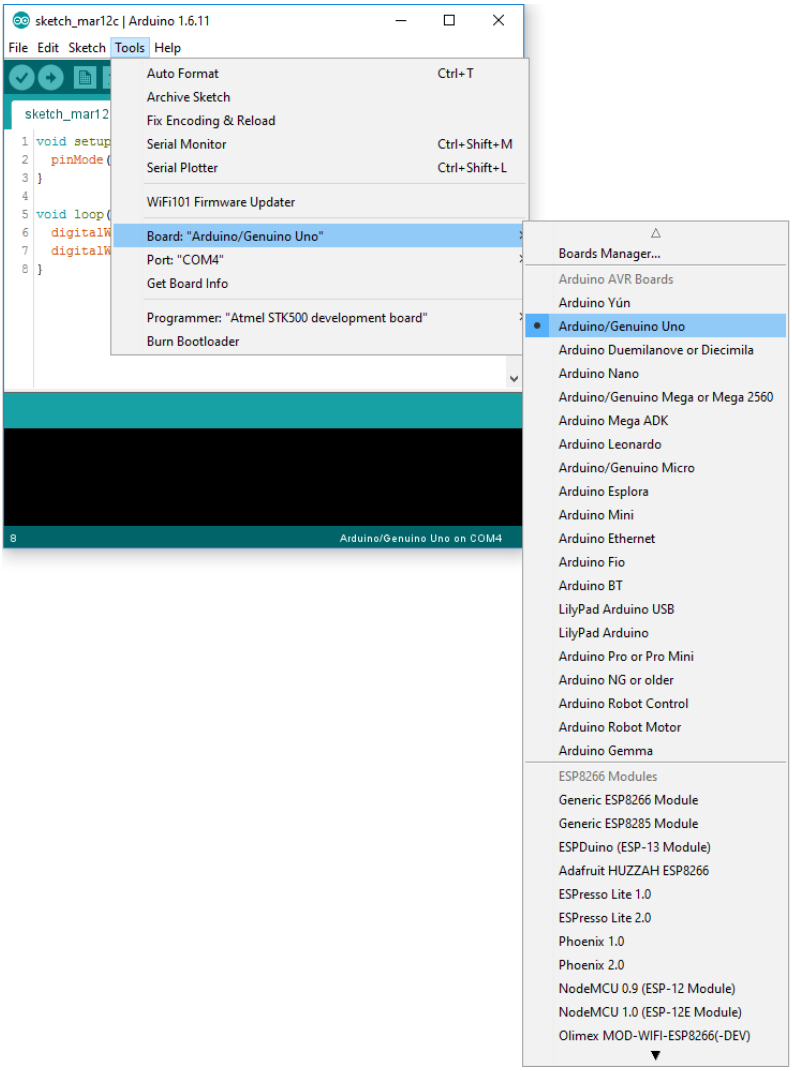
This is actually very useful. It means that once you've learned how the program the Arduino you also know how to program all the other devices as well. If you need to use a device with more power, or a smaller device, or one which has a built-in WiFi network connection you can write your program in just

the same way as we have written the light flashing program and then send it into the specific board that you are using.

The bad news is that it can be confusing. There are lots of different devices with similar names and we need to get exactly the right device.

This is very important. If you forget this step you will find that really strange things happen when you try to send the program into the Arduino.

Click the **Tools** menu and select the **Board** item to open the menu that lets us specify which board we are using.



Check that the board that is selected is "**Arduino/Genuino Uno**". If that board is not selected, click "**Arduino/Genuino Uno**" to select it. The menu will close.

You can confirm that the correct board is selected by checking the bottom left hand corner of the Arduino editor.

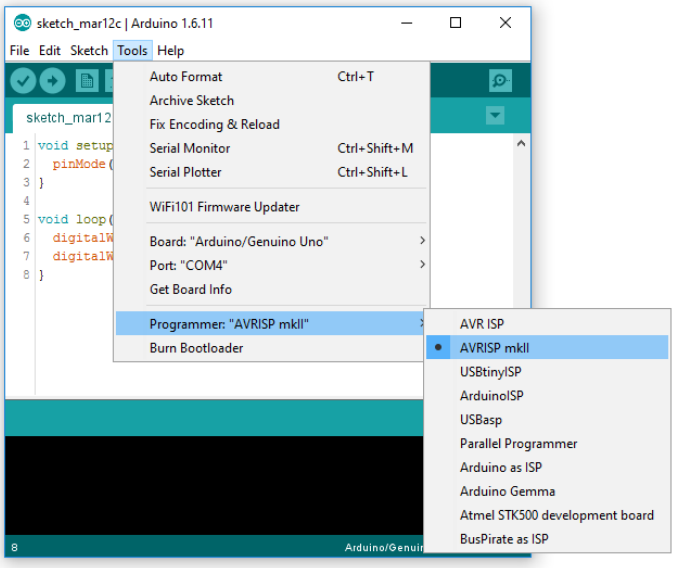
Configuring the programmer

The Arduino software takes your C++ program and converts it into a set of low level instructions that the Arduino processor in your robot can understand. Then it transfers these instructions into the Arduino, where they are written into special memory that holds its contents even when the power is turned off.

This means that we don't need to send the program into the robot each time we want to make the robot do something. We just have to turn the power on, and the program held inside the robot will start to run.

The only time we have to re-program the robot is when we want to change the program for a different one. The part of the Arduino software that writes the program is called the programmer. There are quite a few different kinds of programmer, depending on the particular connection being used for the robot. We need to make sure that we are using the correct one.

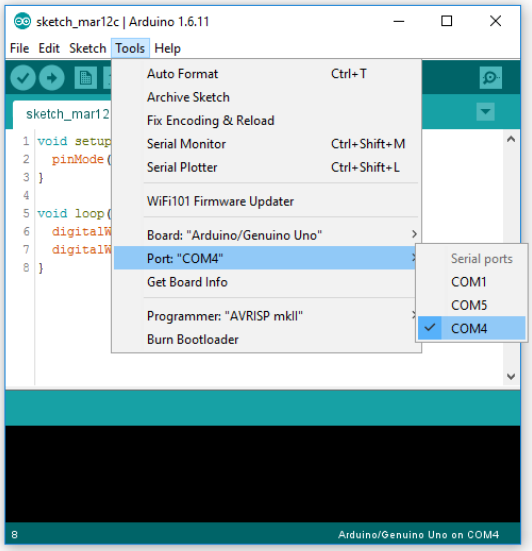
The Arduino software now needs to know which programmer will be used to transfer the program into the device. Select the **Tools** menu, open the **Programmer** and select **AVRISP mkII** as shown below:



Selecting the serial connection

Finally, you need to tell the editor the connection that the robot is using. At any given moment, a number of "serial communication" or "com" ports might be in use on your computer, depending on what is plugged into it. You have to determine which one has the Arduino connect and tell the editor program this information.

Now you need to select the serial port that you will be using. Click the **Tools** menu and select the **Port** option.



If you are lucky there will be only one serial (com) port connected. As you can see above, I've not been lucky, in that there seem to be three ports in use, one of which will be the Arduino. This is because my computer has a Bluetooth adapter, which also provides serial connections.

The best way to find out which of these ports corresponds to the Arduino is to close this menu, un-plug the Arduino, open the menu again and note which serial port has disappeared. Close the menu, plug the Arduino in again and, when you open it, you should be able to find that com port and select it

This might seem a bit of a pain to do, but the good news is that the Arduino editor remembers these settings and will automatically use them next time you start the program.

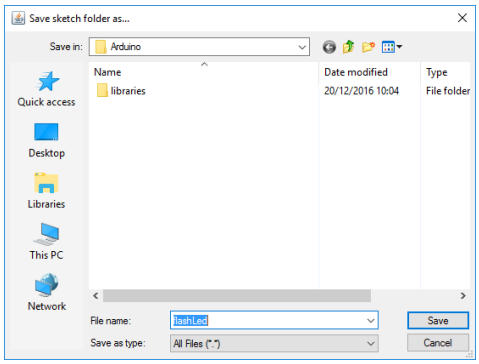
One tiny piece of final advice about com port settings. Each USB socket on your PC will be associated with a particular COM port number. If you plug the robot into a different socket on your computer you'll find that you'll have to change the configuration of the Arduino editor to use a different serial port.

Building and deploying the program

Finally, we need to get the Arduino editor to build the program and send it to the robot.

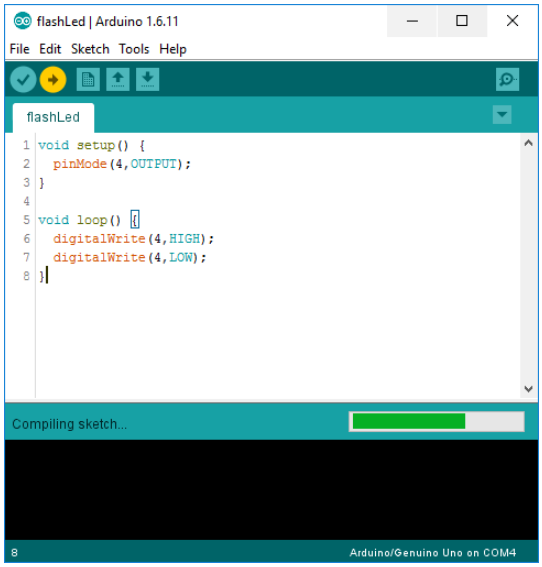
The button that triggers this action is on the Arduino editor.

Click the run button (indicated with the arrow).



The very first time that you try to run a brand-new program you'll be asked to save it somewhere. I've given the program the name `flashLed`. Unless you say otherwise the programs are stored in the **Arduino** folder in your **Documents** folder. However, you can save them anywhere you like.

Pick a sensible name and location for your program and click **Save**.

