
esphomelib Documentation

Release 1.6.2

Otto Winter

Jun 07, 2018

CONTENTS

1	Welcome to the esphomelib API docs!	1
2	Web Server API	3
3	Indices and tables	5
3.1	esphomeyaml	5
3.2	Web Server API	176
3.3	API Reference	179
Index		357

WELCOME TO THE ESPHOMELIB API DOCS!

esphomeyaml docs can be found here: [esphomeyaml docs](#).

Here you will find all of esphomelib's API documentation together with some examples.

Eventually, guides will also be setup here on how to create your own custom components and some basic stuff.

CHAPTER

TWO

WEB SERVER API

See [Web Server API](#) for documentation of the web server REST API.

INDICES AND TABLES

3.1 esphomeyaml

Hi there! This is the documentation for esphomeyaml, a project that aims to make using ESP8266/ESP32 boards with Home Assistant very easy with no programming experience required.

esphomeyaml will:

- Read your configuration file and warn you about potential errors (like using the invalid pins.)
- Create a custom C++ sketch file for you using esphomeyaml's powerful C++ generation engine.
- Compile the sketch file for you using [platformio](#).
- Upload the binary to your ESP via Over the Air updates.
- Automatically start remote logs via MQTT.

3.1.1 Features

- **No programming experience required:** just edit YAML configuration files like you're used to with Home Assistant.
- **Flexible:** Use [esphomelib](#)'s powerful core to create custom sensors/outputs.
- **Fast and efficient:** Written in C++ and keeps memory consumption to a minimum.
- **Small binaries:** Only the sensors/devices you actually use will appear in the binary.
- **Made for Home Assistant:** Almost all Home Assistant features are supported out of the box. Including RGB lights and many more.
- **Easy reproducible configuration:** No need to go through a long setup process for every single node. Just copy a configuration file and run a single command.
- **Smart Over The Air Updates:** esphomeyaml has OTA updates deeply integrated into the system. It even automatically enters a recovery mode if a boot loop is detected.
- **Powerful logging engine:** View colorful logs and debug issues remotely.
- **It's Open Source**

3.1.2 Guides

Getting Started through Command Line	Getting Started through HassIO Add-On	Configuration Types
Migrating from Sonoff-Tasmota	Migrating from ESPurna	Migrating from ESPeasy
Automations	FAQ and Tips	Contributing

Devices

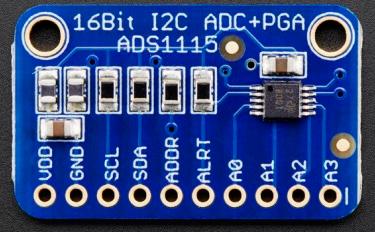
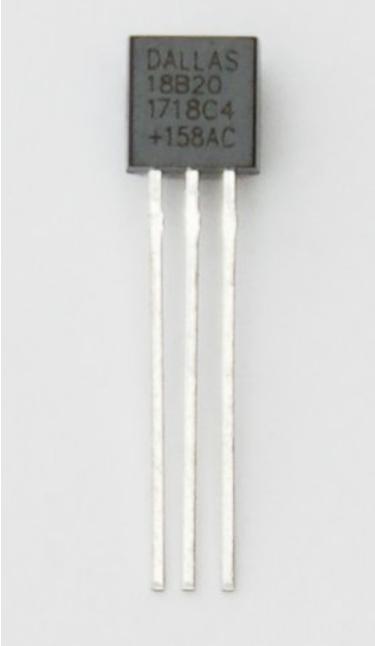
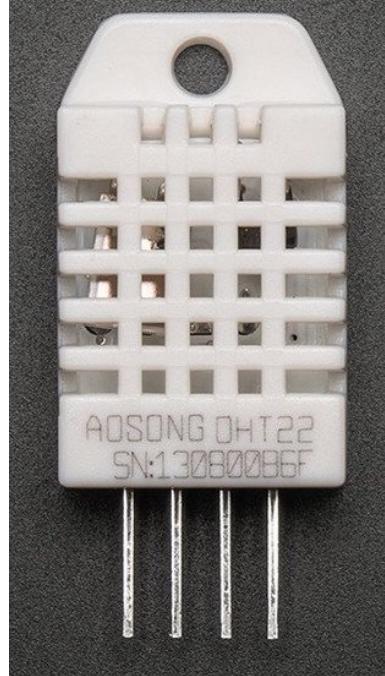
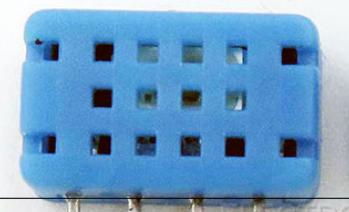
This list contains a bunch of getting started and more advanced guides for using esphomeyaml. Technically, all ESP8266/ESP32 devices (and therefore all Sonoff devices) are supported by esphomeyaml/lib. These are only the devices for which I've had the time to set up dedicated guides.

		
Generic ESP8266	Generic ESP32	NodeMCU ESP8266
		
NodeMCU ESP32	Sonoff S20	Sonoff 4CH
		
Generic Sonoff	Sonoff Basic	

3.1.3 Core Components

		
Core	WiFi	MQTT
I ² C Bus	OTA Updates	Logger
Web Server	Power Supply	Deep Sleep

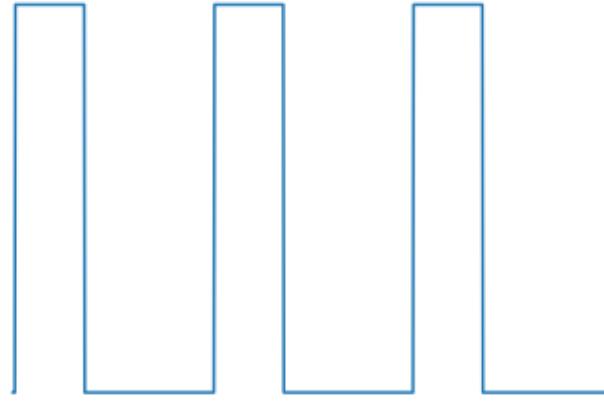
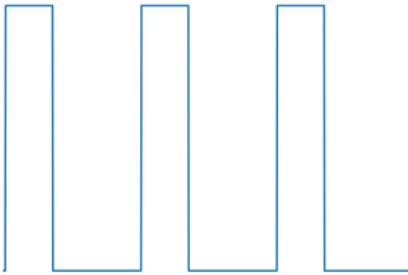
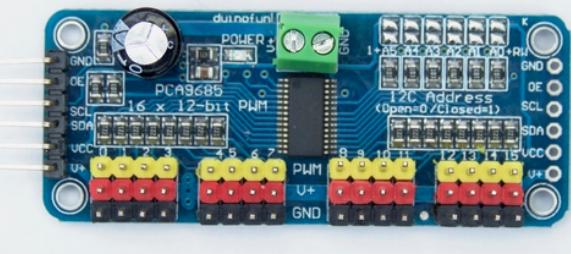
3.1.4 Sensor Components

		
Sensor Core	ADC	ADS1115
		
BH1750	BME280	BME680
		
BMP085	Dallas	DHT
		
10		

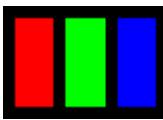
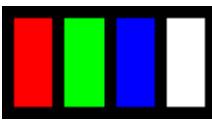
3.1.5 Binary Sensor Components

Binary Sensor Core	GPIO	Status
ESP32 BLE Device	ESP32 Touch Pad	Template Binary Sensor

