

ARDUINO WORKSHOP GUIDEBOOK



Revision: C

EVOLOCITY



THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING
DEPARTMENT OF MECHANICAL
ENGINEERING



THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

ENGINEERING
DEPARTMENT OF ELECTRICAL,
COMPUTER, AND SOFTWARE ENGINEERING

Table of Contents

Table of Contents	2
Introduction to Arduino and Electronics	5
What is Arduino?	5
The Fundamentals of Arduino	6
Resources	6
Microcontrollers	6
Input and Output (I/O) Pins	6
Power	7
Communication	7
How Arduino Works	8
Why Arduino is Useful	8
Programming Fundamentals	9
How Programs Work	9
Operations	9
Variables	9
Conditional Statements	10
Data Types	10
Functions	11
Flow Control Statements	11
Loops	12
Comments	13
The Structure of an Arduino Sketch	13
Includes and Defines	13
Variables and Functions	14
Communication Protocols	18
SPI	18
I2C	19
Serial/UART	20
Electronics Fundamentals	21
What is Electricity?	21
Voltage (V)	21
Current (I)	21
Resistance (R)	21
Ohm's Law	22
Alternating Current and Direct Current	23
Analog vs. Digital Signals	24

Analog Signals	24
Digital Signals	24
Why They Differ and Their Uses	25
Computers and Analog Signals	25
Electronic Components	27
Resistors	27
Capacitors	27
Transistors	28
Diodes	28
Electronic Structures	28
Voltage Dividers	28
Getting Started	30
EVolocity Arduino Motherboard (AMB)	30
The Arduino IDE	31
Exercises & Examples	32
1 Digital I/O	32
1.1 Blinking an LED	32
1.2 Reading a Button	34
2 The Serial Monitor	37
2.1 Sending a Serial Message	37
3 Analog I/O	40
3.1 Reading the Dial	40
3.2 Using a Potentiometer to Control LEDs	43
3.3 Measuring Battery Voltage	45
3.4 Measuring Battery Temperature	47
4 Communication	51
4.1 Communicating with an LCD	51
4.2 Measuring Data from the BME280 Sensor	54
5 Putting It All Together	57
Conclusion	58
Revision History	60

Introduction to Arduino and Electronics

What is Arduino?

Arduino is an open-source* electronics platform comprised of two parts; the Arduino development board (the hardware), and the Arduino IDE (the software used to program the hardware). Arduino boards can be programmed to read inputs – like light on a sensor, a finger on a button, or a text message – and turn them into outputs – activating a motor, turning on an LED, publishing something online. The Arduino platform is designed to provide a simple and accessible way to develop interactive electronics projects.

The Arduino platform features a vast and active online community with example projects, thorough documentation and tutorials, a wide array of compatible components, and thousands of hobbyists, experts, and professionals who regularly contribute to and support the platform. There is also an entire family of different Arduino boards, each with their own unique strengths and use-cases.

*Open-source means publicly accessible, allowing anyone to use, modify, and distribute it.

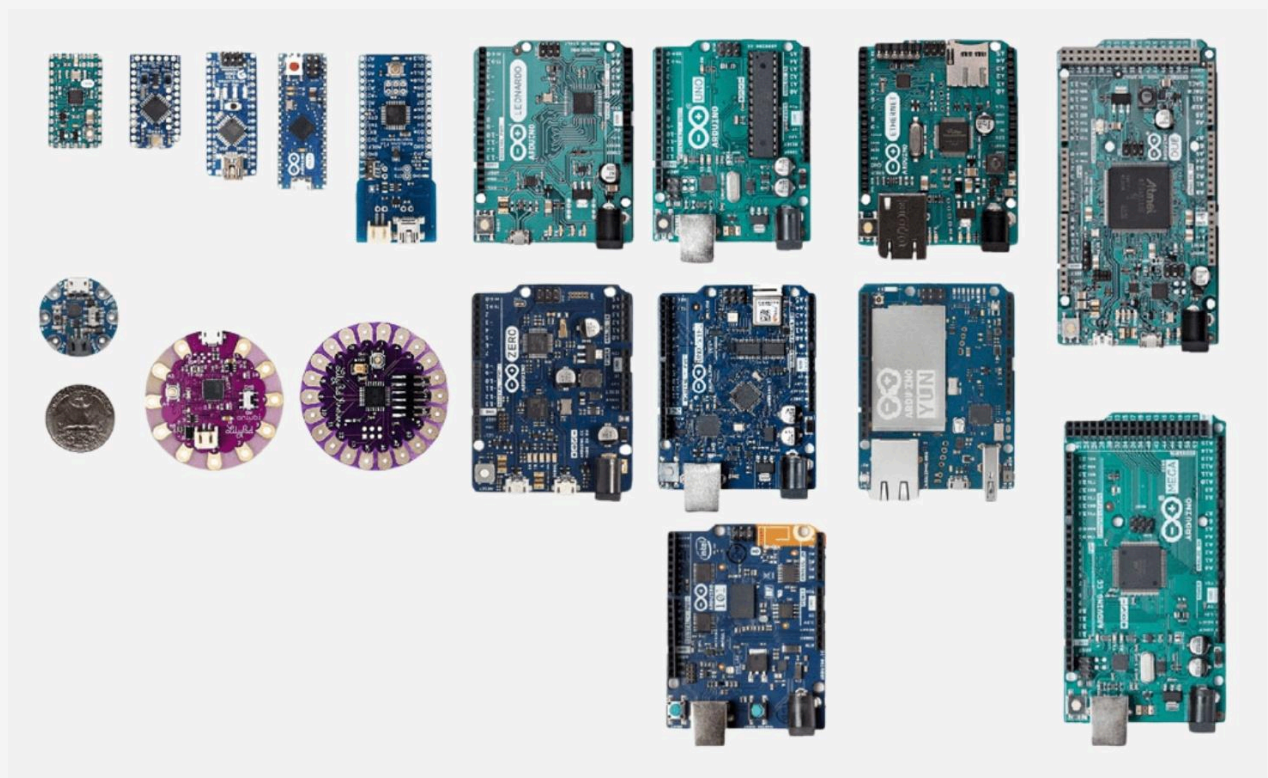


Image source: <https://electronics-fun.com/wp-content/uploads/2022/09/Arduino-Comparison.png>

The Fundamentals of Arduino

Resources

The Arduino platform provides extensive guides and documentation, and learning how to use these resources is an essential part of Arduino proficiency. These provide both a starting point and the necessary demonstrations to allow you to extend beyond the EVolocity Arduino project. Here are two important links to save for later:

- Getting started: <https://www.arduino.cc/en/Guide>
- Arduino Nano: <https://docs.arduino.cc/hardware/nano/>

Microcontrollers

At the heart of every Arduino board is a microcontroller – a small computer on a single integrated circuit (IC) containing a processor core, memory, and programmable input/output peripherals. Your Arduino Nano's microcontroller runs code that you write, and is more powerful than the computer used to land people on the moon!

However, it's important to note that the Arduino board itself is not the microcontroller. The Arduino board *has* a microcontroller - the same way a car has an engine. The Arduino itself is called a development board, and the Nano in particular contains an ATmega328P, an 8-bit microcontroller manufactured by Atmel. Some Arduino boards have different microcontrollers, and some are more or less powerful than others.

Input and Output (I/O) Pins

If you take a look at your Arduino Nano board, you will see some little metal "legs" - these are called pins. Pins can be used as either an input or an output, and these are how the Arduino connects to other devices:

- Input Pins: These receive data from sensors or other devices.
- Output Pins: These send data to other devices, such as turning on an LED or powering a motor.

Some pins have extra features, like being able to measure analog signals, or send high-speed data to other components. Below is a diagram of the Arduino Nano pins and what they are used for. This type of diagram is called a "pinout".

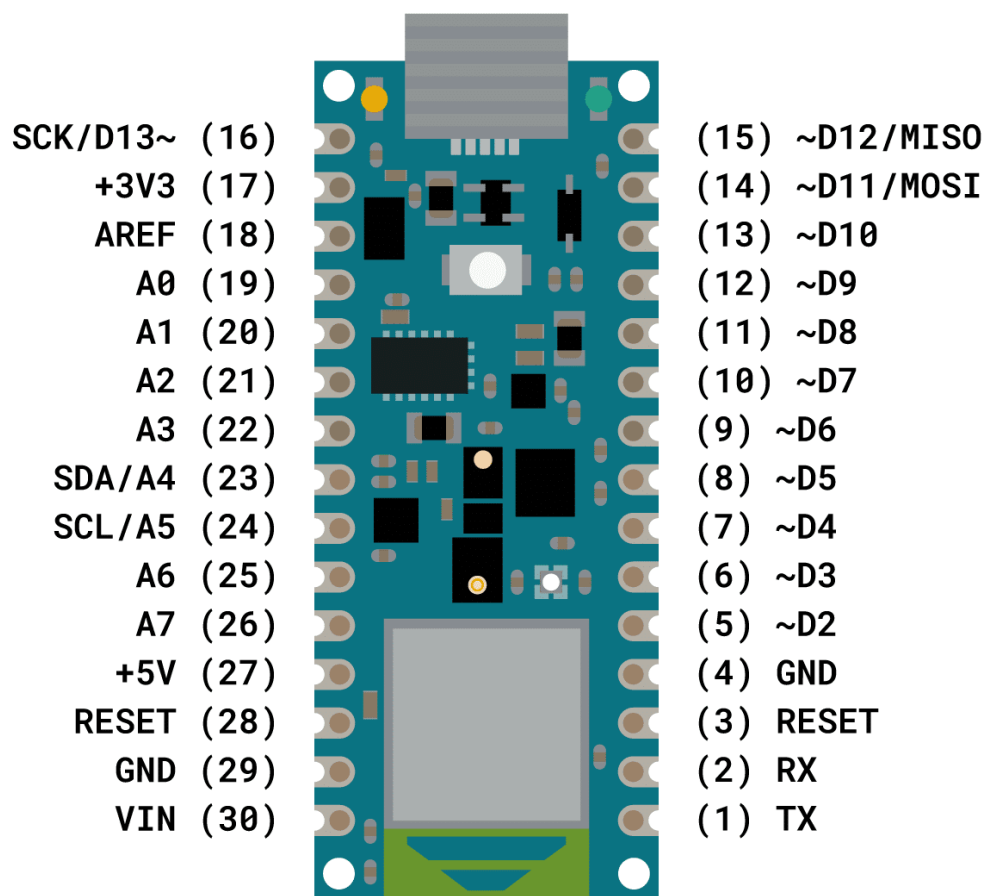


Image source: <https://www.makerguides.com/wp-content/uploads/2020/10/arduino-nano-33-ble-sense-pinout.png>

The official Arduino Nano pinout can be found at the link below.