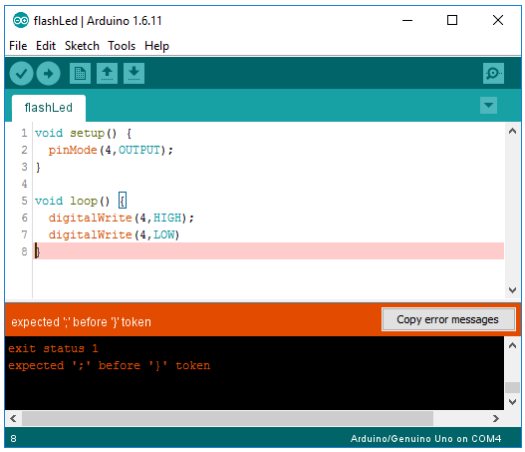


The program will now be *compiled*. When a program is compiled the text in the Arduino editor (the `digitalWrite` commands etc that you have written) is converted into instructions that the computer processor chip in the Arduino can understand. This process is called *compilation*.

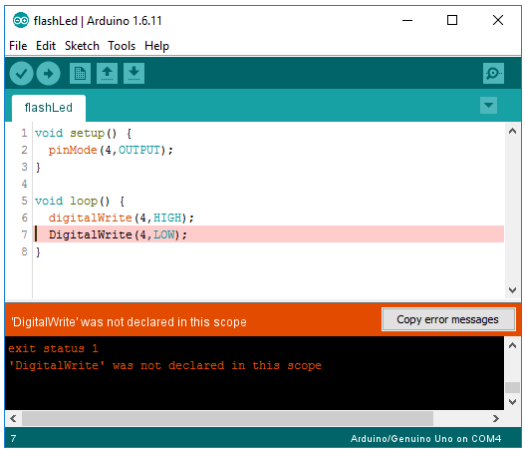
Compilation is performed by a large program called a *compiler* which is started by the Arduino environment when you press the Run button. The compiler will look at each line of the program in turn and try to make sense of it. The compilation will complete perfectly **if** you have typed the program text in completely correctly. If you haven't, then the compiler will produce errors which you'll have to deal with. Unlike humans, who are very good at dealing with incomplete information and making sensible assumptions about the meaning of a statement, the compiler has to have exactly the right input or it will reject the input.

Dealing with compilation errors

We can take a look at a few common errors and consider how to fix them. If your program has worked first time then “lucky for you”. However I think you should still read this section because you might not be so lucky when you write your next program.

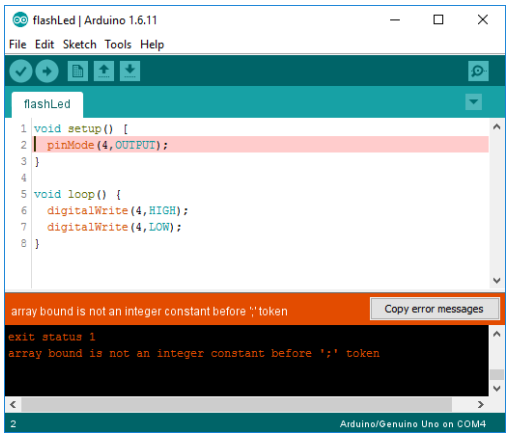


This error is quite helpful. If you look carefully you’ll find that a semi-colon (;) is missing from the last call of the `digitalWrite` function.



This error is a bit less helpful. But after a while you’ll notice that the second call of `digitalWrite` has got incorrect spelling. The compiler is very particular about how you spell things, and it will complain if you use a capital D instead of a lower case one.

You should also remember that the point at which the error is detected is not always where the error occurs:



This is a horrible error. You can stare at line 2 until you are blue in the face (other facial colours are available). And you can also have lots of the wrong kind of fun trying to find out what on earth is going wrong with the program array bounds (whatever they are). But if you look at line 1, you'll find that the curly bracket character "{" which should be there (it is how a program starts a block of statements) has been replaced with a square bracket "[" (which is how a program creates an array). The compiler finds this very confusing, and produces an error when it notices the problem. But this is not the line where the error is.

If you get errors, see if you can work them out. Note that they will be caused by bad typing (I know that the program works OK). Worst case you can type in the entire program a second time. Hopefully you'll not make a mistake in the same place as last time.

Once the compiler is happy with your program it turns over control to the programmer component of the Arduino environment which will then upload the instructions to the Arduino.

Once this has finished you should see your program running in the robot. Well done, you've just deployed your first Arduino program. Although it might not seem to work properly…..

Debugging the light flashing program

The title of this section might not fill you with confidence. It's as if I'm not expecting the program to work. This is not quite true. The program works fine, but it doesn't do what we want. When you run the program you should notice two things wrong with it:

- The light doesn't flash, it says on all the time
- Another light comes on as well

What we want to do is make one of the lights flash on and off. What we've got is two lights lit, and neither of them flashing. At this point there are two things you can do:

1. Change the specification of the program from: "I want to write a program that makes one light flash" to "I want to make a program that makes two lights turn on".
2. Find out what is wrong and fix it.

Option 1 isn't really an option I'm afraid. Particularly if we have a paying customer who is expecting to be given a program that flashes a light. So, we need to work out why the light doesn't work as we expect.

Why doesn't the light flash?

Actually, the light is flashing. It's just that it is happening far too fast for us to see it. The processor chip inside the Arduino can process several million instructions a second, and so the light is being turned on and off many thousands of times a second. Our eyes can't see things flashing that fast, so it looks to us as if the led is just lit all the time. What we need is a way of delaying the flashes so that our eyes can catch up.

```
void setup() {
  pinMode(4,OUTPUT);
}

void loop() {
  digitalWrite(4,HIGH);
  delay(500);
  digitalWrite(4,LOW);
  delay(500);
}
```

`Ex02 FlashLed Delay`

This version of the program uses a new function, called `delay`. This does exactly what the name implies. It delays the program. The number that follows is the length of the delay that we want, in *thousandths* of a second. The code above is asking for 500 thousandths of a second, which is half a second. We put a delay after the program has set the output high, and another after it has set the output low.

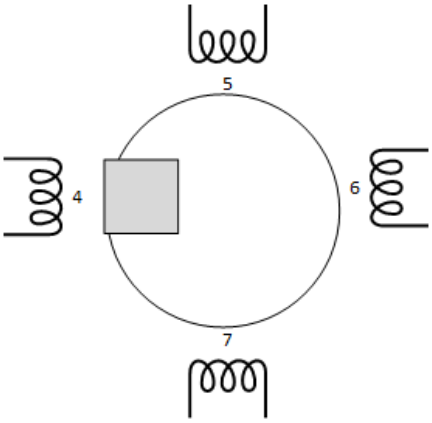
Modify your program so that it includes the delay. Use the Run button to send the program to the Arduino and see if it works.

With a bit of luck, provided that you don't miss-type any of the words, you should find that one led now flashes on and off at a speed that we can see.

And, as a bonus you should also notice that the other led no longer seems to be lit up. Except, that if you look at it very closely you'll find that the extra led flicks on just as the flashing led turns off. What is going on here?

Magnets and stepper motors

The answer turns out to be magnetism. It's all to do with the stepper motor that is being driven by the motor board.



Above you can see the four coils that are fitted inside the stepper motor. I've numbered each one with the Arduino pin that it is connected to. The round thing in the middle is the motor shaft, and there's a block on the shaft that is a little magnet. If we turn a coil on, the magnet is attracted to that coil. At the moment the coil connected to pin 4 is turned on, and so the magnet is attracted to that coil.

You can discover how well this works by trying to turn the robot wheel while the light flashing program is running. When the led is lit, you'll find the wheel much harder to turn. This is because when the coil is being driven it attracts the magnet on the motor shaft and holds it there.

However, you might be wondering what this has to do with the other led lighting up when our program is running. Well, it turns out that two coils which are facing the same way make a thing called a *transformer*. We use these all the time. We are surrounded by power supplies that take the nasty, dangerous high voltage mains supply in our houses and convert it into a much friendlier five-volt supply that we can use to charge up our phones and tablets. These are actually made using two coils.

One coil in the transformer is driven by the mains voltage. This coil is switched on and off rapidly and, because of the magic of magnetism, this induces a voltage in the other coil. Exactly the same thing is happening inside our stepper motor. You can see that the coils on pins 4 and 6 are facing the same way, and so changes in the magnetic field generated by the coil on pin 4 will cause a voltage to appear on pin 6.

You might not believe that this is the cause. If you don't there are two ways you can test my theory.

1. You can remove the connection to the motor. According to my theory, that should stop the extra light from coming on.

2. You can change the program to drive pin 5. That should cause a different pair of pins to light up.

If you're wondering why we saw that tiny flicker of the led when the program was running the flash program with the delays in it, this is because voltage is only induced in the "receiving" coil when the voltage in the "transmitting" coil changes. When the program switches off a coil a magnetic field in the motor collapses, inducing a voltage in the "receiving" coil as it does so.

There's a device you use every day that uses this principle to extremely good effect. In fact, it wouldn't work if we couldn't use coils in this way. And millions of people rely on this device to get to work each day. What do you think the device is?

Ans: It's a car engine. A petrol engine uses a *spark plug* to ignite the petrol vapour which then explodes and produces the power to move the car. At the heart of the spark plug is a tiny gap, across which a spark is made to jump at just the right instant to fire up the engine. The voltage needed to make a healthy spark is much higher than the measly 12 volts provided by the battery in the car, so a coil is used to generate a much higher voltage. The principle is exactly the same as the one that causes our extra led to illuminate.

If you find this interesting, you should find a decent book about electronics and take a look in there for more about transformers and how they work.

Challenge: Using the built-in LED on the Arduino

The Arduino Uno has a rather useful built-in led that you can use in exactly the same way as the ones above. It is connected to pin digital pin number 13. You can use this any time you want to send a signal to a user.

Use what you have learned to control the led on the Arduino.