3.1.6 Output Components

		
Output Core	ESP8266 Software PWM	GPIO Output
		
ESP32 LEDC	PCA9685	

3.1.7 Light Components

Light Core	Binary Light	Monochromatic Light
		
RGB Light	RGBW Light	FastLED Clockless Light
FastLED SPI Light		

3.1.8 Switch Components

Switch Core	GPIO Switch	IR Transmitter
Restart Switch	Shutdown Switch	Generic Output Switch
Template Switch		

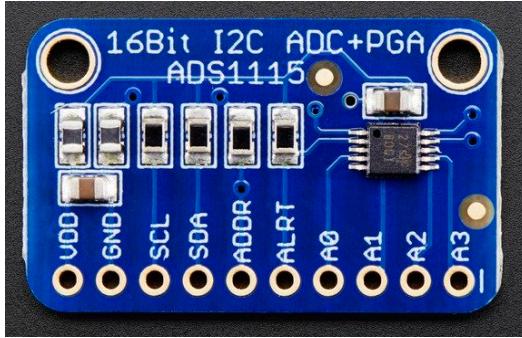
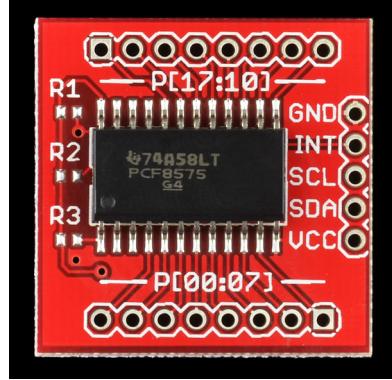
3.1.9 Fan Components

Fan Core	Binary Fan	Speed Fan

Cover Components

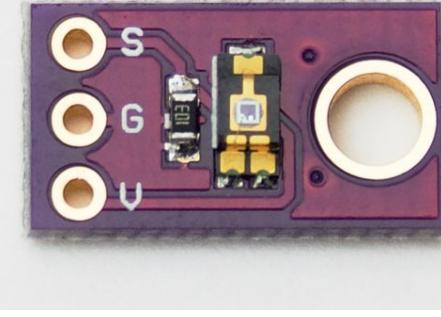
Cover Core	Template Cover	

3.1.10 Misc Components

		
Dallas Hub	IR Transmitter Hub	PCA9685 Hub
		
ADS1115 Hub	Debug Component	PCF8574 I/O Expander
3.1. esphomeyaml		15
ESP32 BLE Hub	ESP32 Touch	

Cookbook

This list contains items that are technically already supported by other components.

		
Garage Door	PIR Sensor	Relay
		

Do you have other awesome automations or 2nd order components? Please feel free to add them to the documentation for others to copy. See [Contributing](#).

Download the documentation as a PDF

Components

Binary Sensor Component

With esphomelib you can use different types of binary sensors. They will automatically appear in the Home Assistant front-end and have several configuration options.

Base Binary Sensor Configuration

All binary sensors have a platform and an optional device class. By default, the binary will chose the appropriate device class itself, but you can always override it.

```
binary_sensor:
  - platform: ...
    device_class: motion
```

Configuration variables:

- **device_class** (*Optional*, string): The device class for the sensor. See https://www.home-assistant.io/components/binary_sensor/ for a list of available options.
- **inverted** (*Optional*, boolean): Whether to invert the binary sensor output, i.e. report ON states as OFF and vice versa. Defaults to `False`.
- **on_press** (*Optional*, [Automation](#)): An automation to perform when the button is pressed. See [on_press](#).
- **on_release** (*Optional*, [Automation](#)): An automation to perform when the button is released. See [on_release](#).
- **on_click** (*Optional*, [Automation](#)): An automation to perform when the button is held down for a specified period of time. See [on_click](#).
- **on_double_click** (*Optional*, [Automation](#)): An automation to perform when the button is pressed twice for specified periods of time. See [on_double_click](#).
- All other options from [MQTT Component](#).

Binary Sensor Automation

The triggers for binary sensors in esphomeyaml use the lingo from computer mouses. This naming might not perfectly fit every use case, but at least makes the naming consistent. For example, a `press` is triggered in the first moment when the button on your mouse is pushed down.

You can access the current state of the binary sensor in [lambdas](#) using `id(binary_sensor_id).value`.

`on_press`

This automation will be triggered when the button is first pressed down, or in other words on the leading edge of the signal.

```
binary_sensor:
  - platform: gpio
    # ...
  on_press:
    then:
      - switch.turn_on:
          id: relay_1
```

Configuration variables: See [Automation](#).

`on_release`

This automation will be triggered when a button press ends, or in other words on the falling edge of the signal.

```
binary_sensor:  
  - platform: gpio  
    # ...  
    on_release:  
      then:  
        - switch.turn_off:  
          id: relay_1
```

Configuration variables: See [Automation](#).

on_click

This automation will be triggered when a button is pressed down for a time period of length `min_length` to `max_length`. Any click longer or shorter than this will not trigger the automation. The automation is therefore also triggered on the falling edge of the signal.

```
binary_sensor:  
  - platform: gpio  
    # ...  
    on_click:  
      min_length: 50ms  
      max_length: 350ms  
      then:  
        - switch.turn_off:  
          id: relay_1
```

Configuration variables:

- **min_length** (*Optional, Time*): The minimum duration the click should last. Defaults to 50ms.
- **max_length** (*Optional, Time*): The maximum duration the click should last. Defaults to 350ms.
- See [Automation](#).

on_double_click

Warning: This trigger is not final and could be replaced by a `on_multi_click` option which would allow triggering for any number of clicks.

This automation will be triggered when a button is pressed down twice, with the first click lasting between `min_length` and `max_length`. When a second leading edge then happens within `min_length` and `max_length`, the automation is triggered.

```
binary_sensor:  
  - platform: gpio  
    # ...  
    on_double_click:  
      min_length: 50ms  
      max_length: 350ms  
      then:  
        - switch.turn_off:  
          id: relay_1
```

Configuration variables:

- **min_length** (*Optional, Time*): The minimum duration the click should last. Defaults to 50ms.
- **max_length** (*Optional, Time*): The maximum duration the click should last. Defaults to 350ms.
- See [Automation](#).

lambda calls

From [lambdas](#), you can call several methods on all binary sensors to do some advanced stuff (see the full [API Reference](#) for more info).

- **publish_state()**: Manually cause the binary sensor to publish and store a state from anywhere in the program.

```
// Within lambda, publish an OFF state.
id(my_binary_sensor).publish_state(false);

// Within lambda, publish an ON state.
id(my_binary_sensor).publish_state(true);
```

- **value**: Retrieve the current value of the binary sensor.

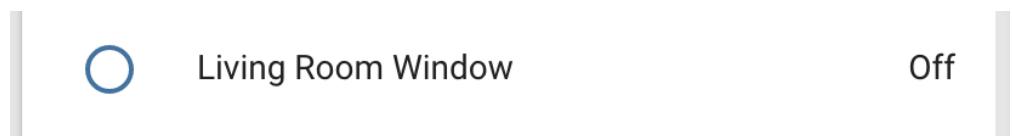
```
// Within lambda, get the binary sensor state and conditionally do something
if (id(my_binary_sensor).value) {
    // Binary sensor is ON, do something here
} else {
    // Binary sensor is OFF, do something else here
}
```

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

GPIO Binary Sensor

The GPIO Binary Sensor platform allows you to use any input pin on your device as a binary sensor.



```
# Example configuration entry
binary_sensor:
- platform: gpio
  pin: D2
  name: "Living Room Window"
  device_class: window
```

Configuration variables:

- **pin** (**Required**, *Pin Schema*): The pin to periodically check.
- **name** (**Required**, string): The name of the binary sensor.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Binary Sensor* and *MQTT Component*.

