

ARDUINO SIGNAL DRIVER FOR SCALE TRAINS

v 1.0 - Darío Calvo-Sánchez, 2021

<https://elgajoelegante.com>

English version from original "Control de semáforos de trenes a escala con Arduino"

Arduino-based LED signal control software for scale railroad models, using PCA9685 PWM boards, with the following key characteristics:

- Control of up to 12 signals with a maximum of 48 lights. Practically any signal can be controlled:
 - In common-anode configuration, the maximum number of lights per signal is 4 (up to 3 of which can be lit up simultaneously)
 - In common-cathode configuration, the maximum number of lights per signal is 5 (all of which can be lit up simultaneously if required)
- Compatible with both analogue and digital systems
- A mix of common-anode and common-cathode signals can be connected to the system
- A mix of signals with different number of lights can be connected to the same or different PCA9685 boards. Even one signal can have some lights connected to one board and the rest to a different one
- All lights in a common-anode signal will have the same brightness, no matter the number of lights switched on simultaneously on that signal
- Selectable dimming rate, independent of the number of signals or lights connected, allowing realistic representation of any signal type
- Selectable blinking rate
- All commanded lights in a signal will light up or turn off at the same time
- No practical limit in the number of PCA9685 boards that can be connected (up to its maximum of 62)
- Definitions are included for typical Spanish RENFE/ADIF signals, although any signal from any railroad can be defined by the user
- Rocrail SVG files included

TABLE OF CONTENTS

1	Introduction	3
2	Requirements.....	3
3	Installation	4
3.1	Overall view	4
3.1.1	Digital systems	4
3.1.2	Analogue systems	5
3.1.2.1	Manual mode	6
3.1.2.2	Automatic mode.....	7
3.2	Connecting Arduino and PCA9685 boards	7
3.3	Signal connections.....	9
3.3.1	Common cathode signals	9
3.3.2	Common anode signals	10
3.4	Electrical feeding	10
3.5	Program set-up.....	11
3.5.1	Common parameters	11
3.5.2	Digital version specific parameters	13
3.5.3	Analogue version specific parameters	14
4	Using the program	15
4.1	Digital version specifics	16
4.2	Analogue version specifics	20
5	Troubleshooting.....	21
	Appendix 1: Arduino DCC decoder	22
	Appendix 2: Common-anode converter circuit.....	25
	Appendix 3: Modifying the signal system	27

1 INTRODUCTION

This system is a driver for scale railway LED signals. It just makes each signal to show the appropriate commanded aspect. It therefore requires a way to send that command to the driver (see section 3.1).

There are hundreds of algorithms like this, but most of those only care about red & green semaphores, or do not allow mixing different types of signals. This system offers greater flexibility, allowing accurate representation of practically any real-life railroad light signal, both in digital and analogue systems. It is also fully configurable in terms of dimming and blinking.

The main difference with other systems is the use of PCA9685 PWM boards, which are designed mainly for servo-motor controlling, but can handle LED lightning effectively as a secondary function. Usual Arduino outputs can drive LED lights easily, but using these boards we'll be able not only to get some of the advantages mentioned above (such as the possibility to mix common-anode and common-cathode signals in a simple way) but also to connect a greater number of signals without needing the extra pins on an Arduino Mega. On top of that, pins on the Arduino board will be almost completely free for other purposes (like connecting a keypad to control the signals in an analogue layout). Another advantage is that electrical feeding for the signals can remain totally independent from the Arduino.

The final reason: PCA9685 boards are cheap. Particularly when paired to an Arduino Nano and compared to an Arduino Mega.

2 REQUIREMENTS

This guide assumes some basic knowledge on Arduino and its programming, and some basic skills on electronics (components, set-ups and elementary circuits)

Minimum requirements:

- Arduino board (a Nano will suffice in most of the cases)
- PCA9685 PWM boards: as many as required (normally no more than 3)
- 5 V DC power source (an old USB phone charger can do the trick)
- Common-anode converter circuits (if signals are of the common-anode type), see Appendix 2
- Command method: push buttons, decoders, reeds, automated systems, etc (see 3.1)

Recommended:

- In case of running the system on a digital layout, it is highly recommended that both the command station and the control software (if computer-controlled) support the "Signal Aspect" protocol to handle multi-aspect signals (see Chapter 4). That will simplify the whole system set-up, and it is the only protocol supported on this Guide.

The code requires public libraries **Adafruit_PWMServoDriver.h** to control PCA9685 boards, and **Wire.h** to handle I2C bus communications. Both are available by default in current official Arduino IDE versions.

3 INSTALLATION

3.1 OVERALL VIEW

This program is a signal driver. It manages to show the right signal aspect from those defined on a particular signalling system, including any required blinking or dimming. It therefore requires an input system to command each signal.

Basic schematic setup is shown on Figure 1:

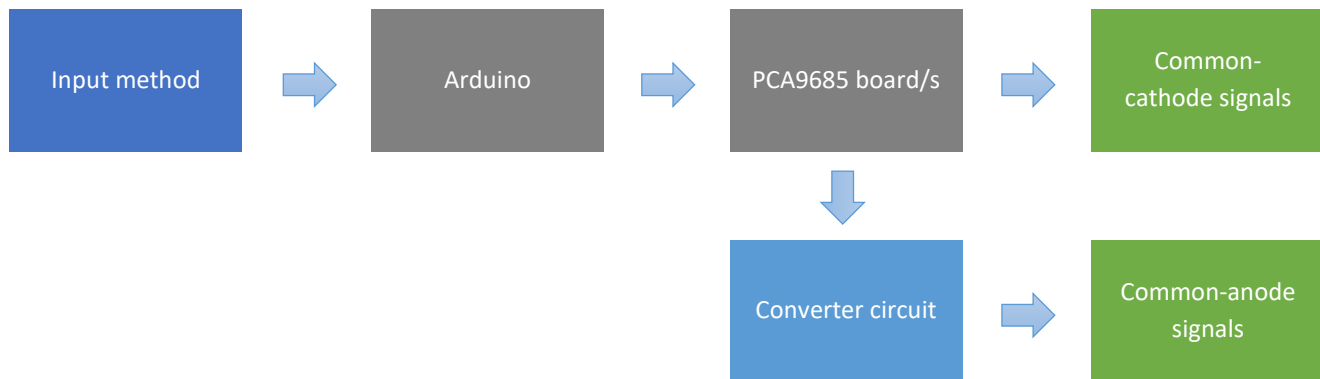


Figure 1: General scheme

There are different possibilities for the input method. On a digital layout, a decoder will be used as an interface between the command station and the Arduino driving the signals. On an analogue layout there will be several options, from simple push buttons to automated systems using reed contacts, relays, etc.

In any case, the Arduino will receive the orders from the input method, will assign the right aspect to each signal, and it will control the PCA9685 boards to power them. If the signals are of the common-anode type, a converter circuit will be required in between (see section 3.3.2).

3.1.1 DIGITAL SYSTEMS

On a digital layout, the input method will be an accessory decoder that will translate the orders from the command station to a command array with the aspects in all controlled signals, using the format defined on Chapter 4). However, most decoders sold currently in the market **will not be adequate** for this purpose. Fortunately, this decoder can be built in a cheap and easy way using another Arduino board, and connecting both boards through the serial port.

Figure 2 shows a connection scheme. The Arduino Nano board on the upper part serves as a decoder, sending translated commands to its sibling directly below, which drives the signals through the PCA9685 boards on the right-hand side of the picture. Note that the wires connecting both Arduinos are crossed, so each TX1 transmitting pin is linked to the RX0 receiving pin and vice-versa.

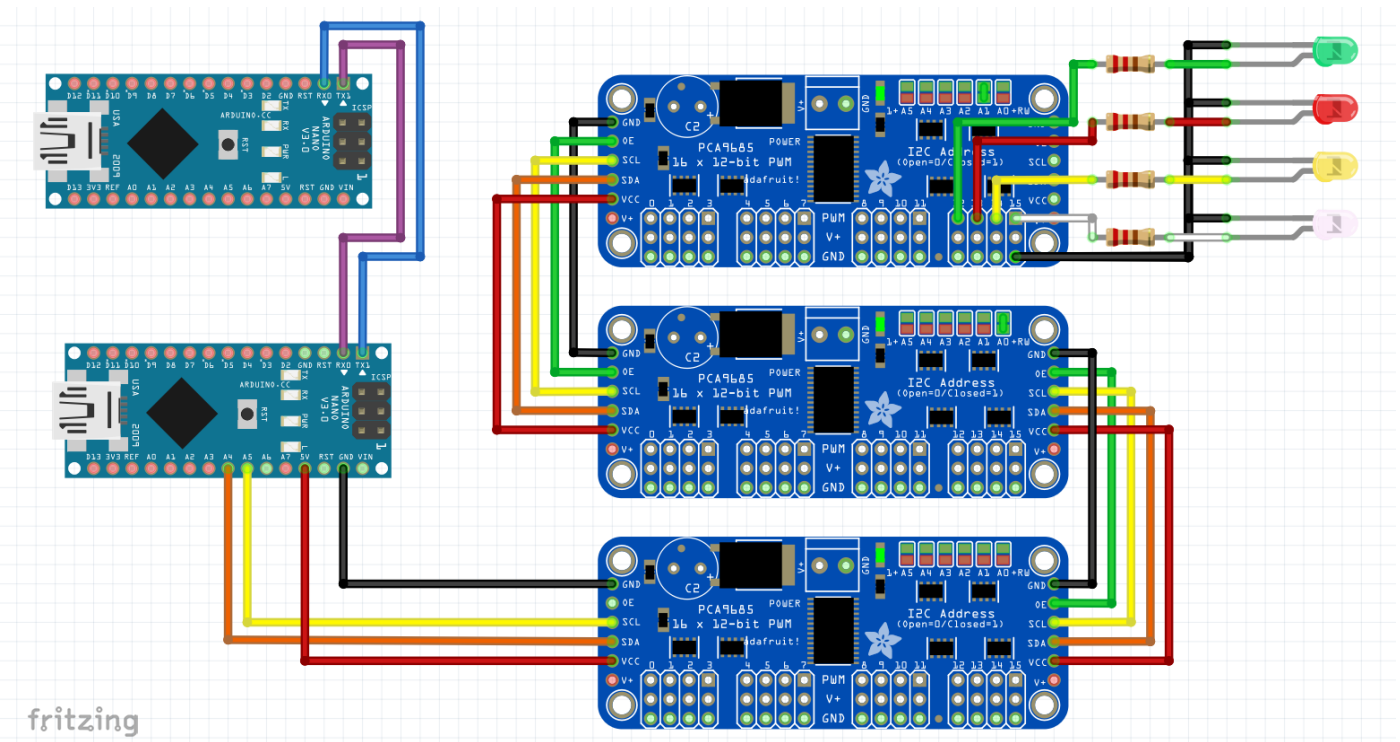


Figure 2: Overall setup on a digital layout

One important remark: it is technically possible to integrate the decoder function on the Arduino board driving the signals, thus avoiding the need to buy another Arduino. However, this is **not recommended** because of the following reasons:

