UK Electronics Skills Foundation

Insight into Electronics

Aston University
Birmingham

© UKESF 2023

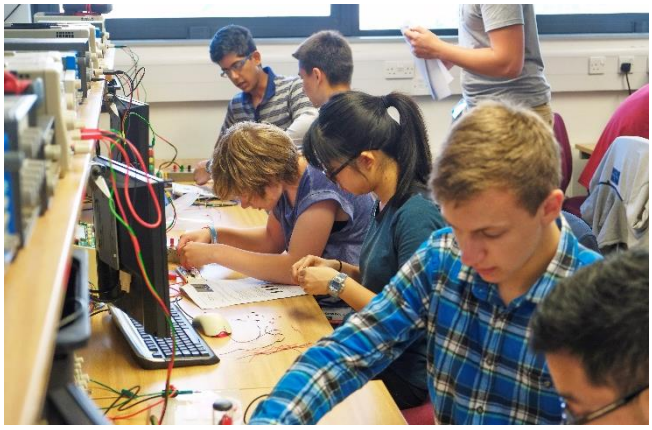
Preface

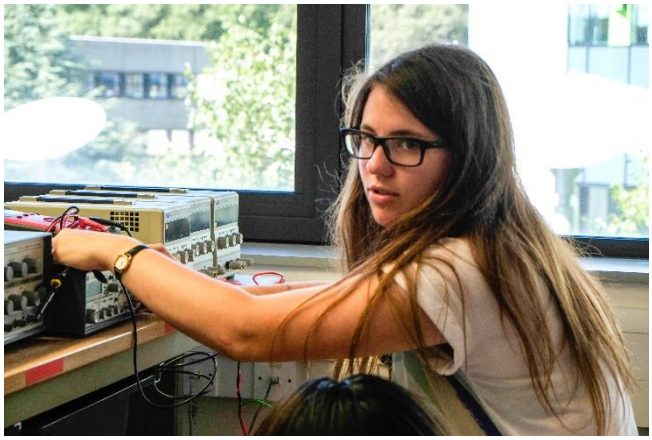
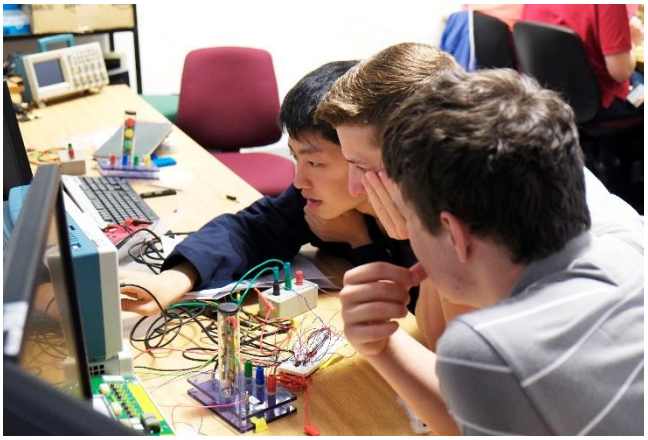
Welcome to the UK Electronics Skills Foundation (UKESF); this Guide supports the practical activities to provide an 'Insight Into Electronics'.

Technology is at the heart of our world; it is the future. Recent advances in communication, transportation, healthcare provision and entertainment have already transformed our society, but there are continued challenges too, of course. Technology offers hope and expectation, and Electronics is a vital part. Our devices and tech products would not be able to work without Electronics. And it is down to Electronic Engineers to develop the microprocessors, design the circuits and write the embedded software code.

Did you know that the UK Electronics industry is one of the largest in the world? You can be part of this industry, developing the next generation of products and helping produce the technological solutions needed by society. There are different routes into careers in Electronics; you can study at university, undertake an apprenticeship, or complete a vocational college course.

This self-paced and interactive course will provide sixth formers with an introduction to microcontrollers, Electronics and programming. The activities can be completed flexibly to fit around other commitments and study at a pace that suits individual circumstances. The course is aimed at those students who are thinking about studying Engineering, Electronics or Computer Science at university, or college. However, no prior knowledge is assumed and support is available to complete the activities. The course is a collaboration between the UKESF and Aston University.



© UKESF 2023

About the UKESF

The purpose of the UKESF is to tackle the skills shortage in the Electronics sector in a coherent way. Our aim is to:

"Encourage more young people to study Electronics and to pursue engineering careers in the sector."

To achieve the aim, we have four strategic priorities:

- Ensure more schoolchildren are **aware** of Electronics. Show these children, their parents and teachers that there are exciting and worthwhile careers available as designers and engineers in the Electronics sector.
- With our partners, provide opportunities for them to develop their **interest** in Electronics and engineering, through to university study and/or apprenticeship.
- At university, ensure that undergraduates are encouraged to pursue careers in the Electronics sector and they are supported in their professional **development** so when they graduate, they are equipped with work-ready skills and experience.
- After graduation from university, we will help create a community of Electronics engineers to secure the future pipeline. We will **build relationships** and act as the representative voice for the sector on skills.

We are an independent charitable foundation, established in 2010, at the nexus of an extensive network of partners and collaborators, including 26 universities and 75 companies. On behalf of the electronics sector, we will build relationships, provide thought leadership and act as the representative voice on skills related matters.

Our undergraduate Scholarship Scheme is recognised as being 'best in class'; we have supported over 650 students since it started in 2010.

Registered charity number: SC043940
www.ukesf.org

About Aston University

Founded in 1895 and a University since 1966, Aston is a long-established university led by its three main beneficiaries – students, business and the professions, and our region and society. Aston University is located in Birmingham and at the heart of a vibrant city, and the campus houses all the university's academic, social and accommodation facilities for our students. Professor Alec Cameron is the Vice-Chancellor & Chief Executive.

Aston University was named University of the Year 2020 by *The Guardian*, and Electrical and Electronic Engineering was ranked 3rd overall in the UK (*Guardian University Guide*, 2021). The University also has TEF Gold status in the Teaching Excellence Framework.

The Electrical and Electronic Engineering programmes focus on project-based, hands-on, practical learning to ensure students develop the skills sought after by employers. A brand new suite of laboratories was opened in 2019 to ensure students learn in a high-spec environment with new equipment. Students are encouraged to design their own projects throughout the course with a focus on innovation, design engineering and product development.

© UKESF 2023

Contents

© UKESF 2023

© UKESF 2023

Introduction – What is Arduino?

The Arduino platform is a microcontroller development system that is aimed at people who want to use programmable-electronic hardware without needing to delve deeply into how a microcontroller operates. This platform is useful as an introduction to programmable systems for young engineers who are yet to learn more about what happens inside microcontrollers.

Arduino Uno

Figure 1 shows the hardware for an Arduino Uno. The microcontroller (a Microchip ATmega328P) is the large integrated circuit (IC) in the lower-left section of the board. The Arduino includes all the software and hardware needed to be powered and programmed through USB, and all its inputs and outputs are brought out on connectors (analogue and digital pins) running on along the side edges.

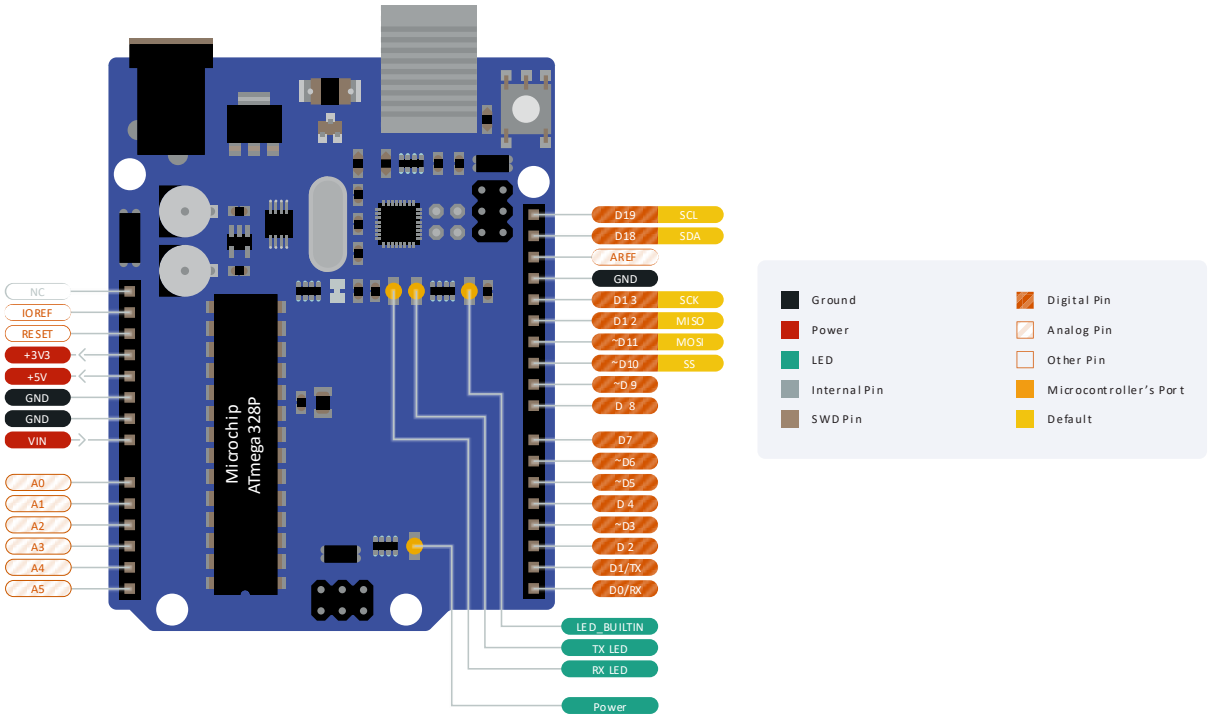


Figure 1. An Arduino UNO and its pins. The pin diagram has been simplified for the purposes of this guide. The full version is available at bit.ly/3jg6PM2.

