Alexandru RADOVICI Ioana CULIC Ovidiu STOICA Daniel ROSNER

Building a Smart City Infrastructure using Raspberry Pi and Arduino with Wylio drin STUDIO

Building a Smart City Infrastructure Using Raspberry Pi and Arduino with Wyliodrin STUDIO

Alexandru RADOVICI, Ioana CULIC, Ovidiu STOICA, Daniel ROSNER

June 28, 2016

Contents

Part I

Introduction

About Raspberry Pi and Arduino

Raspberry Pi and Arduino are the boards most people use in order to build Internet of Things projects. They are widely used especially due to the small price and the high resistance to current spikes and short-circuits.

A common question people eager to get stared in electronics pose is whether they should get a Raspberry Pi or an Arduino board. The answer would be: "it depends on what you want to build". While the Raspberry Pi is powerful and good for processing, it lacks real time processing. On the other hand, Arduino is a real-time processor that lacks a lot of memory and processing power. If you want to connect simple sensors and read real-time data, we recommend you use an Arduino. If you need to process a large amount of data or connect to the network, you better use a Raspberry Pi. Usually, it is best to use them together: acquire data with the Arduino and process it using the Raspberry Pi.

Raspberry Pi

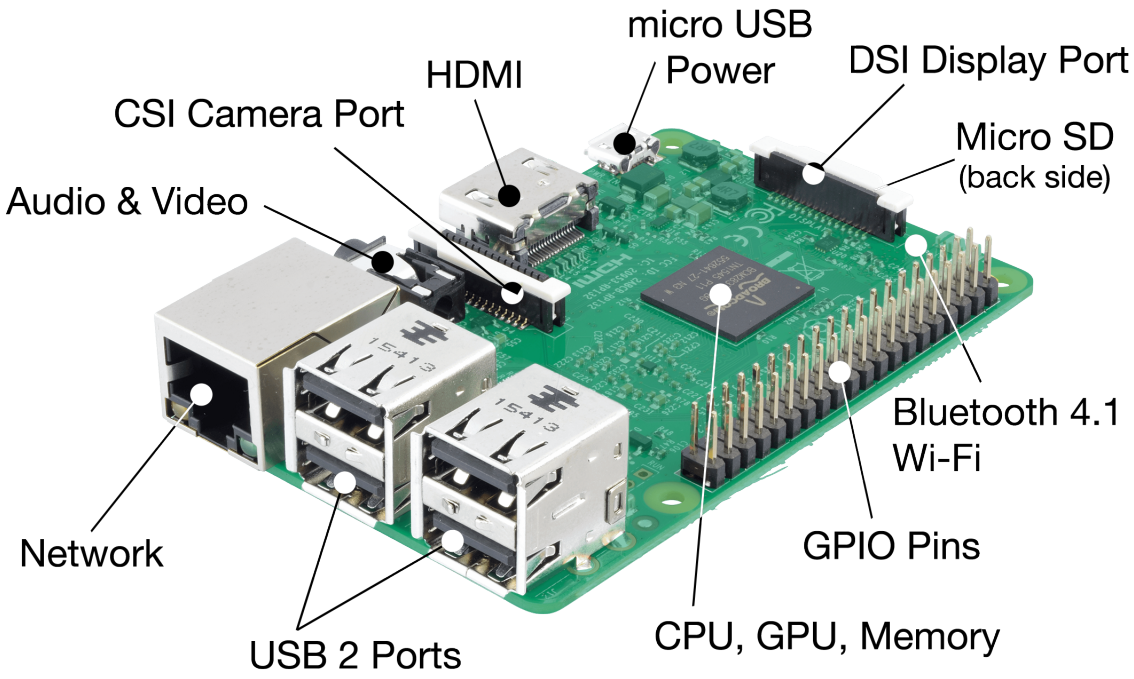


Figure 1: Raspberry Pi 3 Model B Board

Main features of the Rapberry Pi

The Raspberry Pi is a very small computer that has the characteristics of the computers people would use 15 years ago. Raspberry Pi 3 characteristics:

- Broadcom BCM2837 SoC (4 ARM Cortex-A53);
- 1 GB of RAM;
- integrated WiFi and BLE;

- HDMI port, an Ethernet port and four USB ports;
- several pins to use for electronics:
  - two 5V pins;
  - two 3.3V pins;
  - 8 ground pins;
  - 26 data pins;
  - 1 PWM pin.

Since it is a computer, the Raspberry Pi runs an Operating System. That means that you can run multiple programs on it and you can run applications that use Internet services. However, this also implies that the applications you run on the Raspberry Pi are not real-time, thus you cannot estimate when a certain sequence will get executed.

Raspberry Pi Pins Layout

As previously stated, the Raspberry Pi exposes some pins that allow you to connect peripherals to the board. Let's check them out.

The Raspberry Pi 3 has 26 GPIO soldered pins and 5 GPIO pins that you need to solder. These may be used to control electronics connected to the Raspberry Pi. Among the things that can be done are:

- light up LEDs;
- place buttons;
- use relays;

- control motors.

It is important to know how these pins can be accessed. For the Raspberry Pi, Wyliodrin STUDIO uses the WiringPi pins layout described in figure 2.

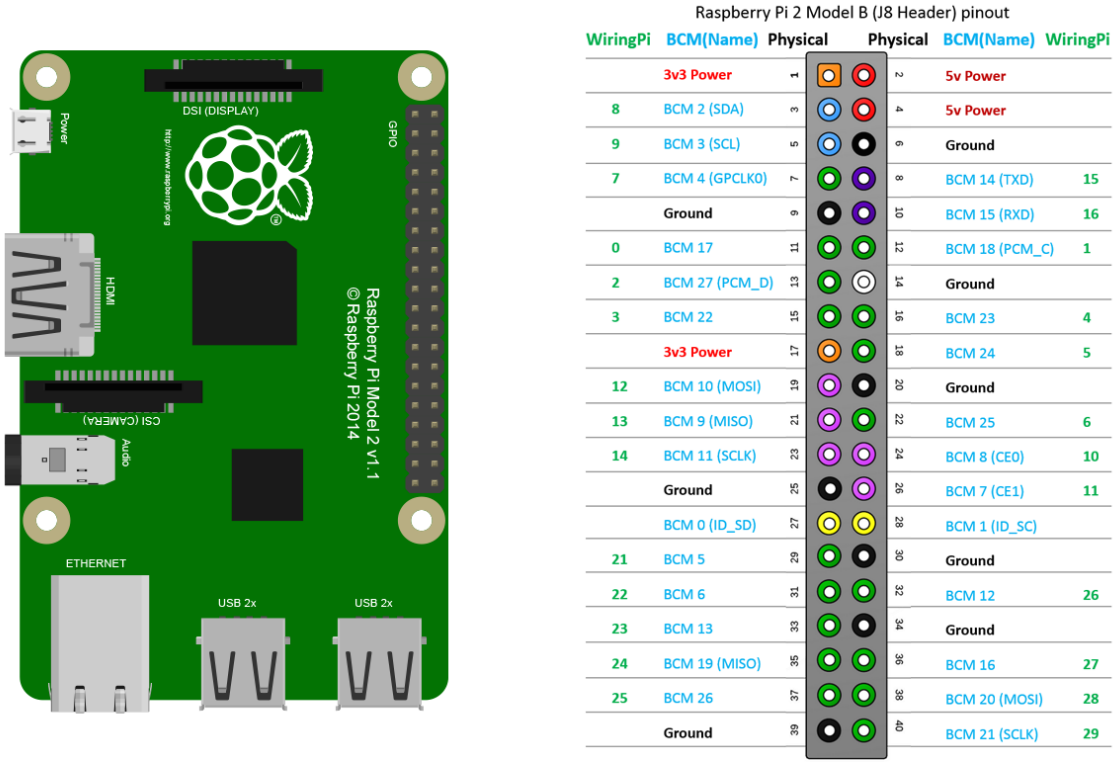


Figure 2: Pins layout Raspberry Pi 3

The data pins consist of *digital* pins but also pins that support *SPI* or *I2C* communication. You will read more about each of these protocols and how to use them in the following chapters.

Tips & Tricks

Be advised that Raspberry Pi pins run at 3.3V. Connecting them to 5 V pins (like the Arduino) might damage the board.

Arduino

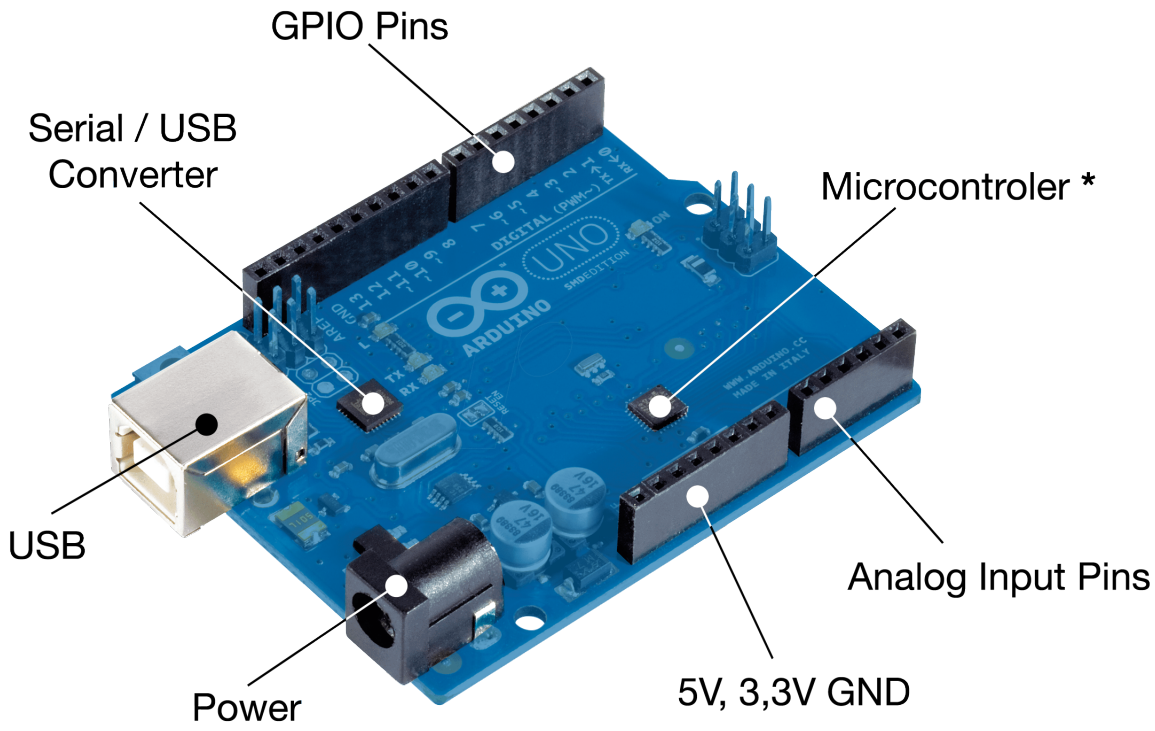


Figure 3: Arduino Board

Unlike the Raspberry Pi, Arduino (figure 3) is a microcontroller board. That means that it does not run an Operating System, what it runs is called Firmware. Basically, it runs only one program. The result is that you can estimate what program sequence gets executed at a certain time.

On the other hand, most of the Arduino boards cannot simply connect to the Internet, they need a shield to support ethernet connection. However, Arduino boards expose a higher number and a greater variety of pins.

Depending on the model, Arduino has around the following characteristics:

- 16 Mhz microcontroller
- 32 KB of program memory
- 2 KB of RAM memory
- USB/serial converter

Programs written to the Arduino remain there until they are replaced with another program. Even when powered off, Arduino stores its software and starts rerunning it at power on.

Arduino Layout

As previously said, Arduino is a simple board that executes only one program, but it exposes a large variety of pins. There are multiple Arduino boards on the market, the main differences between them consisting in the processing power and the number of pins. This is why it is difficult to state the exact number of pins a board has and their numbering. However, you can find the following types of pins on any Arduino board:

- 5V pins;
- 3.3V pins;
- data pins;
- analog input pins;

- reset pin (it allows to reset the board);
- AREF (used as reference when reading analog values).

The data pins can be used for *digital input/output* and also for *SPI* and *I2C* communication.

It is easy to identify the pins' number as next to each pin is written its number (1, 2, 3... for data pins and A0, A1... for analog input pins). In figure 4 you can see an example of an Arduino board with its pins layout.

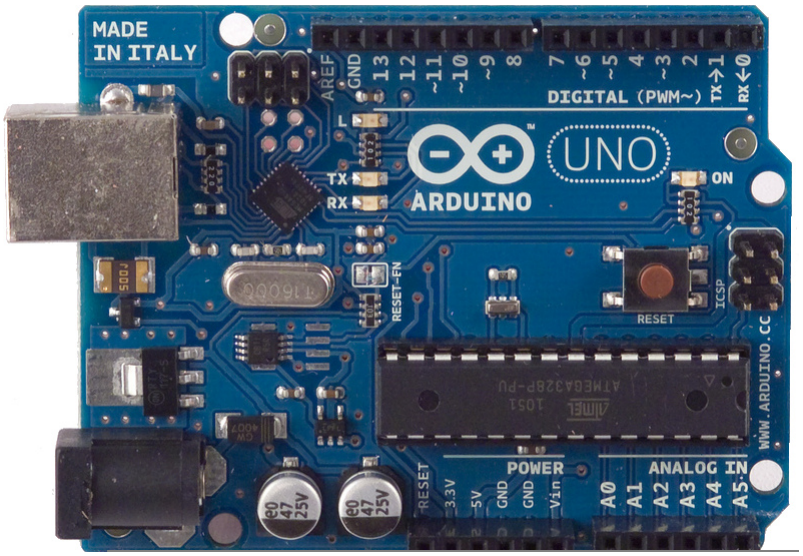


Figure 4: Arduino Uno pins layout

You will get more information on these pins and how to identify them in some future chapters.

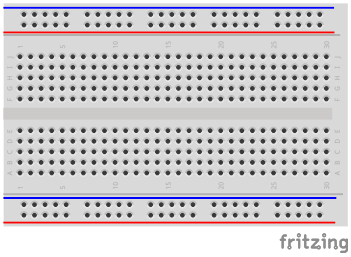
Breadboard usage

The breadboard is a kind of solderless electronic circuit building. It basically replaces the cables you otherwise would have to solder in order to connect the peripherals together.

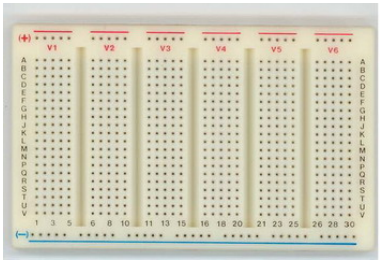
There is a certain number of holes on a breadboard. The common type of breadboard has two areas called strips.

- The bus strips usually get wired to the power supply of the circuit. They are arranged on two rows, one for voltage, usually marked with a red line along the row,the other one for ground, drown in black.
- The socket strips which reunite the components of a circuit, have a layout of multiple columns, each consisting of 5 holes.

Beware, each of the rows and each of the columns behave like a single hole, as their holes are connected as a series. In Figure 5a the columns are oriented vertically as opposed to Figure 5b, where the socket strips columns are horizontal.



(a) Vertical socket strips breadboard



(b) Horizontal socket strips breadboard

Figure 5

Conclusion

All in all, you can say that Raspberry Pi works well for applications that require both mobility and an Internet connection. On the other hand the Arduino is the best option for "electronics-only" projects.

However, the best solution to create amazing Internet of Things projects is

to combine these two. The Arduino can be connected to the Raspberry Pi so that you have both Internet access and a full range of different pins.

Introduction to Linux

The Raspberry Pi board usually runs Raspbian as operating system, which is based on the Debian Linux distribution. A Linux distribution is an operating system built around the Linux kernel. The main difference between different distributions is the package manager, basically, the way you install new software.

As any other operating systems, Raspbian allows you to control the board via a Shell by using standard Linux commands. Although this sounds intimidating, especially nowadays when the GUI makes everything intuitive and extremely easy, sometimes it is the only viable option.

Further on, the book will present the Shell's characteristics together with some basic commands and when you might need to use them.

The Shell

The Shell is a window that allows you to interact with the board. It waits for you to enter a command and it executes it. Once you open a Shell, you will see the prompt. That means that everything works fine and the Shell is waiting for your command.

The prompt also offers you some information. First of all, it shows you the user currently logged in. The user's name is what you see before @. It also

username@hostname:~$

Figure 6: Shell prompt

shows the host name of your board.

The most important information the prompt displays is the working directory. That is the directory where you are currently working in. It is displayed right after the colon in the prompt. You will notice that the default working directory is `~`. That is the user's home directory and its equivalent is */home/username*.

Paths

In order to access a certain file or directory, you have to take into account the path to it. There are two different paths you can use: absolute and relative.

In Linux, the directories' structure is like a tree. The root directory is `/` and it contains all the other directories and files.

If you use an absolute path to a file or a directory, that means that you build the path to it starting with the root directory. Thus, you can say that any path that starts with / is an absolute path.

On the other hand, you can use a relative path, which means that you build it starting from the directory you are working in, your working directory. Thus, all the files and directories are relative to it.

When building paths, there are three symbols you should be familiar with:

- . - current directory
- .. -parent directory

- ~ - home directory (/home/username)

pwd

The *pwd* command makes the Shell print the working directory. It is important to know which directory you are working in and sometimes it is difficult to get it from the prompt. So, anytime you feel lost, use *pwd*.

```
pi@raspberry:~$ pwd
/home/pi
```

Figure 7: Example of pwd output

ls

ls makes the Shell print all the files and directories located in the working directory. If you want to see the contents of some other directory, you can pass that directory as an argument to the command. For instance, if you want to print all the files and directories in /, you will write: *ls /* .

cd

You already know that usually, once you open a Shell, the working directory is your home directory. However, you will need to work in other directories too. In order to change the working directory, you will have to use *cd* followed by the directory you want to go to.

For example, if your home directory contains a directory called *homework* and you want to have that as the working directory, you use *cd homework*. You can notice that you used a relative path. Some other alternatives would

be *cd /home/pi/homework* or *cd ~/homework*. In the last two examples you used an absolute path to refer to *homework* directory.

cat

cat asks the Shell to print the contents of a file. However, it must be clear that you can only see its contents, you cannot modify them. For that you need an editor.

Similarly to the *cd* command, *cat* gets as an argument the file it should display.

Example: *cat /etc/passwd*

htop

By using the *htop* command you can see real-time all the processes that run on your board. Once you entered the command you will notice that the prompt does not appear, that is because you cannot enter another command until you are finished with displaying the processes. So, if you want to go back to what you were doing, just hit the *q* key.

For each process displayed, you can see its PID (Process ID), the user who launched the process, how much CPU and memory it is using, the command that started the process and other information.

What you are most interested in is the PID. That is because each process can be identified by its PID and if you want to interact with it, you have to know its process ID.

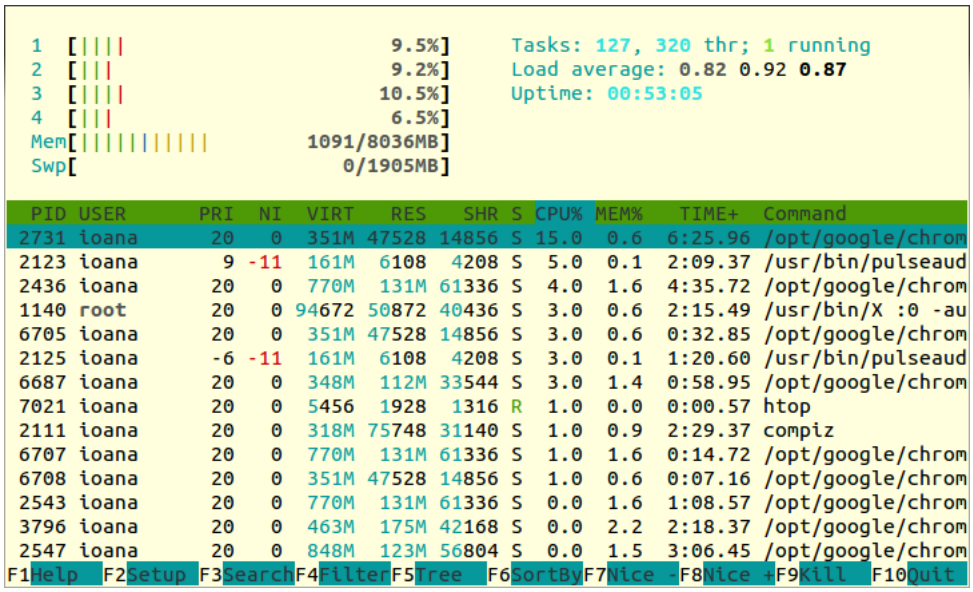


Figure 8: htop output

kill

We know that you can use *htop* to find a process' ID in order to be able to interact with it. *kill* is the command that allows us to interact with another process.