- 1) When integrating the decoder function, available memory on a Nano board reaches its limit, making it unstable.
- 2) Usual decoding routines increase processor load. Controlling more than eight signals will cause the program to behave erratically, skipping command station orders, taking a lot of time to change a signal aspect, or impacting on the blinking or dimming routines. Using an Arduino Mega will not solve these problems, as it has more memory but roughly the same processing capability.
- 3) Installation with a separate decoder is “cleaner”, and more flexible if changes need to be introduced or any breakdown occurs. Current prices of Arduino Nano boards are quite cheap, so it does not make a true difference in terms of economy. On top of that, all decoding functions needed for signals, switches, lightning, and other accessories can be put together in this single additional board.

This software, as it is provided, assumes there will be an external decoder. See Appendix 1 for instructions to build a compatible decoder.

3.1.2 ANALOGUE SYSTEMS

On analogue layouts, wiring and setup will depend on the particular configuration and degree of automation. In this document, only the manual mode will be covered.

3.1.2.1 MANUAL MODE

In this case, push buttons will be connected to the Arduino board pins, serving as an input method to command the signals (see Figure 3). Normally, signal aspects will not have any automated influence on running trains: signal interpretation will be the train driver's responsibility. This is the simplest way to control complex signals with many different aspects.

Each Arduino's free I/O pin will handle a push button, being the other terminal of the button connected to the common GND Arduino pin. Not only the digital pins (D0 to D13, numbered in the software code as 0 to 13) could be used for this purpose, but also the eight analogue ones (A0 to A7, numbered in the code as 14 to 21). The exceptions are the two pins used for the I2C bus: A4 and A5 (18 and 19 in the code). Check the manual in case an Arduino board different than a Nano.

The simplest way to arrange the push buttons, which is the one used in this code, is to dedicate some of those to select the aspect, and the rest to select the signal. In this way, we would push first one of the buttons to select a signal, and then another one to select its desired aspect (go, stop, caution, etc.); optimizing the limited number of pins available.

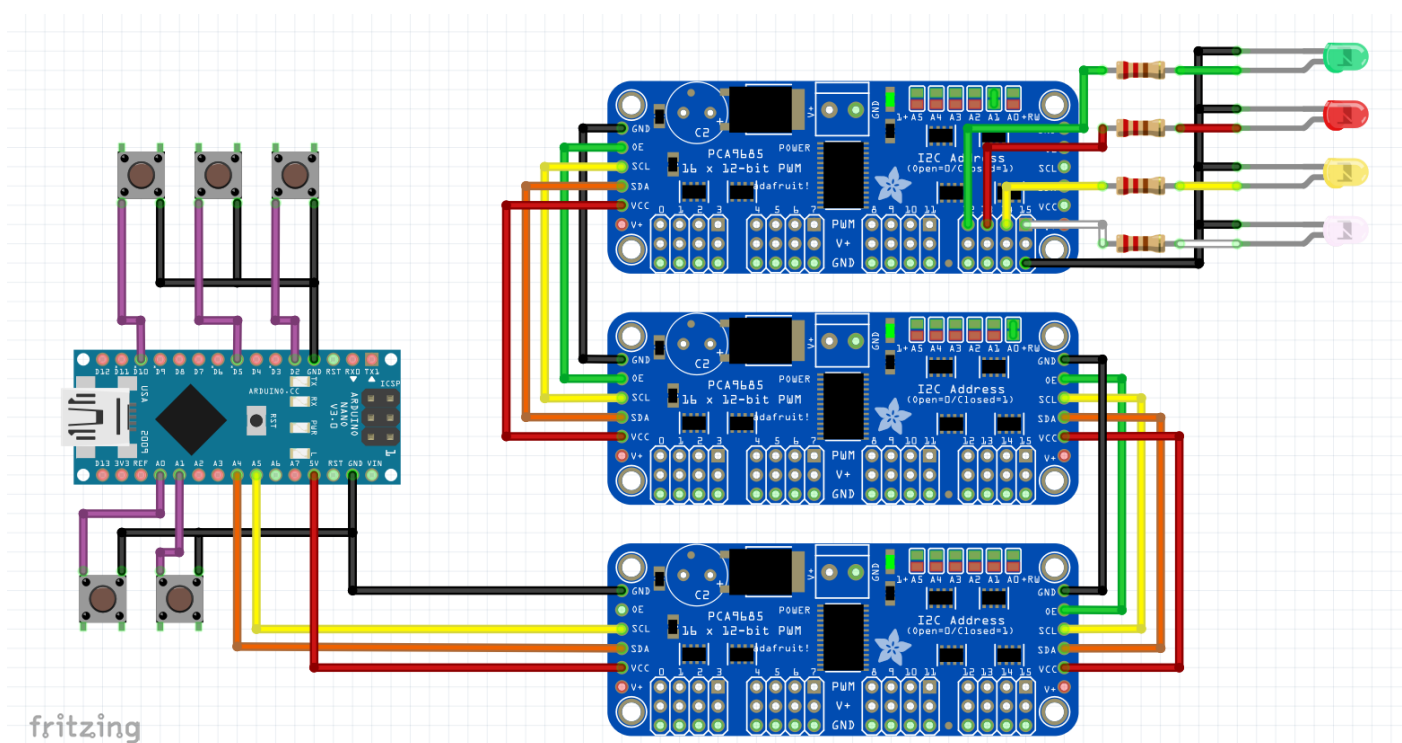


Figure 3: Overall setup on an analogue layout

So, obviously, the only restriction in this way of working comes from the number of free pins available. Assuming that only the I2C bus pins are occupied, from the total of 22 available I/O pins (14 digital plus 8 analogue) we will have 20 free. Based on that limit we will select a combination. For instance, if we have signals with a total of 10 possible aspects, we will need 10 pins to select these aspects, and therefore we would have 10 pins available to control up to 10 signals. Another example: if we need to control 12 signals, we could select a maximum of 8 aspects.

These 20 pins available will cover most of needs. If we need more, we will have two options:

- Using an Arduino Mega as driving board, instead of a Nano. This will provide a total of 68 free pins, once the two I2C bus pins have been subtracted. Enough for practically any signal system in the world. Of course, this is a more expensive option than a Nano, although it still can control only 12 signals and 48 lights. So we would need to carefully make the case, counting the aspects and signals required, and maybe go for two Nanos with their PCA9685 boards each, instead of using a single Mega.

- b) Using a [numeric keypad](#), which allows selecting a signal by an ID number, and then an aspect by another number. This requires a small number of pins, and probably represents the best option in terms of wiring simplicity (always something to consider in an analogue layout). If this option is chosen, an LCD display would be a good add-on, allowing to see what we are typing.

For option a) see section 3.5 to specify the connected pins in the code. For option b) program modifications are required, which is something simple when having basic Arduino programming skills. Option b) is not covered in this document.

3.1.2.2 AUTOMATIC MODE

In this mode, the system will be commanded by an auxiliary automated system to control the traffic of trains in the layout, normally using reed contacts, presence detectors, timers, relays, etc. Track blocks and interlockings for signals and switches will be defined.

For instance, a reed contact can be connected to an Arduino pin (in the same way as a push button on Figure 3). Programming the Arduino board accordingly, a train passing over the contact could change the aspect of a certain signal. Pin number limitation would be the same as explained in previous pages, depending on the type of Arduino board used.

Given the vast number of possible combinations, interlockings, layouts, train block definitions, control types, and all the associated configurations, the programming of the traffic control system is left to the user. If the system is complex enough, probably a separate Arduino board will be required only for that, sending the command to the Arduino signal driver board in a similar manner as seen in section 4.1 for digital layouts.

3.2 CONNECTING ARDUINO AND PCA9685 BOARDS

PCA9685 boards are connected to the Arduino through I2C bus, using “SDA” pin (data) and “SCL” pin (clock) in both devices, as shown in Figure 4. On the Arduino Nano, SDA pin is A4 and SCL is A5. **NOTE:** Please refer to the corresponding documentation in case of using a different Arduino model.

Maximum length of I2C connections is between 20 and 30 cm. This means that PCA9685 boards must be placed close to the Arduino and close to the other PCA boards, extending the output wires to the signals instead.