Apart from allowing an easy interface with external components, these connectors are designed so that add-on printed circuit boards (PCBs), often called shields, can be added. You can buy shields for things like motor control, GPS, and mobile telephony to allow projects to be constructed quickly without knowing much electronics.

There are five pin types that are worth remembering and will be used throughout this guide (refer to Figure 1):

- **Digital inputs** connect to digital output peripherals such as buttons and switches. These are the pins labelled as *D0-D19*.
- **Digital outputs** connect to digital input peripherals such as LEDs, bar graph displays and RGB LEDs. These are the pins labelled as *D0-D19*.
- **Analogue inputs** connect to analogue output peripherals such as potentiometers, light intensity sensors and microphones. These are the pins labelled as *A0-A5*.

© UKESF 2023

- **Analogue outputs** connect to analogue input peripherals such as dimmable LEDs and buzzers. These are the pins labelled as *~D3*, *~D5*, *~D6*, *~D10*, *~D11*. The "analogue" signals from these outputs might not be what you expect, but more on that later.
- **Serial communication** allows the connection to more complex peripherals such as computer, Bluetooth communication peripherals, accelerometers, displays and more! The two serial communication protocols that will be used later in this guide are *I^2C* and *UART*. Their connection pins are *SDA, SCL* and *RX*, *TX* respectively.

Links to explanatory pages are added whenever a new *term* is encountered! Use them if something sounds unfamiliar.

Inside the microcontroller on the Arduino Uno, there are several complex peripherals that the Arduino platform hides from the programmer by employing high-level functions to control them based on relatively simple instructions. The disadvantage of such system is that the user has much less control of the detailed operation of the peripherals. This is only a disadvantage, however, if the programmer wants to have such level of control. This disadvantage is also an advantage to people new to microcontrollers, as it allows the user to make things work relatively quickly. Arduino does support low-level access to the microcontroller's peripherals for those that want or need it. For more introduction to Arduino see the YouTube links below:

Watch "An Introduction to the Arduino" (4:25 minutes): bit.ly/3vc46ZK

Watch "*You can learn Arduino in 15 minutes.*" (16:33 minutes) to get some background on Arduino boards: bit.ly/30urp2H

Grove Beginner Kit for Arduino

This guide is written around the Grove beginner kit for Arduino by Seeed Studio. The Grove system is a modification to the original Arduino Uno which brings various pins out on four-pin connectors and various peripherals that you connect as needed, as shown in Figure 2. The four-pin connectors have a similar naming convention to the Arduino Uno pins discussed in the previous section. Note that this kit is self-contained and you don't need a separate Arduino Uno to go through the guide! The Seeeduino Lotus is essentially a modified Arduino Uno with improved hardware interface.

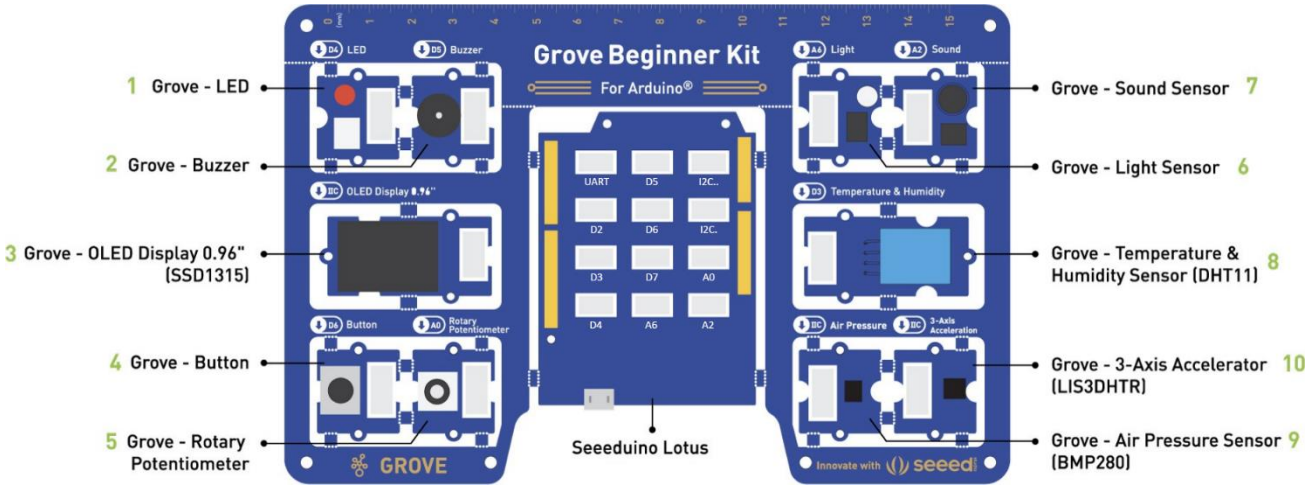


Figure 2. The Grove beginner kit for Arduino comprising of the Seeeduino Lotus and 10 peripherals. (Adapted from bit.ly/2OSLrkZ)

© UKESF 2023

Despite the existence of the four-pin connectors, all peripherals come pre-connected to the Seeeduino Lotus via PCB tracks (see Figure 3), so no cables are needed to connect (unless explicitly mentioned otherwise). Once the peripherals are broken out of the big PCB, the provided Grove cables can be used to connect to the Seeeduino. It is recommended to leave the peripherals attached for this practical!

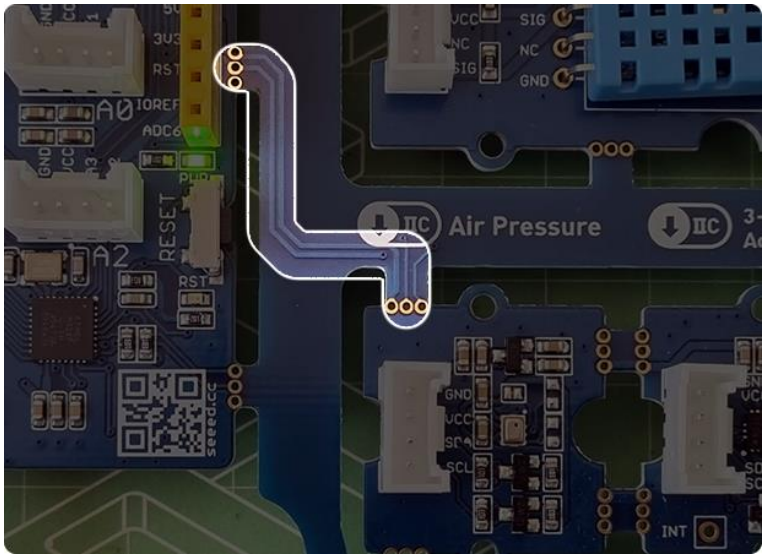


Figure 3. The PCB tracks, that go through the break-out points of the PCB and connect the peripherals to the Arduino. Thanks to these, no cables are needed to use the peripherals.

© UKESF 2023

Prerequisites

Before diving into the fun stuff, the Arduino Software (IDE) and the Seeeduino Lotus board need to be installed – this will make the programming of the device possible.

The Arduino IDE

The Arduino IDE is available for Windows, Linux, and Mac OS X and can be downloaded from arduino.cc/en/software (Note: the Arduino IDE is currently not easily compatible with Chromebooks and ChromeOS). Once installed, launching Arduino should show a window like the one in Figure 4 (without the annotations).

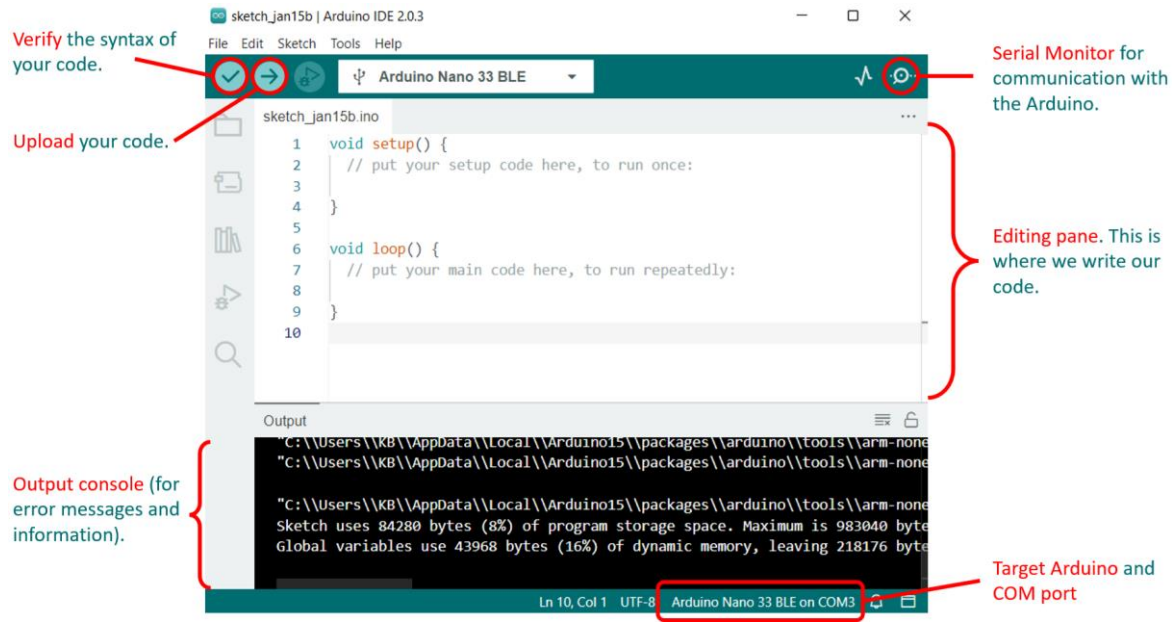


Figure 4. The Arduino IDE (v2.0.3).