<https://docs.arduino.cc/resources/pinouts/A000005-full-pinout.pdf>

Power

Arduino boards can be powered either through a USB connection or with an external power supply. They have an on-board voltage regulator that takes higher voltages and reduces them to the operating voltage of the board, meaning they can be powered by batteries or other power supplies that may provide a voltage otherwise too high or inconsistent. This makes them versatile for different types of projects. More on voltages later.

Communication

Arduino boards can communicate with a computer, other Arduino boards, or other devices like humidity sensors, temperature sensors, or SD card readers. This can be done through various communication protocols (languages) like Serial, I²C, and SPI, and the Arduino Nano

has specific pins dedicated to these functions (see the pinout above). Again this will be discussed in more detail throughout this guide.

How Arduino Works

The process of working with Arduino can be broken down into a few key steps:

1. **Writing code:** You write a program (called a sketch) using the Arduino programming language, which is based on C/C++. This is done in the Arduino Integrated Development Environment (IDE), the official Arduino programming software.
2. **Uploading code:** Once the code is written, you upload it to the Arduino board via a USB connection. The program gets stored in memory inside the microcontroller.
3. **Running the program:** Once powered, the microcontroller on the Arduino board runs your code automatically, interacting with any connected devices through the I/O pins.
4. **Testing:** Based on the behaviour of the Arduino and connected devices, you can adjust your code and re-upload it as needed (going back to step 1).

We suggest working in small steps. Make one small change at a time and test before moving on. You will always encounter problems, and these are much harder to diagnose when there are multiple possible failure points. Taking smaller steps will help you finish your project faster and with fewer headaches.

Why Arduino is Useful

Arduino is a powerful tool for learning and prototyping because:

- **It simplifies complex processes:** Arduino abstracts many of the complexities of electronics and programming, making it easier for beginners to get started.
- **It's versatile and flexible:** You can use Arduino for a wide range of projects, from simple LED blinkers to complex robots.
- **Has a large community:** Being one of the most popular development platforms on the planet, there is a large community of Arduino users and many resources available online, including tutorials, forums, and project examples. Much of the time you can get a project working with almost no coding knowledge.
- **It's open-source:** Both the hardware and software are open-source, which means you can modify and adapt them to suit your needs.

Programming Fundamentals

How Programs Work

At their core, computers are tiny machines that execute a list of instructions. It's kind of like following a recipe, and a computer will always do exactly what you tell it to do.

Unfortunately, these instructions are pretty hard for a human to understand. The computer only understands binary digits, or bits, which are nothing more than 1's and 0's. That's why we use programming languages, and we use software called a compiler or interpreter to turn human-readable programming languages into machine code (binary). The Arduino IDE comes with a compiler.

Each programming language is different, but they're all designed to make writing the computer's instructions easier for humans. While we're going to focus on the Arduino programming language, all of the most common programming languages use the same basic building blocks: operations, variables, functions, flow control statements, and loops. The difference lies in their syntax - the way instructions are written.

Operations

If you've ever used a calculator, then you already understand operations! You can add, subtract, divide, multiply, and more.

```
1 + 1 // Add
1 - 1 // Subtract
1 / 1 // Divide
1 * 1 // Multiply
```

Variables

Variables allow us to assign labels to a value, and we can even change the variable over time. This allows us to do more complicated things in our code, like increment a number, count how many times a button has been pressed, or track certain states like altitude or position. Try and figure out what the value of the variable called "a" is at the end of this piece of code.

```
a = 1 + 1;
b = 4 / 2;
a = b * 4;
a = b - 2;
a = b / 2; // What is the value of "a" now?
```

Conditional Statements

Conditional statements allow us to perform comparisons between variables and make decisions based on the outcomes. You will be familiar with these from mathematics, and it's important to note these symbols can have different meanings when used in certain ways.

```
A < B; // Less than: returns true if A is less than B.  
A > B; // Greater than: returns true if A is greater than B.  
A != B; // Not equal: returns true if A is NOT equal to B.  
A == B; // Equal: returns true if A is equal to B.  
// This is not to be confused with A = B, which makes A equal to B.
```

The results can then be used in conjunction with flow control statements like if-else. More on these shortly.

Data Types

Many programming languages also have the concept of “data types” for each variable. The data type tells us what kind of information we can store in a variable, and affects the way that we can use our variables. Here are some of the most common data types you'll encounter, and with just a little bit of practice, you will probably be designing and using your own!

- **int**: Used to store a whole number, also known as an *integer*. It is “signed”, meaning it can be positive or negative. It must be between the values -32768 and 32767.

```
int sensorValue = 1;
```

- **unsigned int**: Used to store a *positive* whole number in the range 0 to 65535. These can be used in place of **int** if you need to hold larger values and you know for certain your number will only ever be positive, e.g when counting button presses.

```
unsigned int sensorValue = 1;
```

- **long**: A longer integer type. It can be positive or negative, and can hold values between -2,147,483,648 and 2,147,483,647. There is also the unsigned variant, which allows twice the positive values but no negative values.

```
long sensorValue = 0;  
unsigned long sensorValue = 0;
```

- **float**: Used to store a floating-point number (a number with decimals). Note that a float takes more time to process, especially when doing mathematical operations, so they should only be used when necessary!

```
float temperature = 1.1;
```

- **char**: Used to store a single character/letter.

```
char letter = 'A';
```

- **bool**: Used to store a Boolean value (true or false). These are used when you expect one of only two possible outcomes. These can also contain a 1 or a 0 which equate to true and false respectively.

```
bool ledState = false; // The LED is off
```

Functions

Functions are blocks of code that let us tidy up and simplify our program. We make our own functions and give them names to make our code easier to read, and reuse code instead of copy-pasting large chunks. If you end up writing two pieces of code that do the same thing, it's the perfect opportunity to use a function!

```
// Without a function
pi = 3.14159;
a = pi * 10 * 10; // a is the area of a circle with a radius of 10!
b = pi * 5 * 5; // b is the area of a circle with a radius of 5!

// With a function called "area()" (note that the function definition is
// not shown here, only how the function would be used)
a = area(10);
b = area(5);
```

Flow Control Statements

Sometimes we might not want to run a piece of code, or we might want to run different pieces of code based on the value of a variable or a function. This is where we can use flow control statements. For example, we can use the **if-else** statement to decide whether a light

should be turned on or off. We have used a conditional statement inside our flow control statement to make our decision:

```
if (light == ON) {    // If the light is on...
    light = OFF;      // do this.
} else {              // Otherwise..
    light = ON;       // do that.
}
```

Loops

Loops let us run the same piece of code over and over again. Let's try counting to 10 with a *while* loop:

```
count = 0;
// This block of code will be run until count is 10.
while (count < 10) {
    light = ON;
    delay(1000);
    light = OFF;
    delay(1000);

    count = count + 1; // This increases count by 1 each loop.
}
```

The above code block runs only *while* conditions are true. We can do the exact same thing with a *for* loop too, and whether to use a *while* or a *for* loop depends on context.

```
// This block of code will be run until count is 10.
for (count = 0; count < 10; count++) {
    light = ON;
    delay(1000);
    light = OFF;
    delay(1000);
}
```

Even though the *for* loop might look more complicated, once you understand the syntax you can write cleaner code within the loop itself. That doesn't mean a *for* loop is better; each loop type has its own strengths and use-cases, and you will discover these yourself as you venture deeper into Arduino and programming.

Comments

```
// Two forward-slashes turn the rest of the line into a comment.  
// Comments are not part of the program and are purely used by humans  
// to annotate or describe their code.
```

```
/*  
If we want to comment an entire section,  
we place our comments between  
a slash/star-star/slash pair.  
This turns everything between them into comments.
```

Comments can also be used to disable lines of code without deleting them. For example if we want to test that our new button works without it driving our motor, we can temporarily disable the line where it gets activated.

```
*/  
  
void loop() {  
  if (button == ON) {  
    //driveMotor();  
    ledOn();  
  } else {  
    //stopMotor();  
    ledOff();  
  }  
}
```

The Structure of an Arduino Sketch

An Arduino program is called a **sketch**. The basic structure of a sketch consists of two main functions: **setup()** and **loop()**. The **setup()** function runs once when the Arduino is powered on or reset. The **loop()** function then runs continuously.

Aside from the most common programming language features we went over above, there are some extra features that you will encounter when writing Arduino programs.

Includes and Defines

At the beginning of your sketch you may include libraries and define constants.

Includes

Libraries can be included in your sketch using **#include**. Libraries give your sketches additional functionality by providing functions that simplify complex tasks, while keeping their code separate from your sketch. One example is using the LiquidCrystal library to control an LCD screen. Typing this single line of code gives us access to LCD-related functions that enable us to quickly and easily control our screen, saving us time.

```
#include <LiquidCrystal.h>
```

However, every library will have unique syntax, and you must know how to use its functions. Luckily, most libraries are well documented and have full details on how to use the functions they contain. They are generally easy to use; after all they are designed to make things easier for us. See the following link for an example of library documentation:

<https://www.arduino.cc/reference/en/libraries/liquidcrystal/>

Defines

You can use **#define** to create constants/macros that make your code more readable. Like variables, they allow you to replace numbers with words, and have their values copied and pasted into your program wherever you use them. However, unlike variables, defines don't have data types, and they cannot be changed throughout the program. Here's how we might use one; to define the pin connected to an LED:

```
#define LED_PIN 13 // LED_PIN is a "macro" definition, and the number  
                  // "13" is copy-pasted wherever it is used.
```

Defining pins this way is good practice because if we decide to change which pin our LED is connected to later in development, the only thing we need to change is the number in our definition of LED_PIN. If we had used the number in every reference to our pin instead, then we would have to manually change every instance of the number throughout our program if we decided to change the pin.

Variables and Functions

Any variables and custom functions that will be used throughout the program are usually declared after your **#include** and **#define** sections at the top, and before **setup()**. This is not the only place we can declare variables or functions, but we need to make sure they are declared before we attempt to use them, so it's generally good practice to put them here.