See Also

- *Binary Sensor Component*
- *Pin Schema*
- *API Reference*
- Edit this page on GitHub

Status Binary Sensor

The Status Binary Sensor exposes the node state (if it's connected to MQTT or not) for Home Assistant. It uses the *MQTT birth and last will messages* to do this.



Living Room Status

Connected

```
# Example configuration entry
binary_sensor:
  - platform: status
    name: "Living Room Status"
```

Configuration variables:

- **name** (**Required**, string): The name of the binary sensor.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Binary Sensor* and *MQTT Component*. (Inverted mode is not supported)

See Also

- *Binary Sensor Component*
- *MQTT Client Component*
- *API Reference*
- Edit this page on GitHub

ESP32 Bluetooth Low Energy Device

The `esp32_ble` binary sensor platform lets you track the presence of a bluetooth low energy device.

Note: See the [ESP32 BLE Hub Page](#) for current limitations of this platform



ESP32 BLE Tracker Google Home Mini

Away

```
# Example configuration entry
esp32_ble:
    scan_interval: 300s

binary_sensor:
- platform: esp32_ble
  mac_address: AC:37:43:77:5F:4C
  name: "ESP32 BLE Tracker Google Home Mini"
```

Configuration variables:

- **mac_address** (**Required**, MAC Address): The MAC address to track for this binary sensor.
- **name** (**Required**, string): The name of the binary sensor.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from [Binary Sensor](#) and [MQTT Component](#).

Setting Up Devices

To set up binary sensors for specific BLE beacons you first have to know which MAC address to track. Most devices show this screen in some setting menu. If you don't know the MAC address, however, you can use the `esp32_ble` hub without any binary sensors attached and read through the logs to see discovered Bluetooth Low Energy devices.

```
# Example configuration entry for finding MAC addresses
esp32_ble:
```

Using above configuration, first you should see a `Starting scan...` debug message at boot-up. Then, when a BLE device is discovered, you should see messages like `Found device AC:37:43:77:5F:4C RSSI=-80` together with some information about their address type and advertised name. If you don't see these messages, your device is unfortunately currently not supported.

Please note that devices that show a `RANDOM` address type in the logs cannot be used for tracking, since their MAC-address periodically changes.

See Also

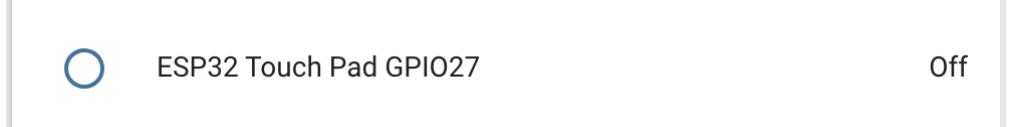
- [ESP32 Bluetooth Low Energy Hub](#)

- [Binary Sensor Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

ESP32 Touch Pad Binary Sensor

The `esp32_touch` binary sensor platform lets you use the touch peripheral of the ESP32 to detect if a certain pin is being “touched”.

First, you need to setup the [global touch hub](#). Then you can add individual touch pads as binary sensors. When a touch is detected on these pins, the binary sensor will report an **ON** state. And, of course, if no touch is detected, the binary sensor will report an **OFF** state.



ESP32 Touch Pad GPIO27

Off

```
# Example configuration entry
esp32_touch:

binary_sensor:
  - platform: esp32_touch
    name: "ESP32 Touch Pad GPIO27"
    pin: GPIO27
    threshold: 1000
```

Configuration variables:

- **pin** (**Required**, *Pin*): The pin to detect touch events on.
- **threshold** (**Required**, int): The threshold to use to detect touch events. Smaller values mean a higher probability that the pad is being touched.
- **name** (**Required**, string): The name of the binary sensor.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from [Binary Sensor](#) and [MQTT Component](#).

Touch Pad Pins

8 pins on the ESP32 can be used to detect touches. These are (in the default “raw” pin names):

- GPIO0
- GPIO2
- GPIO4
- GPIO12
- GPIO13
- GPIO14

- GPIO15
- GPIO27
- GPIO32
- GPIO33

Finding thresholds

For each touch pad you want to monitor, you need to find a threshold first. This threshold is used to determine if a pad is being touched or not using the raw values from the sensor. Lower raw values mean that it is more likely that a touch is happening. For example, values around 1000 to 1600 usually mean the pad is not being touched, and values in the range of 600 and less mean the pad is probably being touched.

To find suitable threshold values, first configure the *ESP32 touch hub* to output measured values using the `setup_mode`: configuration option. Next, add some binary sensors for the touch pads you want to observe. Also put some threshold in the configuration as seen below to make the validator happy, we are going to find good thresholds in a moment anyway.

```
# Example configuration entry for finding threshold values
esp32_touch:
  setup_mode: True

binary_sensor:
  - platform: esp32_touch
    name: "ESP32 Touch Pad GPIO27"
    pin: GPIO27
    threshold: 1000
```

Then upload the program and open the logs, you will see values like these. Try touching the pins and you will (hopefully) see the value decreasing a bit. Play around with different amounts of force you put on the touch pad until you find a good value that can differentiate between touch/non-touch events.

```
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 1274
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 1272
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 1269
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 1263
[V][mqtt.client:193]: Publish(topic='test/binary_sensor/esp32_touch_pad_gpio27/state' payload='ON' retain=1)
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 228
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 40
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 38
[D][binary_sensor.esp32_touch:100]: Touch Pad 'ESP32 Touch Pad GPIO27' (T7): 43
```

Finally, put your threshold parameter in the configuration. Do not forget to disable the `setup_mode` option again by setting it to `False`. Otherwise you will end up spamming the logs and slowing the device down.

See Also

- [ESP32 Touch Pad Hub](#)
- [Binary Sensor Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Template Binary Sensor

The template binary sensor platform allows you to define any *lambda template* and construct a binary sensor out of it.

For example, below configuration would turn the state of an ultrasonic sensor into a binary sensor.

```
# Example configuration entry
binary_sensor:
  - platform: template
    name: "Garage Door Open"
    lambda: >-
      if (isnan(id(ultrasonic_sensor1).value)) {
        // isnan checks if the ultrasonic sensor echo
        // has timed out, resulting in a NaN (not a number) state
        // in that case, return {} to indicate that we don't know.
        return {};
      } else if (id(ultrasonic_sensor1).value > 30) {
        // Garage Door is open.
        return true;
      } else {
        // Garage Door is closed.
        return false;
      }
```

Possible return values of the lambda:

- `return true;` if the binary sensor should be ON.
- `return false;` if the binary sensor should be OFF.
- `return {};` if the last state should be repeated.

Configuration variables:

- **name** (**Required**, string): The name of the binary sensor.
- **lambda** (**Required**, *lambda*): Lambda to be evaluated repeatedly to get the current state of the binary sensor. Only state *changes* will be published to MQTT.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Binary Sensor* and *MQTT Component*.

See Also

- *Binary Sensor Component*
- *Template Sensor*
- *Automations And Templates*
- *API Reference*
- [Edit this page on GitHub](#)

Cover Component

The `cover` component is a generic representation of covers in esphomelib/yaml. A cover can (currently) either be *closed* or *open* and supports three types of commands: *open*, *close* and *stop*.