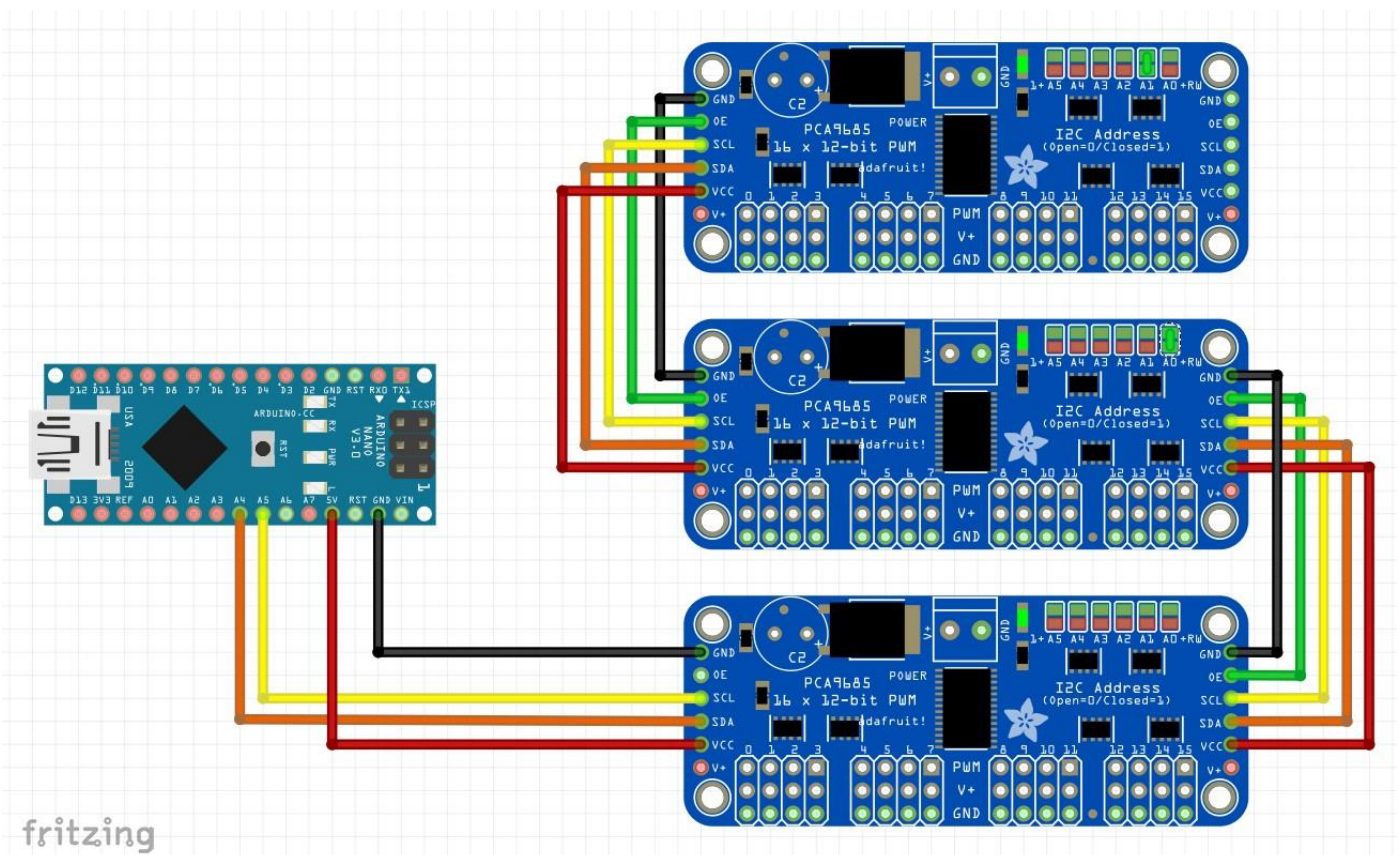


Figure 4: Linking PCA9685 boards to Arduino

In case of requiring more than one PCA9685, these boards can be chained together as shown in Figure 4, using the connections on their ends. Pins marked as “OE” will be connected also between these boards. The only important remark is **the need to assign different addresses to each of the boards**. See [here](#) for further information. This is done by soldering some small pads on the board, which will define the address. The address set in the factory is always “0x40”, so if we just need one board, we do not need to assign any address. If we need more, most likely there will be no more than three before reaching the limits of signals and lights for the program. In that case we could assign the address “0x41” to the second board by soldering its pad marked as “A0”, and the address “0x42” to the third board, by soldering its pad marked as “A1”. Figure 5 shows a board with address 0x42 as an example, where its pad “A1” has been soldered:

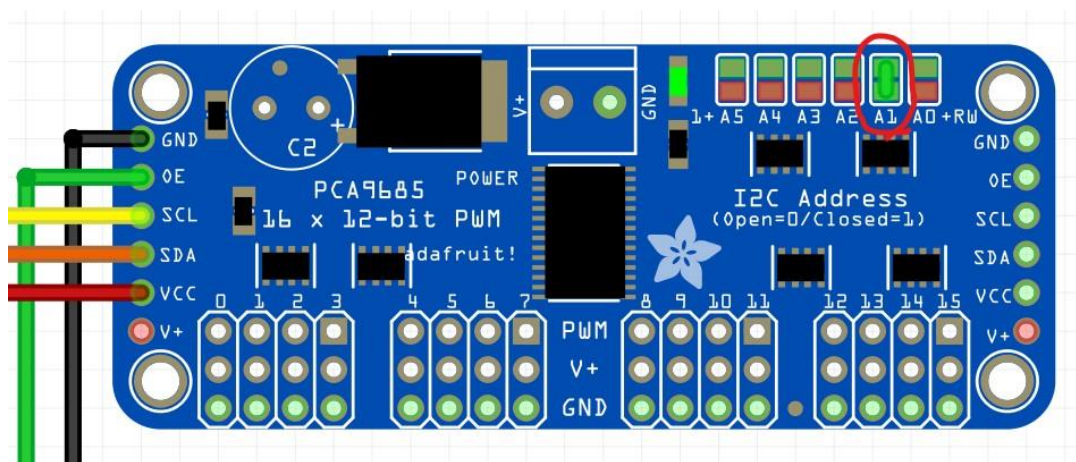


Figure 5: Assigning 0x42 address by soldering pin A1

3.3 SIGNAL CONNECTIONS

Now it is the turn to connect each signal to PCA9685 board outputs.

Each board has 16 outputs, numbered starting from 0 to 15. Each light on each signal will connect to “PWM” and “GND” pins on one of the board exits. It is not required for all lights in a single signal to be connected to the same PCA9685 board: the position of each light and signal will be “mapped” later in the software settings (see 3.5.1).

But now it comes the distinction between common-anode and common-cathode signals:

- Common-cathode LED signals have a common negative wire, with one positive connection per light.
- Common-anode LED signals have a common positive wire, with one negative connection per light.

Both types can be connected indistinctly, but keeping in mind its technical particular aspects, detailed as follows.

3.3.1 COMMON CATHODE SIGNALS

PCA9685 boards, as well as many other commercial microcontrollers (including Arduino), provide common-cathode outputs. “GND” pins are common to all outputs, although they are repeated for practical reasons. Therefore, common-cathode signals can be connected directly, as seen on Figure 6. LED cathodes will be linked to “GND” pin or pins, and anodes will be connected to its selected “PWM” pin, through to a suitable resistor. This resistor will depend on the LED type and colour.

Common-cathode signals are typically found in home-built signals.

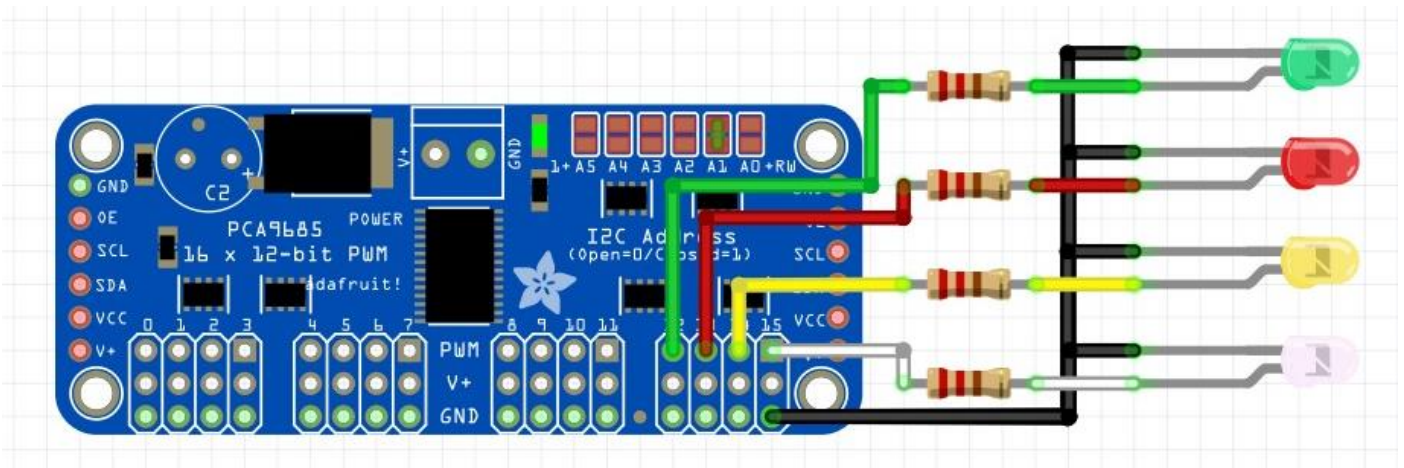


Figure 6: Example of a four-light common-cathode signal connection

The PCA9685 board output is limited to a 5V / 10 mA current. Given that, it is technically possible to connect the LEDs to the board directly (without the intermediate resistor) but this is not a recommended practice. It is always desirable to add the resistors, which also will help us to perfectly balance the brightness of each light in the signal. As said above, the resistor value for each light will depend on the type and colour of the corresponding LED. This information can be found on each LED datasheet. For computing the adequate value, the board output current of 5 V / 10 mA will be considered.

3.3.2 COMMON ANODE SIGNALS

Most scale trains manufacturers make common-anode signals. If we have such a type of signal, we will need a **converter circuit** between the PCA9685 board and the signal (see Figure 7). For each signal, one of these circuits will be required. The converter will have as many branches as lights has the signal. See Appendix 2 for the schematics and material required to build it.