The setup()

The **setup()** function is a special function that runs once at the beginning of your program. You use it to set up your pins as either input or output, initialise certain functionality like serial communication, and prepare other features of our Arduino like turning on a light to indicate it is running:

```
void setup() {  
  pinMode(LED_PIN, OUTPUT);    // Initialize the LED pin as an output.  
  digitalWrite(LED_PIN, HIGH); // Turn the LED on.  
  Serial.begin(9600);          // Start Serial port at 9600 baud.  
}
```

The loop()

The **loop()** function is another special function that runs over and over again until your Arduino is powered down. It is used to continuously perform the main logic/decision-making in your program. For example:

```
void loop() {  
  digitalWrite(LED_PIN, HIGH); // Turn the LED on  
  delay(1000);                 // Wait for a second  
  digitalWrite(LED_PIN, LOW);  // Turn the LED off  
  delay(1000);                 // Wait for a second  
}
```

Your sketch must have exactly one **setup()** and one **loop()** function.

Custom Functions

We can make our own functions that turn large amounts of code into smaller ones, hide away large sections of code, and allow us to reuse code without having to rewrite it. We've already seen several built-in functions, such as **digitalWrite()** and **delay()**, and under the hood these functions are doing a bit more than what we see on the surface. We could blink an LED like above but use a function instead of our main loop, so that if we wanted to blink the LED from elsewhere in our program as well, we could use one line of code instead of an extra four.

Let's alter the previous code block to see how we would do this:

```
void blink() {                // Create our function here  
  digitalWrite(LED_PIN, HIGH); // Turn the LED on  
  delay(1000);                // Wait for a second  
}
```

```
digitalWrite(LED_PIN, LOW); // Turn the LED off
delay(1000);
}

void loop() {
  blink(); // Here we use, or "call" our blink function
}
```

Now we have access to a function called **blink()** that we can use anywhere in our program to blink our LED.

Furthermore, and for readability's sake, we might want to hide all of our "backend" code down the bottom so that all we see are the variables, the setup, and the loop. This is perfectly valid and is a great way to make our code tidy. But as mentioned previously, if we attempt to call a function from within our loop and the function hasn't been declared yet, our program might break; it's trying to execute code that doesn't exist yet! We get around this by either defining what our function does *before* it's used, or *declaring* the function before it's used and *defining* what it does later. The latter of these is called a *forward declaration*. This is a more advanced approach - skip to page 17 if you're new to Arduino.

```
void myFunction() { // Declare and define the function at the same time.
  a = a + 1;        // This is the standard approach.
}

// Setup and loop at the bottom.
void setup() {

}

void loop() {
  myFunction(); // Call (use) the function.
}
```

Or

```
void myFunction(); // Declare the function (forward declaration).

void setup() {

}
```

```
void loop() {  
  myFunction();    // Call the function.  
}  
  
void myFunction() { // Define the function down where it's less visible.  
  a = a + 1;  
}
```

The Arduino IDE is generally smart and will “look ahead” for you to make sure functions are defined before they are needed. It does this during “compiling” (the process where the software converts our human-readable code to computer-readable 1’s and 0’s). This happens after we click upload and before the program gets uploaded to the Arduino. However, the Arduino IDE doesn’t always look ahead and this can’t necessarily be expected from other programming languages or other development environments. For this reason, it’s best practice to do it the way that will work most consistently across all programming setups.

As mentioned, this is getting into more advanced programming practices and there are a lot of fancy terms here, so don’t worry if this doesn’t make sense to you yet. This section is more about making our code look nice and less about making our code work, so it’s not essential. But do note that making our code work properly and making it look nice generally go hand-in-hand!

Communication Protocols

A communication protocol is a set of processes that one device must follow in order to talk to another. It's the rules of the language used, and the order in which certain actions take place. Let's demonstrate by comparing it to a human "communication protocol".

A generally accepted communication protocol among humans is as follows:

- When introduced to a stranger, provide a greeting in a known language, English, for example.
- Await a response.
- Once a response is received, determine whether the response was also English.
- Good, you both speak the same language. Provide the stranger with your name.
- Allow a moment for the stranger to provide their name.
- Shake hands and provide an obligatory "nice to meet you", regardless of true inner sentiments.
- You are now introduced and can communicate freely.

Electronic devices perform a similar sequence of steps, and these vary between protocols the same way they would across human cultures and languages.

The main communication protocols you will encounter in Arduino are **SPI** (Serial Peripheral Interface), **I²C** (Inter-Integrated Circuit; said eye-two-see, or sometimes eye-squared-see), and **serial/UART** (Universal Asynchronous Receive / Transmit)*. Let's look at these in more detail.

*UART is also commonly referred to as simply "serial", and while its meaning is generally understood among programmers, "serial" can technically refer to any communication protocol that transmits information *in series*. "Series" literally means one after another, so SPI, I²C and UART are all serial communication protocols, as they transfer information in bits, one after the other. However, for the purpose of this guide and unless indicated otherwise, "serial" will refer to UART.

SPI

SPI (Serial Peripheral Interface) is a synchronous (meaning it's coordinated by a clock signal) serial communication protocol used for short-distance communication. SPI devices communicate in full-duplex mode using a master-slave architecture with a single master. Multiple slave devices are supported through individual slave select (SS) connections.

Features:

- **Full Duplex:** Communication can occur simultaneously in both directions.
- **High Speed:** Faster data transfer rates compared to I²C.
- **Master-Slave:** One master device controls one or more slave devices.

- **Multiple Devices:** Multiple devices can be connected to the same data wire, also known as a *bus*.

Pins:

- MOSI (Master Out, Slave In)
- MISO (Master In, Slave Out)
- SCK (Serial Clock)
- SS or CS (Slave Select, or Chip Select), one per slave

SPI requires a minimum of 4-wires to operate, plus 1 for each additional slave. While this provides it with many of its strengths, it makes it a suboptimal choice if you are short on pins, or if you don't need fast transfer speeds or full-duplex communication.

I²C

I²C (Inter-Integrated Circuit) is also a synchronous serial communication protocol used for short-distance communication. Similar to SPI, I²C devices communicate using a master-slave architecture, where multiple slave devices can be connected to a master through a shared bus. However, because it uses a single bi-directional data bus instead of separate send and return buses, only one I²C device can communicate at a time. Also, instead of a Slave Select wire, the master identifies each slave using its unique name, or "address". An I²C device's address usually looks something like "0x3C", and will be indicated by the manufacturer.

Features:

- **Half Duplex:** Communication occurs in one direction at a time.
- **Addressing:** Each device on the bus has a unique address, allowing multiple devices to share the same bus.
- **Master-Slave:** One master device controls the communication with one or more slave devices.
- **Two Wires:** Requires only two lines for communication, simplifying wiring and reducing pin usage.

Pins:

- SDA: Data
- SCL: Clock

I²C is excellent for connecting multiple devices efficiently when speed is not a major concern.

Serial/UART

UART (Universal Asynchronous Receiver/Transmitter) is an asynchronous (no clock signal) serial communication protocol used for communication between two devices over short distances. Because there is no clock signal, each device must know how fast the other device intends to send information. This rate is known as the Baud Rate, and the programmer programs it into the device. The most common Baud Rates are 9600 and 115200.

Features:

- **Asynchronous:** No shared clock signal; synchronisation is achieved using a predeclared speed.
- **Simplex/Duplex:** Supports simplex (one-way), half-duplex (two-way but not simultaneously), and full-duplex (two-way simultaneous) communication modes.
- **Point-to-Point:** Typically used for communication between only two devices.
- **Two Wires:** Requires only two wires, making it simple to implement and use.

Pins:

- Rx: Receive
- Tx: Transmit

UART is a simple communication protocol useful for basic data transfer between two devices.

It's very common practice to use a combination of these communication protocols to suit your needs. Many "Arduino compatible" devices will be pre-programmed to communicate using one or more protocols and you will have to plan your connections accordingly. The Arduino Nano has pins dedicated to each of these communication protocols; see if you can identify where you would plug in a device for each protocol.

Electronics Fundamentals

What is Electricity?

Electricity is the flow of electrical charge, typically carried by electrons through a conductor such as metal. It's one of the main forms of energy that we use for transmission because it is easily converted into other forms, such as kinetic energy, and with today's battery technologies, it is also easily stored as chemical energy.

Electricity can be thought of like water flowing through pipes. For water to flow (current), we need the following:

- Something to create pressure, like a pump.
- A complete circuit of pipes.
- Pipes wide enough to carry the expected amount of water.

Unlike water, with electronics we also need something that resists the flow of electrons and takes away some of its energy.

Let's talk about these in more detail.

Voltage (V)

Voltage, measured in Volts (V), is the electrical equivalent of water pressure. The battery or power supply is the pump that provides pressure to the circuit, producing a flow of *current*. Current flows from a state of high voltage (5V in the case of Arduino Nano) to low voltage (0V, aka ground (GND)).

Current (I)

Current is the electrical equivalent of the quantity of water flowing through the pipes, and is measured in Amperes, or Amps (A). For current to flow, our wire (pipes) must be sufficiently large, and there must be enough pressure in the system to overcome any resistance. It is the flow of water in pipes that provides the energy required to perform work, like turning a water turbine, or in an electrical circuit, turning a motor.

Resistance (R)

Resistance can be thought of as anything that impedes/resists the flow of water - sand, sediment, or other obstructions in our pipes, and is measured in Ohms (Ω). In an electrical circuit, it's our components, like motors, sensors, and lights, that provide resistance to the system. Some components, like motors, provide greater resistance to the circuit than, for

example, an LED, and so generally a greater voltage is required for them to operate. If no current flows, our components don't work.

We must provide enough pressure (voltage) to overcome the resistance for current to flow, and our components to function.

On the other hand, if there is *no* resistance in our system, our water gets to flow very quickly which burns out the pump and makes the pipes hot. This is called a "short circuit".

Ohm's Law

The relationship between these three principles is described by Ohm's Law:

$$V = IR$$

While this equation calculates the voltage from a known current and resistance, in most real-world scenarios we already know the voltage; usually because it's written on the battery! However, we often need to know the expected current before we plug things in. We first need to ensure our power source is capable of delivering the required current, and then we might also want to get an idea of how much energy our circuit uses. This is important for estimating runtime if we are using a finite power source, like a battery, or when plugging devices into pins (like on your Nano) that are only capable of delivering a small amount of current.