Two processes can interact by using signals. A signal is a number a process sends to another. Both processes know that each number represents an action. You can refer to a signal either by the number or by its name.

```
ioana@ioana:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Figure 9: List of signals

The format of the *kill* command is the following: *kill -signal pid*, where signal is the number representing the action you want to process to do and pid is the process ID.

The two signals you are most interested in are *SIGTERM* (number 15) and *SIGKILL* (number 9).

SIGTERM tells the process to stop its execution. Normally, the process should save all its data and stop running. However, this signal can be ignored by the process. There are times when you cannot kill a process by using SIGTERM.

On the other hand, SIGKILL, kills the process no matter what. The downside is that the process does not have the opportunity to save its data, so killing it like this can result in loss of data. Neverthless, if something happened and your process must be forced to stop, you have to use SIGKILL.

In case the running process has a Shell attached and you can access it, you can simply use a key combination to send the SIGTERM signal to it and make it stop, *Ctrl+C* .

Example: kill -9 1279

killall

killall has the same effect as *kill*, except that you do not have to know the PID of the process, but its name. Instead of passing the process ID as an argument, you have to pass the process name.

Example: killall -9 python

Tips & Tricks

Getting used to working with a Linux Shell is not difficult, especially if you know the following tricks:

- Whenever you are typing a command use the *TAB* key. It will auto complete what you wanted to type, thus eliminating spelling errors. In case there are multiple possibilities, press TAB once more and they will be displayed. If by pressing TAB the command or the argument you want to type is not automatically filled in, it means the command is not valid.

- The most important command you should know is *man*. By using *man* followed by another command name, you have access to that command's manual and you can find how to use it and all it can do.

Hardware and Electronics for the Internet of Things

Electric Signals

A signal is a function that conveys information about a phenomena.

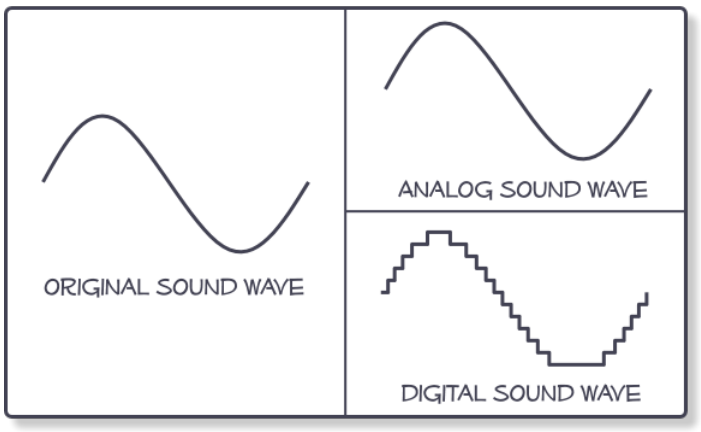


Figure 10: Digital and analog signal representations

If you imagine a sound wave, an analog representation of the signal would correspond to the original one, but a digital representation would create *steps* as for each level is assigned a value. The digital representation can have different accuracy rates.

The representation of a signals can be referred to as a stream of measure-

ments. The accuracy of the measurements, depends on two important factors.

- The number of bits assigned to a measurement will either improve or damage the quality of the received signal. The signal as a whole is recomposed after many samples taken. For each sample, or measurement, the hardware can allocate one or more bits. A one-bit sample will split the signal's values in above half and under. The former will turn to 1 and the latter to 0. This signal representation will be far from accurate. Now, if there is a larger scale of bits, the hardware can adapt the signal closer to the original one.

  Supposing you have n bits to represent your signal, when taking a sample, the hardware decides which of the values from 0 to $2^n - 1$ to assign to the measurement. Basically, the wider the range of possible values, the more accurate the digital representation is.

- The sampling rate is just as important as the number of bits. The samples will be taken from time to time. In the interval between measurements the signal is lost. This way, the signal can be more or less approximated. The ideal case is when the sampling is made frequently, so little signal is missed. The time intervals will be shown on the graphics, following the horizontal axes.

Nyquist theorem says that in order to reproduce an accurate signal, you need to sample at least twice faster than the highest frequency of the signal. This means that if the bandwidth is lower than the highest frequency, the signal will be altered.

Introduction to Electronics

Basic electronics knowledge is required to be able to build the simplest IoT projects. All the necessary devices, pieces and boards follow the basic laws

that rule this field. These laws are presented and explained in the next few paragraphs. Not knowing and not respecting the electronics laws can get you into real trouble with your board or any other sensor or device you are using.

You are going to build IoT projects using Raspberry Pi and Arduino boards. However, the boards are only a part of the projects, they do the computing, but you also need I/O devices that you will connect to them. The devices are mainly sensors, LEDs, LCDs and servos. In order to correctly connect the peripherals, you need to be acquainted to basic electronics notions, otherwise, you risk to burn the I/O devices and even the boards.

Ohm's Law

The Ohm's law states that in a circuit the current (I) is directly proportional to the applied voltage (U) and inversely proportional to the resistance (R) of the circuit.

$$I = \frac{U}{R} \tag{1}$$

Kirchhoff's Circuit Laws

Before stating the two laws, you need to understand the following notions:

- junction/node - the place where at least three conducting branches meet;
- loop - a closed path, including at least two nodes.

Kirchhoff's First Law

Kirchhoff's First Law states that in a node, the sum of the currents is 0.

$$\sum_k i_k = 0 \tag{2}$$

Please keep in mind that currents have directions. Currents incoming have negative values, while currents outgoing have positive values.

Kirchhoff's Second Law

Kirchhoff's Second Law states that the total voltage in a circuit loop is equal to the power source voltage.

$$\sum_k E_k = \sum_k R_k I_k \tag{3}$$

Example:

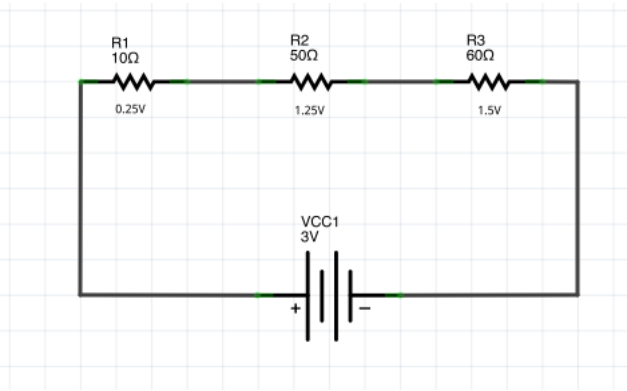


Figure 11: Kirchhoff's second law example

You have a 3V source and three resistors of different resistance (figure 11). The sum of voltage drops on each of them is equal to the source voltage.

$$R1 + R2 + R3 = VCC1 => 0.25v + 1.25v + 1.5v = 3v \quad (4)$$

Voltage Divider

A voltage divider is a circuit that outputs a voltage which is a fraction of the input voltage by using two resistors and an input voltage. It is important in giving you an understanding on how you should connect sensors and read their value. This is because most of the sensors are resistors with a variable resistance.

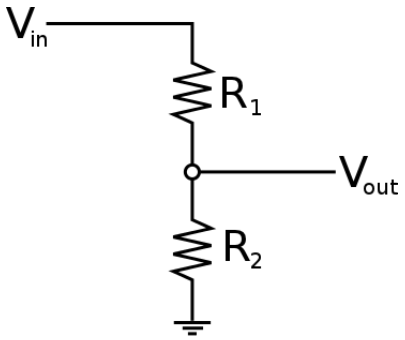


Figure 12: Voltage divider

Figure 12 depicts a voltage divider that uses two resistors while the V_{in} is the input voltage (imagine a battery connected to it) and the V_{out} is the output voltage, where a measuring device should be connected. You can imagine that in the projects you are going to build, one of the resistors will be replaced with a sensor, the V_{in} will be connected to a 5V or 3.3V pin and the V_{out} will be connected to an analog pin, while the ground gets connected to a GND pin.

By applying the Kirchhoff's laws to the voltage divider circuit, we obtain the following equation:

$$V_1 + V_2 = V_{in} \tag{5}$$

We also apply Ohm's law and these equations result:

$$V_1 = I * R_1 \tag{6}$$

$$V_2 = V_{out} = I * R_2 \tag{7}$$

Now let's consider the following particular cases:

- $R_1 = 0$

  This means that $V_1 = 0$ so $V_2 = V_{out} = V_{in}$.

- $R_2 = 0$

  This means that $V_2 = V_{out} = 0$.

- $R_2 = \infty$

  This means that no current will pass through R_2 and all the current will go directly to the measuring device, just like R_2 would not be connected to the rest of the circuit, so $V_{out} = V_{in}$.

- $R_1 = 0$ and $R_2 = 0$

  This means that the current flows directly from the power source to the ground, so we obtain a short circuit, which might damage the components connected in the circuit.

  Please be careful when building circuits so that you don't have a short circuit.

In order to read data from sensors, you need to replace one of the resistors with the sensor. The sensors will change its resistance varying from 0 to infinite, as a result, modifying the V_{out} value. Depending on which of the two

resistors is replaced, you can have a pull-up or a pull-down resistor.

For the pull-up resistor, R_1 remains as a resistor and R_2 is replaced by a sensor. R_1 is called pull-up because it connects the power source to the measuring device, so when the sensor has very high resistance (its resistance can be considered infinite) the current passes through R_1 and $V_{out} = V_{in}$. As the sensor gets its resistance lower, the value of V_{out} decreases getting to 0 when the sensor's resistance becomes 0.

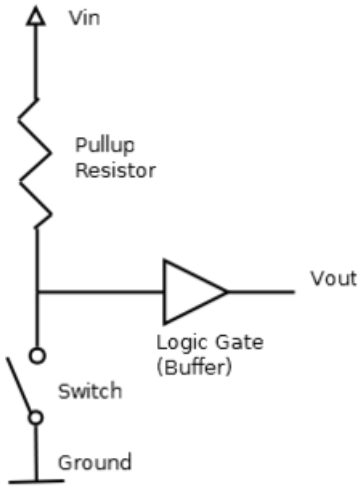


Figure 13: Pull-up resistor

For the pull-down resistor, R_1 is replaced by the senor, while R_2 remains unchanged. R_2 is called pull-down because it connects the ground to the measuring device. When the sensor has very high resistance, which makes it act like it is not connect to the circuit, the measuring device is directly connected to ground via R_2 and $V_{out} = 0$. As the sensor gets its resistance lower, the value of V_{out} increases in value, reaching the value of V_{in} when the sensor's resistance becomes 0.

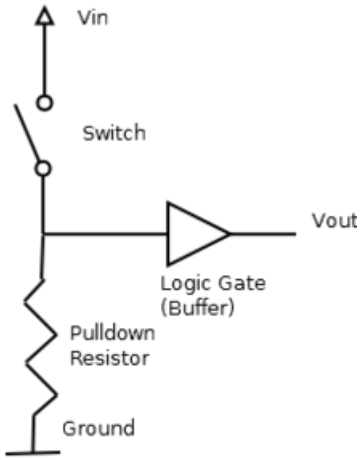


Figure 14: Pull-down resistor

GPIO pins

So far, you were presented with how to build a correct circuit. However, you will need to include the Raspberry Pi and the Arduino in the circuit in order to control it. This can be obtained by connecting the peripherals to the boards' pins.

There are two types of pins: input and output.

Input pins act like a voltage meter. They measure the voltage that comes in. Digital pins will read 0 for a voltage under half of the maximum expected and 1 for values above the half. Analog pins, on the other hand, will usually read values from 0 to 1024, depending on what the respective sensor is detecting.

Notice that for other boards, the range can be different.

Output pins are split into digital ones and PWMs.

Digital pins know 1 or 0 as values. They can basically either output current, or not. In a nutshell, output pins have the functionality of a battery.

Another type of outputs are represented by the PWMs. You might not know what a PWM (Pulse Width Modulation) does. It is basically a digital way of control, which means it oscillates between the values 1 and 0.

First of all, you should know that the boards have a clock that counts quanta of time. You can decide to split this quanta in two different parts: one during which you will send the value 0 and the other in which you send the value 1. Essentially, what the PWM brings in addition is a counter that memorizes the percent of the quanta during which the output value was 1.

The period while the signal has the value 1 will be the pulse width.

In a nutshell, you simulate an analog pin by changing the pulse width. The value of the PWM increases when you send more values that equal 1. However, this is not at all equivalent to having an output of 2.5 volts, for instance, instead of 5. This is why the PWM cannot be used with peripherals that require less than 5 V.

However, the PWM works perfectly as an example for the LED. What's important to know is that you basically have an LED that changes its light intensity from bright to none so fast that the human eye can't perceive the change, so you and I see it as a continuous light but dimmer or brighter. This is due to the ability of human eye to integrate these values.

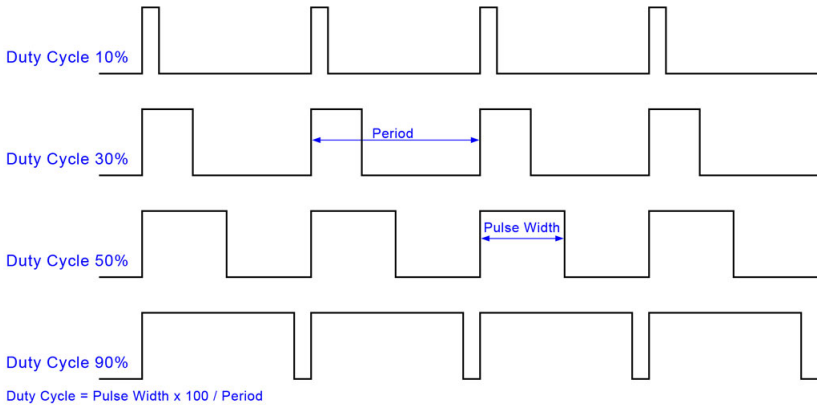


Figure 15: Pulse width

Analog to digital

ADCs or analog to digital converters turn the analog signal to a digital one. It is defined by the two parameters we described in the Electric Signals section: the sampling speed and the number of bits per sample. In addition to this, their quality stands in the speed by which they read the analog pins.

First of all, the mechanism work like this: the signal can be received through one or more of the analog pins: A0, A1 and so on. The ADC on a board is multiplexed, which means there is a controller which connects the ADC to pin A0, reads the value on it and reports it back. Next, the same will happen to pin A1, then A2 so on and so forth.

As a conclusion, the multiplexing speed is just as important as the sampling one.

Boards which can't read analog pins, such as the Raspberry Pi, don't have ADCs. On the other hand, Arduinos have ADCs, this is why you will need an Arduino together with a Raspberry Pi in the projects you are going to build.

Sensors and peripherals

Sensors

Sensors are devices that scan the environment and get data from it. Some examples of sensors are : thermistors, buttons, photo resistors, infra-red sensors, distance sensors.

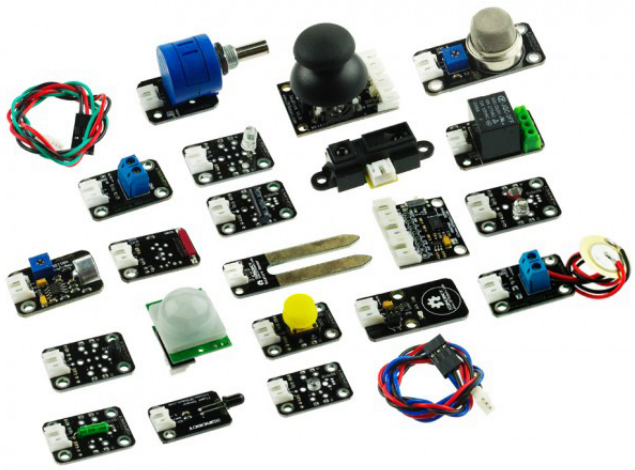


Figure 16: Sensors

Types of sensors:

- Analog Sensors - send an analog value that *needs to be processed by the ADC*;
- Digital Sensors - send a digital value.

Basically, all the sensing parts are analog. For the analog sensor the sensing part is directly connected to a microcontroller. They are easy to interface, you only need an ADC and to do some computation to find the real value.

Digital sensors are more complicated because the microcontroller that processes the data is already incorporated in the sensor and it sends the data out on a communication channel that implements some type of protocol. In order to get the data, the device needs to communicate with the sensor using that protocol.

Analog sensors should always be connected in a *voltage divider* circuit as they are resistors the change their resistance based on some environment feature (for example for the *photo resistor* the more light, the lower the resistance, for the *temperature sensor*, the higher the temperature, the lower the resistance). This is why measuring the sensor's value reduces to measuring the voltage

drop on the top of the resistor.

The sensors that have 3 pins usually have the voltage divider integrated: one pin goes to VCC, one goes to the ground and the third goes to the analog input. You need to look on the data sheet of each sensor in order to know where each pin should be connected.

