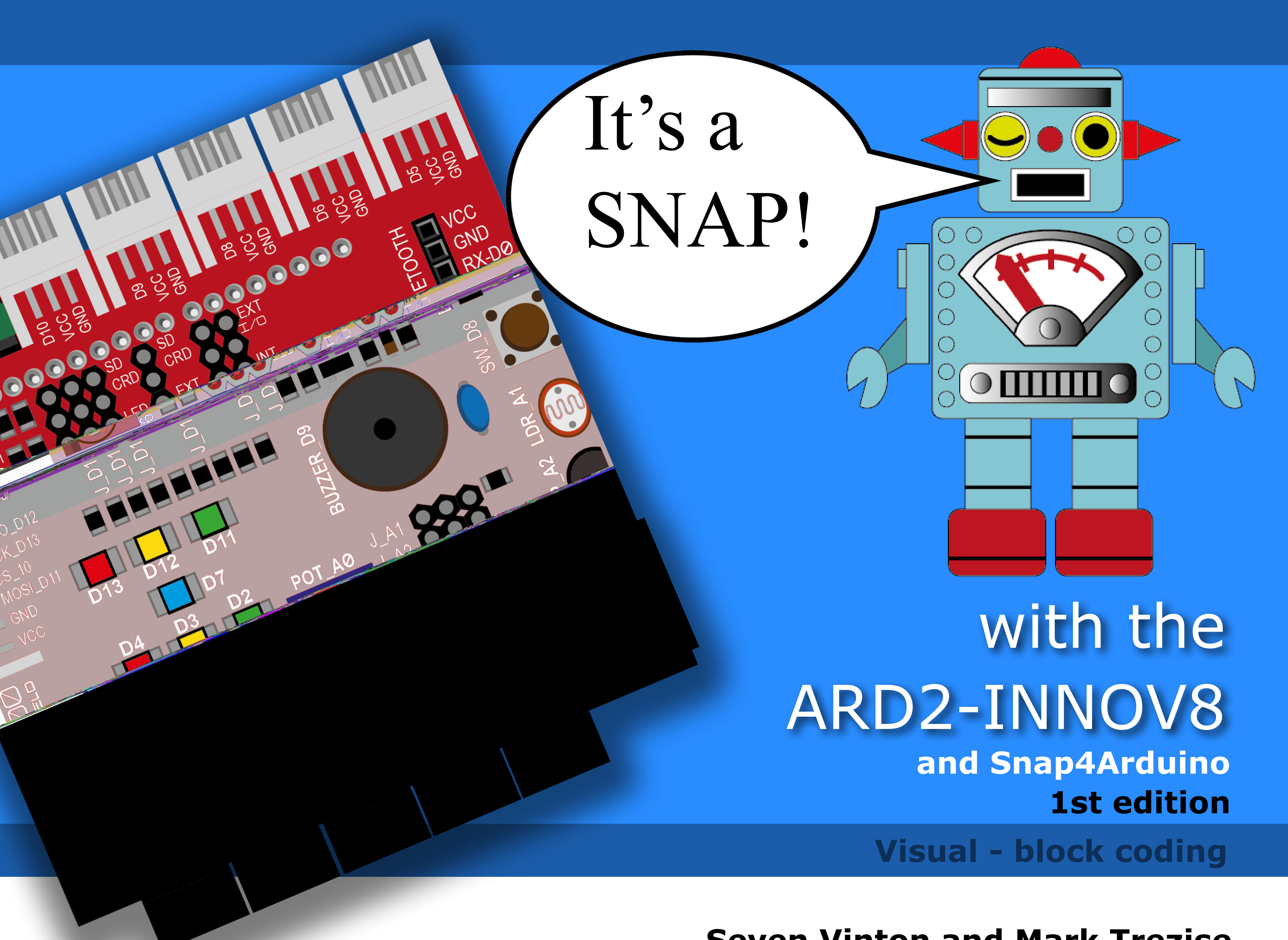
01001100 01100101
01110100 00100000
01110100 01101000
01100101 00100000
01100110 01110101
01110100 01110101
01110010 01100101
00100000 01110100
01100101 01101100
01101100 00100000
01110100 01101000
01100101 001000
01110100 01
01110101 011
01101000 0010
00100000 01100
01101110 011
00100000 011
01110110 0110
01101100 01110
01100001 01110100
01100101 00100000
01100101 01100001
01100011 01101000
00100000 01101111
01101110 01100101

Coding Innov8ion



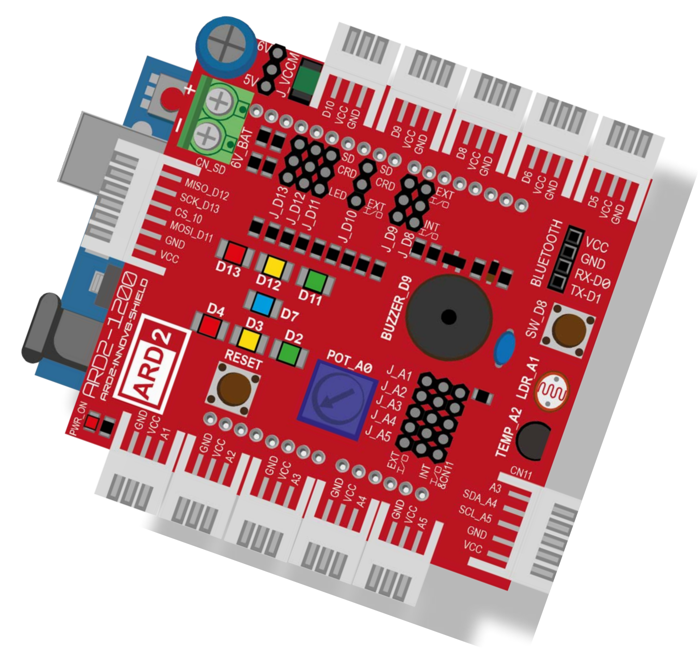
with the ARD2-INNOV8

and Snap4Arduino

1st edition

Visual - block coding

Seven Vinton and Mark Trezise



Coding Innov8ion, it's a Snap!

with the ARD2- INNOV8 and Snap4Arduino

Visual-block programming guide

Coding INNOV8ion it's a Snap! – with the ARD2-INNOV8 and Snap4Arduino - visual-block programming guide by Seven Vinton and Mark Trezise. A self-published eBook.
© 2017 Copyright Seven Vinton & Mark Trezise
All rights reserved. No portion of this book may be reproduced in any form without permission from the publisher, except as permitted under the Copyright Act 1968.
The code and video materials associated with this publication may be shared as long as this book and its authors are acknowledged.
Educational organisations may purchase an annual site licence for the use and for the copying of this book for the intention of supporting learning within classes. For more information and to purchase an "Annual Site Licence" go to www.wiltronics.com.au/ard2
Illustrations by Seven Vinton.
Some of the photographs in this book were obtained under the creative commons CC0 1.0 Universal (CC0 1.0).

"Arduino" is a trademark of Arduino.cc
"ARD2-INNOV8" is a trademark of Wiltronics Research Pty Ltd.

ISBN: 978-0-6480792-0-0

Acknowledgements

Thank you to Joan Guillén i Pelegay and Bernat Romagosa (Snap4Arduino Developer) for their help and assistance with Snap4Arduino code blocks.

Thank you to Ethan Zerafa (aged 11) for testing the book and the code.

Thank you to Rain Richardson for proofreading.

Firebugs Educational Resources

Disclaimer

The information provided in this book is for educational purposes only. Whilst great care has been taken in the preparation of the contents of this book, the authors of this book and the ARD2-INNOV8 team take no liability or responsibility for any errors or omissions, or any loss or damage resulting from the use of any information provided in this book or any of the linked websites or online video tutorials. Any use of the information provided is at your own risk.
There are no warranties, expressed or implied, about the accuracy or reliability of the information provided, or as to the availability and suitability of any of the technologies discussed in this book, as this will vary from case to case.

Content Index

Contents

Section A - Getting Started

What is the INNOV8 Shield? 4
The ARD2-INNOV8 story 5
What is Arduino and why do I need it? 5
Why use the ARD2-INNOV8? 6
Getting started with your ARD2-INNOV8 7
Using Snap4Arduino 8
How to set up Snap4Arduino on your computer 9

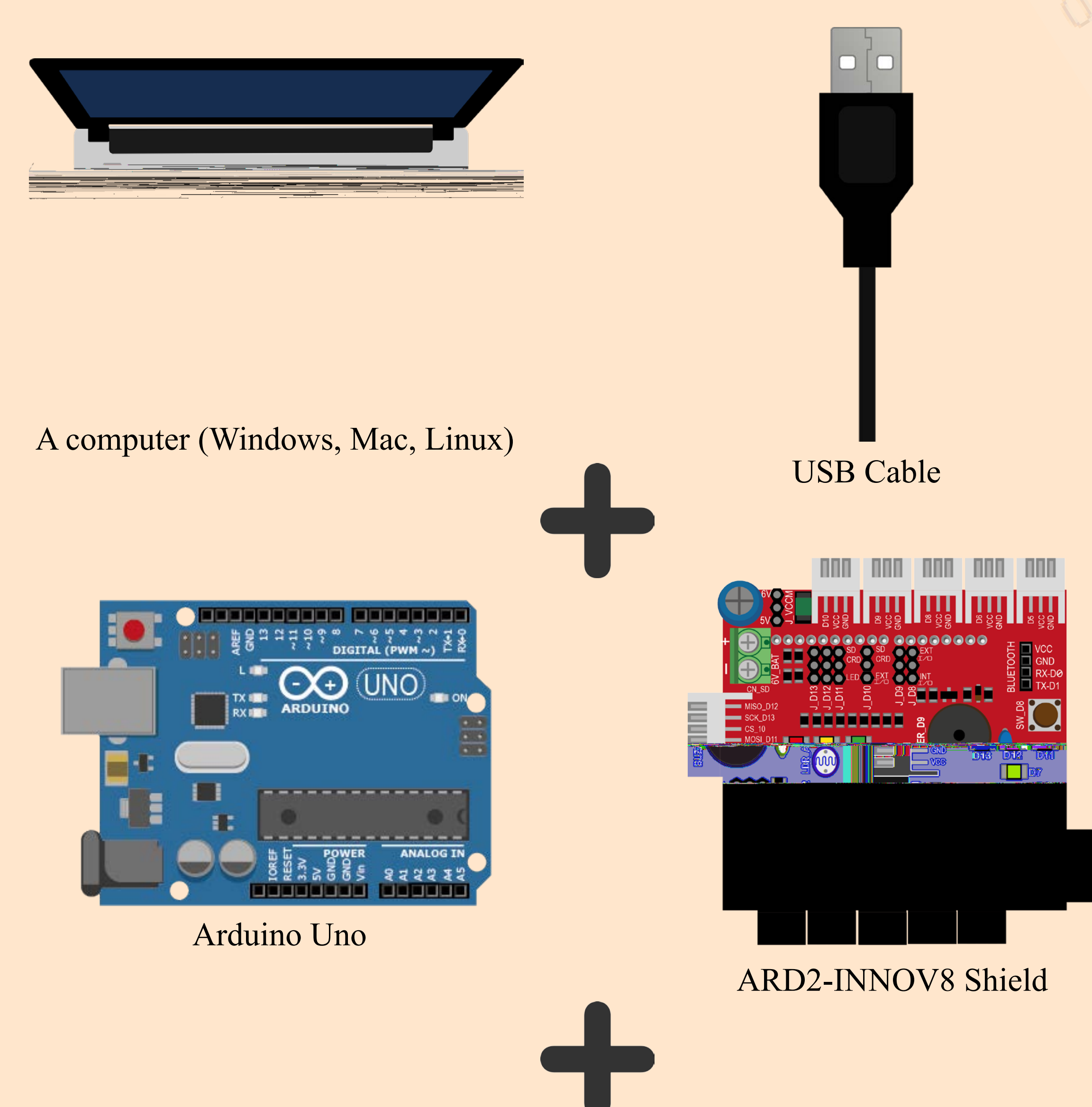
Section B - ARD2-INNOV8 Programming Lessons

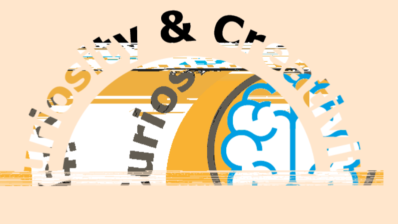
About the lessons in this book – note to teacher 11
Loading the firmware 11
Some basic electronics 13
Lesson 1 - Blink – Your first sketch 14
QuickBlocks - Quick and easy code 18
Lesson 2 – Using variables 20
Lesson 3 - Using control structures to simplify code 24
Lesson 4 – Introducing chance 27
Lesson 5 – Dice 30
Lesson 6 - Variable Tones 33
Lesson 7 – Banana Keyboard 36
Lesson 8 - Temperature Sensor 39
Lesson 9 - Traffic lights 41
Lesson 10 – Using servo motors 47
Final Challenge – Putting it all together 50

Section C - Reference

Double traffic light solution 55
Glossary of terms 56
Author information 58

Here's what you need to start programming with the ARD2-INNOV8





Content Index

What is the INNOV8 Shield?

The ARD2-INNOV8 shield is a piece of technology which has various *outputs* and *inputs*, and sits on top of an *Arduino Uno* compatible *micro-controller* board (other compatible boards can be found here: http://playground.arduino.cc/Main/SimilarBoards)
The ARD2-INNOV8 shield has been fitted with 7 LEDs (light emitting diodes), a piezo buzzer, a bush button switch, a potentiometer, a light sensor, a temperature sensor, a 6 volt input, and convenient snap in connectors to use with a wide range of external modules such as servo motors, joystick module, and a real-time-clock module.
The ARD2-INNOV8 shield was designed to take the headache out of learning how to program a microcontroller like the Arduino Uno. It allows the user to learn a wide range of Arduino lessons without the hassle of wiring up all of the required components. The ARD2-INNOV8 shield helps make learning Arduino programming easy, by allowing the user to concentrate on the programming without being distracted by bugs in the circuit.

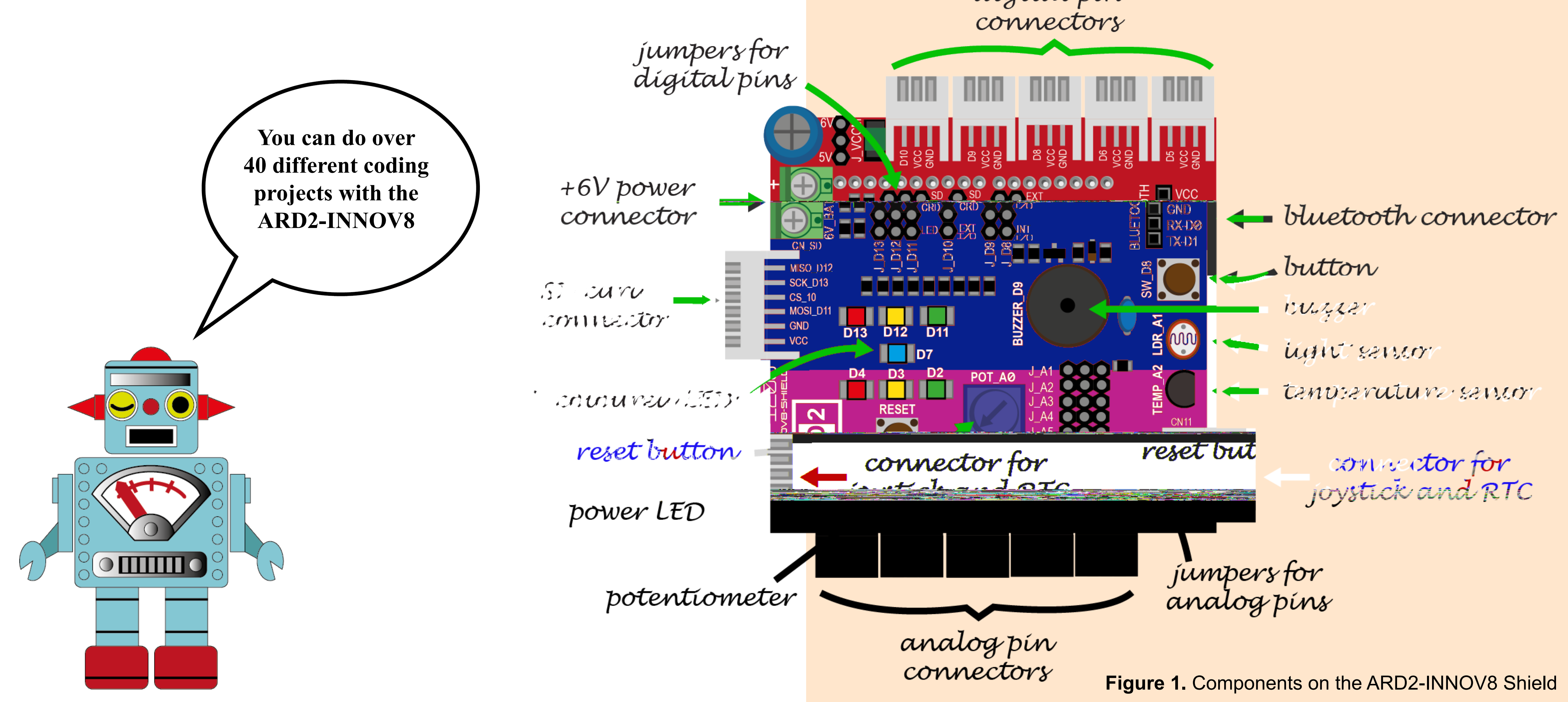


Figure 1. Components on the ARD2-INNOV8 Shield

The ARD2-INNOV8 story

The ARD2-INNOV8 story is a story of innovation and collaboration; it is a story of perseverance, life-long learning, and community.
The two authors of this book Seven Vinton and Mark Trezise are both technology enthusiasts. Their passion for learning inspired them to form a partnership to help make this learning available to others via online videos, learning resources, and learning seminars & expos.
During one of these student seminars Seven and Mark started to discuss ways to make Arduino programming easier for students. They had used many devices which aimed to make Arduino programming easier, however, all of these devices fell short in one way or another.
Seven and Mark met with the CEO of *Wiltronics*, Richard Wilson in December of 2016 to discuss how they could work together to promote learning of technologies like the Arduino devices, and at this meeting they aired their frustrations about the failings of the available technologies. Richard's answer was simple, "Let's make our own!"
From this meeting the prototype design for the ARD2-INNOV8 shield was developed by Seven and Mark along with its name, and upon receiving these designs Richard enlisted the help of his colleague Ben Sieira at *CognetronicS* to design the wiring layout and electronics engineering for the shield.
As previously mentioned, this story is a story of collaboration, however, this story of collaboration is a little different from other stories you may have heard, because Seven is a secondary school teacher, and Mark is a student (year 10 at the time). This collaboration between teacher, student, and industry experts is rare, but it is a form of collaboration which needs to happen more if our country is going to be competitive in the world's technology market.
This partnership is truly innovative, which is why the name ARD2-INNOV8 was chosen for the shield.
The ARD2-INNOV8 shield is a piece of technology designed for students and teachers by a student and teacher in collaboration with two great Australian technology companies.
This story should inspire you to realise that with effort, creativity, and thinking, you could become the creator of the next innovative piece of technology.

What is Arduino and why do I need it?

Your ARD2-INNOV8 shield can let you do many cool things, however, it cannot think on its own, because it does not have its own *micro-controller*. To run programs through your ARD2-INNOV8, you need to sit it on top of a micro-controller like the Arduino Uno.
Arduino is a hardware platform which is built around a micro-controller. Many different boards have been developed as part of the open source movement by the Arduino community. Arduino boards have been used in thousands of projects over the years by innovators both young and old, producing a wide range of designs from: robots to: musical instruments, from: children's toys to: scientific measuring devices.
Arduino boards are used in schools for learning how to program, and they are used by professional electrical engineers to make innovative products.

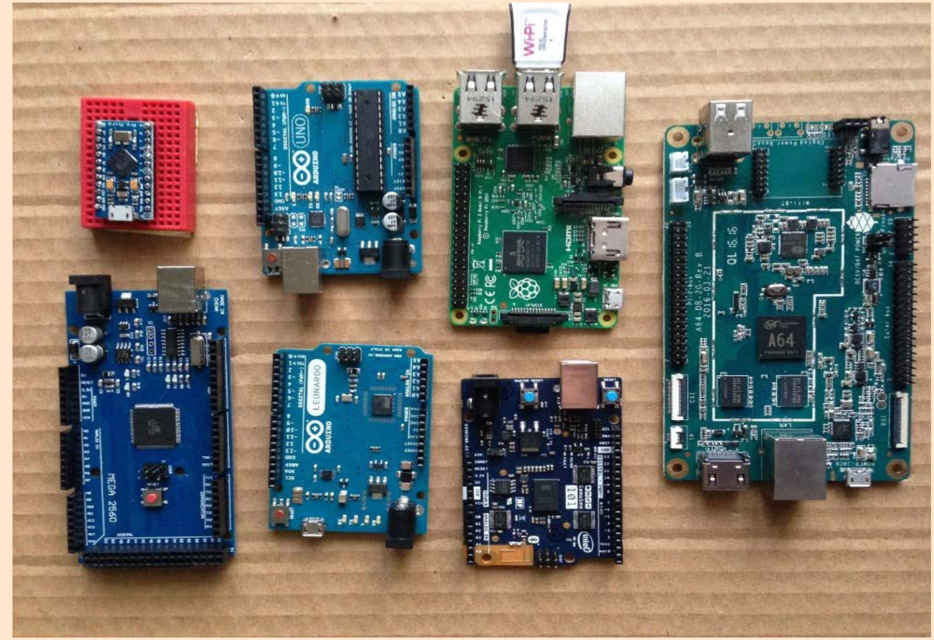


Figure 2. A selection of different micro-controller development boards

The most widely known and used board is the Arduino Uno. The Arduino Uno board is very versatile and reliable, and the design has been replicated by a number of technology companies and novices worldwide, ensuring that this technology is easily available across the world. The Arduino Uno will be the board used in all of the lessons and projects in this book.

The first Arduino board was created in 2005 at the Interaction Design Institute in Ivrea, Italy, as a low cost and easy way of helping students to *prototype* and program with micro-controllers. The ARD2-INNOV8 shield follows this same guiding principle – low cost and easy to use.
Connections to the micro-controller are called pins; the Arduino has 13+ *digital* pins available for use, and 5 *analog* pins. These pins are linked to your ARD2-INNOV8 shield by connecting it onto your Arduino board. These connections to the Arduino mean that you can control the LEDs, buzzer, and other components on the ARD2-INNOV8 by creating programs which tell the Arduino's micro-controller what to do.

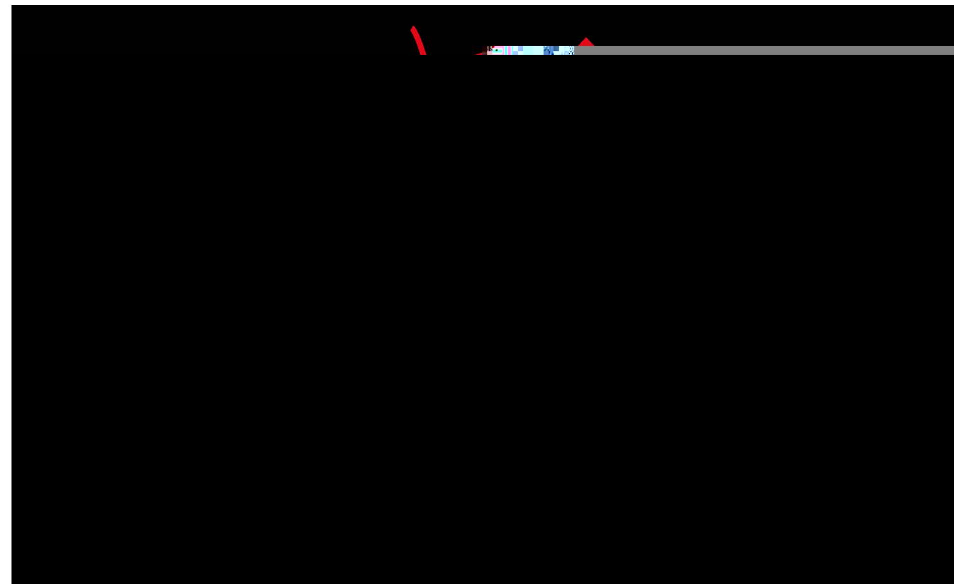
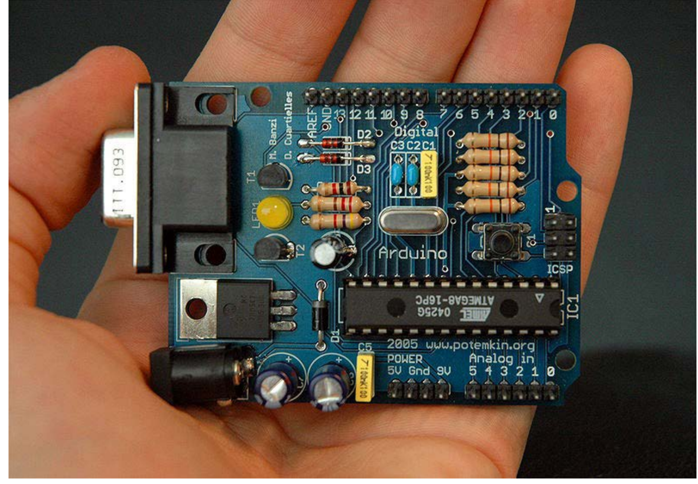


Figure 3. The Arduino Uno board, *analog* pins are numbered A1 to A5, and



Why use the ARD2-INNOV8?

Firstly, it is a pretty inspiring story to share with others, about how, with persistance and effort, any student can work collaboratively with experts to do amazing things.
Secondly, the ARD2-INNOV8 is like a set of training wheels that you use on a bike when you are first learning to ride. These training wheels make the job of balancing easier so that you can concentrate on other things, like steering and stopping.

Because there are no messy wires to deal with and no chance of placing components the wrong way round, users of the ARD2-INNOV8 can more thoroughly concentrate on the programming language and algorithm design. Once you have gained confidence with Arduino programming, you can remove the ARD2-INNOV8 shield and move on to more complex systems.

The ARD2-INNOV8 shield's interface gives learners an easy way to learn the fundamental lessons in micro-controller programming, therefore providing a good solid start to learning about embedded systems and programming.
Seven and Mark designed the ARD2-INNOV8 shield in collaboration with Richard Wilson (Wiltronics) and Ben Sieira (CognetronicS) with the aim of providing students, teachers, and enthusiasts an easier pathway to learning how to program an Arduino micro-controller board.