So, we can rearrange the equation to calculate for current instead:

$$I = \frac{V}{R}$$

But how do we know what R is? Well there are many ways to find out the resistance of our components, one of them being the **datasheet** - a full specification of the component's electrical characteristics, provided by the manufacturer. You can find these by Google searching the component's model number followed by the word "datasheet", or sometimes "spec sheet".

If we know the current in our system is within an acceptable range but we want to know precisely how much is being drawn, perhaps to calculate expected battery life, we can simply measure it with an Ammeter or multimeter.

We can also measure resistance with a multimeter, but note that this won't work on all components, for reasons that are beyond the scope of this guide.

To summarise; we usually choose the voltage we apply to the circuit, and our resistance is usually determined by our components. Which all leaves current as the "floating" variable;

the number that will change as our energy use changes - going up hills in an EV, turning on an LED, activating the shutter on a digital camera.

This is an oversimplification of a complex relationship between voltage, current and resistance, but it is a useful way to conceptualise the basics.

Alternating Current and Direct Current

For one reason or another, most people would have heard of AC and DC. These are two important concepts in electricity which we will touch on briefly.

Alternating current (AC) is where the voltage in a circuit “alternates” from positive to negative and back. And what happens to the current in a circuit when we provide a negative voltage? Well if you guessed it flows in the opposite direction, you’d be correct! The electrons in an AC circuit flow forward, stop, then get pulled the other way when the voltage changes polarity. As mentioned before, it’s the movement of electrons (current) that provides the energy that we use in electrical and electronic systems.

*Fun Fact! “**Electrical**” usually refers to systems that **transfer energy** (motors, transformers), while “**electronic**” refers to systems that **process information** (laptops, phones).*

AC is most commonly associated with high voltage applications and power transmission, i.e. electrical systems. Some places you will find AC power are:

- Power lines, due to greater long-distance efficiency compared to DC
- Large household appliances like washing machines, dishwashers, air conditioners
- Industrial machinery
- Household wall sockets, though this doesn’t mean that every device that plugs into a wall socket is an AC device

In DC, the voltage remains positive, and current flows in only one direction. DC is more often associated with processing and transmission of data or information, i.e. electronic systems, but can also be used to power things like the DC motors used in EVs.

Some places you would find DC power are:

- Computers - laptops, smartphones, gaming consoles etc.
- Solar panels
- DC motors
- Batteries

While the wall sockets in your house provide AC power, most electronic devices have a power supply that converts AC power from the wall to the DC power they use to operate.

This is the chunky black box that lives somewhere along the power cable of a laptop, or the main part of a USB wall charger. Other devices like stereo amplifiers or desktop computers may have these built-in.

Within EVelocity, you are unlikely to encounter any AC systems aside from the wall sockets you plug your battery charger into.

Analog vs. Digital Signals

“Signal” is the term we use to describe electrical information that we measure or control. It’s through signals that we can use our Arduino to interact with the real world. For an Arduino, these signals are the things that are measured and set by its pins. There are two main types of signals which are essential to understand: analog and digital.

Analog Signals

Analog signals are *continuous* signals that *vary over time* and can take any value within a given range. These signals are typically used to represent real-world phenomena such as temperature, light intensity, and sound. In electronics, analog signals are often used with sensors that measure these physical quantities.

Examples of Analog Signals:

- Temperature readings from a thermistor
- Light intensity from a photoresistor
- Sound waves from a microphone

Digital Signals

Digital signals, on the other hand, are *discrete* signals that have two states: HIGH (1) and LOW (0). These signals are used in digital electronics to represent binary data (0’s and 1’s). Digital signals are less susceptible to noise and interference compared to analog signals, making them ideal for communication.

Examples of Digital Signals:

- Turning an LED on or off
- Reading the state of a button (pressed or not pressed)
- Communicating with other digital devices using protocols like I²C or SPI

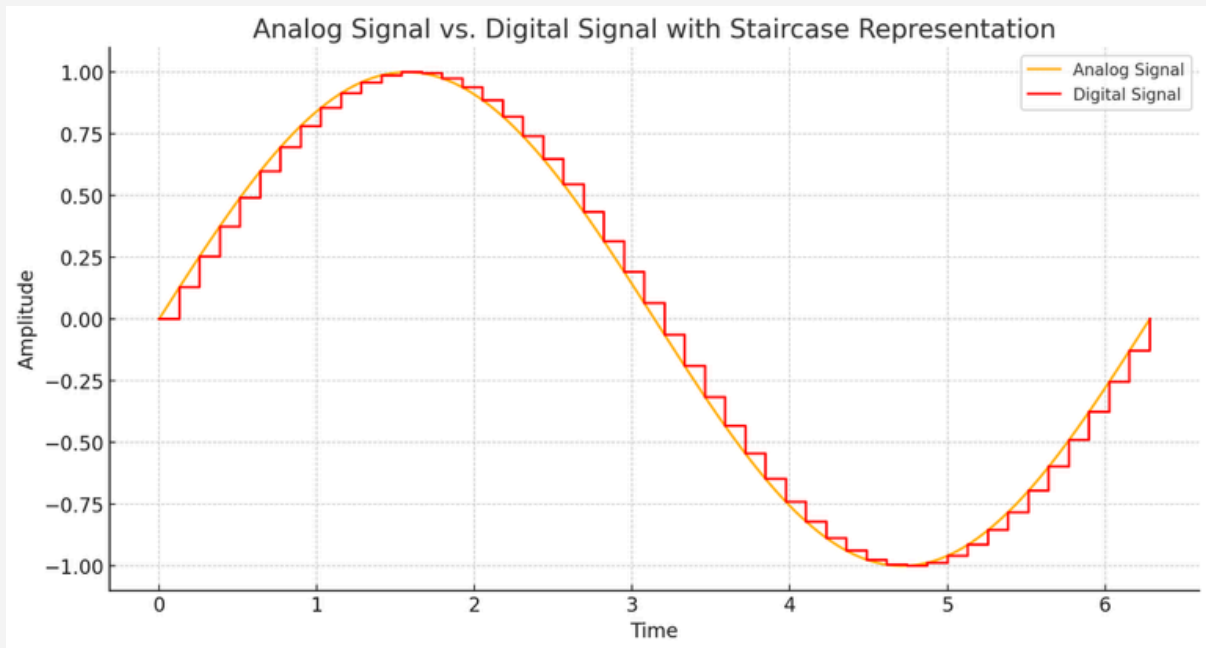
Why They Differ and Their Uses

- **Accuracy and Precision:** Analog signals can represent a wide range of values, providing more detailed information about a physical quantity. Digital signals, being binary, can only represent two states but are more precise in those states.
- **Noise and Interference:** Analog signals are more prone to noise and interference, which can affect the accuracy of the measurement. Digital signals are less affected by noise, making them more reliable for digital communication.
- **Applications:** Analog signals are used in applications where a continuous range of values is needed, such as audio processing and sensor measurements. Digital signals are used in applications requiring binary data, such as computer systems and digital communication.

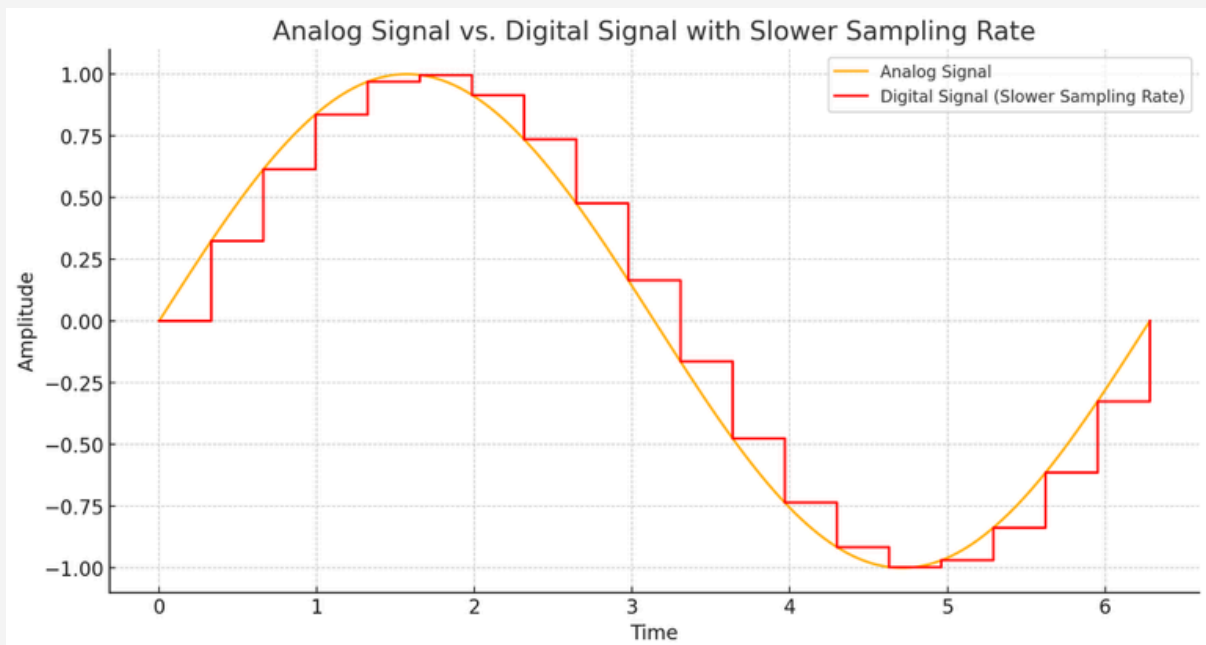
Computers and Analog Signals

Computers are the perfect example of a digital system, and this includes Arduinos. All pins on the Arduino Nano are naturally digital, however, some pins are equipped with an analog-to-digital converter (ADC), that allows analog signals to be read by the Arduino. Under the hood, the analog signals get converted to a digital representation of that signal for the Arduino to use. The Arduino Nano has 10-bit ADCs on the pins labelled A0, A1 etc, and we call these “analog pins” for simplicity.

But how do we represent an analog signal digitally? The graph below shows an example. Each “stair” is a digital value that represents the measured voltage at that given time. These are called “samples”, and are taken at very fast intervals. The rate at which samples are taken is called the Sample Rate, usually measured in thousands of samples per second, or kilohertz (KHz). This is how things like sound - a fundamentally analog signal, can be stored and processed digitally.