`cover.open` Action

This action opens the cover with the given ID when executed.

```
on_...:
  then:
    - cover.open:
      id: cover_1
```

`cover.close` Action

This action closes the cover with the given ID when executed.

```
on_...:
  then:
    - cover.close:
      id: cover_1
```

`cover.stop` Action

This action stops the cover with the given ID when executed.

```
on_...:
  then:
    - cover.stop:
      id: cover_1
```

lambda calls

From *lambdas*, you can call several methods on all covers to do some advanced stuff (see the full *API Reference* for more info).

- `publish_state()`: Manually cause the cover to publish a new state and store it internally. If it's different from the last internal state, it's additionally published to the frontend.

```
// Within lambda, make the cover report a specific state
id(my_cover).publish_state(cover::COVER_OPEN);
id(my_cover).publish_state(cover::COVER_CLOSED);
```

- `state`: Retrieve the current state of the cover.

```
if (id(my_cover).state == cover::COVER_OPEN) {
  // Cover is open
} else if (id(my_cover).state == cover::COVER_CLOSED) {
  // Cover is closed
```

(continues on next page)

(continued from previous page)

```
} else {
    // The cover hasn't reported any state yet.
}
```

- `open()`: Manually cause the cover to open from code. Similar to the `cover.open` action, but can be used in complex lambda expressions.

```
id(my_cover).open();
```

- `close()`: Manually cause the cover to close from code. Similar to the `cover.close` action, but can be used in complex lambda expressions.

```
id(my_cover).close();
```

- `stop()`: Manually cause the cover to stop from code. Similar to the `cover.stop` action, but can be used in complex lambda expressions.

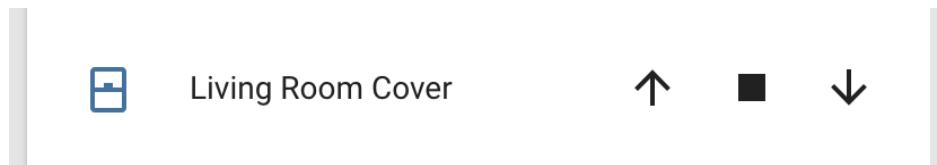
```
id(my_cover).stop();
```

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

Template Cover

The `template` cover platform allows you to create simple covers out of just a few actions and a value lambda. Once defined, it will automatically appear in Home Assistant as a cover and can be controlled through the frontend.



```
# Example configuration entry
cover:
- platform: template
  name: "Template Cover"
  lambda: >-
    if (id(top_end_stop).value) {
      return cover::COVER_OPEN;
    } else {
      return cover::COVER_CLOSED;
    }
  open_action:
  - switch.turn_on:
    id: open_cover_switch
  close_action:
  - switch.turn_on:
```

(continues on next page)

(continued from previous page)

```

    id: close_cover_switch
stop_action:
  - switch.turn_on:
    id: stop_cover_switch
optimistic: true
  
```

Possible return values for the optional lambda:

- `return cover::COVER_OPEN;` if the cover should be reported as OPEN.
- `return cover::COVER_CLOSED;` if the cover should be reported as CLOSED.
- `return {};` if the last state should be repeated.

Configuration variables:

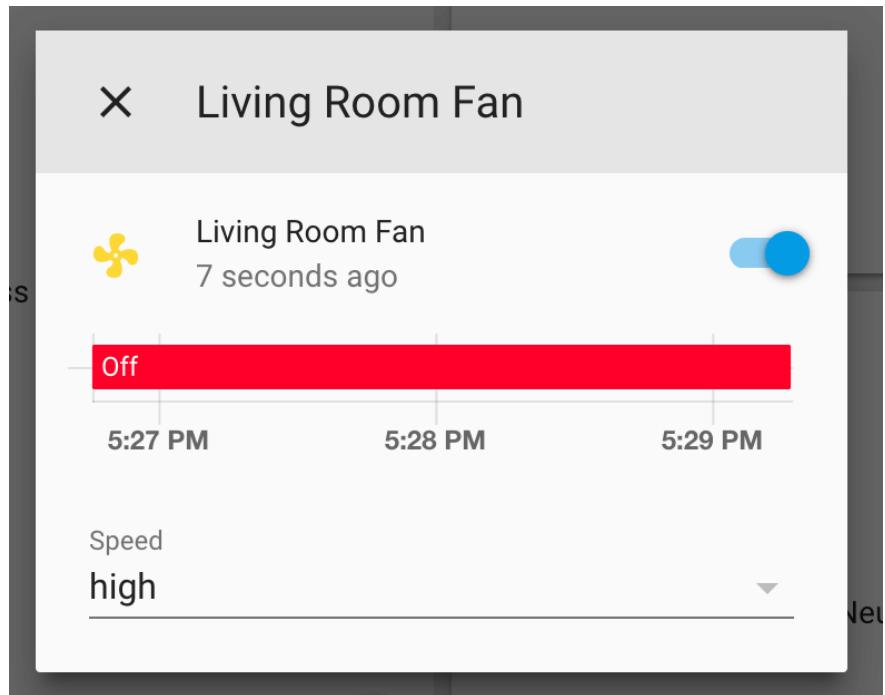
- **name** (**Required**, string): The name of the cover.
- **lambda** (*Optional*, `lambda`): Lambda to be evaluated repeatedly to get the current state of the cover. Only state *changes* will be published to MQTT.
- **optimistic** (*Optional*, boolean): Whether to operate in optimistic mode - when in this mode, any command sent to the template cover will immediately update the reported state and no lambda needs to be used. Defaults to `false`.
- **open_action** (*Optional*, `Action`): The action that should be performed when the remote (like Home Assistant's frontend) requests the cover to be opened.
- **close_action** (*Optional*, `Action`): The action that should be performed when the remote requests the cover to be closed.
- **stop_action** (*Optional*, `Action`): The action that should be performed when the remote requests the cover to stopped.
- **id** (*Optional*, `ID`): Manually specify the ID used for code generation.
- All other options from *Binary Sensor* and *MQTT Component*.

See Also

- [Cover Component](#)
- [Automations And Templates](#)
- [Simple Garage Door](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Fan Component

With the `fan` domain you can create components that appear as fans in the Home Assistant frontend. A fan can be switched ON or OFF, optionally has a speed setting (LOW, MEDIUM, HIGH) and can have an oscillate output.



Base Fan Configuration

```
fan:  
  - platform: ...  
    name: ...
```

Configuration variables:

- **name** (**Required**, string): The name of the fan.
- **oscillation_state_topic** (*Optional*, string): The topic to publish fan oscillation state changes to.
- **oscillation_command_topic** (*Optional*, string): The topic to receive oscillation commands on.
- **speed_state_topic** (*Optional*, string): The topic to publish fan speed state changes to.
- **speed_command_topic** (*Optional*, string): The topic to receive speed commands on.
- All other options from [MQTT Component](#).

fan.toggle Action

Toggles the ON/OFF state of the fan with the given ID when executed.

```
on_...:  
  then:  
    - fan.toggle:  
      id: fan_1
```

`fan.turn_off` Action

Turns the fan with the given ID off when executed.

```
on_...:
  then:
    - fan.turn_off:
      id: fan_1
```

`fan.turn_on` Action

Turns the fan with the given ID off when executed.

```
on_...:
  then:
    - fan.turn_on:
      id: fan_1
```

Configuration options:

- **id (Required, *ID*)**: The ID of the fan.
- **oscillating (Optional, boolean, *templatable*)**: Set the oscillation state of the fan. Defaults to not affecting oscillation.
- **speed (Optional, string, *templatable*)**: Set the speed setting of the fan. One of OFF, LOW, MEDIUM, HIGH. If you template this value, return `fan::FAN_SPEED_...`, for example `fan::FAN_SPEED_HIGH`.