Getting started with your ARD2-INNOV8

A match made in heaven - Putting the ARD2-INNOV8 and Arduino together

There are a few things that you need to do before you can start programming and interfacing with your ARD2-INNOV8 Shield.
The most important of these things is that you need to have an Arduino compatible board such as the Arduino Uno that you can attach to the ARD2-INNOV8.
The reason that you need this Arduino compatible board is the ARD2-INNOV8 shield only provides an interface to components like LEDs and a button, but it does not have its own micro-controller, and without a micro-controller you cannot do any programming.
So the first step in programming with the ARD2-INNOV8 is to ensure that you have your ARD2-INNOV8 firmly attached to an Arduino Uno compatible board.
When attaching your ARD2-INNOV8 to an Arduino Uno compatible board, you should take extra care to ensure that all of the pins under the ARD2-INNOV8 are perfectly aligned with the socket headers on the Arduino (see **Figure 6**).

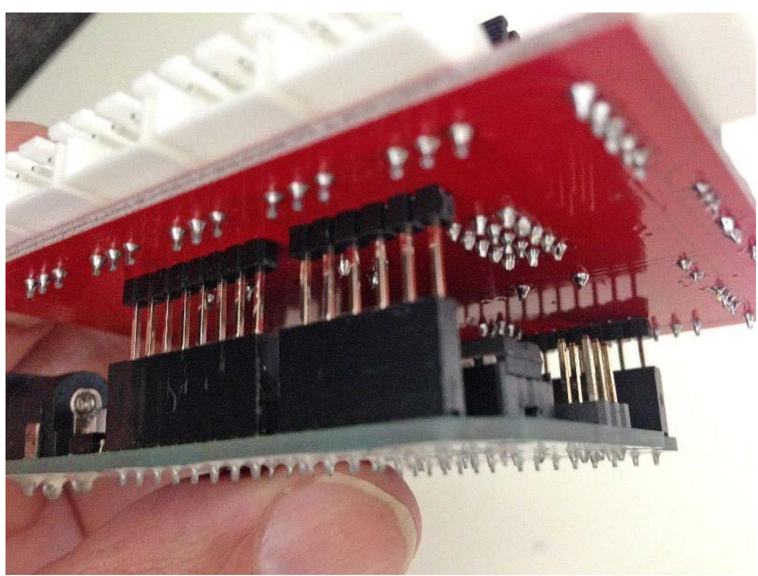


Figure 6. Ensure the pins under your ARD2-INNOV8 line up with the header sockets on your Arduino board

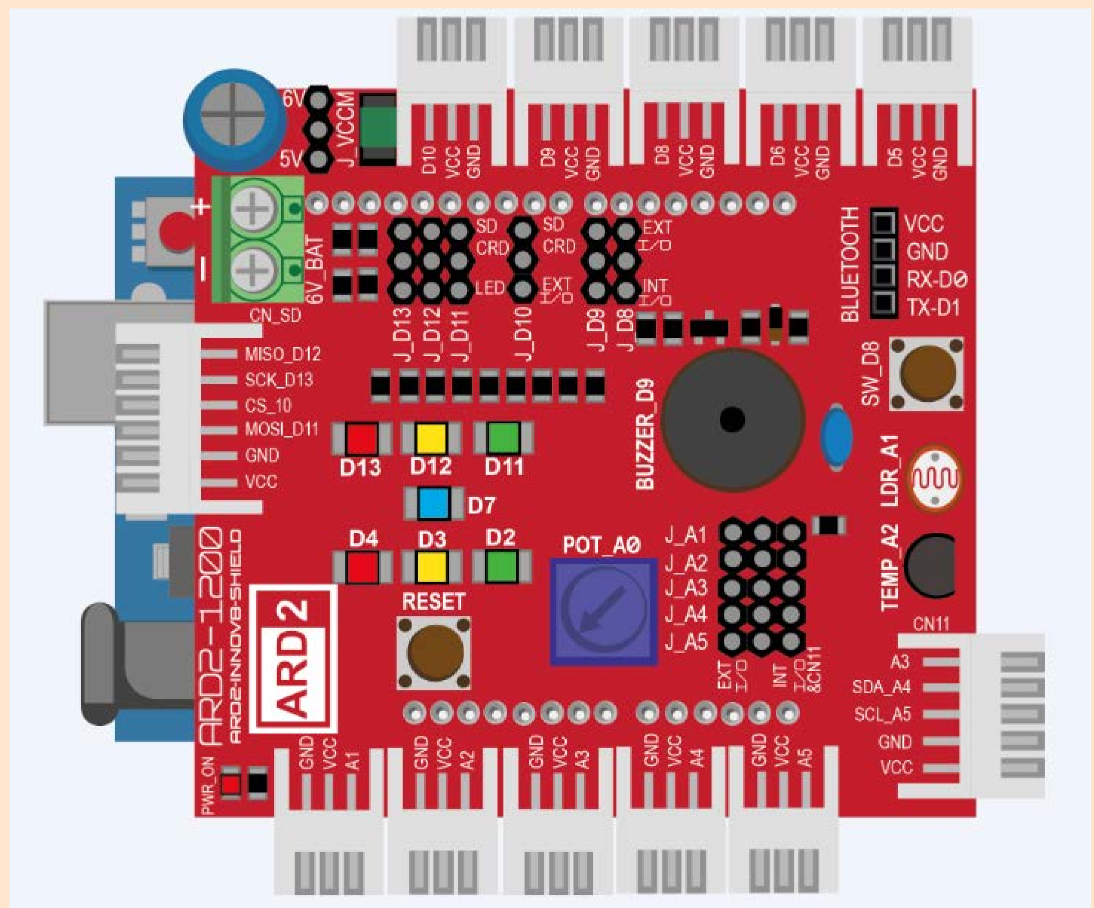


Figure 7. Your ARD2-INNOV8 and Arduino board should look like this when fitted together

Using Snap4Arduino

Programs run on Arduino micro-controller boards are referred to as *sketches*. There are many programming platforms available to use for Arduino programming, but most use *text-based code*.
Snap4Arduino is a modification of Snap!, a *visual block* programming environment which lets you drag-and-drop *code blocks* to create programs. Snap! (presented by University of California at Berkeley) is an alternative extension to Scratch (presented by the MIT Media Lab).
Snap4Arduino was created to allow drag-and-drop programming of Arduino boards and shields. It is a great place to start for younger learners and first-time Arduino programmers, but you may wish to use *text-based* code with the Arduino *IDE* (integrated development environment) when you advance to more complex programs.

Programs created in the Arduino IDE are uploaded (*compiled*) to the board, whereas programs created in Snap4Arduino are sent as instructions to the board when the program is running via a *firmware* program which has been loaded onto the Arduino board.

When you design code for your ARD2-INNOV8 shield and use its hardware components, you will be doing what is known as *prototyping*. Prototyping is when you make a limited working model of something to test a concept or idea. Prototyping with the ARD2-INNOV8 is faster and much easier to setup.

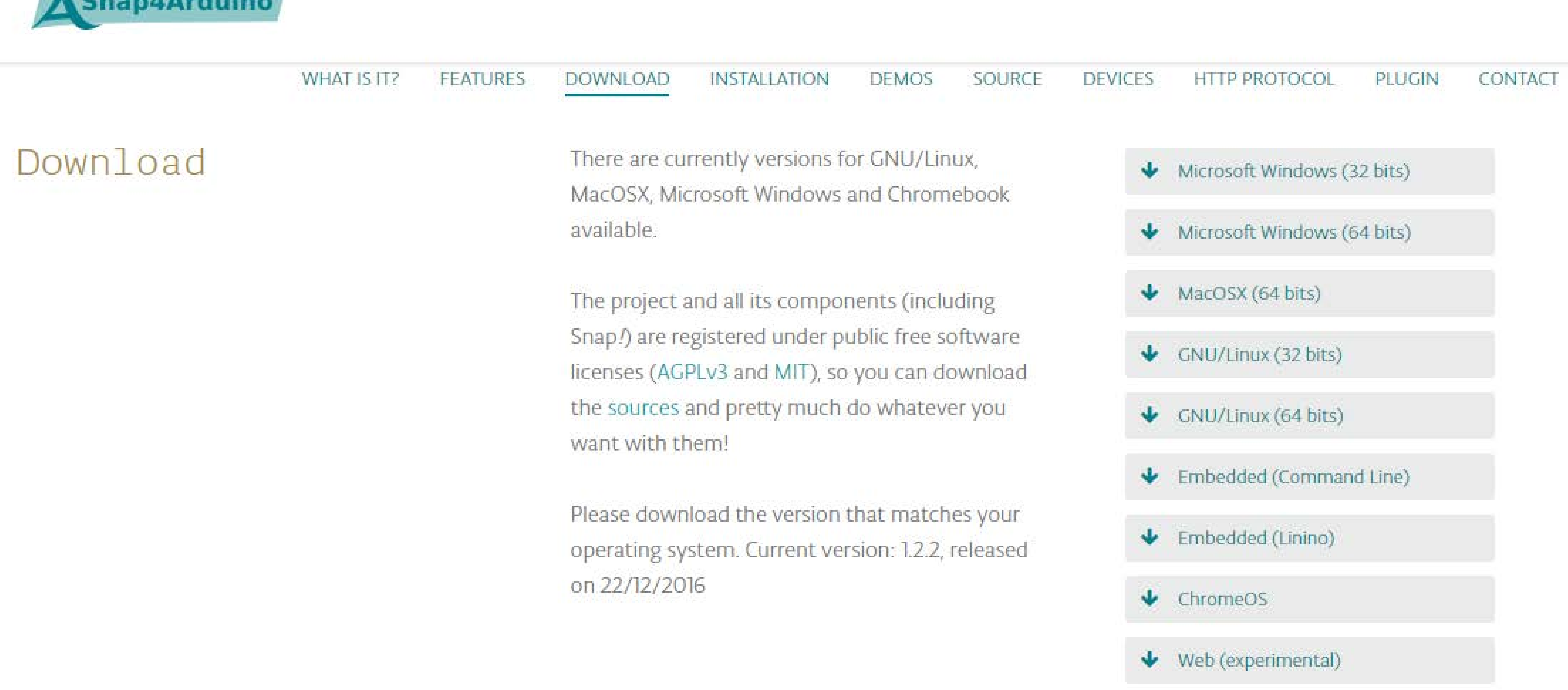


Figure 8. The Snap4Arduino website, where you can download the programming software

How to set up Snap4Arduino on your computer

The Snap4Arduino website will provide you with all of the information you will need to successfully install Snap4Arduino onto your computer. You will also need to download and install the Arduino IDE from the Arduino website. This is discussed in a later section of this book.
Video guides demonstrating how to install this software are available on the Firebugs Youtube channel.

The Snap4Arduino programming environment

The Snap4Arduino programming environment or IDE (integrated development environment) is very easy to use, and is very similar to Scratch and other drag-and-drop programming environments.
You can select code blocks from the programming block libraries: *Motion*, *Looks*, *Sound*, *Pen*, *Control*, *Sensing*, *Operators*, *Variables, Arduino*, and *Other*.

In this guidebook we will be using code blocks from most of these libraries, but not all of them.
The centre area of the IDE is where you build your program by dragging on blocks from the code block libraries (**Figure 9**).

On the left of the IDE is where you access the code libraries, and on the right of the IDE is the ***Stage*** where you see the action and see your *variables* and how they are changing. You will get to see how this works when you start programming.
The Snap4Arduino IDE has a special library which contains code blocks that are for programming the Arduino. This library also contains buttons to connect and disconnect your Arduino board (**Figure 9**).

The ***Other*** code block library does not contain any blocks, but instead contains a single button which allows you to create your own code block *functions*. You will get to see how this works in **Lesson 9**.

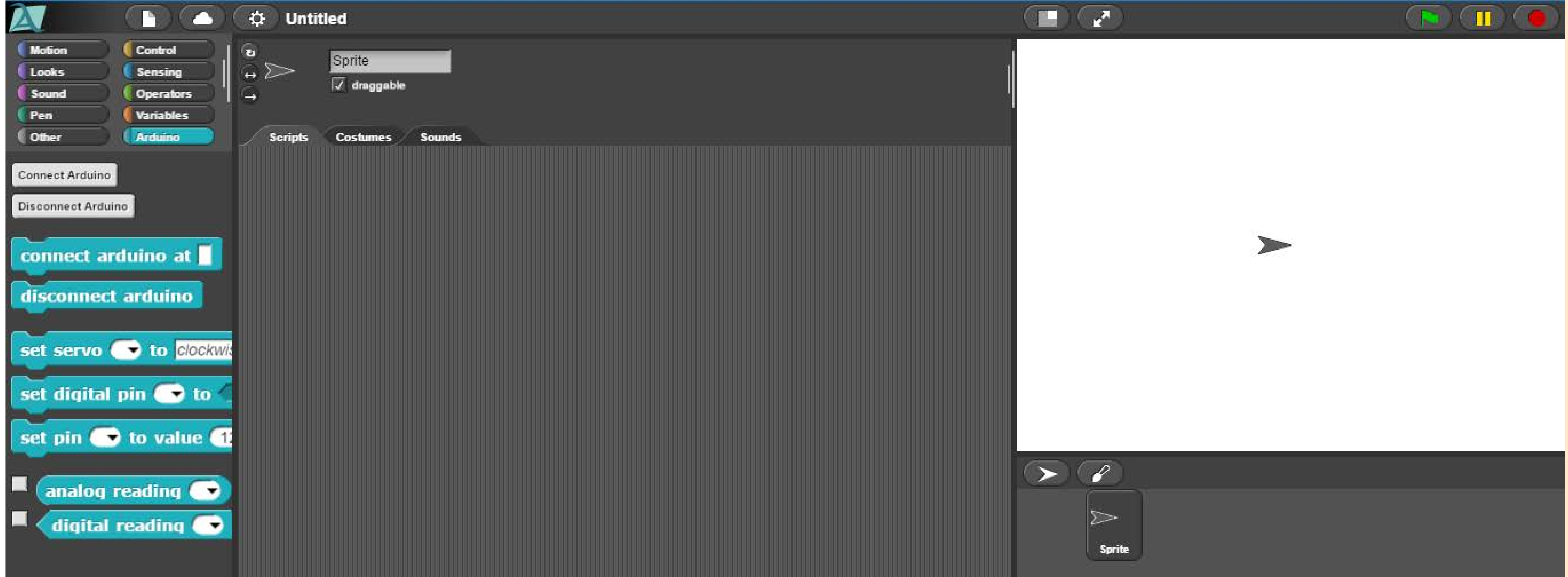


Figure 9. The Snap4Arduino programming environment

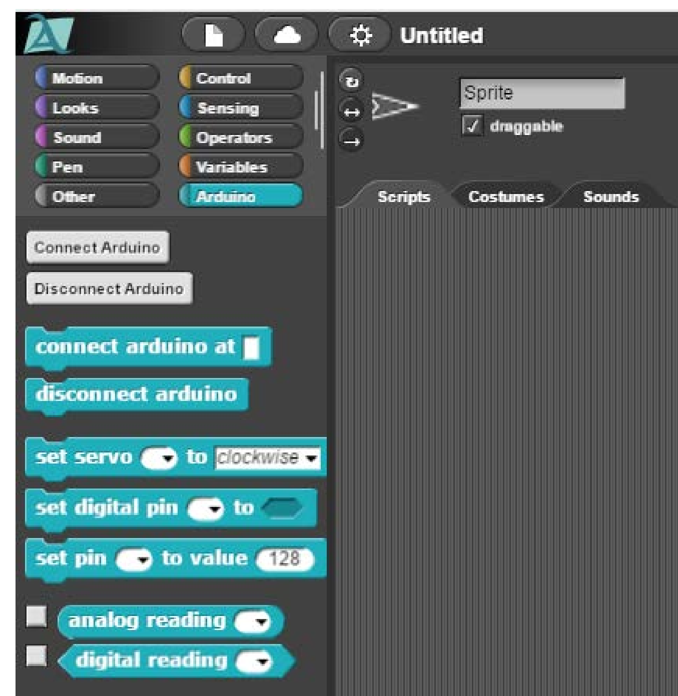


Figure 10. Programming blocks within the Arduino library

Creating and duplicating blocks

Creating new code blocks is easy; simply drag the block you want from its library into the programming window (**Figure 11**).

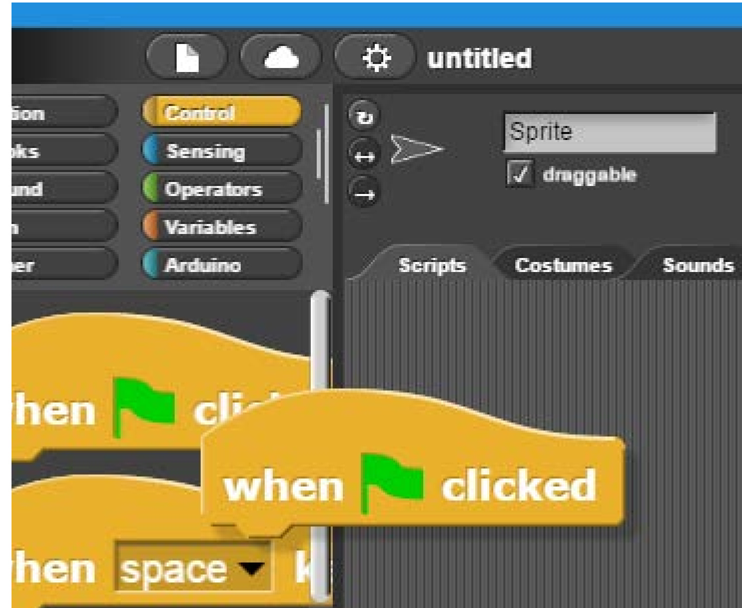


Figure 11. New code blocks can be created by dragging blocks into the programming window

You can easily duplicate blocks by right-clicking on the block you want to copy, and then selecting ***duplicate.*** You can also delete blocks this way, or simply drag the block back to its library and it will disappear.

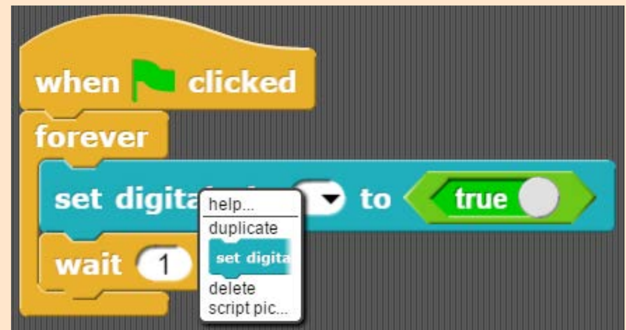


Figure 12. Options available when right-clicking on a block

Zooming in and out

You can increase or decrease the size of your blocks so they are easier to see by selecting ***Zoom blocks*** from the *Settings* menu (cog icon – **Figure 13**).
In the *Zoom blocks* window you can choose what size you want your blocks to be.

Figure 13. The File, Cloud, and Settings menus

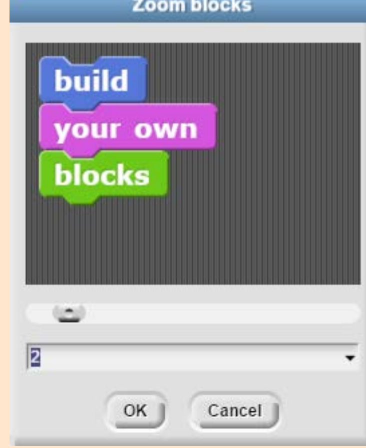


Figure 14. The Zoom blocks menu, where you can choose the size of your blocks

Saving, importing, and exporting

Saving, importing, and exporting is easy, and is done via the *File* menu.
Common key shortcuts like *Ctrl-S* (Command +s) for save can also be used, and you can choose to save your projects into the projects folder or to the Snap4Arduino cloud if you have an account (account activation is free).

About the lessons in this book

Note to teacher

Each lesson in this book is designed to build capacity with a particular set of programming knowledge, skills, and problem solving. There is a *key focus*, and *key words* in each lesson, and most lessons build onto each other. Learning scaffolding is reduced as the lessons proceed, but the teacher may need to introduce further scaffolding when teaching junior levels.
There is a difficulty or challenge rating for each lesson represented by a number of heads:

This head of course belongs to Nicola Tesla, the famous inventor from the early 1900s whose creativity and ingenuity knew no bounds.
Learners who are having difficulty grasping the concepts in the more challenging lessons should skip them and come back to them later. Lessons can be approached by first copying and implementing the code, making observations, and then returning to the text once an understanding of the working system has been gained.
There is also a ***Quick Blocks*** section just after Lesson 1, which provides all of the basic sketches needed to run components on the ARD2-INNOV8 shield.

Loading the lesson code and resources onto your computer

Download the ***ARD2-INNOV8-SNAP*** resources folder from the Wiltronics or Firebugs websites.
Unzip this folder to an easy to find location somewhere on your computer.
This folder contains all of the lesson code which you can import into Snap4Arduino if you get stuck, and it also contains other learning resources.

Loading the firmware

To program your board with Snap4Arduino you will first need to install *firmware* using the Arduino IDE.Download and install the Arduino IDE onto your computer from the Arduino website.
Once you have this software installed onto your computer:

- Run the Arduino program on your computer.
- Connect your Arduino board with ARD2-INNOV8 shield attached to your computer using the USB cable.
- Check that your board is connect by looking under: ***Tools > Port:***
- You should see a ***tick*** next to one of the *serial ports* listed (this looks slightly different on a MAC and Linux computer, but the process is much the same) [see **Figure 15**].

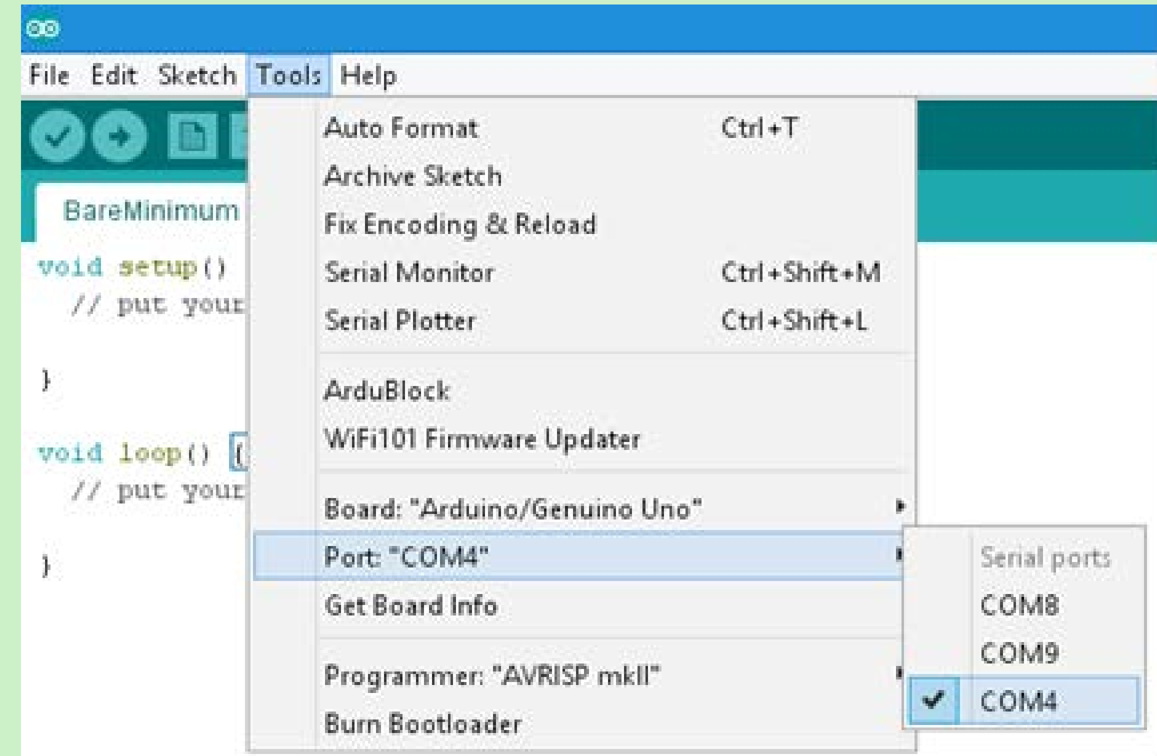


Figure 15. How to check for port connection under the Tools menu (PC screenshot)

Open the ***StandardFirmata*** sketch from: File > Examples > **Firmata**

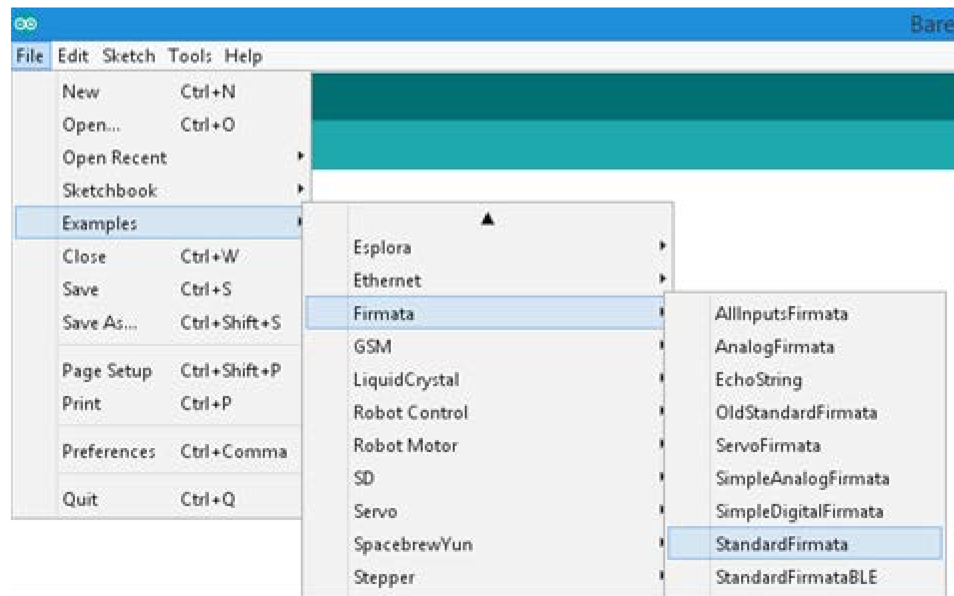


Figure 16. Where to find the *StandardFirmata* sketch (firmware)

Load this *StandardFirmata* sketch onto your board by clicking on the ***Upload*** button

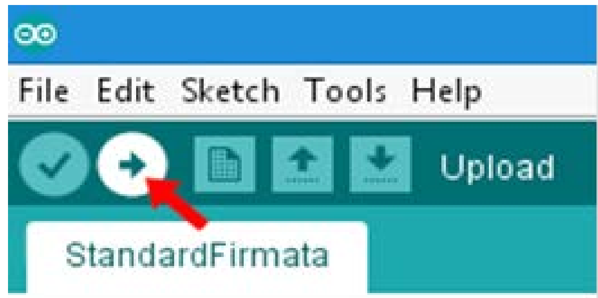


Figure 17. Clicking the Upload button loads the sketch to your board

If you see the message ***Done uploading*** at the bottom of the programming window, then you are all set to program with Snap4Arduino.