Things get less and less accurate if we don't use a high enough sampling rate. The graph below shows a slower sampling rate, and if you can imagine it slowing down further, our digital representation of the signal becomes a few squares that hardly represent the original signal at all. In the opposite direction, the higher the sampling rate, the closer and closer we get to an accurate representation of the original signal.



Our Arduino Nano has a total of 7 analog pins and 11 general-purpose digital pins. Handily, the analog pins can also be used as digital pins, giving us a lot more options if we run short. However, this does *not* apply the other way around, as the digital pins aren't equipped with ADCs.

Electronic Components

There are a variety of different components used in electrical and electronic circuits. To keep things relevant and because of the sheer number of different component types, we'll focus on only the most fundamental of these.

Resistors

A resistor's primary purpose is to provide resistance - like a lump of sand in our pipes. However, they do have some secondary characteristics, such as producing heat and/or light, that expand their variety of use-cases. Some of these include filament light bulbs and heaters.

We might use resistors to slow the flow of current to safe levels for low-powered components. Components like LEDs may have such low resistance themselves that if they were to be connected to a 5V power source, too much current would flow and they would fry. We might also use one in a circuit with a button, so that we don't draw unnecessary current when the button is pressed and closes the circuit.

Our Arduino Nano has built-in **pull-up resistors**, which we won't go into in this guidebook. Just know that you will come across these often, and that they're used to set a default HIGH state on our pins where they would otherwise "float".

Capacitors

Another fundamental component is the capacitor, which can be thought of as an electrical reservoir - a pool that fills with electrons once they start flowing. This pool can be drawn from if ever the power source fluctuates or drops, and they make sure there is enough water in our pipes so that our components continue to operate.

This might sound like they do what a battery does - store energy to be used later, right? While it's true they both store energy for later, capacitors hold far less energy, and they don't hold it for long once the main power source drops off (think nanoseconds or milliseconds). They should instead be thought of as a small emergency water tank that gets used during drought. In the real world, capacitors are used to smooth out or slow down signals or power sources that might otherwise change rapidly.

Some practical uses are:

- Equalising audio signals to get rid of certain frequencies by smoothing out rapid voltage fluctuations
- Camera flashes - several capacitors are charged/filled and then triggered to release their charge all at once in a blinding flash

Transistors

The foundational component of all computing, transistors are an electronic switch, one that is turned off and on using an electrical signal. We use an electrical signal to either allow or disallow current to flow through a circuit, creating a binary state - on/off, 1/0 - that can then be used to encode information. We can also use them to amplify a low-powered signal by making them open and close a higher-powered circuit in an identical fashion.

Diodes

These allow current to flow in only one direction, and are the main component used in "rectification", the process of converting AC to DC.

Some diodes emit light when a current is passed through them. These are known as Light Emitting Diodes, or LEDs.

Electronic Structures

These are specific layouts of fundamental components that serve unique purposes. Because of the vastness of the electronics world, we will only touch on the structure most relevant to the EVelocity Arduino Workshop, the voltage divider.

Voltage Dividers

What is a Voltage Divider?

A voltage divider is a simple circuit that reduces voltage. It is made using two resistors. When voltage is applied across these resistors, the output voltage is a fraction of the input voltage. The formula for a voltage divider is:

$$V_{out} = V_{in} \frac{R_2}{R_1 + R_2}$$

Where:

- V_{out} is the output voltage.
- V_{in} is the input voltage.
- R_1 and R_2 are the resistances of the two resistors.

Why are these useful?

Most of the time, when we power an electronic circuit we want to provide a little bit more voltage than is necessary at the power source, then we regulate* the voltage down to a

level more appropriate for our system. This provides a “clean” power source, which means our power is free of fluctuations and interference. This makes our devices run better, last longer, and be free of errors. Our Arduino Nano’s operate at 5V, and anything higher than this on any of its pins could be disastrous.

To find out how much energy our battery has left, we need to measure its voltage, and we would do this using one of our analog pins. But then how can our Arduino monitor the voltage of its own power source if it’s too high for its own pins?

Voltage dividers allow us to literally divide the voltage into a smaller fraction - this could be a tenth, half, a third etc, so that the voltage coming in is at an acceptable level for our delicate Arduino pins at a known fraction of the source voltage. Once our Arduino has taken its measurements, it’s simply a matter of doing the math to recalculate the voltage at source.

Now we can program low-battery warnings/indicators, or perform failsafe procedures; ensuring we stop writing to an SD card before our battery dies, or moving a machine into a safe position before it runs out of battery/fuel.

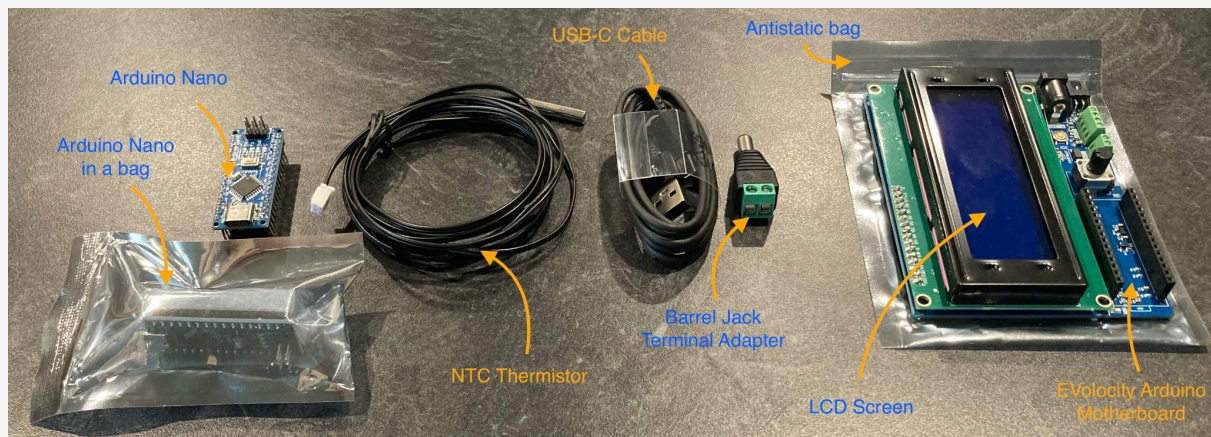
*Our Arduino Nano has a built-in voltage regulator that accepts between 7V and 12V and reduces it to a consistent 5V. This onboard regulator is connected to the pin labelled “Vin”, and we shouldn’t provide power directly to any of the other pins. If we were to power the Arduino’s 5V pin directly with anything greater than 5V it would die - the 5V pin should only be used as an output to power other devices or components.

Getting Started

With all the basics covered, let's dive into the tutorial with some hands-on exercises to consolidate our knowledge.

Check that you have these parts in your kit:

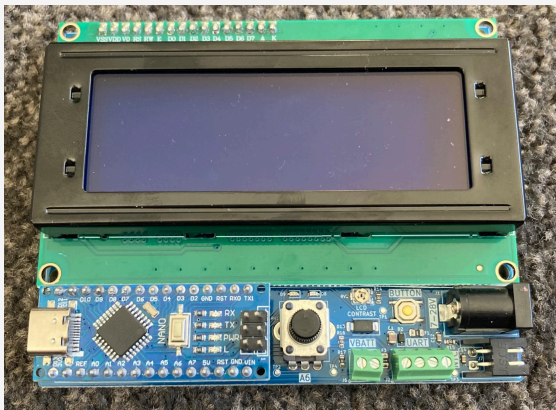
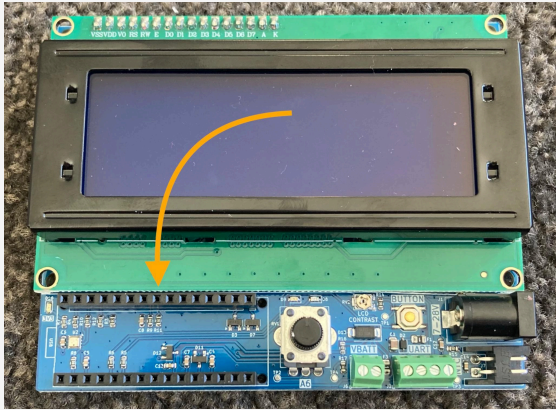
- Arduino Nano
- EVolocity Arduino Motherboard (AMB)
- Barrel jack terminal adapter
- USB-A to USB-C cable
- NTC thermistor
- 20x4 LCD screen
- Antistatic bag



EVolocity Arduino Motherboard (AMB)

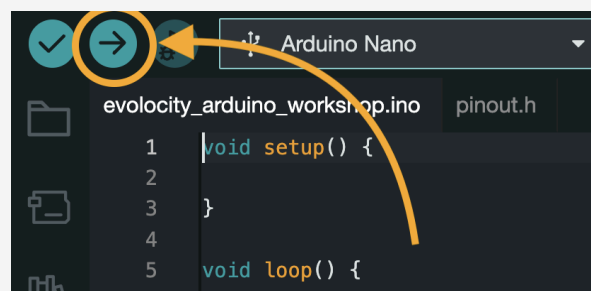
The AMB is the main board in the kit. It lets us read our battery voltage, monitor our battery temperature, and display this information to the driver.

Plug your Arduino Nano into the AMB with the USB-C port pointing left.



The Arduino IDE

1. Download the Arduino IDE from the [Arduino website](https://www.arduino.cc/en/software). We will be using version 2.3.2 throughout this guide, but newer versions should also work.
2. Connect the Arduino board to your computer.
3. Perform a test upload using the example sketch "Blink":
 - Go to File > Examples > Basics > Blink
 - Use the **Select Board** dropdown to select the Arduino Nano
 - Click the **Upload** button



- After a successful upload, the LED on the Arduino Nano should be blinking once per second

This is simply a test to make sure our Arduino is working and we can upload to it. If you're having trouble with this step, contact your EVelocity coordinator.

Exercises & Examples

Here is where we will start programming our Arduino board to interact with the world. Throughout this section, practical examples will be followed by the explanation of concepts and functions so you can use them in your own programs.

*Make sure you have first completed the **Getting Started** section above.*

You can download example sketches for each section here:

<https://evolocity.co.nz/introduction-to-arduino/>

1 Digital I/O

1.1 Blinking an LED

What You'll Learn:

- How to control an LED using Arduino
- How to change pin states with `digitalWrite()`
- How to use `delay()`

Setup

1. Unplug the USB cable from your Arduino if you have already plugged it in.
2. Plug your Arduino Nano into the motherboard with the USB port facing **left**.
3. Press down firmly until the legs disappear into the headers. You may need to rock it side to side.
4. Plug the USB cable into the Arduino.
5. Open a new sketch by going to File > New Sketch.
6. Paste the following code into an empty sketch and click upload.