Note: If you switch VCC with ground, some sensors will burn.

Analog sensors do not return an exact value. If the sensor has only two pins (variable resistor), the error is large because there is a disturbance in the environment that spreads to measuring the voltage. The ones with the voltage divider integrated usually have some corrections embedded, so the errors are smaller, but they still exist.

Note that almost all of the sensors are not linear.

A simple way to correct environment or reading errors is to average multiple readings (usually average 1000 samples). This usually works for micro controllers, but for computer boards, sometimes averaging values will result on reading the same value from memory (the pin will never update).

Digital sensors

Digital sensors usually have some microcontroller embedded in their building. Thus, reading values coming from that sensors revolves around making the two devices (sensor and embedded board) communicate. For the communication to be effective, there is the need of a protocol. Most of the digital sensors work with either of the two protocols: SPI and I2C.

SPI

This protocol involves several sensors acting like slaves and a master, the embedded devices such as the Raspberry Pi. The communication is always

initiated by the master. The pins for such a sensor are SS (slave select), MISO, MOSI, Clock, each with their role.

- MOSI - stands for master out, slave in, which means that a communication line is created to send data from the master to the slave;
- MISO - master in, slave out, is the line that facilitates the communication in the opposite direction: the slave writes on the line and the master samples;
- Clock - is generated by the master and sets the frequency of data transmission;
- SS pin - will be 0 when the slave is active and conversely 1 when the slave is inactive. In the first case, the slave waits for the clock to transfer data. One master has pins for each slave, but it cannot work with more than one slave at a time, since the MISO and MOSI lines would get impossible to use.

Communication in this protocol is always an exchange.

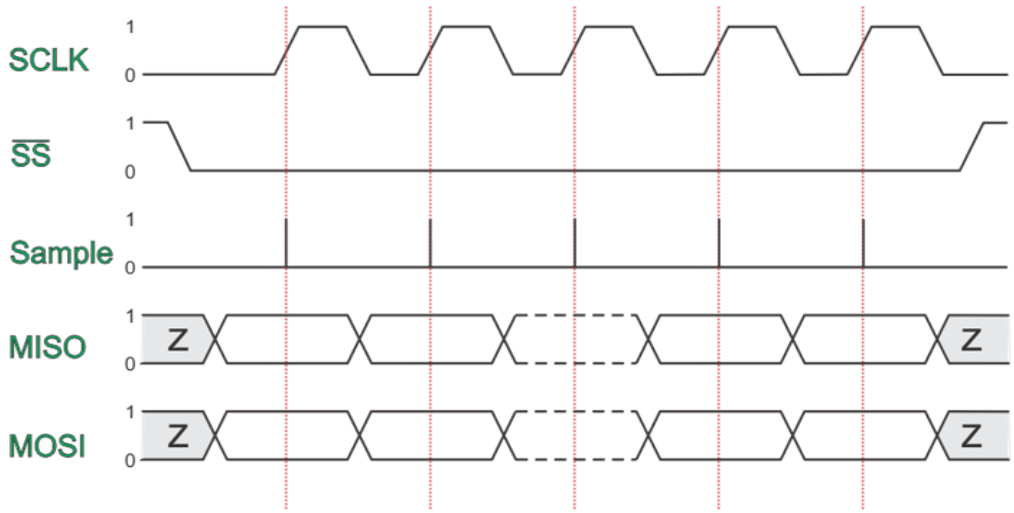


Figure 17: SPI protocol

I2C

This protocol works with one master and several slaves as well. This time there is no slave selection, but each slave is identified by fixed address. This can also represent a disadvantage as for most of the peripherals you are limited to connecting only one peripheral per type.

It only uses two lines: SDA and SCL: serial data and serial clock line. This protocol is called a half duplex as it only uses one line for communication. The transfer of data goes as follows. The master sends the address. The slave identifies itself, sends out and acknowledgement message, then the master writes or reads the data, all this on the same line.

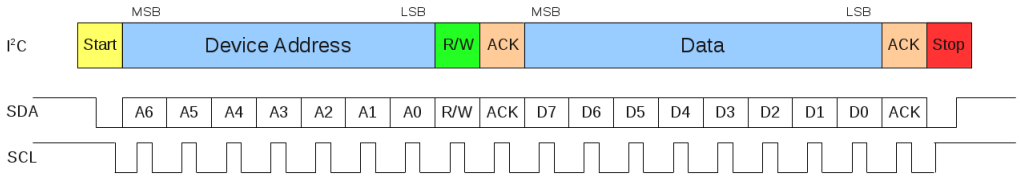


Figure 18: I2C protcol

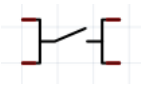
In general microcontrollers can be masters or slaves, but the computers can only be masters.

Button

The button, also called a switch, is an electronic component that can break an electrical circuit by interrupting the current. It is one of the simplest sensors that you can connect. It is, in fact, an analog sensor, but it only reports 2 values: either 1, either 0.

When used in schematics, there are multiple possible symbols to depict it (figure 19).

(a)



(b)

Figure 19: Button symbols

Also, figure 20 depicts an example of circuit that uses a switch.

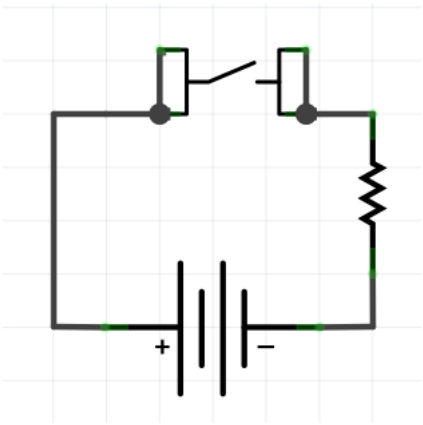


Figure 20: Button example circuit

When the button is pressed, it acts like a wire and it will let the current flow through the circuit. If the button is not pressed, the circuit is interrupted. So you can consider the button a resistor that either has zero resistance, or has infinite resistance.

As a result, you can connect it in a voltage divider circuit.

Figure 21 presents how to connect a button to a Raspberry Pi with a pull-up resistor. In this case, when the button is pressed, the V_{out} will be equal to 0 and when the button is released, $V_{out} = V_{in}$, so the value read on the pin will be 0 or LOW for a pressed button and 1 or HIGH of a released button. For this to work, you need a really small resistor so that when the button is not pressed, the voltage drop is insignificantly low, so you can still read 5 volts.

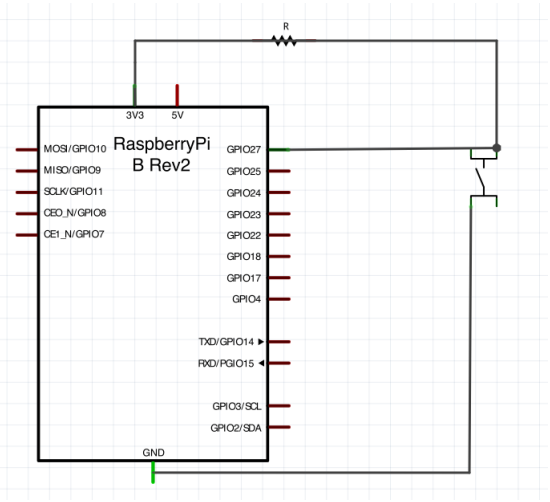


Figure 21: Button correctly connected to a Raspberry Pi

You can also connect the resistor to the ground. Now you have a pull-down resistor and the pin's value will be HIGH when the button is pressed and LOW otherwise.

When you press the button, the current is going to drain the 5V really fast. If it is pressed only for a short period of time there is no problem. It might become an issue only when the button is pressed for a longer amount of time. In conclusion, we need a resistor sufficiently high that we won't drain the source too fast, but sufficiently low to be able to read a value of HIGH.

Button Debounce

Button debounce is a phenomenon visible on micro controllers because they sample fast. Buttons are imperfect so, the moment you start pressing the button, you have a short time when the button's connectors start approaching and the distance is really small. Then you have an electric discharge and the resistance will be neither 0, nor infinite, so for a short period of time the resistance will vary.

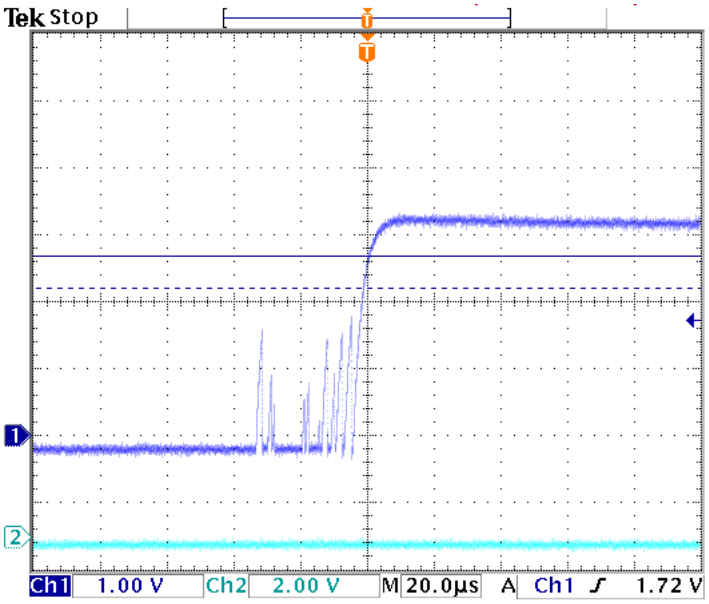


Figure 22: Button Debounce

With the micro controllers, which sample fast, you will see LOWs and HIGHs continuously changing. One way to debounce is to *average the values* and if the result is different from 0 or 1, then discard the measurement because the button bounced.

Peripherals

Peripherals are devices connected to and controlled by the embedded boards (LED, LCD, relay, piezo).

LED

This chapter explains how to correctly connect an LED to an embedded board.

First of all, you need to know what a diode is.

A diode is an electronic component that has a positive and a negative side and it basically allows the current to flow only in one direction, from positive to negative.

The LED is also a diode. When current is flowing through the LED, it lights up. So in order to light up an LED you need to put the high voltage at the anode and the low voltage at the cathode.

Another important aspect you must take into account when connecting LEDs is that they don't have any internal resistance. That means that if you simply connect an LED to a power source and to the ground, you will have a short circuit, so you need to also connect a resistor in order to limit the current flow.

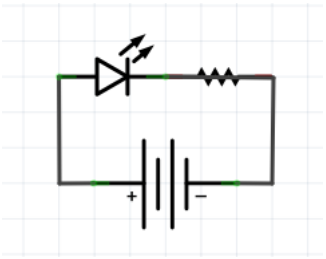


Figure 23: LED circuit

Based on the schematics, you can connect an LED to a Raspberry Pi by replacing the power source with a 3.3V pin and the ground with a GND pin. This will light up the LED. However, in order to control if the LED is turned on or off, you need to connect it to a GPIO pin which outputs current or not. This way, depending on the state of the pin, the LED will light up or not.

There are two ways of connecting the LED:

- You can connect the anode to a GPIO pin and the cathode to the GND. This means that once the pin outputs current, this will flow through the diode and the LED will light up. If the GPIO pin is set to LOW, there will be no current flowing and the LED will be turned off (figure

24).

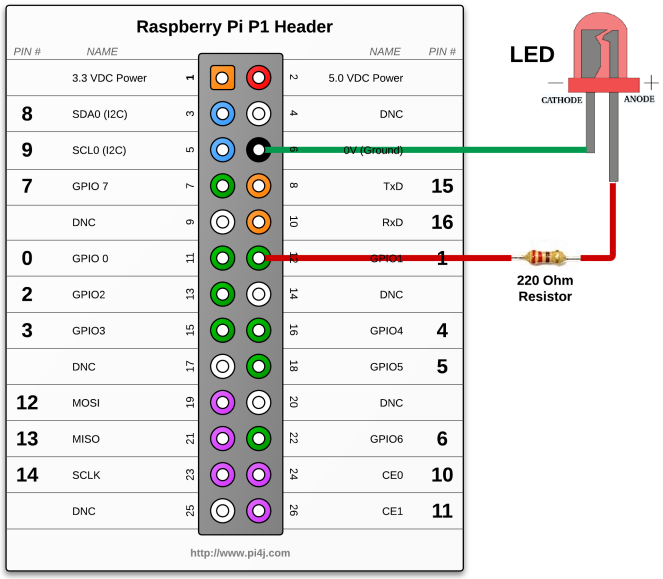


Figure 24: LED connected to the Raspberry Pi's pins

- You can connect the cathode to a GPIO pin and the anode to the 3.3V pin. This means that once the pin is set to LOW, it acts as a GND pin and current will flow from the 3.3V pin to the GPIO. If the GPIO pin is set to HIGH, both end will be connected to 3.3V, so there is no potential difference, as a result, the current won't flow through the circuit, hence, the LED will be turned off.

Safety Instructions

Whenever you connect anything to your board you must assure that it is not powered up. Otherwise you might accidentally create a short-circuit and burn the board.

Only after you assured that everything is correctly connected, you may safely power it up.

Microcontrollers and Computers

There is a big difference between microcontrollers and computers (embedded boards).

The first ones are low powered, have low memory and processing power but they are really good at controlling hardware because of the various type of pins they have (GPIO, PWM and ADC). Their big advantage is that they are real time, as the only program running on the board is the program you wrote.

On the other hand, embedded boards run some OS and have a bigger processing power and large memory but they cannot really control hardware.

Usually the solution is to use a computer board to get data from a micro controller, process the data and then use the microcontroller to control some hardware.

Software for IOT

Wyliodrin STUDIO

Wyliodrin STUDIO is a web-based platform for Internet of Things applications development.

The platform is an IDE that has all the features a complete IoT development solution should offer. It connects to your boards and allows you to write applications in multiple programming languages, deploy the applications on multiple boards and visualize data coming from sensors using graphs in a dashboard.

The main characteristics of the platform are:

- the Dashboard: you can add charts to your projects. These have a debugging function that allows you to watch the progress of your application, the graphic representation of the values sent by the sensors and check for mistakes. It is not for long time usage or for deploying software.

- Wyliodrin STUDIO has an open source universal pin-control library which allows you to write platform independent applications. libwyliodrin exposes Arduino-like functions in order to control the pins of embedded boards such as the Raspberry Pi. The library is build on top of several embedded boards, which means that any application that uses

libwyliodrin functions can be directly deployed on any of the supported boards.

libwyliodrin is open source, which means that you can include it in your projects independent of the IDE you use.

- The languages that you can use in Wyliodrin STUDIO to build a project are varied. If you want to code, you can start doing it in either C, C++, JavaScript, Python and others, but you can also decide to use Visual programming or Streams.

Programming Languages

As mentioned before, you can choose from a wide array of programming languages when building an Internet of Things application with Wyliodrin STUDIO .

One of the most used languages is C, which is mostly used for programming microcontrollers. For small, simple projects (prototyping), there is the option of visual programming. This uses intuitive blocks which generate python code, so they can be used by anyone. The visual programming language is based on Google blockly, with special hardware control blocks added. On the other hand, for more complex applications writing code is more practical.

You also can use visual programming. Drag some blocks and see the code equivalent to them in Python or JavaScript. That will help you if you do not know how some function work or what parameters they use.

Another programming language Wyliodrin STUDIO supports is Streams, which is based on Node-RED, developed by IBM. This language is a data driven one, as opposed to imperative programming languages. It is based on events, so an action occurs at the moment some data arrives in a certain

point.

The language uses elements as the one in figure 25. These blocks can be connected and they transmit messages one to another. Once a block receives a message, it processes it and sends it forward to the next node(s).

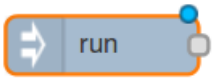


Figure 25: Run node

For instance, in figure 26, the *run* node will send every one second a message to the *print* node, which will print the received message in the console. You can consider that the *print* node has a callback function, which gets called once the message has been received.

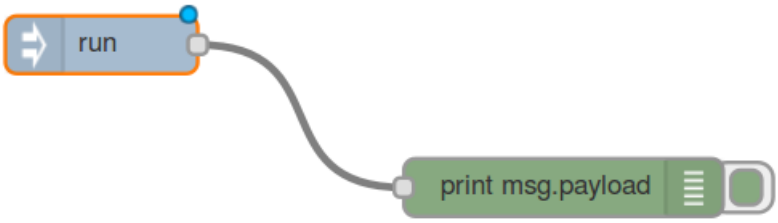


Figure 26: Run node

These blocks have several properties, some are general and all the nodes have them, while others are specific to a certain type of node. The general properties are:

- payload - stores the message;

- topic - contains information about the payload (is optional).

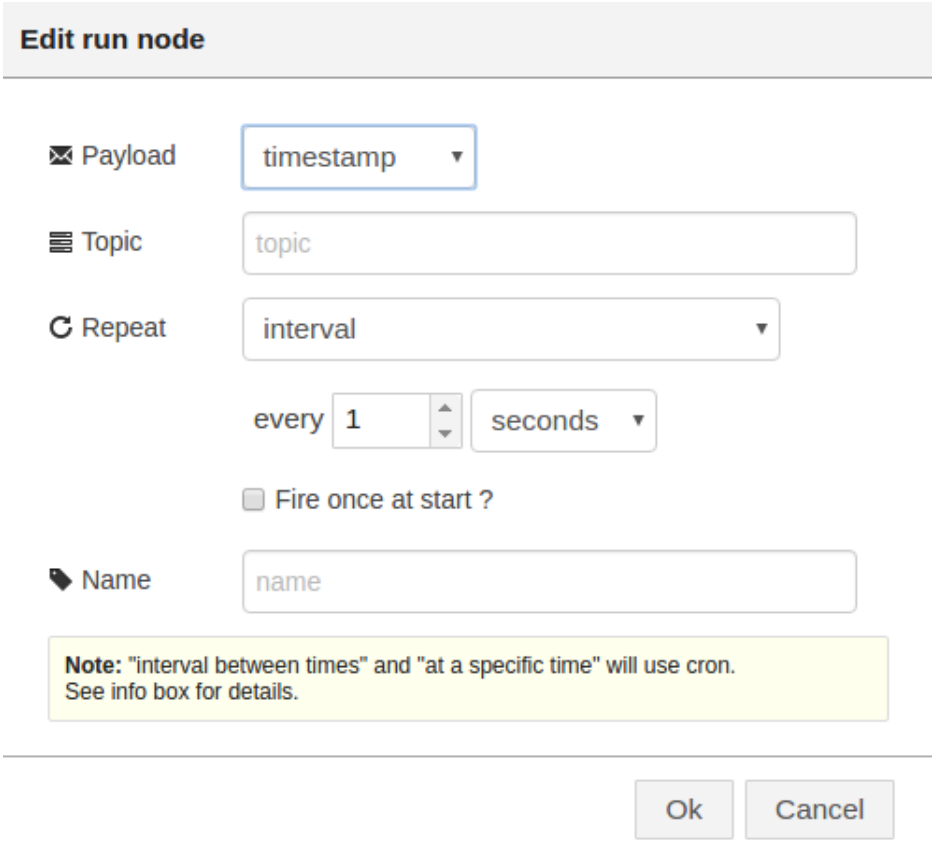


Figure 27: Properties of run node

After a Streams program is built and run, it is converted to imperative language, as this is the only way a computer can understand and execute it.

Most of the nodes are explained in section III.

Raspberry Pi Setup

For the projects you are going to build you will use a Raspberry Pi connected to Wyliodrin STUDIO . Let's see how to set up your Raspberry Pi so you can access it via Wyliodrin STUDIO .

Besides the Raspberry Pi, you will need an SD Card with minimum 4 GB (class 10 is recommended). You also need to assure the Raspberry Pi has a network connection or a serial connection to your computer (Ethernet or serial cable). Setting up the board requires the following steps:

1. Launch Wyliodrin STUDIO ;
2. Download the Raspberry Pi SD Card Image;
3. Write the image to an SD Card;
4. Connect the board to your computer;
5. Connect the board to Wyliodrin STUDIO .

Launch Wyliodrin STUDIO

First of all, please make sure you have Google Chrome installed and launch it. Go to studio.wyliodr.in , add the application to the Chrome browser and launch it.

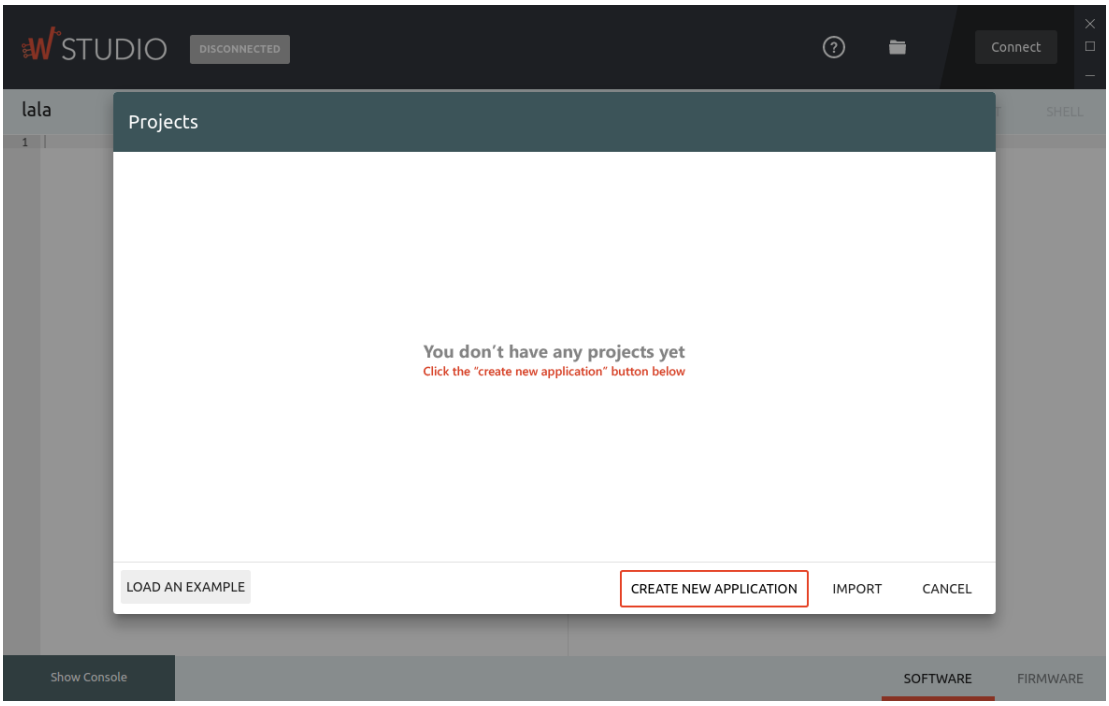


Figure 28: The Wyliodrin STUDIO interface

Download the Raspberry Pi SD Card Image

This is necessary only if you don't already have your Raspberry Pi set up.

In case your board is already running Raspbian Jessie and you can connect it to your the network your computer is connected to and also to the internet, you can skip this step.

From the Wyliodrin STUDIO interface, choose *Help->Setup* and choose Raspberry Pi as the board you want to connect. A tutorial on how to connect the Raspberry Pi will appear.

You have to download the SD Card image that you are going to use. You can find a link to that image in the window that depicts how to connect the