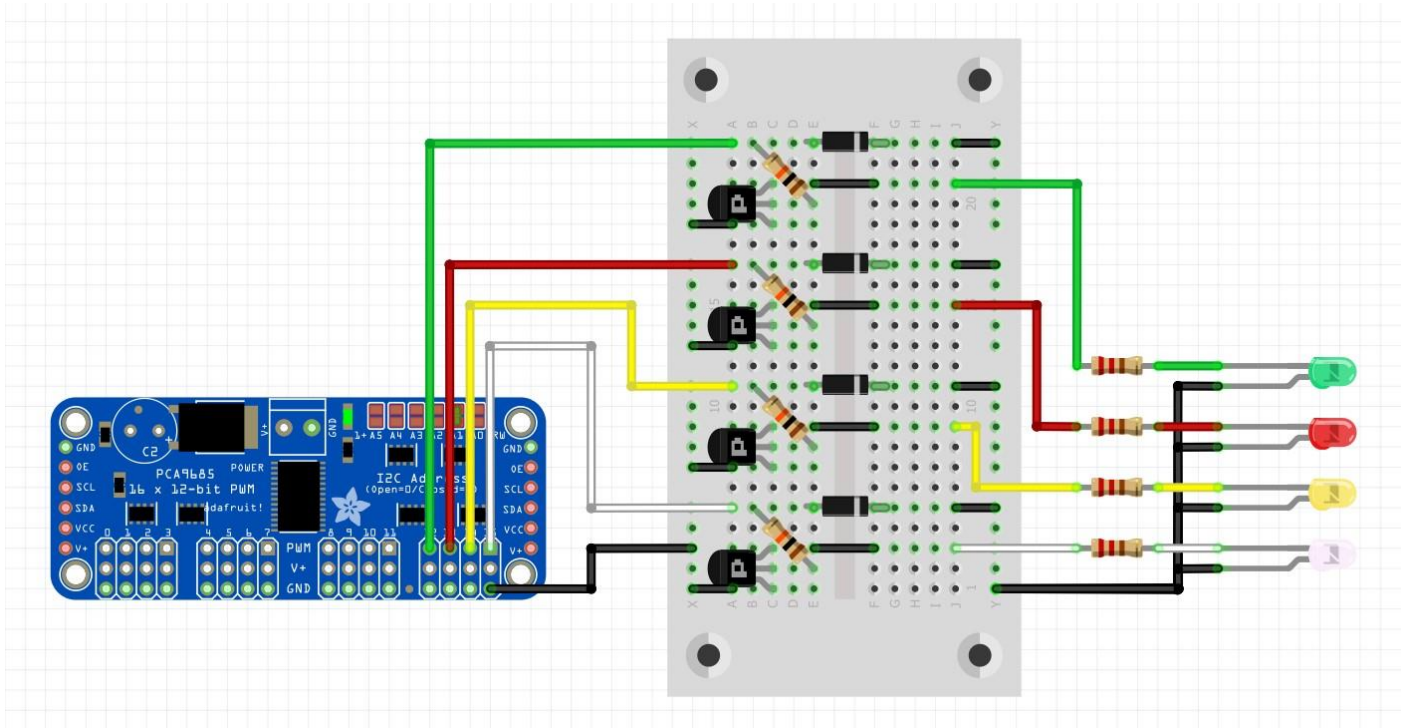


Figure 7: Example of a four-light common-anode signal connection, using a converter circuit

These circuits *invert the way the signal works*. Whereas in a common-cathode signal we simply gave current to the green LED to light it, in a common-anode signal connected to a system like ours, *we will need to power all the rest of lights and leave the green LED unpowered if we want to light it*. Fortunately, this is done automatically in the software code, thanks to the use of the PCA9685 boards and their libraries, not requiring any user intervention. This way of working could also cause some imbalances in terms of light intensity, depending on the number of lights lit-up simultaneously, but this is also considered and corrected on the software code. However, it must be noted that these compensations **will reduce the LED brightness by a factor of 3** compared to a common-cathode system, so this must be considered when computing the resistor value for each LED.

Speaking about LED resistors, commercially available signals often have pre-installed resistors correctly balanced for each colour, but these are normally computed for 12-16 V currents. If we keep these resistors in place, an excessively dimmed light will normally be obtained. It is therefore recommended to replace these resistors by new ones, computed accordingly to the PCA9685 board output (5V / 10 mA) and considering the reduction factor of 3 mentioned above. It is of course a matter of personal taste, but a good starting point for each resistor value will be to take the original mounted in the signal and divide its value by three.

3.4 ELECTRICAL FEEDING

The Arduino board will be feed as desired (through USB input, Vin pin, or whatever other applicable method). It is obviously important to consider all applicable restrictions and instructions for the selected method.

PCA9685 boards require 5 V DC feeding through the “Vcc” (positive) and “GND” (negative) pins on their ends. Note that “V+” pins will not be used, as these are intended to electrically feed servos instead of LEDs.

It is recommended to use an external stable power source to provide this 5 V DC current to the PCA9685 boards. This will be almost a requirement if we have a significant number of signals and lights). An old USB mobile phone charger could be used for this purpose (and could be used also to feed the Arduino through its 5V pin).

IMPORTANT NOTE: If the power sources for the Arduino and the PCA9685 boards are different, negative or ground (GND) pins of all the boards and the Arduino must be linked together.

3.5 PROGRAM SET-UP

Before uploading the sketch or program code to the Arduino board, it is needed to modify certain program parameters to suit the configuration of each user. Two sketches are provided: one for digital layouts (inside “digital_version” folder) and one for manual-mode analogue layouts (inside “analog_version” folder). Required inputs are detailed as follows:

3.5.1 COMMON PARAMETERS

Number of boards and signals

- Number of PCA9685 boards connected to the Arduino (integer value between 1 and 62; normally equal or less than 3)

```
const uint8_t Numero_placas = 1;
```

- Number of signals connected to the system through the PCA9685 boards (integer value between 1 and 12):

```
const uint8_t Numero_semaforos = 2;
```

Common lightning parameters

- Frequency (integer value in Hertz) of the output current reaching the LEDs. It can be adjusted between 40 and 10000 Hz. A reasonable starting value is 200 Hz: reduce it if a high-pitched noise can be heard, or increase it if there is any visible blinking effect. It does not affect LED brightness.

```
const uint16_t frec = 200;
```

- Dimming/lightning time (integer value in milliseconds; max. 65535). Defines the time it will take for the LED to reach its full brightness or to completely fade out. This value will be chosen depending on the type of real-life signal being represented and its original bulb. 120 ms is a good starting point for classic semaphore signals.

```
const uint16_t encendido_ms = 120;
```

- Blinking time (integer value in milliseconds). Defines the blinking time for signals with blinking aspects, not including the dimming time defined above (i.e. this will be the time during which the light will be *completely on* or *completely off*).

```
unsigned long periodo = 425;
```

Therefore, the total time a light will be lit on during a blinking phase will be defined by:

$$(2 \cdot \text{encendido_ms}) + \text{periodo}$$

- Pause during aspect change (integer value in milliseconds; max. 65535). Defines a pause when changing aspects in a signal, with all lights off for aesthetical reasons. 100 ms can be a nice starting value.

```
const uint16_t pausa_aspectos = 100;
```

Parameters for PCA9685 boards

There are three blocks, corresponding to the maximum supported number of boards. Objects "placa[]" are numbered from 0 to 2 (placa[0], placa[1] and placa[2]). Example for board 2:

```
// Board 2
placa[2] = Adafruit_PWMServoDriver(0x42);
placa[2].begin();
placa[2].setPWMFreq(frec);
```

IMPORTANT: All three boards are activated by default. Any unused board must be commented adding two slashes (//) before each line.

In case of needing more than three boards, copy/paste one of these blocks and follow the numbering sequence of the array "placa[]".

Apart from that, the only parameter required in this section is the physical address of each board (see section 3.2) in the function "Adafruit_PWMServoDriver()". In the following example, the first board (0) corresponds to the physical address "0x40":

```
placa[0] = Adafruit_PWMServoDriver(0x40);
```

Parameters for signals

Here the working parameters for all installed signals will be introduced. Signals are numbered starting from 0 up to a maximum value of 11. Note that the maximum index must be consistent with the maximum number of signals specified: the highest index will be equal to Numero_semaforos - 1.

All 12 blocks are active by default, but only the signals that are actually connected must be active, the rest must be commented with two slashes (//). For instance, if we have 6 signals, we will only use blocks from 0 to 5, and we will comment the rest.

Each group or block will contain signal data as follows:

```
// Signal 0
// Reference: Mafen 4131.15 (RENFE 4-light departure signal)

anodo_comun[0] = true;           // Type
pin_dummy[0] = false;           // Not in use

semaforo[0][0][0] = 1;           // Green light board
semaforo[0][0][1] = 4;           // Green light pin

semaforo[0][1][0] = 0;           // Red light board
semaforo[0][1][1] = 9;           // Red light pin

semaforo[0][2][0] = 0;           // Yellow light board
semaforo[0][2][1] = 10;          // Yellow light pin
```

```

semaforo[0][3][0] = 1;           // White light board
semaforo[0][3][1] = 5;           // White light pin

semaforo[0][4][0] = 99;          // Blue light board
semaforo[0][4][1] = 99;          // Blue light pin

comando[0] = 13;                 // Initial aspect

```

First, the signal type (common anode or common cathode) is defined by setting the parameter “anodo_comun” to “true” or “false”, respectively. Currently, “pin_dummy” parameter has no influence, as it is included for a further evolution of the code. It will be set to “false”.

Next, the array “semaforo[][][]” defines the PCA9685 board and the pin to which each signal light is connected, as follows:

semaforo[*Signal Id*][*Light Id*][*Element*]

“*Signal Id*” corresponds to the index assigned to the signal, as explained above.

“*Light Id*” is assigned from 0 to 4, corresponding to the maximum of five lights per signal. To ease the identification of each light, comment tags have been added with typical colours: green (for light 0), red (for light 1), yellow (for light 2), white (for light 3) and blue (for light 4). Obviously, it is not required that these tags match the actual signal colours.