Code

```
#define LED_Y A0
#define LED_R A1

void setup() {
  pinMode(LED_Y, OUTPUT);
  pinMode(LED_R, OUTPUT);
}
```

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

Watch what happens: LEDs D9 and D8 will begin to blink!

Note: D8 and D9 are the names printed next to the LEDs on the motherboard (D = Diode). But in your code, you've used A0 and A1 - these are the actual Arduino pins they're connected to.

Try This

- Make the LEDs blink faster or slower
- Make both LEDs blink at the same time
- Leave one LED on constantly and blink the other

*Each time you tweak your code, remember to click **Upload** again.*

How it Works

Digital signals are those that can have only one of two states; HIGH/LOW, 1/0, TRUE/FALSE, ON/OFF. We use this to control devices like LEDs, motors, and more. The EVoloccity Arduino Motherboard features two LEDs of its own that can be programmed, and these are connected to pins A0 and A1 on the Arduino.

*While pins A0 and A1 are actually analog pins, they can also be used as digital pins. The reverse is **not** true - only analog pins can read an analog signal.*

Initialisation

Before using a pin, we tell the Arduino what it's for:

```
void setup() {  
  pinMode(A0, OUTPUT); // Set pin A0 to be an output  
  pinMode(A1, OUTPUT); // Set pin A1 to be an output  
}
```

Looping the Blink

Writing the pins HIGH tells the Arduino to apply a high voltage on the indicated pin. The `loop()` function keeps running forever, and this makes the LED blink continuously:

```
void loop() {  
  digitalWrite(A0, HIGH); // LED on  
  digitalWrite(A1, LOW);  // LED off  
  delay(500);             // Wait 0.5 seconds  
  digitalWrite(A0, LOW);  // LED off  
  digitalWrite(A1, HIGH); // LED on  
  delay(500);             // Wait again  
}
```

1.2 Reading a Button

What You'll Learn:

- How to read a button using `digitalRead()`
- How to use a variable to track state

Setup

1. If you haven't connected your Arduino according to the instructions in 1.1, do so now.
2. Open a new sketch by going to File > New Sketch.
3. Paste the following code into an empty sketch and click upload.

Save your old sketch if you would like to keep it, but remember all example sketches are available on the [EVolocity website](#).

Code

```
#define LED_ONBOARD 13  
#define BTN 6  
  
void setup() {  
  pinMode(LED_ONBOARD, OUTPUT);  
  pinMode(BTN, INPUT);  
}
```

```
}

void loop() {
  int buttonState = digitalRead(BTN);

  if (buttonState == HIGH) {
    digitalWrite(LED_ONBOARD, HIGH);
  } else {
    digitalWrite(LED_ONBOARD, LOW);
  }
}
```

Watch what happens: The Arduino's onboard LED will light up when the button is pressed!

Try This

- Make the LED turn OFF when the button is pressed, instead of ON
- Make LEDs D8 and D9 turn on as well
- Add code so that the LEDs will stay on for one second after the button has been pressed

How it Works

Digital inputs, like outputs, can be either HIGH or LOW, and we use this to check the state of electronic components like buttons or switches.

Initialisation

In the `setup()` function, we configure the pin as an **input**:

```
void setup() {
  pinMode(6, INPUT);
}
```

Reading the State

To read a digital input, we use the `digitalRead()` function. This checks the voltage on a specific pin.

```
int buttonState = digitalRead(6); // Read the state of pin 6.
```

If there is a HIGH voltage on pin 6, *buttonState* will equal 1.

If there is a LOW voltage on pin 6, *buttonState* will equal 0.

*Note: if we use **digitalRead()** without a variable, the result just gets lost. You must save it to a variable to be able to use it later.*

2 The Serial Monitor

The Serial Monitor in the Arduino IDE is a text-based window that lets our computer and Arduino talk to each other. This is an important debugging tool as it allows you to interact with your code and diagnose failure points.

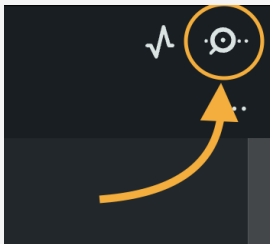
2.1 Sending a Serial Message

What You'll Learn:

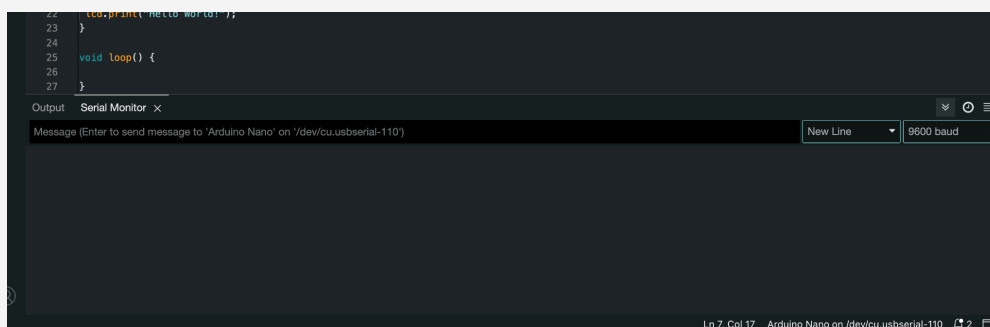
- How to send a message from your Arduino to your computer

Setup

1. Open a new sketch by going to File > New Sketch.
2. Paste the code below into an empty sketch and click upload.
3. Once uploaded, open the Serial Monitor by clicking on the magnifying glass icon in the top-right corner of the Arduino IDE, or by pressing ctrl+shift+M.



This should open the Serial Monitor:



4. Make sure that the baud rate matches the number in **Serial.begin()**, in this case, 9600.

Code

```
void setup() {  
  Serial.begin(9600);  
}  
  
void loop() {  
  Serial.println("Hello, world!");  
  delay(1000);  
}
```

Watch what happens: The Arduino should say "Hello, world!" once per second.

Try This

- Make the Arduino say hi to you using your name
- Make it say hi only when you press the button - you'll need to take some code from previous exercises to do this!
- Make it tell you the state of the button

How it Works

The Arduino can talk to your computer using UART, also known as **Serial** (for more detail, see page 19 on [Serial/UART](#)).

Serial is asynchronous, meaning both sides must agree on how fast data will be sent. This is called the baud rate, and you must match it in the Serial Monitor for the messages to be readable.

Initialisation

Start the serial connection inside **setup()** and set the baud rate (in this case, 9600 baud, or bits per second):

```
void setup() {  
  Serial.begin(9600); // Start the Serial communication at 9600 baud  
}
```

Hello, World!

Use `Serial.println()` to print messages or values to your computer. This sends the text and adds a new line after each message:

```
Serial.println("Hello, world!");
```

If you want to keep everything on the same line, use `Serial.print()` instead:

```
Serial.print("Hello, world!");
```

You can also print the value of a variable by leaving out the quotation marks:

```
Serial.println(buttonState); // Print the value of the variable
```

Debugging with Serial

If your code isn't doing what you expect, try placing `Serial.println()` statements at different points in your sketch. This helps you trace what's happening step by step and pinpoint exactly where things are going wrong.

***Remember:** if your code doesn't compile or upload, read the error messages! Most small sketches should take no longer than 5-10 seconds to upload.*

3 Analog I/O

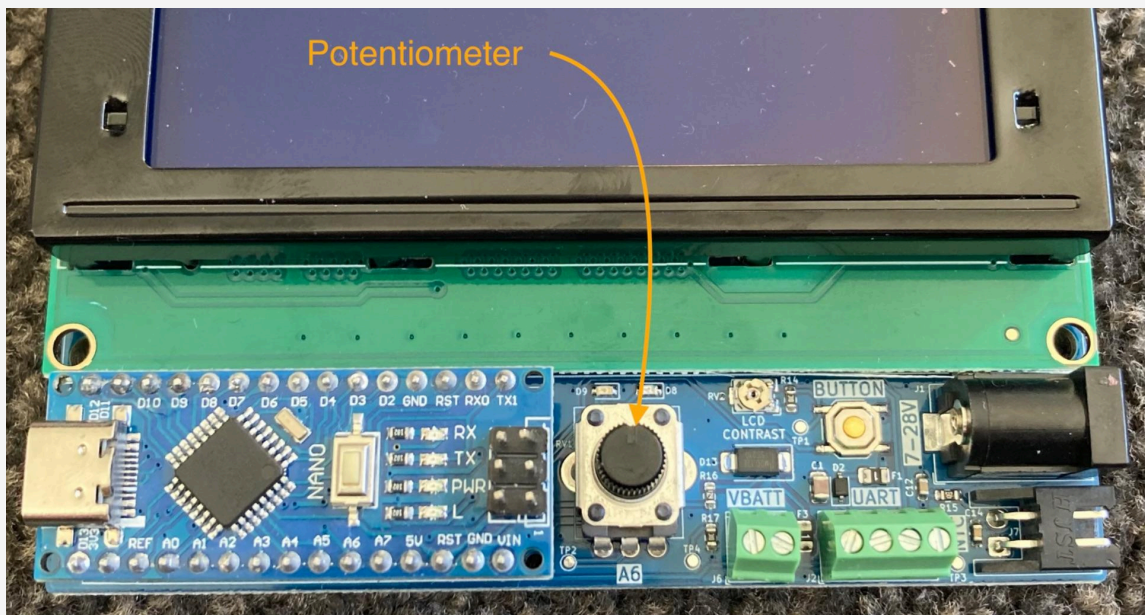
3.1 Reading the Dial

What You'll Learn:

- How to read and display an analog signal using `analogRead()`

Setup

1. Identify the potentiometer on your motherboard:



2. Open a new sketch by going to File > New Sketch.
3. Paste the code below into an empty sketch and click upload.
4. Once uploaded, open the Serial Monitor.

Code

```
#define DIAL A6

void setup() {
  Serial.begin(9600);
  pinMode(DIAL, INPUT);
}

void loop() {
  int dial_value = analogRead(DIAL);
  Serial.println(dial_value);
}
```

Watch what happens: You should see numbers being continuously printed that change when you turn the dial.