Raspberry Pi to Wyliodrin STUDIO . Once you click on the link, the image starts downloading. You need to *unzi*p the archive to obtain the SD Card image.

Write the Image to an SD Card

Writing the image to an SD Card is different depending on your system. you are going to present the steps you need to do depending on you computer's operating system.

Windows

1. Insert the SD card into your SD card reader and check what drive letter it was assigned. You can easily see the drive letter (for example E:) by looking in the left column of Windows Explorer. You can use the SD Card slot (if you have one) or a cheap Adapter in a USB slot.

2. Download the Win32DiskImager utility

   (http://sourceforge.net/projects/win32diskimager).

3. *Extract* the img file from the zip file and run the Win32DiskImager utility; you may need to run the utility as Administrator! Right-click on the file, and select *Run as Administrator*.

4. Select the *unzipped* Wyliodrin STUDIO SD Card Image for the Raspberry Pi.

5. Select the drive letter of the SD card in the device box. Be careful to select the correct drive; if you get the wrong one you can destroy your data on the computer's hard disk! If you are using an SD Card slot in your computer (if you have one) and can't see the drive in the

Win32DiskImager window, try using a cheap Adapter in a USB slot.

6. Click Write and wait for the write to complete.

7. Exit the imager and eject the SD card.

Linux

1. Insert the SD card into your SD card reader. The card will appear in */dev* usually under the name of *mmcblk0*.

2. Open a terminal and enter the following command :

   ```
   dd if=raspberrypi_image_file of=/dev/device_name
   ```

   where *raspberrypi_image_file* is replaced with the name of the unzipped Wyliodrin STUDIO SD Card Image for the Raspberry Pi and */dev/device_name* is replaced with the path to the SD Card, usually it will be */dev/mmcblk0*.

3. Wait until the process has finished.

4. Eject the SD card.

Mac OS

Option 1 PiWriter

1. Insert the SD Card into the SD Card reader or use a cheap SD Card adapter for your computer.

2. Download PiWriter utility. This will be used for writing the Wyliodrin STUDIO SD Card Image (https://github.com/Wyliodrin/PiWriter).

3. Run PiWriter. You will be prompted for an administrator user and

password. You will need to have administrator right to use PiWriter. If unsure what to do, just type in your password.

4. Follow the instructions on screen

5. When prompted to select a file, select the unzipped Wyliodrin STUDIO SD Card Image for Raspberry Pi.

6. Wait for the write to complete.

7. Exit the imager and eject the SD card.

Option 2 Apple-Pi Baker

1. Insert the SD Card into the SD Card reader or use a cheap SD Card adapter for your computer.

2. Download Apple-Pi Baker utility. This will be used for writing the Wyliodrin STUDIO SD Card Image (http://www.tweaking4all.com/downloads/ in the Raspberry Pi section)

3. Run Apple-Pi Baker. You will be prompted for an administrator user and password. You will need to have administrator right to use Apple-Pi Baker. If unsure what to do, just type in your password.

4. Just open the software and follow these three steps.

   - In the left top corner you will be asked to select the SD-Card you want to write the image on, if it's not automatically detected.

   - Select in the right top corner the image you just downloaded.

   - Click *Restore Backup*

   That is all you have to do. Just wait until the process is completed and your card should be ready to use. You have the option of making a Backup copy for the SD-Card you are writing.

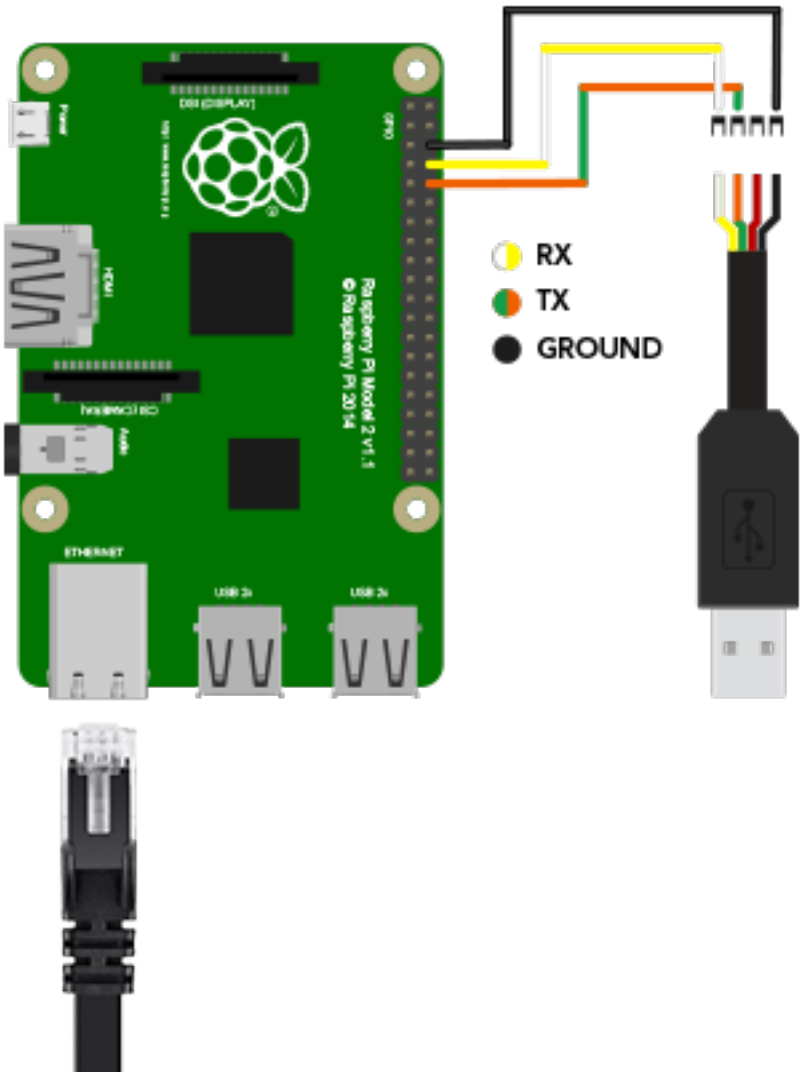


Figure 29: How to connect FTDI cable to Raspberry Pi

Connect the Board to Computer

In order to be able to control the Raspberry Pi from Wyliodrin STUDIO , the board needs to have a connection to the computer running the software. The connection can be made in two ways: via serial or via network.

Connect via Serial

Make sure you followed the previous step. You cannot connect the Raspberry Pi to Wyliodrin STUDIO via the serial connection if you don't have the Wyliodrin STUDIO SD card image.

In order to connect the Raspberry Pi via a serial connection, you need a serial FTDI USB-to-TTL cable. You need to connect the cable like depicted in figure 29.

Connect via Network

If you don't have an FTDI cable, you can simply connect the Raspberry Pi (via ethernet or via a WiFi dongle) to the network the computer is also connected to. Once the board will be powered on, Wyliodrin STUDIO will automatically detect it and will allow you to connect to it.

If your board is running Raspbian Jessie and by connecting it to your network, it also has an internet access, Wyliodrin STUDIO will detect the board and automatically install all the required libraries.

Connect the Board to Wyliodrin STUDIO

After the previous steps are completed, you need to power up the Raspberry Pi. Afterwards, click the *connect* button and choose your board. You will be asked for the username and the password. By default, the username is *pi* and the password is *raspberry*.

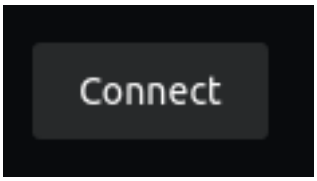


Figure 30: Connect button

In case your board is running the Wyliodrin STUDIO SD Card image, it will connect to the software. If your board is running the standard Raspbian Jessie OS, the necessary software will get installed on the board and then it will connect to the development platform.

Once you see your board connected to the software, we suggest you change its name so you can recognize it easier. You simply need to click the board's name and you can change it.

In addition, we recommend you also change the password of the user *pi* so other in the same network cannot connect to your board.

For that, once the board is connected, go to the *Shell* tab and type the following command:*passwd*.

You will asked for the current password and then for the new one. The terminal won't display anything while typing in the password, so don't panic, simply write it and press *ENTER* afterwards.

Part II

Tutorials

Light Blink

In this project you will make an LED blink, simulating the behaviour of a traffic light that marks a pedestrian crossing place.

What you need

- One Raspberry Pi board connected to Wyliodrin STUDIO ;
- One LED;
- One 200 Ω resistor;
- Jumper wires;
- Breadboard.

The Setup

Connect the LED to the Raspberry Pi following the schematics in figure 31.

First of all, you need to distinguish the anode from the cathode. The LED has two legs: one is longer, that one is usually the anode and the shorter one is the cathode. Another way of identifying them is by looking inside the

LED. There are two metal parts in it. The smaller one is connected to the anode and the bigger one is connected to the cathode.

In the schematics, the anode is connected to the GPIO pin 0 of the Raspberry Pi. The shorter leg is connected to the resistor and then to the ground pin of the board. However, the position of the resistor can vary.

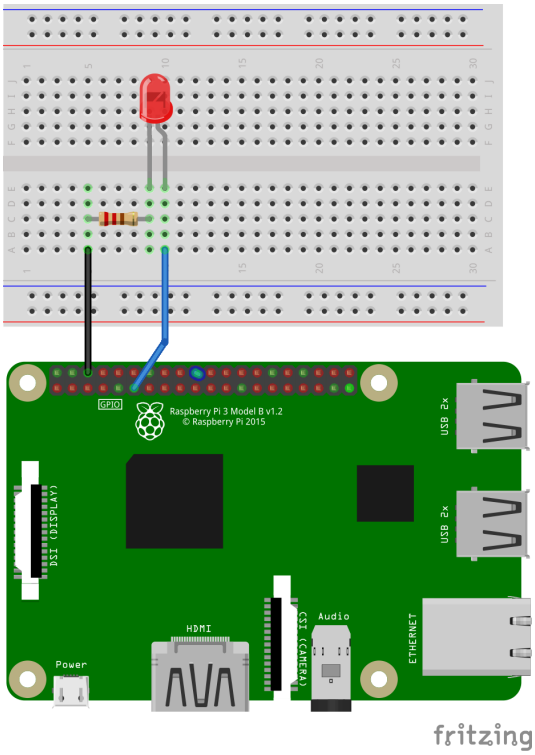


Figure 31: LED schematics

As the anode is the one connected to the GPIO pin, that means that the LED will light up when the GPIO is set to HIGH and it will be turned off when the GPIO is set to LOW. If you put the legs the other way around, the effect will be the opposite.

When connecting devices, you can follow pin layout schematics presented in the previous chapters in order to choose the pins. However, please be aware and use only the pins that have no other utility for the GPIO operations, otherwise you will have a random behaviour.

The Code

In Wyliodrin STUDIO , go to *Projects* and select *New Application*. Name the application *LED Blink* and select *Streams* as the programming language.

Once created, you click on the new application's name to open it. You need to connect the blocks similar to figure 32 in order to make the LED blink.

Figure 32: LED blink application

Any action in Streams starts with the *run* node. As you can see in its description, this node will set a payload at a certain interval. More specifically, you can choose in its settings to send the message at a certain time interval or at a specific time.

For this project, you need to specify the node to send the number of *times* it has been blinking, rather than the timestamp. To do that double click the node and select *times* for the payload. This way the node will send a payload with value 0 when it first fires, then 1, followed by 2 and so on.

In this example, the *run* node will send a message every 1 second to the *function* node. Double click on the latter to control its behaviour. The *function* block allows you to write JavaScript code in order to analyze the received message and send some message further on.

Figure 33 shows the function's code.

What the code does is to take the payload from the message sent by the first node (msg.payload), divide it by 2 and store the result in the message

payload. This means the new message will be either 1 or 0. The function returns just one of these values, depending on the case.

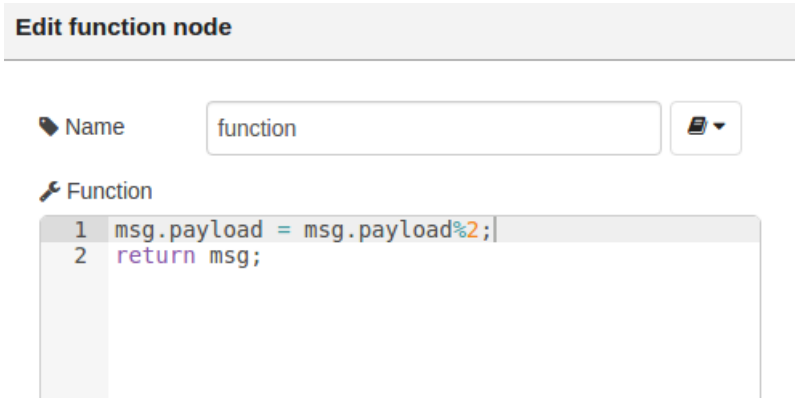


Figure 33: Function block properties

The last node is simply the digital write function. It will ask for the pin number as parameter, so double click on it and specify pin 0, the one the LED is connected to. The node will write the value received (0 or 1) on the selected pin.

In order to find out the number of the pin you wish to control, select the *Pin Layout* tab and you will see the figure describing the Raspbery Pi's pins. Notice that next to the GPIO pins there are two numbers, a smaller one which basically represents the order number, and another one, bigger. The bigger number is the one that identifies the pin within the application. So you simply need to count the pin the LED is connected to and the big, green number next to it is the pin number.

Run the code and the LED will start blinking.

Traffic Lights

What you need

- One Raspberry Pi board connected to Wyliodrin STUDIO ;
- Five LEDs;
- One button;
- Six 200 Ω resistors;
- Jumper wires;
- Breadboard

The Setup

You are going to build a traffic lights system consisting of a semaphore for cars and one for pedestrians. When pedestrians wish to cross the road, they need to push a button and the traffic lights will turn red and the pedestrian lights will turn green for 25 seconds.

First of all, you need to connect the LEDs and the button to the Raspberry Pi following the schematics in figure 34.

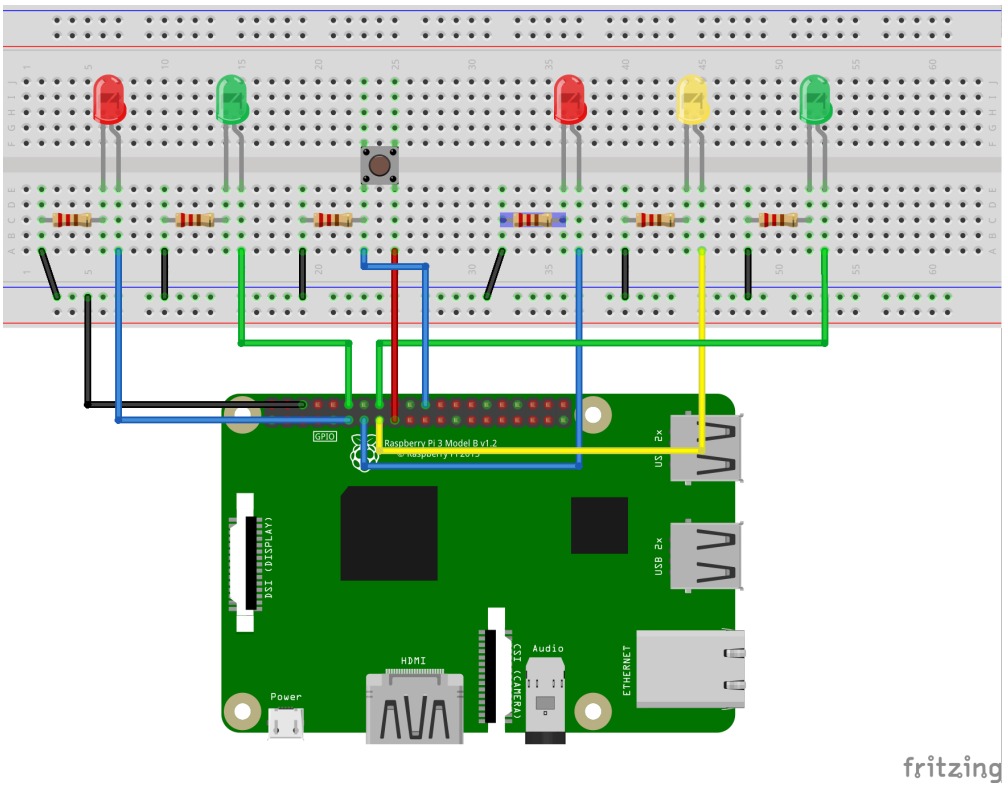


Figure 34: Traffic lights schematics

The button is connected similar to the schematics in the *Hardware and Electronics for the Internet of Things* chapter, in a voltage divider circuit. You can notice that the button has 4 legs. Usually, one leg from one side is connected to the one in front of it (figure 35). However, there might be the case that the legs care connected side by side, this is why we encourage you to use a measuring device in order to identify how the legs are connected. Once the button is pressed, all the legs get connected.

In the schematics, the button is connected to a pull-down resistor, that means that the GPIO pin will read the value 1 if the button is pressed and 0 otherwise.

You can check the pins' numbers by selecting the *Schematics* tab. For instance, in this example, the GPIO pin connected to the button has number

6.

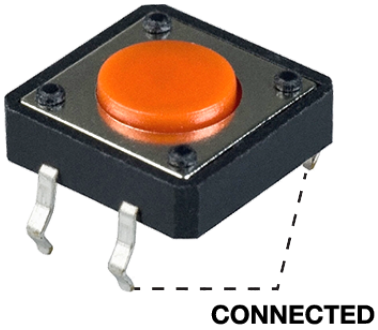


Figure 35: Button

The Code

Create a new Wyliodrin STUDIO application and name it *traffic lights*.

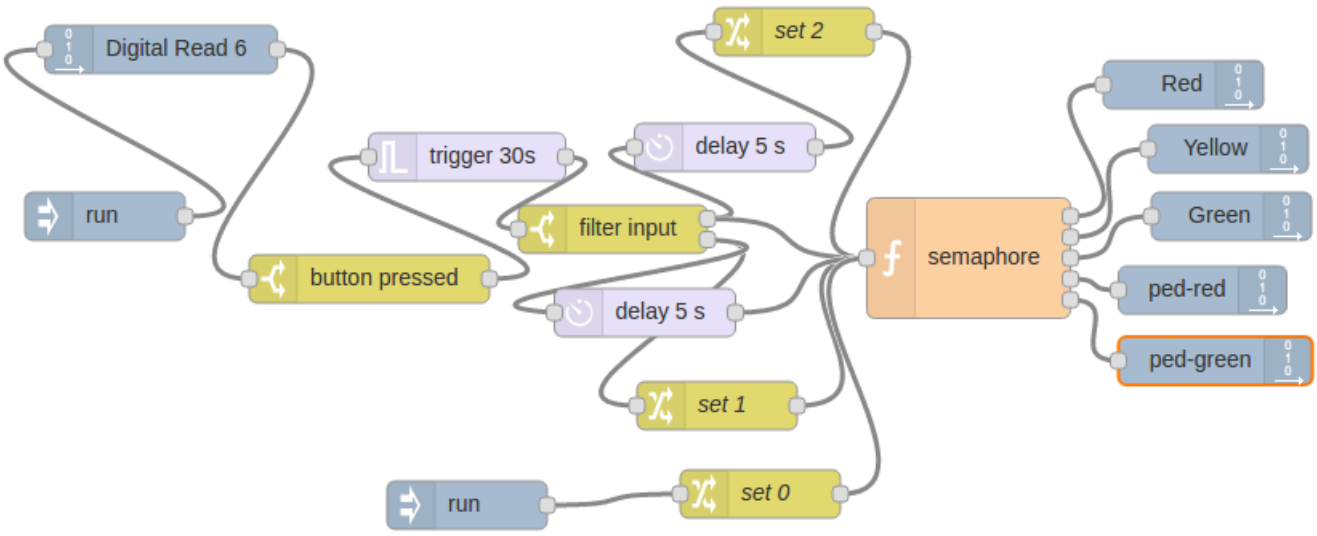


Figure 36: Traffic lights application

Figure 36 shows the nodes you need to use and how to connect them. Now let's see what each node does.

First of all, notice the *semaphore* function node. This node is connected to the nodes controlling the GPIO pins connected to the LEDs and it outputs the values that need to be written on each pin.

Figure 37 shows the code of the function. Basically, it verifies the value of the payload it received and depending on the value, it describes a certain state. The value 0 corresponds to the initial state when the cars lights have their green light turned on and the rest turned off and the pedestrian lights have the red light turned on and the green one turned off. This is why the function returns a list containing the values [0,0,1,1,0]. The block automatically knows to distribute each of these values in a message on the corresponding output (the first value is transmitted on the first output, the second value on the second output and so on). As a result, the *digital write* nodes (red, yellow, green, ped-red and ped-greed) receive the values 0,0,1,1 and respectively 0.

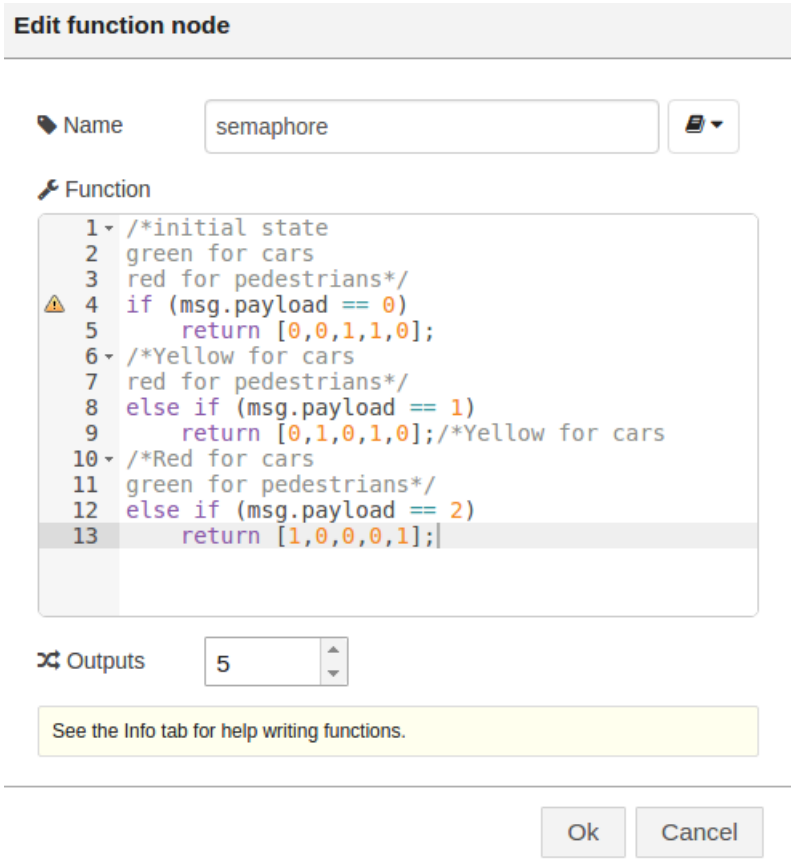


Figure 37: Semaphore function

The *digital write* nodes have names according to the LED they control (the red node controls the red LED of the cars' semaphore and so on). Each node controls a pin by either setting it to HIGH or to LOW. The pin will be set to HIGH if the payload of the message is 1 and to LOW if it equals 0. For each node, you need to select the number of the pin they are controlling. For the *red* node you need to type the number 2 for the pin, as the LED representing the red light is connected to GPIO 2.

Another important aspect are the two *run* nodes. The one directly connected to the *set 0* node will get triggered once the application start running and then it will send a message every 48 hours. For this, you need to double click the node and mark the *Fire once at start?*. Afterwards, you also need to set

the interval to every 48 hours.

The node connected to *run*, *set 0* is a *change* node and it allows you to change the payload of the message. As the first state of the system is 0, the payload is set to 0 and sent to the function. Basically, the run-set 0-function connection resets the traffic lights every 48 hours. All the othe *set* nodes work the same, but they set some other values to the payload.

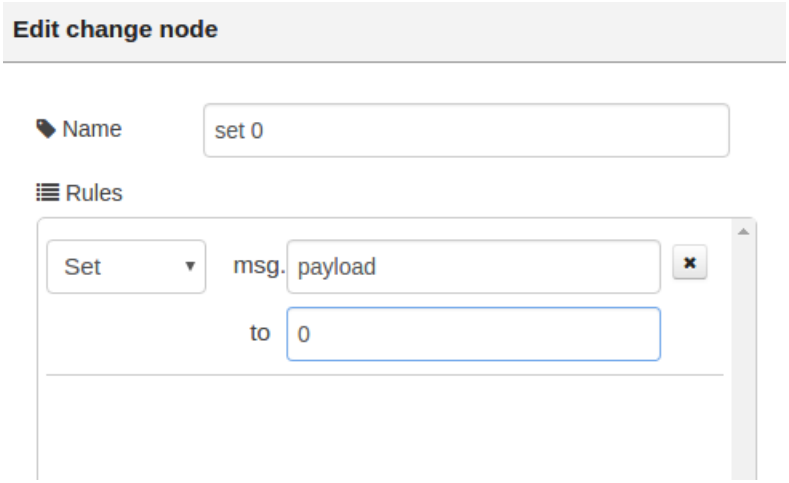


Figure 38: Change node properties

The other *run* node is connected to *digital read*. As you want the sampling to be done 10 times a second, set the *run* node to send a payload every 0.1 seconds.

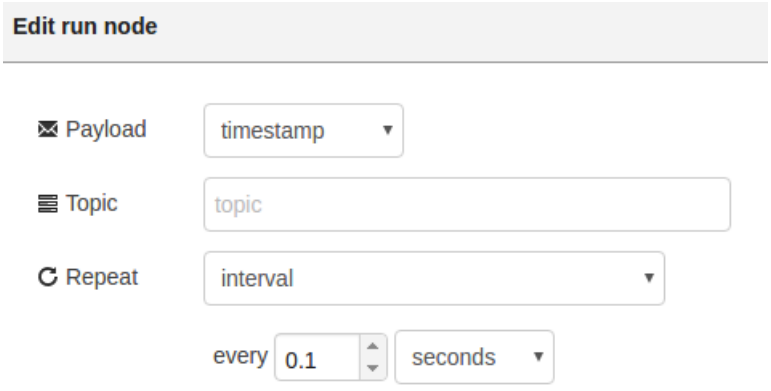


Figure 39: Run every 0.1 seconds