The IDE settings can be altered via **File > Preferences**. Here you can change the font, turn line numbers on, etc.

The Seeeduino Lotus Board

To get started with the Seeeduino Lotus board, the CP2102 USB Driver for your OS (e.g. Windows or macOS) needs to be downloaded and installed from the Silicon Labs website bit.ly/3rT2eD8. The installation process is simple, just follow the instructions.

After the driver is installed, connect the Seeeduino Lotus board to your computer using the provided USB cable and open the Arduino IDE.

In the Arduino IDE click on **Tools > Board > Arduino Uno** to select the correct development board model (refer to Figure 5). Then click on **Tools > Port > COMN**, where **N** is the port that your operating system has assigned to the Arduino (in this case it is **COM7**). Note that this number may change each time you connect the board.

© UKESF 2023

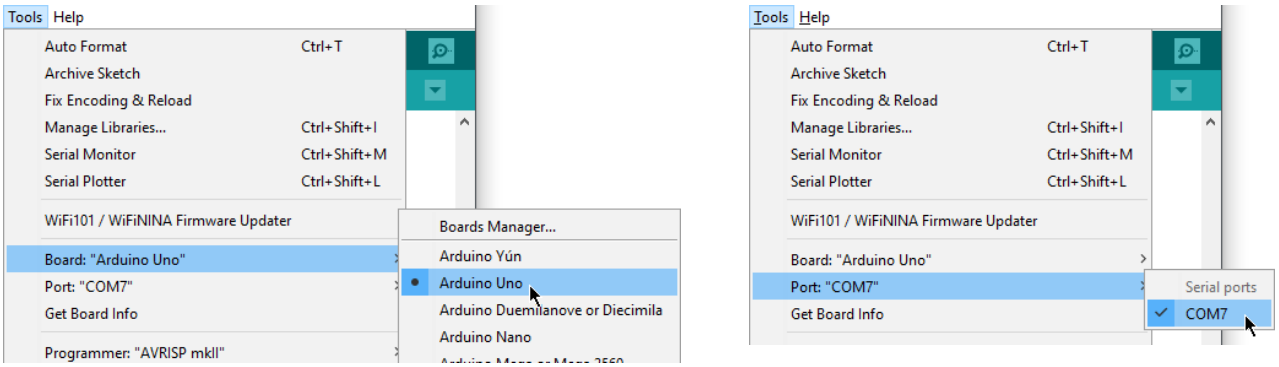


Figure 5. Selecting the correct Board and Port in the Arduino IDE.

The UKESF Sixth-Formers Library

This guide comes with an Arduino library, called *UKESF Sixth-Formers*. A library packs a set of functions that allow you to easily do more with an Arduino that is available out of the box. We'll worry about the details later, but for now we need to install this library so that it is ready. To do this:

- In your Arduino IDE go to **Sketch > Include Library > Manage Libraries…**
- Type in *UKESF Sixth-Formers* and click **Install** as shown in Figure 6 (Make sure to install the **latest version** possible which may differ from the one in the figure below).
- You may be asked to install "…*other library dependencies not currently installed*" – click **Install all.**

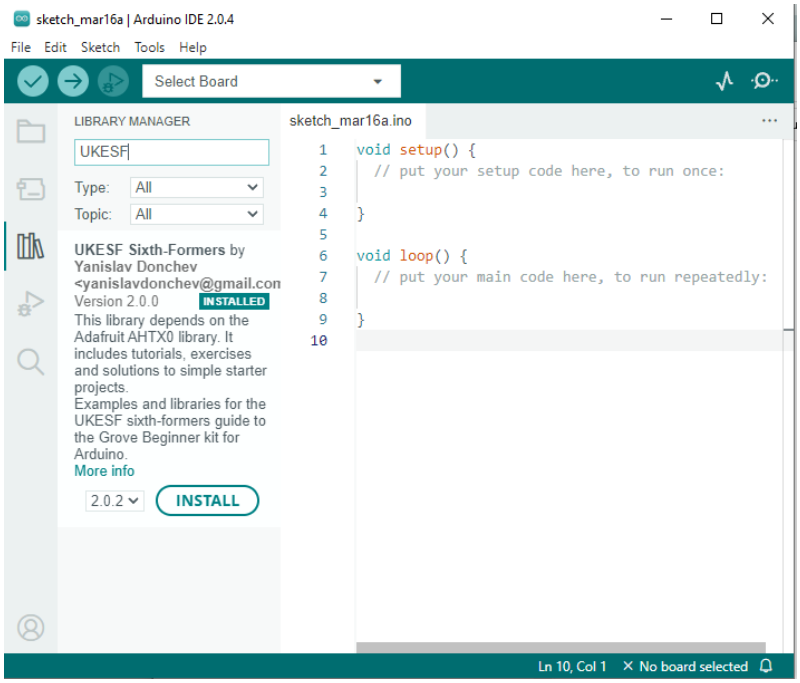


Figure 6. The Library Manager window.

Apart from expanding the Arduino functionality, this library also installs some Arduino sketches, which you'll be using throughout this guide. When doing the tutorials, you will often see comments like "The same code is available in **tutorials/01-LED-Blink/**". When you see this, you would go to **File > Examples > UKESF Sixth-Formers > tutorials > 01-LED-Blink**. This is shown in Figure 7.

© UKESF 2023

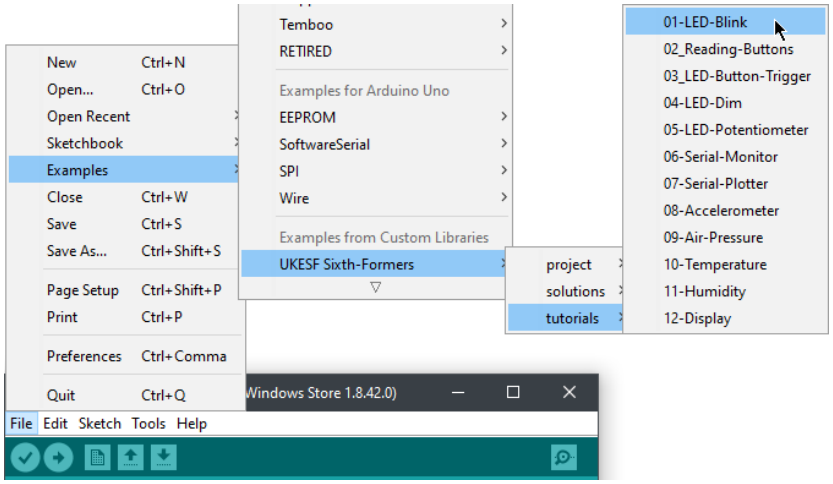


Figure 7. A visualisation of where the UKESF Sixth-Formers Arduino programs can be found.

A Note on the Temperature and Humidity Sensor

The Temperature & Humidity sensor on the Grove kit comes in two types, and you will need to know which one you have later in the guide. Identifying which sensor type you have on your board is straight forward:

- The **DHT11** sensor is housed in a *blue* plastic package and has the label "D3" next to its name on the Grove PCB.
- The **DHT20** sensor is housed in a *black* plastic package and has the label "IIC" next to its name on the Grove PCB.

Once you've identified which sensor type you have, note it down so that we can use it later in our code examples. The two sensor types and their corresponding labels are shown in Figure 8.

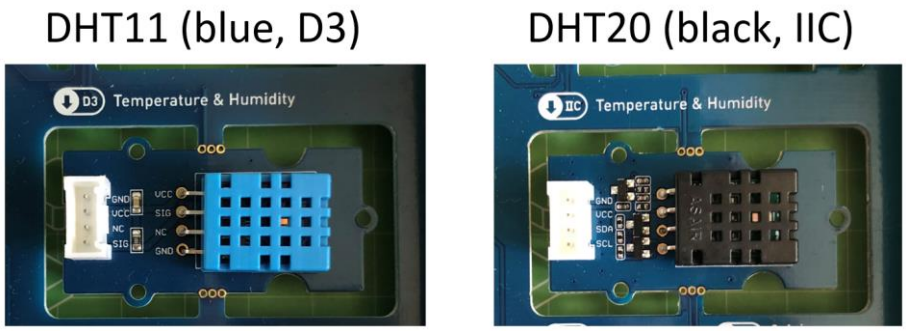


Figure 8. You can identify which Temperature and Humidity sensor type you have on your Grove PCB based on its colour and the label on the top left corner above the sensor.

© UKESF 2023

Digital Outputs

Digitals signals are ones that have a *discrete amplitude* – in other words, the amplitude is one of a limited set of values. In Arduino, digital signals have two states: HIGH / 5 V / 1, and LOW / 0 V / 0. These could represent the state of an LED: setting a pin HIGH will light an LED connected to it up; setting it to LOW will have the opposite effect.