Full Fan Index

- *API Reference*
- Edit this page on GitHub

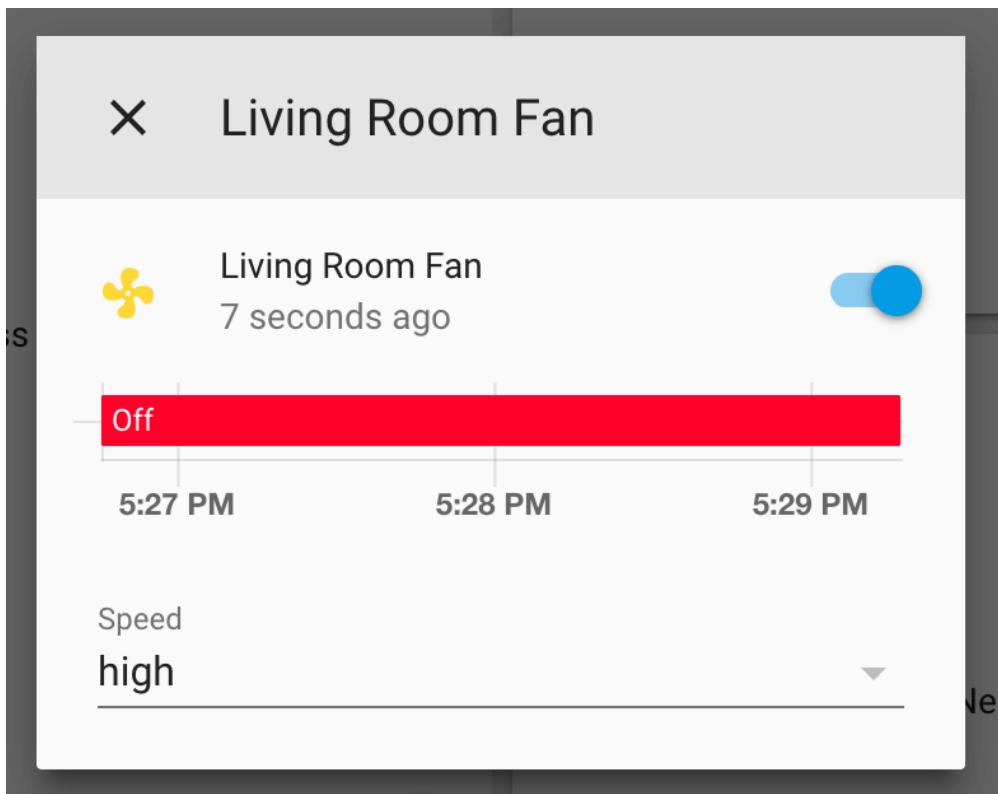
Binary Fan

The `binary` fan platform lets you represent any binary *Generic Output Switch* as a fan.

```
# Example configuration entry
fan:
  - platform: binary
    output: myoutput_1
    name: "Living Room Fan"
```

Configuration variables:

- **output (Required, *ID*)**: The id of the binary output component to use for this fan.
- **name (Required, string)**: The name for this fan.
- **oscillation_output (Optional, *ID*)**: The id of the `output` to use for the oscillation state of this fan. Default is empty.
- **id (Optional, *ID*)**: Manually specify the ID used for code generation.



- All other options from [MQTT Component](#) and [Fan Component](#).

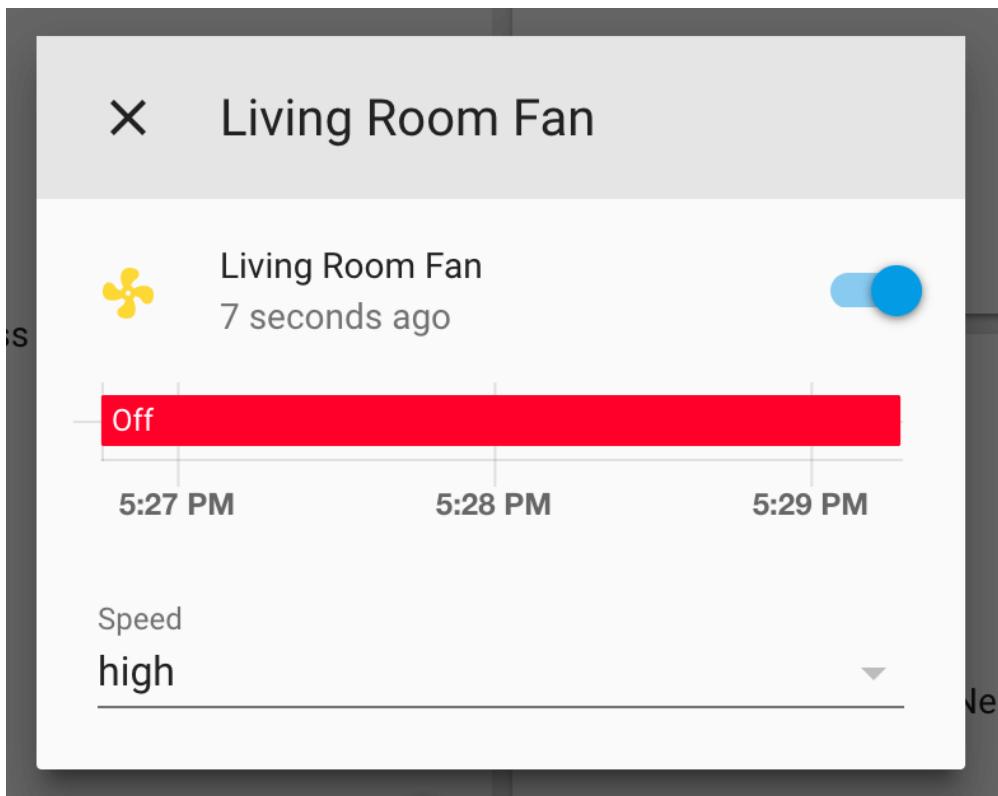
See Also

- [Output Component](#)
- [GPIO Output](#)
- [Fan Component](#)
- [API Reference](#)
- Edit this page on GitHub

Speed Fan

The `speed` fan platform lets you represent any float `Generic Output Switch` as a fan that supports speed settings.

```
# Example configuration entry
fan:
- platform: speed
  output: myoutput_1
  name: "Living Room Fan"
```



Configuration variables:

- **output** (**Required**, *ID*): The id of the *float output* to use for this fan.
- **name** (**Required**, string): The name for this fan.
- **oscillation_output** (*Optional*, *ID*): The id of the *output* to use for the oscillation state of this fan. Default is empty.
- **speed** (*Optional*): Set the float values for each speed setting:
 - **low** (*Required*, float): Set the value for the low speed setting. Must be in range 0 to 1. Defaults to 0.33.
 - **medium** (*Required*, float): Set the value for the medium speed setting. Must be in range 0 to 1. Defaults to 0.66.
 - **high** (*Required*, float): Set the value for the high speed setting. Must be in range 0 to 1. Defaults to 1.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *MQTT Component* and *Fan Component*.

See Also

- *Output Component*
- *Fan Component*
- *ESP32 LEDC Output*

- [ESP8266 Software PWM Output](#)
- [PCA9685 PWM Output](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Light Component

The `light` domain in `esphomeyaml` lets you create lights that will automatically be shown in Home Assistant's frontend and have many features such as RGB colors, transitions, flashing and effects.