The *digital read* node reads the value coming from the button. You have to set the number of the pin the reading should be done from. In this case, the button is connected to pin 6, so go to the node's properties and set 6 for the pin.

If the value read is 1 (HIGH), you need to change the states of the lights, otherwise you can simply ignore the reading. The node named *if button pressed* is a *switch* node that routes messages based on their properties. This specific node has only one output which will send further on any message that has the payload 1, basically it will act as a filter which sends a message only when the button is pressed. Figure 40 depicts how to filter the values.

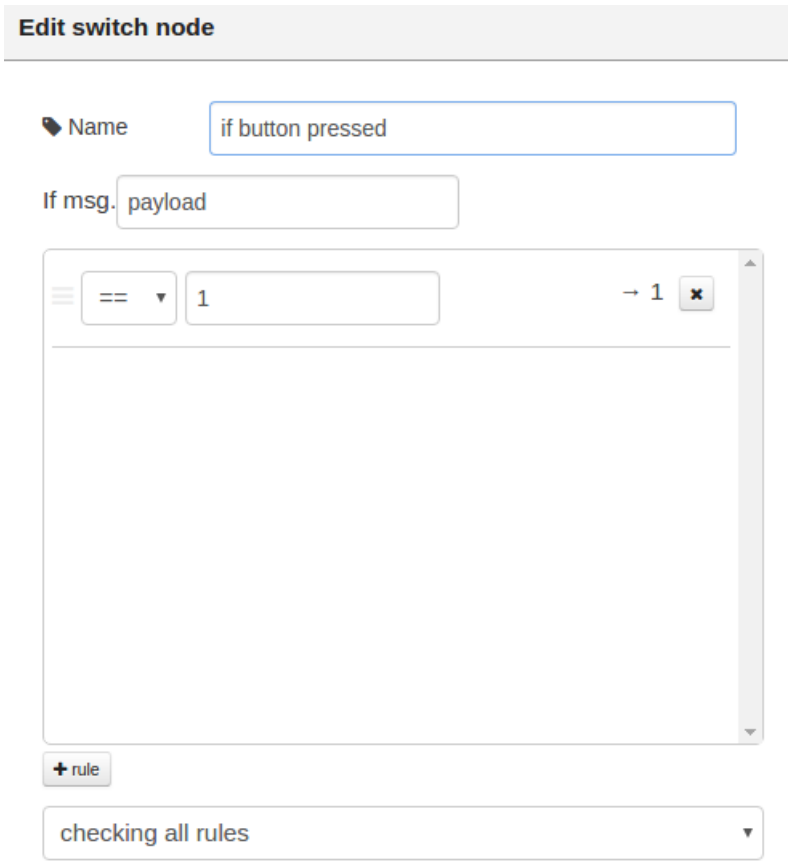


Figure 40: Switch node properties

The *switch* node is then connected to a *trigger* node. This node creates

two messages whenever a message arrives on the input and it outputs the messages separated by a distance specified in the properties. For this project the node allows you to automatically send a message after 30 seconds from the pushed button so that the traffic lights system returns to its initial state. The first message has the payload 1 and the second has the payload 0, thus identifying the states the lights need to get to.

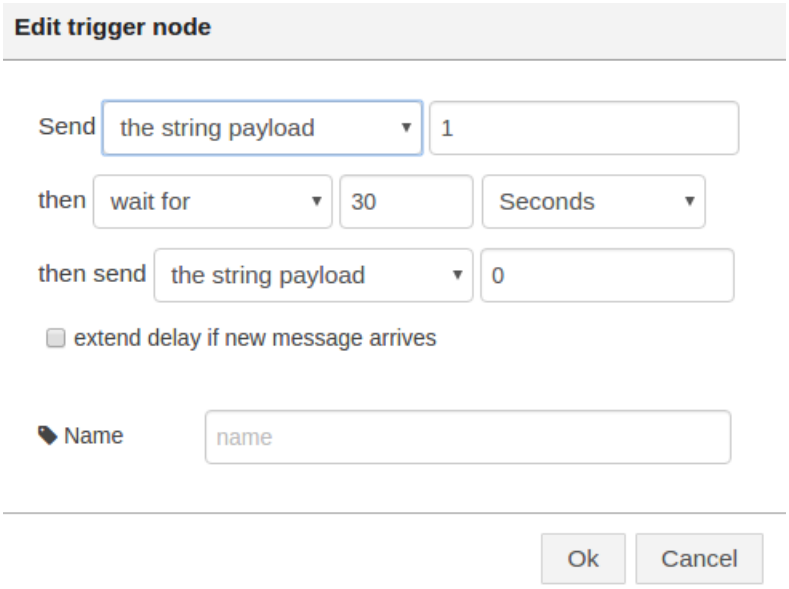


Figure 41: Trigger node properties

Further on, depending on the value the *trigger* node outputs, the *filter input* node will set the lights to the intermediate state and after 5 seconds (*delay* node) it will send another message with the value of final state stored in the payload.

The *filter input* node is also a *switch* node which has two outputs, in this case. The first output is for the value 0 (triggered when the button has just been presses) and the second for the value 1 (triggered when the system needs to get back to its initial state). This way you can differentiate between the two.

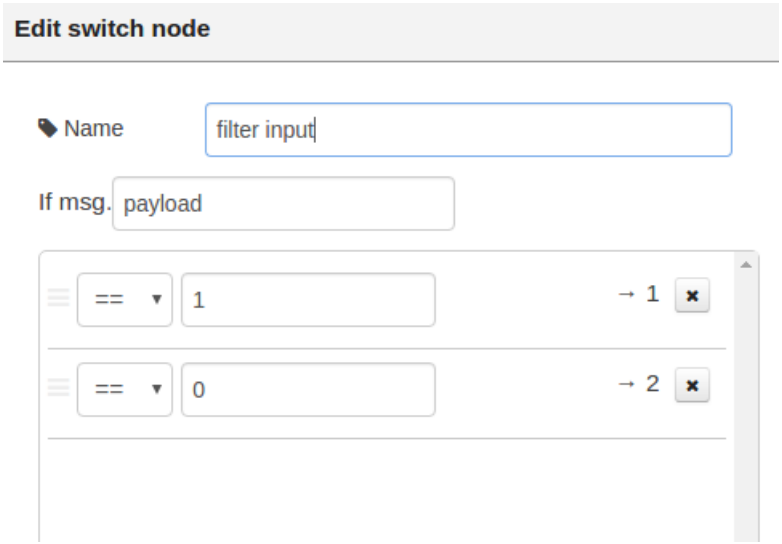


Figure 42: Filter input node properties

You can now run the code and have fun with the traffic lights system you've just build.

Traffic Lights Monitor

Now that you have a functional traffic lights system, let's make it more interesting by allowing you to monitor its state from your computer. In order to do that, you will use the Wyliodrin STUDIO dashboard.

What you need

You will use the setup from chapter *Traffic Lights*.

Dashboard

The dashboard allows you to plot data to graphs. You can drag-and-drop various widgets and use them to display values. It also has a debugging purpose.

Buttons and sliders

You can use these elements to control the peripherals from the dashboard:

- Button - it is basically an on - off switch. In order to use it, you have to specify a name for the signal that the button will send to your

application. When the button is first pressed it sends a signal named accordingly with the value 1 (HIGH) and if you press it again, it will send the same signal, but with the value 0 (LOW).

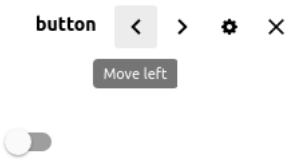


Figure 43: Dashboard button

Inside the application you need to use the *receive signal* node in order to get the signal from the button. The signal's value is stored inside the message's payload.

- Push button - it is a *button* widget that has the *push* parameter selected. When you press it, it sends a signal with the value 1, it waits an amount of time and then it sends the value 0.

- Slider - it acts like a button, but sends a signal with a value from the specified range. Each time you move the slider it sends its new value to the application.

You can send a signal back to the slider/ button in order to change its value. You can add more sliders/ buttons with the same properties.

Graphs

Graphs receive signals from the application and plot the values. There multiple types of graphs you can choose from:

- Line - it is the most complex graph available. You can display more than 1 signal on the graph and you can choose a color for each. The

graph has a name, you can display the legend or choose to have fixed axes (by default it will be relative axes). For logarithmic sensors you can use the logarithmic axes that will display the logarithm of the value received. It can be a time or a point series. The time series have on the x-axis the time stamp, so if you need to plot an array you need to unchecked the option.

It also has a minimum and maximum value to be displayed and you can choose the number of points to be displayed at once. By default the graph will display 10 point and then is going to shift it, if you want to see more you need to check the related option. Between two points it assumes that the signal is a straight line.

- Step line - it is suitable for digital signals. For example to sample a clock, a button. It has the same properties as the line. The period where the signal is horizontal we don't know the value of the signal, we suppose it stays fixed.

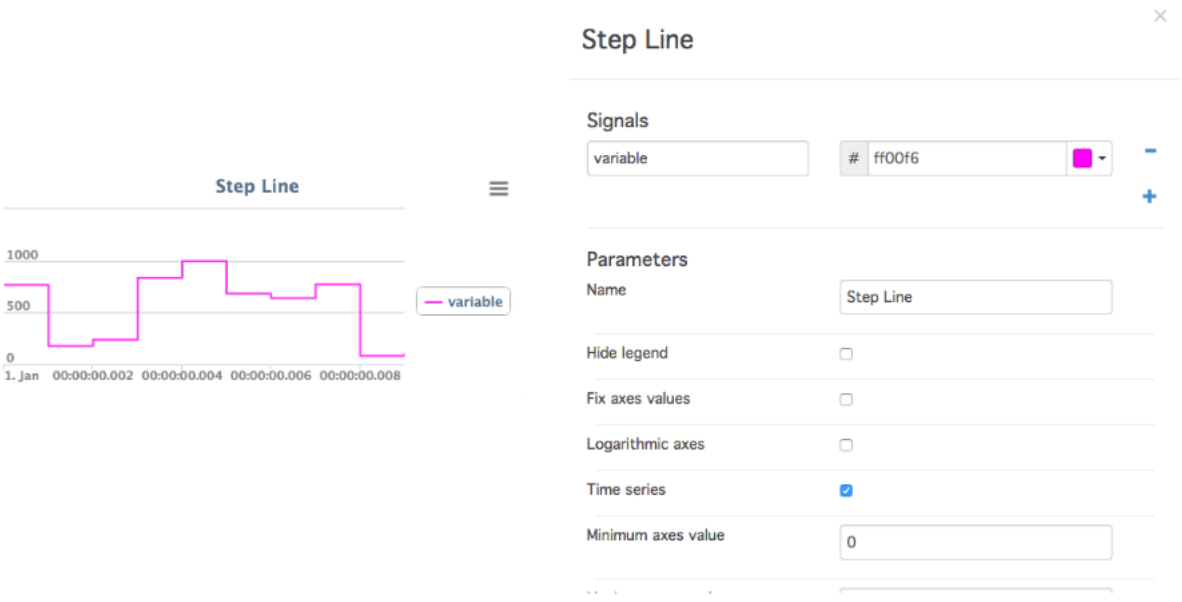


Figure 44: Step line

- Spline line - it is a normal line but the interpolation is not a straight line between two points, it is a *Bézier curve*. At the spline line is not very clear where it's sampled, it tries to give you an idea about how the signal might look (fitted curve).

- Points marker - it displays only the points, showing the real signal (only where it was sampled). It can be logarithmic, time series or vector series.

- Bar graph - it is a widget which displays one value of the signal (it goes up or down depending on the last value it receives). It can display more signal, and for each signal, it displays the last value.

  Other widgets that work exactly like the bar: solid gauge, VU meter, thermometer.

- Extra widget - it displays a picture depending on the value of the signal. In the edit box put on line 0 a link to the picture you want to display for the signal value 0, put on line 1 a link to the picture for the value 1 and so on.

  The picture needs to be HTTPS, otherwise the browser will block it! The simplest way to do it is to use *Dropbox*. You need to insert *dl* in the link instead of WWW, as it will download the file.

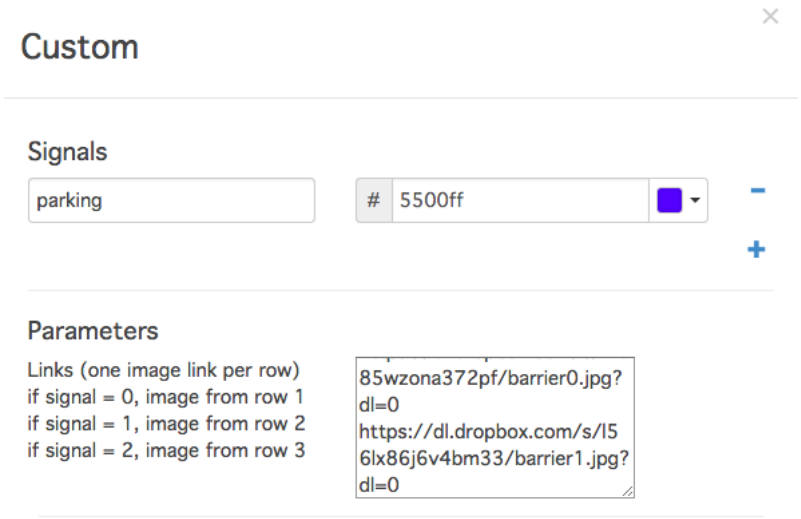


Figure 45: Extra widget

Sending Signals

In the streams programming there is a node called *Send signal* which will send by default a time series (it attaches the time-stamp to the value and sends it to the dashboard). It can send a payload consisting of single point or an array. The signal can also have an x component attached to the value. In that case, it will not be considered a time-series. If you have a number and you want to send the signal with an x component, you can use the *set x value* node.

Once the signal gets to the dashboard, any graph that is set to receive that signal will plot the value.

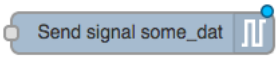


Figure 46: Send Signal Node

The Code

You will start with the application you built in chapter *Traffic Lights*, to which you need to add the *send signal* nodes.

First of all, you need to build the dashboard. You will monitor the lights system by displaying different pictures representing them. This is why you will use the *extra widget* in Wyliodrin STUDIO dashboard. You will need two widgets, one for the cars lights and another for the pedestrians.

Let's start with the cars. Go to the *Dashboard* tab and select the *extra widget*. The configuration pop-up will appear. Set the signal name to *cars* and in the text area put the links to three images:

- on the first line put an image of traffic lights with the red light on;
- on the second line put an image of traffic lights with the yellow light on;
- on the third line put an image oftraffic lighhts with the green light on.

Figure 47: Cars extra graph settings

You can use the following pictures:

- red light on: https://dl.dropbox.com/s/6xl3vmpk8p4wgfc/trafic-lights-red.png?dl=0
- yellow light on: https://dl.dropbox.com/s/5mg4wecyi5xi6md/trafic-lights-yellow.png?dl=0
- green light on: https://dl.dropbox.com/s/83d90kghg0d53k4/trafic-lights-green.png?dl=0

Do the same thing for the pedestrians semaphore, but replace the signal name with *pedestrians* and use these two pictures:

- red light on: https://dl.dropbox.com/s/sjr52ohr8pxb9he/pedestrian-lights-red.png?dl=0
- green light on: https://dl.dropbox.com/s/m1ritf34ethvgt2/pedestrian-lights-green.png?dl=0

Now that the dashboard is set up, you need to send the two signals from the application.

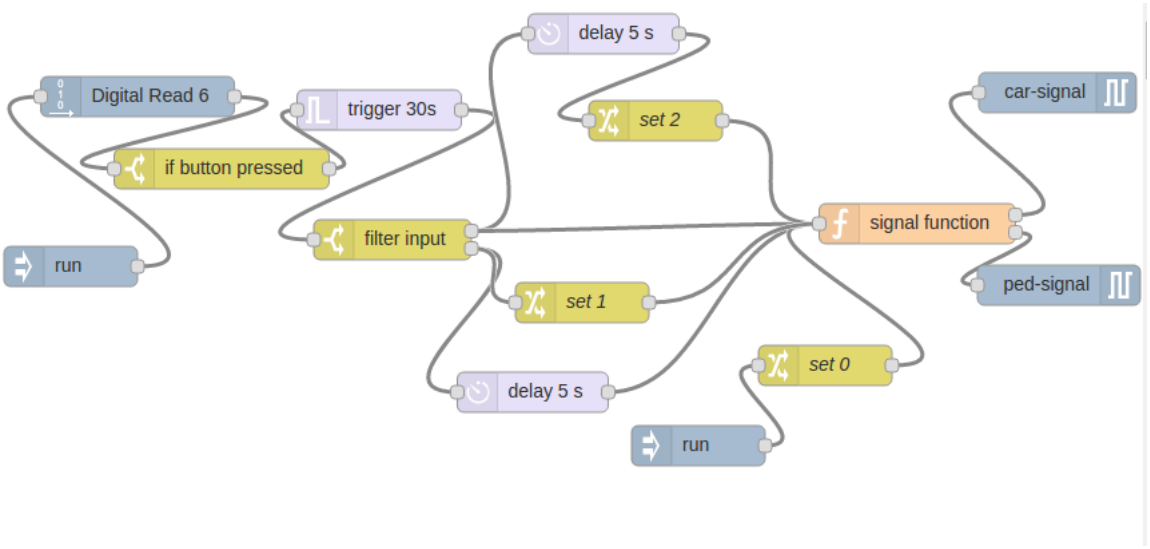


Figure 48: Send signal blocks added

Picture 48 displays the new application. In order to be make the changes visible, the *semaphore* node has been disconnected. However, you don't need to do that, just add the two new nodes and the connections.

Do not disconnect the semaphore *node!*

The two *send signal* nodes are connected to a *function* node similar to the one computing the values to be written on pins in order to light on and off the LEDs. First outputs sends the values corresponding to the *cars* signal and the second one sends the ones corresponding to the *pedestrians* signal.

The *send signal* node will send received values to the dashboard and the

appropriate images will appear. In order to do that you need to double click the nodes and for the *cars-signal* node write *cars* in the *signal* input and *pedestrians* for the other node. These are the names of the signals to be sent to the dashboard.

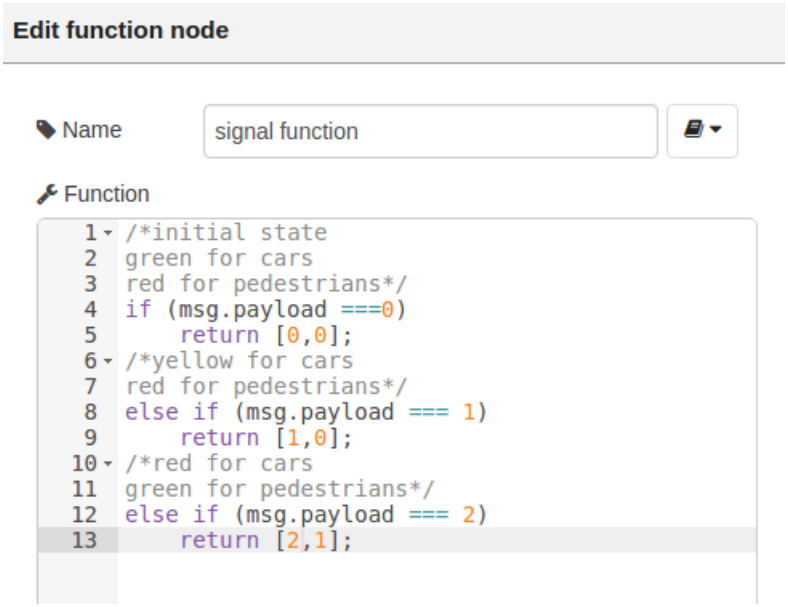


Figure 49: Send signal function

Tips & Tricks

You can use Dropbox to upload the pictures, then get the picture's links and add the links to the *extra* graph. Pay attention to the link, you need to replace the *www* with *dl* (eg. https://dl.dropbox.com/s/6xl3vmpk8p4wgfc/trafic-lights-red.png?dl=0).

Run the application.

Monitoring the Environment

Now that you've built a traffic lights system, you can improve the traffic by monitoring the environment as drivers need to adapt to weather and light conditions.

For the following applications, you will read data coming from sensors, which means that you will also need to use the Arduino board. As the Raspberry Pi has no ADCs, you cannot read values coming from analog sensors.

Arduino Setup

What you need

- One Raspberry Pi board connected to Wyliodrin STUDIO ;
- One Arduino board;
- One USB cable.

First of all, you need to understand how the Raspberry Pi and the Arduino will work together and why you need both of them.

As previously stated, the classical Arduino board is basically a micro-controller, which is capable of running one piece of software at once and that has little

processing power and no network connectivity. So you will use the board to gather data from the environment and then pass it on to the Raspberry Pi.

The Raspberry Pi is a computer that is capable of processing data and communicating with other smart devices. For instance, you could visualise the temperature on your smart phone.

The two boards need to be connected in order to send data between them. You can connect them via the USB cable and a serial connection will be established between the two. Once this is done, they can exchange data and the Arduino board can be controlled via the Raspberry Pi. This is done by the *firmata* protocol. The protocol allows the Raspberry Pi to send the Arduino messages in which it requests for a certain action or information and the Arduino will respond accordingly.

In order to implement the protocol, you need to flash the Arduino with the *StandardFirmata* firmware.

In the Wyliodrin STUDIO interface you have two tabs referring to *SOFTWARE* and *FIRMWARE*. So far, you used only the *SOFTWARE* tab, as you wrote applications for the Raspberry Pi solely. Now, select the *FIRMARE* tab and there is where you can write code which will be run on the Arduino. In this case, you will import an existing project.

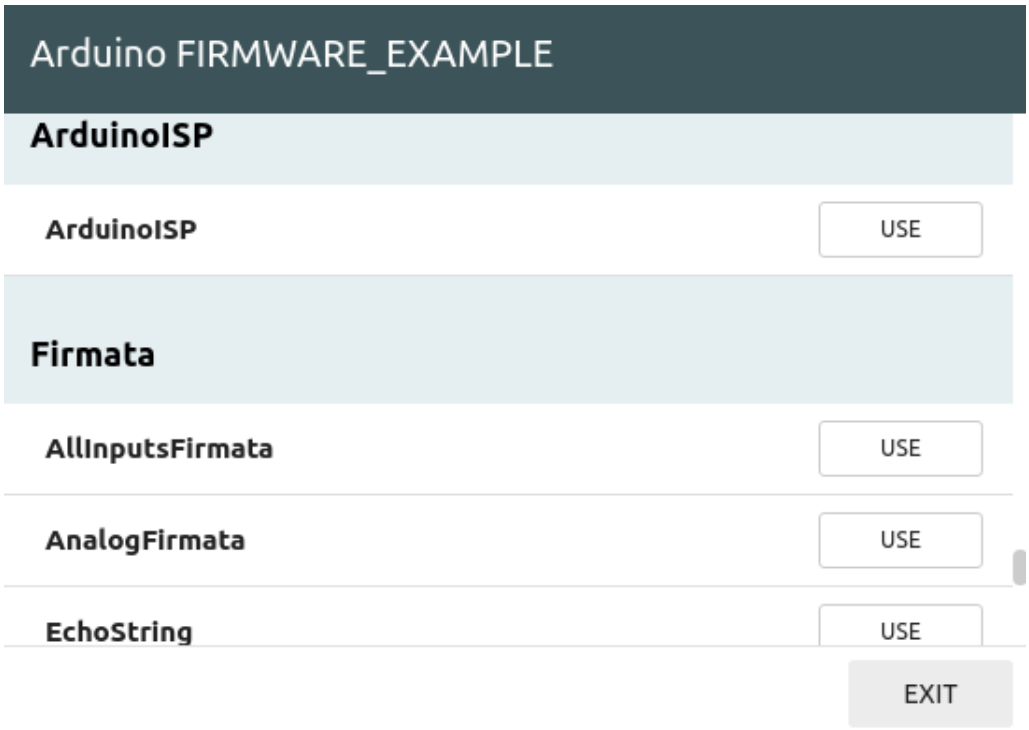


Figure 50: Select firmware box

Hit the *Show examples* button and select *Arduino*. Then *use* the *Firmata/-StandardFirmata* example. Now that you have the software to run on the Arduino, once you hit the *run* button, you will be asked the tyype of the Arduino board and if you want to flash it or not. Select the board you are using and *RUN AND FLASH*. The *StandardFirmata* firmware will be deployed on the Arduino.

As you know, any micro-controller, including the Arduino, once flashed, runs the same firmware until another one is uploaded on the board. Thus, you don't have to flash the board each time you run a new Raspberry Pi application. If you are confident that the Arduino is running *StandardFirmata*, you can skip this step.

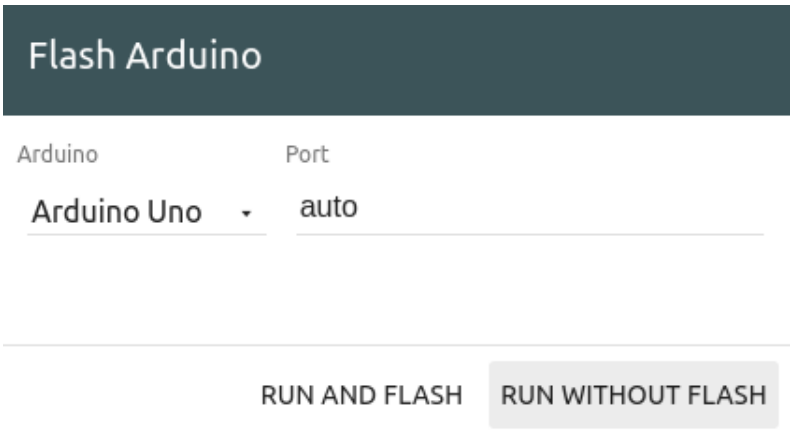


Figure 51: Flash Arduino

Street Lightning

In this application, you will build a system to monitor the light level and if there is the case, you will turn on the street lights brighter or dimmer, depending on the amount of light.

What you need

- One Raspberry Pi connect to Wyliodrin STUDIO ;
- One Arduino connected to the Raspberry Pi;
- One photocell;
- One LED;
- One 220 Ω resistor;
- One 10 kΩ resistor;
- Jumper wires.

The Setup

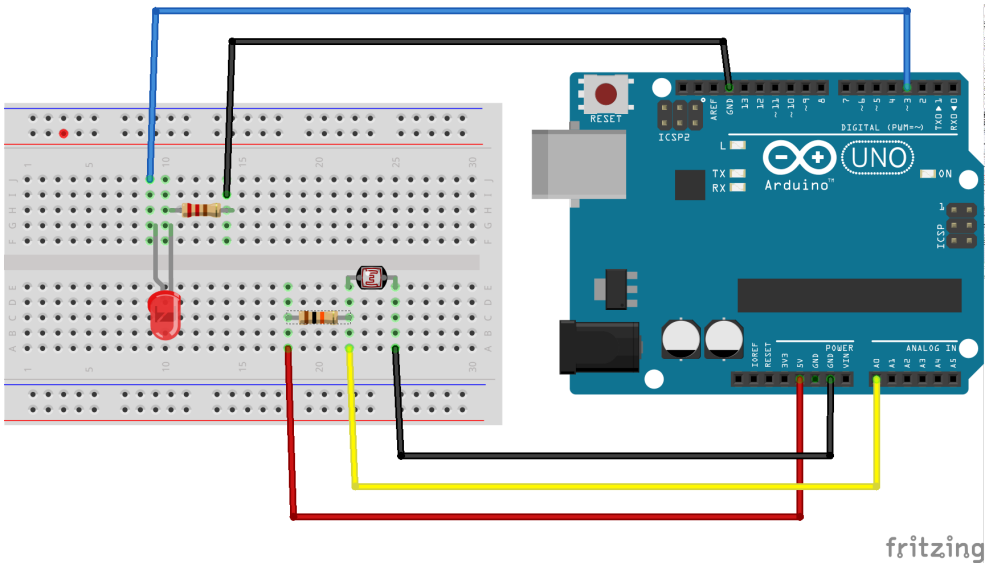


Figure 52: Light sensor schematics

The photocell works just like resistor with a variable resistance. Depending on the amount of light it receives, its resistance gets higher or lower. As a result, you can connect it in a voltage divider, as well (check chapter *Hardware and Electronics for the Internet of Things* for more details) to read its values. Figure 52 represents the schematics.

The sensor is connected to the 5V via the 10 K Ω resistor and to the GND pin on the other side. In order to read the sensor's value, the yellow jumper wire is connected to the A0 pin of the board. The A0-A5 pins can be used for reading digital values, in this case values ranging from 0 to 1024.

For the photocell, the resistance decreases proportionally with the amount of light, so for this schematics, the brighter the environment, the higher is the read value.

In addition, the photocell has a resistance varying from hundred of ohms to mega ohms and in order for the voltage divider to work, you should connect

the sensor to a pull-up resistor, while the actual resistor needs to have a resistance comparable with the sensor's, otherwise its effect is not visible. This is why you need a 10 kΩ resistor.