Connecting your board to Snap4Arduino

Next you will need to connect your board to the Snap4Arduino program. Within the ***Arduino*** blocks library you will see two buttons at the top. These are ***Connect Arduino*** and ***Disconnect Arduino***.

Figure 18. The connect and disconnect buttons for your board

When you click on the *Connect Arduino* button you should see the port to which your board is connected (the port number should be the same as when you connected to the Arduino IDE) [this looks slightly different on MAC and Linux]. Select the correct port number and you should then be greeted with the message in **Figure 20**.

Figure 19. COM ports are visible when the Connect Arduino button is pressed

If you have trouble connecting your board, watch the video called ***Blink – First Sketch*** under the Snap4Arduino section on the Firebugs Youtube channel.
You will need to do this every time you open or create a new sketch.

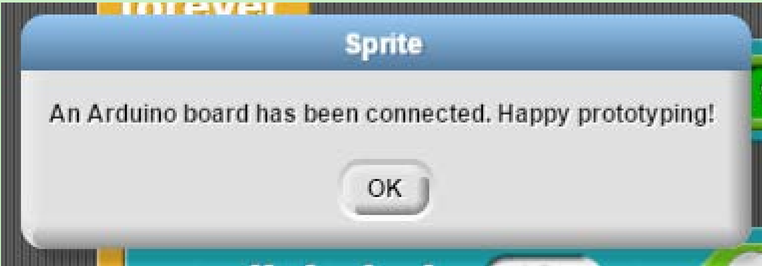


Figure 20. The message seen when a board has been successfully connected

Alternatively, you can include this block at the top of your sketch which will then automatically connect the Arduino board at the chosen port once the sketch is run.

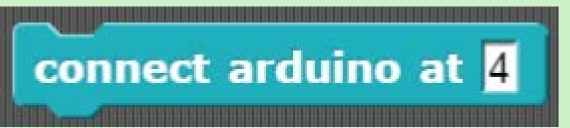


Figure 21. When this block is initiated it will connect the Arduino at COM port 4

Some basic electronics

You will need to know a little bit of electronics to fully understand the lessons in this book. Here are explanations of some of the electronic components found on the ARD2-INNOV8 shield.

Image	Diagram	Name	What it does
	Anode Cathode	LED (light emitting diode)	Gives off light when an electric current is applied. Electricity can only flow one way through a LED. There are many colours available. This image shows a standard LED; the LEDs on the ARD2-INNOV8 are small '*surface mount'* LEDs.
		Resistor	Slows the electric current down. It acts like a bumpy road slowing cars down. The coloured bands on a resistor show its value – measured in Ohms. You use a resistor in series with a LED to protect the LED. This image shows a standard resistor; the resistors on the ARD2-INNOV8 are small *'surface mount'* resistors.
		Push Button	Connects a circuit when the button is pressed. There are many different types of push buttons; this one is known as a momentary SPST switch. There are two of these on the ARD2-INNOV8: one can be used in program sketches, and the other is used to reset the code.
		Potentiometer	Much like a resistor, but its resistance value can be changed by turning its knob. It has three legs. Used as a control knob to vary a value. Can be used to turn volume up and down, or to dim a LED from dull to bright.
		LDR (light dependent resistor)	Changes its resistance value when light falls onto it. Can be used as part of a circuit to switch lights on when it gets dark.
		Temperature Sensor	Used to sense temperature. Can read temperature from -40C to +125C. Reads as a voltage which then needs to be converted in the code to temperature.

Lesson 1 - Blink – Your first sketch

Key words: firmware, input, output, process, system, LED, sketch, control structure, loop, operator, Boolean, digital, pseudo-code, debugging
Key focus: Snap4Arduino IDE, basic coding, basic electronics, systems knowledge, using loops

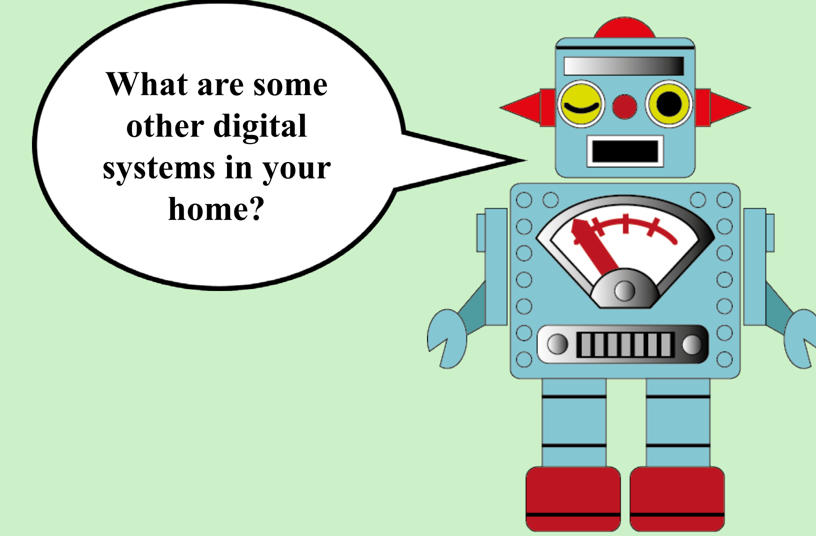
Difficulty Level

Inputs, Processes, and Outputs

In all of the lessons in this book we are going to be working with digital devices (the ARD2-INNOV8 and Arduino Uno) which have *inputs*, *processes*, and *outputs*.
All systems have at least one input, one process, and one output, however, most systems have more than one of each of these.
In the following lessons you will be working on the process side of *digital systems*, and programming them to use a number of inputs (such as a button press) and outputs (such as the lighting of a LED).
To get us started, think about some of the digital systems in your home and try to fill in the missing information in the table opposite.

Fill in the blanks in the table below:

System	Inputs	Process	Desired Outputs
Washing Machine	Electricity, button presses, sensor data	Fill (water), wash (agitate), spin, various options	Motion (rotary), sound (beeps), light(LEDs)
Smart TV	Electricity, data(signal), button presses, IR remote signal, Internet data, BlueTooth data		
Game System		User interface, provide games, connect to internet, connect to hand-controller, play disks	
Dish Washer			Motion (rotary), water pressure, heat, sound (beeps), light (LEDs)
Air Conditioner	Electricity, button presses, sensor data		



Setting up the code blocks

The very first sketch that Arduino programmers run on their boards is the *Blink* sketch.
This sketch blinks an *LED* (light emitting diode) on and off at specific timings.
The Blink sketch allows you to test your board, and gives you an opportunity to learn the code basics.
The blocks that we will use in this sketch are:

Figure 22. The When clicked block

The ***when clicked block***, found within the *Control* blocks library, runs/starts the program when you click on it.
This block is placed at the top of your code blocks, with all other blocks connected underneath.
There are a few different blocks which you can use instead of this block such as the ***when _ key pressed*** block.

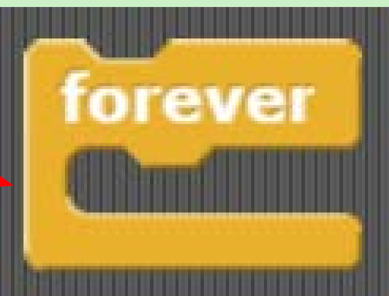


Figure 23. The forever block

The ***forever*** block is a *control structure*, found within the *Control* blocks library.
This block runs the code within it forever (in a *loop*), until the program is stopped by clicking back onto the ***when clicked block*** or if the board is disconnected.

Figure 24. The Set digital pin block

The ***set digital pin_ to_*** block, found within the *Arduino* blocks library.
This block does two things: it *defines* the pin number which will be used, and it sets this pin to either *TRUE* or *FALSE* (HIGH or LOW – ON or OFF) with the use of the ***true/false*** *operators* (**Figure 25**). Pin numbers can be typed into the space provided.

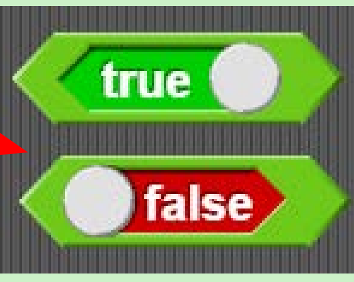


Figure 25. The true/false operators

The ***true/false*** operators, found within the *Operators* blocks library, are used to set the state of something. Their value is *Boolean* - either TRUE or FALSE. In the case of the *Blink* sketch, we will use these operators to set the LED attached to pin 13 to ON or OFF. These operator blocks can be dropped inside other blocks by dragging them over the appropriate part of the block.

Figure 26. Blocks can be dropped inside of other blocks

The ***wait*** block, found within the *Control* blocks library, causes the program to pause for the time specified. The block pictured in **Figure 27** shows a block which will cause the code to pause for **1 second**.

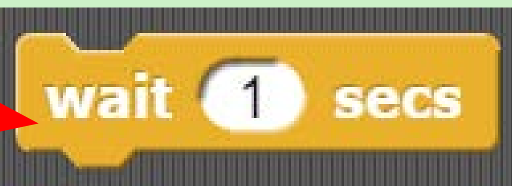


Figure 27. The wait block

Content Index

Jumper position

The ARD2-INNOV8 board has ***pin jumpers*** which need to be placed in certain positions to use components on the shield. For lessons 1 to 9, make sure that your ARD2-INNOV8 shield has its pin jumpers placed in the position shown in **Figure 28**.

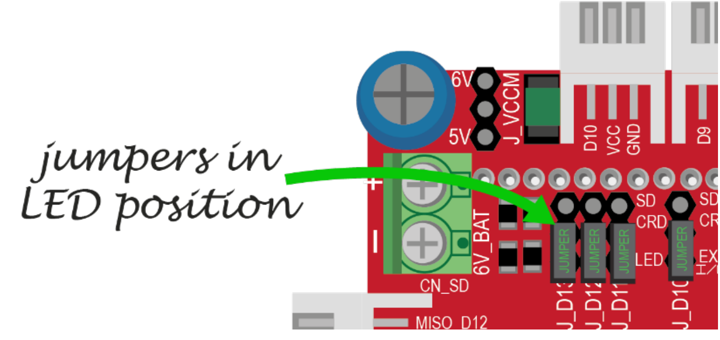


Figure 28. Ensure the jumpers are in this position for the Blink sketch

The Blink Sketch

The sketch shown in **Figure 29** would look like this if written in *structured English* (*pseudo-code*).

```
Blink Sketch
WHILE
     SET digital pin 13 to TRUE
     WAIT 1 second
     SET digital pin 13 to FALSE
     WAIT 1 second
ENDWHILE
EXIT
```

Study this code and the code in **Figure 29** and try to work out what will happen, then create the blocks as shown.
After you have constructed your sketch using the right blocks, make sure that your board is connected and then click on the ***when clicked block***.

You should see a halo form around the blocks (**Figure 30**), and your ARD2-INNOV8 shield should be doing something.

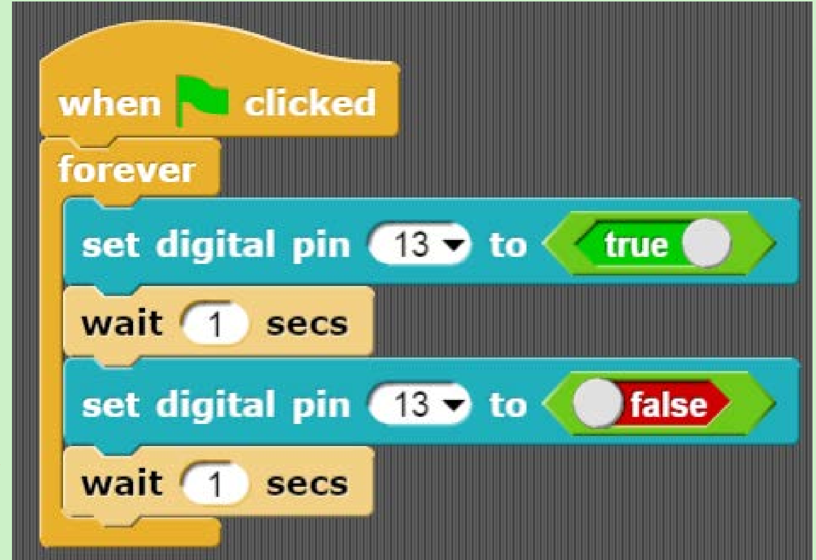


Figure 29. The Blink sketch

What is happening on your ARD2-INNOV8, did you see any LEDs blinking on and off?
If you did not see one of the LEDs on your ARD2-INNOV8 blinking on and off, then something has gone wrong.
When something goes wrong with digital systems we say that there is a *bug* in the system. The process of fixing the problem is known as *debugging*.
Fix your system and sketch by *debugging* the code. Check that your board is connected and that the pin jumpers are in the correct position (**Figure 28**).

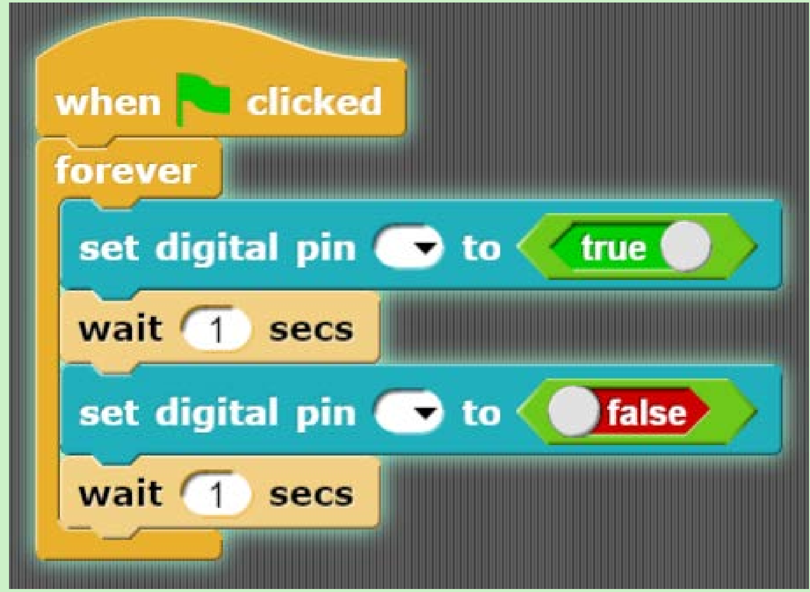


Figure 30. Halo around the blocks when active (clicked)

Content Index

Now try this

Add in more digital pin blocks and select different pin numbers to see what happens.
The LED pins on your ARD2-INNOV8 are **2, 3, 4, 7, 11, 12, and 13**.

Try playing with the code blocks to see if you can:

- Make all 7 LEDs blink on and off
- Make the 7 LEDs blink on and off one after the other
- Changing the blinking speed (rate) to create a light pattern

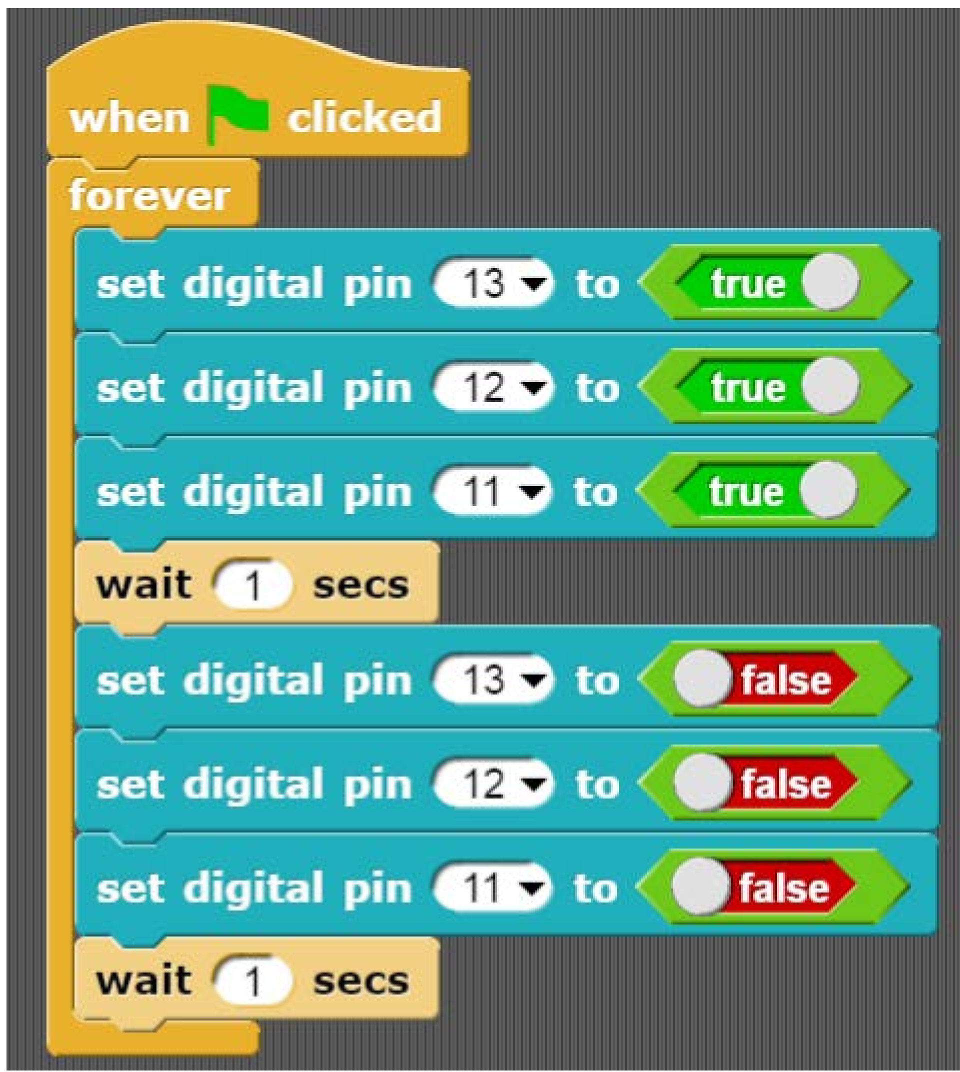
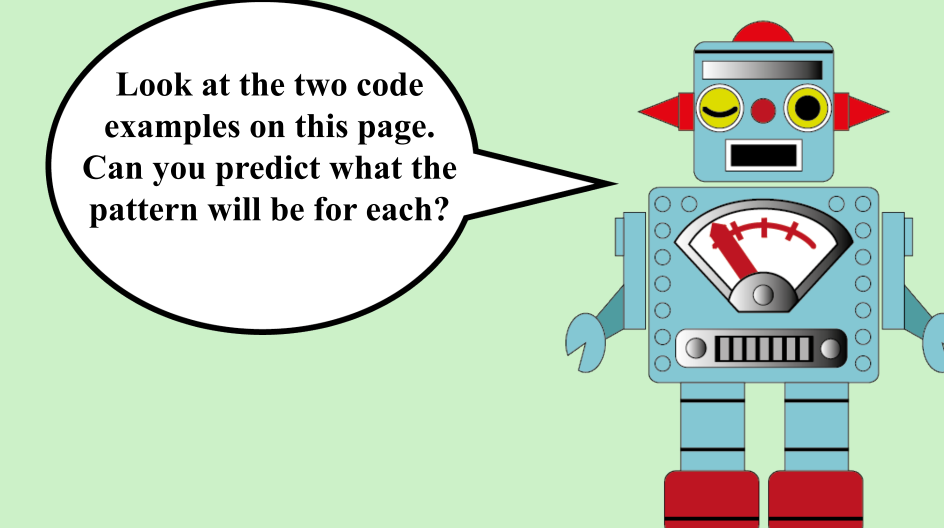


Figure 31b. The Blink sketch with added LED pins

```
when clicked
forever
  set digital pin 13 to true
  wait 1 secs
  set digital pin 12 to true
  wait 1 secs
  set digital pin 11 to true
  wait 1 secs
  set digital pin 13 to false
  wait 1 secs
  set digital pin 12 to false
  wait 1 secs
  set digital pin 11 to false
  wait 1 secs
```

Figure 31b. A different version of the sketch in Figure 31a



Content Index

QuickBlocks

Difficulty Level

The following seven images show how you can build quick program sketches to use the inputs and outputs on the ARD2-INNOV8 shield. These examples provide only the very basics, however, the following lessons will explain how to build proper programs.
If you found **Lesson 1** difficult, then you may wish to play with the 7 examples in this Quickblocks section, but if you are keen to continue to learn real programming then you can skip ahead to **Lesson 2.**

Blink a light

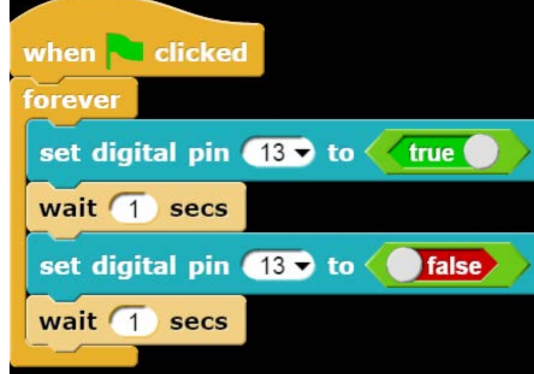


Figure 32. Blink an LED (this code will blink the LED on pin 13)

Blink a LED with a button

Press button D8 and observe what happens.

Figure 33. Turn an LED on with a button press

Fade a LED by turning a knob

Twist the knob of the potentiometer and observe what happens.

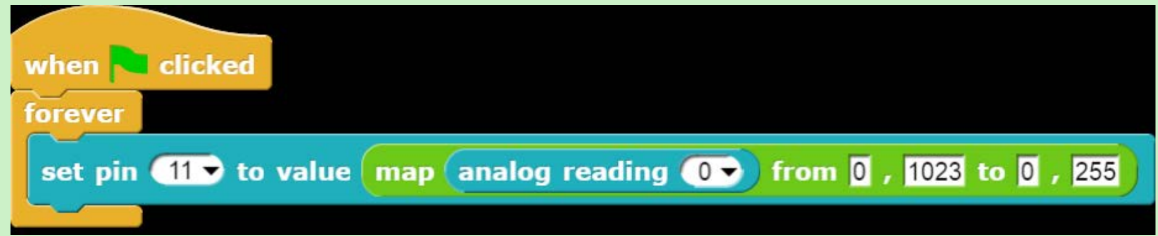


Figure 34. Fade an LED with the potentiometer (you need to import the map block for this program – refer to Lesson 6)

Change sounds by turning a knob

Twist the knob of the potentiometer and observe what happens.

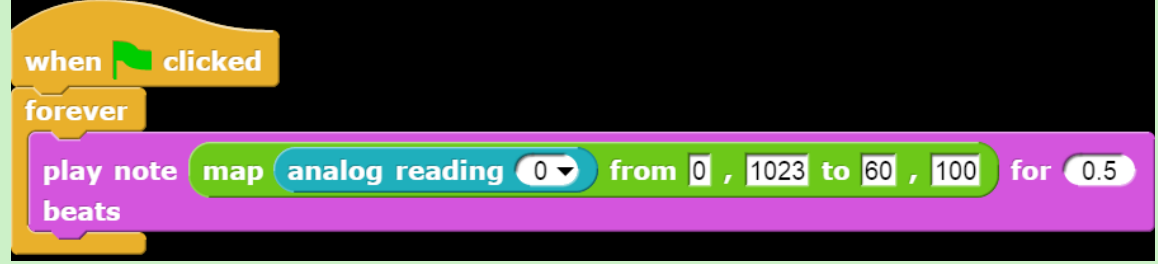


Figure 35. Change the sound pitch with the potentiometer (you need to import the map block for this program– refer to Lesson 6)

Move a motor with a button

Press button D8 and observe what happens.