Try This

- Find and note the minimum, maximum, and middle values when you turn the dial

How it Works

An analog signal is one that can vary smoothly across a range, not just ON or OFF like a digital signal. This makes our analog pins perfect for reading sensors that output a range of values, like temperature sensors or potentiometers (dials).

Initialisation

Again, we initialise our pin within setup:

```
void setup() {
  pinMode(A6, INPUT);
}
```

Reading the Pin

To read an analog value from a pin, use the `analogRead()` function:

```
int dial_value = analogRead(A6); // Read the pin voltage.
```

This will return a number between 0 and 1023, but not the actual voltage. If we need to know the actual voltage, the section below details how to calculate it. Otherwise, skip ahead to Section 3.2.

Voltage to Numbers

Your Arduino has an **Analog-to-Digital Converter (ADC)** that turns voltages into numbers your code can work with. It has a 10-bit resolution, meaning it can represent voltages using numbers from 0 to 1023 *inclusive*.

- 1023 = 5V
- 0 = 0V

We can calculate everything in between using the following equation:

$$V = Value \times \frac{Reference}{Resolution}$$

Where:

- *Value* refers to our ADC value; the number returned by `analogRead()`
- *Reference* refers to the voltage of our system (5V for Nano)
- *Resolution* refers to the highest possible value our ADC can give (1023)

To demonstrate, let's plug in our numbers assuming we received a reading of 500.

$$V = 500 \times \frac{5V}{1023}$$

$$V = 2.44V$$

This is just below half, as expected.

Now let's see how we would do this conversion in a sketch:

```
float voltage = dial_value * (5.0 / 1023.0);
```

Because our expected voltage is a decimal, we need to remember to use a **float** data type. **Int** can only hold whole numbers, which is insufficient for this task.

What's a Bit? What's Resolution?

You might be wondering:

- What *is* a bit?
- Why do 10 bits give values up to 1023 specifically?
- What does resolution really mean?

These are all great questions, and if you'd like to know more, go and read the section on [Computers and Analog Signals](#) on page 24. But for now just remember:

- The analog pin gives a number between 0 and 1023, not the number of volts
 - If we need the actual voltage, we calculate it
-

3.2 Using a Potentiometer to Control LEDs

What You'll Learn:

- How to make decisions using conditional logic
- How to use the `if` statement

Setup

1. Open a new sketch by going to File > New Sketch.
2. Paste the code below into an empty sketch and click upload.

Code

```
#define LED_Y A0
#define LED_R A1
#define DIAL A6

int dial_value;

void setup() {
  pinMode(LED_Y, OUTPUT); // Initialise the yellow LED.
  pinMode(LED_R, OUTPUT); // Initialise the red LED.
  pinMode(DIAL, INPUT); // Initialise the potentiometer.
}

void loop() {
  dial_value = analogRead(DIAL); // Read the potentiometer voltage.
```

```
if(dial_value < 400) { // If the dial is pointing left...
    digitalWrite(LED_Y, HIGH); // Turn on the yellow LED.
    digitalWrite(LED_R, LOW); // And make sure the red LED is off.
} else if (dial_value < 600) { // If the dial is about midway...
    digitalWrite(LED_Y, LOW); // Make sure the yellow LED is off.
    digitalWrite(LED_R, LOW); // Make sure the red LED is also off.
} else {
    // If neither of the other cases are true...
    digitalWrite(LED_Y, LOW); // Make sure the yellow LED is off.
    digitalWrite(LED_R, HIGH); // Turn on the red LED.
}
}
```

Watch what happens: You should see the yellow LED turn on when you point the dial left, and the red one turn on when you point it right.

Try This

- Increasing the “dead zone” - the range in which both of the LEDs will turn off
- Decreasing the dead zone - how small can you get it and have it still work reliably?
- Changing the sketch so that in the dead zone, both LEDs turn on
- Making the opposite LED turn on, i.e. yellow turns on when you point the dial **right**

How it Works

A potentiometer is a simple electronic component that provides a resistance that varies based on the position of the dial. We can use this to raise or lower the voltage that we read on pin A6.

Initialisation

Initialise our pin within setup:

```
void setup() {
    pinMode(A6, INPUT);
}
```

Reading the Pin

Read the raw value:

```
int dial_value = analogRead(A6);
```

Now *dial_value* can be used alongside conditional logic to make decisions.

If-else

The **if** statement is used to make decisions in your code.

It checks whether a condition is true, and if it is, executes a block of code. You can also add an **else** statement, which is executed if the condition is false.

Here we check if the potentiometer value is less than 511 (about half of 1023):

```
if(dial_value < 511) { // If the dial is pointing left...
    digitalWrite(LED_Y, HIGH); // Turn on the left LED.
    digitalWrite(LED_R, LOW); // And make sure the right LED is off.
} else {
    digitalWrite(LED_Y, LOW); // Turn off the left LED.
    digitalWrite(LED_R, HIGH); // And make sure the right LED is on.
}
```

To simplify the concept, here we've used 511 as our mid-point, where in the previous example we created a "dead zone" - a range between 400 and 600 where both LEDs turned off. Both are correct but you'll find the dead-zone method works a little better.

If you have multiple different conditions to check, you can use **else if()**, as we did in creating our dead-zone, and provide another set of conditions before using your final **else** as a catch-all. You can also have *no* else if you want your program to do nothing if your conditions aren't met.

3.3 Measuring Battery Voltage

By now, you've learned many of the key skills needed to tackle a task like this, so let's treat it as a milestone challenge.

This activity will require you to build most of the sketch yourself, applying what you've learned about reading analog voltage and displaying information using the Serial Monitor.

If you get stuck or want to check your work, a complete example sketch is available for download on the [EVolocity website](#).

What You'll Learn:

- How to measure voltage using a voltage divider
- How to bring together skills like input reading, variable maths, and serial printing

What You'll Use:

- Pin and variable initialisation
- Serial communication
- Analog input
- Basic calculations

Setup

1. Open a new sketch by going to File > New Sketch.
2. Write and upload a program that:
 - Sets up an analog pin as an **input**
 - Use **A6** (the potentiometer) if no battery is connected
 - Use **A5** if reading from VBATT
 - Initialises Serial
 - Calculates the battery voltage from an analog reading (use the code below)
 - Prints the voltage value in the Serial Monitor

Code

Use the following code to calculate the voltage, adding it to where you think it fits best in your sketch:

```
float Vout = analogRead(A6) * (5.0 / 1023.0); // Change to A5 if needed.  
float Vin = Vout * (120.0 + 10.0) / 10.0;
```

Try This

- Create a function called **print_voltage()** to handle your serial output, and call that function from your **loop()**
- Create a low-battery warning

Not sure how to make a function? Revisit [Custom Functions](#) on page 14 for a quick refresh.

How it Works

The motherboard has a voltage divider circuit connected to the terminals labelled “VBATT”, which connects to pin A5 on the Arduino. This is how we monitor our EVelocity vehicle’s battery voltage, and it takes a maximum of 60V.

Recall from [Voltage Dividers](#) on page 27 that its purpose is to reduce a voltage to a fraction of the input voltage. Therefore, to determine the original voltage, we need to perform two calculations; one to calculate the voltage measured on the analog pin of the Arduino, and one to convert the measured voltage back into the original voltage. Our resistor values are $R_1 = 120,000\Omega$ and $R_2 = 10,000\Omega$. Let’s plan the code for our two equations.

Analog-to-voltage Conversion

$$V_{out} = Value \times \frac{Reference}{Resolution}$$

$$V_{out} = analogRead(A5) \times \frac{5V}{1023}$$

```
float Vout = analogRead(A5) * (5.0 / 1023.0);
// Vout is the actual voltage measured on pin A5.
```

Voltage Divider

Then we calculate the actual battery voltage *before* it was reduced by the voltage divider:

$$V_{out} = V_{in} \frac{R_2}{R_1 + R_2}$$

$$V_{out} = V_{in} \frac{10k\Omega}{120k\Omega + 10k\Omega}$$

We calculated V_{out} before, so we rearrange for V_{in} :

$$V_{in} = V_{out} \frac{120k\Omega + 10k\Omega}{10k\Omega}$$

```
float Vin = Vout * (120.0 + 10.0) / 10.0;
// Vin is the voltage at our VBATT terminal.
```

3.4 Measuring Battery Temperature

This task is another opportunity to apply what you’ve learned about reading analog signals, this time using a thermistor to measure temperature.

Your goal is to write a sketch that reads an analog value, converts it to a temperature, and displays it in the Serial Monitor.

Because this process is very similar to the battery voltage task in 3.3, you're encouraged to **reuse your existing code** and make adjustments, rather than starting from scratch.

What You'll Learn:

- How to measure temperature using a thermistor
- How to convert analog readings into real-world values

Setup

1. Connect the thermistor to the **JST XH** connector on the motherboard.



2. Copy your entire sketch from 3.3.
3. Open a new sketch by going to File > New Sketch.
4. Paste your code into the new sketch, replacing any existing **setup()** and **loop()** functions (remember: you can only have one of each).
5. Update your code and upload:
 - Change the input pin to **A7** (if you've used **#define**, this should be easy)
 - Replace the voltage divider calculation with the thermistor calculations (use the code below)

Code

```
int analogReading = 1023 - analogRead(A7);  
float voltage = analogReading * (5.0 / 1023.0);  
float resistance = 5100 * ((5.0 / voltage) - 1);  
float temp = 1 / ((1 / 298.15) + (1 / 3950.0) * log(resistance / 10000)) - 273.15;
```

Try This

- Hold the thermistor in your hand - does it heat up?
- Move the math into a function called `calculate_temperature()`
- Print the result with another function called `print_temperature()`
- Create a high-temp warning at 25°C

How it Works

A thermistor is a type of resistor whose resistance changes with temperature - the name is a combination of “thermal” and “resistor”.

The thermistor used in your Arduino kit is an **NTC** (Negative Temperature Coefficient) thermistor, meaning its resistance decreases as the temperature increases. This results in a higher voltage on pin A7 as things heat up.