1. I want a led that flashes for a tenth of a second every second.
2. I want a led that glows at half brightness.
3. **Grand challenge:** I want a led with adjustable lighting levels, from off to full brightness. I set the brightness in the setup method by giving a number from 0 (completely off) to 100 (completely on).

Hint: You are using a technique known as "Pulse Width Modulation". Which sounds very posh. The width of the on and off pulses controls the brightness of the light that we see. This is a technique frequently used in electronic control, for both light intensity and motor power. One way to do this is to have two variables, one for the on time and another for the off time and then set these values to give the required brightness levels.

In other words: for a dim light the led could be on for 1 millisecond and off for 90 milliseconds. For a bright light the led could be off for 1 millisecond and on for 90. You'll have to experiment with different timing values so that you get the level of brightness that you want without flickering. For maximum points (and perhaps a prize) you could make a light that fades slowly up and down.

Driving the stepper motors

What you will do in this chapter

After the previous chapter we can create and deploy programs that can control digital outputs, but what we really want to do is make the stepper motor turn so that we can use it to drive our robot around. In this chapter, you are going to discover how stepper motors work and how an Arduino program can make a motor turn. Along the way you'll discover more features of the C++ programming language, including how a program can stored data and make decisions. By the end you'll be able to create software that will make your robot turn and move precise distances.

You'll finish off with some challenges where you can show how well your programs can control the position of your robot.

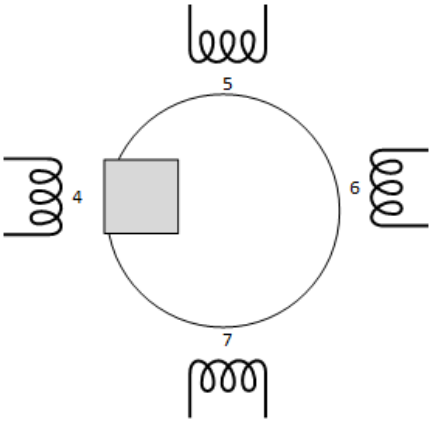
Stepper Motors and Electric Motors

You might think that we can get the robot motors to turn by just giving them power. Some electric motors work like this. You turn them on and the start spinning. They are sometimes called DC (short for *direct current*) servo motors. Such motors are cheap, powerful and you find them in things like fans, pumps and hair dryers. However, the motors that we are using in the robot don't work like that. They are *stepper* motors. We make them turn by making them move in a series of steps. Rather than just on the power, we need to create a set of electrical signals that will make them run. To do this we will have to use our software skills.

A motor that requires software to make it turn sounds like a lot of work, but the good news is that we can use our software to get very precise control of the robot movement. Devices that require very precise positioning of components, for example printers, make use of stepper motors to allow this.

Making a stepper motor turn

We have seen that a stepper motor is made up of a number of coils that surround a shaft in the motor. The shaft has a magnet fitted on it. If we turn on one of the coils in the motor it will attract the magnet and cause it to move towards that magnet. By carefully managing the sequence in which the coils are switched on and off a program can make the shaft turn continuously.



Take a look at the diagram for our motor. If coil 4 is turned on the magnet on the motor shaft is pulled towards that coil. If we want to make the magnet pull the shaft around we can do this by turning on different coils in turn. How about a sequence like this?

We could start with coil 4 turned on, which we could call the starting positon.

- Turn on coil 5 (magnet is pulled between coils 4 and 5)
- Turn off coil 4 (magnet is pulled to coil 5)
- Turn on coil 6 (magnet is pulled between coils 5 and 6)
- Turn off coil 5 (magnet is pulled to coil 6)
- Turn on coil 7(magnet is pulled between coils 6 and 7)
- Turn off coil 6 (magnet is pulled to coil 7)
- Turn on coil 4 (magnet is pulled between coils 7 and 4
- Turn off coil 7 (magnet is pulled to coil 4)

We know how we can turn coils on and off, so we just have to string together the required sequence:

```
void setup() {
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
  pinMode(6,OUTPUT);
  pinMode(7,OUTPUT);
  digitalWrite(4,HIGH);
}

void loop() {
  digitalWrite(5,HIGH);
  delay(2);
  digitalWrite(4,LOW);
  delay(2);
  digitalWrite(6,HIGH);
  delay(2);
  digitalWrite(5,LOW);
  delay(2);
  digitalWrite(7,HIGH);
  delay(2);
  digitalWrite(6,LOW);
  delay(2);
  digitalWrite(4,HIGH);
  delay(2);
  digitalWrite(7,LOW);
  delay(2);
}
```

Ex03 TurnMotor

This is a complete program which will cause the motor to turn. There's nothing particularly scary about this program (although you might not agree). It's just a much-expanded version of our first light flashing program that turns on and off different coils rather than just one. It sets the data pins for each of the coils to outputs (in the `setup` function) and then turns on each coil in sequence to get the motor turning (in the `loop` function). The code in the loop function goes through the 8-signal sequence to make the motor turn 1 revolution. Because loop is called repeatedly, this causes the motor to turn continuously.

Enter the program and run it. It should make the left-hand motor on the robot turn clockwise.

The program above has a delay of 2 milliseconds after each write. This is to allow the motor time to move to the next position. What do you think would happen if you changed the delay to 3? Try it.

What about a delay of 1? This might not work so well.

The need for speed

When you run the program the motor actually turns quite slowly. If you look at the code, and the 2 millisecond delays between each change of signal, you might expect the motor to run really quickly. But it doesn't. There are two reasons for this:

1. The motor contains some tiny gears that step down the output from the actual motor shaft, so that it takes 64 turns of the motor to create one turn of the output. This is done so that the motor runs at a more manageable speed, and also to allow the motor to provide more power.
2. The poles that attract the rotor are actually duplicated around the rotor, rather than having 4 poles as we have in our diagram, the motor has 8 poles, so there are 32 different positions. This makes very good sense, as it would be really difficult to make a magnetic field strong enough to move the motor a quarter of a turn. However, it also means that we need to go through the sequence 8 times to make the motor turn once.

Put these two things together and we find that we have to provide 512 (64 * 8) pulse sequences to make the motor output turn once. So, with these motors high speed racing is not going to happen. But the good news is that we do have very precise control of the motor position, which makes it very useful for positioning things.

If your motor is like mine, you'll find that a delay of 2 makes a slow-moving motor with plenty of power, whereas a delay of 1 makes a motor that runs more quickly, but is very easy to stall. What we really want is a delay somewhere between 1 and 2 milliseconds.

To get much smaller delays we can use a different delay function, which lets a program specify the delay in *microseconds*. A microsecond is a millionth of a second, i.e. a thousandth of a second. I've found through experiment that a delay of 1200 microseconds (i.e. 1.2 milliseconds) works well with the motors.

Change all the occurrences of `delay(2)` to `delayMicroseconds(1200)` in the program (or load the example code in "`Ex04 TurnMotorMicros`" – your choice).

If you're feeling lucky you can try smaller values of the delay to squeeze even more speed out of your motors.

Why are some motors "faster" than other? Why can we use the same value in all motors?

The answer here is "because hardware". When you write a computer program you can be pretty sure that it will run in exactly the same way on every computer. However, when you make 10,000 motors you'll find subtle differences between each one. Some will have slightly stiffer motor shafts than others, some will have stronger magnets than others. This means that whatever software you make to control the hardware needs to make allowance for the "worst case" hardware; where you have a stepper motor with a weak magnet and hard to turn gears.

> Proper robotics research is always performed with real, physical robots for just this reason. While you achieve a lot by using software to simulate how a mechanical device works there is no substitute for real hardware.

Getting control of speed using a variable

We can further improve the program by using a *variable* to hold the delay value. At the moment, if you want to change the speed of the motor you have to go through and change all the values in the delay. Programs can contain *variables* which can be used to hold a value. You can think of a variable as a named box that holds a value that our program needs to remember. We decide the name of the box and the type of information that we want it to hold. Then we tell the compiler (the thing that converts our program into something that runs inside the Arduino) about the variable by *declaring* it. The compiler **must** be told about a variable before it is used in a program. We don't need to worry about precisely where the value is stored, that is sorted out for us automatically. The compiler generates the instructions that will fetch and store the contents of the variable as the program runs.

In the case of our motor program we will create a variable that will hold a value that represents the motor delay value for each delay when the motor is moving. It seems sensible to give such a variable a sensible name, perhaps `motorDelay`.

We can use whatever names we like for our variables in a program, subject to a couple of rules:

- A variable name must start with a letter
- A variable name can only contain letters, digits and the underscore (_) character

But apart from that, there are no rules. We could call the variable `fred`, or the name of our favourite soccer team. But we are not going to do this because it would be stupid. We are going to make sure that the variables names that we use all make sense in the context of the program being written, and actually convey information about what they are being used for. This is mainly for self-preservation, in that you'd be surprised how quickly you can forget what parts of a program are for. Sensible variable names make it much easier for me to understand code.

> You might be wondering why variable names are not allowed to start with a digit. It seems a rather arbitrary restriction, bearing in mind that we can use digits in the rest of the name.
>
> This is so that the compiler can easily tell the difference between a number (for example `20`) and a variable name (for example `sensor20`). If the thing the compiler is looking at starts with a letter the compiler knows that this must be a variable. If it starts with a digit the compiler knows it must be a number.

Take a look at this code:

```
int motorDelay;

void setup() {
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
  pinMode(6,OUTPUT);
  pinMode(7,OUTPUT);
  digitalWrite(4,HIGH);
  motorDelay=1200;
}

void loop() {
  digitalWrite(5,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(4,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(6,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(5,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(7,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(6,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(4,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(7,LOW);
  delayMicroseconds(motorDelay);
}
```

Ex05 TurnMotorVar

The variable we are using is called `motorDelay`. It has been *declared* at the top of the program, outside the `setup` and `loop` functions. It has been created as an integer variable, since that is the type of value that `delayMicroseconds` works with. Integer values don't have any fractional part which means that we can't ask for a delay of 10.5 microseconds. We have to go for values like 10 or 11.