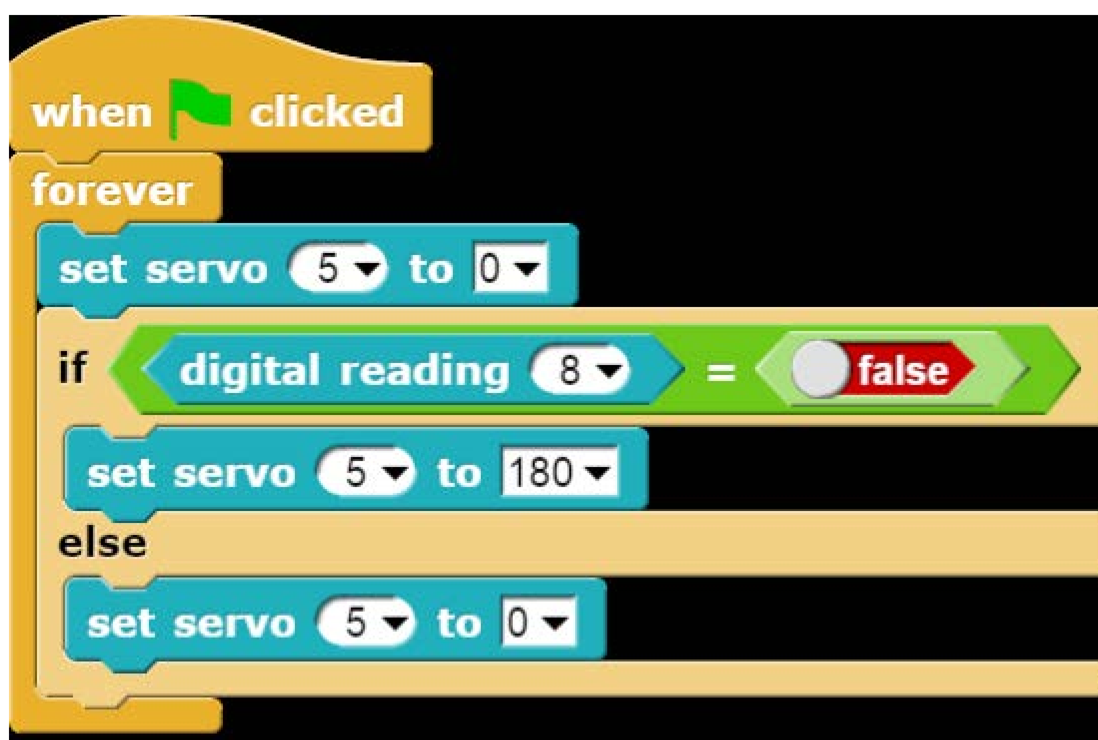
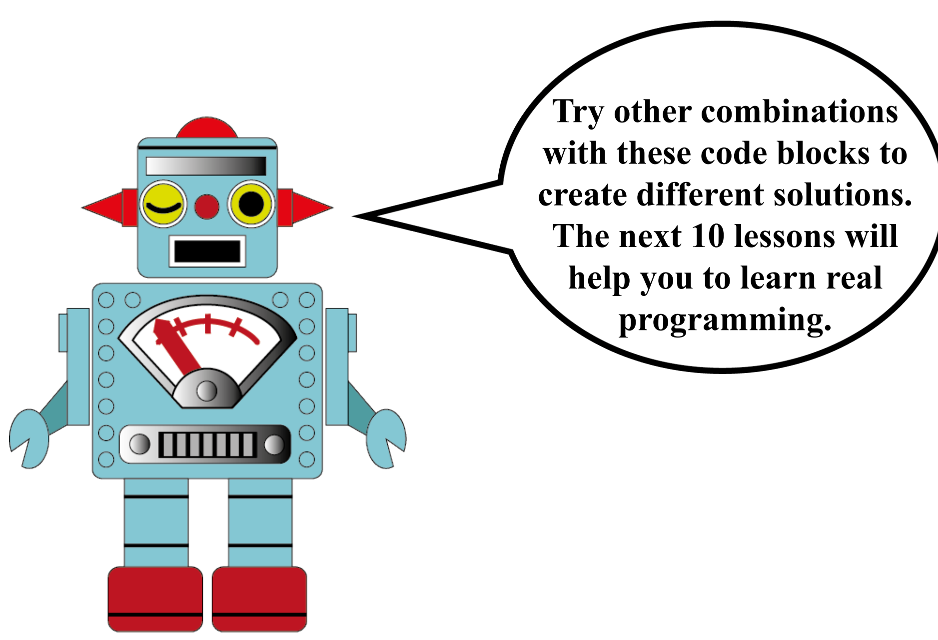


Figure 36. Move a servo motor with a button press (Lesson 10 shows how to attach the servo motor)



Turn a LED on when it gets dark

Place your finger over the LDR and observe what happens.

Figure 37. Turn an LED on when the light level falls on the LDR sensor (you may need to adjust the number ‘50’ up or down)

Turn a LED on when it gets hot

Press your finger against the temperature sensor and observe what happens.

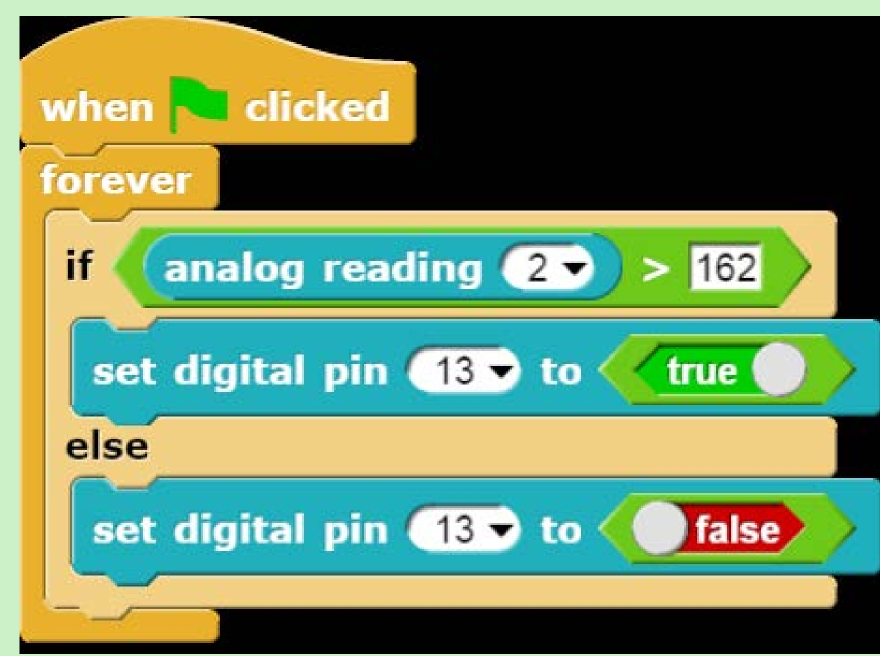


Figure 38. Turn an LED on when the temperature rises above a certain point (you may need to adjust the number ‘162’ up or down a little)

Content Index

Lesson 2 – Using variables

Key words: Visual-block, text-based, loop, variable, declare, define, data structure, control structure
Key focus: How to create (declare and define) and use variables

Difficulty Level

In **Lesson 1** you were able to blink the LED attached to pin 13 ON and OFF by using the correct code blocks.
You used the s***et digital pin*** code block to set the LED pin true (HIGH) or false (LOW), and you used the ***wait*** block to delay the time between the ON and OFF state of the LED.
In actual Arduino code the *Blink* sketch looks like this:

```
int led1 = 13; // a variable for the LED pin
void setup(){
// initialize led1 pin as an output
     pinMode(led1, OUTPUT);}
void loop(){
     digitalWrite(led1, HIGH); // turn LED1 on
     delay(1000); // wait for a 1000 milliseconds
     digitalWrite(led1, LOW); // turn LED1 off
     delay(1000); // wait for a 1000 milliseconds}
```

In this code you can see that we use words in place of blocks. Text-based code is the preferred option for most professional programmers, because once you know how to use the code it is much easier to write programs with text rather than using blocks. When you are first learning how to program however, it is much easier to understand if you can see the program structures. This is why many students learn programming by starting with visual-block programming, like the code used by Snap4Arduino.

Learning programming by first using blocks is helpful, because mistakes are easier to spot, and many of the programming structures are the same as they are for text-based code.

Programmers also place comments within their code to help other people understand how parts of the code work. Within this text-based code you can see text which looks like this: `// a variable for the LED pin`
In Arduino programs text which has two forward slashes before it like this **//** is ignored by the micro-controller. This text is only for the human reader.

In the text-based code above you can see that instead of the ***set digital pin*** code block we have used the `digitalWrite()` function, and instead of the ***wait*** code block we have the `delay()` function.

You can also see that there is a *setup* section (`void setup`), and a *loop* section (`void loop`). The loop section works exactly the same as the ***forever*** block in our Snap4Arduino block code.

One of the main differences between this text-based code and our visual-block code is that in this text-based code we have used a *variable* to hold the pin number instead of just using the number 13.

```
digitalWrite(led1, HIGH);
```

A *variable* is simple a placeholder (container) which can hold a value that can be changed. Variables are very handy things within programs because they allow values to be changed as the program runs, for example: Imagine if you were playing a computer game where the score did not change when you scored a new point. This would make the game very boring. Variables in this situation allow the score to change in value every time you make a new score in the game.

In order to use a variable, we first need to create it at the top of our code.
In text based Arduino programming we do it like this:

`int led1 = 13;` (here we have *declared* our variable [named it], and *defined* its ititial value).

This statement tells the micro-controller (compiler) that we have a variable named led1 that holds the value of 13.

Creating variables

Within our Snap4Arduino IDE we create variables like this:
Click on the *Variables* library and then click on the ***Make a variable*** button at the top of this library.
Name this variable *led1*. This is called *declaring* a variable.

Figure 39. Creating a variable (declaring) called led1

Once you have done this and clicked OK you will see this variable appear at the top of the library.

Figure 40. Newly created variables appear at the top of the Variables library

In text-based programming variables need to be named in a specific way. Variable names usually start with a lowercase letter and must not begin with a number.

You can use this variable and set its initial value by using the ***set_to*** block, and you will find the name of your new variable within this block's drop down menu. This is called *defining* a variable.

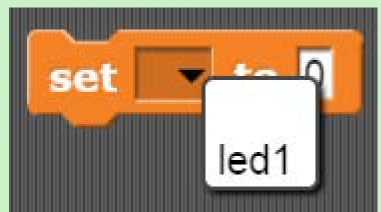


Figure 41. Variable names can accessed from a dropdown menu on some blocks

Now we have a variable to hold the pin number of our LED, but we should also create a variable to hold the value for delay amount between each blink of the LED.
We will name our second variable *blinkMore*, because we will use it to blink our LED more or less.

You will notice that upon creating these two variables, they appear in the stage window on the right side of the programming window (**Figure 43**).

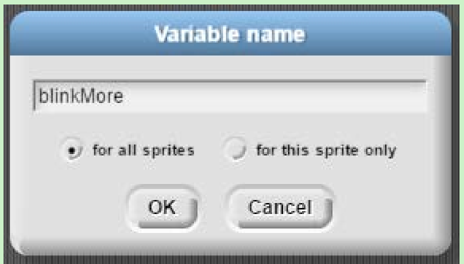


Figure 42. Creating a variable called blinkMore

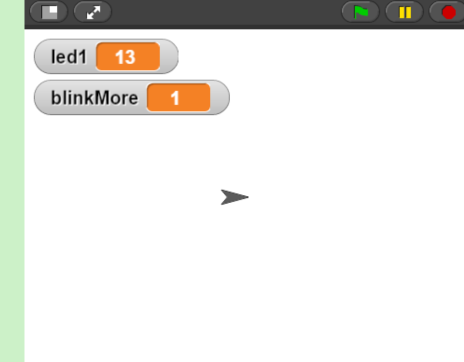


Figure 43. The Stage area on the right side of the SanapArduino IDE

Create the sketch

Set up a new sketch using your new variables as has been done in **Figure 45**. In this sketch we will make *led1* equal to 2, so that it will blink the LED connected to pin 2 on the ARD2-INNOV8.
To use your variables within blocks, simply drag the variable name from the library and drop it onto the appropriate part of the block (see **Figure 44**).

Figure 44. Variables can be dragged into other blocks

Figure 45. The blinkMore sketch which uses a variable called blinkMore

We can now blink all 7 of our LEDs by creating variables for each of them: *led2, led3, led4, etc.*

Figure 46. Creating a variable for each of the 7 LEDs

Apply your knowledge

Using the knowledge gained in this lesson and the previous lesson, create a new variable for each of your 7 LEDs, then create a sketch which turns the LEDs ON one by one, and then turns them OFF one by one. Use the wait block (with your *blinkMore* variable) to include a 1 second delay between each LED switching ON, and then each LED switching OFF.
You will need to define the value of each of your LED variables using the correct pin numbers.
The top part of your code blocks should look like this:

Figure 47. Defining each of the variables

When using the ***wait*** block you should use it with the *blinkMore* variable like this:

Figure 48. Using the *blinkMore* variable to set the delay for the *wait* block

Test your new sketch and observe what happens.
If you have constructed your code correctly then you should observed all 7 of the LEDs switching on one by one with a second delay between each of them, and then all 7 LEDs switching off one by one with a 1 second delay between each.

Now that you have successfully created this sketch, change the delay rate within the wait block to 0.5 seconds by changing the value of the blinkMore variable.
Note that because you have created the *blinkMore* variable, you do not need to change each ***wait*** block separately. To change the values in all of the ***wait*** blocks, you simply need to change the value of the *blinkMore* variable from 1 to 0.5.
See how easy it is to change values in your code with variables?
In the following lessons you will see how we can use other data structures and control structures to make our programs more elegant; which means that we will make them using less code blocks and make them flow more smoothly.

Content Index

Now try this

Create another variable so that you have two variables which you can use within the wait block. You could call this new variable *blinkMore2*.
Use this variable within your code to produce a sketch which blinks the 7 LEDs in a set sequence or pattern, such as one which would be used for party or Christmas lights.
Can you think of a way that you could adjust this code to reduce the number of blocks in your sketch?
In the next lesson we will learn how to achieve this using a specific *control structure* and a new variable.

In 1867 Captain Philip Colomb of the Royal British Navy used the flashing light from a ship lantern to communicate with other ships. This method of communication eventually adopted the Morse code language which is still used for communication today.

Can you think of a way to communicate a message to someone using the knowledge that you have gained in this lesson and the LEDs on the ARD2-INNOV8?

```
when [flag] clicked
set led1 to 2
set led2 to 3
set led3 to 4
set led4 to 7
set led5 to 11
set led6 to 12
set led7 to 13
set blinkMore to 1
forever
  set digital pin led1 to true
  wait blinkMore secs
  set digital pin led2 to true
  wait blinkMore secs
  set digital pin led3 to true
  wait blinkMore secs
  set digital pin led4 to true
  wait blinkMore secs
  set digital pin led5 to true
  wait blinkMore secs
  set digital pin led6 to true
  wait blinkMore secs
  set digital pin led7 to true
  wait blinkMore secs
  set digital pin led1 to false
  wait blinkMore secs
  set digital pin led2 to false
  wait blinkMore secs
  set digital pin led3 to false
  wait blinkMore secs
  set digital pin led4 to false
  wait blinkMore secs
  set digital pin led5 to false
  wait blinkMore secs
  set digital pin led6 to false
  wait blinkMore secs
  set digital pin led7 to false
  wait blinkMore secs
```

Figure 49. Solution for switching all 7 LEDs on, then off, one after the other

Content Index

Lesson 3 - Using control structures to simplify code

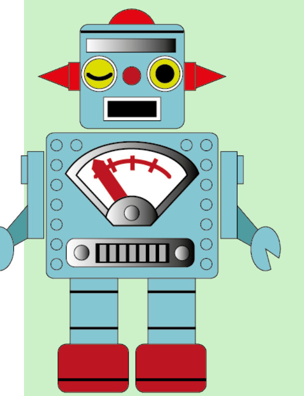
Key words: iteration, list, array, element
Key focus: using control structures, using tracking variables, pattern recognition

Difficulty Level

In the previous lesson you learned how to use variables to make changes easier to manage, however, when we needed to blink all 7 LEDs our code blocks became very large and difficult to read.
Thankfully there are structures that exist within programming languages like Snap4Arduino and Arduino text-based code which allow us to simplify any repetitive task.
If you look at the code block structure of the sketch in **Figure 50**, you can see that this code simply turns all 7 LEDs ON, then waits for 1 second, and then turns all LEDs OFF. To do this simple action the sketch uses more than 24 blocks. Surely there must be a better way?

Thankfully we can use data structures and control structures to make these types of repetitive jobs simpler.
A *data structure* is simply a thing that can hold data. There are several types of data structures which can be used to hold different types of data. A variable is a simple data structure.
A *control structure* is a block of code which controls the flow of a program. The forever loop is a control structure, but there are several more.
In this lesson we will learn how to use a data structure called a *list* (*array*) to contain all of our LED pins, and we will learn how to use the ***repeat until*** control block to set all of our LED pins to ON and OFF.

A process which is repeated over and over is called *iteration*. Control structures can simplify these tasks.



```
when [flag] clicked
set led1 to 2
set led2 to 3
set led3 to 4
set led4 to 7
set led5 to 11
set led6 to 12
set led7 to 13
set blinkMore to 1
forever
  set digital pin led1 to true
  set digital pin led2 to true
  set digital pin led3 to true
  set digital pin led4 to true
  set digital pin led5 to true
  set digital pin led6 to true
  set digital pin led7 to true
  wait blinkMore secs
  set digital pin led1 to false
  set digital pin led2 to false
  set digital pin led3 to false
  set digital pin led4 to false
  set digital pin led5 to false
  set digital pin led6 to false
  set digital pin led7 to false
  wait blinkMore secs
```

Figure 50. Code to blink all 7 LEDs ON and OFF

Creating the Blink_list sketch

Create a new sketch from the file menu or use the keyboard shortcut *Ctrl+N* (*command+N)*. Call this new sketch *Blink_list*, and save it into your projects folder by selecting Save from the File menu or hit *Ctrl+S (command+S)*.
Create a *blinkMore* variable as you did in the previous lesson, and then create a second variable called *led_pins*.
Once you have done this, create a third variable and name it *turn*. We will use this variable to keep track of how many turns our control structure takes.
Drag on a ***When clicked*** block and then set up your code blocks the same as the image in **Figure 55**.
You will need to set up a list structure to hold all of your LED pins.
To do this, drag a ***list*** block from the *Variables* library onto a ***set_to*** block (see **Figure 51**). Add the 7 numbers for each of the LEDs to this list block by clicking on the right-arrow in this block.

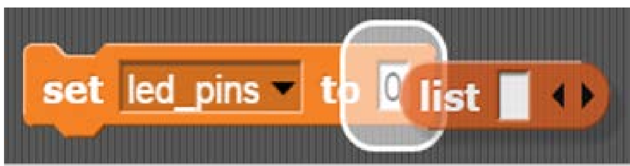


Figure 51. A list block can be dragged onto a set_ to block to change this variable into a list

The ***repeat until*** block can be found within the *Control* blocks library, and the *equals* operator can be found within the *Operators* blocks library.
You will also need to use the ***item_of_*** block which can be found within the *Variables* library.
Drag this ***item_of_*** block onto the first space of a ***set digital pin*** block (**Figure 52**).

Figure 52. Placing an item_of block inside a set digital pin block

Drag the turn variable onto the first space of the ***item_of_*** block (Figure 53).

Figure 53. Placing the turn variable inside the item_of block

Then drag the ***led_pins*** variable to the second space of the ***item_of_*** block (**Figure 54**).

Figure 54. Placing the led_pins variable inside the item_of block

Remember that there is a video tutorial available on the Firebugs Youtube channel to help you with this lesson.

Figure 55. Code blocks in the blinkList sketch

Once you have set up your code blocks the same as pictured in **Figure 55** connect your board and run this sketch.
If you have constructed the blocks correctly all 7 LEDs should blink ON for one second and OFF for one second repeatedly, just as they did when the sketch in **Figure 50** was run.
The sketch in **Figure 50** uses more than 24 blocks, whereas the sketch in **Figure 55** only uses 15 blocks. While this may not seem much of a difference, if you needed to blink 1000 LEDs instead of only 7, you would save a lot of time when writing out the code.

The turn variable

Figure 56. The *turn* variable used as a counter to add 1 to itself each time

Notice within our sketch that we have used the ***set_to*** block at the end of each ***repeat until*** block to change the turn variable by one for each turn of this loop. This is to ensure that the loop only executes 7 times; one turn for each of our 7 LEDs (until turn = 8).

Figure 57. Using a *set_to* block to reset the *turn* variable

Notice also that we have used a further ***set_to*** block after each of the ***repeat until*** block to reset the *turn* variable back to 1. If we did not reset this variable after the repeat blocks then only the first repeat until loop would run, so the LEDs would turn ON but not OFF, because the turn variable would remain at the value of 8.

Accessing items in the list

We mentioned that items in a list are counted from 1, which means that pin number 2 in our led_pins list is the 1st *element*.
We count the elements inside our led_pins list like this:

Element	1	2	3	4	5	6	7
Variable (LED pin number)	2	3	4	7	11	12	13

Table 2. Elements inside the *led_pins* list

Within each of the ***repeat until*** blocks our code sets a digital pin by using the turn variable. On the first turn through this loop the *turn* variable is equal to 1, and so the code sets the 1st element of the *led_pins* list to TRUE, which turns the LED at pin 2 ON. On the second turn through this loop the *turn* variable is equal to 2, and so the code sets the 2nd element of the *led_pins* list to TRUE, which turns the LED at pin 3 ON. It does this until the last turn through this loop where the *turn* variable is equal to 7, and so it finally sets the 7th element of the *led_pins* list to TRUE, which turns the LED at pin 13 ON. After this the *turn* variable is equal to 8, so the loop ends.

Now try this

If you look at the stage window you will notice that the turn variable always remains at 8. This is because the code is *executing* (running) so fast that the changes cannot be seen.
Place a ***wait*** block at the end of each of the two ***repeat until*** blocks (**Figure 58**) and observe what happens with the LEDs on the ARD2-INNOV8 and to the *turn* variable in the *Stage* window.

Figure 58. Placing a *wait* block at the base of the *repeat block*

What else can you do within this sketch now that you understand how these structures work?
Can you change the code so the LEDs turn ON in order from led1 up to led7, and then turn OFF in order from led7 down to led1, with a half second delay between each step?

What other light patterns can you create?
Here is another pattern. Look at the code and see if you can work out what this sketch does.

Figure 59. A variation of the *blink_list* sketch

Content Index

Lesson 4 – Introducing chance

Key words: Random, input/output, pseudo-code
Key focus: IF/ESLE blocks, using the random function, using a button input, using simple data structures & control structures

Difficulty Level

The learning in our lessons so far has been very valuable, however, everything has been very predictable. When asking young people interested in programming what kind of programmer they would like to be, most would say that they want to be a *games programmer*. To be a good games programmer you need to understand how to use *chance* in your programs; therefore you need to know how to generate and use *random numbers*.
In this lesson you will learn how to generate and use random numbers within the structures you used in the previous lessons, to create a very simple random game.

Adding a button

So far in our lessons we have only used *outputs* in the form of LEDs, however, in this lesson we are going to learn how to use an *input* in the form of the pushbutton attached to pin 8 on our ARD2-INNOV8 shield.
We will start by creating a very basic sketch which will just turn an LED ON when the button is pressed (when the D8 button on the ARD2-INNOV8 is pressed it pulls the pin to *false / LOW*).

The *pseudo-code* for this simple button sketch looks like this:

```
LED Button Sketch
WHILE
Read button status
           IF button status is equal to LOW
                SET led1 to TRUE
           ELSE
                SET led1 to FALSE
           ENDIF
ENDWHILE
EXIT
```

If you look back at the pseudo-code in **Lesson 1**, you will notice that the code for this lesson is different because it includes an *IF* and *ELSE* statement. The *IF/ELSE* statement blocks are *control structures* which allow the program to test if a condition is TRUE or FALSE, if a value is equal or not equal, or if a value is greater or less than a value being tested.

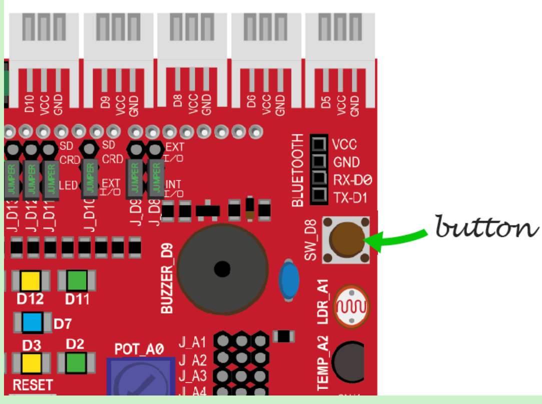


Figure 61. The push button on the ARD2-INNOV8 which is connected to digital pin 8

The text-based Arduino code for this sketch, looks like this:

```
int led1 = 2; // variable to hold the led1 pin
int button = 8; // variable to hold the button pin
bool buttonState = HIGH; // a variable to hold the
state of the button - HIGH or LOW, true or false
void setup(){
pinMode(led1, OUTPUT); // initialise led1 pin as an output
pinMode(button, INPUT); // initialise button1 as an input}
void loop() {
buttonState = digitalRead(button); // read the
state of button1 and store this value
// if the button state is reading low (pressed)
then set led1 to high, else, set led1 to low
if(buttonState == LOW){
     digitalWrite(led1, HIGH);}
else {
     digitalWrite(led1, LOW);}
}
```

Create the sketch

For this sketch we need to create a new variable for the button pin, as well as a variable to hold our LED pin.

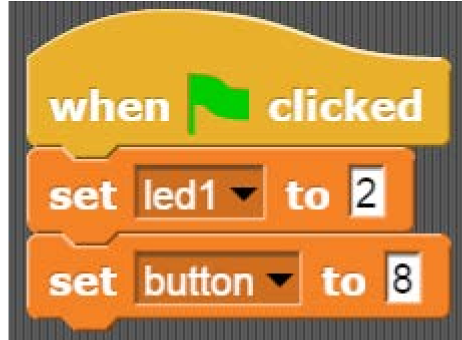


Figure 62. The top of the *buttonBlink* sketch

In this sketch we will be using the ***IF/ELSE*** control block.

Figure 63. The *IF/ELSE* block

We will also need to create a third variable to hold the value of the **state** of our button, so that we can check if it is TRUE or FALSE. We will call this variable *buttonState*.

Figure 64. Creating a variable called *buttonState*

We will use this variable inside our ***forever*** loop to read the state of the button by using a ***set_to_*** block with a **digital reading** block dropped onto it (**Figure 65**).

Figure 65. Placing a *digital reading* block inside a set*_to block* to read the button

We will need to drop the button variable on to this digital reading block so that this set of blocks reads and stores the state of the button – TRUE or FALSE (**Figure 66**).

Figure 66. Placing the button variable inside the digital reading block

Apply your knowledge

Apply your understanding of the code above by creating this simple LED Button sketch.
Copy the blocks in **Figure 67** and then run and test your code.
Debug your code if there is a problem, and modify the code to improve it.
Try changing the value of the *led1* variable and observe what happens.