“*Element*” can only take 0 or 1 as possible values: 0 will tell the code we are indicating the PCA9685 board to which that light is connected, whereas 1 will tell the code we are indicating the pin for that light.

IMPORTANT: In case of not using some of the lights in a signal, do not comment the lines. Instead, the values (both for pins and boards) in this array will be set to 99. The whole block will be commented only if the entire signal is not used.

Note that different lights from a certain signal can be connected to different boards, taking benefit of the maximum pin capability of each board.

Finally, the array comando[] will define the initial aspect shown by the signal when starting the system, according to aspect definition (see Chapter 4).

In the example above, the signal is the first on the list (id number 0) and has four lights. Some of those lights are connected to board 0, and others to board 1: green light is connected to pin 4 in board 1, and white light to pin 5, also in board 1. But red and yellow lights are connected to pins 9 and 10 respectively, both in board 0. There is no blue light, so both pin and board values are set to 99. Initial aspect is set to 13, which corresponds to “Authorized movement” according to Table 1 in Chapter 4.

3.5.2 DIGITAL VERSION SPECIFIC PARAMETERS

Communications

Adjust here the link rate of the Arduino’s serial port (baud rate):

```
Serial.begin(9600);
```

Obviously, it must match the emitter's speed (no matter if it is another Arduino, a PC, or any other device). In this example, it is set to 9600 bauds.

NOTE: The use of Arduino's main serial port is assumed. If any other one is used (either a physical one in an Arduino Mega, or a software-emulated port in any other Arduino type) it must be modified here accordingly, and also in "recepcion_comando" function.

3.5.3 ANALOGUE VERSION SPECIFIC PARAMETERS

Only manual mode is covered in this guide (see 3.1.2)

Amount of controllable aspects

The total number of aspects that can be selected (through their corresponding push buttons) will be indicated here:

```
const uint8_t Num_estados_analog = 8;
```

Signal parameters

On top of common parameters described in 3.5.1, it is required to specify the Arduino pin to which the push button assigned to each signal is connected:

```
pin_control[0] = 7;
```

In this example, signal 0 could be selected through a push button connected to Arduino pin 7.

Aspect push buttons

Here we define the pins to which the aspect selection push buttons are connected, through the "pin_estado[]" array. The total amount of elements in this array must be consistent with the value "Num_estados_analog" indicated above.

```
pin_estado[0] = 14;    // "Stop" aspect control pin
pin_estado[1] = 15;    // "Proceed" aspect control pin
pin_estado[2] = 16;    // "Proceed on condition" aspect control pin
pin_estado[3] = 17;    // "Precaution warning" aspect control pin
pin_estado[4] = 20;    // "Stop warning" aspect control pin
pin_estado[5] = 21;    // "Immediate stop warning" aspect control pin
pin_estado[6] = 0;     // "Authorized pass (stopping on signal)" aspect control pin
pin_estado[7] = 1;     // "Authorized movement" aspect control pin
```

See Chapter 4 to define aspect values.

4 USING THE PROGRAM

The code wakes up from idle state when receiving an entry command. It then builds up the array “comando[]”, which has as many elements as signals are connected to the system. For example:

```
comando[4] = 8
```

will assign the aspect “Stop” to signal number 4, according to Table 1 below. This table is coded by default, and contains all main aspects used in Spanish signals (RENFE/ADIF) since Epoch IV (roughly corresponding to the 70’s, see NEM809E) up to present day, although some of those aspects were present also in older signals. “Postponed stop” and “Departure indication” aspects are commented, as these are not defined in this version of the software.

Some signals require the use of a channel tagged with a different colour, to light up a board or a an already used colour light: this does not represent any problem. One example can be found in the “Distant stop warning” aspect, where normally the blue light channel will be used to light up the signal’s board. Other examples are the “Switch entrance indication” aspects, where white colour is repeated using another channel.

Aspect	Assigned aspect ID	Comments
Signal off	0	
Proceed	1	
Proceed on condition	2	
Precaution warning	3	
Distant stop warning	4	Board connected to blue channel
Stop warning	5	
Immediate stop warning	6	
Postponed stop	7	Not defined
Stop	8	
Authorized pass (stopping on signal)	9	
Authorized pass (no stopping on signal)	10	
Selective stop (fixed blue light)	11	
Selective stop (blinking blue light)	12	
Authorized movement (on white light signals)	13	
Authorized movement (on blue light signals)	14	
Switch entrance indication (straight track)	15	Upper white light connected to green channel
Switch entrance indication (diverging track)	16	Side white light connected to yellow channel
Departure indication	17	Not defined
Stop (on dwarf signals with two red lights)	18	Upper red light connected to green channel

Table 1: Aspect definition (RENFE/ADIF signals)

See Appendix 3 in case of needing to replace, modify, or extend this set of aspects.

The building of the array “comando[]” will depend on the system used (digital or analogue; see 3.1) and is detailed in the following sections.

4.1 DIGITAL VERSION SPECIFICS

Internal way of working

In a digital system, the Arduino board will receive from the decoder an array named “comando_recibido” through its main serial port (see note in 3.5.2) which will contain one element per signal. To ease the integration of possible future enhancements, this array is in fact a data structure named “comando_semaforos”, which so far only includes an integer value named estado. This value represents the aspect number according to Table 1.

```
struct comando_semaforos
{
    uint8_t estado;
};

comando_semaforos comando_recibido[Numero_semaforos];
```

(In this way, it will be possible to send additional information apart from the signal aspect, without requiring significant modifications in the code)

Then, the code compares the received command with each signal’s current aspect, and builds up the “comando[]” array to apply the new aspects, as follows:

```
comando[i] = comando_recibido[i].estado
```

(Being “i” each signal’s ID number)

Therefore, to get the system working, we just require the decoder to send the array “comando_recibido” in the right format.

Recommended decoders and command stations

To take benefit of the capabilities of this system (namely handling many different signals with a great variety of aspects) is **highly recommended to have at one’s disposal both a command station and a decoder supporting the “Signal Aspect” protocol**. This protocol allows controlling multi-aspect signals (see *Extended Accessory Decoder Control Packet Format* in [NRMA S-9.2.1](#)).

Configuration of any other protocol, although feasible, can be limited and way more complex, so it is not supported in this guide.

Regarding the decoder, see Appendix 1 for a simple design of a compatible DCC decoder, using a separate Arduino board.

For the command station, a simple and cheap option supporting this protocol could be a home-made mini-command DCC station made with Arduino (there are several sites on the Internet explaining how to do it). It is also possible to acquire a SPROG budget command station (which are intended for its use with personal computers, tablets, and mobile phones).

In case of controlling the layout with a computer, some programs allow the connection of several command stations. In this case, one of the budget command stations mentioned above could be used only to control the signals, leaving the main command station for driving the trains. This has some advantages, such as balancing energy consumption in big layouts.

Use with computer software (Rocrail, JMRI, etc)

To take benefit of all possibilities, it is also recommended to configure and handle this signal system using a computer or mobile device software (JMRI, Rocrail, etc), provided it is compatible with the protocol mentioned above. This will ease a lot not only the initial setup, but also the handling of complex signals.

In case of using Rocrail as controlling software, a ZIP file with SVG aspects for the most common Spanish RENFE/ADIF signals is included in this package. These SVG files are graphical representations of the signals to be shown in Rocrail's control panel. These are symbolic representations (and so are not pretending to be realistic) but serve the purpose of showing each type of signal with its correct aspects. There is a ReadMe text file with detailed installation instructions and prefixes for setting the signals up. As an example, Figure 8 shows a 3-light main signal showing the "Precaution warning" aspect.

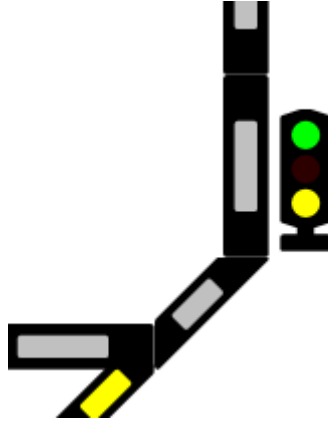


Figure 8: SVG representation of a signal in Rocrail

To use these SVG files in Rocrail, the folder inside the ZIP file must be copied into Rocrail installation's "/svg/themes/" folder. Then, we will need to modify some parameters in the program's configuration. First, in "Rockview Properties" - "SVG" tag, we will indicate the path to the folder we have just copied. We shall **add** the folder in a new line, without replacing any of the existing ones (see Figure 9).