`light.toggle` Action

This action toggles a light with the given ID when executed.

```
on_...:  
  then:  
    - light.toggle:  
      id: light_1
```

Configuration options:

- **id (Required, *ID*)**: The ID of the light.
- **transition_length (Optional, *Time, templatable*)**: The length of the transition if the light supports it.

`light.turn_on` Action

This action turns a light with the given ID on when executed.

```
on_...:  
  then:  
    - light.turn_on:  
      id: light_1  
      brightness: 100%  
      red: 100%  
      green: 100%  
      blue: 1.0  
  
      # Templated  
      - light.turn_on:  
        id: light_1  
        brightness: !lambda >-  
          // output value must be in range 0 - 1.0  
          return id(some_sensor).value / 100.0;
```

Configuration options:

- **id (Required, *ID*)**: The ID of the light.
- **transition_length (Optional, *Time, templatable*)**: The length of the transition if the light supports it.

- **brightness** (*Optional*, percentage, *templatable*): The brightness of the light. Must be in range 0% to 100% or 0.0 to 1.0. Defaults to not changing brightness.
- **red** (*Optional*, percentage, *templatable*): The red channel of the light. Must be in range 0% to 100% or 0.0 to 1.0. Defaults to not changing red.
- **green** (*Optional*, percentage, *templatable*): The green channel of the light. Must be in range 0% to 100% or 0.0 to 1.0. Defaults to not changing green channel.
- **blue** (*Optional*, percentage, *templatable*): The blue channel of the light. Must be in range 0% to 100% or 0.0 to 1.0. Defaults to not changing blue channel.
- **white** (*Optional*, percentage, *templatable*): The white channel value of RGBW lights. Must be in range 0% to 100% or 0.0 to 1.0. Defaults to not changing white value.
- **flash_length** (*Optional*, *Time*, *templatable*): If set, will flash the given color for this period of time and then go back to the previous state.
- **effect** (*Optional*, string, *templatable*): If set, will attempt to start an effect with the given name.

light.turn_off Action

This action turns a light with the given ID off when executed.

```
on_...:
  then:
    - light.turn_off:
      id: light_1
```

Configuration options:

- **id** (**Required**, *ID*): The ID of the light.
- **transition_length** (*Optional*, *Time*, *templatable*): The length of the transition if the light supports it.

This action turns a switch with the given ID off when executed.

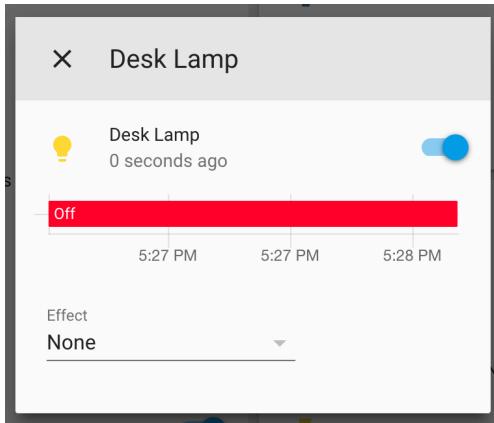
See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

Binary Light

The **binary** light platform creates a simple ON/OFF-only light from a *binary output component*.

```
# Example configuration entry
light:
  - platform: binary
    name: "Desk Lamp"
    output: output_component1
```



Configuration variables:

- **name** (**Required**, string): The name of the light.
- **output** (**Required**, *ID*): The id of the binary *Generic Output Switch* to use for this light.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from [MQTT Component](#).

See Also

- [Output Component](#)
- [Light Component](#)
- [GPIO Output](#)
- [Power Supply Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Monochromatic Light

The `monochromatic` light platform creates a simple brightness-only light from an *float output component*.

```
# Example configuration entry
light:
  - platform: monochromatic
    name: "Kitchen Lights"
    output: output_component1
```

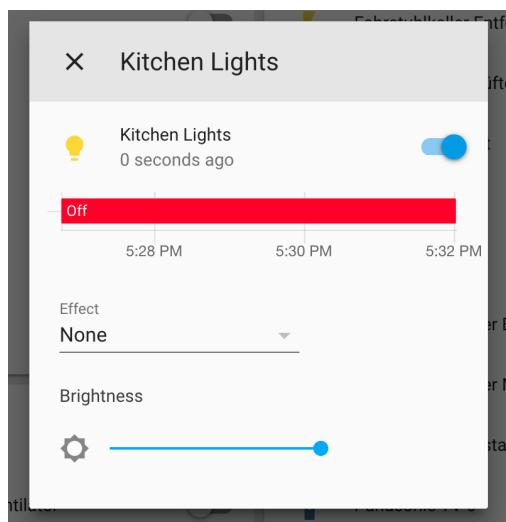
Configuration variables:

- **name** (**Required**, string): The name of the light.
- **output** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for this light.
- **gamma_correct** (*Optional*, float): The `gamma` correction factor for the light. Defaults to 2.8.



© Otto Winter, 2018

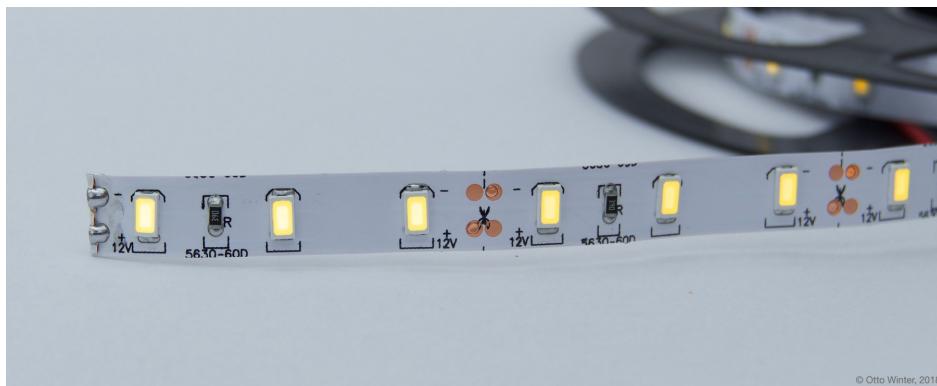
Fig. 1: Example of a brightness-only LED strip that can be used with this component.



- **default_transition_length** (*Optional, Time*): The length of the transition if no transition parameter is provided by Home Assistant. Defaults to `1s`.
- **id** (*Optional, ID*): Manually specify the ID used for code generation.
- All other options from [MQTT Component](#).

Note: The `monochromatic` light platform only works with `float outputs` that can output any light intensity percentage like the `ESP32 LEDC` or `ESP8266 PWM` components and does **not** work with output platforms like the `GPIO Output`.