In the `setup` function the value of `motorDelay` is set to 1200, and this is then used in each call of the `delayMicroseconds` function. This is neat because we can change the delays by just changing that one line of code.

```
motorDelay=2400;
```

This would halve the speed of the motor.

Run the program above and notice how you can change the speed of the motor by just changing the one statement.

Slowing down the motor

It might be fun to change the program so that the speed of the motor changes as it runs, slowing down. If you think about it, this involves changing the value of `motorDelay` and making it larger. This increases the delay between motor pulses and slows the motor down.

```
void loop() {
  motorDelay = motorDelay + 200;
  digitalWrite(5,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(4,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(6,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(5,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(7,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(6,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(4,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(7,LOW);
  delayMicroseconds(motorDelay);
}
```

`Ex06 MotorSlow`

Take a look at the line above that has been highlighted. This makes the value of `motorDelay` larger (by 200) each time that the `loop` function is called.

Create the program above (by modifying your previous one). Run the program. Notice anything strange?

Debugging the motor slowdown program

The title of this section doesn't imply much confidence with the possible outcome of our program. And it does seem that we have a problem to fix.

You would expect the wheel to get slower and slower as successive calls of `loop` make the value of `motorDelay` larger and larger, increasing the delay between each pulse. If you run the program you should notice that the wheel does slow down at first, but then it suddenly speeds up again. What's happening?

Well, what's happening is something specific to the way that programs work. (and something really worth knowing about). Computers don't actually hold numbers, instead they hold patterns of bits. A *bit* is an electrical signal that can be in one of two states. That means if you have a single bit you can hold two possible values. Useful for recording the result of a coin toss (heads or tails) but not much else.

Add another bit and you double the number of things you can store (now we can hold four different values). Each time we add another bit we double the number of possible values we can store. But, (and this is the important point) the number of bits that we end up using determines the range of values that can be stored in a computer. If we try to store a number that is outside this range, bad things will happen.

Our program will do just that. If you think about it, it is not sensible to keep increasing the value of a variable, any more than we can keep pouring water into a pint bottle. The difference is that if we add more water to a full bottle it will overflow and the bottle remains full, whereas with the `motorDelay` variable it seems that adding a value to it will make it smaller. This has to do with the way that the patterns of bits in the computer memory are mapped onto numeric values. It turns out that adding 1 to the largest possible positive value that the variable can hold with change the value of that variable to the largest possible negative value that the variable can hold. It is as if you pressed the accelerator pedal in your car a bit further down, and suddenly your car tried to go full speed in reverse.

When we write our programs, we need to be careful that we never exceed the capacity of the variables that we are using; particularly if our programs are controlling hardware.

What we really want is a way of stopping the wheel from moving when it has finished slowing down.

```
void loop() {

  if(motorDelay>15000)
    return;

  motorDelay = motorDelay + 200;

  digitalWrite(5,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(4,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(6,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(5,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(7,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(6,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(4,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(7,LOW);
  delayMicroseconds(motorDelay);
}
```

The program above is our first attempt to do this. It adds a conditional statement at the beginning of the `loop` function which checks the value of `motorDelay`. If this value is above a particular threshold (I've used the value 15,000) the `loop` method returns without driving the motor. The C++ return keyword means "Go home from this method, I've no desire to do anything else". So, the motor will stop running once it has slowed down.

Conditional statements

The new C++ element that is used in the above code is a conditional statement. This allows a program to "decide what to do" based on values in variables. A condition statement is expressed as the word `if` followed by something that is either true or false.

```
if(motorDelay>15000)
    return;
```

In the case of the above conditional statement we are testing to see if the value in `motorDelay` is greater than `15000`. If this is the case (i.e. the condition is true), the statement that follows the condition will be obeyed, causing the program to `return` from the `loop` function. The `return` statement tells the compiler that the program should not obey any more of the function. Since the statements that move the motors are all after this condition you should notice that the wheel stops moving once the `motorDelay` value reaches `15000`

We'll be using conditional statements throughout our programs to make decisions.

Make the above modification and run the program. You should notice that the wheel will slow down and stop.

The value of 15000 gives the largest delay that will be used before the wheel stops. You might like to experiment to find out what happens if you change this value.

You should notice that if the value is larger the wheel will be moving more slowly when it stops. If you're not sure what this means, try the program with a few different values.

Turning off the motors properly

The program above stops the wheel turning it has slowed down, but it does have a small bug. When the motor stops turning one of the stepper motor coils is left switched on. You can see this because one of the leds on the driver board stays lit.

This will not cause a problem for the motor, but it won't do very good things for the power consumption of our robot. It would cause the battery to go flat more quickly. My tests have shown that a single motor consumes 125 milliamps of current (around a tenth of an amp). If you use standard alkaline batteries you'd find that these would probably go flat after around 90 minutes if the motor is left in this state.

```
if(motorDelay>15000)
{
  digitalWrite(4,LOW);
  return;
}
```

`Ex07 MotorStopPower`

This code shows how we fix the problem. Before we return from the loop the program above turns off the coil that has been left switched on. If we do this the consumption of the robot drops to below 10 milliamps. Notice that I'm using a C++ trick here, so that I can control two program statements with a

single condition. The call of `digitalWrite` that turns the drive off and the `return` are both enclosed in a pair of curly brackets.

In the C++ language, a number of statements enclosed in these brackets is called a *block* and is treated as a single statement for the purpose of conditions. You've seen blocks in use before without really noticing it, in that the body of a function (i.e. the statements that make up the `setup` and `loop` methods) are actually blocks of code.

Is it a problem that the program is still running when the motor has stopped turning?

If you think about it, once the `motorDelay` value exceeds our stop threshold there is no point in the program being active any more. However, the program will continue running, even though there is no motor for it to drive.

In some situations, for example in mobile phones, the computer processor will actually stop when it is not being used, and there are special tricks you can use to put an Arduino "to sleep" if you really want to save as much power as possible, but as you can see from above, the motors consume so much power compared to the computer that for this application there is no need to do this.

If you do want your motors to move at a very low speed (for example to make a computer controlled clock), you can use the original `delay` function.

Driving both motors

We can now make one of the robot motors move. Making the second motor move is simply a matter of repeating the drive code but for the second set of coils. My first attempt looked like this:

```
int motorDelay;

void setup() {
  Serial.begin(1200);
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
  pinMode(6,OUTPUT);
  pinMode(7,OUTPUT);
  digitalWrite(4,HIGH);

  pinMode(8,OUTPUT);
  pinMode(9,OUTPUT);
  pinMode(10,OUTPUT);
  pinMode(11,OUTPUT);
  digitalWrite(8,HIGH);

  motorDelay=1200;
}

void loop() {
```

```
  digitalWrite(5,HIGH);
  digitalWrite(9,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(4,LOW);
  digitalWrite(8,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(6,HIGH);
  digitalWrite(10,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(5,LOW);
  digitalWrite(9,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(7,HIGH);
  digitalWrite(11,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(6,LOW);
  digitalWrite(10,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(4,HIGH);
  digitalWrite(8,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(7,LOW);
  digitalWrite(11,LOW);
  delayMicroseconds(motorDelay);
}
```

Ex08 BothMotors

To make the second motor turn I've just duplicated the first motor code and changed the pin numbers so that the coils in the second motor are driven as well as those in the first motor.

Do you think this will work? Why not?

It turns out that the program is fine. But the effect is not. Rather than moving forwards the robot will turn on the spot. This is because both motors are turning clockwise. But the motor on the opposite side of the robot must turn in the opposite direction.

And there is also the little matter of the fact that at the moment the wheels are driving the robot backwards…..

Run the example program (or alternatively type it in if you really want to suffer). Note that the wheels do go round the wrong way. Most annoying.

Fixing the motor movement

If you think about it (and I have) it turns out that the sequence of signals that we are sending to the motors is correct, but they are being sent in the wrong order. There are two ways we can fix this:

- We could reverse the order of the wires between the Arduino and the stepper motor driver board
- We could change the order that the program drives the pins

As a rule, I much prefer fixing software to fixing hardware, and I think it is useful if we have a hardware standard where a motor will always move the same way if the same signals are sent to it, so I thought about the best way to make it easy to control the direction of the motor. One way to do this is to use variables again.

```
int motorDelay;

byte left1,left2,left3,left4;

byte right1,right2,right3,right4;

void setup() {
  left1=7; left2=6; left3=5; left4=4;
  right1=8; right2=9; right3=10; right4=11;

  pinMode(left1,OUTPUT);
  pinMode(left2,OUTPUT);
  pinMode(left3,OUTPUT);
  pinMode(left4,OUTPUT);
  digitalWrite(left1,HIGH);

  pinMode(right1,OUTPUT);
  pinMode(right2,OUTPUT);
  pinMode(right3,OUTPUT);
  pinMode(right4,OUTPUT);
  digitalWrite(right1,HIGH);

  motorDelay=1200;
}

void loop() {

  digitalWrite(left2,HIGH);
  digitalWrite(right2,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left1,LOW);
  digitalWrite(right1,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(left3,HIGH);
  digitalWrite(right3,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left2,LOW);
  digitalWrite(right2,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(left4,HIGH);
```

```
  digitalWrite(right4,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left3,LOW);
  digitalWrite(right3,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(left1,HIGH);
  digitalWrite(right1,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left4,LOW);
  digitalWrite(right4,LOW);
  delayMicroseconds(motorDelay);
}
```

EX09 BothMotorsForward

Above you can see the idea. Rather than using the values 4,5,6, and 7 for the pin numbers for the left motor, I've instead used variables called `left1`, `left2`, `left3` and `left4`. I've done the same for the right-hand motor. In the `setup` function I set the variables to be the number of the pins that I want to control then use the variables in the code that moves the motors.

```
left1=7; left2=6; left3=5; left4=4;
right1=8; right2=9; right3=10; right4=11;
```

If you look carefully, you'll see that I've reversed the order of the pin numbers for the left-hand motor, so that it effectively turns "backwards" when I want to move forwards. The right-hand motor pins are numbered in ascending order, because I want the right-hand motor to turn clockwise to make the robot move forwards. However, the left-hand pin numbers are reversed because I want that motor to turn anti-clockwise when it moves.