Tutorial – Blinking an LED

Blinking an LED is the "Hello, World!" program for Arduino. It is typically the first program to write when learning a new microcontroller, as it's a simple program but will introduce you to the development environment and test your connection to the hardware, as well as the hardware itself. Once you have done this, you should be comfortable with the basic Arduino system.

The first step is to connect the LED peripheral to the Arduino as shown in Figure 9. Note that if you haven't broken out the peripherals and Arduino from the kit, this step is not necessary since they are connected via the PCB traces; this is applicable to all projects in this practical.

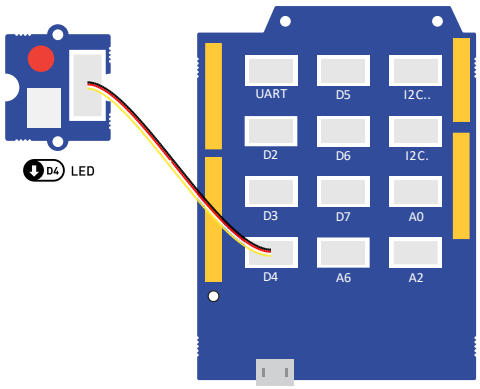


Figure 9. The LED peripheral connected to digital output D4.

Before we examine the typical Arduino program, type Code 1 below into the Arduino IDE. If you opened the Arduino IDE for the first time, you would see the code for an "empty" program with the `setup()` and `loop()` functions which you need to fill in. This is referred to as *sketch.* A sketch is simply a text file which describes your program in the Arduino language. It is recommended that you save your sketches for the programs in this guide to a convenient location using **File > Save As…**. Once you write your program, you can **verify** that its syntax is correct, and **upload** it to your Arduino (refer to Figure 4 if you are unsure how to do this). If there are errors in your code it will not upload, so will not run. Don't ignore errors! Try to work out the cause(s) instead. A common cause is missing semi-colon(s) (;) from the end of statements. If the code ran successfully, you should see the LED blink with a period of half a second.

© UKESF 2023

```
const int ledPin = 4;  // Define the LED pin; It makes the code more readable.

void setup() {
  pinMode(ledPin, OUTPUT);  // Initialize ledPin as an output.
}

void loop() {
  digitalWrite(ledPin, HIGH);  // Turn the LED on (HIGH is the voltage level).
  delay(250);                  // Wait for 250 milliseconds.
  digitalWrite(ledPin, LOW);   // Turn the LED off by making the voltage LOW.
  delay(250);                  // Wait for 250 milliseconds.
}
```

*Code 1. Code that will blink the built-in LED with a period of 500 milliseconds. The same code is available in **tutorials/01-LED-Blink/**.*

Whilst it may seem a little time consuming to get you to type this in yourself it is important that you get used to writing code and finding the typos. The microcontroller can't interpret what you type unless it is 100% correct so it requires an attention to detail.

Now let's understand what the code above does, and how a typical Arduino program looks:

- **The first section**, at the top, is where you put the `const int` and `#include` items. These should be at the top, as the Arduino software reads the program top-to-bottom. The `const int` line creates a *constant integer* variable, called `ledPin` and sets its value to `4`, which refers to pin D4 of the Arduino. The `#include` statements are used to *include* other pieces of code, called libraries, in your program – more on that later!
- **The second section** is the first function you need and is called `setup()` (don't worry about the `void` part). This function is called once at the start of the program and is used for code that initialises things before your main program executes. In this example, the `ledPin` is made an output with the function call `pinMode(ledPin, OUTPUT)`. `pinMode()` takes two arguments: the first is the pin you are controlling, and the second is what you want the pin to be. You can only have one `setup()` function in your program.
- **The third section** is the main body of the program, called `loop()`. It is called `loop()` as it does exactly that – the code is executed in the order it appears, and when the program reaches the closing brace (}), it returns to the top of `loop()` and starts again. In this program, there are two functions that are called twice. The first one is called `digitalWrite(pinNumber, status)`. This function takes two arguments: the first is the pin you are driving (e.g. `ledPin`), and the second is the logic level, which can be `HIGH` or `LOW`. The other function is `delay(ms)` which stops the code execution for the specified time (250 ms in the case above). You can only have one `loop()` function in your program.

You should have also notice the *human-readable* parts, starting with `//`. These are called comments and are used to describe what the code is doing. You can write anything inside these comments, since the Arduino program ignores everything on a line that follows `//`.

Exercise – Changing the Period

As an exercise, try adjusting the arguments in the `delay()` so that the LED is off for 750 ms and on for 250 ms.

© UKESF 2023

Key Points

Digital outputs provide on/off functionality to elements, which are connected to the digital output pins (*D0* to *D19*) of the Arduino.

`const int` is used at the top of an Arduino program to define global variables, such as pin definitions. Each Arduino program has two compulsory functions - `setup()`, which is executed once at the beginning of the program, and `loop()`, which is executed repeatedly.

The `pinMode()` function configures a given pin to behave as an input or an output; the value of a digital output pin can be written using `digitalWrite()`. The `delay()` function pauses the program execution for a given amount of time.

The functions above, and all other core Arduino functions, are described in detail in bit.ly/39QD7du.

© UKESF 2023

Digital Inputs

Like digital outputs, digital inputs in Arduino also have two states: HIGH and LOW. For example, these can be the states of a button. In this part of the practical, you will learn how to switch the LED on and off using a button.

First, we need to take a deeper look at *variables*. In the previous section we used a `const int` variable to store the value of an Arduino pin. The type of that variable was `int`, which means that it can store integers. The `const` keyword made that variable a constant, which means that it is not allowed to change its value while the program is running. Arduino has another type called `bool`, which comes from Boolean algebra, and stores a single two-state bit which can either be `true` (`HIGH`) or `false` (`LOW`). We will use this variable type below to store the state of a switch.

Tutorial – Reading Button

To start with the practical part of this section, connect the LED and the button as in Figure 10.

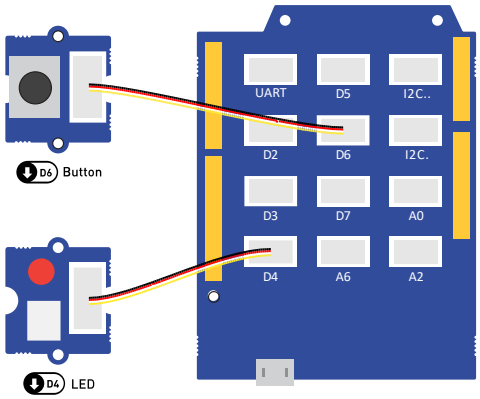


Figure 10. The LED peripheral connected to digital output D4, and the Button connected to digital input D6.

Then, before running the program in Code 2, let's understand it first. On the second line, the button pin is defined the same way as the LED pin is. In `setup()`, there is a new function call to `pinMode()`, which sets the button pin to be an `INPUT` pin. On line 4, a `bool` variable, called `buttonState`, is declared that will store the state of the button. Note that the `const` keyword is not used this time, since we want this value to be able to change during the program execution. The last new thing in the code below is the call to the `digitalRead()` function, which returns the state of the button pin and stores it in the `buttonState` variable. This variable is then used in the `digitalWrite()` to set the state of the LED.

© UKESF 2023

```
const int ledPin = 4;     // Define the LED pin.
const int buttonPin = 6;  // Define the button pin.

bool buttonState;  // A bool variable to store the state of the button.

void setup() {
  pinMode(ledPin, OUTPUT);    // Initialize ledPin as an output.
  pinMode(buttonPin, INPUT);  // Initialize buttonPin as an input.
}

void loop() {
  buttonState = digitalRead(buttonPin);
  digitalWrite(ledPin, buttonState);
}
```

Code 2. Code that will control the LED based on the button state. Note that if we want to refer to pin D4, we only specify 4 in the code and skip the D! The same code is available in ***tutorials/02_Reading-Buttons/****.*

After uploading the code above to the Arduino, you should observe that the LED lights up when you press the button.

Exercise – Inverting the Button State

What if we wanted the LED to be on when the button is not pressed, and have it off when it is pressed? Putting an exclamation mark (`!`) in front of a Boolean will invert its value. For example, if the value, stored in `buttonState` is `true`, then `!buttonState` would equal `false`. Modify the pin state inside the `digitalWrite()` function of Code 2, so that the state that is written to the LED is the opposite to the state of the button. If you need help, a solution is available in **solutions/ 01-Inverting-the-Button-State/**.

Key Points

Digital inputs allow us to read digital output peripherals, which are connected to the digital input pins (*D0* to *D19*) of the Arduino.

When using digital inputs, the `INPUT` mode is used inside the `pinMode()` function.

The value of a digital input pin can be read using `digitalRead()`.

Boolean (`bool`) variables are used to store two-state values which can either be `true` (`HIGH`) or `false` (`LOW`).

Putting an exclamation mark (`!`) in front of a Boolean will invert its value.

© UKESF 2023

Conditional Logic

Although the examples above were useful for learning about digital inputs, they were so simple that the same functionality could be achieved without a microcontroller. For example, the functionality of Code 2 could be replicated by connecting a battery, an LED, and a switch in series. *Conditional logic* allows us to run certain parts of a program when certain conditions are met. For example, suppose that we wanted the LED to turn on when the button is pressed but the LED stays on even after the button is released. To turn the LED off, we would need to press the button again. This is visualised in Figure 11. The switch of a digital signal from low to high is called the rising edge of the signal – the high to low transition is the falling edge.