The other part of the setup, the LED is connected similar to the previous schematics except that you can notice that its behaviour is controller by pin 3. That pin has a tilde next to it. That means that it is a PWM pin (check chapter *Hardware and Electronics for the Internet of Things* for more details) so you can control if the LED should light up brighter or dimmer. You can write values ranging from 0 to 255 on these pins, 0 being the equivalent of digital 0 and 255 the equivalent of digital 1.

The Code

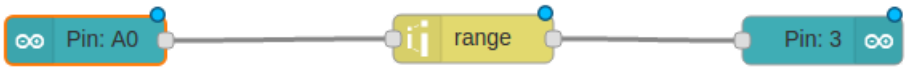


Figure 53: Street lightning code

As you can see in figure 53, the code is very simple.

First of all, you need to use an *Arduino in* node in order to read the values from the sensor. This node gets triggered each time the value on the pin it reads from changes. To actually configure the node's behaviour you need to set its properties.

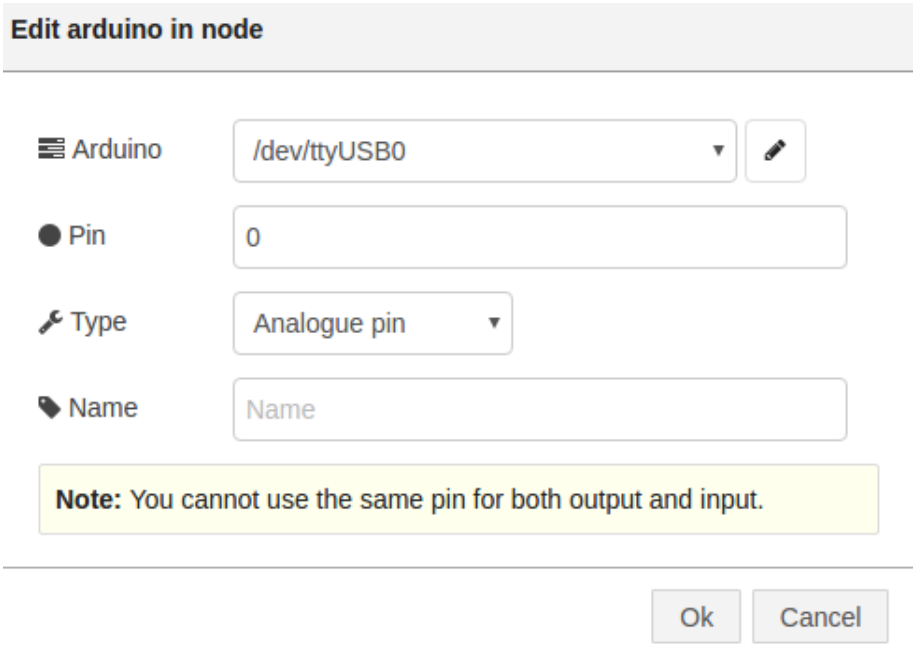


Figure 54: Arduino in node properties

The first property allows you to set the Raspberry Pi connection to the Arduino board so the Raspberry Pi knows the port through which it should send the messages. Usually, the port is */dev/ttyACM0*. However, in order to make sure, go to the *Shell* tab and type *dmesg*. There you should see messages stating that an FTDI cable has been connected to the board and the port it is connected to. Concatenate that value to */dev/* and place the value as the first property of the node.

Further on, you need to specify which pin you want to read from. In this case, the pin is 0 and is an analog pin. Afterwards, you will use a *range* node in order to map the read values to values that should light up the LED. For this, you need to adapt the system to your environment. Depending on the regular amount of light, usually you will read a value of around 400. So in this case, the range scales values going from 400 to 1023 to 0 to 255. However, if the regular value is different, you will need to scale a different range.

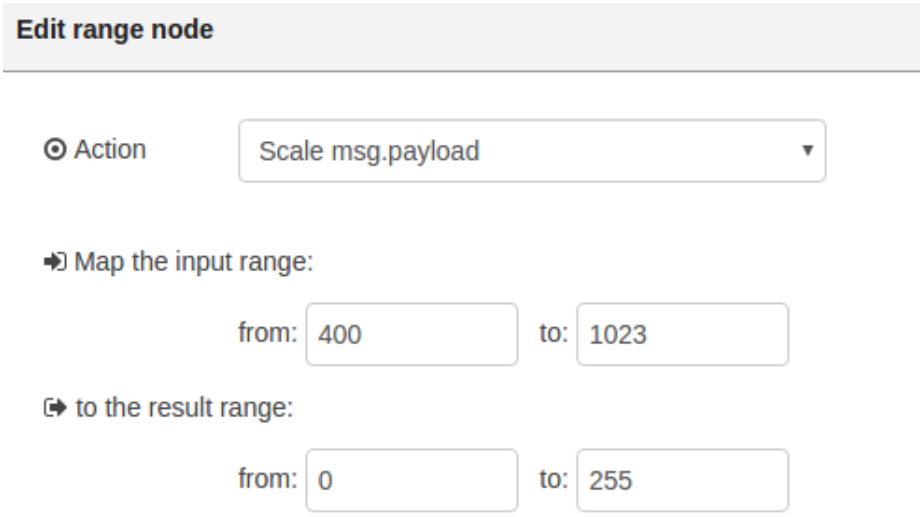


Figure 55: Scaled values

The last node to use is *Arduino out*. This node acts just like the *digital write* node used in the previous chapters. However, you still need to select the port for the Arduino board and you can choose from analog or digital write. As you want to make the LED light up brighter or dimmer, you will set the type to *analog* and the *pin* property has to be 3.

If you use more than one Arduino node in the application, add a new Raspberry Pi board only for one node. For the rest of the nodes, choose the previously added board from the drop-down.

Now you can run the application and watch how the LED gets brighter as you cover up the sensor. You can also add a *gauge* graph to your dashboard and monitor the values coming from the sensor.

Monitoring the Temperature

You will improve the city monitoring system by adding a display to show the temperature so that drivers know if the road gets too hot and slow

down.

What you need

- One Raspberry Pi connect to Wyliodrin STUDIO ;
- One Arduino connected to the Raspberry Pi;
- One temperature sensor (TMP36);
- One 16×2 LCD;
- One potentiometer (optional);
- Jumper wires;
- Breadboard.

The Setup

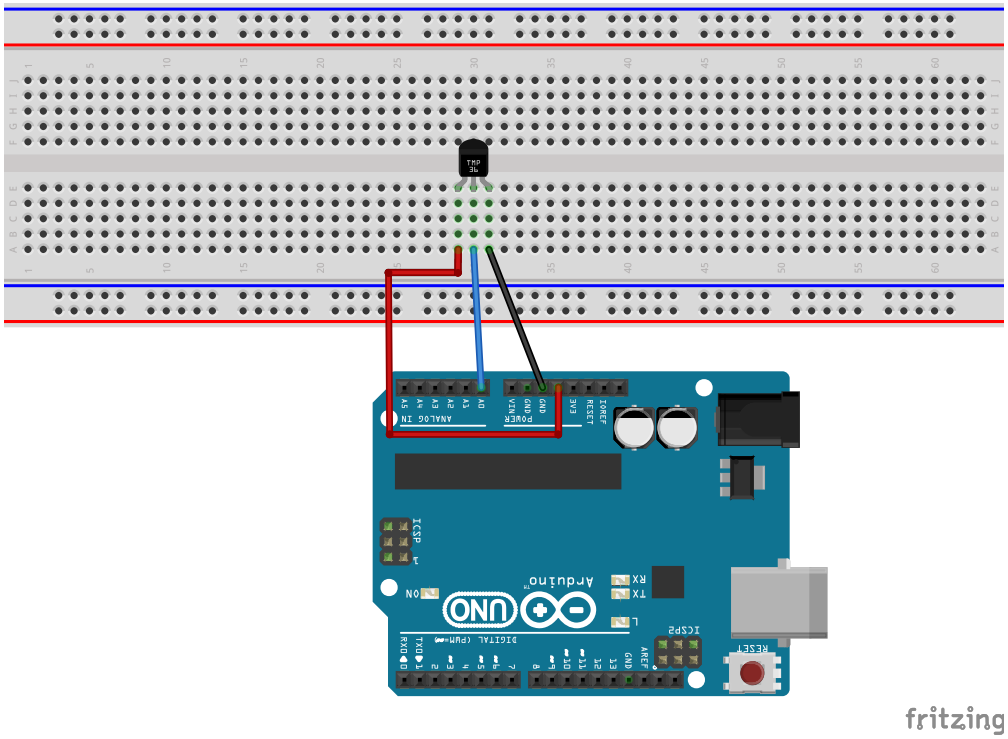


Figure 56: Temperature sensor connection schematics

In figure 56 you can see a 3-leg temperature sensor (TMP35). As stated in chapter *Hardware and Electronics for the Internet of Things*, this sensor has a voltage divider integrated in it. This sensor consists of a thermistor and several other electronic pieces. The leg situated in the middle outputs a different voltage depending on the temperature.

In this case, the sensor needs to be connected to the GND pin, the VCC and an analog pin. Connect the sensor in the breadboard with the flat side down, then connect the left leg to the 5V pin, the right leg to the GND and the middle one to one of the analog pins.

It is very important to connect the sensor exactly like in the schematics,

otherwise you might burn it.

Other temperature sensors might need to be connected in a different way. We recommend you check the sensor's datasheet before connecting it.

The TMP36 datasheet displays figure 57 which clearly depicts how each leg should be connected.

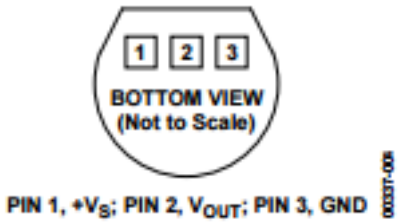


Figure 57: TMP36 datasheet excerpt

Secondly, you need to connect the LCD. We choose to connect the LCD directly to the Raspberry Pi as it only needs digital pins and there is no use to load the Arduino with more messages.

This is an 16×2 LCD, which means that you can you can display data on 16 columns and 2 rows . The LCD has a parallel interface, which means that it exposes several pins which need to be controlled at once in order to control the display:

- *RS* (Register Select) - controls where exactly in the LCD's memory you are writing data to;
- *Enable* - enables writing to the registers;
- *R/W* (Read/Write) - selects the mode (reading or writing);
- *8 data* pins (D0 -D7) - the states of these pins (1 or 0) are the bits that you're reading to a register when you read, or the values you are writing when you write;

- *power supply* pins (5V and GND) - for powering the LCD;
- *Bklt- and Bklt+* (LED Backlight) - turn off and on the LCD's backlight;
- V_o (display contrast) - controls the display contrast.

In order to display data on the LCD you need to put the data to be displayed in the 8 registers (write to pins D0-D7) and then put instructions in the instruction register (R/W).

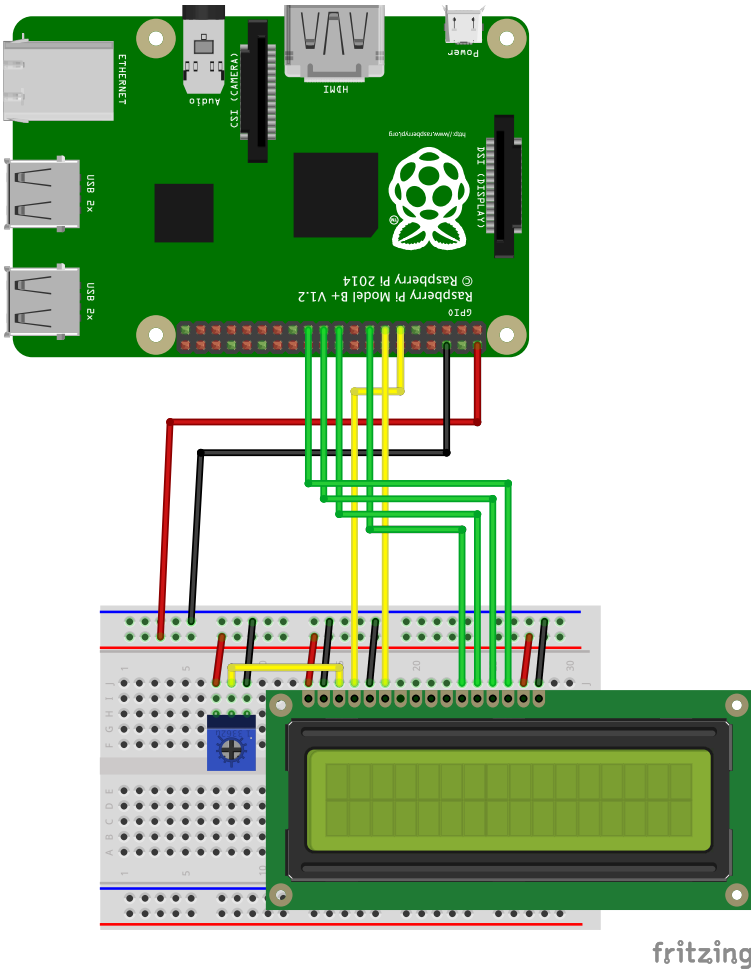


Figure 58: LCD connection schematics

You place the LCD on the breadboard and connect it to the Raspberry Pi as shown in figure 58. You can use the same breadboard you connected the temperature sensor on, just make sure the pins you use do not overlap.

The first two pins on the right are used to power up the LCD. The next four pins, the ones connected by green cables, are the four of the data pins (you won't need the rest). The next two pins connected by yellow cables are the control pins. There are also three pins used for contrast. Two of them are used to power on the backlight and there is one more pin directly connected to the potentiometer, which will output a different voltage depending on its angle. This way you can control the contrast just by rotating it.

Another way of connecting the LCD is to connect it to a port expander (MCP23008). This allows the LCD to be controlled via the I2C protocol, thus reducing the number of pins you need to connect the LCD to (figure 59). In this case, you just need to connect the expander to GND, 5V, SCL and SDA pins, according to the indicators on the expander. You can find the appropriate pins in the *Pin Layout* tab. You just need to connect the SDA port on the port expander with the SDA pin on the board and so on.

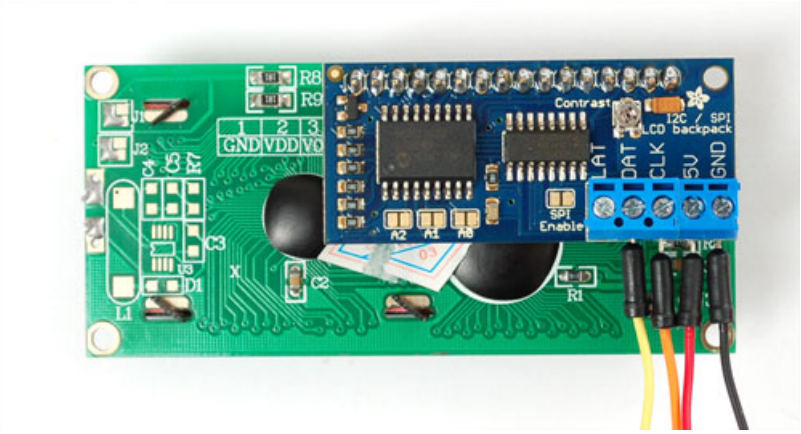


Figure 59: LCD connected to MCP230008 expander

The Code

First of all, let's simply read the temperature from the sensor and display it on a thermometer in the dashboard.

Firstly, open the *Dashboard* tab and select the *thermometer*. Set the *Signal Name* to tempera(figure 59)ture and the maximum value to 50.

Figure 60: Read temperature application

Afterwards, you need to connect the nodes similar to figure 60. In order to get the values from the temperature sensor you will just have to use the *Arduino out* node and read the analog pin. So double click the node and set the node to 0 and select *Analog* for the type. Also remember to set the port for the Arduino board. The values that you obtain are in a range of 0 to 1024. However, this doesn't look at all like the values you would expect from a thermometer.

To transform these values into Celsius degrees, you are going to use this formula.

$$temperature = (value * \frac{5}{1024} - 0.500) * 100 \tag{8}$$

The previous equation is computed in the *compute celsius* node (figure 61).

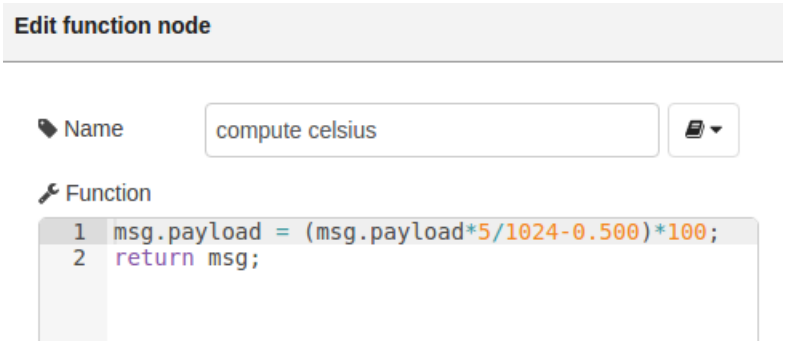


Figure 61: Compute Celsius node code

Lastly, the *send signal* node is used to send the *temperature* signal. For this you only need to set the name of the signal to be send to the dashboard and the message with the computed payload will be sent.

Now run the application and watch the temperature in the dashboard.

Now that you assured that you can obtain the temperature in Celsius degrees, let's display it on the LCD.

Figure 62 depicts the nodes you need to use.

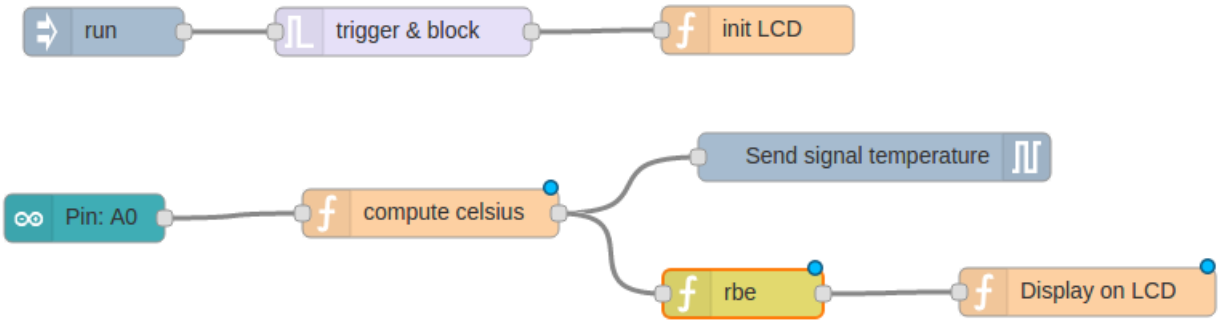


Figure 62: Code for displaying the temperature on LCD

First of all, you need to initialise the LCD. For this, you drag the *run* node and select *Fire once at start* so you make sure that once you run the appli-

cation a first message is sent. This is only required to trigger the *init LCD* node, so you can assure there is no other message being sent again. For this, drag the *trigger* node and double click on it.

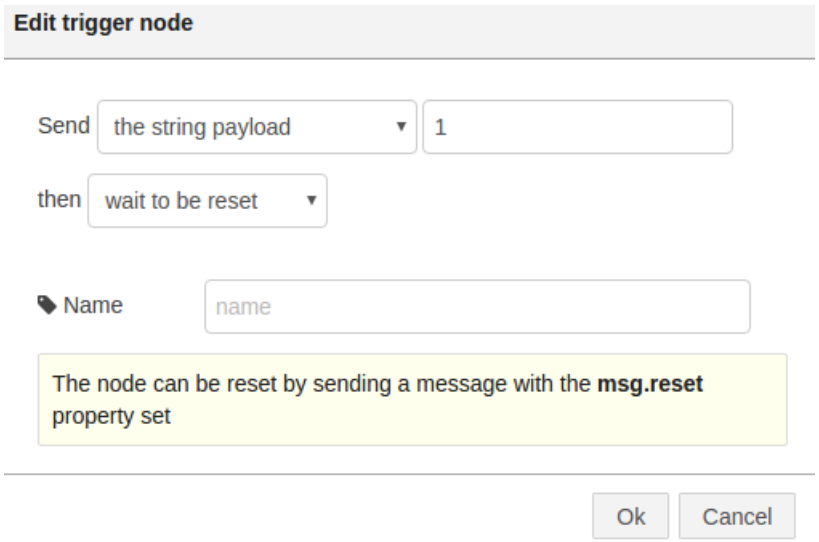


Figure 63: Trigger node properties

You need to set this node to *wait until reset*. This will make the node to send the first message and then wait until it receives a message containing the property *reset*, which will never happen. Next, the *init LCD* node initializes the LCD.

As discussed above, there are two types of LCD and regarding the code, the two are different only by the way they get initialised.

First, picture 64 depicts the code to initialise the LCD that has no expander.

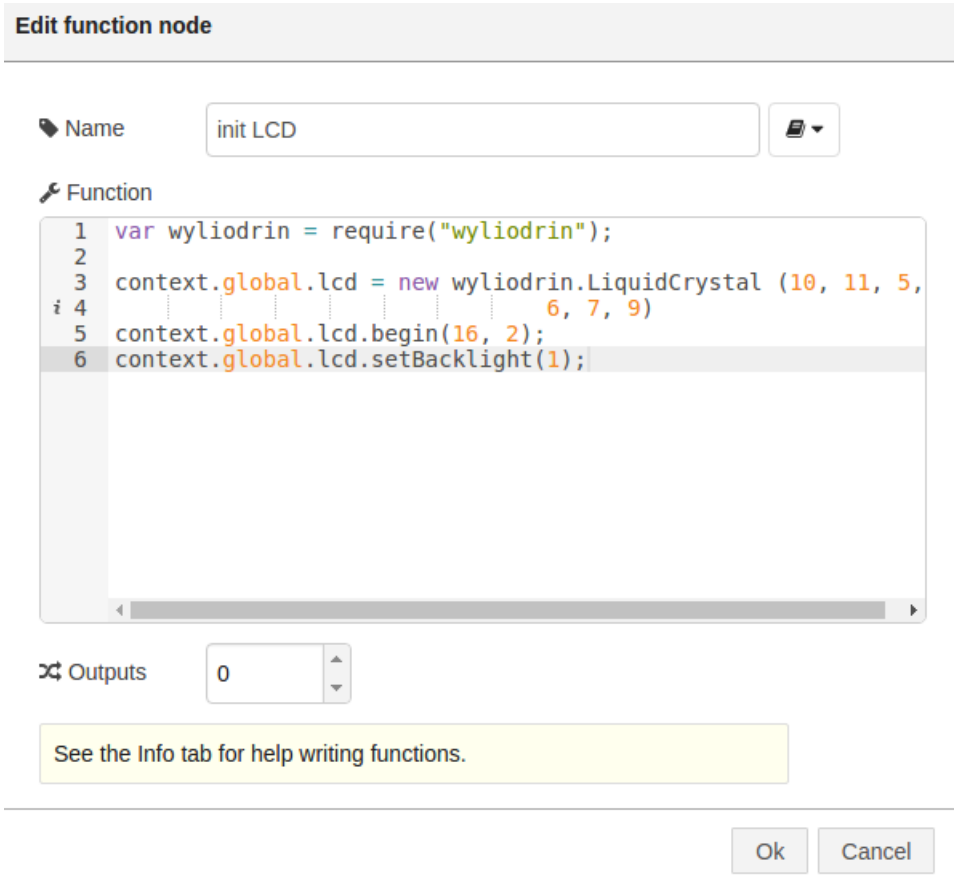


Figure 64: Init LCD without expander function

First of all, you need to require the *wyliodrin* module, which exposes some simple functions in order to control the LCD. Otherwise you would need to implement the protocol in order to control it.

Once you have the *wyliodrin* variable, call the *LiquidCrystal* constructor in order to obtain a *LiquidCrystal* object. The constructor gets as parameters the following pins numbers, in this exact order:

- RS;
- Enable;
- D5, D6, D7, D8 (the four data pins).

Afterwards, call the *begin* function and the LCD is ready to be used. You can also call *setBacklight* with the parameter 1 in order to turn on the backlight.

On the other hand, for an LCD that has the MCP23008 expander connected to it, you simply need to call the constructor with the parameter 0 (figure 65). The rest remains the same.

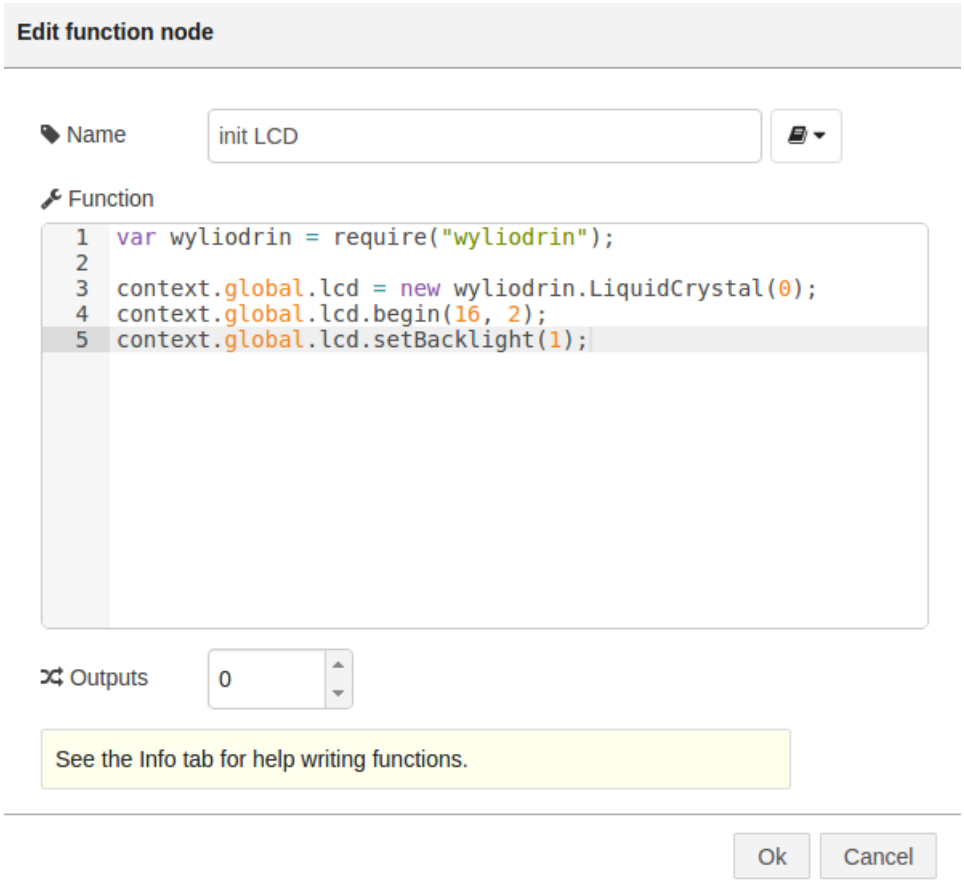


Figure 65: Init LCD function

Why do you need to set 0 as parameter? This type of LCD communicates with the Raspberry Pi board via that I2C protocol. Taking into account how the I2C protocol behaves, you would need to mention the bus specific to the LCD, that usually is 32. However, in order to be able to connect multiple

LCDs at once, they have three jumpers which can be connected in order to represent values from 0 to 7, thus being able to identify eight different LCDs. This is the value you need to pass to the constructor, in this case, no jumper is connected so we passed the value 0.

What is new at the code presented above is the *context.global.lcd* structure. You need to use this because you want the *lcd* variable to be available from other nodes also (Display on LCD node) and *context.global* is a variable accessible to all the nodes, so you just set the lcd property and you will be able to use it from any other node.

Now that the LCD is set up, you can use it to display the temperature. However, if the values change too quickly, the LCD will change the text to fast, and if won't be readable. Thus, you have to use the *rbe* node which sends the received message only if the payload value changes with a certain amount. In this case, a tolerance of 3% is sufficient (figure 66).

Figure 66: rbe node properties

This is followed by the *Display on LCD* node. This node receives the message with the temperature value, clears the LCD and calls the *print* function on the *lcd* global variable. The value to be printed is converted to a string keeping only two decimals, using the *toFixed(2)* function. You cannot print a number on the LCD, it needs to be a string value.

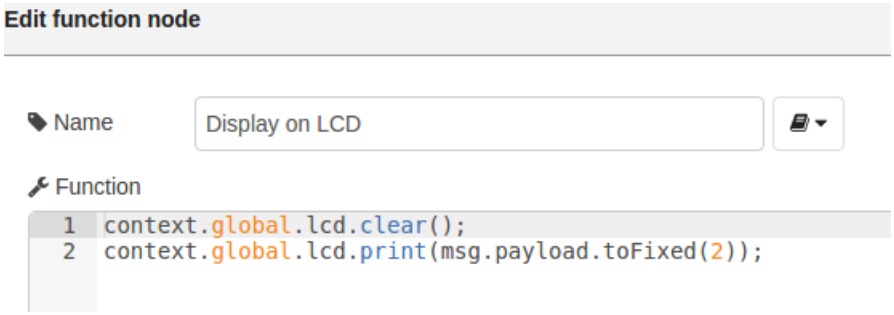


Figure 67: Display on LCD function

Run the project and the temperature will appear on the LCD.

Measuring the Exhaust Gas Level

To make the environment monitoring system complete, in this project you will measure the exhaust gas levels and if they are too high, you will decrease the allowed maximum speed.

What you need

- One Raspberry Pi connect to Wyliodrin STUDIO ;
- One Arduino connected to the Raspberry Pi;
- One gas sensor (MQ2);
- One 16×2 LCD;
- Jumper wires;
- Breadboard.

The Setup

The schematics is similar with the previous one, you just need to replace the temperature sensor with the gas sensor.

In this case, the sensor's support for jumpers has marked where each pin should be connected. You simply need to connect the GND on the sensor with the GND on the board, the VCC to a 5V pin on the board and the OUT to any analog pin of the Arduino.

Figure 68: MQ2 gas sensor

These sensors are for prototyping use only. We recommend you not to use them in production.

Pay great attention when handling the sensor. It gets hot after functioning for some time.

The Code

The code is also similar to the previous one.

Figure 69: Code for reading the gas level

The difference is that you can now connect the *Arduino in* node directly to the *rbe* node then to *Display LCD* function node. The code inside the function is depicted in figure 71.

The *report if changed* node allows the message to pass only if the received value has a difference of at least 50 from the previous one (figure 70).

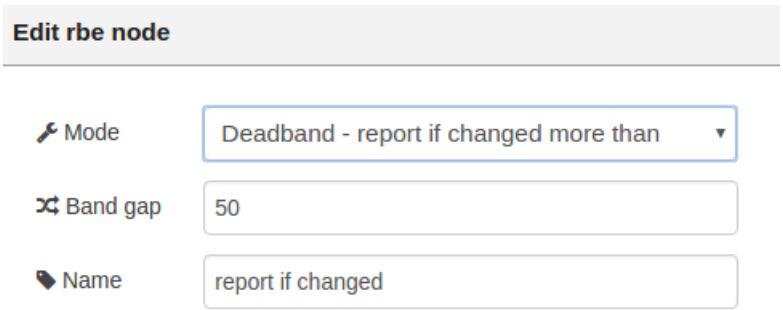


Figure 70: Report if changed node

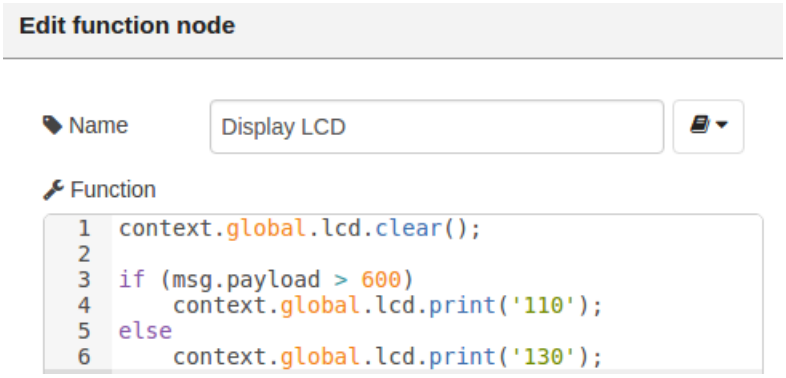


Figure 71: Display LCD function

The function verifies the received value and displays a different speed limit if the sensors returns a value above 600 or below 600.

Run the application.

We leave it up to you to create a weather station that displays on the LCD the temperature, exhaust levels and the light level. You can select which of the three values to be displayed by pressing a button.

Traffic Counter

A nice feature any smart city system should have is a traffic counter. In this project you will use a Hall sensor to count the number of cars passing through a certain spot during one day.