You might be wondering why these variables are declared as `byte` variables, rather than `int`. The reason is that I'm trying to save memory. We know that the Arduino stores numbers as patterns of individual bits, where a bit can be either on or off. We also know that each time you add another bit to the pattern the number of values that can be represented will double. A group of 8 bits used in this way is called a *byte*. If you do the maths you'll work out that 8 bits can hold 256 different patterns of ons and offs. That's plenty for our pin numbers, which don't go above 11. However, an integer variable uses two bytes, so that a program can hold a larger range of possible values.

The Arduino doesn't have much space for variables. In fact it only has space for 2,000 bytes. An average computer today has 4 Gbytes (i.e. four thousand million) bytes of memory, which is two million times as much storage space.

I've learned to be very careful with memory when writing programs for the Arduino. It turns out that when an Arduino program runs out of space to store things it doesn't stop with a nice helpful message. It just goes off and does lots of crazy, random things. Which is almost always a bad thing.

Run the example program above. Note that the wheels now go around the correct way. What changes would you need to make to change the direction of the robot movement?

Controlling Motor Direction and Distance

At the moment, we can drive our motors forwards and this works well. However, we'd rather like to be able to control the direction of movement and the distance of movement, so now we can investigate how to do this.

Controlling Direction

Let's start by controlling direction. If use variables for our pin numbers we can change the direction the robot moves in a step by changing the pin numbers. We've already seen this in action in the previous program, where we reversed the direction of movement of one motor just by reversing the order of the pin numbers in the variables.

```
void leftForwards()
{
  left1=7; left2=6; left3=5; left4=4;
}

void leftReverse()
{
  left1=4; left2=5; left3=6; left4=7;
}
```

This snippet of a program declares two functions, one called `leftForwards` and the other called `leftReverse`. A function is a block of code with a name. We've already used two functions in our programs, the `setup` and `loop` elements of the program are both functions. We can call these methods to set the pin numbers and control the direction the motor will move when it is triggered. We could create similar methods to allow the direction of the right-hand motor to be controlled. Let's use these methods in a program that can move the robot forwards and backwards.

```
int motorDelay;

byte left1,left2,left3,left4;

byte right1,right2,right3,right4;

int moveCount; // counter for the number of steps

void leftForwards()
{
  left1=7; left2=6; left3=5; left4=4;
}

void leftReverse()
{
  left1=4; left2=5; left3=6; left4=7;
}

void rightForwards()
```

```
{
  right1=8; right2=9; right3=10; right4=11;
}

void rightReverse()
{
  right1=11; right2=10; right3=9; right4=8;
}

void setup() {

  leftForwards();
  rightForwards();

  pinMode(left1,OUTPUT);
  pinMode(left2,OUTPUT);
  pinMode(left3,OUTPUT);
  pinMode(left4,OUTPUT);
  digitalWrite(left1,HIGH);

  pinMode(right1,OUTPUT);
  pinMode(right2,OUTPUT);
  pinMode(right3,OUTPUT);
  pinMode(right4,OUTPUT);
  digitalWrite(right1,HIGH);

  motorDelay=1200;

  moveCount=0; // set the counter to 0
}


void loop() {

  moveCount = moveCount + 1; // increase the counter by 1

  if (moveCount==512) // is the counter 512?
  {
    leftReverse(); // if it is - reverse the movement
    rightReverse();
  }

  if(moveCount== 1024) // is the counter 1024?
  {
```

```
    leftForwards();// if it is - go forwards again…
    rightForwards();
    moveCount=0;//…  and put the counter back to 0
  }

  digitalWrite(left2,HIGH);
  digitalWrite(right2,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left1,LOW);
  digitalWrite(right1,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(left3,HIGH);
  digitalWrite(right3,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left2,LOW);
  digitalWrite(right2,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(left4,HIGH);
  digitalWrite(right4,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left3,LOW);
  digitalWrite(right3,LOW);
  delayMicroseconds(motorDelay);
  digitalWrite(left1,HIGH);
  digitalWrite(right1,HIGH);
  delayMicroseconds(motorDelay);
  digitalWrite(left4,LOW);
  digitalWrite(right4,LOW);
  delayMicroseconds(motorDelay);

}
EX10 MotorDirections
```

This is our most complicated program yet. It repays careful study.

Comments

The green comments in the text above do make the program a bit easier to understand. I've called them out in large green text for the listing above because I want you to be able to find them. However, it turns out that you can actually add your own comments to C++ programs. The rule is that whenever the compiler sees the characters // (two forward slashes) in a program it completely ignores what follows on the line. If you take a look at the example code you'll find the comment text on the end of the lines. You should form a habit of putting comments in your program. If you want to write a block of text as a comment you can enclose the text in the /* and */ characters:

```
/*
 Direction control demonstration

Rob Miles

Move the stepper motors 512 steps forwards
and then 512 steps back

April 2017
Version 1.0

 */
```

This the heading for my program. The compiler will ignore all this text because it is enclosed in comment start and end characters.

Motor motion control

The idea is that I want my robot to move forwards and backwards. The program uses a counter that counts how many times the motors have stepped. When the counter reaches 512 the direction of the motors is set to reverse. When the counter reaches 1024 the direction of the motors is set to forward and the counter is reset to zero. This will cause the whole process to repeat.

Run the example program above. Note that it works.

Consider the following questions, and try and provide an answer for each.

1. What would I change to make the robot move twice as far forwards as backwards?
2. What would I change to make the robot rotate rather than move forwards or backwards?

Controlling distance

You may be wondering about the significance of the value 512 in the context of the program above. If you are still wondering, take a very careful look at the amount each wheel turns when the program is running. You should find that each wheel makes exactly one revolution in each direction. And re-read the section “The need for speed” above. This explains why the motor will turn 1 revolution for every 512 pulse sequences. We can use this behaviour to create very precise movements of the robot.

The diameter of the robot wheel, with its rubber band tyre fitted, is around 68.5 or so mm. The circumference of a circle is pi times the diameter, which means that each time the wheel goes around the robot will travel pi times 68.5mm or 215.2 mm.

We can now work out the distance that the robot will move per step simply by dividing 215.5 by 512. The answer is around 0.4 mm per step.

So, if I wanted the robot to move 10mm I’d move 10 / 0.4 steps, or 25 steps. Here’s how I’d put this into code:

```
float wheelDiameter = 68.5;
float stepsPerRevolution = 512;
float mmsPerStep = (wheelDiameter * 3.1416) / stepsPerRevolution;
```

I've set up three variables. The variable `wheelDiameter` is set with the diameter of the robot wheel. The variable `stepsPerRevolution` contains how many steps are required to make one turn of the wheel. The final variable, `mmsPerStep` is calculated from these two variables using the equation that I've just described above. I've written the program out like this because it is much easier to understand what is going on. It's also very obvious to the reader how we could change the program to handle a different sized wheel.

Floating point variables

The code above uses a new kind of C++ variable, called `float`. The program uses a `float` variable because it wants to perform calculations that involve fractions. It turns out that the number of mms that the robot will move per motor per step is less than 1 (around 0.4). We can't use an integer variable to hold this value because integers can't hold fractional values. C++ provides a storage type called float that can hold values that are going to need fractions. You might wonder why we don't' use floating point values for everything, to keep things simple. There are two reasons for this:

1. The Arduino has to work a bit harder to perform floating point maths, which can slow down our program
2. Floating point numbers might end up using more memory than an int or byte. And we worry a lot about memory when we are writing Arduino programs.

Calculating steps

The next thing we can do is write a function that will calculate the steps required for a particular distance.

```
int calculateDistanceSteps(float distanceInMM)
{
  return distanceInMM / mmsPerStep + 0.5;
}
```

`EX11 PreciseMovement`

This function is called `calculateDistanceSteps`. It is different from the ones we've seen before, in that it returns a value. A C++ function can be given a type (in this case int) and return a value (in this case the calculation after the word `return`). When the method runs the value of `distanceInMM` is passed into the call of the method and then used in the calculation. The result of the calculation is then returned by the method as an integer. This makes sense because we the program can't make the motors move in fractions of steps. A program can use the method as follows:

```
int moveSteps;
moveSteps = calculateDistanceSteps(20);
```

The above code would set the value of the `moveSteps` variable to the number of steps required to move the robot 20 mm.

Floating point and integer values

You might be wondering how the floating point and integer values are used in the code above. The `calculateSteps` method uses floating point variables and then returns an integer, and, to make things even more confusing, it adds 0.5 to the calculated steps value before returning the value to the caller. What's going on?

In C++ it's perfectly OK to put a floating point value (which has a fractional part) in an integer variable (which has no fractional part). When you assign a floating point value to an integer variable the fractional part of the floating point value is discarded. In other words, if a program puts the value 1.99 into an integer the integer is given the value of 1. This is a bad thing for our program, in that it means that the robot will not move as many steps as it should to get the most accurate position. To solve the problem we just add 0.5 to the value before C++ converts it into an integer. The value 1.99 would be converted into 2.49, which would be truncated to 2. Problem solved. You need to be aware of this issue when you are working with floating point and integer values.

Challenge: Turning the robot

According to my measurements the wheels on the Hull Pixelbot are around 90mm apart. This means that if one wheel moves 90*Pi mm clockwise and the other wheel moves 90*Pi mm anti-clockwise the robot should turn rotate on the spot. We can use this to create code that we can use to precisely turn the robot.