The Equations

Let's break down what each line is doing. The motherboard's circuit inverts the signal, so first we subtract the reading from 1023 to flip it right-way-up:

```
int analogReading = 1023 - analogRead(A7);
```

The next line is a standard analog-to-voltage conversion, the same as we used for battery voltage.

```
float voltage = analogReading * (5.0 / 1023.0);
```

This calculates the thermistor's resistance using the voltage divider formula.

```
float resistance = 5100 * ((5.0 / voltage) - 1);
```

Finally we have the **Beta parameter equation**, which converts thermistor resistance to a temperature in Celsius. It's a standard equation for NTC thermistors and can be found online.

```
float temp = 1 / ((1 / 298.15) + (1 / 3950.0) * log(resistance / 10000))  
- 273.15;
```

- 298.15 is room temperature in Kelvin
- 3950.0 is the thermistor's Beta value
- 10,000 is the thermistor's resistance at 25°C

4 Communication

4.1 Communicating with an LCD

What You'll Learn:

- How to display information on an LCD screen
- How to use libraries

Setup

1. Open a new sketch by going to File > New Sketch.
2. Paste the code below into an empty sketch and click upload.

Code

```
#include <LiquidCrystal.h>

#define LCD_RS 9
#define LCD_E 8
#define LCD_D4 5
#define LCD_D5 4
#define LCD_D6 3
#define LCD_D7 2
#define LCD_BACKLIGHT 7
#define LCD_WIDTH 20
#define LCD_HEIGHT 4

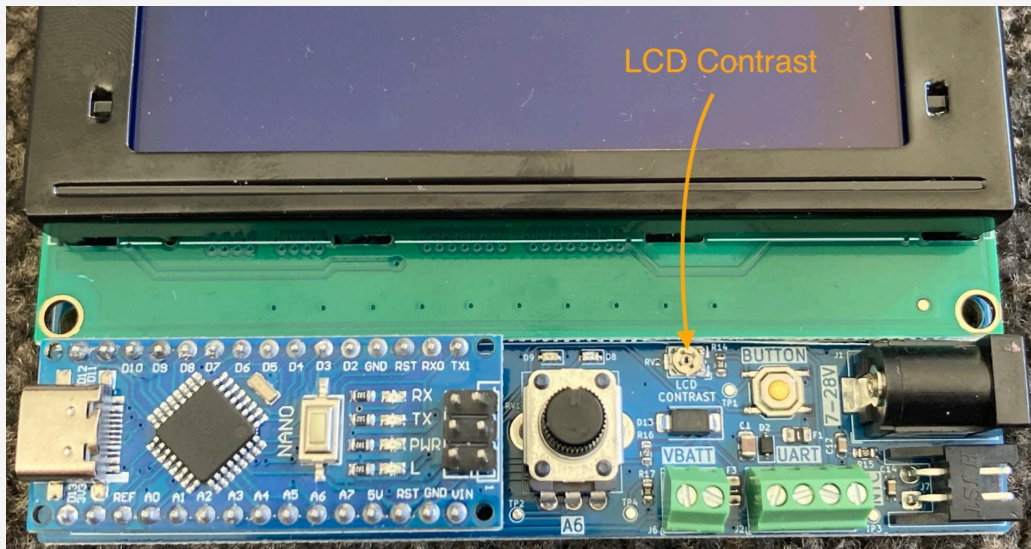
LiquidCrystal lcd(LCD_RS, LCD_E, LCD_D4, LCD_D5, LCD_D6, LCD_D7);

void setup() {
  pinMode(LCD_BACKLIGHT, OUTPUT);
  digitalWrite(LCD_BACKLIGHT, HIGH);
  lcd.begin(LCD_WIDTH, LCD_HEIGHT);

  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Hello, world!");
}

void loop() {
}
```

Watch what happens: You should see “Hello, world!” printed on your LCD. If you don’t, try turning the LCD contrast screw.



Try This

- Turn the backlight off
- Display a measurement on the LCD from one of the previous exercises
- Set up data labels for your measurement - “Voltage: ”, “Temp: ” etc.
- Display two sets of data, both labelled and on different lines

You may need to open up the LiquidCrystal library documentation to see what functions you have available:

<https://www.arduino.cc/reference/en/libraries/liquidcrystal/>

⚙️ How it Works

Devices such as LCDs and digital humidity sensors generally have their own microcontrollers, and we must program our Arduino to talk to it.

But how do we know what language the other device speaks?

Until now, we have done all of the programming ourselves - every line of code that has been executed is one that we have written and one that we can see. Now, however, we need to interface with a device that has been designed and programmed by someone else.

This is where we use **libraries**.

Under the Hood

Libraries give us access to functions that are specific to certain devices, and are usually written by the device manufacturer. These functions hide away all of the inner workings of, for example, delivering a message to another device, and provide us with a single line of code that does it for us. See the [Includes](#) section for more on Libraries.

Initialisation

To use a library, we need to use an **#include** at the top of our sketch:

```
#include <LiquidCrystal.h>
```

This is the library used to talk to the LCD screen on our motherboard. The library handles low-level communication for us.

Sometimes we also need to initialise the device. Your library documentation will tell you if this is required. LiquidCrystal requires the following line before **setup()** to tell our Arduino which pins the LCD is connected to:

```
LiquidCrystal lcd(rs, r, D4, D5, D6, D7);
```

The letters should either be defined beforehand using **#define**, or replaced with your actual Arduino pin numbers.

Library Functions

To clear the screen before we write to it, we use the **lcd.clear()** function:

```
lcd.clear();
```

Then to print to the screen, we use the **lcd.print()** function:

```
lcd.print("Hello, world!");
```

These are two of several important functions that we need to use to have a working LCD, and the rest are detailed in the library documentation found in the link above.

The best way to learn a library is to use the built-in examples. You can access these in the Arduino IDE by going to File > Examples, and then finding your specific library.

The Arduino IDE has libraries for common components built in, but you can download additional libraries using the Library Manager. More on this in the next section.

4.2 Measuring Data from the BME280 Sensor

What You'll Learn:

- How to take readings from a digital sensor
- How to download libraries

Setup

1. Open a new sketch by going to File > New Sketch.
2. Click the book icon on the left-hand side



3. Search for "Adafruit BME280". We have used **version 2.3.0** in this guide, so to avoid compatibility issues please use the same version. Feel free to use a newer version in your own time.
4. Click **Install**.
5. Arduino IDE may ask you to install dependencies. Click **Install All**.
A dependency is a library that the library itself uses, and if we don't also install these our code won't work.
6. Once complete, click the book icon again to close the side-panel and reclaim our screen real-estate.
7. Paste the code below into an empty sketch and click upload.

Code

```
#include <Adafruit_BME280.h>

#define BME_SCK 13
#define BME_MISO 12
#define BME_MOSI 11
#define BME_CS 10

Adafruit_BME280 bme(BME_CS);

void setup() {
  bme.begin();
  Serial.begin(9600);
}

void loop() {
  float temperature = bme.readTemperature();
  Serial.print("Motherboard Temp: ");
  Serial.print(temperature);
  Serial.println(" C");
  delay(1000);
}
```

Watch what happens: You should see the motherboard temperature printed on your LCD.

Try This

Open up the BME280 library docs to see how to use the available functions (note that we are using SPI to connect):

<https://learn.adafruit.com/adafruit-bme280-humidity-barometric-pressure-temperature-sensor-breakout/arduino-test>

- Read and display humidity and pressure
- Add code to read temperature from your thermistor and compare temperatures

How it Works

The BME280 is a combined digital humidity, pressure, and temperature sensor, designed for mobile applications where size and low power consumption are key design constraints.

It's labelled "U2" on the motherboard and lives underneath the USB port of the Arduino; see if you can find it!

Features:

- Temperature measurement: Accurate measurement of ambient temperature.
- Humidity measurement: Provides precise humidity data.
- Pressure measurement: Can measure barometric pressure, useful for altitude measurements.

It communicates with the Arduino via [SPI](#) (see page 16 for more information), but can also be connected via I²C.

5 Putting It All Together

Let's apply all we have learned - write a sketch that does it all.

Your sketch should:

- Measure battery voltage
- Measure battery temperature
- Measure ambient (BME280) temperature
- Display it all on-screen

If you still have time, you might also like to:

- Use an LED to indicate low battery voltage
- Use an LED to indicate dangerous battery temperatures
- Program the button and the dial to do something - we'll leave it up to you to decide what!

If you get stuck, remember you can download sketches for your board from the [EVolocity website](#).

Conclusion

Congratulations - you've made it to the end of a long and potentially challenging introduction to Arduino and electronics! Now you should feel confident in:

- Understanding the structure of an Arduino sketch, including **setup()** and **loop()**.
- Using digital and analog inputs and outputs.
- Working with different types of signals, such as analog and digital.
- Connecting and programming sensors and components on the EVelocity Arduino Motherboard, including LEDs, buttons, a thermistor, a potentiometer, a voltage divider, an LCD screen, and the BME280 sensor.
- Downloading and using libraries.
- Using the Serial Monitor for debugging and interaction.

As well as this, you now have all the required code to operate each of the motherboard components, and you can dismantle/reconstruct/combine these in any way you like.

All that's left for you to do is design and construct an enclosure to house your Arduino system, and to connect your batteries and power supply.

An important feature to note is the port that says UART, next to VBATT on the motherboard - this has an exciting purpose and will be detailed at a future date.

We can't wait to see how our hardware gets utilised. Good luck, and see you on the track!

Glossary of Terms

- Microcontroller: A small computer on a single integrated circuit.
- Analog Signal: A continuous signal that varies over time.
- Digital Signal: A discrete signal with two states: HIGH (1) and LOW (0).
- ADC (Analog-to-Digital Converter): Converts an analog signal to a digital value.
- PWM (Pulse Width Modulation): A technique for getting analog results with digital means.
- Baud Rate: The speed of communication over a data channel, often used in serial communication.
- I²C (Inter-Integrated Circuit): A communication protocol used to connect low-speed devices like microcontrollers, sensors, and other peripherals.
- SPI (Serial Peripheral Interface): A communication protocol used for short-distance communication, primarily in embedded systems.
- LCD (Liquid Crystal Display): A type of flat-panel display technology commonly used in screens.
- Library: A collection of functions and precompiled routines that a program can use.
- Serial Monitor: A feature in the Arduino IDE that allows you to send and receive data from your Arduino board.
- Resolution: The degree of detail that can be observed or captured.
- Logic: A system of reasoning used to evaluate arguments and make decisions based on consistent and structured rules.

Revision History

Revision	Date	Changes
A	22/07/2024	Initial release.
B	23/07/2024	Removed code examples and shifted these to pre-made sketches downloadable from evolocity.co.nz . General tidying and rewording.
C	17/07/2025	Removed background from images, wording changes and general tidying. Rewrote Exercises & Examples section - simplified content and included code within the document.