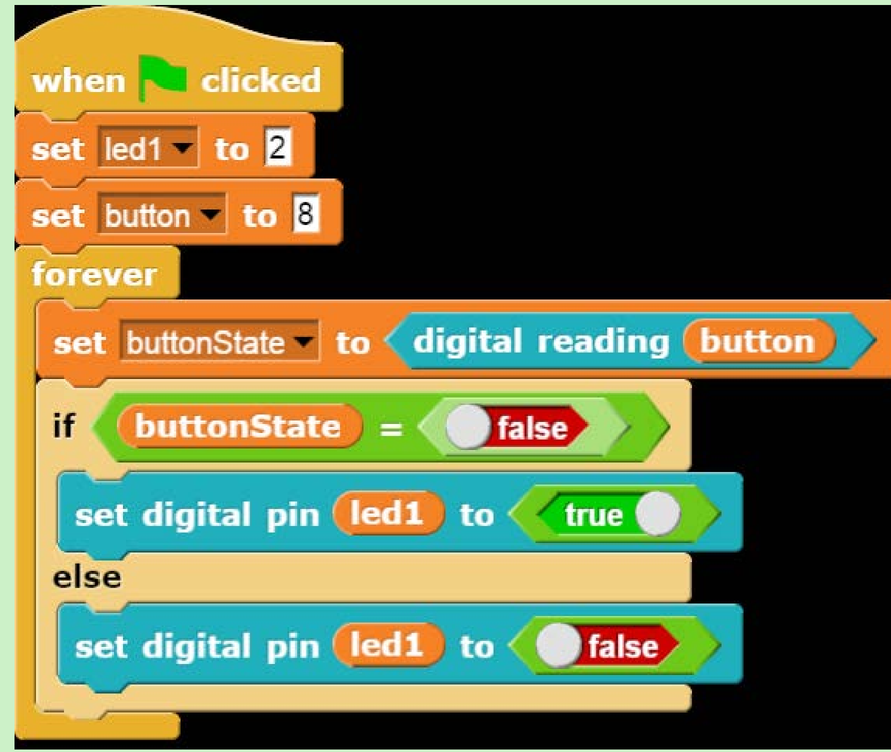


Figure 67. Code blocks in the *buttonBlink* block

Content Index

The Chance sketch

Now that we understand how to use a button in our sketch, we can introduce the element of *chance* by using the ***pick random*** function block. This block picks a random number between the numbers specified on the block, for example:
This block will give us a number between 1 and 10 (inclusive):

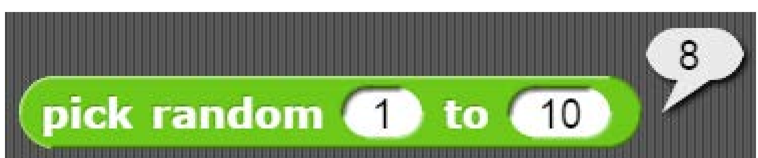


Figure 68. The *pick random* block – random number from 1 to 10

This block will give us a number between 3 and 7 (inclusive):

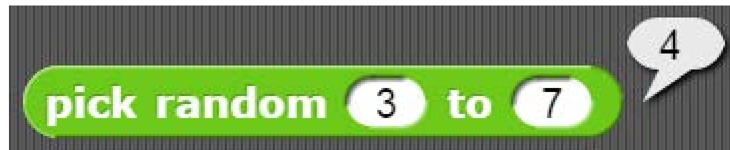


Figure 69. The *pick random* block – random number from 3 to 7

Which means that it will only return the number 3, 4, 5, 6, or 7.

If you click on the block it will give you a random number in a *speech bubble* beside the block. In **Figure 69** the block has *returned* the number 4 when it was clicked.
You will aslo need to create a new variable to store this returned value; call this new variable *random*.
In this *Chance sketch* we will be using all 7 LEDs, and we want our randomly chosen LED to stay ON after the button is pressed, so instead of using an ***IF/ ELSE*** block, we will use an ***IF*** block. We will use one of the repeat until blocks that we constructed in **Lesson 3** to turn all of the LEDs OFF just after the button is pressed each time. This will ensure that only one randomly chosen LED will light each time the button is pressed.
You will also notice that we have changed the *blinkMore* variable from 1 second to 0.5 seconds. This is to provide just enough delay so that the button won't activate more than once when pressed.

Create the sketch

Construct this *Chance sketch* by copying the blocks in **Figure 70**.
Your system should light one of the 7 LEDs each time the button attached to digital pin 8 is pressed
Run, test, and debug your code and system if necessary.

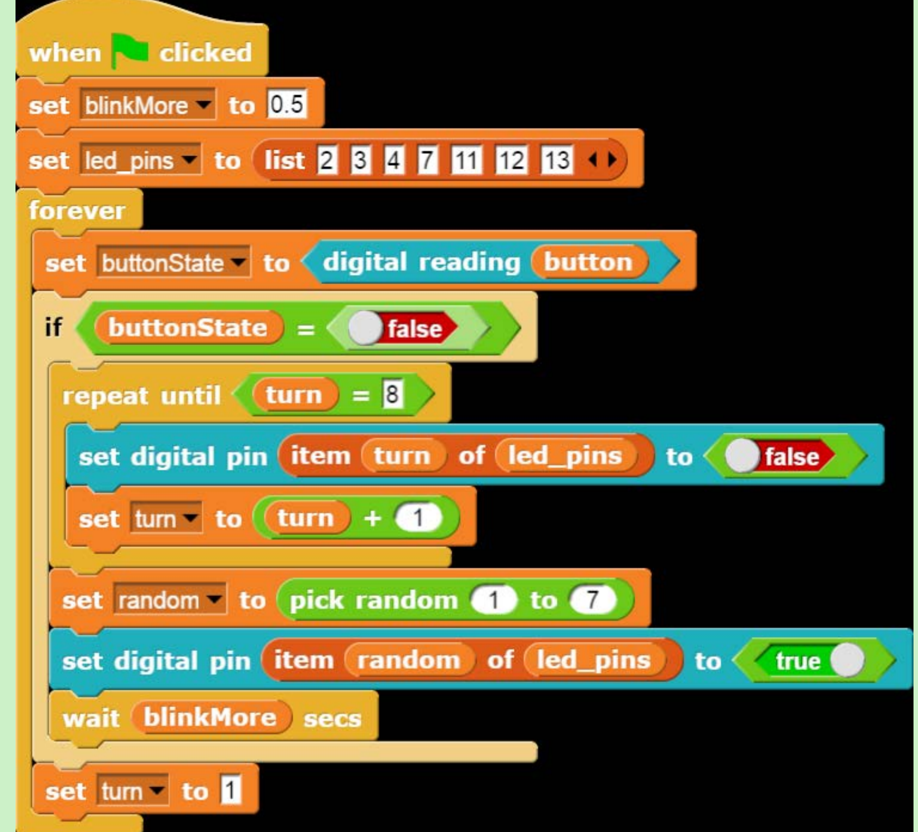


Figure 70. Blocks within the *Chance sketch*

Now try this

Did your code work as expected?
If not, debug both your code and system to fix the issue.

Now that you understand how this works, try adjusting your code so that two or more randomly chosen LEDs light each time the button is pressed.
Will there be any problems with this new system? Can you think of a way to fix this problem?
Could you design the code so that the randomly chosen LED rapidly flashes on and off until the button is pressed again? Might this cause a problem?

Lesson 5 – Dice

Key words: 2 dimensional array, Matrix, pattern
Key focus: pattern recognition, accessing data in a matrix (array), finding the length of a list, working with data structures & control structures

Difficulty Level

In the previous lesson we learned how a list could be used to store all of our LED pin variables, and how we could easily access any of these variables by calling the right element number inside our list using the ***item_of_*** block.

Figure 71. The *item_of_* block

We learned how the elements inside one of these lists is counted from 1 (most text based languages count from zero), and we learned how we could use the ***repeat until*** code block to perform repetitive tasks (*iteration*).
In this lesson we are going to combine these two programming tools to produce a system which will represent our randomly chosen number in visual form.
A standard dice is one of the oldest and simplest random number generators, and its number symbols are easily recognisable by people from all cultures (Correct English for a singular dice is die, however, dice is commonly used for both singular and plural these days).

We will be basing our system on the traditional dice, but to do this we will need to understand how to use a *two dimensional array*, or what is also known as a *matrix*. This may sound very difficult to understand, but you have actually already used a 1 (row) by 7 (column) *matrix* in **Lesson 4** in the form of a *list*. The simple explanation is: a 2D array or matrix is just a list which contains other lists.
In text based programming and mathematics, a matrix can only be rectangular, meaning that there can be no empty spaces or missing elements.

1		
1	2	
1	2	3

This is not a proper matrix
There are missing elements

1	2	3
1	2	3
1	2	3

This is a proper matrix
There are no missing elements

These rules however, are not followed by visual-block coding languages like Snap4Arduino and others. These languages break the rules so that you can have uneven matrices in your programs.
To make our dice system we will need to create one *list* for each of the numbers on our dice; and we will need to store all of these *lists* inside a further *list* to make all of the variables easy to access.
The light pattern which we need to make for each of our dice numbers looks like this:

ONE

LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

TWO

LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

THREE

LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

FOUR

LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

FIVE

LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

SIX

LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

Table 4. Patterns for each of the 6 dice numbers

If we put this data (using the pin numbers) into lists, then these lists will look like this:
ONE = {7}
TWO = {3, 12}
THREE = {3, 7, 12}
FOUR = {2, 4, 11, 13}
FIVE = {2, 4, 7, 11, 13}
SIX = {2, 3, 4, 11, 12, 13}

You can see that these lists are not even in size; the first list has 1 element and the last has 6, however, Snap4Arduino will allow us to add all of these lists into another list to create our matrix.

Creating the Dice sketch

In this sketch we will use several of the blocks which we created in the previous lesson.
To make the creation of this sketch easier, you can open the *Chance sketch* from **Lesson 4**, and then choose *Save As…* from the *File* menu and save the sketch as *Dice.*
For this sketch you will need to use the following variables which you created in **Lesson 4**:

Figure 72. Variables from Lesson 4 needed for the *Dice sketch*

You will also need to create these new variables:

Figure 73. New variables needed for the *Dice sketch*

You will need to set each of your dice number variables up as a *list* to hold the pattern for each number, for example, you will set the number 3 up like this by dragging a ***list*** block on top of a ***set_to*** block:

Figure 74. The block setup for the variable three

The first 10 lines of your code should look like this:

Figure 75. The lists defined at the top of the *Dice* sketch

To create the matrix, you will need to place all of your number lists into a the dice variable list.
You do this exactly the same way that you created each of the lists for your numbers, however, this time, instead of placing a number in each box of the list, this time you drag each of the dice number variables onto it like this:

Figure 76. Dropping variables into the *Dice* list

Most of the code in this sketch is the same as it was in the sketch for **Lesson 4**, such as the button code and the code to turn all of the LEDs off, however, this time we are generating a random number from 1 to 6 instead of 1 to 7, and because each of our dice number lists are different sizes, we need to find a way to tell our program the size of each list is so that it runs through the correct number of loops for each of these lists. This is where variables really come in handy.
We can use the ***length of_*** block to find the length of the randomly chosen list, and we can store this value in our ***num_length*** variable.
Don't worry if you don't understand this, because this is more advanced programming. This type of programming is best learnt by doing, and by trial and error, so if you don't understand it now, keep practicing until you do.
Remember also that this lesson along with all of the other lessons in this book are able to be viewed on the Firebugs Youtube channel.

Build the blocks and test your sketch

Build all of the blocks for this Dice sketch by copying the code structure in **Figure 77**.
This structure is a lot more complex than the other sketches you have created so far, so pay careful attention to how each block group has been created.
Once you have completed the blocks, connect your Arduino, click on the ***When clicked*** block and test your program.
With each push of the button, you should see the LEDs light in a pattern representing the randomly chosen number.
If this is not happening, debug your code and test the system again.

```
when [flag] clicked
set blinkMore to 0.5
set turn to 1
set led_pins to list 2 3 4 7 11 12 13
set one to list 7
set two to list 3 12
set three to list 3 7 12
set four to list 2 4 11 13
set five to list 2 4 7 11 13
set six to list 2 3 4 11 12 13
set dice to list one two three four five six
forever
  set buttonState to digital reading button
  if buttonState = false
    wait blinkMore secs
    repeat until turn = 8
      set digital pin item turn of led_pins to false
      set turn to turn + 1
    set turn to 1
    set random to pick random 1 to 6
    set num_length to length of item random of dice + 1
    repeat until turn = num_length
      set digital pin item turn of item random of dice to true
      set turn to turn + 1
  set turn to 1
```

Figure 77. Blocks within the *Dice* sketch

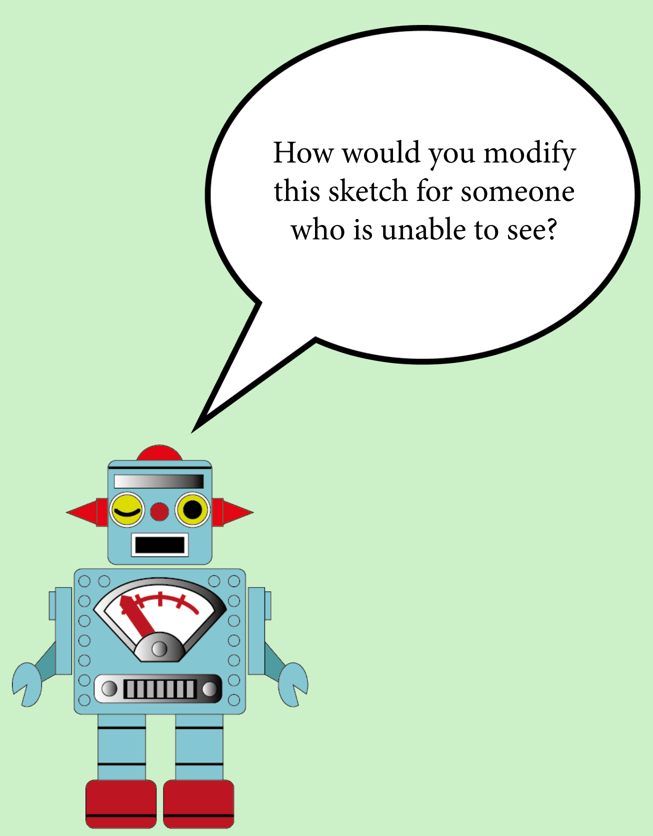
Now try this

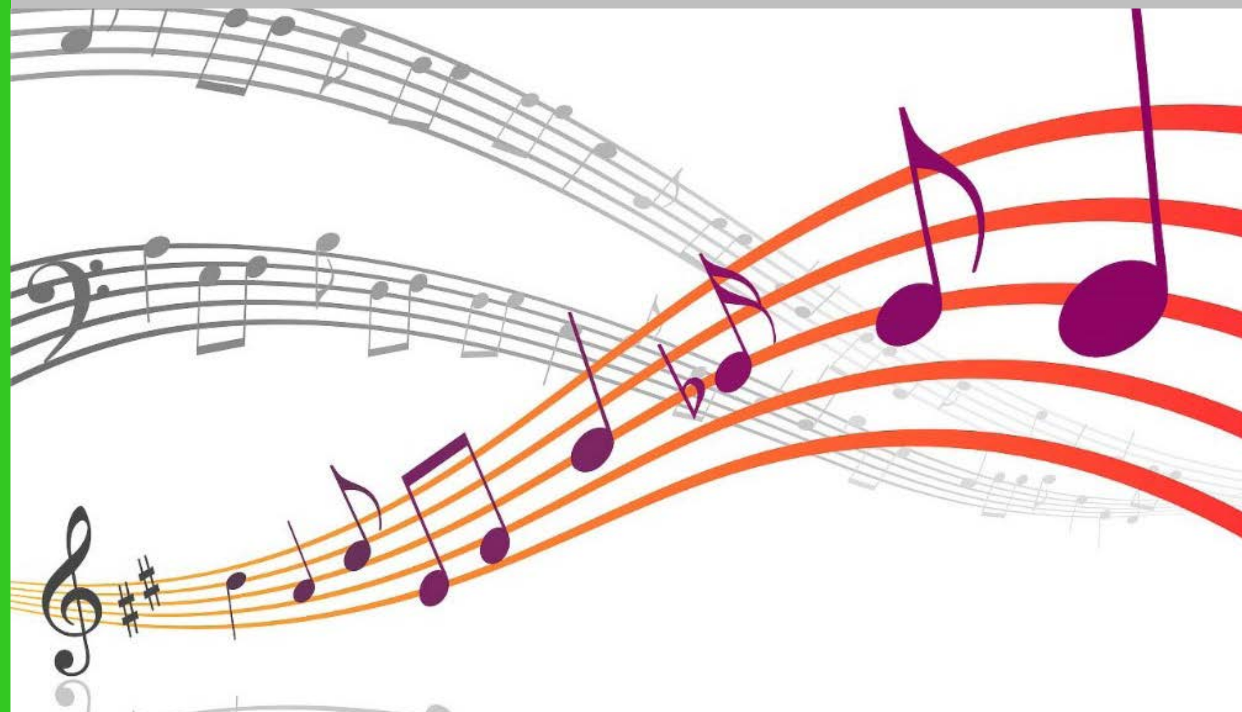
There are a number of things which can be added to this sketch to make the function more interesting, and there are different ways that this *Dice* system can be built.

Modify your code so that each time the button is pressed the LEDs flash from LED 1 to LED 7 before the final dice number is shown.

Modify your code so that each time the button is pressed the LEDs flash a number of random numbers before finally settling on one number.

We will learn how to use sound in the next lesson, however, if you wish to be innovative and adventurous, try adding sound to your Dice system to see if you can modify it to produce a number of beeps with each randomly chosen number.





Lesson 6 - Variable Tones

Key words: pitch, duration, digital, analog, potentiometer
Key focus: using analog data, using the map function

Difficulty Level

Producing sound

In the previous lessons we learned how to use an *input* in the form of the ARD2-INNOV8's push button, and *outputs* in the form of the ARD2-INNOV8's coloured LEDs.
In this lesson we are going to learn how to produce sound as an output for our system.
The ARD2-INNOV8 shield has a built in *buzzer* attached to digital pin 9, which produces quite good sound, however, because we are using Snap4Arduino we will be unable to use this buzzer in this book. The code needed to use this buzzer is a little too complex for the learning in this book, requiring additional software to be loaded onto the Arduino board.

Luckily the Snap4Arduino IDE has a built in Sounds section which will allow us to play sounds through our connected computer. The sounds produced by this method are of high quality, and you can even create and use your own sound files library.

A basic sound program

In this simple sketch, we are going to use some of the code from our previous lessons. This sketch will be very similar to our button sketch in **Lesson 4**, where we used a button to turn a LED ON. In this sketch, however, we want to produce sound instead of light, so we will replace the code which lights the LED with code that produces a sound. The block that we need for sound is the ***play note_for_ beats*** block, which can be found in the Sound library.

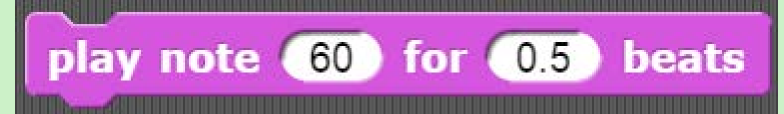


Figure 78. The *play note_for_* block

This block lets you choose the *pitch* and *duration* of the sound.
Most of the code in this sketch you have seen and used before in **Lesson 4 and 5**.
Create the *button_sound* sketch by copying the code blocks in **Figure 79**, and then run and test it on your ARD2-INNOV8.

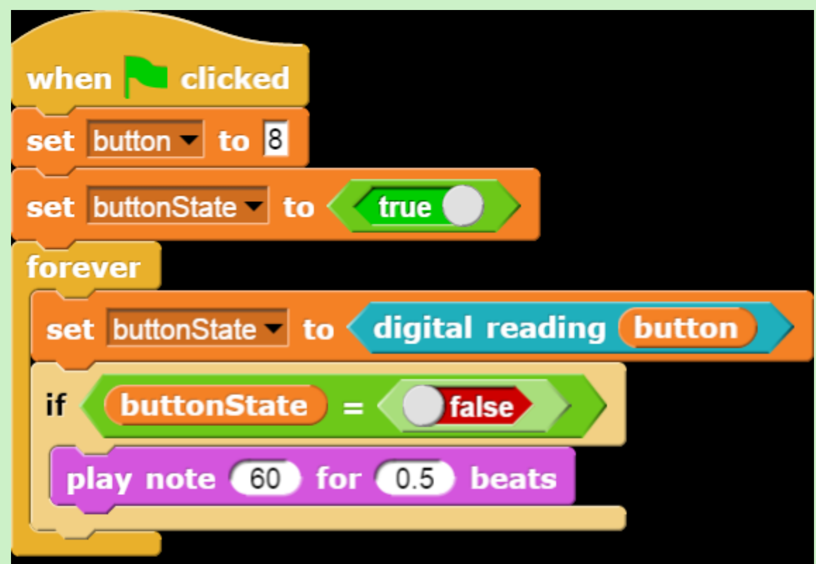


Figure 79. Code blocks in the *button_sound* sketch

Test, Evaluate, Improve

What could be improved with this system?
What happens when you hold your finger down on the button? Could you improve this?
Could you change the code so that the sound changed with each new button press?
Can you change the code to make this system more interesting?

Adding variation control

Digital and Analog

So far we have only used *digital* inputs and outputs with our ARD2-INNOV8 shield, however, this shield with help from Arduino, is capable of reading *analog* data, and outputting *simulated analog* data.
What is meant by *analog data*, and how does it differ from *digital data*?
Digital data has fixed values, and in the case of our Arduino board is a value of either 5 volts or 0 volts (HIGH or LOW, TRUE or FALSE).
Analog data does not have fixed values. It can be anywhere on a sliding scale. In the case of our Arduino board this analog data can be anywhere from 5 volts to 0 volts, meaning it could be 5, 4.5, 3.298, 2.004, 1, or 0 volts.
Data in the real world is analog data: the sun shining through the clouds, the sound of a bird singing, the amount of rain falling on a leaf.
The Arduino can convert this analog data to digital data by using an analog-digital-converter (ADC), which converts the input voltages from sensors to a number between 0 and 1023.
We can use this process to help us control the pitch of the sound that is played each time our button is pressed.
In this sketch we will need to create (*declare*) the following variables, and *define* them with pin numbers at the top of our code:

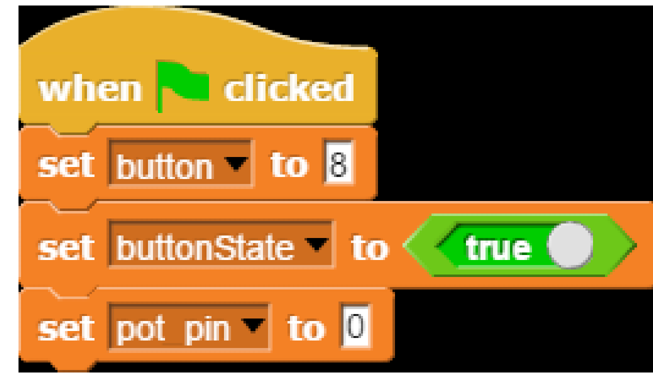


Figure 80. Variables *defined* at the top of the *variable_sound_button* sketch

Two of these variables: *button* and *buttonState*, you have used before, but the third: *pot_pin*, is new. This *pot_pin* variable will be used to help read the value on the *potentiometer* (blue knob) which is attached to pin A0 on the ARD2-INNOV8 shield. The potentiometer is like a volume control knob on a sound system, however, in this sketch we are using it more like a variable switch to change the *pitch* of our sounds.

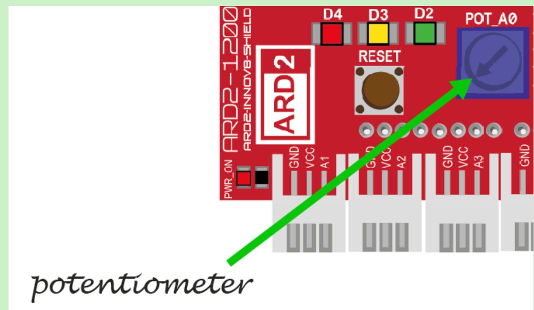


Figure 81. The *potentiometer* on the ARD2-INNOV8

In this sketch we will need to change the value of the data that is collected from this potentiometer, because the values coming from the potentiometer are from 0 to 1023, but the ***play note_for_beats*** function block works best with values from 60 to 100.
For this we will need to use the ***map*** function block.
You will need to import this block into your libraries from the **ARD2-INNOV8-SNAP** folder which you have extracted to your computer.
To do this choose *Import* from the *File* menu.

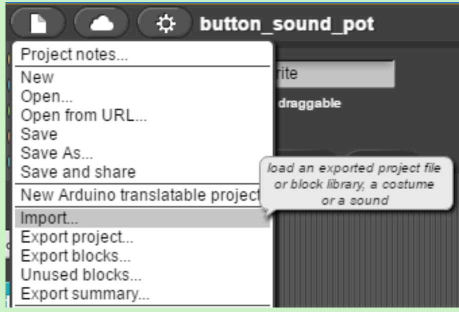


Figure 82. Import blocks via the *File* menu

Navigate to the ***map.xml*** file inside your **ARD2-INNOV8-SNAP** folder, and open it.

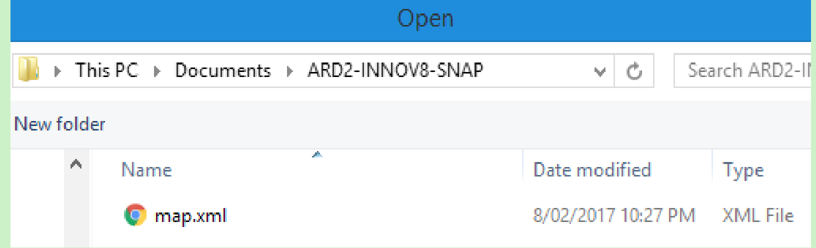


Figure 83. Navigate to the folder containing the *map* block

You should now be able to see the ***map from_to_*** block within the Operators library.

map from , to ,

Figure 84. The *map* block

We will use this block within a ***set (read_pot) to_ block*** to set the *read_pot* variable to the value of our potentiometer, and we will use the ***round*** function block to round off the value which is being read. The round function block will give us the value 67 instead of 66.67899, or the value of 100 instead of 99.98777.

```
set read_pot to round map read_pot from 0 , 1023 to 60 , 100
```

Figure 85. Setting the *read_pot* variable to a number between 60 and 100 by using the *map* and *round* blocks

Build and test the *variable_sound_button* sketch by copying the code blocks in **Figure 86**, and then run this on your board. For this sketch to work, you will need to turn the knob on the potentiometer and also press the button.

```
when clicked
set button to 8
set buttonState to true
set pot_pin to 0
forever
  set buttonState to digital reading button
  if buttonState = false
    set read_pot to analog reading pot_pin
    set read_pot to round map read_pot from 0 , 1023 to 60 , 100
    play note read_pot for 0.5 beats
```

Figure 86. Blocks within the *variable_sound_button* sketch

Now try this

The potentiometer on the ARD2-INNOV8 shield is not the only component capable of reading analog information. There is also the light sensor (*LDR*) and temperature sensor. All of the pins on the analog side of the shield are also capable of reading analog information.