Use what you have learned to perform the following challenges:

1. Create a program that will make the robot rotate once on its axis.
2. **Grand Challenge:** Write a method called `calculateAngleSteps` that takes in an angle in the range 0 to 360 and calculates how many steps the motors must turn to make the robot rotate that number of degrees clockwise. To win a bonus prize you could try to make the robot rotate anti-clockwise if the angle is given as negative.

Avoiding collisions

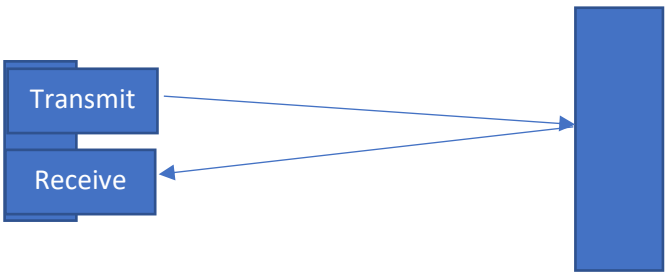
What you will do in this chapter

After the previous chapter we can drive the robot and turn it. Now we are going to add a sensor that we can use to allow the robot to react to surroundings. You'll find out how we can make a robot "see" what is in front of it. You'll fit a distance sensor to the robot that will allow it to detect obstacles before it hits them. You'll then create some software that will use the distance information to make the robot able to perform simple navigation.

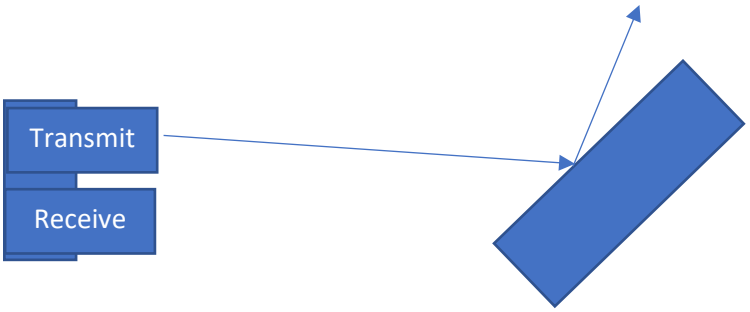
You'll finish off with some challenges where you can show how well your programs can control the movement of your robot.

Distance sensors and robots

Detecting things in front of us is something that humans do very well indeed (although we do spend the first few years of our lives working out how to do it). Making robots do the same thing turns out to be quite tricky. There are a variety of different ways that a robot can detect obstacles. We are going to use a sensor which uses high frequency sound waves. The sensor sends out a high frequency (ultrasonic) sound pulse and detects when the sound pulse has returned.



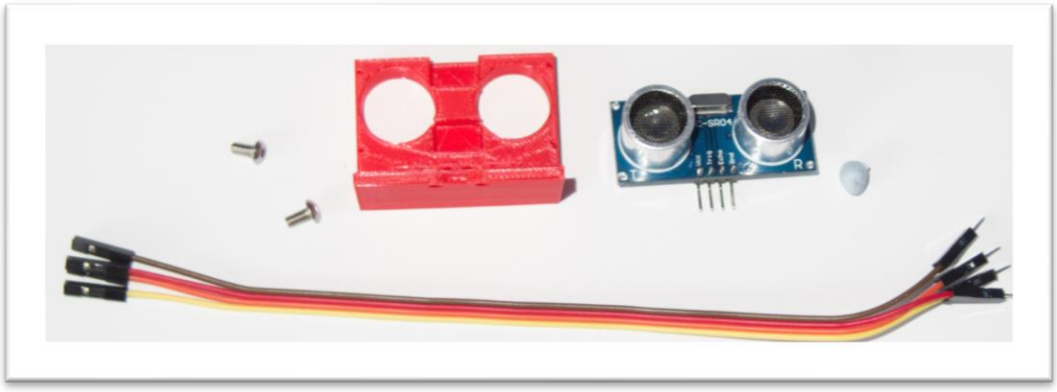
Above you can see how this works. The technique is used in nature, it's how bats can navigate in darkness.



It's not a perfect technique. It works well if the object in the way is directly in front of the sensor, but if it is at an angle the transmitted sound "bounces off" the object and away from the sensor, so the return sound never arrives and the obstacle is not detected. You'll see this when you start using the robot. The best way to fix the problem is to have multiple sensors around the front of the object, each at a different angle.

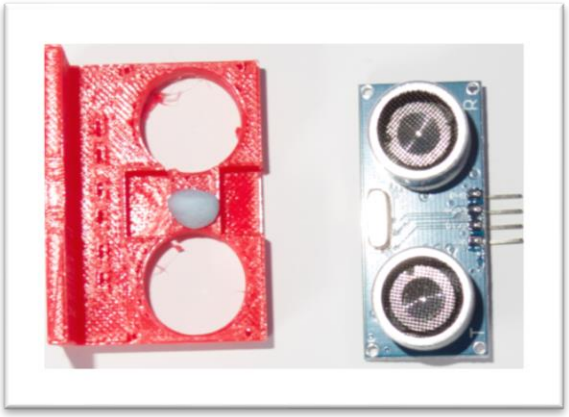
The good news is that the sensor we are going to use is very cheap, you can obtain them for less than a pound via the internet. So, it is perfectly possible to use a lot of them. We are going to start with just one distance sensor for now.

What you will need



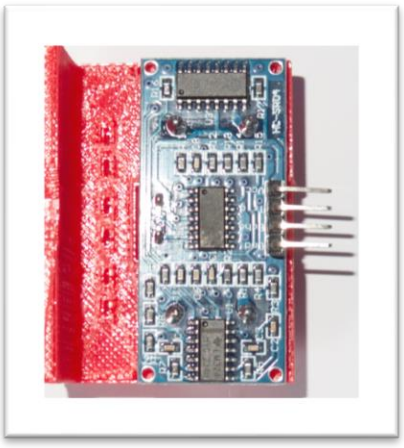
For this exercise you'll need the distance sensor, two 6mm bolts (the smallest) and the holder that attaches to the front of the robot. You'll also need a tiny blob of blu-tack (or other sticky adhesive type thing). Finally, you'll need a four-way Male to Female cable.

Fitting the sensor



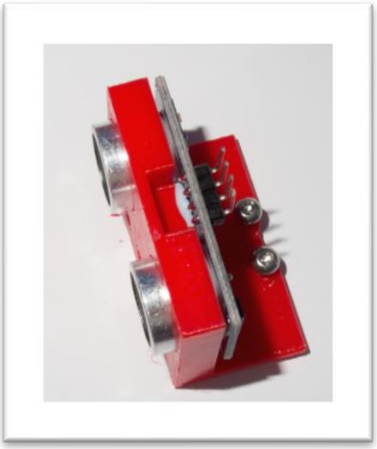
I'm using blu-tak to hold the distance sensor onto the holder because it provides a slightly more damped connection to the robot that simply screwing the sensor into place. I've done this because the sensor can be slightly sensitive to vibrations, and this form of fitting helps a bit.

Start by rolling a tiny ball of blu-tack and placing it on the holder, as shown above.

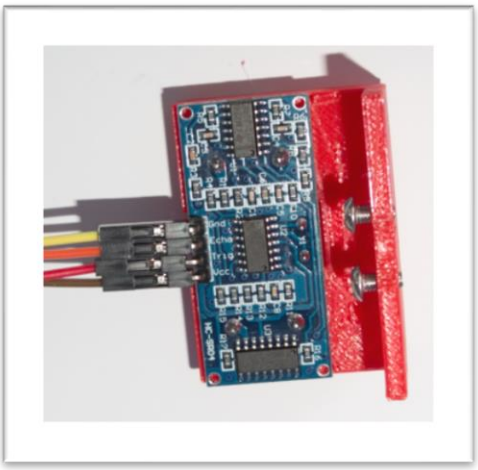


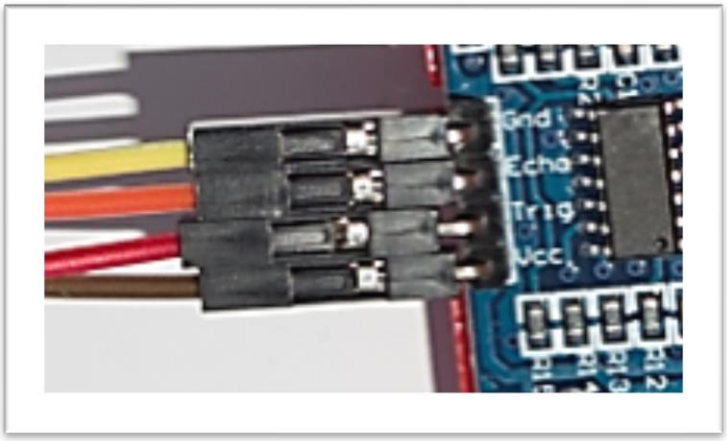
Now take the distance sensor and press it into the holes (not too hard) so that the sensor is held firmly. Make sure that the connectors are on the top of the holder, as shown in the picture.

Now we can fix the sensor assembly to the front of the robot. You'll need to use 6mm bolts for this.