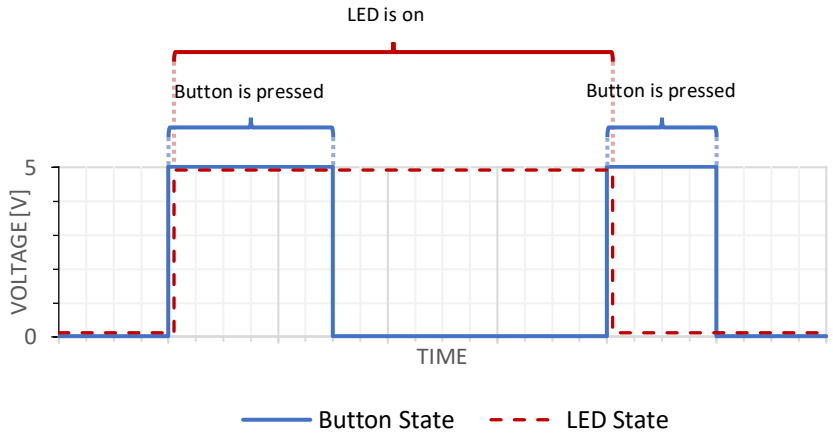


Figure 11. The digital signals of a button and an LED which is turned on/off on the rising edge of the button.

Tutorial – LED Button Trigger

To achieve the desired functionality above, we would need to write a program that monitors the previous and the current state of the button. Whenever the previous state was low, and the current state is high, a rising edge has occurred, and the button must have just been pressed down.

To better understand the code below, the whole functionality of the program can be formulated in an **if … then …** statement: **if** the previous button state was LOW **and** the current button state is HIGH, **then** toggle the LED state. This same functionality is written as Arduino code in the snippet below:

```
if (previousButtonState == LOW && currentButtonState == HIGH) {
  ledState = !ledState;            // Toggle the LED state.
  digitalWrite(ledPin, ledState);  // Update the LED to the new state.
}
```

You should have noticed that the Arduino syntax for the **if … then …** conditions is:

```
if (this condition is met) {
  then do this
}
```

You should have also noticed that the symbol for equality in Arduino is ==, instead of = which is the symbol for assignment. Lastly, you would have noticed the && symbol is the *logical **and** operator*. After putting the

© UKESF 2023

logic above in a complete program, we get Code 3. For this tutorial keep the same connection as in Figure 10. Upload the code to your Arduino and test if the functionality is as expected.

```
const int ledPin = 4;     // Define the LED pin.
const int buttonPin = 6;  // Define the button pin.

bool ledState;             // A bool to store the LED state.
bool previousButtonState;  // A bool to store the previous button state.
bool currentButtonState;   // A bool to store the current button state.

void setup() {
  pinMode(ledPin, OUTPUT);    // Initialize ledPin as an output.
  pinMode(buttonPin, INPUT);  // Initialize buttonPin as an input.
}

void loop() {
  currentButtonState = digitalRead(buttonPin);  // Read current button state.

  // If a rising edge of the button is detected.
  if (previousButtonState == LOW && currentButtonState == HIGH) {
    ledState = !ledState;            // Toggle the LED state.
    digitalWrite(ledPin, ledState);  // Update the LED to the new state.
  }

  previousButtonState = currentButtonState;  // Current state becomes previous.
  delay(10);
}
```

Code 3. Code that will make the LED turn on the rising edge of the button. The same code is available in ***tutorials/03_LED-Button-Trigger/****.*

Exercise – Blinking LED Button Trigger

Suppose that we now wanted a system that would blink twice every time the LED state changes. Assuming that we have set up an output pin called `ledPin`, we could add the following snippet in the conditional statement of Code 3 to get the LED blink twice:

```
digitalWrite(ledPin, HIGH);
delay(100);
digitalWrite(ledPin, LOW);
delay(100);
digitalWrite(ledPin, HIGH);
delay(100);
digitalWrite(ledPin, LOW);
delay(100);
```


© UKESF 2023

That was a long snippet of code for a relatively simple functionality. Now suppose that for some reason we wanted to make the LED blink one hundred times. We could of course write 400 lines of code as described above, but there is a much neater way! The code snippet below allows you to generate any number of blinks, simply by storing the desired number in the `numberOfBlinks` variable. Your task now is to get familiar with the `for` control structure from the Arduino website bit.ly/39QD7du, understand the snippet below and add the functionality described in this subsection to Code 3. As always, if you get stuck, a solution with the completed code is provided in **solutions/02-Blinking-LED-Button-Trigger/**.

```
for (int i = 0; i < numberOfBlinks; i++) {
  digitalWrite(ledPin, HIGH);
  delay(100);
  digitalWrite(ledPin, LOW);
  delay(100);
}
```

Key Points

Conditions in Arduino are written using the following syntax:

```
if (this condition is met) {
  then do this
}
```

The symbol for equality in Arduino is ==, whereas the symbol for assignment is =. The && symbol is the *logical **and** operator*.

The `for` control structure is used to repeat a block of statements enclosed in curly braces.

The signal edge in electronics is the term that describes the transition of a digital signal.

© UKESF 2023

Analogue Outputs

It is unusual for microcontrollers, especially small ones, to include some form of analogue output. However, Arduino includes a function called `analogWrite(pin, val)`. How does it do it? It uses something called *pulse width modulation* (PWM), where `pin` is the pin that you want to control and `val` is the *width* of the PWM ranging from 0 to 255.

But what is PWM exactly? Let's have a look at Figure 12, which shows how the waveform from an "analogue output" of the Arduino would look. As the name PWM suggests, the Arduino generates a series of pulses. The signal, however, is digital since there are only two states: 0 V and 5 V. In fact, if you try to imagine an LED driven by this signal it would be blinking. So why is it called analogue? Your eyes can only respond to relatively slowly changing light. Anything flashing faster than about 50Hz will appear as a constant level. Therefore, if the signal in Figure 12 blinks fast enough, you would notice a dimmer LED, rather than a blinking LED.

Now let's dissect the *width modulation* part in PWM. It essentially means that we can control (or modulate) the width of the pulses. The *duty cycle* describes the proportion of "on" time of the pulse within one period of the waveform. A duty cycle of 50% means that an LED driven by PWM will be "on" 50% of the time. Figure 12 shows that as the duty cycle is increased, the LED appears brighter and vice versa. Referring to the `analogWrite(pin, val)` function, the duty cycle can be inferred using $100 \times \frac{\text{val}}{255}\%$.

© UKESF 2023

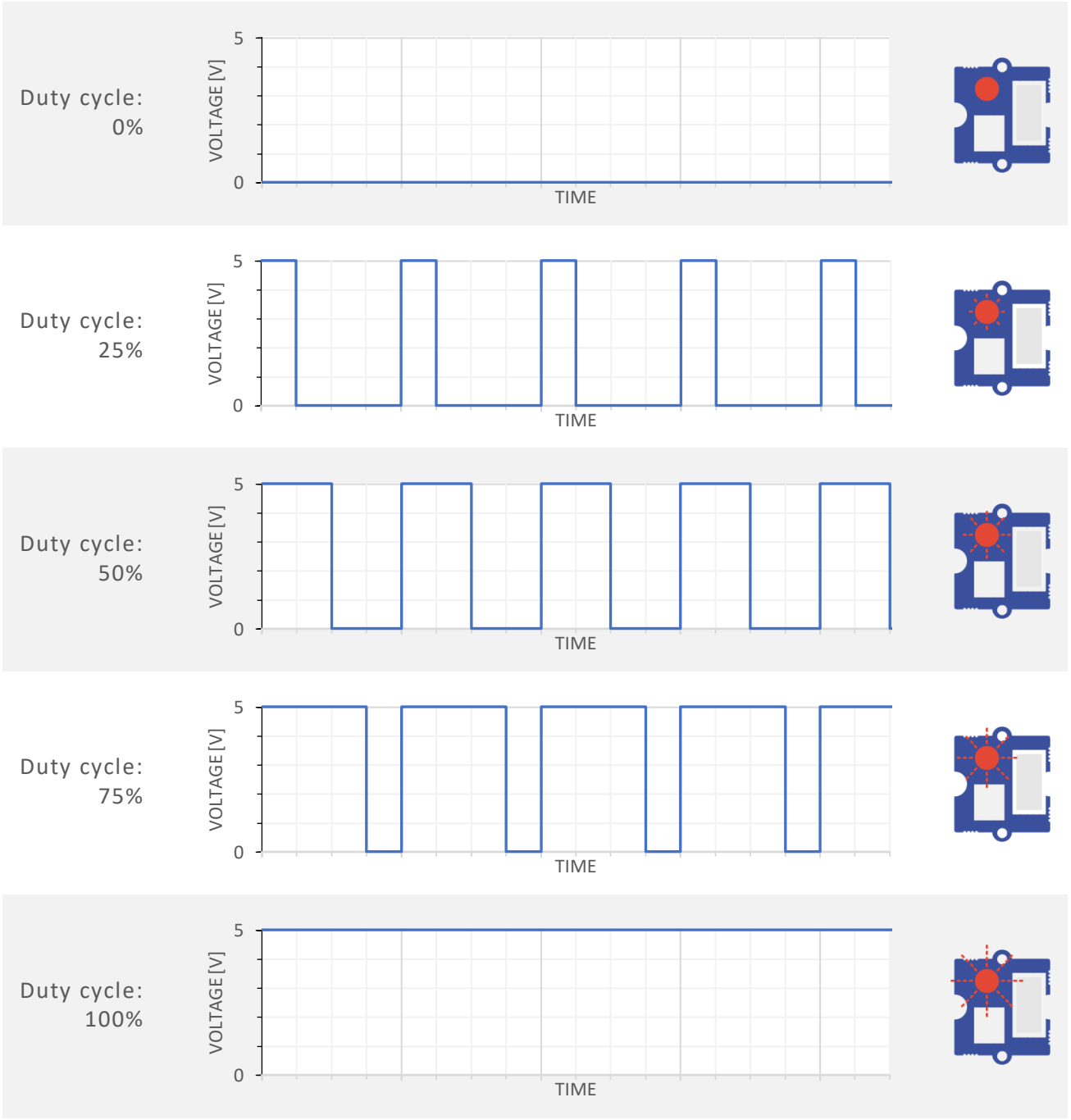


Figure 12. A visualisation of the perceived brightness from an LED driven using PWM for 5 different duty cycles.