See Also

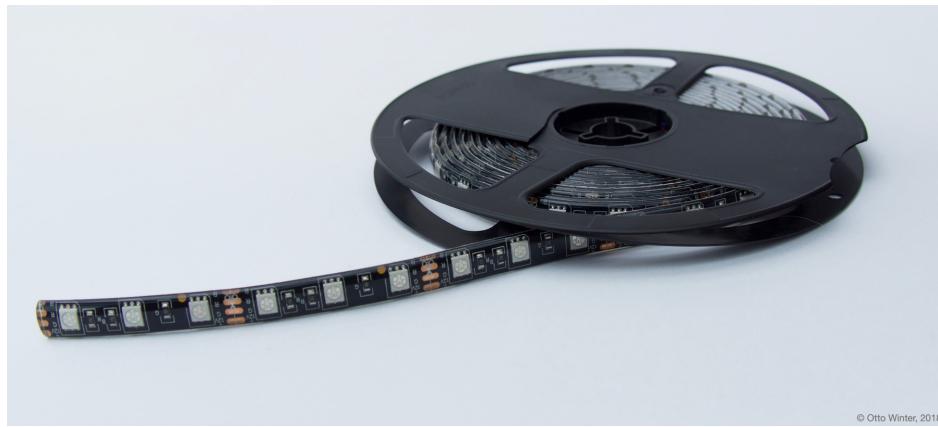


- [Output Component](#)
- [Light Component](#)
- [Binary Light](#)
- [Power Supply Component](#)
- [ESP32 LEDC Output](#)
- [ESP8266 Software PWM Output](#)
- [PCA9685 PWM Output](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

RGB Light

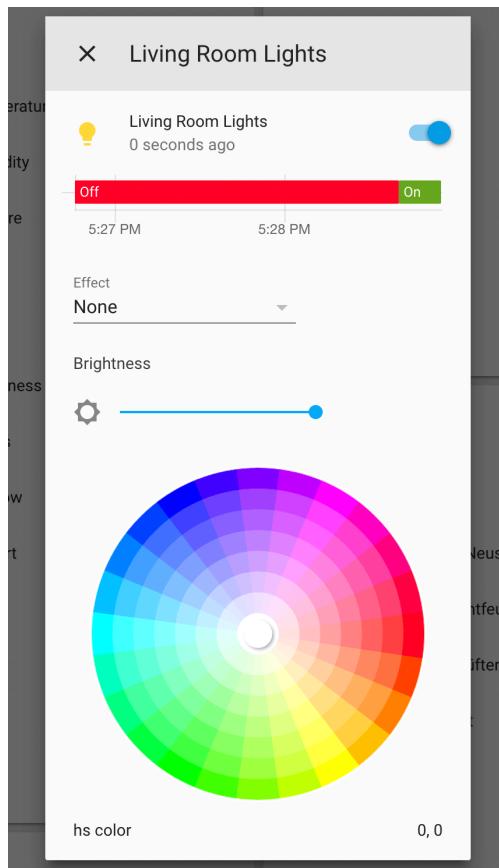
The `rgb` light platform creates an RGB light from 3 `float output components` (one for each color channel).

```
# Example configuration entry
light:
  - platform: rgb
    name: "Living Room Lights"
    red: output_component1
    green: output_component2
    blue: output_component3
```



© Otto Winter, 2018

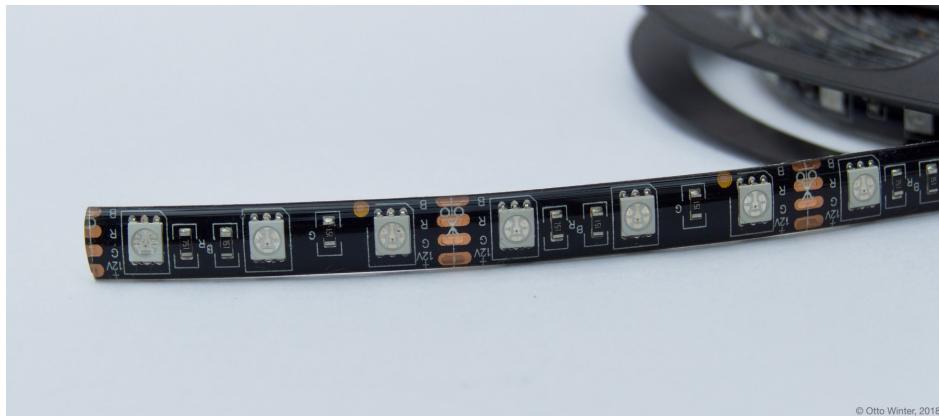
Fig. 2: Example of an RGB LED strip that can be used with this component.



Configuration variables:

- **name** (**Required**, string): The name of the light.
- **red** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the red channel.
- **green** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the green channel.
- **blue** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the blue channel.
- **gamma_correct** (*Optional*, float): The *gamma* correction factor for the light. Defaults to 2.8.
- **default_transition_length** (*Optional*, *Time*): The length of the transition if no transition parameter is provided by Home Assistant. Defaults to 1s.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *MQTT Component*.

Note: The RGB light platform only works with *float outputs* that can output any light intensity percentage like the *ESP32 LEDC* or *ESP8266 PWM* components and does **not** work with output platforms like the *GPIO Output*.

See Also© Otto Winter, 2018

- *Output Component*
- *Light Component*
- *RGBW Light*
- *Power Supply Component*
- *ESP32 LEDC Output*
- *ESP8266 Software PWM Output*
- *PCA9685 PWM Output*
- *API Reference*
- Edit this page on GitHub

RGBW Light

The `rgbw` light platform creates an RGBW light from 4 *float output components* (one for each channel).

```
# Example configuration entry
light:
  - platform: rgbw
    name: "Livingroom Lights"
    red: output_component1
    green: output_component2
    blue: output_component3
    white: output_component4
```

Configuration variables:

- **name** (**Required**, string): The name of the light.
- **red** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the red channel.
- **green** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the green channel.
- **blue** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the blue channel.
- **white** (**Required**, *ID*): The id of the float *Generic Output Switch* to use for the white channel.
- **gamma_correct** (*Optional*, float): The *gamma* correction factor for the light. Defaults to 2.8.
- **default_transition_length** (*Optional*, *Time*): The length of the transition if no transition parameter is provided by Home Assistant. Defaults to **1s**.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *MQTT Component*.

Note: The RGBW light platform only works with *float outputs* that can output any light intensity percentage like the *ESP32 LEDC* or *ESP8266 PWM* components and does **not** work with output platforms like the *GPIO Output*.

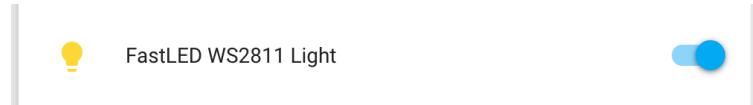
See Also

- *Output Component*
- *Light Component*
- *RGB Light*
- *Power Supply Component*
- *ESP32 LEDC Output*
- *ESP8266 Software PWM Output*
- *PCA9685 PWM Output*
- *API Reference*
- [Edit this page on GitHub](#)

FastLED Clockless Light

The `fastled_clockless` light platform allows you to create RGB lights in esphomelib for a *number of supported chipsets*.

Clockless FastLED lights differ from the *FastLED SPI Light* in that they only have a single data wire to connect, and not separate data and clock wires.



```
# Example configuration entry
light:
- platform: fastled_clockless
  chipset: WS2811
  pin: GPIO23
  num_leds: 60
  rgb_order: BRG
  name: "FastLED WS2811 Light"
```

Configuration variables:

- **name** (**Required**, string): The name of the light.
- **chipset** (**Required**, string): Set a chipset to use. See *Supported Chipsets* for options.
- **pin** (**Required**, *Pin*): The pin for the data line of the FastLED light.
- **num_leds** (**Required**, int): The number of LEDs attached.
- **rgb_order** (*Optional*, string): The order of the RGB channels. Use this if your light doesn't seem to map the RGB light channels correctly. For example if your light shows up green when you set a red color through the frontend. Valid values are RGB, RBG, GRB, GBR, BRG and BGR. Defaults to RGB.
- **max_refresh_rate** (*Optional*, *Time*): A time interval used to limit the number of commands a light can handle per second. For example 16ms will limit the light to a refresh rate of about 60Hz. Defaults to the default value for the used chipset.
- **gamma_correct** (*Optional*, float): The gamma correction factor for the light. Defaults to 2.8.
- **default_transition_length** (*Optional*, *Time*): The length of the transition if no transition parameter is provided by Home Assistant. Defaults to 1s.
- **power_supply** (*Optional*, *ID*): The *Power Supply Component* to connect to this light. When the light is turned on, the power supply will automatically be switched on too.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *MQTT Component*.