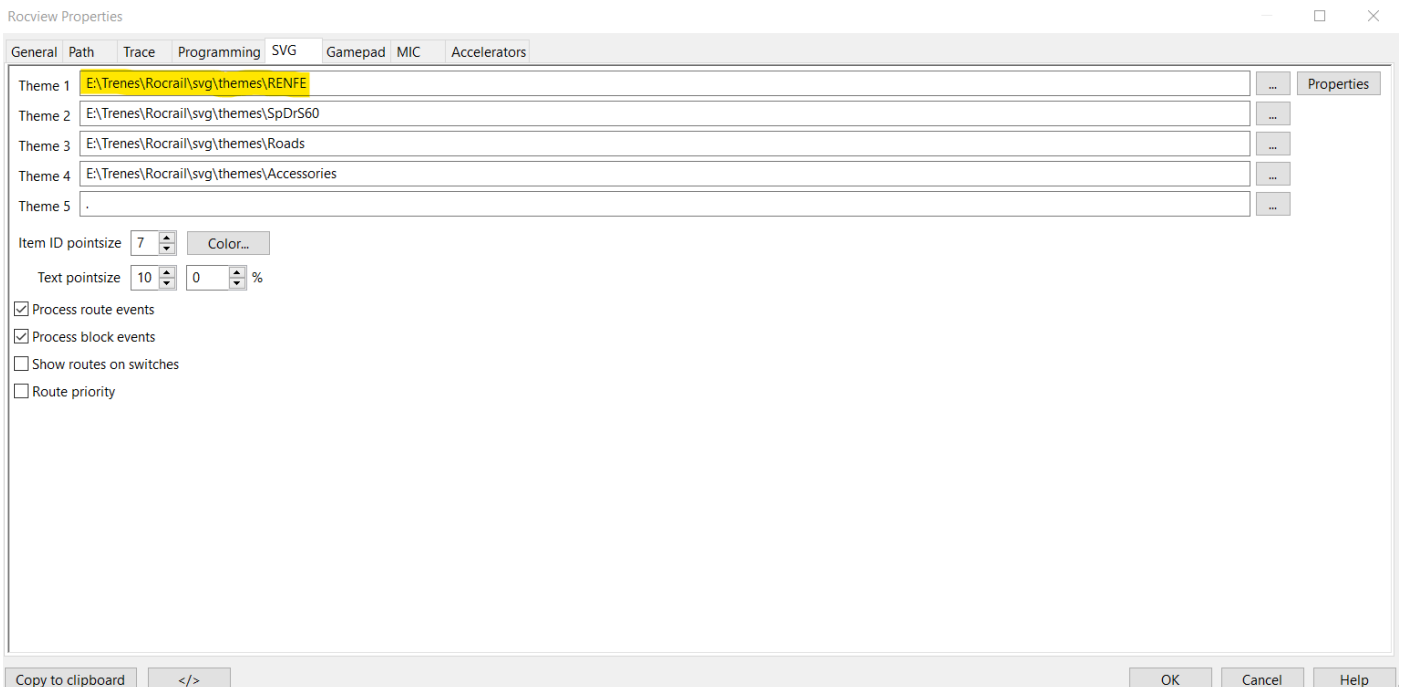


Figure 9: Rocrail setup

If we use a mobile device or a web browser to control the system, we will need also to configure the folder path in the section "Rocrail Properties" - "RocWeb" (see Figure 10).

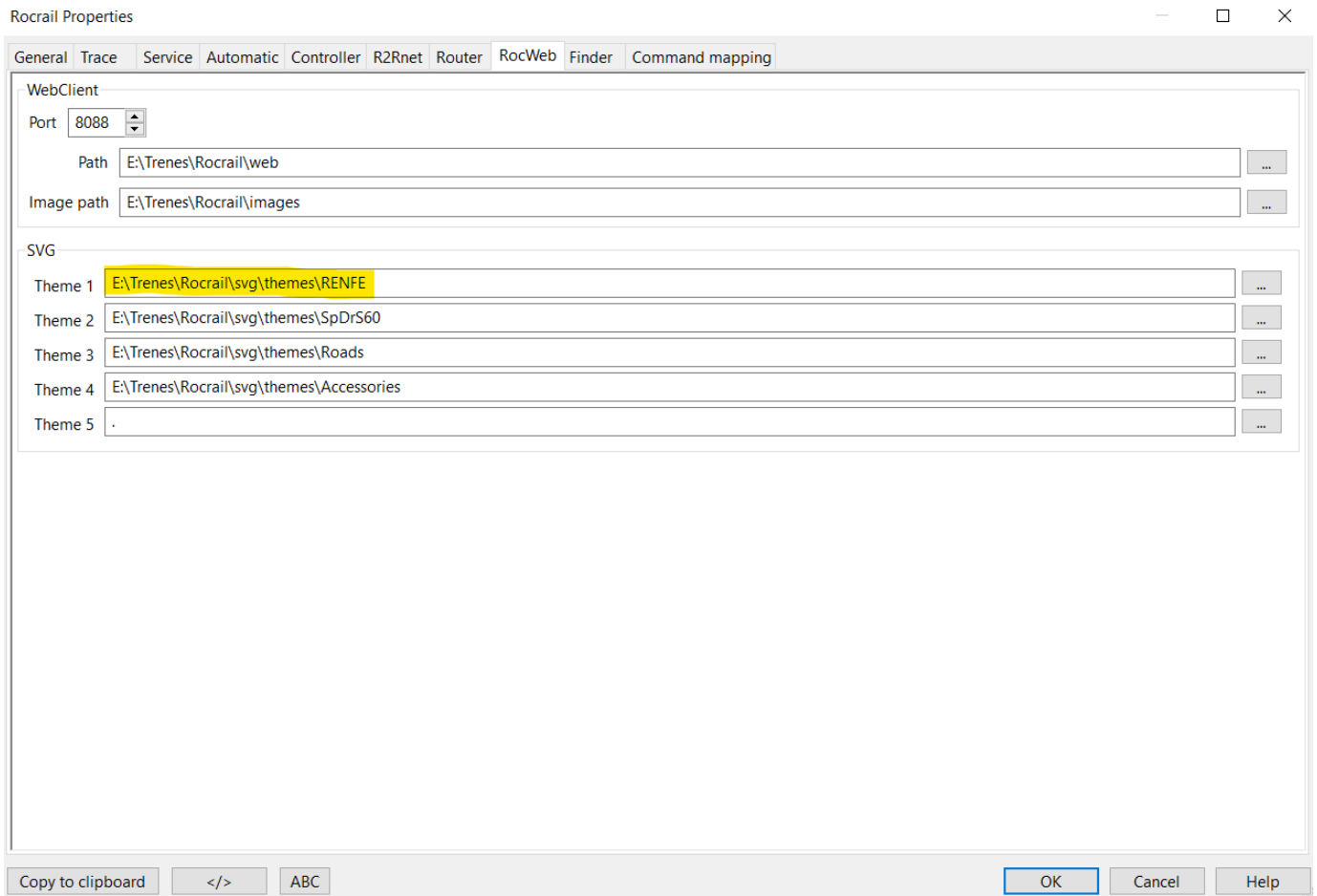


Figure 10: Rocview setup for web/mobile device

Next, we need to modify the properties of each signal in our panel layout. In the “Interface” section of each signal’s configuration page (see Figure 11) we need to mark the “Aspect numbers” control type and the “Accessory” tick-box. We will also specify **only one** DCC address for the signal in the “RED” field, leaving “Port” with value 1. The rest of colours will be set to 0 in all fields.

Now we will select the “Details” tag (see Figure 12) and we will fill in the details for each signal depending on its type, according to the SVG package’s ReadMe file. We will specify the maximum number of possible aspects for that signal in the “Aspects” box. We will tick the “Use prefix” box, and we will type the prefix corresponding to the signal type (again, as indicated in the ReadMe). If we are configuring a dwarf signal, we will obviously mark the “Dwarf” option. Finally, we will copy-paste all aspect names indicated in the ReadMe file for this signal into the “Aspect names” box, separated by commas.

Last but not least, if the signal is able to reproduce an aspect including blinking lights (such a case will be properly specified in the ReadMe file) we need to go back to the panel and select the option “Enable alternative SVG” by right-clicking on the signal.

Signal Semáforo 2 (4/11)

Index General Interface Wiring Details Usage

Interface ID SPROG DCS

Node ID 0 0x00000000 UID-Name

RED
Address 7 Port 1 ☒ red ☐ green

GREEN
0 0 ☒ red ☐ green

YELLOW
0 0 ☒ red ☐ green

WHITE
0 0 ☒ red ☐ green

Protocol Default

Dim 10

Brightness 100

Parameter 0

Control
☐ Default
☐ Patterns
☒ Aspect numbers
☐ Linear
☐ Binary
☐ Function

☒ Accessory
Type
☒ Output
☐ Lights
☐ Servo
☐ Sound
☐ Motor
☐ Analog
☐ Macro
☐ Backlight
☐ LED

☒ Invert ☐ Pair gates ☒ Switch ☐ Switch time 0 ms

Command time 0 ms

< > </> + ABC OK Cancel Apply Help

Figure 11: Configuring a signal in Rocrail (1)

Signal Semáforo 2 (4/11)

Index General Interface Wiring Details Usage

Signal type
☒ Semaphore signal
☐ Light signal

Signification
☐ Distant signal ☒ Main signal
☐ Shunting signal ☐ Block state

Aspects 7

Prefix m_s3_1-

☐ Dwarf signal
☒ Use prefix

Patterns / Aspects

	RED Address	GREEN Address	YELLOW Address	NumberValue	NumberValue
RED	<input checked="" type="radio"/> R1 <input type="radio"/> G1 <input type="radio"/> N	<input checked="" type="radio"/> R2 <input type="radio"/> G2 <input type="radio"/> N	<input checked="" type="radio"/> R3 <input type="radio"/> G3 <input type="radio"/> N	0 0	0 0
GREEN	<input checked="" type="radio"/> R1 <input type="radio"/> G1 <input type="radio"/> N	<input checked="" type="radio"/> R2 <input type="radio"/> G2 <input type="radio"/> N	<input checked="" type="radio"/> R3 <input type="radio"/> G3 <input type="radio"/> N	0 0	0 0
YELLOW	<input checked="" type="radio"/> R1 <input type="radio"/> G1 <input type="radio"/> N	<input checked="" type="radio"/> R2 <input type="radio"/> G2 <input type="radio"/> N	<input checked="" type="radio"/> R3 <input type="radio"/> G3 <input type="radio"/> N	0 0	0 0
WHITE	<input checked="" type="radio"/> R1 <input type="radio"/> G1 <input type="radio"/> N	<input checked="" type="radio"/> R2 <input type="radio"/> G2 <input type="radio"/> N	<input checked="" type="radio"/> R3 <input type="radio"/> G3 <input type="radio"/> N	0 0	0 0
BLANK	<input checked="" type="radio"/> R1 <input type="radio"/> G1 <input type="radio"/> N	<input checked="" type="radio"/> R2 <input type="radio"/> G2 <input type="radio"/> N	<input checked="" type="radio"/> R3 <input type="radio"/> G3 <input type="radio"/> N	0 0	0 0

Aspect names Parada,Via libre,Via libre condicional,Anuncio de precaucion,Anuncio de parada,Anuncio de parada inmediata,Apagado

< > </> + ABC OK Cancel Apply Help

Figure 12: Configuring a signal in Rocrail (2)

4.2 ANALOGUE VERSION SPECIFICS

In an analogue layout, everything will depend on the command entry method and the automation systems installed, as it was seen in section 3.1.2.