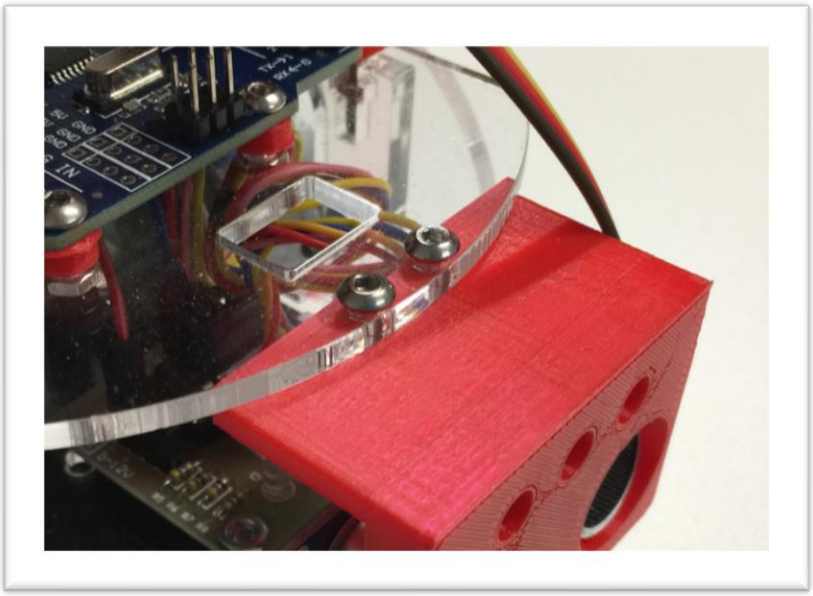


Screw the 6mm bolts into the holder to get them started. They are a fairly tight fit, but they will go in OK.





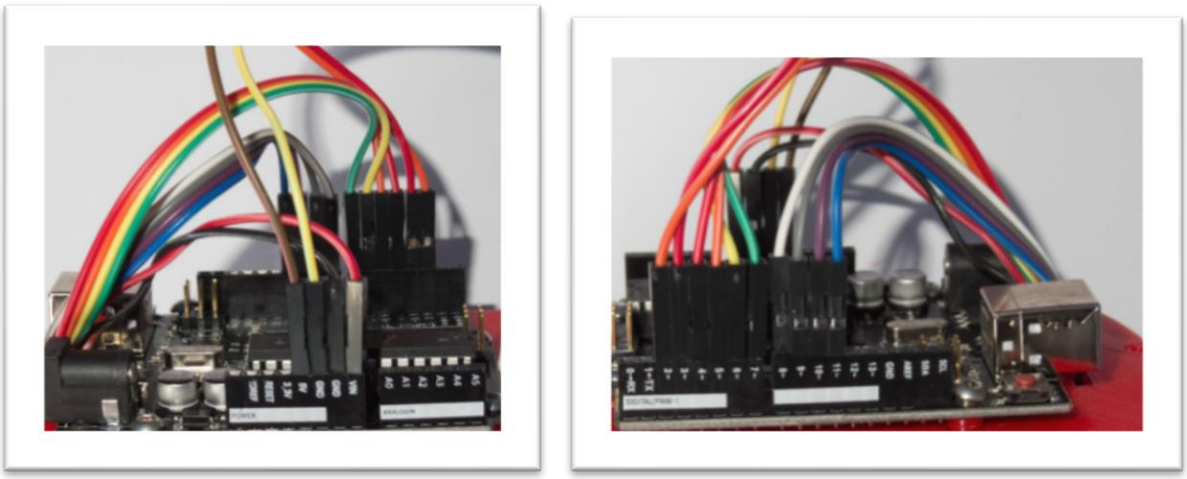
Just before you attach the holder to the robot, you can connect the cables to the pins. We need four connections.



Now fit the distance sensor to the front of the robot as shown. Use the 6mm bolts in the holes in the top plate

Now we need to make the connections

Cable	Arduino Connection
Gnd - Power Ground (yellow)	GND
Vcc - Power Live (brown)	5 volts
Trig - Sensor trigger (red)	D3
Echo - Sensor echo (orange)	D2



Connect the cables as shown above. The two outer wires go to the power side of the Arduino and the two inner wires go to the data pins.

We now have a distance sensor connected to our Arduino. Now we need to discover a way that we can use this from software, and view the readings it produces. To do this we are going to take a slight digression into Arduino serial communications.

Using the Serial Port to Move Messages between an Arduino and PC

The Arduino device is connected to the host computer by means of a USB connection. The “Universal Serial Bus” platform can support a wide range of physical devices including mice, keyboards, storage devices and a serial communications mechanism often referred to as a “com” port. The word “com” is an abbreviation of “communication”.

A com port sends an eight-bit item of data (a byte) down a single communications channel one bit at a time. The data is sent out at a particular rate, called the “baud rate”. It is important that the sender and the receiver of the data agree on this data rate, otherwise the data will be garbled. The baud rate gives the number of bits that are to be sent each second. To calculate how many data bytes can be sent per second you can divide the baud rate by 10 rather than 8 (the number of bits in a byte). This allows for extra bits that are sent to signal the start of a data byte.

What the data in the byte means is up to your software. It could be a single character of an ASCII text message, a number that represents an analogue voltage or part of a larger structure. The Ardiuno library provides methods that your program can use to send messages via the serial port.

Setting Up the Serial Port

A program sets up the serial port in the `setup` function. This is called once when the program starts running.

```
void setup() {
  Serial.begin(9600); // Open the serial port and set the
                      // baud rate to 9600
}
```

The `Serial.begin` method is called to set the baud rate of the connection. The rates which can be used include:

```
1200, 2400, 4800, 9600, 19200,38400,57600,115200
```

The Arduino can handle very high speed communications, but it can't store much data. You need to make sure that the receiver of the serial data from the Arduino can handle the incoming data too. The slower speed of 9600 is frequently used by serial devices such as GPS receivers and Bluetooth adapters.

Sending data out of the Arduino

The Arduino library provides functions that can send values out of the serial port. The print and println functions can be used to send text messages to the remote device (which is usually the host computer).

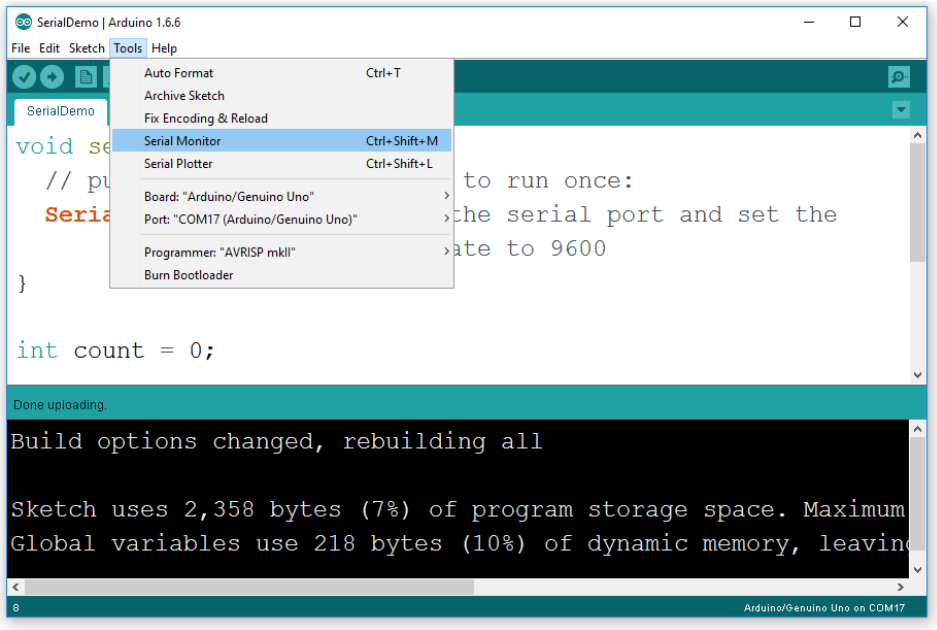
```
int count = 0;

void loop() {
  Serial.print("The count is : ");
  Serial.println(count);
  count = count + 1;
  delay(1000);
}
```

`EX12 SerialDemo`

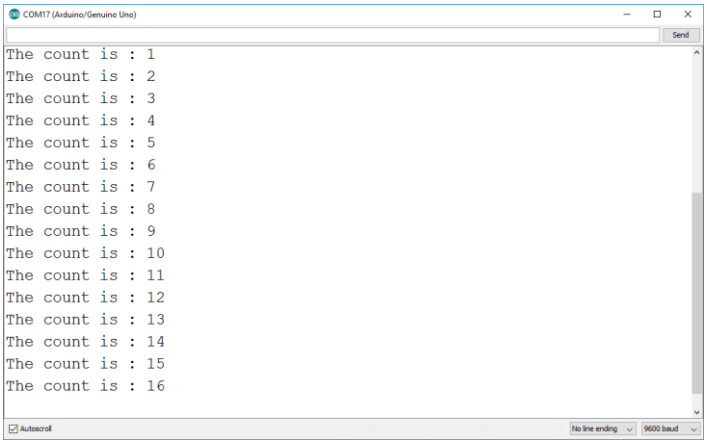
This program prints out a message which is updated each second. The `print` function prints without taking a newline. You can use it to assemble a line of text which you then complete with the `println` function.

Receiving the messages on the PC

The Arduino development environment also provides a program that can connect to the serial port and send and receive messages. The program is called the Serial Monitor and you can find it in the Tools menu:



When you select the Serial monitor the program starts and resets the Arduino, so that the first thing you see will always be the program starting up.



You can use the combo-box on the bottom right of the window to select the baud rate that is to be used. This must match the baud rate set by the `Serial.begin` method in the Arduino. You can also control how the line endings are displayed and whether or not the screen will scroll as further input is received.

Run the program and open the terminal window to see the messages being sent from the Arduino.

Sending Messages to the Arduino Board from the PC

It is also possible to send messages from the PC to the Arduino. To do this you can type text into the text entry field and press the Send button.

Receiving Messages on the Arduino

The `Serial` object can be used to read bytes from the serial port. The `available` method returns the number of bytes that are available to be read and the `read` method returns the next character that was received.

```
void loop() {
  if(Serial.available() > 0)
  {
    byte b = Serial.read();
    Serial.print("Just received:");
    Serial.println(b);
  }
}
```

`EX13 SerialFromPC`

The above version of the loop method checks for incoming characters and prints them back out to the Serial port. The Arduino will store a limited number of incoming bytes but if the program takes too long to get around to reading them they will be missed. A program on the PC can send commands to the Arduino in this way.