Tutorial – Making an LED Dim

Now let's try this in practice. Firstly, connect the LED as shown in Figure 13. Then, in the Arduino IDE type Code 4. You should be comfortable with reading and understanding the code by now – it is like the digital code from before, but it uses the `analogWrite()` function in the `loop()`.

© UKESF 2023

If you didn't break-out any of the sensors, the LED would be automatically connected to pin D4. This pin does not have the ~ sign in front of it, and so it can't be used as an analogue output. To get the code below work correctly, you have to manually connect the LED to one of the analogue pins, such as ~D6. You **don't** have to break-out the LED board for this to work.

Verify and upload the code to the Arduino. What do you observe? You should see an LED that shines at half of its full brightness. Experiment with different values for the `WIDTH` variable, such as 0, 64, 192, 255. What happens to the brightness of the LED as the width is changed?

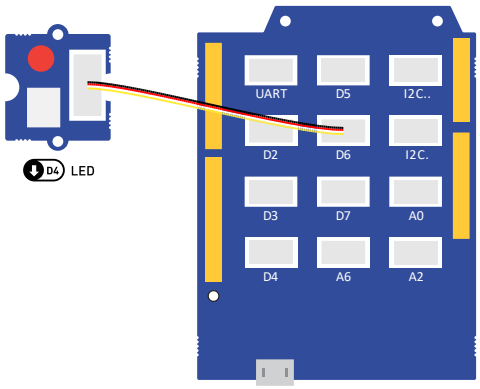


Figure 13. The LED peripheral connected to analogue output D6.

```
const int ledPin = 6;   // Define the LED pin;
const int width = 128;  // Set the duty cycle to (100 * 128 / 255) = 50%.

void setup() {
  pinMode(ledPin, OUTPUT);  // Initialize ledPin as an output.
}

void loop() {
  analogWrite(ledPin, width);  // Drive the LED using PWM.
}
```

*Code 4. Code that will make an LED dim, by driving it using PWM with duty cycle of 50%. The same code is available in **tutorials/04-LED-Dim/**.*

Exercise – Soft Blinking LED

In the digital output section, we started by making an LED blink. In this section we learned how to set an arbitrary brightness level to the LED, however its brightness would stay constant. In this exercise you will learn how to make the LED blink "softly" in a sinusoidal fashion. The complete program is provided for you in Code 5.

© UKESF 2023

```
const int ledPin = 6;          // Define the LED pin;
const float frequency = 0.5;   // Frequency [Hz] for soft-blinking.

void setup() {
  pinMode(ledPin, OUTPUT);  // Initialize ledPin as an output.
}

void loop() {
  // Set the width to 128 * (1 + sin(2 * pi * f * t)).
  int width = 128 * (1 + sin(2 * 3.14 * frequency * millis() / 1000));
  analogWrite(ledPin, width);  // Drive the LED using PWM.
}
```

*Code 5. Code that will blink make an LED dim, by driving it using PWM with duty cycle of 50%. The same code is available in **solutions/03-Soft-Blinking-LED/**.*

Your task is to understand this code, then upload it to the Arduino and observe what happens. The new concepts to look at are `float` variables, the `sin()` and `millis()` functions. As a reminder, all core things to know about Arduino code are available at bit.ly/39QD7du. Once you are comfortable with understanding the code, try different `frequency` values, such as 0.25, 1, 2. What do you observe?

Key Points

Analogue outputs provide a way of generating "analogue" signals using PWM. The relevant Arduino pins are labelled with ~, such as *~D3*, *~D5*, *~D6*, *~D10*, *~D11*.

The analogue value of a signal is controlled by the duty cycle of the PWM signal. This can be calculated using $100 \times \frac{\text{val}}{255}\%$.

The `analogWrite()` is used to control the duty cycle and accepts values between 0 and 255.

The `float` variables allow you to store non-integer values. The `sin()` function allows you to calculate the sine of a given input and the `millis()` function returns the number of milliseconds passed since the Arduino board began running the current program.

© UKESF 2023

Analog Inputs

Arduino programs work in the digital domain. This means that they cannot work with analogue signals directly. Instead, analogue signals are first passed through an analogue-to-digital converter (ADC). This is a circuit that reads an analogue voltage and assigns a number to it that can be processed digitally. In the case of the ADC on the Arduino, it splits the input voltage into 1024 possible steps, with 0 V assigned the value 0 and 5 V assigned to 1023. It is linear, so we can predict the number the ADC will return for any input (see Figure 14). This number is always an integer and is equal to $\left\lfloor \frac{V_{in}}{V_{max}} \times 1023 \right\rfloor$, where V_{in} is the input voltage and V_{max} is the maximum input allowed and $\lfloor \cdot \rfloor$ is the math floor function (essentially it rounds down to the nearest integer). For the Arduino, V_{max} is 5 V. An input of 1.8 V should give us $\left\lfloor \frac{1.8}{5} \times 1023 \right\rfloor = \lfloor 368.28 \rfloor = 368$.

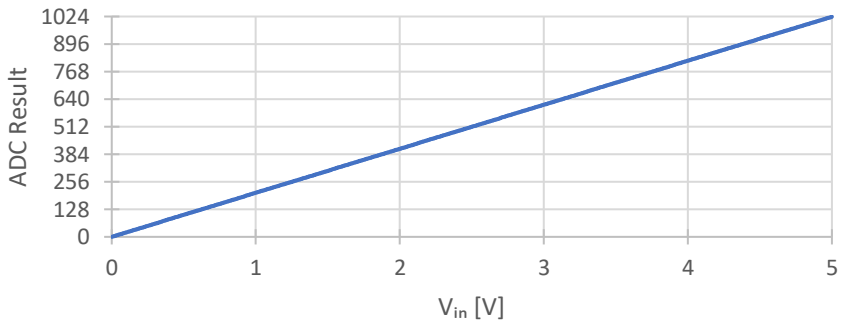


Figure 14. Mapping between analogue input voltages and ADC readings on the Arduino.

Reading an analogue input value in Arduino is done using the `analogRead(pinNumber)` function. As mentioned above, this function returns an integer in the range $[0, 1023]$, which can be stored in a variable of type `int`.

Tutorial – LED with Controllable Intensity

In the previous section, we created a programmatically dimmable LED. However, it was impossible for someone to change its brightness without reprogramming the device. Let's change this and create a program which allows us to change the LED's brightness using a potentiometer. The potentiometer is a device, which changes its resistance based on its rotation. It can be used as a variable voltage divider and the voltage output of this divider, which is in the range of 0 to 5V, can be read using the Arduino's analogue input pins.

We already know how to write an analogue value (from the previous section), and just learned how to read one (using `analogRead()`). One problem that you might have noticed, however, is that the values for analogue outputs range between 0 and 255, whereas those for analogue inputs range between 0 and 1023. We could, of course, convert between these two ranges by multiplying or dividing by 4. However, Arduino has a convenient function called `map()`, which can do the mapping between these ranges for us. You can read about the `map()` function at bit.ly/3tbziqq, or try and infer how it works from the code below – it is quite simple.

To complete this tutorial, connect the rotary potentiometer and the LED as shown in Figure 15. Then, in the Arduino IDE type Code 6 and upload it to the Arduino. After the code is uploaded, try rotating the potentiometer's shaft. You should observe that this changes the LED's brightness.

© UKESF 2023

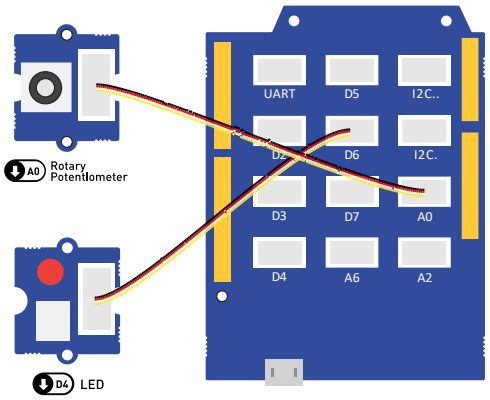


Figure 15. The LED peripheral connected to digital output D4, and the rotary potentiometer connected to analogue input A0.

```
const int ledPin = 6;              // Define the LED pin.
const int potentiometerPin = A0;   // Define the potentiometer pin.

void setup() {
  pinMode(ledPin, OUTPUT);           // Initialize ledPin as an output.
  pinMode(potentiometerPin, INPUT);  // Initialize potentiometerPin as an input.
}

void loop() {
  // Read the potentiometer value in the range [0, 1023].
  int potentiometerValue = analogRead(potentiometerPin);
  // Convert the value from the range [0, 1023] to the range [0, 255].
  int ledValue = map(potentiometerValue, 0, 1023, 0, 255);
  // Set the converted value to the LED.
  analogWrite(ledPin, ledValue);
}
```

*Code 6. Code that will make an LED dim based on the input from a rotary potentiometer. Note that in code, we refer to analogue input pins with their full name (e.g. A0), whilst for digital pins, we omit the D and only type the number! The same code is available in **tutorials/05-LED-Potentiometer/**.*

Now try and change the code such that the `map()` function maps from $[0, 1023]$ to $[255, 0]$ (instead of $[0, 255]$. This can be achieved by swapping the last two arguments (…, `0`, `255`), to be (…, `255`, `0`) and uploading the modified program to the Arduino. What change in behaviour do you observe?