If default manual mode is used (as explained in 3.1.2.1), we will push first one of the buttons to select the signal we want to command, and then one of the aspect-selection buttons to tell the system the aspect we want on that signal. For example, let us suppose we have a layout with 12 signals and 8 possible aspects. Each one of the signals will have an associated button to select it (12 in total) and each one of the aspects will have an associated button as well (8 in total, labelled as “Stop”, “Proceed”, and so on). If we wanted to assign the aspect “Stop” to the signal #7, we would push first the associated button to that signal (selecting it) and then the button “Stop”.

Any other way of working will depend heavily on the particular configuration and automation of each layout, and so will require specific programming.

5 TROUBLESHOOTING

Common tips:

- Check wiring is according to the code (each cable must be attached to its proper I/O pin).
- Check the parameters in the code for the number of boards and signals match the actual ones installed.
- Check the parameters in the code for the number of boards and signals match the listings of properties for each one.

In digital layouts:

- Check the array size sent by the decoder device matches the actual number of signals expected by the Arduino board, arranged in proper order, and with the same aspect numbering.
- Check the serial communication is properly set-up between the decoder device and the signal control Arduino board, the wires between both are crossed ("TX1" and "RX0" if both are Arduino boards) and the baud rate is set to the same value in both devices.

In analogue layouts (manual mode):

- Check the parameters in the code for the number of push buttons for signal and aspect selection match the actual ones installed.
- Check the parameters in the code for the number of push buttons for signal and aspect selection match the listings of properties for each one.

APPENDIX 1: ARDUINO DCC DECODER

Should we want to decode the DCC signal using an Arduino board, we need to:

- 1) Install an opto-isolator circuit in between the DCC output (normally the tracks) and the Arduino.
- 2) Install a decoding program on the Arduino decoder.

An Arduino Nano board will suffice for this purpose.



Figure 13: Decoder connection schematics

NOTE: It is highly recommended to use a different Arduino board for decoding tasks than the one used to control the signals. Although it is possible to use a single board for everything, this imposes some limitations (see 3.1.1).

Opto-isolator circuit

This circuit is required since the DCC tension from the command station or the tracks is often higher (12-18 V in usual gauges) than the maximum admitted by an Arduino board (5 V). The purpose of this circuit is to electrically isolate both sides, but allowing signal passthrough. Many schemes can be found on the Internet, all are quite similar. See [here](#) for a good example.

It is possible to assembly the opto-isolator circuit on a simple practice breadboard, but a simple printed circuit board design in Gerber format ("DCC_adaptor-Gerber.zip") is attached inside the folder "PCB", allowing to build a better-looking circuit.

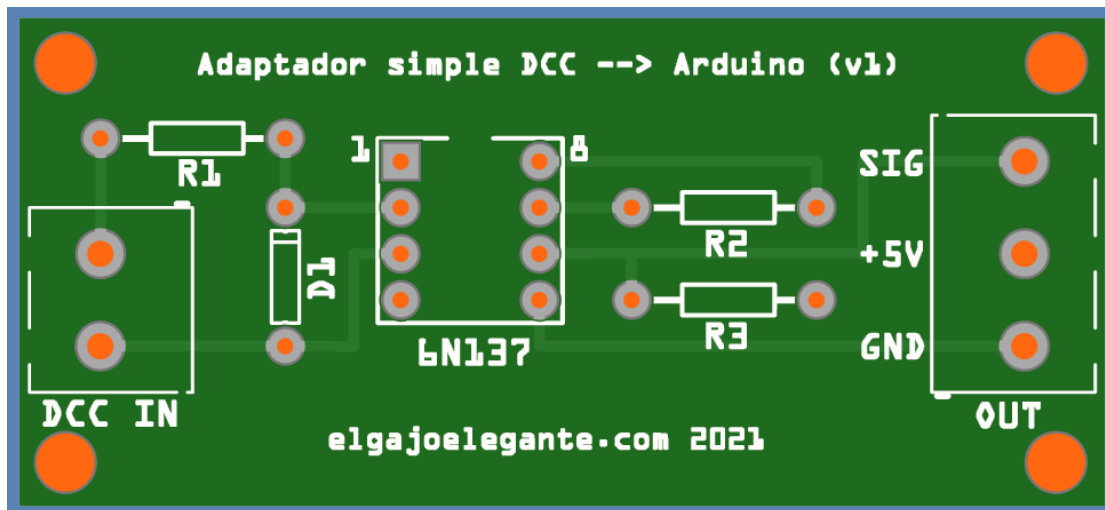


Figure 14: Opto-isolator printed circuit board

Required components will be as follows:

- D1: A typical LED (to act as a control light when receiving DCC tension) or 1N4148 (if DCC control light is not required)
- R1: Resistor, value depending on track/DCC tension and command station type. Usually, 1 kΩ will be adequate for typical track tensions (9-14 V); and 2.2 kΩ for higher ones (16 or 18 V)

- 6N137: Opto-isolator integrated circuit 6N137
- R2: 10 k Ω resistor
- R3: 10 k Ω resistor
- 1x 2-pin connector, 5.08 mm pitch (DCC input)
- 1x 3-pin connector, 5.08 mm pitch (output to the Arduino decoder)

Tracks or DCC output will be connected to “DCC IN”, and the Arduino decoder to “OUT” as follows:

- “SIG” (signal) will be connected to the Arduino decoder’s input pin (normally pin 2, see below)
- “+5V” will be connected to the Arduino decoder’s 5V pin
- “GND” will be connected to the Arduino decoder’s “GND” pin

Arduino decoder sketch

We will assume here that the Arduino board performing as a decoder is not doing any other tasks.

There are several sketches on the Internet, based on many different libraries, each one with its own advantages and drawbacks. In the “decoder” folder accompanying to this document there is one example based on the “NmraDcc.h” library (its original version can be found [here](#)).

This sketch uses Arduino’s interruption 0, which is generally associated to pin 2, so this will be the default input pin for the signal coming from the opto-isolator circuit.

Parameters to be configured before uploading the sketch are detailed as follows:

Installed signals and possible aspects

- Total number of different possible aspects displayed by the installed signals: integer value between 1 and 32. It must be consistent with “diccionario_tipos” function, and with the signal definition on the control boards:

```
const uint8_t Numero_estados = 19;
```

- Number of connected signals (integer value; maximum 255). Note that the limit is higher than 12, as we can connect more than one signal control board to each decoder, using several serial ports:

```
const uint8_t Numero_semaforos = 12;
```

DCC function parameters

- DCC address mode. There are two options (lines): select one **and comment the other**. The first one corresponds to “Board Addressing” mode (one DCC address with several sub-addresses for each pair of lights in the signal) and the second corresponds to “Output addressing” mode (unique DCC address for each pair of lights in the signal):

```
Dcc.init( MAN_ID_DIY, 10, CV29_ACCESSORY_DECODER, 0 );
Dcc.init( MAN_ID_DIY, 10, CV29_ACCESSORY_DECODER | CV29_OUTPUT_ADDRESS_MODE, 0 );
```

This setting will have no effect if the system uses a third mode, “Signal Aspect” (which is the mode supported in our control system) but we need anyway to comment one of those two lines. The first one is commented by default (“Output addressing” is therefore selected).

Signal parameters

Signal parameters will be stored into the “semaforo[]” structure array, including the signal type, its DCC address and its initial aspect. The signal type (`.tipo`) is an integer number which must be consistent with the definition in the

“diccionario_tipos” function (see below). The DCC address (.direccion_DCC) is also an integer number, that will match the one assigned to the signal in the command station. Note that we assume a “Signal Aspect” working mode, as the DCC address will be unique for each signal. Finally, the initial aspect (.estado_inicial) that the signal will show when turning on the whole system, must be consistent with the aspect definition on the signal control board.

So, for each “semaforo[]” element (counting from 0), there will be a block like the following one:

```
// Signal 7:
// Reference: Mafen 4131.11 (RENFE 3-light main signal, type 2)

semaforo[7].direccion_DCC = 19;
semaforo[7].tipo = 2;
semaforo[7].estado_inicial = 2;
```

There will be as many of these blocks as the number of signals defined in “Numero_semaforos” parameter, so unused blocks will be commented, or more blocks will be added if needed.

In the example above, signal number 7 is a type 2 signal (main 3-light signal), its DCC address is 19, and its initial aspect is 2 (proceed on condition).

Signal type dictionary

The “Signal Packet” standard is based on defining a unique DCC address for each signal, to which the system will send an integer number between 0 and 31, indicating the commanded aspect for that signal. Many command stations or control programs supporting the “Signal Packet” standard do not, however, allow to choose freely the number for each aspect. So, if a signal has seven possible aspects, these will be always numbered from 0 to 6. This is the case when using Rocrail, for instance.

Problems arise if we have several types of signal mixed in the same system: it can be impossible to set up an aspect list for each type of signal, in a way that numbers are consecutive starting from 0. For example, if we have a signal capable to show three aspects (stop, proceed, and precaution warning) these will be numbered from 0 to 2 (at this point we could select the desired order). But if we have another signal, capable of showing two aspects only (precaution warning and authorized movement) these will have to be numbered from 0 to 1. And there will be no way to match both signals in the same system, no matter how we play with the order, as the aspects are different, but the numbering convention is the same. On top of that, the NMRA standard recommends the first aspect (0) to be always “Stop”, making things even more complex.

The solution is to introduce a function in the decoder program, acting as a “dictionary” between the command from the station and the instruction sent to the signal control board with the signal aspect number (which will be unique for that signal system; see example in Table 1). This function is named “diccionario_tipos”. Values defined in this function will also allow to indicate the signal type (as seen in previous paragraphs): each signal model will correspond to a case value in this function.