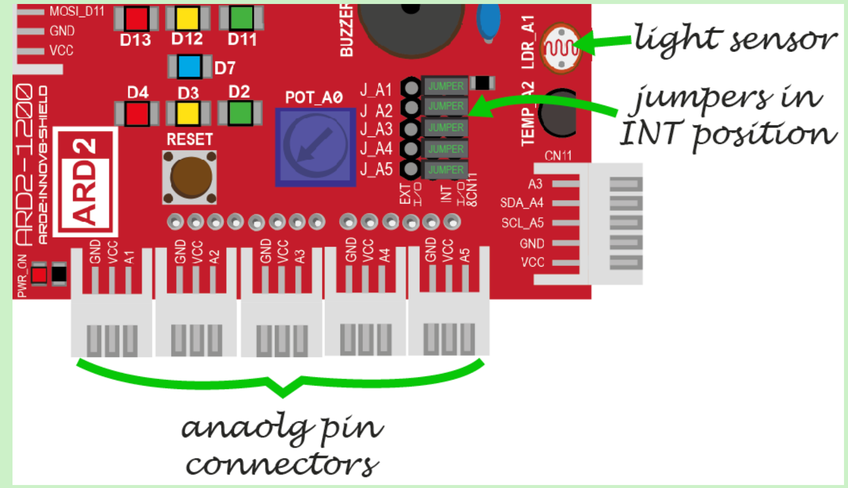


Figure 87. The ARD2-INNOV8 analog pins and pin jumpers (set to *INT*)

Make sure than all of the analog pin jumpers are set to the internal position (*INT*) [**see Figure 87**], then change the value of the *pot_pin* variable from 0 to 1 (**Figure 88**) to use the LDR instead of the potentiometer.

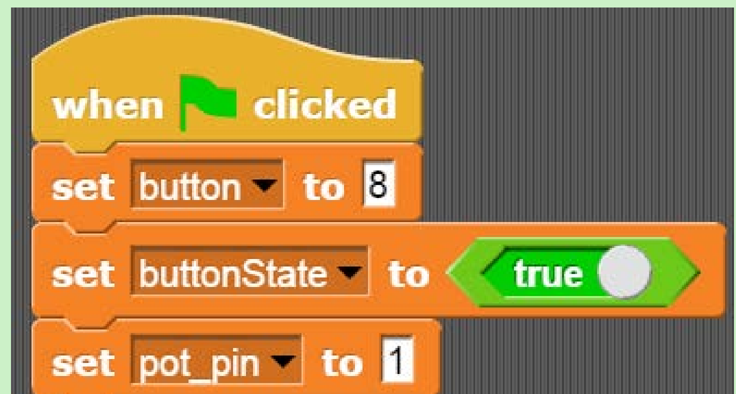


Figure 88. Changing the value of *pot_pin* from 0 to 1 reads the LDR instead of the potentiometer

Now instead of reading the value across the potentiometer, you are reading the value of light hitting the light sensor (LDR) attached to pin A1.
Shine a light over this sensor and push the button to observe what happens.
Try also to create a melody using a number of sounds rather than just one, and use either the potentiometer or LDR to change this group of sounds so that the whole melody changes in pitch.

Content Index

Lesson 7 – Banana Keyboard

Key words: pullup-resistor, resistance, frequency, threshold
Key focus: using control structures, working with thresholds

Difficulty Level

Now that we know how to use our code to make sounds, we can make something that is a lot of fun and a little unusual.
In **Lesson 4, 5, and 6** we used a push button to activate part of our code, however, there is only one button which we can use on our ARD2-INNOV8, so we will not be able to use buttons to make a musical keyboard.
Luckily we don't need any buttons for this because the ARD2-INNOV8 has been fitted with special resistors attached to each of the anolog pins from **A1 to A5**. These special resistors are called *pullup resistors*. We have already used one of these which is attached to our push button, however, the pullup resistors attached to the anaolg pins are much weaker.
These weak pullup resistors mean that we can use anything that will conduct a small amount of electricity to form part of a switch. The other part of the switch will be our finger, and so we will be using every day bananas as our piano keys.
The weak pullup resistors will *pull up* each of our analog pins to HIGH (TRUE) when the bananas are not being touched. Without this pullup the sounds would be switched ON and OFF randomly when the bananas are not being touched.

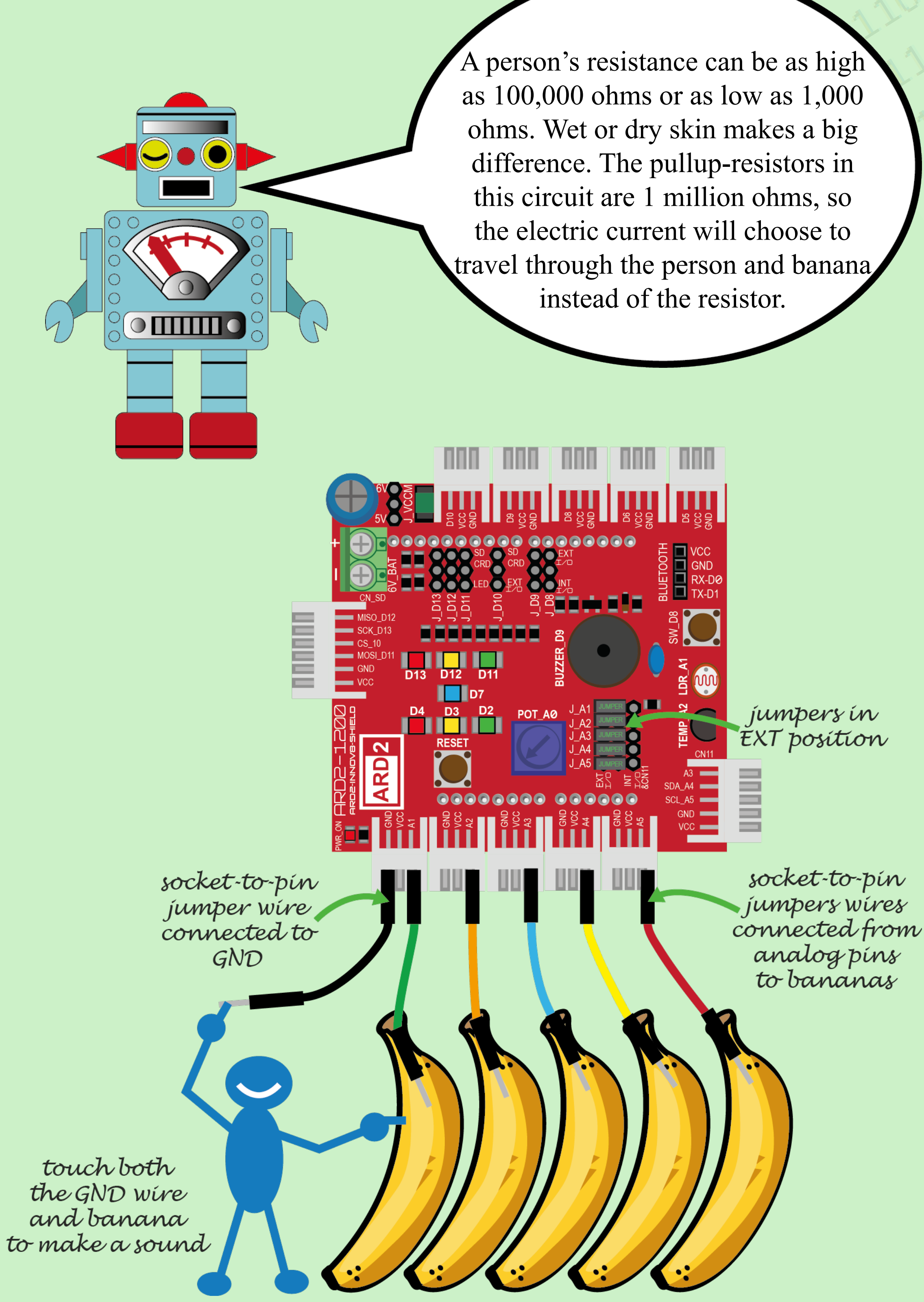


Figure 89. The setup for the *banana_keys* sketch

Creating the code for the banana_keys sketch

To create the *banana_keys* sketch we need to create a *list* and five variables.
The *notes* list will hold all of the sound values which will be played via the play ***note_for_beats*** block when one of our bananas is touched.
The five variables: *read_1, read_2, read_3, read_4, read_5*, will hold the values being read by each of our 5 anaolg pins. When these values drop below a certain point the code will react and a note will play.

Figure 90. Variables used for the *banana_keys* sketch

Our *notes* list can be set up to contain any notes that you wish, however, you will need to include one note for each of our five banana keys.
The exact values for each of the scaled musical octave can be seen in **Table 5**, however, we have adjusted our notes slightly to be like this (**Figure 91**):

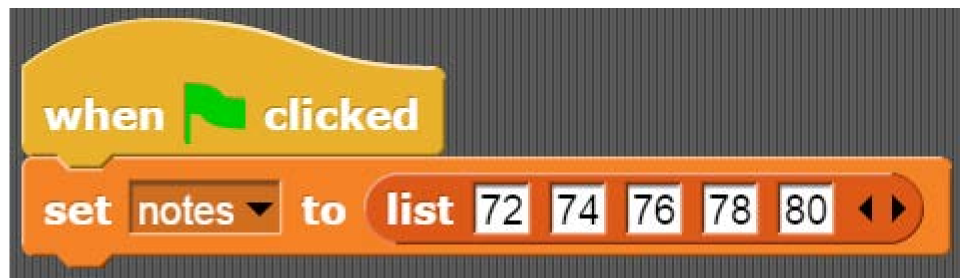


Figure 91. Musical note values within the *notes* list

Note	C_4 (mid C)	C♯$_4$	D_4	D♯$_4$	E_4	F_4	F♯$_4$	G_4	G♯$_4$	A_4	A♯$_4$	B_4	C_5
MIDI number	60	61	62	63	64	65	66	67	68	69	70	71	72
Frequency (Hz)	262	277	294	311	330	349	370	392	415	440	466	494	523

Table 5. The musical notes with MIDI number and frequency

You can download a free frequency meter for your smartphone or tablet and test to see if the frequencies played by Snap4Arduino match the ones in this table. You will find one of these apps by searching *"frequency meter"* in your phone's app store.

Creating the banana_keys sketch

In this *banana_keys* sketch we will be using some of the control structures that we have used in the previous lessons, and we will be using a *list* data structure to hold the values for our notes.
We will be using the ***forever*** loop block, as we have done for all of our sketches, and we will need to use an ***IF*** block to control what happens when each of the banana keys is touched.
Before we set up the control blocks in our sketch, we need to set up our *read* variables to ensure that they are reading the data coming in to each of our analog pins. We do this at the top of our forever loop, like this (**Figure 92**):

Figure 92. Setting of the *read* variable at the top of the *forever loop*

After these variables are set up to read the analog data coming in, we set up our control structures. This is done by using an ***IF*** block which tests against a specific threshold value. For each of our banana keys we need to set the code blocks up like this (**Figure 93**):

Figure 93. One of the *IF* control blocks in the *banana_keys* sketch

Here we see that if the value of *read_1* drops below the threshold value of *800* the first item in the *notes* list is played for half a beat. We set up our control blocks like this for each of our five analog pins.
The value of 800 may need to be adjusted to suit the resistance of the person playing the banana keyboard. Electricity flows differently through people depending on their body size and type. Also, having either moist or dry hands can affect the way electricity flows through the body. If you find that your system is not behaving as expected, change the value in this block from 800 to whatever works for you. Watch the video for this lesson on the Firebugs Youtube channel to see how this system should behave.

Test, Evaluate, Improve

Build the *banana_keys* sketch by copying the code blocks in **Figure 94**.
Before you run the sketch, make sure that you have your ARD2-INNOV8 set up the same way as shown in **Figure 89**.
Use 5 *socket-to-pin* jumper wires to connect the bananas to the analog pins on the analog side of your ARD2-INNOV8 (*A1, A2, A3, A4, A5*), and a further jumper wire for the ground wire (*GND*). Use the socket end to plug into the side connectors on the shield, and the pin end to pierce the skin of the bananas.
If all of these things have been done, connect your Arduino, then run and test the sketch.
You should now be able to play a nice tune on your bananas.
Debug and adjust the code to ensure that your system is functioning as expected.
If you can see an area that needs improvement, try your solution to see if it works.

```
when [flag] clicked
set notes to list 72 74 76 78 80
forever
  set read_1 to analog reading 1
  set read_2 to analog reading 2
  set read_3 to analog reading 3
  set read_4 to analog reading 4
  set read_5 to analog reading 5
  if read_1 < 800
    play note item 1 of notes for 0.5 beats
  if read_2 < 800
    play note item 2 of notes for 0.5 beats
  if read_3 < 800
    play note item 3 of notes for 0.5 beats
  if read_4 < 800
    play note item 4 of notes for 0.5 beats
  if read_5 < 800
    play note item 5 of notes for 0.5 beats
```

Figure 94. Blocks within the *banana_keys* sketch

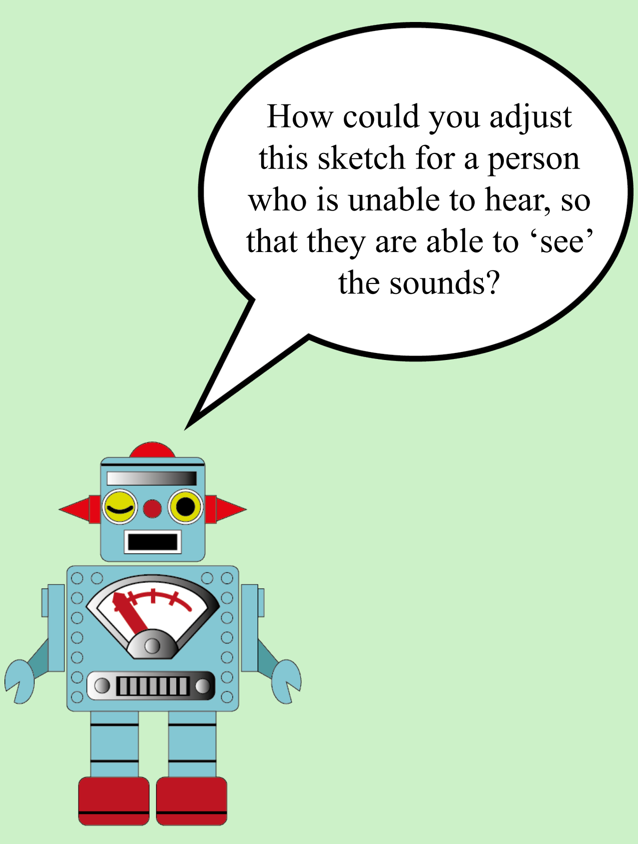
Now try this

This *banana_keys* sketch is fairly simple, because it only uses five *IF* control blocks, however, there is a way to make it even simpler and more elegant, by using the one of the ***repeat*** control structure blocks that we have used in some of the previous lessons. Try to adjust your code to reduce the number of *IF* blocks by adding a ***repeat*** block. A solution for this can be found on the Firebugs website under Snap4Arduino, and the code for this is in your ARD2-INNOV8-SNAP folder.

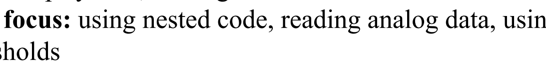
Using your knowledge from the previous lessons:

- Create a system which produces both a sound and the lighting of one of the coloured LEDs on the ARD2-INNOV8 each time one of the bananas is touched.
- Create a system where the sounds for this banana keyboard can be adjusted in pitch by either turning the potentiometer of shining a light on the LDR.

If you think up a unique solution, post it on the Firebugs and Wiltronics Facebook page; you could win yourself a small prize for your effort.
Solutions for these challenges can be found on the Firebugs website under Snap4Arduino.



Lesson 8 - Temperature Sensor



Key words: degrees Celsius/Fahrenheit, closed-loop/open-loop system, nesting
Key focus: using nested code, reading analog data, using thresholds

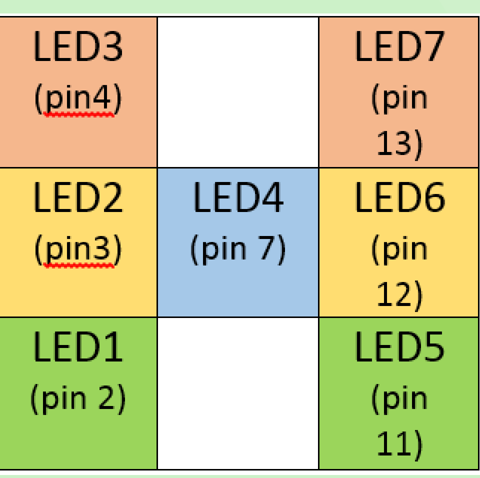
Difficulty Level

We have learnt how to write some complex code so far, and we had a bit of fun applying this in **Lesson 7** with our banana keyboard.
In this lesson we are going to make something which is a little more serious, and a system which is very useful in everyday life.
We are going to create a sketch for a system which will use the temperature sensor on our ARD2-INNOV8 to indicate a temperature range to us.
To do this we will be using the TMP36 temperature sensor on our ARD2-INNOV8 along with the coloured LEDs.
We will be making a temperature sensor which reads temperatures in degrees Celsius, however, this is very easy to change if you live in a country which uses Fahrenheit.
The system that we create in this lesson will be able to work even when a person is not present; meaning that the system will continue to read the temperature when it is on its own. We call these automatic systems *closed-loop systems*, and systems which need a human controller are called *open-loop systems*.

Creating the temperature sketch

Firstly we need to *declare* five variables: *led_cold, led_warm, led_hot, temp_read, and temp_C* (change this to temp_F for Fahrenheit) [**Figure 95**].

Figure 95. Variables used in the *temperature* sketch



LED3 (pin4)		LED7 (pin 13)
LED2 (pin3)	LED4 (pin 7)	LED6 (pin 12)
LED1 (pin 2)		LED5 (pin 11)

Table 6. ARD2-INNOV8 LED pin numbers and colours

The LED variables get *defined* at the top of our code, using the relevant pin numbers as values.

Figure 96. The LED variables at the top of the *temperature* sketch

We once again use the ***forever*** loop like we have done in all of our lessons, but this time we are going to make use of two ***IF/ELSE*** control blocks, by *nesting* one inside the other. *Nesting* is a very common practice in computer programming that allows you to have precise control over how your code flows.
Above these ***IF/ELSE*** blocks we first need to make our program read, convert, and store the data coming into the temperature sensor, like this:

```
forever
  set temp_read to analog reading 2
  set temp_read to (temp_read × 5.0) / 1024
  set temp_C to (temp_read − 0.5) × 100
  set temp_F to (temp_C × (9.0 / 5.0)) + 32
```

Figure 97. Reading the temperature sensor and converting the data

The first block at the top of the forever loop reads the data from the temperature sensor on analog pin 2, then the next block converts this data to a voltage between 0 volts and 5 volts, the third block converts this voltage to a value in degrees C, and the final block converts the Celsius data to Fahrenheit (**Figure 97**).
The control section of this sketch has two ***IF/ELSE*** blocks (one inside the other) giving us the three paths that our program can take. Two of these options contain a threshold value to be tested, and the final ***ELSE*** statement within the first ***ELSE*** statement gives the default option when none of the other conditions are met.

This is what this looks like written in pseudo-code:

```
Temperature Sketch
WHILE
Read temperature
      IF temperature greater than 28
            SET led_hot to TRUE
            SET led_warm to FALSE
            SET led_cold to FALSE
      ELSE
            IF temperature less than 25
                  SET led_hot to FALSE
                  SET led_warm to FALSE
                  SET led_cold to TRUE
            ELSE
                  SET led_hot to FALSE
                  SET led_warm to TRUE
                  SET led_cold to FALSE
            ENDIF
      ENDIF
ENDWHILE

EXIT
```

Test, Evaluate, Improve

Build the temperature sketch by copying the blocks in **Figure 98**. Run the sketch, then hold your finger on the temperature sensor and watch the LEDs. The BLUE LED should be lit first, and then as you keep your finger on the temperature sensor it should change from BLUE to YELLOW to RED.
You may need to adjust the threshold values in the ***IF/ELSE*** blocks to suit the temperature conditions of your local environment. The temperature thresholds which we have used here should work when you hold your finger on the temperature sensor.

You can watch the temperature variables change within the *Stage* window. This is helpful if you need to adjust the threshold values (**Figure 99**).

Figure 99. You can watch the temperature variables change in the *Stage* window

Figure 98. The *temperature* sketch

Now try this

In the real world this type of system could be used to do a number of useful things, such as turn on a fan, or open a window when the temperature becomes too hot. This type of system is often found in agriculture, and in particular they are used in greenhouses to help control temperature.
Use the skills and knowledge gained in this lesson and previous lessons to create a useful or interesting system. You could introduce sounds and other variables to create interesting outputs for your system. You could use your understanding of how the potentiometer works to create a system which has a variable threshold.
Work together with your classmates to brainstorm, plan, and design a system which uses sensors such as the temperature sensor and LDR to read data, and then provides outputs in the form of light and sound.

Lesson 9 - Traffic lights

Key words: synchronisation, abstraction, encapsulation, local/global variables, parameters & arguments, algorithm, sequence
Key focus: defining & calling functions, pattern recognition, algorithm design, computational thinking

Difficulty Level

This lesson is perhaps the most challenging lesson so far. The idea for this sketch is very simple, however, the second part of this lesson will definitely test your thinking and programming skills.
This lesson is separated into two parts: **Part A** is the creation of a single traffic light program, which is quite easy, and **Part B**, which is the creation of a double traffic light program.
The programming of **Part B** will require you to design the flow of your program so that the two traffic lights work in perfect *synchronisation*.
The main learning focus in this lesson is how programs can be simplified with the creation of functions.
Programmers create functions within their programs to act as containers for parts of their code, so it can be access again and again by other parts of the program.
Creating functions means that you can hide all of the complicated parts of your program in another section (or even in a separate file), so that your code looks neater and is easier to follow.
Two programming terms that are used to describe this process are *abstraction* and *encapsulation*.

We use *abstraction* when we eliminate any unnecessary or repetitive code and place code parts into functions so that the working details are hidden away. We are using *encapsulation* when we create functions because we are enclosing parts of the program within a container so that this code is protected and cannot be changed by other parts of the program when the code is run.

Creating your own function blocks

When creating complicated programs the code can become complex and messy. In these cases programmers create functions to hold and protect parts of the program, and to make the code easy to follow.
Snap4Arduino also allows you to create your own function blocks. Just like with text-based code, it is easy to create a function block with Snap4Arduino code, because it works much the same as building regular code blocks.
In this lesson we will be creating two functions to create our traffic light program.

Building a basic function block - example

Before we begin our traffic light sketch, we will create a simple function as an example so that you can understand how this process works.
Within the *Other* blocks library, you will find a button labelled *Make a block* (**Figure 100**).

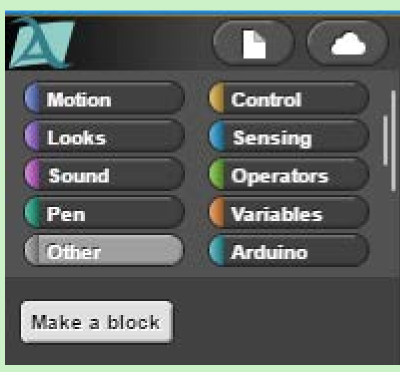


Figure 100. The Make a block button within the *Other* block library

Clicking on this button brings up the *Make a block* window, which will let you choose a library for your block, and the type of block. Here we will choose the *Arduino* library, and *Command* block, because we want this function block to perform a command with the Arduino board. We will name this block *Blink* and we will select the *for all sprites* option (**Figure 101**) [this means we can use it with any attached board].

Figure 101. Creating a function block in the *Make a block* window

Content Index

Next we are taken to the *Block Editor*, where we can begin to build our function block (**Figure 102**).

Figure 102. The *Block Editor* window

Local Variables Vs Global Variables

In our function blocks we will be using *local variables*. These are variables which can only be used inside their function, and therefore cannot be used by other parts of the program. In all of the lessons in this book, you have been using *global variables*. *Global variables* are variables which can be used by all parts of the program, and therefore they must be declared (created) at the top section of the code.
We can build our *function* blocks as we have done in previous lessons, and we will be able to create and access local variables within these blocks which will allow us to use these blocks in other sketches; We can even share them for use with other programmers (this will make sense when you start building). Here we have recreated our original *Blink* sketch from **Lesson 1** as a new function block (**Figure 103**). Notice that we have used two local variables as *parameters* to this function. This means that in our top block we have included the variables *led* and *pause*, and we have used these variables within the functions other blocks.
A *parameter* to a function is a variable which tells the function what input to expect. You can set up a function with as many parameters as you need.

+Blink+ led + pause +
set digital pin led to true
wait pause secs
set digital pin led to false
wait pause secs

Figure 103. Our *Blink* function created in the block editor

Defining a function means that you create the code to allow the function to be used in your sketch.

Local variable inputs can be created by clicking on the *plus* **+** signs next to the function name on the top block (**Figure 104**).

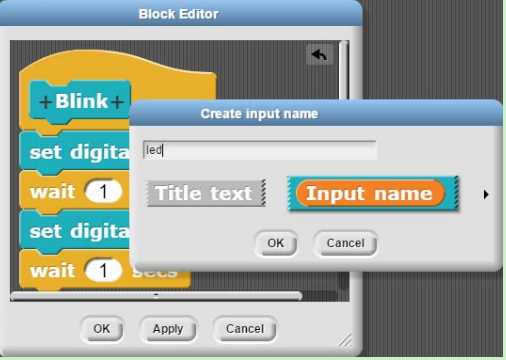


Figure 104. Creating *parameters* (inputs) for the function

These local variables can then be dragged from the top block to places where they are needed within the function code (Figure 105).

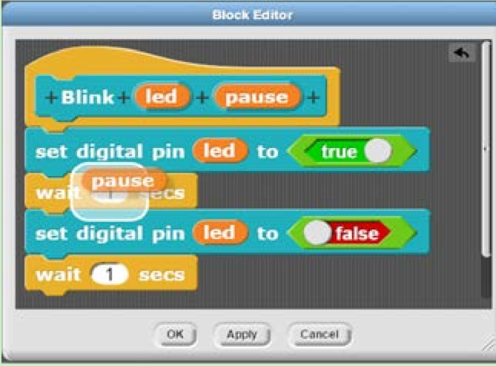


Figure 105. *Local variables* are dragged into their required positions

This process *declares* and *defines* the function so your block will appear within the *Arduino* blocks library, but the function will only be run if it is *called* within a control block in the programming window. To do this you simply drag the block into the programming window as you have done with all other blocks, like this (**Figure 106**).

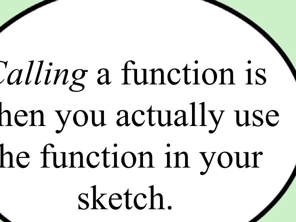


Figure 106. *Calling* the *Blink* function within the *forever* loop with the *arguments* 13 and 1

You will need to give your new function the *arguments* that it expects. An *argument* to a function is the input which it expects to use. Your *Blink* function expects two *arguments*: the pin number and pause amount. This is because we set up this function with these two input *parameters*.
This sketch will blink the LED on pin 13 at one second intervals.
You can also use your own variables as *arguments*, and you can use this block with any of the LED pins and as often as you need (**Figure 107**).