Exercise – Replacing the Potentiometer

Conveniently, all compatible analogue inputs act the same. This means that you can replace the potentiometer with the light sensor or the sound sensor and have an LED that is reactive to light or sound without changing the logic of your program. Try this for yourself by changing `potentiometerPin` to `A2` or `A6`!

© UKESF 2023

Key Points

Analogue input signals need to be converted to digital via an ADC, before using them in Arduino. The relevant Arduino pins are labeled as $A\#$ (e.g. $A0$).

The `analogRead(pinNumber)` function is used to read analogue inputs. It returns an integer in the range $[0, 1023]$.

The `map()` function can be used to map between two ranges, such as the range of analogue inputs $[0, 1023]$ and analogue outputs $[0, 255]$.

© UKESF 2023

Serial Communication

So far, we interfaced the Arduino to both digital and analogue inputs and outputs, and this has probably sparked your creativity and exposed you to some of the many things Arduino can do. But how do we interact with more complex devices, such as a digital accelerometer, OLED displays and even your own computer? The answer is serial communication. Essentially, serial communication works by converting information to a stream of bits, which are sent over a wire. Don't worry if this sounds complicated, you don't need to understand the details to get it working.

Tutorial – Communicating with a Computer

The Serial family of functions in Arduino allow us to use the UART communication protocol, which can be used to communicate to a computer. To use these functions, we need to first call `Serial.begin(9600)` (don't worry about the `9600`, it is related to the speed of communication). Once you start using Arduino, you'll notice the `begin()` come often.

At this point, you can use functions from the Serial family, such as `Serial.print()`, which sends the text in the parentheses to the computer, and `Serial.println()`, which does the same, but also starts a new line. To test this functionality, type Code 7 in the Arduino IDE, and upload. Then, in the Arduino IDE, go to **Tools > Serial Monitor** (or click the looking glass icon in the top-right), which will open a window, like the one in Figure 16, showing the message we sent from Arduino.

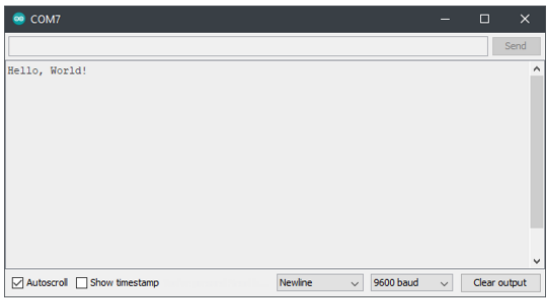
Try sending different messages using both, `Serial.print()` and `Serial.println()`, to get a feeling of how they work.

```
void setup() {
  Serial.begin(9600);               // Begin the Serial communication.
  Serial.println("Hello, World!");  // Send a message to the computer.
}

void loop() {}
```

*Code 7. Code that will send a "Hello, World!" message to a computer over the USB cable. The same code is available in **tutorials/06-Serial-Monitor/**.*

Serial monitor in Arduino v 1.8



Serial monitor in Arduino v 2.0

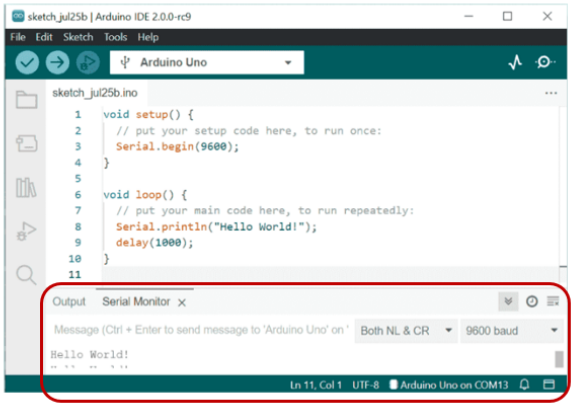


Figure 16. A preview of the Serial Monitor tool in Arduino IDE 1.8 and 2.0, showing the "Hello, World!" message that was sent over USB from the Arduino.

© UKESF 2023

Another useful tool in the Arduino IDE is the *Serial Plotter*. To learn what it does, make the connections show in Figure 17, then type in Code 8, and upload it to the Arduino. Then, go to **Tools > Serial Plotter** (Note this is not the same as the **Serial Monitor**).

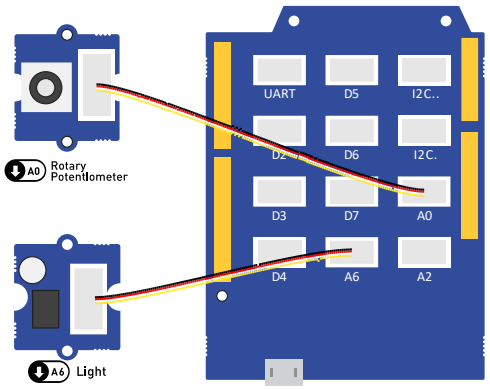


Figure 17. The rotary potentiometer connected to analogue input A0, and the light sensor connected to analogue input A6.

```
const int potentiometerPin = A0;  // Define the potentiometer pin.
const int lightSensorPin = A6;    // Define the light sensor pin.

void setup() {
  pinMode(potentiometerPin, INPUT);  // Initialize potentiometerPin as an input.
  pinMode(lightSensorPin, INPUT);    // Initialize lightSensorPin as an input.
  Serial.begin(9600);                // Begin the Serial communication.
}

void loop() {
  // Read the analogue values in the range [0, 1023].
  int potentiometerValue = analogRead(potentiometerPin);
  int lightSensorValue = analogRead(lightSensorPin);
  // Send the readings to the computer in a space-separated format.
  Serial.print(potentiometerValue);
  Serial.print(" ");
  Serial.println(lightSensorValue);
  // Wait for 100 milliseconds.
  delay(100);
}
```

*Code 8. Code that will send readings from the potentiometer and light sensor to a computer over the USB cable. The same code is available in **tutorials/07-Serial-Plotter/**.*

You should see something similar to the window in Figure 18, which plots the readings from the potentiometer and the light sensor in real-time. Try rotating the potentiometer and putting your finger on top of the light sensor to block some of the light and see observe the readings change in the Serial Plotter window.

© UKESF 2023

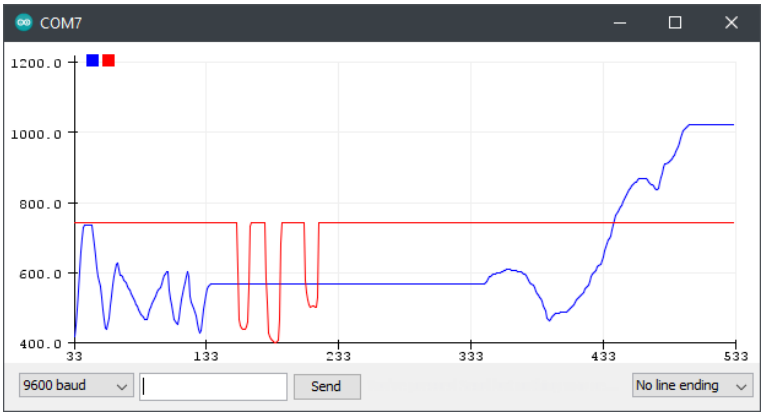


Figure 18. A preview of the Serial Plotter tool in Arduino IDE, showing the readings from the potentiometer and light sensor that were sent over USB from the Arduino.

Tutorial – Reading and Plotting Acceleration Data

So far, you have learnt how to send data over to a computer, using UART and the `Serial` functions in Arduino. In this tutorial, you will learn how to use I²C to read data from digital peripheral, such as a digital accelerometer. If you looked at the I²C specification, you would notice that it is rather complicated and requires the use of awkward device and register addresses. Luckily, many of the Arduino sensors come with libraries, which provide user-friendly programming interfaces and hide away all the address handling. This guide comes with the `UkesfSixthFormers` library, which you should have installed since the Prerequisites sections. This library allows you to easily use the accelerometer, air pressure sensor, temperature and humidity sensor, and OLED display.

To learn how to use this library, make the connection in Figure 19, then study Code 9, type it in, and upload to the Arduino. You should have noticed four things:

- At the top, the `UkesfSixthFormers` library is included, using `#include <UkesfSixthFormers.h>`.
- After that, an *instance* of the accelerometer is made using `Accelerometer accelerometer`; here `Accelerometer` is the type of device we are instantiating, and `accelerometer` is the name we assign to it, so that we can refer to it later in the code.
- In `setup()`, we call `accelerometer.begin()` to begin the accelerometer, the same way we did with `Serial`.
- All this allows us to use functions, such as `accelerometer.readX()`, to read the accelerometer readings.

After uploading this code, start the Serial Plotter, and shake the accelerometer. You should see the waveform moving accordingly. If you leave the accelerometer stationary, you will notice that the acceleration doesn't go to zero. That's because it also measures acceleration due to gravity.