Run the program and open the terminal window to see the messages being sent from the Arduino.

Characters and messages

When you ran the program you would notice that when you input a letter, the program displayed a number. This is because the serial data is transferred in the form of eight bit values (in the range 0-255). To see value received as a character you can use a cast to convert a value into a character:

```
void loop() {
  if(Serial.available() > 0)
  {
    byte b = Serial.read();
    Serial.print("Just received:");
    Serial.println((char)b);
  }
}
```

`EX14 SerialAsChar`

By placing `(char)` before the value that is to be printed the C program in the Arduino will print the value as a character, rather than a numeric value. The characters in an Arduino program are mapped onto particular values using the ASCII character code.

Run the modified version and note that the characters themselves are now displayed.

Code Instrumentation and conditional compilation

Code instrumentation is the posh phrase for “putting in print statements”. When you are creating Arduino programs you can add a print statement at the start of each function so that when the method runs it gives you a log of execution.

```
void buttonPressed()
{
  Serial.println("Entered buttonPressed");
}
```

Each time that the function runs it will print a message. However, adding lots of print statements can make a program larger, and the program space in the Arduino is limited. Each print statement takes a while to complete and will slow a program down. What you want is a way of turning these statements on and off.

You can do this by using the C feature called *conditional compilation*. This lets you mark elements of code to be discarded when the program is built. Conditional compilation uses the pre-processor which is part of the C compiler. The pre-processor is the part of the compiler that reads the program code form the source code file and passes it on for compilation. A program can contain commands to the pre-processor. All macro commands start with a # character and are given on a single line. We’ll consider more of these later, for now we are going to take a look at the #define command, which is used to define a symbol for use by the pre-processor.

```
#define FUNCTION_TRACKING
```

Once we have a symbol we can then use it to control the compilation of code in the program source file:

```
void buttonPressed()
{
#ifdef FUNCTION_TRACKING
  Serial.println("Button pressed");
#endif
}
```

The `#ifdef` and `#endif` directives tell the pre-processor to only pass the code they contain to the compiler if the given symbol is defined. So we can completely remove the print statement from the program by removing the symbol definition.

A program can contain many different symbols so you can selectively enable or disable different program elements very easily.

An important point to remember is that the conditional compilation directive does not cause the selection of statements at runtime. Instead the statements are never even passed to the compiler at all.

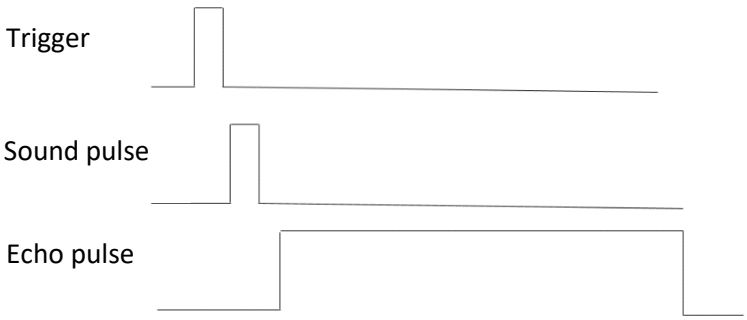
Use conditional compilation to perform selective code instrumentation in one of the earlier programs that you have written. You should be able to enable and disable tracking message produced when functions are called.

Reading the distance sensor

Now that we have a way of viewing the results, we can write some code to read the distance sensor. The distance sensor contains a tiny micro-controller which we can send commands to. When the distance sensor is triggered it makes a sound and then times how long it takes for that sound to echo. Because we know the speed of sound we can convert the time into distance.

Distance sensor signals

The sequence of reading the sensor is as follows:



The Arduino sends the distance sensor a trigger to say “I’d like to take a distance reading please”. The distance sensor sends a pulse of ultrasonic sound out of its speaker.

Once the sound has been produced the distance sensor then lifts the echo pulse signal. It then drops this signal when the microphone on the distance sensor receives the pulse echo.

Using the distance sensor from software

By timing the length of the echo pulse a program can work out the distance to the object that produced the reflection. The good news is that the Arduino has a function specifically designed to measure the length of a pulse.

```
const int trigPin = 3;       // trigger pin for distance
const int echoPin = 2;       // echo pin for distance


void setup() {
  pinMode(trigPin, OUTPUT);  // make the trigger an output
  pinMode(echoPin, INPUT);   // make the echo an input
  Serial.begin(9600);        // start up the serial port
}

void loop() {
  float duration;
  digitalWrite(trigPin, LOW);  // drop the trigger pulse
  delayMicroseconds(2);        // wait a tiny amount
  digitalWrite(trigPin, HIGH); // raise the trigger pulse
  delayMicroseconds(10);       // want a pulse width of 10 microseconds
  digitalWrite(trigPin, LOW);  // drop the trigger pulse
  duration = pulseIn(echoPin, HIGH); // measure the pulse
  Serial.println(duration);          // print out the distance
  delay(500);                        // wait half a second
}
```

EX15 DistanceReader

The program above reads the distance sensor every half a second and displays the time produced by the distance sensor each time. The really clever part is the `pulseIn` function, which we didn't have to write. It returns a result in microseconds which give the length of the pulse that it measured. The bigger the number, the longer the time taken for the sound to bounce off the obstacle.

Run the program above. Open the terminal window to take a look at the distance values that are coming back from the sensor. If you move your hand in front of the sensor you should see the values change.

Converting time into distance

We can use the distance sensor to get the time in microseconds that it took for the sound pulse to be echoed back to the sensor. To convert this time into a distance we need to know the speed of sound. Apparently, it takes sound 29.1 microseconds for sound to travel a single centimeter. So, if we divide the time value by 29.1 this should give us a distance value in centimeters.

```
void loop() {
  float duration;
  float distance;
  digitalWrite(trigPin, LOW);  // drop the trigger pulse
  delayMicroseconds(2);        // wait a tiny amount
  digitalWrite(trigPin, HIGH); // raise the trigger pulse
  delayMicroseconds(10);       // want a pulse width of 10 microseconds
  digitalWrite(trigPin, LOW);  // drop the trigger pulse
  duration = pulseIn(echoPin, HIGH); // measure the pulse
  distance = (duration/2.0)/29.1;    // calculate the distance
  Serial.println(distance);          // print out the distance
  delay(500);                        // wait half a second
}
```

EX16 DistanceMeasure

Run the program above. You should be able to measure distance using this program.

You might be wondering why we divide the duration time by 2 in the above code. This is because the duration is the time it takes the sound pulse to travel both to and from the obstacle. You might be able to think of a simplification of this statement to make the code run slightly faster.

Creating a `readDistance` function

We can create a CD++ function that reads the distance for us. We can then use this anywhere in our program.

```
const int trigPin = 3;       // trigger pin for distance
const int echoPin = 2;       // echo pin for distance

float readDistance()
{
  float duration;
  float distance;
  digitalWrite(trigPin, LOW);  // drop the trigger pulse
  delayMicroseconds(2);        // wait a tiny amount
  digitalWrite(trigPin, HIGH); // raise the trigger pulse
  delayMicroseconds(10);       // want a pulse width of 10 microseconds
  digitalWrite(trigPin, LOW);  // drop the trigger pulse
  duration = pulseIn(echoPin, HIGH); // measure the pulse
  distance = (duration/2.0)/29.1;    // calculate the distance
  return distance;
}

void setup() {
  pinMode(trigPin, OUTPUT);  // make the trigger an output
  pinMode(echoPin, INPUT);   // make the echo an input
```

```
  Serial.begin(9600);        // start up the serial port
}

void loop() {
  float distance = readDistance();
  Serial.println(distance);          // print out the distance
  delay(500);                        // wait half a second
}
```

EX17 ReadDistanceFunction

A program can use the `readDistance` function any time it wants to know how far the robot is from an obstacle. Remember that for the function to work the values of `trigPin` and `echoPin` must have been set correctly and the functions of the two pins must have been set up.

Challenge: Environment Interaction

You now have all the tools that you need to be able to create a robot that will avoid crashing into things. You can use the motor control code that you wrote in previous chapters, along with the distance sensor reading code above to allow you to make a robot that will avoid things.

Use what you have learned to perform the following challenges:

1. Create a program that will make the robot move forwards until it sees an object.
2. Change the program so that the robot dances around the object, perhaps by moving away and back, and maybe turning as well. Prizes for the best dance moves.
3. Create a program that will follow something which moves in front of it. The code should try to make the robot stay a set distance from the object.
4. Create a robot that can be stopped and started by waving your hand in front of it. This raises the prospect of lots of gesture controlled devices.
5. Create an escaping behaviour, where the robot rotates on the spot trying to find the direction with the greatest distance of movement, and then heads off in that direction. For best results you need to make the robot rotate 360 degrees and remember the rotation angle at which the greatest distance was found.
6. **Grand Challenge:** Write a program that attempts to move forwards and will try to go around an obstacle. It can do this by turning moving across until it things it has got past the object, and turning back and checking there is nothing in the way. You can use the repeatable nature of the robot positioning to good effect here; generally speaking; your robot will always move and turn a precise distance.

Challenge: Remote control

Since you have a serial connection to your robot, you might like to have a go at controlling it from the PC.

Use what you have learned to perform the following challenges:

1. Create a wired remote controlled robot. You can send characters from your PC to the robot via the serial port, so why not make a program that you can use to steer the robot around. You might need a longer usb cable for this…
2. You could expand your remote control to produce something that receives data from the robot distance sensor, to use it as some kind of remote controlled mapping tool.
3. Remember that this technology also provides a neat way that you can get things moving from your PC. Any task that can be motorised can now be controlled by your PC.