Figure 107. An example of how the *Blink* function can be called

Build this example sketch and experiment by *calling* it with different *arguments* like the example in **Figure 107**.

Creating the single traffic light sketch

The LEDs on the ARD2-INNOV8 shield are set up in a way which provides two traffic light patterns.

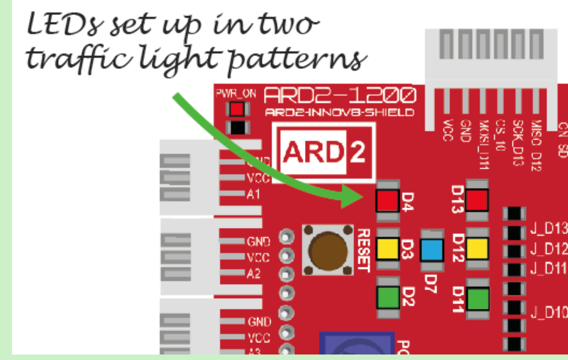
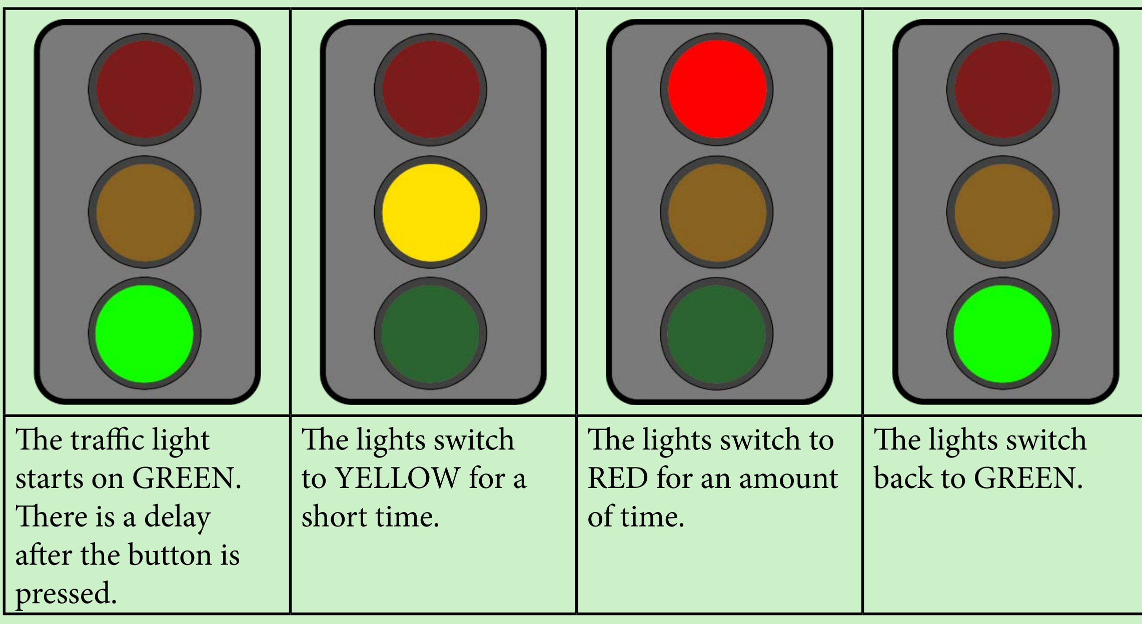


Figure 108. Traffic Light LEDs on the ARD2-INNOV8

First we are going to make a single traffic light program.
When a program is written to solve a problem or provide a solution, it is called an *algorithm*.
An *algorithm* is a set of instructions which a computer follows.
We have been using algorithms to solve problems in each of our lessons so far, and in this lesson you will have the chance to create your own algorithm for the double traffic light sketch.
For our single traffic light sketch we will create two special function blocks to contain our main algorithms.
The algorithms for this sketch are very simple and easy to understand.
These functions will control the light sequence for our single traffic light.
The pattern we need is this:



The traffic light starts on GREEN. There is a delay after the button is pressed.	The lights switch to YELLOW for a short time.	The lights switch to RED for an amount of time.	The lights switch back to GREEN.

Table 7. The single traffic light sequence

Creating the function blocks

We start by making the function block which will set our traffic light to GO (green).

Figure 109. Creating the *traffic_go* function block

The algorithm for our *traffic_go* function block is very simple; it only contains two blocks and has only two *local variables* as *paramter* inputs (**Figure 110**). This function block will set the GREEN LED to TRUE and the RED LED to FALSE.

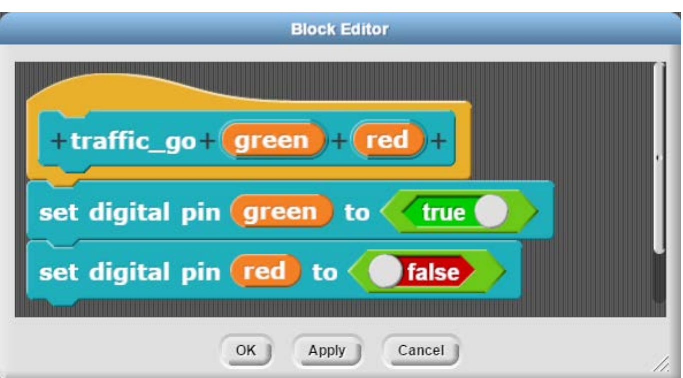


Figure 110. The *traffic_go* function block

Create your ***traffic_go*** function block by copying the blocks in **Figure 110**.

Next we need to create a function block to make the traffic light run through the STOP sequence.

Figure 111. Creating the *traffic_stop* function block

The ***traffic_stop*** function block contains 8 blocks and is responsible for making the traffic light run through the STOP sequence. It uses three local variables as inputs, because it needs to control all three of the lights: RED, YELLOW, GREEN.

The algorithm in the ***traffic_stop*** function block is still very simple. This function block also *calls* (uses) the traffic_go function block as its last block to set the traffic light back to GREEN.

The algorithm in this function block will wait 2 seconds, then switch the lights from GREEN to YELLOW, wait a further second, then switch the lights from YELLOW to RED, then wait a further 4 seconds before calling the ***traffic_go*** function, which switches the lights from RED to GREEN.

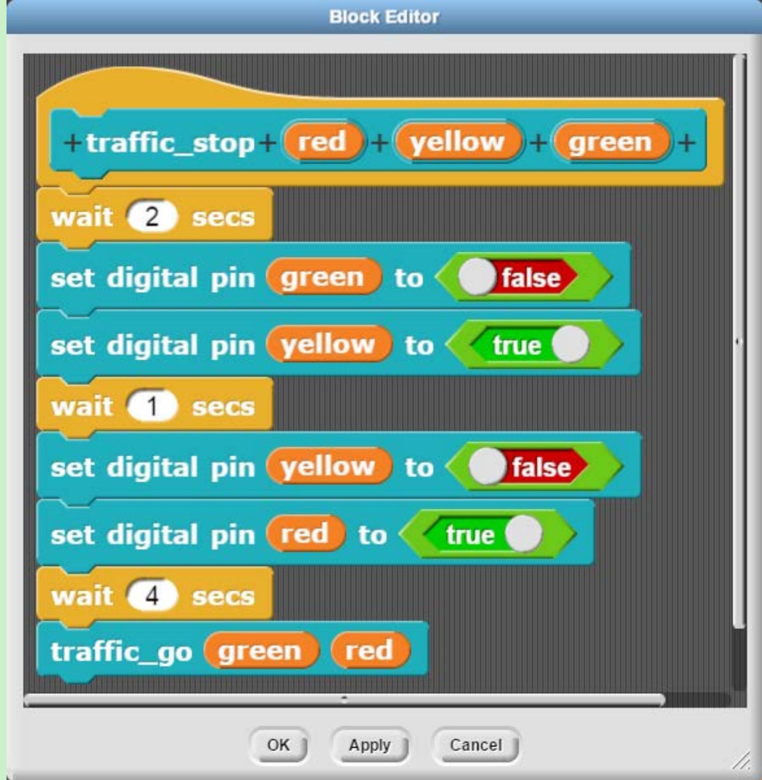


Figure 112. The *traffic_stop* function block

Create your ***traffic_stop*** function block by copying the blocks in **Figure 112**. After completing and saving these function blocks, you should be able to see them appear within the *Arduino* blocks library (**Figure 113**).

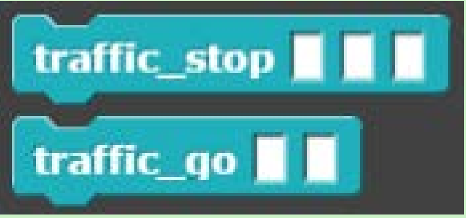


Figure 113. The *traffic_stop* and *traffic_go* function blocks in the Arduino blocks library.

Creating the traffic_light1 sketch

Now that we have created our function blocks, we can begin to build the blocks for the main part of our sketch.
So far we have used *local variables* within our function blocks, but we now need to create the *global variables* which will be used throughout our sketch and as *arguments* (inputs) to our function blocks.
We start by declaring the three global variables that will hold our LED pins for the three traffic light colours, and we also need a global variable for our button, and a further one to hold the state of the button.
Declare each of these variables as you have done for all of the previous lessons (**Figure 114**).

Figure 114. The global variables needed for the *traffic_light1* sketch

You will define these variables at the top of your sketch like this (**Figure 115**):

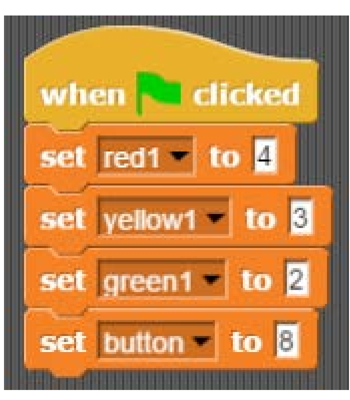


Figure 115. Defining the pin numbers for the global variables

The code in the main part of the sketch is quite simple: it first runs the ***traffic_go*** function to set the lights to GREEN, and then continually checks to see if the button has been pressed.
If the button has been pressed it runs the ***traffic_stop*** function to run the lights through the STOP sequence and then back to GO.
Build your *traffic_light1* sketch by copying the blocks in **Figure 116**.
Then connect your board to run and test the sketch.

Figure 116. The *traffic_light1* sketch

Test, Evaluate, Improve

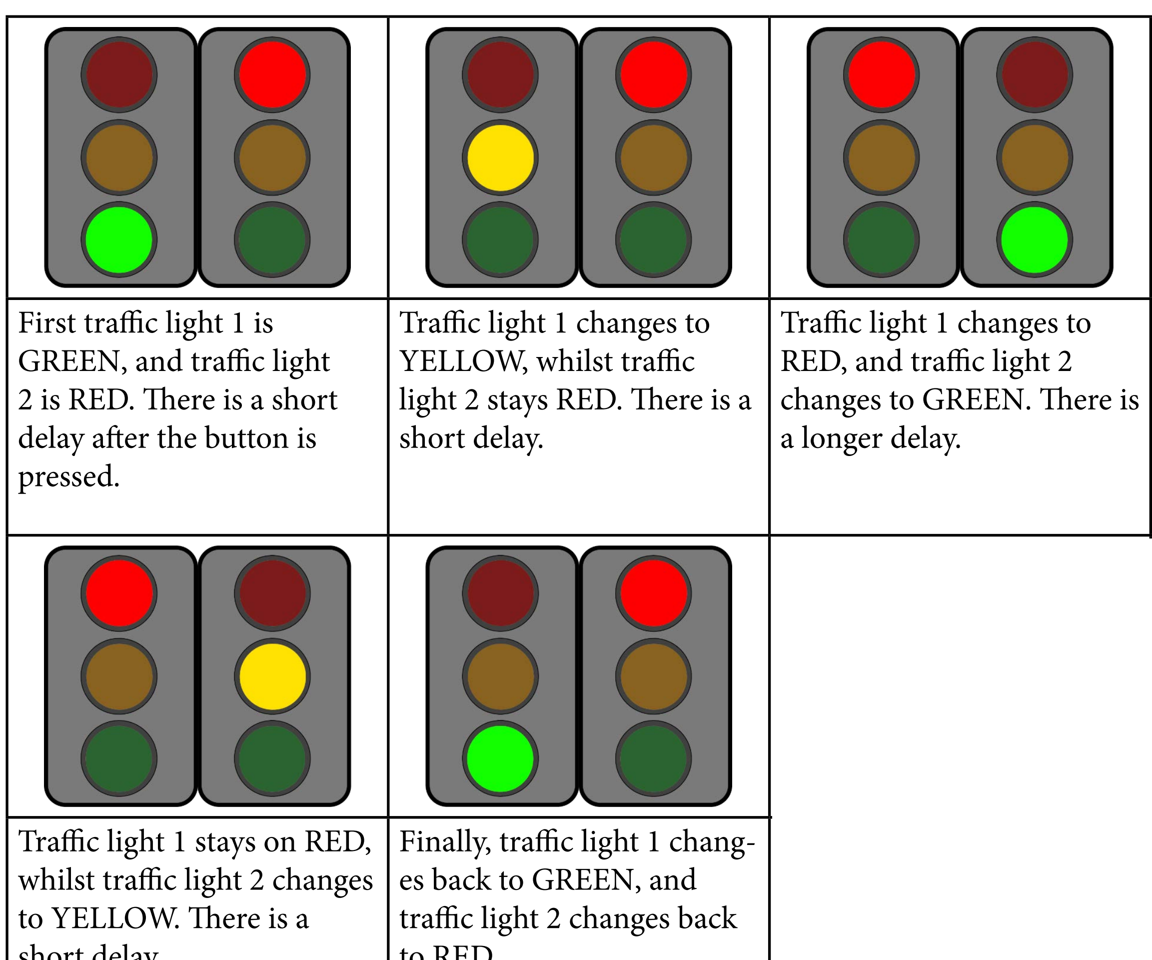
Did your traffic light perform as expected?
If so, well done! If not, inspect your code to see if you can debug the issue. You may have used the wrong pin number/s, or you may have missed a block or variable somewhere. Debug your sketch and then run it again. Remember, if you are having trouble understanding this sketch, you can watch the video tutorial for this lesson on the Firebugs Youtube channel.
You may need to make improvements to this sketch to get it running to your preferences, such as adjusting the timings inside the ***traffic_stop*** function block.

Now try this

The single traffic light was the easiest of the traffic light sketches. The next sketch *traffic_light2* involves using 6 LEDS to run 2 traffic lights through a sequence. Here we will give you an opportunity to work this sketch out on your own, but we will give you some hints, and there is a solution in the back of this book, as well as a video tutorial on the Firebugs Youtube channel in case you get stuck.

Creating the double traffic light

The sequence for the double traffic light is given in **Table 8**, however, you should use your programming pattern recognition skills to try to work the double traffic light sequence out yourself before looking at this solution.
The sequence that you will need to follow for your algorithms in this sketch is:



First traffic light 1 is GREEN, and traffic light 2 is RED. There is a short delay after the button is pressed.	Traffic light 1 changes to YELLOW, whilst traffic light 2 stays RED. There is a short delay.	Traffic light 1 changes to RED, and traffic light 2 changes to GREEN. There is a longer delay.
Traffic light 1 stays on RED, whilst traffic light 2 changes to YELLOW. There is a short delay.	Finally, traffic light 1 changes back to GREEN, and traffic light 2 changes back to RED.	

Table 8. The sequence for the double traffic light

Instead of starting from scratch with this sketch, you can choose *Save As* from the File menu to save your *traffic_light1* sketch as *traffic_light2*. This way you can simply adjust the part of code that needs to be changed.

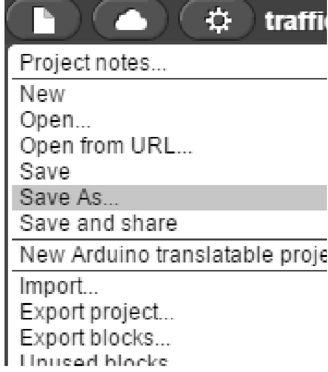


Figure 117. Saving *traffic_light1* as *traffic_light2*

To build the double traffic light sketch you will need to once again create two function blocks: *traffic_go* and traffic_stop; but this time the algorithms inside these functions will be more complicated.
You will need to declare the following global variables and define them at the top of your code:

Figure 118. The global variables needed for the *traffic_light2* sketch

A small hint for you

The tops of the function blocks in the solution are set up like this:

Figure 119. The top block inside the *traffic_go* function block

Figure 120. The top block inside the *traffic_stop* function block

Now test your programming skills

Set up your function blocks for the double traffic light sketch ***traffic_light2*** to see if you can figure out the algorithms for this sketch. Remember that there is a solution at the end of this book, and there is also a video tutorial on the Firebugs Youtube channel to help you if you get stuck.
Share you solution with others, and discuss and compare how you overcame programming challenges. You can also share your solution for this challenge on the Firebugs & Wiltronics websites or Facebook pages.
Happy prototyping!

Content Index

Lesson 10 – Using servo motors

Key words: square wave, PWM, Cartesian plane, horizontal/vertical axis
Key focus: using a servo motor, using the map function

Difficulty Level

So far in our lessons in this book we have used the LEDs and buzzer on the ARD2-INNOV8 as output devices in our sketches. In this lesson we are going to learn how to use a special motor called a *servo motor* as an output device, which will allow us to add movement to our systems.
Servo motors have their own inbuilt sensor which feeds back the position of the servo's arm, and allows it to keep very precise positions.
Servo motors are the motors which are used to control the movement of wing flaps on remote controlled aeroplanes, the ailerons of RC helicopter blades, and the steering movement on RC cars. They are also used in many other systems, and larger servo motors are used for the movement of some large industrial robots.
To make a servo motor move using a digital controller like our Arduino micro-controller, you need to send it a signal in the form of a *square wave*. The servo will move to a position between 0° and 180° depending of the width of the pulse in this square wave. **Figure 121** shows how this works. If the pulse width is 1 millisecond the servo will move to 0°, if the pulse width is 1.5 milliseconds the servo will move to 90°, and if the pulse width is 2 milliseconds the servo will move to 180°.

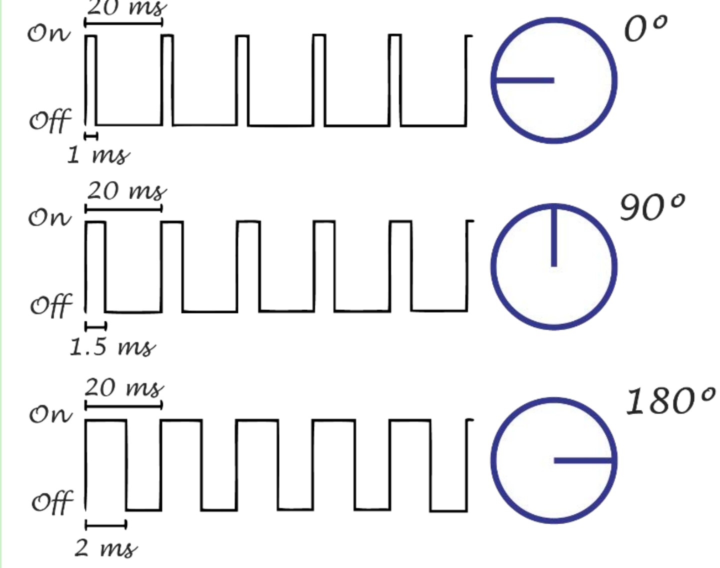
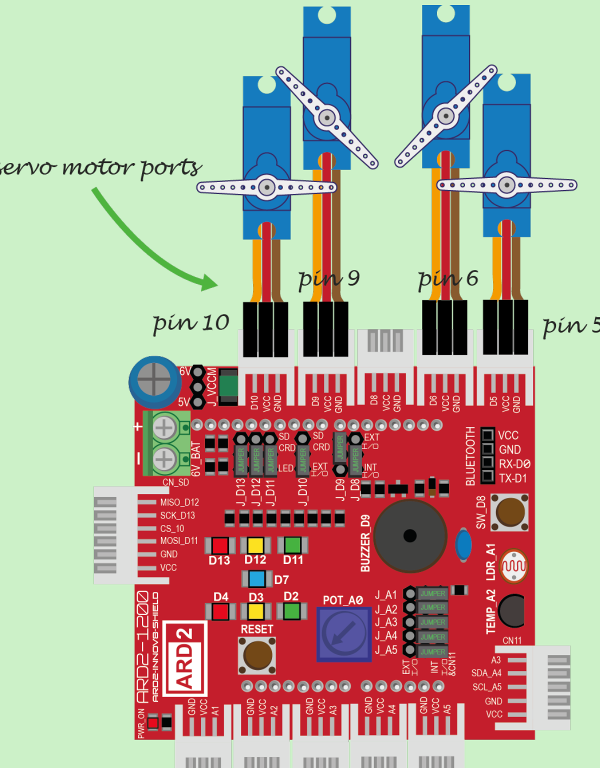


Figure 121. The pulse width sets the position of the servo motor

You do not need to understand how a servo works or remember the width of the pulse signal, however, you do need to know that there are only 6 pins on the Arduino micro-controller which we can use to control our servo motor, these are digital pins *3, 5, 6, 9, 10,* and *11* (on the ARD2-INNOV8 pins *5, 6, 9, &10*). These special pins are known as *PWM* pins, which stands for *pulse width modulation*. On the Arduino board these pins are marked with the *tilde* symbol **~**. On the ARD2-INNOV8 shield you can attach up to 4 servo motors using the linker connectors on the digital side of the board (**Figure 122**).



If you want to run 4 servo motors at the same time, you can do this on the ARD2-INNOV8 shield, however, in this case you should power the servos via the +6V power terminal (see the video for how to do this on the Firebugs Youtube channel).

Figure 122. The ARD2-INNOV8 can run up to 4 servo motors

Controling the position of a servo motor with a potentiometer

For this lesson you will need one small servo motor attached to your ARD2-INNOV8

Connect the three wire socket of your servo motor to the linker connector at digital **pin 5** on your ARD2-INNOV8 shield as has been done in **Figure 123**, making sure that the brown wire is connected to the *GND* pin, and the yellow wire is connected to the *D5* pin.

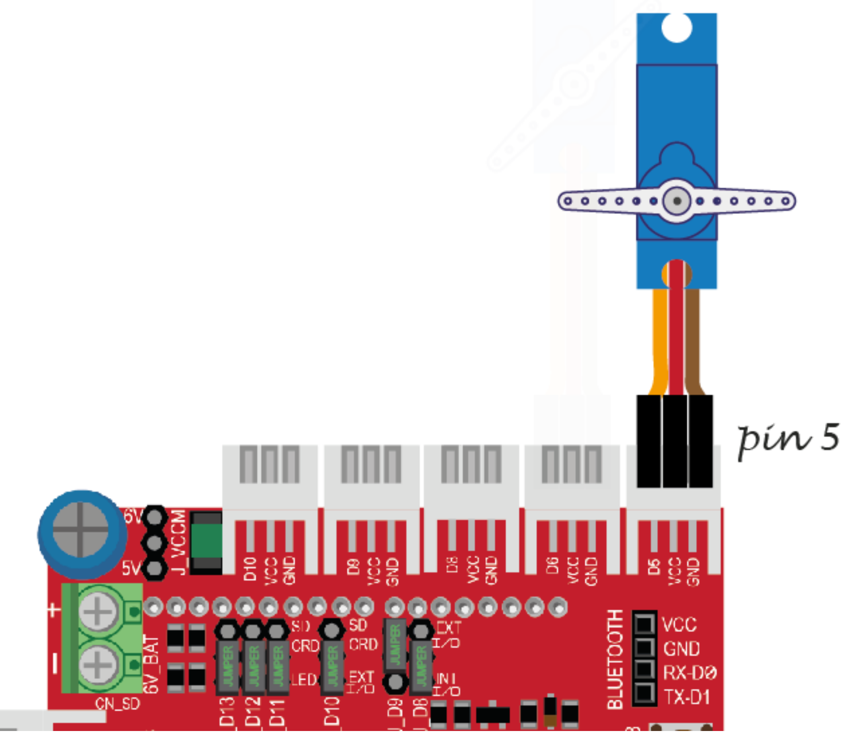


Figure 123. Servo connection for the *pot_servo* sketch

In this lesson we will only be controlling the position of a single servo, however, you can easily extend this lesson and the final project in this book to run more than one servo motor.
Remember back to **Lesson 6** where we used the potentiometer on the ARD2-INNOV8 to change the pitch of the sounds made by our system; this time we are going to use this potentiometer to control the position of our servo motor.
For this sketch we will once again need to use the ***map from_to_*** block to convert our potentiometer data to data which the servo can read.

Creating the pot_servo sketch

This is a very simple sketch compared to the previous few lessons, however, it is an important lesson which will help you to create interesting prototyping ideas. We first need to declare 3 global variables, and define two of them at the top of our sketch: One for our servo motor, one for our potentiometer, and one to hold the data coming from the potentiometer.

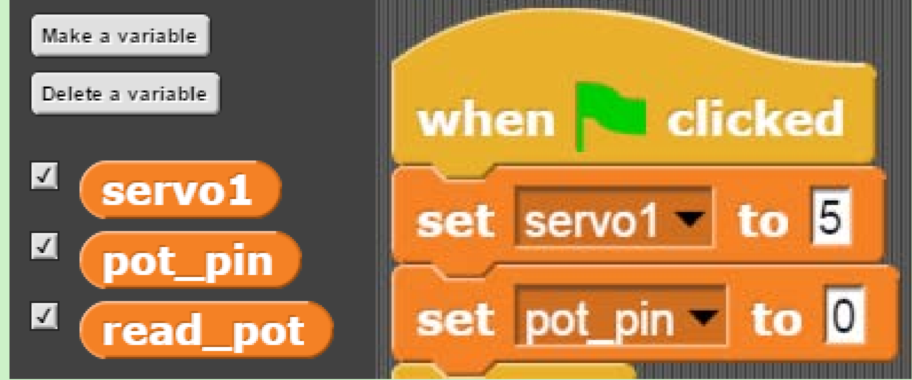


Figure 124. Global variables needed for the *pot_servo* sketch

We construct our code blocks so that our *read_pot* variable is reading and storing the data coming in from the potentiometer on pin 0. We then map this value from a value between 0 and 1023, to a value between 0 and 180. The 0 to 180 is a measurement in degrees. These values are converted by the Arduino micro-controller to a signal pulse which the servo can read.