In a real smart city system, a coil is used. The coil changes its properties when a car passes by. However, in this project, as it is at a smaller scale, you will use a Hall sensor and a magnet to simulate the cars passing by.

What you need

- One Raspberry Pi connect to Wyliodrin STUDIO ;
- One Hall sensor (SS441A);
- One 220 Ω resistor;
- Jumper wires;
- Breadboard.

The Setup

The SS441A sensor is a device that reacts to magnetic field. Its behaviour can be compared to the one of a push button: if any magnetic object approaches the sensor, it acts like a pressed button, while in normal state, the connection is open (button is not pressed). However, there are sensors which work the other way around, such as the one used in this project, which outputs HIGH in the normal state and LOW when a magnet is near it. Thus, the sensor should be connected similar to a button. However, just like the temperature sensor, the voltage divider is integrated in the sensor, exposing 3 legs (figure 72).

Figure 72: SS441A sensor

Insert the sensor in the breadboard and with the smaller side facing you, then connect the left leg to the 3.3V pin, the middle one to the GND and the right leg to a GPIO pin of the Raspberry Pi.

You also need to connect the resistor between the 3.3V pin and the GPIO pin, similar to figure 73.

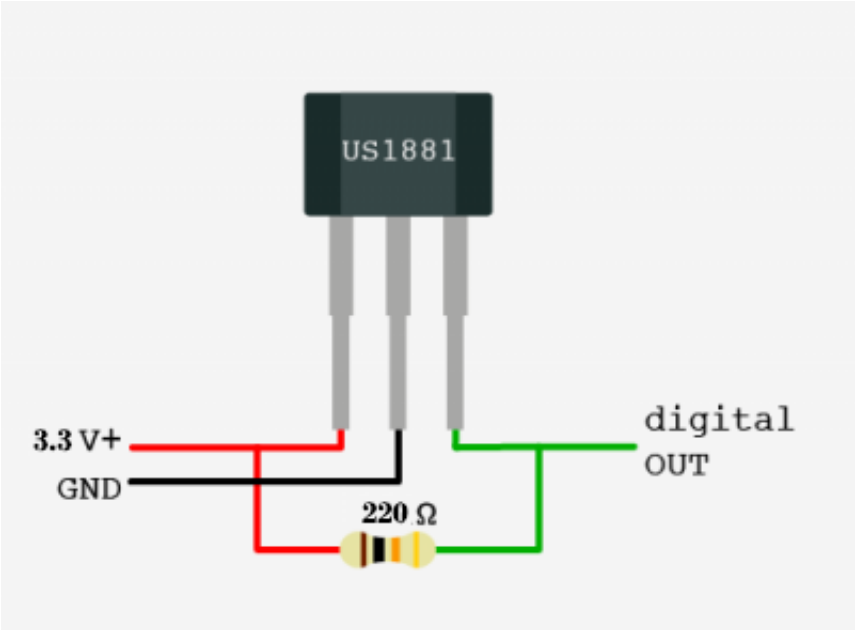


Figure 73: Hall sensor schematics

Pay attention to how you connect the sensor. If you connect it wrong you risk burning it. If you are not sure how to connect the sensor, search its data-sheet and follow the schematics.

The Code

You will display the number of cars in the dashboard, on a line graph. So, the first step in building this project, is to go to the *Dashboard* tab. There you have to choose the *line* and set the signal name to *counter*. Also, mark the *Fix axes values* option, otherwise it will be more difficult to notice the changes. You can also alter the *Maximum axes value* depending on the maximum number of cars you expect to pass in one day.

Now, let's go to the *Application* tab and check the code (figure 74).

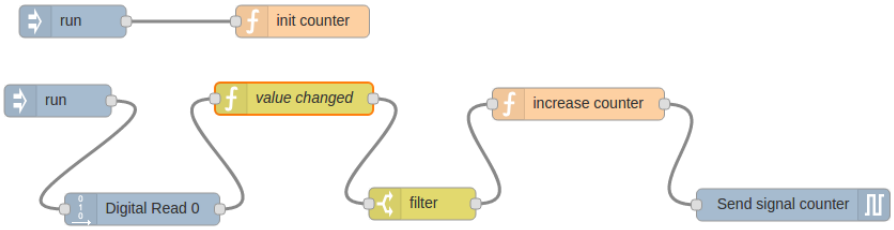


Figure 74: Traffic counter code

First of all, you have a *run* node connected to an *init counter* function node. The *run* node starts once the application start running, so you need to check the *Fire once at start* option and you also need to make it send a message every 24 hours. This way, the *init counter* node gets activated only once a day.

The *init counter* node initializes the counter to 0 (figure 75). Notice that you need to attach the counter to *context.global*, as you need to access it from another block in order to increment its value;

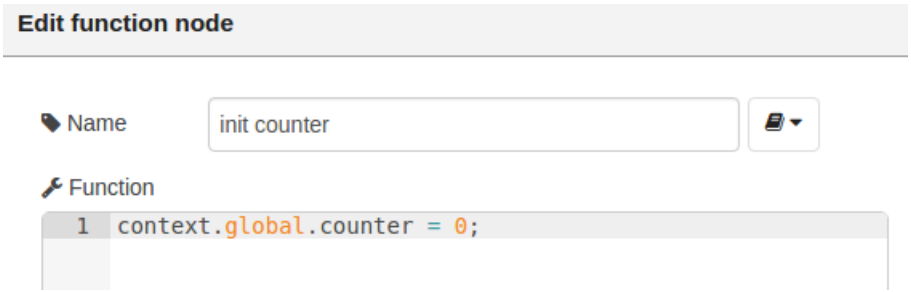


Figure 75: Init counter function

You need to set the other *run* node to send a message every 0.1 second and activate the *digital read* node, which reads the value coming from the pin the sensor is connected to. You are interested only when a car passes next to the sensor, which reduces to the event when the read value changes from 1 to 0. This is why you need to use the *rbe* node which forwards the message only when the received payload is different from the previous one. The result is

that one message will be transmitted when a car approaches the sensor and the next message will appear when the car passed the sensor.

Similar to the traffic lights system, you are only interested to do something when the sensor has a magnet near it (the value read is 0). This is why you need to use the *filter* node.

You have to double click the *filter* node and make it pass on the message only if the payload is equal to 0 (if msg.payload == 0).

The next node is *increase counter*, which implements the function that increments the *counter* variable (figure 76), then sends the message to the *send singal* node.

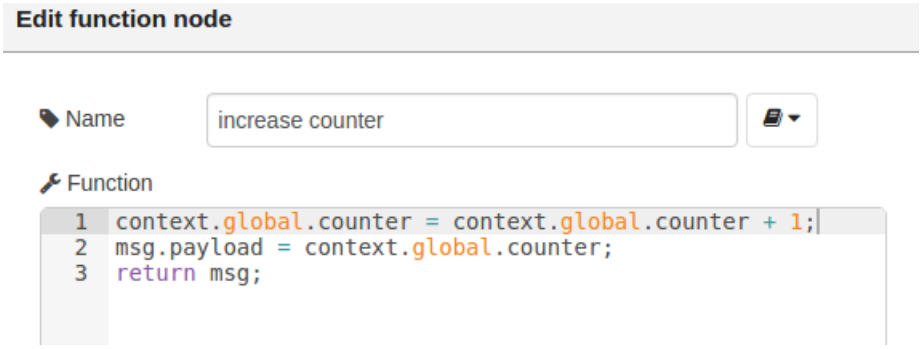


Figure 76: Increase counter function

For the *send signal* node you simply have to set the *signal name* to *counter*.

Run the project and you can see the number of cars passing.

License Plate Recognition

This tutorial will guide you through building a license plate recognition system by using a web service.

What you need

- One Raspberry Pi connected to Wyliodrin STUDIO ;
- One Pi Camera;
- One LCD;
- One Hall sensor (SS441A);
- Jumper wires;
- Breadboard.

The Setup

First of all, you need to connect the Pi Camera. The Raspberry Pi board has a port especially designed to connect the camera on. Figure 77 depicts how to connect it. You need to place the part with the connections towards the thinner side of the port.

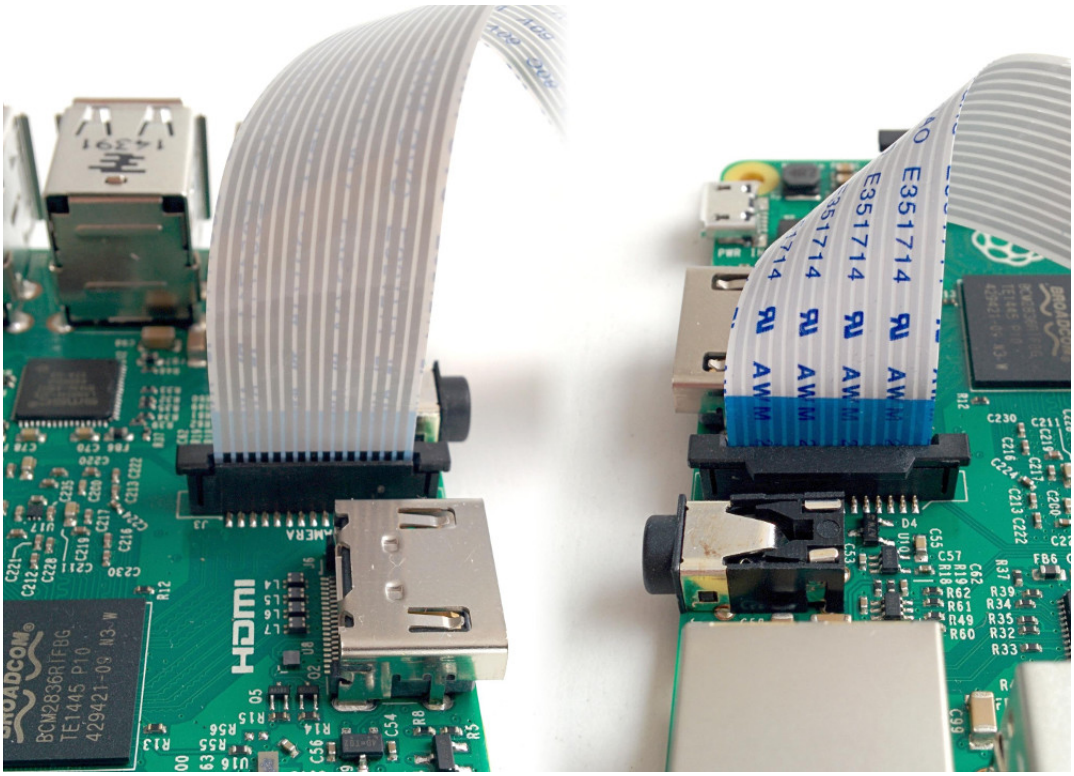


Figure 77: Pi camera connected to a Raspberry Pi

In addition, you have to connect the LCD to the Raspberry Pi (see chapter *Monitoring the Environment*).

Creating an openALPR Account

For this application, you will use a web service in order to extract the license plate number from the pictures taken with the Pi Camera.

The web service is *openALPR* and it allows you to upload pictures and get details such as the car model, colour, license plate and other information. In order to use the service, you need to create an account.

Go to `www.openalpr.com` and create an account. You only need a valid email

address to do that. You will be sent an email to confirm the email address and you can log in and use the service.

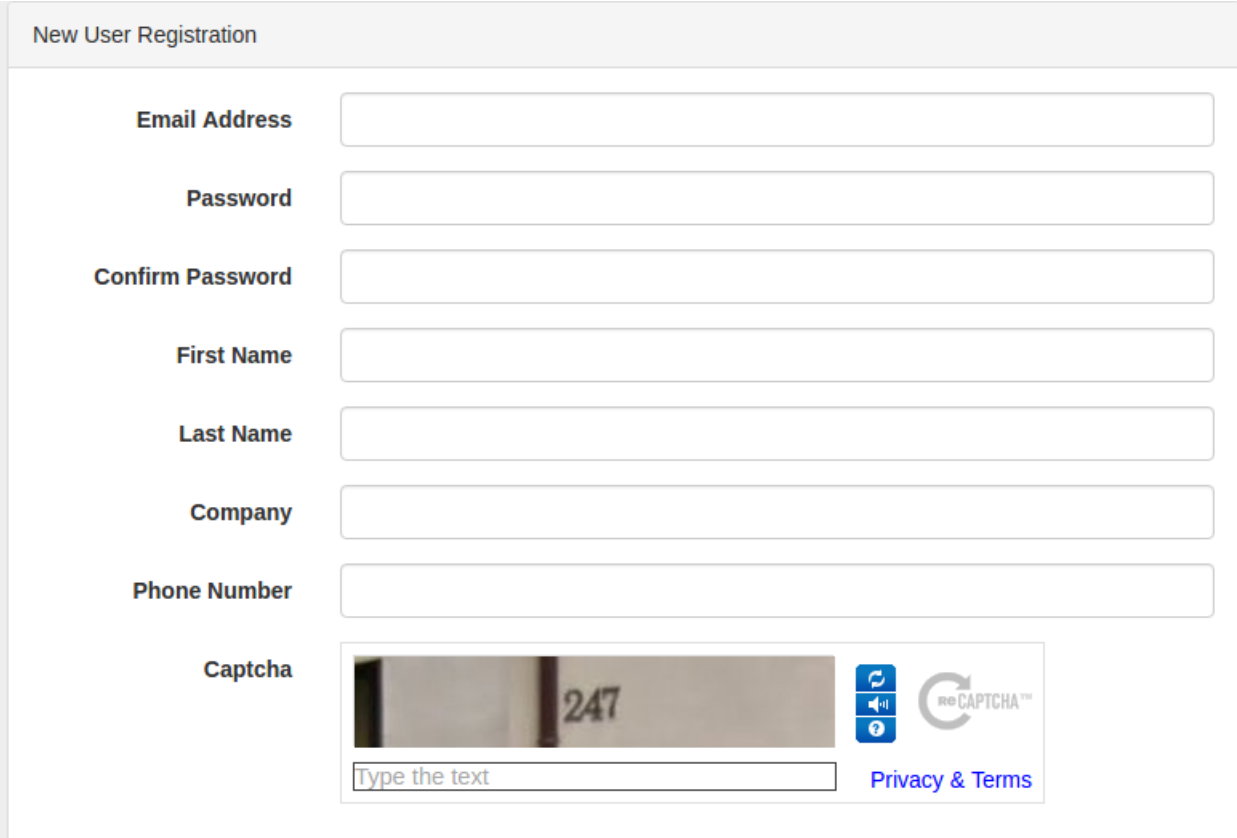


Figure 78: openALPR new account

Once you logged in, go to *Cloud API* and check your Cloud API credentials. You will need the *secret key* later on.

The Code

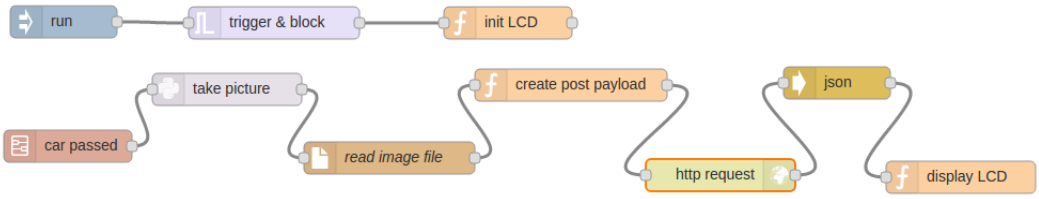


Figure 79: License plate recognition code

Figure 79 depicts the code that you need to build in order to create the license plate recognition system. The application is based on the previous one, that counts the cars passing. Now, each time a car passes, the camera takes a picture of it and displays its license plate number on the LCD.

First of all, you need to initialise the LCD (for details on how to do that see chapter *Monitoring the Environment*).

The next flow of nodes starts with the *car passed* node. That node is actually a subflow element. To create the node, go to the options menu in the top right of the workspace and select *Subflows/Create Subflow*. Once the subflow is created, select the *edit properties* option and name it *car passed*. Also set the number of outputs to 1.

The code you have to insert in the subflow is the same with the code you created in the previous chapter, except the counter (figure 80) which is missing. We also recommend to increase the delay to 8 seconds, as cars will move slowly in front of the camera. Each time a car passes, the *car passed* node outputs a message.

You can copy and paste connected nodes from one application to another. Select the nodes you wish to copy, go to menu and select *export*. Copy the text and in the other application, go to menu, select *import* and paste the

text.

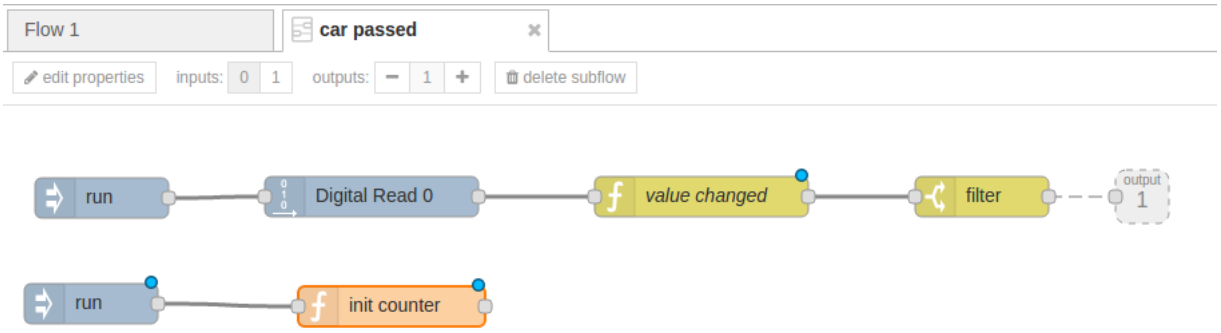


Figure 80: Subflow code

The node connected to *car passed* is a *python* node. It is similar to the *function* node, but it allows you to write code in Python instead of JavaScript. You can follow the necessary code to take a picture using the Pi Camera in figure 81.

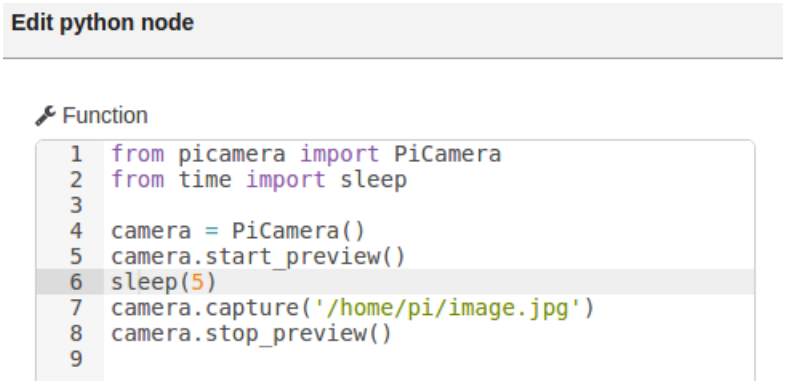


Figure 81: Python code for taking a picture

As you can notice, the piece of code uses the *PiCamera* module. However, this module is not installed on the board. In order to install it go to the *Shell* tab and type the following command:

sudo pip install picamera

You will be asked to insert the user's password. By default you are logged in with the user *pi*, which has the password: *raspberry*.

The python code initializes the connection to the Pi Camera, waits for five seconds to finish initializing and calibrating the camera, then it takes the picture and stores it as */home/pi/image.jpg*.

Once the picture is saved, you need to send it to a web service that will process it and extract the license plate. The *read image file* node parses the file and sends further on a message storing the file's contents in a buffer(figure 82).

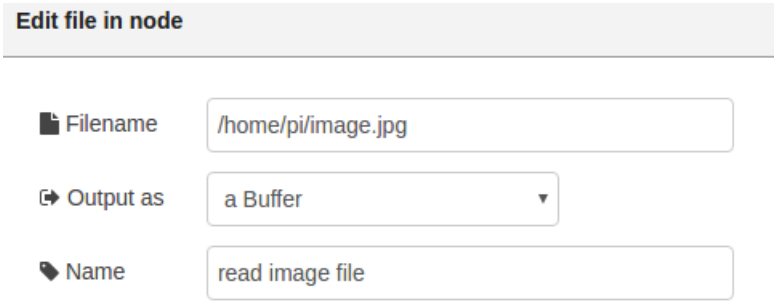


Figure 82: Read image file node

Further on, you will send a POST request to *openalpr*. In order to do that, you need to create the payload of the POST request. This is what the *create post payload* node does (figure 83).

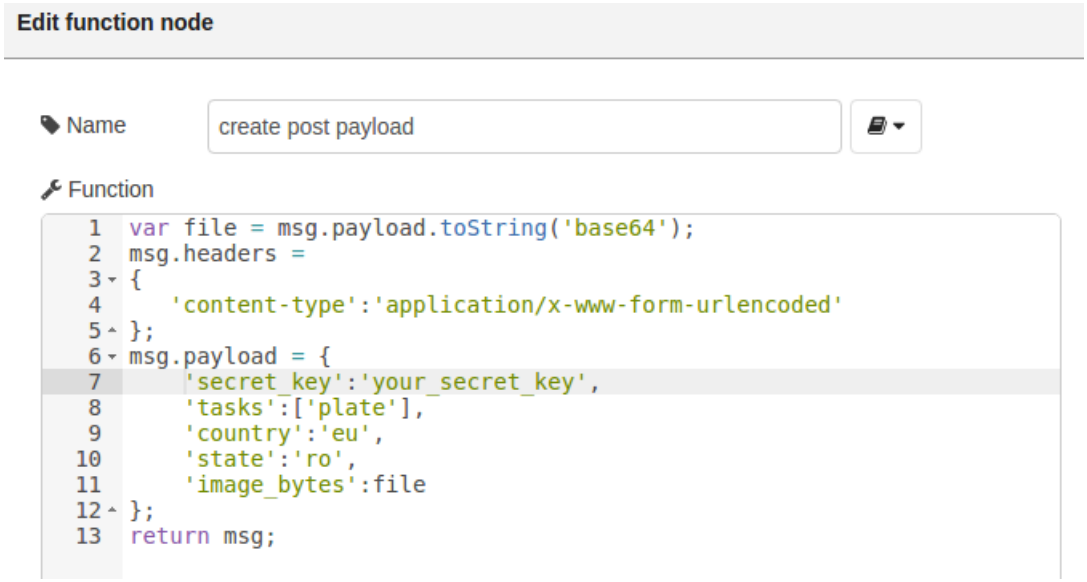


Figure 83: Create post payload node

First of all, you need to store the payload of the received message in a variable, as that is the image file. The format of the string needs to be *base64*. Then, you can store a new payload consisting of a JSON structure. The *secret_key* is the one the openALPR website generates once you create your account. *tasks* asks for a list of processing tasks you need the service to do. In your case you only need to extract the license plate number. To make it easier to recognize, you can specify the area the license plate is from. In this case, the service needs to recognize plates from Europe, Romania. A full list of supported country codes can be found at `https://github.com/openalpr/openalpr/tree/master/runtime_data/config`.

Once the payload is created, you send it to the *http request* node, which will do a POST request at the following address: `https://api.openalpr.com/v1/recognize`.

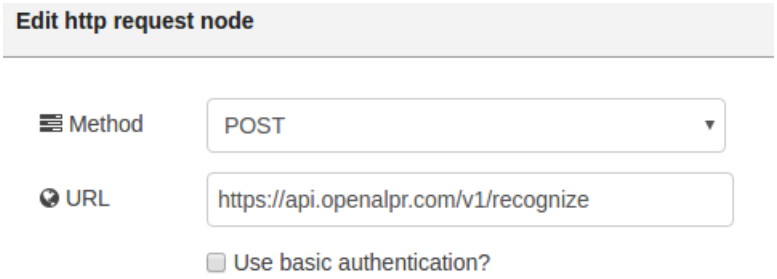


Figure 84: http request node

The request returns a JSON structure under a string format. Use the *JSON* node to obtain a JSON structure containing the *plate.results property*. That is a list of possible license plates numbers, the first element being the most likely to be correct. This is why you need to extract the first element of the list and get the *plate* property. That is a string representing the license plate number. Use the *print* function to display it on the LCD (figure 85).

Figure 85: Display function

You can test the prototype of by placing a picture of a license plate in front of the camera while you pass a magnet next to the Hall Sensor.

Red Signal Violation Detection

Based on the previously built systems, you are now going to build a red signal violation detection, so you can identify the license plates of the cars passing on red light.

What you need

- One Raspberry Pi board connected to Wyliodrin STUDIO ;
- Five LEDs;
- One button;
- Six 200 Ω resistor;
- One Hall sensor (SS441A);
- One Pi Camera;
- One LCD;
- Jumper wires;
- Breadboard

The Setup

For this project, you will use the setup presented in chapter *Traffic Lights* and the one in chapter *License Plate Recognition*.

The Code

For this application, you need the code created in chapter *Traffic Lights* and *License Plate Recognition* and put them together in one application.

You will create a system that knows when the cars red light is turned on and will take a picture only if that is the case.

For the traffic light system, you need to alter the *semaphore* function in order to store the *isRed* variable in the global context. You need to do that because you will use this variable later, in other nodes.

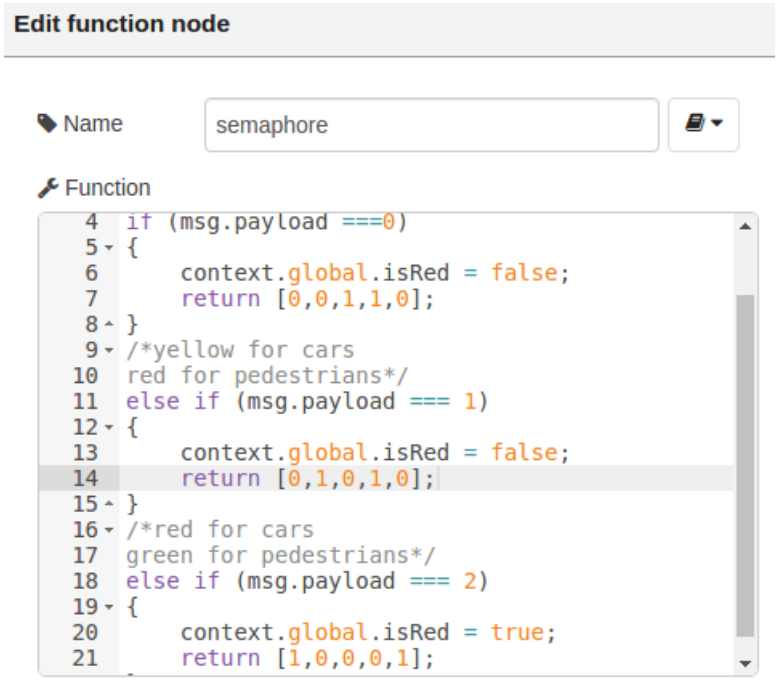


Figure 86: Semaphore function

The function sets the variable to *false* when the red light is off an *true* when it is on.

Further on, you need to check if *redSignal* is *true* when before taking the picture (figure 87).

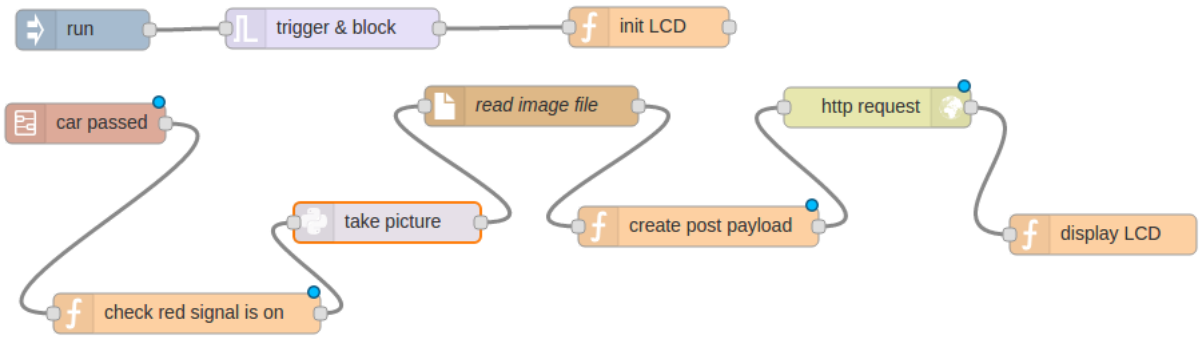


Figure 87: Take picture code

The additional node, *check red signal is on*, verifies the value of *isRed* and sends the message further on only if it has the value *true* (figure 88).

Figure 88: Check red signal is on function

Now the picture will be taken only when the car passes on the red light and take into account that it takes around 5 seconds for the camera to calibrate, so place it a some distance from the Hall sensor.

Web Dashboard

In this project, you will run a web server on your Raspberry Pi and monitor the amount of light in the room from the browser.

What You Need

- One Raspberry Pi board connected to Wyliodrin STUDIO ;
- One Arduino connected to the Raspberry Pi;
- One 10 kΩ resistor;
- One photocell;
- Jumper wires;
- Breadboard.

The Setup

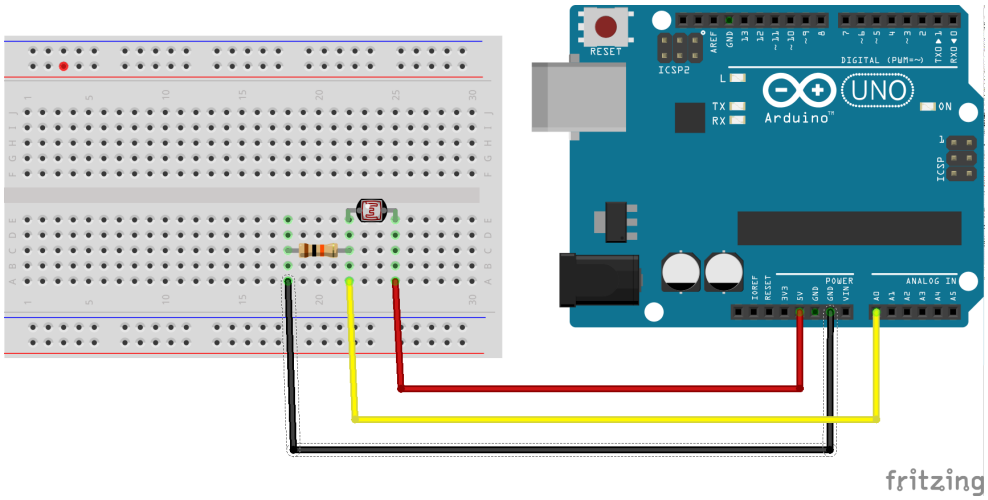


Figure 89: Light sensor schematics

The schematics resembles the one in the previous project which used the light sensor, except that in this case you need to connect the sensor as a pull-down resistor. This is because you want to read a higher value as the amount of light increases.

In addition, you need to have the Raspberry Pi board connected to the same network as your computer. You can use either an Ethernet cable or a WiFi dongle to do that.

The Code

Figure 90: Web server code

Figure 90 displays the nodes you need to use in order to create the web server and display the amount of light on a gauge.

Let's analyse each of the used nodes.

First of all, you will use an *Arduino in* node in order to read the values coming from the light sensor. You have to configure the port the Arduino board in connected on and specify the analog pin the sensor is connected to.

The next node is *light*. It is a *value* node and it simply stores the value coming from *Arduino in* into a variable called light. Make sure *Publish on websocket* is marked. This option allows you to access the value from the webserver.

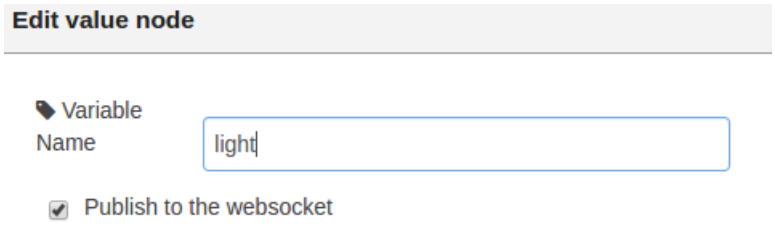


Figure 91: Light node

The *Web route* node allows you to create configure a web server and set a route and a method to access it. You will set the port on which the web server listens (8000) and a route to access from the browser (*/light*) together with a request method (*GET*).

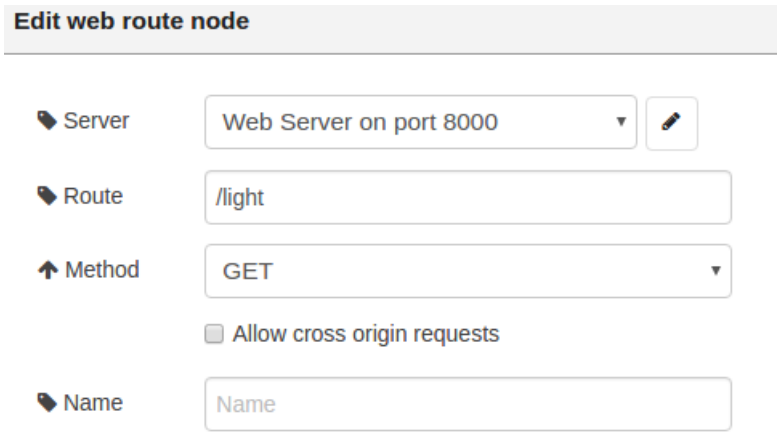


Figure 92: Web route node properties

If you are not familiar with routes, they are the way you can request for a certain resource from a web server. Each route is characterised by a specific path, in this case */light* and by a method, in this case *POST*. For each route, the web server returns different resources (data, HTML files etc).

Once the path and the method is defined, you will describe the behaviour of the web server when the specified route is accessed.

This is what *Web html* node does. In this node you simply describe the HTML page to be returned at the request: HTML Code