This function is defined by default for the most typical Spanish RENFE/ADIF signals, consistently with the rest of the system, and normally it is not needed to modify it unless we want to change the signal system to a different railroad company, or we want to add/remove/modify signal types. See Appendix 3 for instructions to do this.

APPENDIX 2: COMMON-ANODE CONVERTER CIRCUIT

One of these circuits will be needed per signal, and each circuit will have as many branches as lights has the signal. The following list of materials is to be procured per signal:

- 1x 1N4148 diode per each light on the signal
- 1x BC557 transistor per each light on the signal
- 1x 10 k Ω resistor per each light on the signal

Figure 15 shows an example of a three-light signal circuit (branches red, green, and yellow, numbered from 1 to 3) but it can be extended or reduced as required by the number of lights in the signal, just by following the same principle (adding or removing branches). Note that the negative input connection is in fact a unique common input, although it has been represented by one “ground” symbol per branch for the sake of simplicity. The PCA9685 outputs will be connected to this circuit’s inputs (“GND” to the common negative input and “PWR” to each one of the positive inputs). The signal will be connected to the outputs in the figure, **of course including the adequate resistors (or modifying the existing ones in the signal)** between each LED and the negative (-) outputs in the circuit (since this circuit does not reduce tension or current output to the signal).

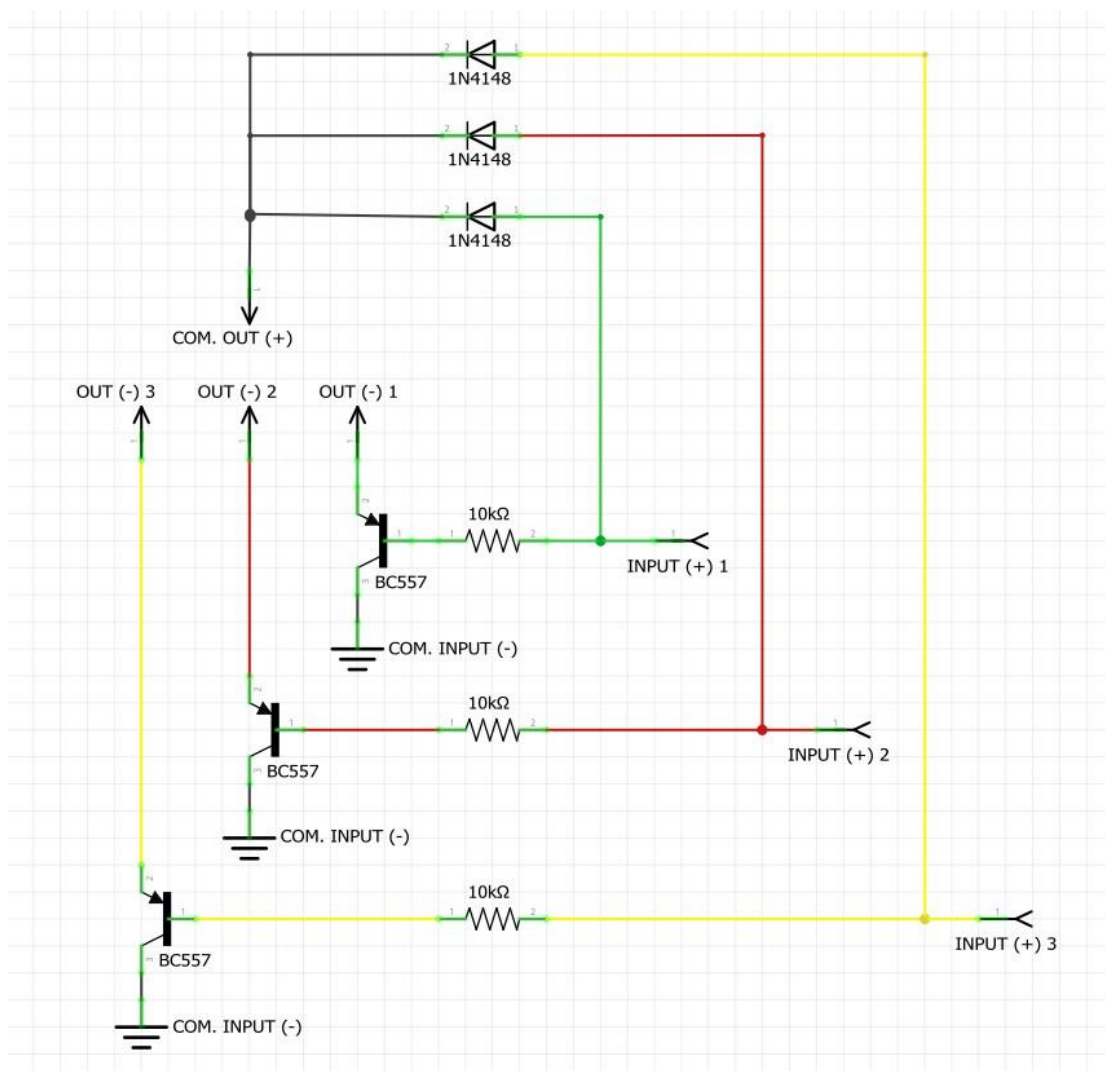


Figure 15: Common-anode converter schematics

This simple converter can be mounted onto a prototyping breadboard, or better onto a customized PCB. Some simple printed circuit board designs for these converters (created with KiCad free software) are included into the “PCB” folder: there are models for signals with 2, 3 and 4 lights, named as “Adaptador_anodo-Anode_adaptor”. In these boards, lights have been numbered from 1 to 4 to avoid confusion with the different colours a signal may have.

These designs include an area to place the adequate resistors for each LED light, up to a maximum of two resistors (“a” and “b”) per light in serial configuration (which will cover practically any need). However, in most cases a single resistor will suffice. In that case, the single resistor can be soldered in diagonal following the line marked as “R eq” (as seen in Figure 16).

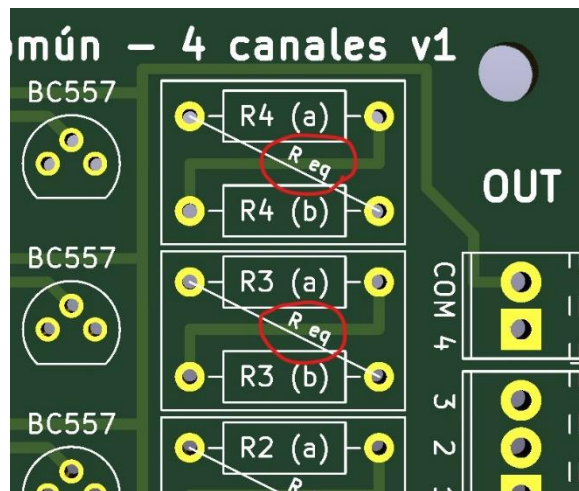


Figure 16: Detail of output resistor connections in the attached PCB designs

APPENDIX 3: MODIFYING THE SIGNAL SYSTEM

Should a modification of the existing RENFE/ADIF aspects is required, or a different signal system (i.e., from a different railroad administration) wants to be represented, the following changes will be required (both in main and decoder programs):

Modifications to be applied in the main control program

The function "nuevo_estado" needs to be rewritten, defining the new aspects. Up to 255 different aspects are supported.

The structure of this function corresponds to a switch() type, with as many cases as aspect numbers considered (case 0, case 1, etc) which will indicate each light's aspect according to the values of "control" and "flag_parpadeo" in Table 2. Note that argument "indice" is an internal variable, and therefore must remain as it is.

"control[indice][light]" value	"flag_parpadeo" value	Light status
0	false	Light off
5	false	Light on
5	true	Light on (blinking)

Table 2: Control variables for each light

Finally, after defining each aspect's lights, it is required to add the value of "focos_activos", indicating the total number of active lights (lights on or blinking) in the signal.

An example is shown hereafter, where an aspect is designated with number 10. In this aspect, red light (1) is on, and white light (3) is blinking, with the rest of lights off. Therefore, the total number of active lights would be two:

```
case 10:      // --- Authorized pass (no stopping on signal) ---

    control[indice][1]      = 5;          // Red light on
    flag_parpadeo[indice][1] = false;
    control[indice][3]      = 5;          // White light blinking
    flag_parpadeo[indice][3] = true;

    focos_activos[indice] = 2;

    break;
```

Modifications to be applied in the decoder program

In this sketch, first it is required to change the value of "Numero_estados" with the new total number of possible aspects.

Then, the function "diccionario_tipos" will be modified to define the new signal types and the associated translation table: from the consecutive list of aspects sent by the command station (see Appendix 1) to the aspects defined in the main control program (see above).

The structure of the "diccionario_tipos" function corresponds to a switch() type, with as many cases as different signal types are defined (case 0, case 1, etc). The total number of aspects that can be represented by this signal will be indicated in

the “aspectos” variable. Then, a list will be created with the so-called translation table between the aspect numbering sent by the command station and the numbering used by the main control program (in a similar manner as defined in Table 1).

```
case 8:
```

```
    // RENFE/ADIF dwarf signal with 2 lights: red, white
    // Example reference: Mafen 4141.02
    // 5 possible aspects

    aspectos = 5;

    semaforo[id_semaforo].aspecto[0] = 8; // Stop
    semaforo[id_semaforo].aspecto[1] = 9; // Authorized pass (stopping on signal)
    semaforo[id_semaforo].aspecto[2] = 10; // Authorized pass (no stopping on signal)
    semaforo[id_semaforo].aspecto[3] = 13; // Authorized movement
    semaforo[id_semaforo].aspecto[4] = 0; // Signal off

    break;
```

In the example above, the signal accepts 5 possible aspects. The command station will number those aspects consecutively, from 0 to 4, so there will be five lines with “semaforo[id_semaforo].aspecto[]” values, indicating the number for each aspect in the main control program. In this manner, the third aspect in this signal (Authorized pass - no stopping on signal) will correspond to number 2 sent by the command station, and number 10 in the main control program (Table 1).