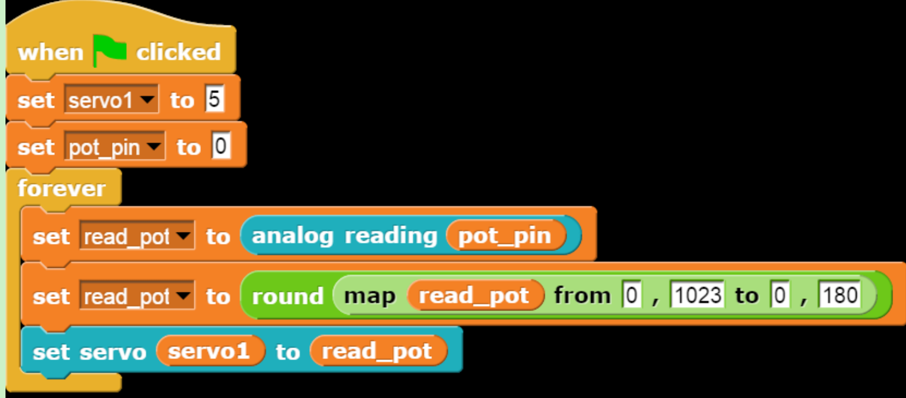


Figure 125. Code blocks in the *pot_servo* sketch

Build this sketch by copying the blocks in **Figure 125**. You may need to reload the ***map*** block from your ARD2-INNOV8-SNAP folder (see **Lesson 6**).
Connect your board ready to run, but before you do, see if you can predict what this sketch is going to do.
Run your code and turn the knob on the potentiometer to see what happens.

Controlling a servo motor with a mouse

Did you predict that when you turned the knob on the potentiometer that the arm on the servo would also turn in the same direction? This type of system allows you to control movement from a distance, and it is exactly the same principle as a robotic arm or giant crane.

In this sketch we are going to use this same principle, but we are now going to use our computer mouse as the control. Snap4Arduino has a code blocks called ***mouse x*** and ***mouse y***. These blocks can be found within the Sensing blocks library. The *X* and *Y* refers to axis on the *Cartesian plane*. The *X* being the *horizontal axis* and the *Y* being the *vertical axis*. We can use these blocks in our sketch to control the movement of our servo motor.
In this sketch we will use the ***mouse x*** block to control the movement of a single servo motor, however, after this is done, you will be given the challenge to add a second servo motor and control its movement using the ***mouse y*** block.

The code in this sketch is much the same as the code in the previous sketch, but this time we are going to use 3 global variables: *servo1* – for our servo motor, *read_mouseX* – to hold the data coming from the mouse position, and *convert_mouseX* – to hold the converted data which has been mapped using our ***map from_to_*** block (**Figure 126**).

Figure 126. Global variables needed for the *mouse_servo* sketch

We only need to define one of these variables at the top of our code, and the rest of the sketch is very much the same as our *pot_servo* sketch.

```
when [flag] clicked
set servo1 to 5
set servo servo1 to 0
forever
  set read_mouseX to mouse x
  set convert_mouseX to map read_mouseX from -1200 , 500 to 0 , 180
  set servo servo1 to convert_mouseX
```

Figure 127. Code blocks in the *mouse_servo* sketch

Build and test the sketch

Build this sketch by copying the blocks in **Figure127**, and then connect your board and run the sketch.
Move your mouse across your computer screen and you should see the servo motor following the direction of your mouse.
You may need to adjust the parameters in the first half of the ***map*** block to suit your screen; this is why we have used the *read_mouseX* variable so that you can see the value of the data coming from the mouse position, and the *convert_mouseX* variable so that you can see what position the servo should be.
You can view the values of these variables within the *Stage* area, on the the top right of the Snap4Arduino IDE (**Figure 128**).

Figure 128. The *Stage* area lets you view the data coming into your variables

Now try this

In the *mouse_servo* sketch we used only one servo controlled by our mouse on the *X axis*, however, we can move our mouse along both the *X* and *Y* axes, which means that we are able to control 2 servo motors (one for each axis).
Adjust this sketch so that you can control two servo motors: one with the ***mouseX*** block, and one with the ***mouseY*** block (you can use digital pin 6 to connect your second servo motor).
Once you have built this sketch, think up an application that uses two servo motors and build yourself a *prototype* to test. This could be a robotic arm, crane, or even a small CNC machine.
You can use cardboard, wood, plastic, glue and tape to build your prototype. The next lesson is an extended project which shows you how to combine inputs and outputs from the ARD2-INNOV8 to build a simple prototype.

Final Challenge – Putting it all together

Key words: prototype, components, anode/cathode, soldering
Key focus: designing, prototyping, constructing, testing, evaluating

Difficulty Level

In most of the lessons in this book we have built projects which used the on-board components on the ARD2-INNOV8 shield. In this project we are going to build a working prototype which gives you experience putting components together to connect and use with your ARD2-INNOV8.
So far we have built projects that output light, sound, and movement; so in this project we are going to design and build a prototype which combines those three types of outputs.
If you think about systems that output light, sound, and movement, there are many possibilities that we could use for our design, but one of the easiest and most fun to build is *railway crossing gates*. Railway crossing gates output light in the form of the red flashing lights, sound in the form of the dinging bell sounds, and movement in the form of the boom arm moving down and up; so this is a perfect system for our project.

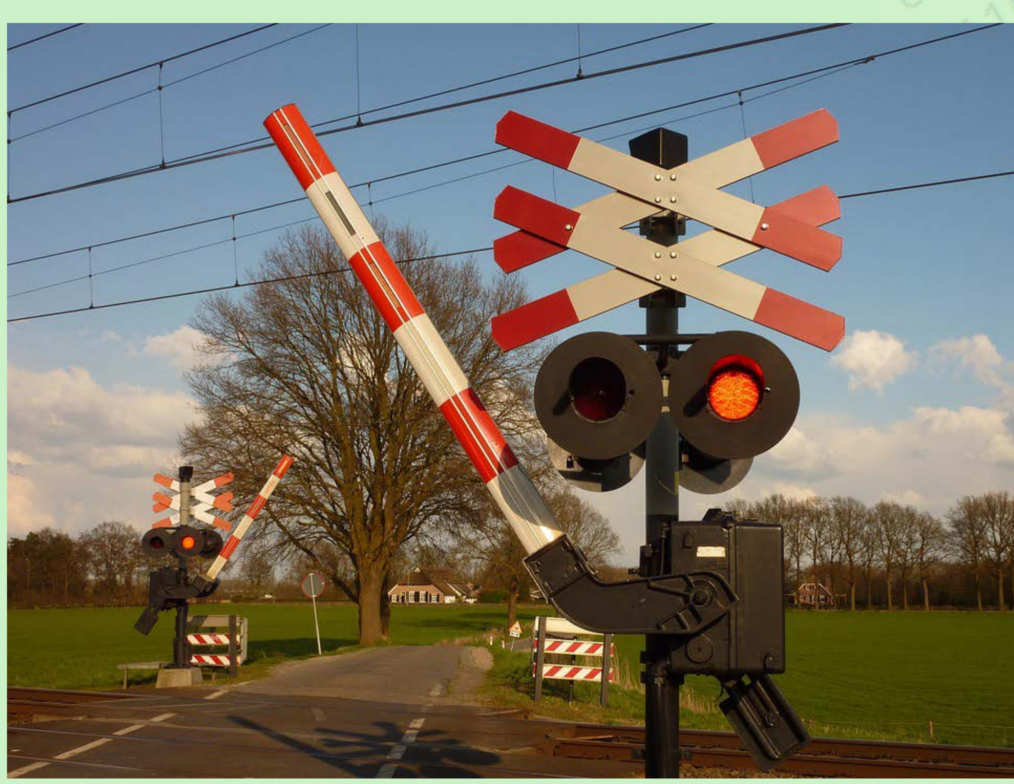


Figure 129. Railway crossing signal gates

Building the prototype

Materials

For this project, you will need:

- Some cardboard or thin plywood
- A 1 cm by 1 cm piece of wood 30 cm long or a rolled up piece of card or paper 30 cm long
- Black paint
- Scissors and glue (PVA wood glue and Superglue)
- Two red 5mm LEDs
- Two 270Ω resistors (anything between 220 and 330 will do)
- Four long breadboard jumper wires with a socket end and one end stripped (2 Red and 2 Black)
- Some solder and a soldering iron (this part will need to be done by an adult)
- Some electrical tape or heat-shrink tube
- A mini or micro servo motor with its arm attachment
- Small blocks of wood or cardboard to use as a base and spacer for the servo
- Resources from the **ARD2-INNOV8-SNAP** folder

Creating the prototype parts

In your ARD2-INNOV8-SNAP folder you should find an A4 template sheet called *TrainCrossing.pdf*, which shows the layout of all of the bits that you need for this project. You can use any materials you wish to build your prototype, however, cardboard or wood may work best and be easier to build with.
Cut out all of the pieces you need to build the prototype according to the instructions on the template sheet in your ARD2-INNOV8-SNAP folder, then glue these pieces together and leave them to dry ready for painting. You will find a video tutorial on how to put this prototype together on the Firebugs Youtube channel.
After painting all of your pieces attach the labels and artwork with glue to the appropriate parts of your model (see the instruction sheet for this).
Finally attach the signal pole to the base with glue, then fix the servo in position with a small amount of Superglue and also glue the boom to the servo arm (you may need to adjust the position of this later, so make sure that the servo arm is not screwed onto the servo).

Attaching the LEDs

*****Important safety information*** This part will need to be done by an adult or a person who has had experience using a soldering iron.**

Study your LEDs and observe which legs are the Anode (positive) and which legs are the Cathode (negative). You will need to know this when you attach the jumper wires to these legs. The Anode (positive) leg is usually longer than the Cathode (negative) leg.
Ask an adult to solder the Anode (positive) leg of each of your LEDs to one of the resistor legs, and then solder the Cathode (negative) leg of the LED to the stripped end of one of the black jumper wires. Solder the other leg of the resistor to the stripped end of one of the red jumper wires (use crocodile clips as an alternative).
Repeat these steps for the 2nd LED.The finished result should look like this:

Figure 130. The soldered LEDs with a resistor on one leg

Once you have done this, insulate your solder joins with electrical tape or heat-shrink. These LEDs should fit neatly into the circular disks which you should have glued either side of your signal pole (see **Figure 132**).

The jumper wires will be plugged in to the linker connectors on the digital side of your ARD2-INNOV8 shield.

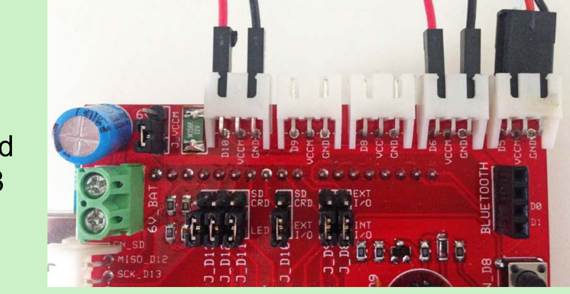


Figure 131. Attaching the LED and Servo wires to the ARD2- INNOV8

Connecting your model to your ARD2-INNOV8

Connecting your model to your ARD2-INNOV8 shield is easy.
Plug the three wire socket from your servo motor into the linker connecter at digital pin 5, just like you did in lesson 10.
Next plug the red jumper wire from one of LEDs into the ***D6 pin***, and the black jumper wire from this LED to the ***GND*** pin which is near the D6 pin.
Repeat these steps with the second LED, but plug it into the ***D10 pin*** (red) and ***GND*** pin (black).
If you have connected these wires correctly then your ARD2-INNOV8 should look the same as the image in **Figure 131**. If your setup looks good, then you are all set to start programming your prototype.

Figure 132. The setup for the *Train_crossing sketch*

Programming the train crossing prototype

You should now be at a stage with your programming where you understand the process and can begin to do things independently; therefore there is no need to fully explain each part of the process for how to build the code blocks for your train crossing system.

For this sketch you will need to declare the following global variables (**Figure 133**). There is a variable for each of the two LEDs, one for the servo, one to hold the position for the servo, one for the button, one to hold the state of the button, and we are once again using our *blinkMore* variable to hold the amount of delay that we want to feed into out ***wait*** block:

Figure 133. Variables in the *Train_crossing* sketch

We have also created two function blocks for this sketch: One to contain the algorithm for when the boom gate comes down, and the other to contain the algorithm for when the boom gate goes up.

Figure 134. The two created function blocks for the *Train_crossing* sketch

We have designed this system to start as soon as the button is pressed, however, you could easily adapt this system to be initiated when a sensor is activated. An *ultrasonic* sensor module would work very well in this case. Look out for extensions to this project on the Firebugs Youtube channel.

The sequence that we want our system to follow is this:

When the button is pressed:
Lower the boom gate, flash the LEDs, and sound the bell until the gate is down.

Then:
Keep the boom gate in the down position and continue to flash the LEDs and sound the bell for a specific length of time.

Then:
Raise the boom gate and flash the LEDs until the gate is fully up (we don’t need the bell sound here).

Because we have no sensors that will tell us if the boom gate is up or down, we need to use another method to ensure that our system knows when to bring the gate back up. In this case we can use *repeat* blocks to move the servo motor a small amount each time for a certain number of *loops*.

The two images on the following page show how to set up your blocks for the ***gates_closed*** and ***gates_open*** functions (**Figure 136 & Figure 137**).
Create these functions and copy these blocks exactly using the process that was shown to you in **Lesson 9**.

You will need to import the bell sound to use in the *gates_closed* funtion.
To import this sound into Snap4Arduino, simply drag it into the sounds area.

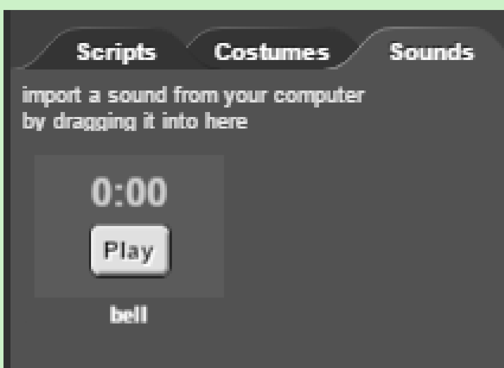


Figure 135. Import the *bell* sound by draging it into the *Sounds* area

```
+gates_closed+ pin1 + pin2 + servo + buzzPin + pause +
set position to 0
repeat 10
    set servo servo to position
    set digital pin pin1 to true
    set digital pin pin2 to false
    play sound bell
    wait pause secs
    set digital pin pin1 to false
    set digital pin pin2 to true
    play sound bell
    wait pause secs
    set position to position + 6
repeat 5
    set digital pin pin1 to true
    set digital pin pin2 to false
    play sound bell
    wait pause secs
    set digital pin pin1 to false
    set digital pin pin2 to true
    play sound bell
    wait pause secs
```

Figure 136. The *gates_closed* function block for the *Train_crossing* sketch

```
+gates_open+ pin1 + pin2 + servo + pause +
set position to 54
repeat 10
    set digital pin pin1 to true
    set digital pin pin2 to false
    set servo servo to position
    wait pause secs
    set digital pin pin1 to false
    set digital pin pin2 to true
    wait pause secs
    set position to position − 6
    set digital pin pin2 to false
set digital pin pin2 to false
```

Figure 137. The *gates_open* function block for the *Train_crossing* sketch

Now that you have created your two function blocks, you should be able to see them within the *Arduino* blocks library. Used these two blocks to build the main part of your *Train_crossing* sketch by copying the blocks in **Figure 138**. Then connect your board, run the sketch, and press the button to observe what happens.

Content Index

```
when [flag] clicked
set light1 to 6
set light2 to 10
set blinkMore to 0.5
set servo to 5
set button to 8
forever
    set buttonState to digital reading button
    if buttonState = false
        gates_closed light1 light2 servo blinkMore
    gates_open light1 light2 servo blinkMore
```

Figure 138. Code blocks within the *Train_crossing* sketch

Now try this

Did your *Train_crossing* sketch perform as expected?
If not, were you able to identify the bug, and then debug the code and system to fix the issue?
If you were not successful with this project, remember that there is a tutorial for this lesson available on the Firebugs Youtube channel.

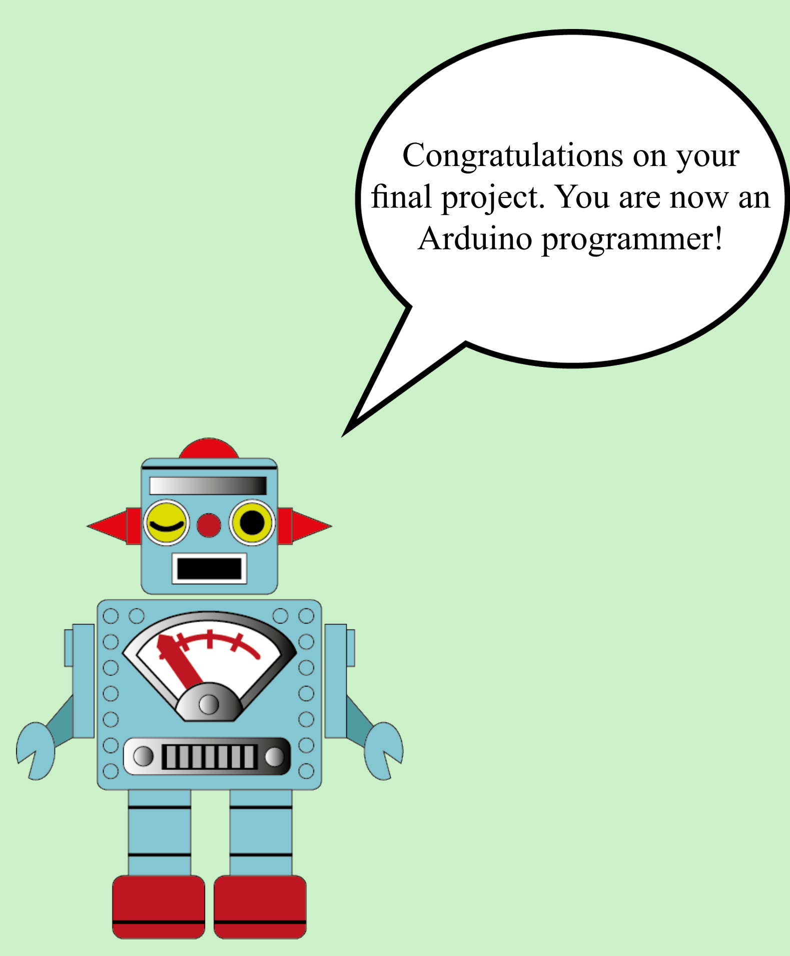
If you were successful with this project, well done! You are well on your way to becoming an experienced Arduino programmer.

Things that you should try:

- Examine and evaluate your system and compare it to how you think the system should function. If you feel that there can be improvement made, adjust the code and system parts to match your solution. Discuss your ideas with your class mates and justify why you have made your decisions.
- Use an alternate trigger device for this system other than the button. You could make it activate by light, temperature, by touching a banana, or if you are feeling adventurous, you could try using an ultrasonic sensor to trigger this system. Solutions and ideas for this will be available on the Firebugs Youtube channel.

Share your solutions and ideas with other people by posting your examples on the Firebugs and Wiltronics websites and Facebook pages.

Best of luck with your programming, and happy prototyping!



Double traffic light solution

```
+traffic_go+ red1 + red2 + green1 + green2 +
set digital pin green1 to true
set digital pin green2 to false
set digital pin red2 to true
set digital pin red1 to false
```

Figure 139. The solution for the *traffic_go* function block

```
+traffic_stop+ red1 + red2 + yellow1 + yellow2 + green1 + green2 +
wait 2 secs
set digital pin green1 to false
set digital pin yellow1 to true
wait 2 secs
set digital pin yellow1 to false
set digital pin red1 to true
set digital pin red2 to false
set digital pin green2 to true
wait 4 secs
set digital pin yellow2 to true
set digital pin green2 to false
wait 2 secs
set digital pin yellow2 to false
traffic_go red1 red2 green1 green2
```

Figure 140. The solution for the *traffic_stop* function block

```
when clicked
set red1 to 4
set yellow1 to 3
set green1 to 2
set button to 8
set red2 to 13
set yellow2 to 12
set green2 to 11
traffic_go red1 red2 green1 green2
forever
  set button_state to digital reading button
  if button_state = false
    traffic_stop red1 red2 yellow1 yellow2 green1 green2
```

Figure 141. The Solution for the *traffic_light2* sketch

Glossary of terms

Abstraction – a process in which the details of code structures are hidden away, leaving only the essential information to be visible in the main program.

Algorithm – a series of steps, processes, operations, or rules which is followed to solve a problem.

Analog – data that is represented as continuously variable in value.

Array – an arrangement of data in a particular structure. A 2D array is represented in rows and columns of data elements.

Axis horizontal/vertical – reference lines which run at right angles to each other – can be left/right/up/down (x axis and y axis).

Boolean – a variable which can have one of two possible values – ON/OFF, HIGH/LOW, 0/1, TRUE/FALSE, YES/NO, etc.

Cartesian plane – a plane made up of an x axis and a y axis.

Closed-loop system – a system which has an automated control and therefore does not require manual control.

Component (electronic) – a device or part which performs a specific function – such as an LED which emits light.

Control structure – Control structures in programming are blocks of code which determine the way that the program will flow; they are basically the decision making parts of the program. An *IF/ELSE* statement block is one example.

Data structure – a system of storing and organising data. In Arduino programming this can be the variable with its types, an array, a function, class, library, file, table, etc.

Debugging – the process of identifying and removing errors in the code and/or circuit. The origin of this word goes back to the days when computers used vacuum tubes and real bugs would sometimes cause issues.

Declare (variable) – to create and identify a variable ready for use – provide its name (usually its type).

Define (variable) – to provide the details about the variable – what its value is: e.g. *set LED1 to 10*.

Degrees Celsius/Fahrenheit – units for the measurement of temperature – Fahrenheit is used mainly in USA.

Digital – having discrete or finite sets of values – 1 to 10, or 0 and 1.

Duration – an amount of time in which something continues.

Element – an essential part of something – the elements in a list of numbers are the numbers themselves.

Encapsulation – the process of structuring blocks of code within functions, classes, libraries, and other containers to restrict access and protect them.

Execute (program) – to initiate, run, or start a program to run through its instructions.

Firmware – preloaded software which provides control and monitoring of a system. This type of software is not accessed or changed by the user.

Frequency – the rate at which something occurs over a given length of time – can be expressed as cycles per second etc.

Input – data used within a system. In Arduino programming physical inputs are converted to digital electrical signals which are then processed by the program code.

Iteration – the repetition of a process. Often used for tasks which involve a large number of repetitions.

LED - anode/cathode – light emitting diode. An electronic device which emits light when a currcnt is applied. The anode is the positive leg and the cathode is the negative leg. Current can only flow one way through a LED.

List – a type of data structure which holds data elements in a single row and more than one column. In text-based computer code lists (arrays) are counted from zero, however, in visual-block code list are counted from 1.

Loop – a control structure that continually repeats itself forever until it is switch off by another control structure.

Matrix – a data structure which contains a collection of data into a fixed number of rows and columns.

Nesting – when control structures are placed inside other control structures to provide further branching options for the program to flow to.

Open-loop system – a system that does not have automated control and must therefore rely on the user to be the controller.

Operator – a character (or code block) that represents a particular action – examples are: TRUE/FALSE, greater than >, less than <, etc.

Output – the result from a system. Outputs can be physical or data, and can be expected or unexpected, desired or non-desired.

Parameters & Arguments (functions) – parameters are *local variables* which define how expected inputs will be used in a function. Arguments are the global variables which are fed into functions providing values to be used within the function.

Pitch (sound) – the rate of vibrations which determines the highness or lowness of a sound.

Potentiometer – an electronic device used to vary values along an analog scale. Provides a varying range of voltages via the change of resistance values.

Process – a series of steps taken in a particular order to achieve a desired result.

Prototype – an initial limited working model which is used to demonstrate that something will work (proof of concept).

Pseudo-code – written words which resemble programming language and provide the reader with a clear idea of how the actual program will flow.

Pullup-resistor – a resister connected in a simple circuit with a switch device which causes the input pin on a micro-controller to read HIGH when the switch is not activated.

PWM – pulse width modulation – a system which is used to produce a pulse of a particular width in order to drive a device in a way which simulates an analog output, such as the dimming of an LED or the changing speed of a motor.

Random – selected by chance. The generation of random sets of numbers are very useful in computer programming.

Resistance – the level of difficulty by which electricity passes through an object. A resistor slows down the flow of electric current.

Sequence – the set of order in which things flow after one another.

Sketch (programming) – The word sketch is usually only used in relation to Arduino programming. It refers to the code and the setup of the system being tested.

Soldering – the joining together of electronic components using metal solder which has been melted by a soldering iron.

Square wave – a wave which moves (oscillates) abruptly between values. In Arduino programming a square wave might oscillate between 0 volts and 5 volts for example.

Synchronisation – an operation where two or more things happen at precise timings.

System – something that has at least one input, one process, and one output.

text-based code – programming language which uses written text.

Threshold (programming) – a point, degree, or magnitude which when exceeded causes a reaction.

variables local/global – a variable is a simple data structure (container for data). A local variable can only be used within its function block. A global variable can be used by all parts of the program.

Visual-block code – programming language which is constructed by dragging visual ‘code blocks’ together. *Scratch* by MIT is a very popular visual-block programming platform, Snap4Arduino is another.

Coding INNOV8ion, it's a Snap! with the ARD2-INNOV8

and Snap4Arduino

1st edition

Introduces the language and concepts of computer programming with easy to use visual – block coding, using the favourite hardware platform loved by engineers and backyard inventors worldwide.

Authors

Mark Trezise & Seven Vinton

Seven Vinton

Seven has shared his knowledge of ICT and digital systems technology with students and teachers via State Conferences over his 18 year teaching career. He has inspired many students to take up careers in technology areas and supports world-wide sharing of knowledge through his Youtube channel ICT Tools for Teachers. Currently the leader of Curriculum at Oberon High School, Seven has provided support and assistance to help schools and teachers throughout the world implement digital learning into their curriculum.

Mark Trezise

At the time of writing this book Mark was studying year 11 at Oberon High School. Mark got into programming via a Code Club at his school, and from there his passion grew and along with his knowledge. Mark has taught digital programming to students around Victoria and interstate. He attended the Questacon Invention Convention in 2016, and was asked back as an ambassador for the 2017 convention. He has been programming for 3 years and has implemented several inventions to make work easier on his farm. Mark has established a large following of learners which he supports via his websites and Youtube channel Trez Tech.

FireBugS

Firebugs Educational Resources