```
<html>
  <head>
    <script src="/socket.io/socket.io.js"></script>
    <script type="text/javascript"
src="https://www.gstatic.com/charts/loader.js"></script>
    <script>
```

```
    var chartData = null;
    var chart  = null;
    var options = {
    width:400, height:120,
    redFrom:800, redTo:1023,
    yellowFrom:600, yellowTo:800, max:1023
    };

    google.charts.load ('current', {'packages':['gauge']});
    google.charts.setOnLoadCallback (drawChart);

    function drawChart ()
    {
        chartData = google.visualization.arrayToDataTable (
              [['Label','Value'],['Light',0]]);
        chart = new google.visualization.Gauge
            (document.getElementById('chart_div'));
        chart.draw(chartData, options);
    }

    var socket = io();
    socket.on('value', function (data) {
        if (typeof wyliodrin.onUpdate === "function")
        {
            wyliodrin.onUpdate (data.variable, data.value);
        }
    });

    var wyliodrin =
    {
        onUpdate: null,
    };
</script>
```

```
  </head>
  <body>
    <div id="chart_div"></div>
    <script>
      // This function is called everytime a variable is updated
        wyliodrin.onUpdate = function (variable, data)
        {

            if (variable == 'light') && (chartData != null)
                        && (chart != null)
            {
                chartData.setValue (0, 1, data);
                chart.draw(chartData, options);}
            }
        }
    </script>
  </body>
</html>
```

Let's look into the code.

First of all, the head imports the scripts which define the functions to be used further on, such as *google charts*.

Next, the script declares the variables defining the gauge graph to be displayed (chartData, chart and options). *chartData* and *chart* are variables that need the google module to be loaded in order to get instantiated. *options* sets the characteristics of the gauge, such as width, height, the maximum value to be displayed, the values for which the scale will be coloured red and the values for which it will coloured yellow.

This is why for now they are set to *null*. Firstly, it specifies the type of the graph (gauge). Then, it implements the *drawChart* function ,which will get automatically called once the page has completely loaded.

Next, you call the *load* function, which loads the necessary modules and setts the *LoadCallback*, a function which will be called once the modules finished loading. The function to be called is *drawChart*. It instantiates the values to be displayed. First of all, you need to specify the format of the values: ['Label', 'Value'] and a first value of ['Light', 0].

Next, the *chart* variable is instantiated with a reference to the HTML component which will contain the chart. That is a *div* with the id *chart_div*. Next, you call *draw* to display the chart.

Further on, you will open the socket to get the changes of variables having *Publish on websocket* checked (light in this case). The socket will receive the *value* event and call a function which calls *wyliodrin.onUpdate*. *data.variable* is the name of the variable that has recently been modified and *data.value* is its new value. Then, the *onUpdate* function is set to null so no other node can call it.

Inside the *body*, *wyliodrin.onUpdate* is defined. For this case, each time *light* changes its value, the function gets called and *variable* and *data* will store the name of the variable, respectively the value. So, the function verifies the variable that changes its value is light, also it has to check that all the other elements have been instantiated and if so, it sets the new value to the data set assigned to the chart (*chartData*) and redraws it.

Now that you have defined the behaviour of the web server, the next step is to actually access the page and see if it was created correctly. For this you need the Raspberry Pi's IP address.

Open the *Shell* tab and type *ifconfig*. This will display the board's IP. Open a browser and type in the board's IP followed by *:8000/light* (eg. 192.168.3.20:8000/light).

Now you should see in the browser the values getting updated.

Part III

Reference

Streams Nodes

Run

Sends a payload at a certain interval of time. This is practically the first data flow sent towards the rest of the nodes you use in an application.

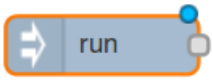


Figure 93: Run node

It has three properties:

- payload - by default, the payload is set to the current time in milliseconds since 1970. This allows subsequent functions to perform time based actions. You can also send the order number of the current message, a blank payload or a string, of your choice;
- topic - can hold data to be used later, in other functions;
- repeat - sets the interval between messages.

The *Fire once at start* option actually waits 50ms before firing to give other nodes a chance to instantiate properly.

Any node can by identified by a name given by the user.

Print

The node can be connected to the output of any node. It will display either the whole message or the payload field of any messages it receives.

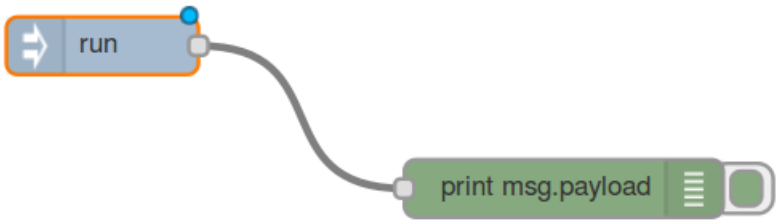


Figure 94: Print the initial payload

The button to the right of the node will toggle it's output on and off so you can de-clutter the console window.

If the payload is an object it will first be converted to a String for display and indicate that by displaying *Object*.

By connecting the previous two blocks you send the output of *run* as input for *print*.

Trigger

The node passes on the message as soon as it receives it. It will, then, wait for a certain amount of time and send another message. The trigger can be extended to send the first value as long as it gets a message and when it doesn't anymore, to send the other value.

Figure 95: Trigger node

You can also set the node to block all messages until it receives a message containing the *reset* property.

Range

The node remaps the payload received to another scale.

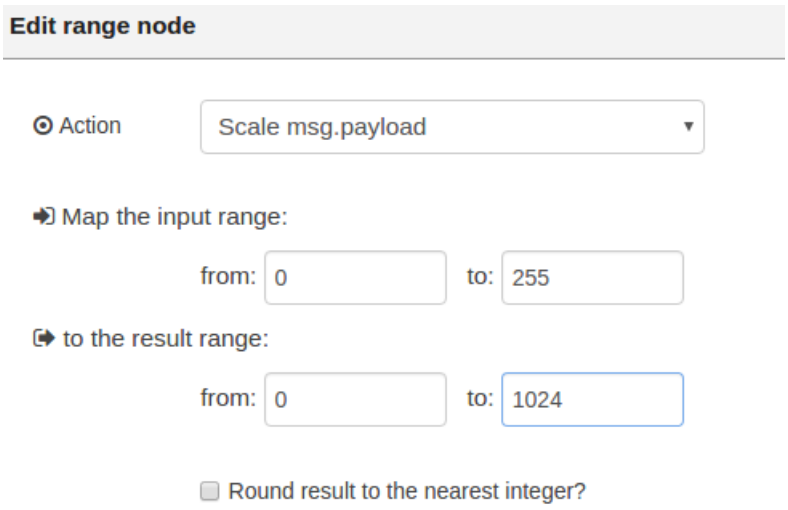


Figure 96: Range node properties

The payload must be a number and the result will be a different value.

You need to specify the interval the payload is from and the interval to which you want to map the received value.

The node has the following characteristics:

- input range min - the lower bound of the interval of the input value;
- input range max - the upper bound of the interval of the input value;
- output range min - the lower bound of the interval to map the value on;
- output range max - the upper‘ bound of the interval to map the value on.

Delay

Introduces a delay into a flow or rate limited messages.

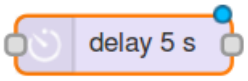


Figure 97: Delay node

By default, the node outputs the received message after 5 seconds (you can specify a different amount of time). You can also set the following:

- random delay - the value of the delay is random, having a value from an interval specified by you;
- limit rate to - limits the amount of messages sent in a certain amount of time; you can also specify to drop the other messages or to delay the

whole stream of messages being transmitted;

- topic based fair queue - adds messages to a release queue tagged by their topic property. At each *tick*, derived from the rate, the next *topic* is released. Any messages arriving on the same topic before release replace those in that position in the queue. So each topic gets a turn - but the most recent value is always the one sent.

Switch

A function node which will evaluate the a property of the message received and pass it on after filtering it through a set of ruled. You can create the rules in the node's settings.

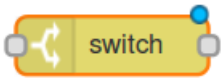


Figure 98: Switch node

You can choose among the following operators:

- >, <, <=, >=, ==, != - to compare the property's value and output the message only if the comparison is true;
- is between - to check if the property's value is between an interval and output the message only if that happens;
- contains, matches regex - outputs the message only for properties storing text that contains a certain sequence of characters or matches a certain pattern;
- is true, is false, is null, is not null - outputs the message with the

property satisfying the statement;

- otherwise - outputs the messages which do not match any rule.

For each rule, the node creates an output where the messages respecting the rule are sent. The rules are verified in the order they are declared.

The node basically acts as an *if* construction.

Change

The change node can modify or delete parts of the message.

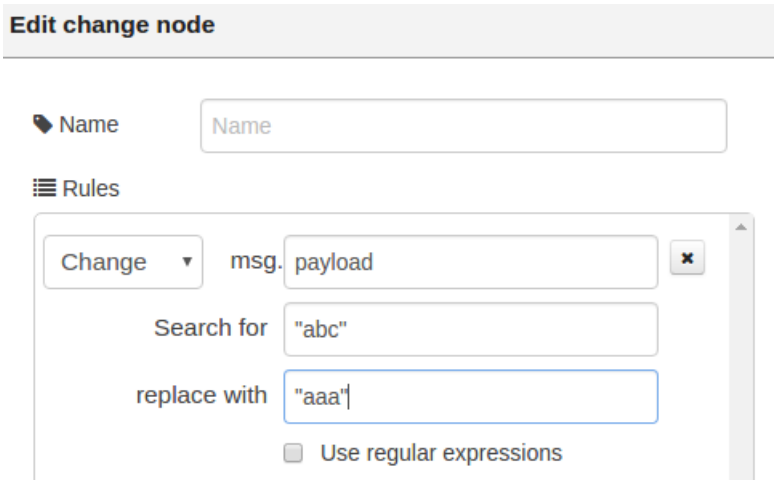


Figure 99: Change node properties

For each of the operations you can do the following:

- set - sets the payload or another property of the message to any value;
- change - searches and replaces parts of the specified property (figure 99);
- delete - deletes the specified property of the message

rbe - Report by Exception

It only passes on messages if the payload has changed.

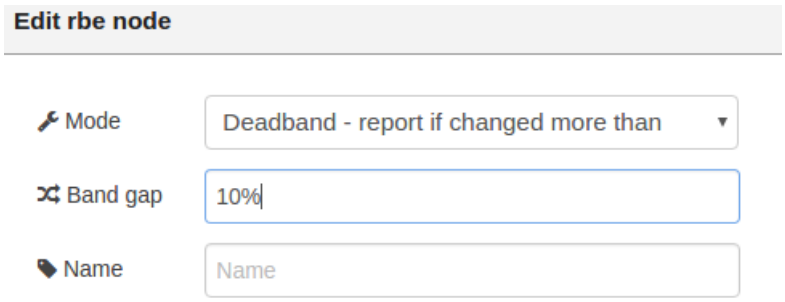


Figure 100: rbe node in deadband mode

In RBE mode this node only outputs the message if the msg.payload is different to the previous one. It works both on numbers and strings.

In deadband mode the incoming payload should contain a parseable number and is output only if the difference between it and the previous payload is of a certain, specified amount. Deadband also supports % and it only sends the message if the input differs by more than x% of the original value (figure 100).

JSON

A function that parses the *msg.payload* to convert a JSON string to/from a JavaScript object. Places the result back into the payload. If the input is a JSON string it tries to parse it to a JavaScript object. If the input is a JavaScript object it creates a JSON string.

Value

Stores the payload in the specified variable in order to use it later. You can set an initial value, which the variable will have before a message reached this node. Otherwise the variable will be undefined until the node is activated.

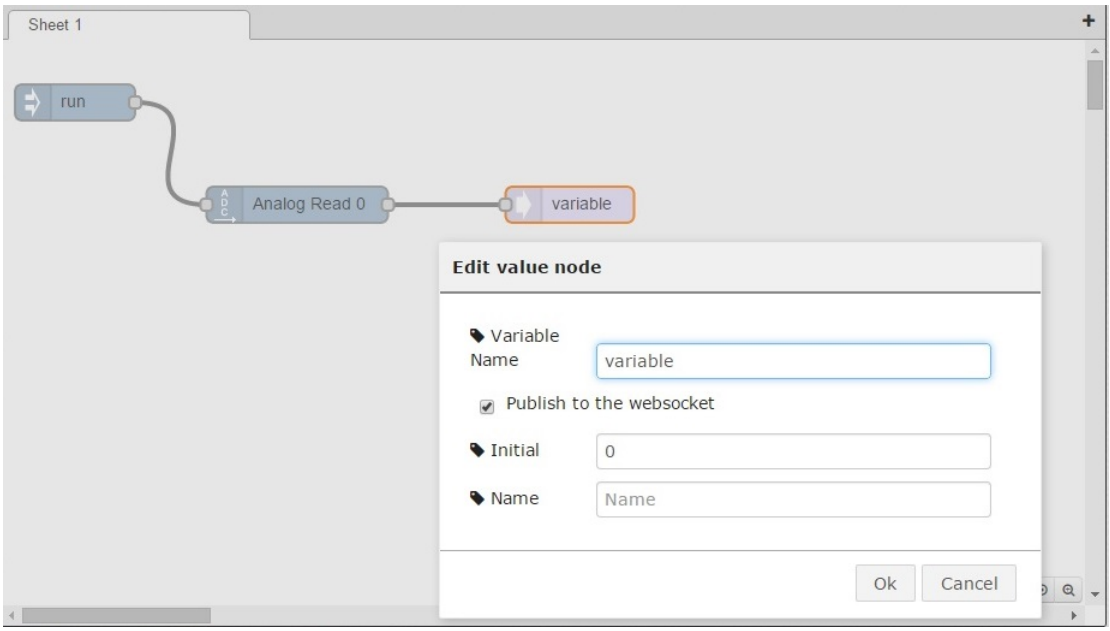


Figure 101: A program using the value node

Subflows

Subflows are similar to the concept of a function. When you have more elements that you use frequently, instead of rewriting them, you can just use the name of the function. Following this judgement, subflow will regroup more nodes into only one, which can be used multiple times. Basically, you will create a new node with a behaviour characterised by the composing nodes.

To create a subflow go to *settings/subflows/create subflow*. You can select the number of outputs and the name of the subflow.

Function

A node inside which you can write JavaScript code to control its behaviour. It receives the message as a JavaScript object as the variable *msg*.

Figure 102: Function node

By convention it will have a *msg.payload* property containing the body of the message.

To log any information, or report an error, the following functions are available:

- node.log("Log");
- node.warn("Warning");
- node.error("Error")

The function can either return the messages it wants to pass on to the next nodes in the flow, or can call *node.send(messages)*.

It can return or send

- a single message object - passed to nodes connected to the first output;
- an array of message objects - passed to nodes connected to the corresponding outputs. For example, If you select the node to have 3 outputs, and return a list having 3 elements, the first element will be

sent on the first output, the second element will be sent on the second output and so on.

If any element of the array is itself an array of messages, multiple messages are sent to the corresponding output.

If null is returned, either by itself or as an element of the array, no message is passed on.

Digital write

It writes 0 or 1 on a digital pin. The pin is specified in the node's properties. The value written on the pin is the value stored in the payload of the received message.

You have to make sure the message reaching the node has a payload with value 0 or 1.

Digital read

It reads 0 or 1 from a digital pin. The pin is specified in the node's properties. Outputs the value read as *msg.payload* and the pin number as *msg.topic*.

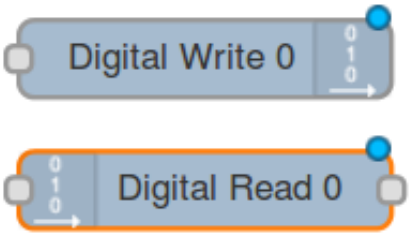


Figure 103: Digital read and Digital write nodes

Analog write

It writes any value on a PWM pin. The pin is specified in the node's properties. The value written on the pin is the value stored in the payload of the received message.

Analog read

It reads any value from an analog pin. he pin is specified in the node's properties. Outputs the value read as *msg.payload* and the pin number as *msg.topic*.

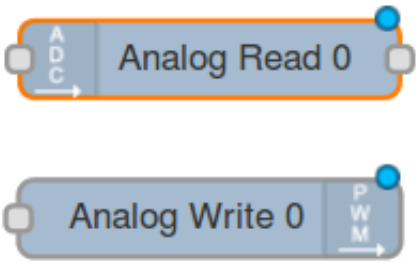


Figure 104: Analog read and Analog write nodes

Arduino input

It connects to local Arduino and monitors the selected pin for changes by using Firmata. You need to specify the port the Arduino is connected on.

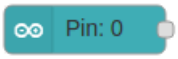


Figure 105: Arduino input node

You can select either *digital* or *analog* input. Outputs the value read as *msg.payload* and the pin number as *msg.topic*.

It only outputs on a change of value - fine for digital inputs, but you can get a lot of data from analog pins, which you must then handle. You can set the sample rate in ms from 20 to 65535.

The Arduino must be loaded with the Standard Firmata sketch available in the Arduino examples.

Arduino output

It connects to local Arduino and writes to the selected digital pin by using Firmata. You can select *digital*, *analog (PWM)* or *servo* type outputs. Expects a numeric value in *msg.payload*. The pin number is set in the properties panel.

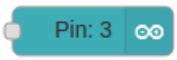


Figure 106: Arduino output node

The Arduino must be loaded with the Standard Firmata sketch available in the Arduino examples.

Send signal

It sends a signal to the dashboard. The signal is characterised by the name you specify in the node's properties and by a value. The value sent is the value of the payload of the message received by the node.

Figure 107: Send signal node

HTTP request

This node can either get, post, pull, delete the data from a web page depending on what method you choose.

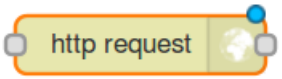


Figure 108: Http Request node

The URL and HTTP method can be configured in the node, if they are left blank they should be set in an incoming message on *msg.url* and *msg.method*. The options you can set in the incoming message are:

- url - is used as the url of the request; must start with http: or https;

- method - is used as the HTTP method of the request; must be one of GET, PUT, POST or DELETE (default: GET);
- headers - should be an object containing field/value pairs to be added as request headers;
- payload - is sent as the body of the request.

When configured within the node, the URL property can contain mustache-style tags. These allow the url to be constructed using values of the incoming message. For example, if the url is set to `example.com/{{topic}}`, it will have the value of *msg.topic* automatically inserted.

The output message contains the following properties:

- payload - the body of the response;
- statusCode - the status code of the response, or the error code if the request could not be completed;
- headers - an object containing the response headers.

Web route

It runs a web server listening on port and registers and registers the route for method. This uses express.js. Whenever a request is made to route, *msg* will contain the following:

- payload - requests data (GET/DELETE - query string, POST/PUT - request body);
- req - request;
- res - response;

- next - next.

The node should be connected to a *Web response* or *Web HTML* node and each time a route is accessed from the browser, the response is what these nodes returns.

Web response

It will set the response of the server for a web request. it resonds with the payload.

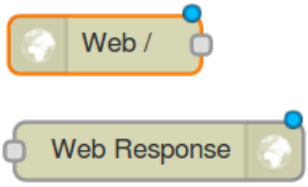


Figure 109: Web route and web response nodes

Web HTML

A template where you can write HTML code. If *Replace variables* is checked, the {{variable}} will be replaced from *msg.payload*.

Figure 110: Web html node

The node should be connected to a *Web route* node and the HTML page created is the resource returned for the request.

Resistor Color Code

Every resistor has colored bands. Those bands and colors are not random at all, they help to identify the specifications of the resistor.

Here is how you can see the value of a resistor depending on its colored bands.

Every color represents a different number.

Here is the table of all the colors and numbers:

Black	0
Brown	1
Red	2
Orange	3
Yellow	4
Green	5
Blue	6
Violet	7
Grey	8
White	9

Counting from right to left, the second color is the multiplier. Digits of the first colors must be multiplied with the number of this color.

Black	1
Brown	10
Red	100
Orange	1000
Yellow	10000
Green	100000
Blue	1000000
Gold	0.1
Silver	0.01

And the last color is the tolerance. Tolerance is the precision of the resistor and it is a percentage.

Brown	1
Red	2
Gold	5
Silver	10
Nothing	20

There are a lot of programs that can calculate the value of a resistor but if you do not have access to the Internet and you need to know a certain value you can use the table from Figure 111.

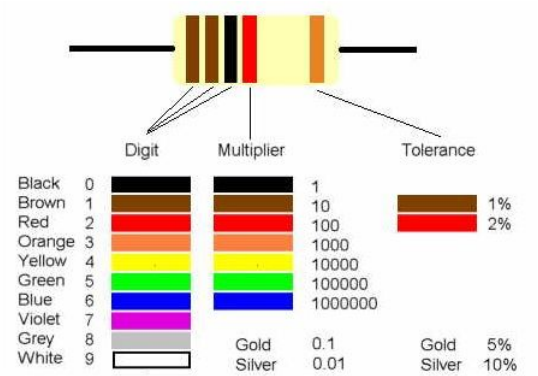


Figure 111: Resistor Color Code

Examples:

1.Yellow and green represent 4 and respectively 5. They represent the first and second digits, so you will have 45.

The third band is orange. As a multiplier, it is x 1000, so you will calculate 45 x 1000 thus, the resistance is 45,000 Ω.

The forth band, silver, represents the tolerance, so the final expression of the resistance is 45,000 +/- 10Ω (Figure 112)

Figure 112: Resistor Color Code

2. A resistor colored Orange-Orange-Black-Brown-Violet would be 3.3 kΩ with a tolerance of +/- 0.1(Figure 113)

Figure 113: Example 2