© UKESF 2023

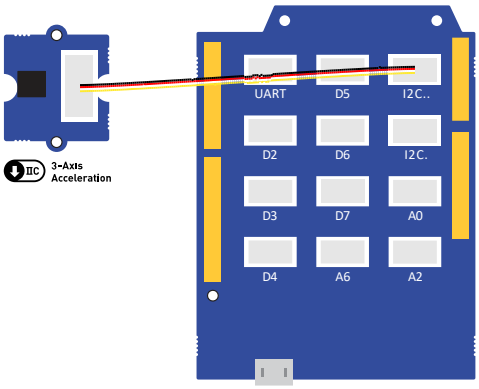


Figure 19. The accelerometer connected to the I²C port.

```
// Include the UKESF library. This will allow you to use peripherals, such as
// the accelerometer, thermometer, OLED display, etc.
#include <UkesfSixthFormers.h>

Accelerometer accelerometer;  // Create an instance of the accelerometer.

void setup() {
  Serial.begin(9600);     // Begin the Serial communication.
  accelerometer.begin();  // Begin the Accelerometer.
}

void loop() {
  float x = accelerometer.readX();  // Read acceleration in the x axis [rad/s].
  float y = accelerometer.readY();  // Read acceleration in the y axis [rad/s].
  float z = accelerometer.readZ();  // Read acceleration in the z axis [rad/s].
  Serial.print(x);
  Serial.print(" ");
  Serial.print(y);
  Serial.print(" ");
  Serial.println(z);
}
```

*Code 9. Code that will send readings from the accelerometer sensor to a computer over the USB cable. The same code is available in **tutorials/08-Accelerometer/**.*

Exercise – Exploring Serial Devices

As mentioned above, the `UkesfSixthFormers` library also allows you to use the air pressure sensor, temperature and humidity sensor, and OLED display. To learn how to use these peripherals, study the provided relevant code in:

- **tutorials/09-Air-Pressure**
- **tutorials/10-Temperature**
- **tutorials/11-Humidity**
- **tutorials/12-Display**

You would notice that the way to use them is very similar to the way we used the accelerometer.

© UKESF 2023

Key Points

Arduino uses the Serial family of functions to communicate with a computer over USB using the UART protocol.

The **Serial Monitor** and **Serial Plotter** tools can be used to visualise data sent from an Arduino to your computer.

Libraries, such as the `UkesfSixthFormers` library can be `#include`d to bring extra functionality to your code. Such libraries are often used to abstract away the complexities of using serial communications peripherals, such as ones that communicate over I^2C.

© UKESF 2023

Final Project – Weather Station

In the final project, you will be using everything you have learnt in this guide, the air pressure sensor, temperature and humidity sensor, and OLED display to create the weather station shown in Figure 20.

Figure 20. A picture of the weather station that you will be building. It shows the current temperature in °C, humidity in %, and air pressure in hPa.

To start, make the connections from
Figure 21. Then type in *Code 10* in your Arduino IDE. If you try to upload this code, you will notice that you get an error. That's because the code is incomplete and is your task to make it work. Throughout the code you will find comments, starting with TODO, that tell you what needs to be filled in. Complete the code, and check if the weather station looks like the one in Figure 20. If you need help, look through the examples in **tutorials/09-Air-Pressure**, **tutorials/10-Temperature**, and **tutorials/11-Humidity** to remind yourself how to use the sensors. Also remember to identify which DHT temperature & humidity sensor you have (see Figure 8 for help with this). As a final resort, you can find the complete solution in **project/Weather-Station-Solution**.

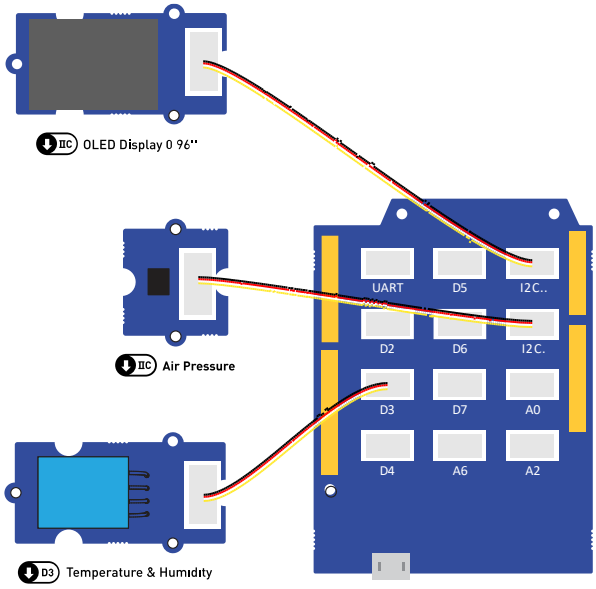


Figure 21. The OLED display, air pressure sensor, and temperature and humidity sensor connected to the serial port of the Arduino.

© UKESF 2023

```ino
#include <UkesfSixthFormers.h>

// TODO: Define the barometer and temp & humidity sensor here.
WeatherStation weatherStation;

void setup(void) {
  // TODO: Begin the barometer and temp & humidity sensor here.
  weatherStation.begin();
}

void loop(void) {
  float pressure = // TODO: Read the barometer here.
  float humidity = // TODO: Read the hygrometer here.
  float temperature = // TODO: Read the thermometer here.

  weatherStation.setAirPressure(pressure);
  weatherStation.setHumidity(humidity);
  weatherStation.setTemperature(temperature);
  delay(10);
}
```

*Code 10. The starter code that you are provided for the weather station. Fill in the rest of the comment in place of the TODO comments to complete the functionality. The same code is available in **project/Weather-Station-Todo/**.*

© UKESF 2023

Next Steps

Congratulations on making it this far, you've learned a lot! What's important is that you remember it and apply it. There are lots of other things you can do with the kit you have and what you have learned so far.

If you paid attention, you would notice that one peripheral was not used in this guide. That's the buzzer! You should feel comfortable enough to learn how to use it on your own now and create a program that generates Morse code. I'll give you a hint, however – there is a function in Arduino, called `tone()`, which is specifically designed to control buzzers.

Remember, coding requires the following:

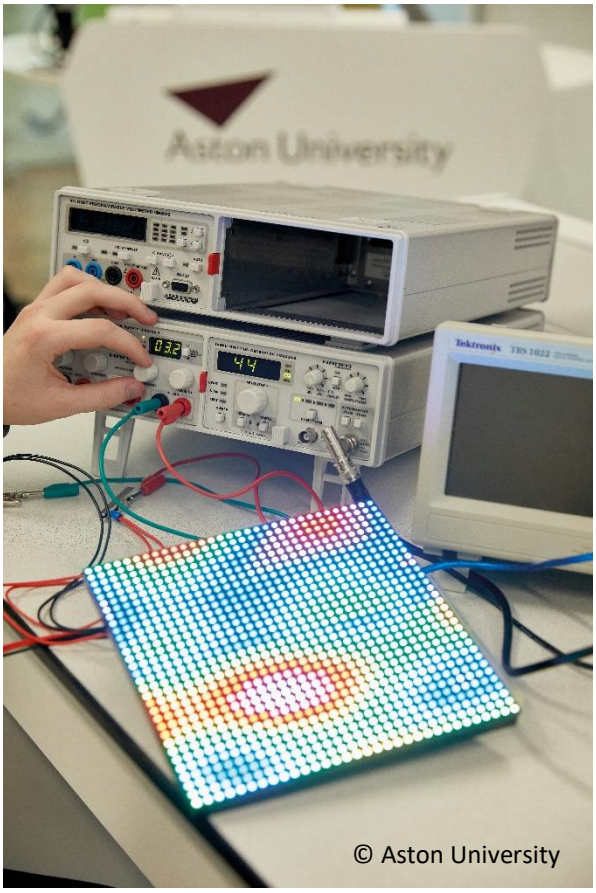
- Care that you get the syntax correct. For those of us that find remembering syntax a challenge it is useful to compile a lookup sheet.
- Well thought out logic – even if a program runs it will only do what you have told it to do, so if your logic is incorrect the outcome will be incorrect.
- Calling things sensible names means that your programmes are easier to find and debug.
- Lastly, it requires practice…

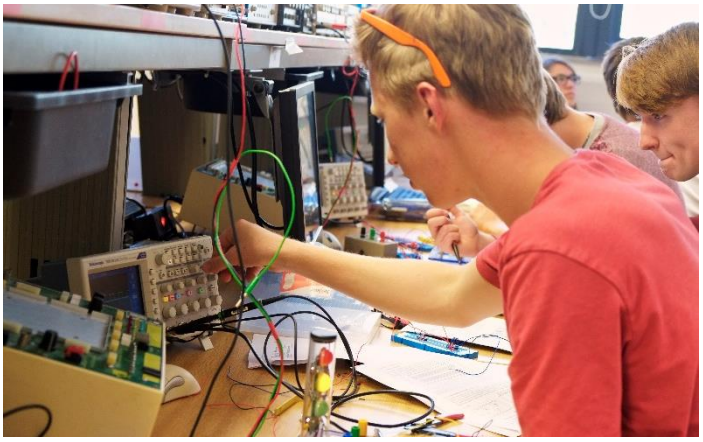
© UKESF 2023

Acknowledgements

This material was developed by Yanislav Donchev, a former UKESF Scholar. It was reviewed by Kasper Buckbee, another former UKESF Scholar.

It is based upon some Digital Engineering practical labs originally developed by Richard Reeves and Prof Kate Sugden from the Electronic Engineering Department at Aston University.





© UKESF 2023