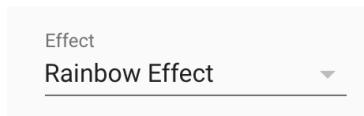
Supported Chipsets

- NEOPIXEL
- WS2811
- WS2811_400 (WS2811 with a clock rate of 400kHz)

- WS2812B
- WS2812
- WS2813
- WS2852
- APA104
- APA106
- GW6205
- GW6205_400 (GW6205 with a clock rate of 400kHz)
- LPD1886
- LPD1886_8BIT (LPD1886 with 8-bit color channel values)
- PL9823
- SK6812
- SK6822
- TM1803
- TM1804
- TM1809
- TM1829
- UCS1903B
- UCS1903
- UCS1904
- UCS2903

Light Effects

Currently, only a rainbow effect is supported. In the future, more light effects will be added and supported out-of-the box. Creating custom effects is, however, quite easy with esphomelib. See the [fastled example](#) in the esphomelib repository for a simple example.



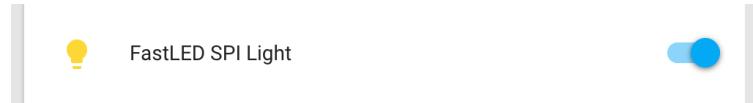
See Also

- [Light Component](#)
- [FastLED SPI Light](#)
- [Power Supply Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

FastLED SPI Light

The `fastled_spi` light platform allows you to create RGB lights in esphomelib for a *number of supported chipsets*.

SPI FastLED lights differ from the *FastLED Clockless Light* in that they require two pins to be connected, one for a data and one for a clock signal whereas the clockless lights only need a single pin.



```
# Example configuration entry
light:
  - platform: fastled_spi
    chipset: WS2801
    data_pin: GPIO23
    clock_pin: GPIO22
    num_leds: 60
    rgb_order: BRG
    name: "FastLED SPI Light"
```

Configuration variables:

- **name** (**Required**, string): The name of the light.
- **chipset** (**Required**, string): Set a chipset to use. See *Supported Chipsets* for options.
- **data_pin** (**Required**, *Pin*): The pin for the data line of the FastLED light.
- **clock_pin** (**Required**, *Pin*): The pin for the clock line of the FastLED light.
- **num_leds** (**Required**, int): The number of LEDs attached.
- **rgb_order** (*Optional*, string): The order of the RGB channels. Use this if your light doesn't seem to map the RGB light channels correctly. For example if your light shows up green when you set a red color through the frontend. Valid values are RGB, RBG, GRB, GBR, BRG and BGR. Defaults to RGB.
- **max_refresh_rate** (*Optional*, *Time*): A time interval used to limit the number of commands a light can handle per second. For example 16ms will limit the light to a refresh rate of about 60Hz. Defaults to the default value for the used chipset.
- **gamma_correct** (*Optional*, float): The *gamma correction factor* for the light. Defaults to 2.8.
- **default_transition_length** (*Optional*, *Time*): The length of the transition if no transition parameter is provided by Home Assistant. Defaults to 1s.
- **power_supply** (*Optional*, *ID*): The *Power Supply Component* to connect to this light. When the light is turned on, the power supply will automatically be switched on too.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *MQTT Component*.

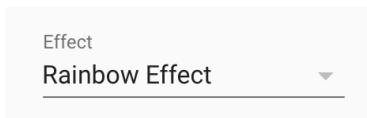
Supported Chipsets

- APA102

- DOTSTAR
- LPD8806
- P9813
- SK9822
- SM16716
- WS2801
- WS2803

Light Effects

Currently, only a rainbow effect is supported. In the future, more light effects will be added and supported out-of-the box. Creating custom effects is, however, quite easy with esphomelib. See the [fastled example](#) in the esphomelib repository for a simple example.



See Also

- [Light Component](#)
- [FastLED Clockless Light](#)
- [Power Supply Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Output Component

Each platform of the `output` domain exposes some output to esphomelib. These are grouped into two categories: `binary` outputs (that can only be ON/OFF) and `float` outputs (like PWM, can output any rational value between 0 and 1).

Base Output Configuration

Each output platform extends this configuration schema.

```
# Example configuration entry
output:
  - platform: ...
    id: myoutput_id
    power_supply: power_supply_id
    inverted: False
    max_power: 0.75
```

Configuration variables:

- **id** (**Required**, *ID*): The id to use for this output component.
- **power_supply** (*Optional*, *ID*): The *power supply* to connect to this output. When the output is enabled, the power supply will automatically be switched on too.
- **inverted** (*Optional*, boolean): If the output should be treated as inverted. Defaults to `False`.
- **max_power** (*Optional*, float): Only for float outputs. Sets the maximum output value of this output platform. Each value will be multiplied by this. Must be in range from 0 to 1. Defaults to 1.

Full Output Index

- *Generic Output Switch*
- *Power Supply Component*
- *Binary Light*
- *Monochromatic Light*
- *RGB Light*
- *Binary Fan*
- *Speed Fan*
- *API Reference*
- [Edit this page on GitHub](#)

ESP8266 Software PWM Output

The ESP8266 Software PWM platform allows you to use a software PWM on the pins GPIO0-GPIO16 on your ESP8266. As this is only a software PWM and not a hardware PWM (like the *ESP32 LEDC PWM*) and has a few limitations.

- There can be a noticeable amount of flickering with increased WiFi activity.
- The output range only goes up to about 80%.
- It's mostly fixed to a frequency of 1kHz, you can *increase this a bit manually* in code though.

If you need a stable PWM signal, it's definitely recommended to use the successor of the ESP8266, the ESP32, and its *ESP32 LEDC PWM* instead.

```
# Example configuration entry
output:
  - platform: esp8266_pwm
    pin: D1
    id: pwm-output
```

Configuration variables:

- **pin** (**Required**, *Pin Schema*): The pin to use PWM on.
- **id** (**Required**, *ID*): The id to use for this output component.
- All other options from *Output*.

See Also

- [Output Component](#)
- [ESP32 LEDC Output](#)
- [Monochromatic Light](#)
- [Speed Fan](#)
- [Power Supply Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

GPIO Output

The GPIO output component is quite simple: It exposes a single GPIO pin as an output component. Note that output components are **not** switches and will not show up in Home Assistant. See [GPIO Switch](#).

```
# Example configuration entry
output:
  - platform: gpio
    pin: D1
    id: gpio_d1
```

Configuration variables:

- **pin** (**Required**, [Pin Schema](#)): The pin to use PWM on.
- **id** (**Required**, [ID](#)): The id to use for this output component.
- All other options from [Output](#).

Warning: This is an **output component** and will not visible from the frontend. Output components are intermediary components that can be attached to for example lights. To have a GPIO pin in the Home Assistant frontend, please see the [GPIO Switch](#).

See Also

- [GPIO Switch](#)
- [Output Component](#)
- [ESP8266 Software PWM Output](#)
- [ESP32 LEDC Output](#)
- [Binary Light](#)
- [Binary Fan](#)
- [Power Supply Component](#)
- [API Reference](#)

- [Edit this page on GitHub](#)

ESP32 LEDC Output

The LEDC output component exposes a LEDC PWM channel of the ESP32 as an output component.

```
# Example configuration entry
output:
  - platform: ledc
    pin: 19
    id: gpio_19
```

Configuration variables:

- **pin** (**Required**, *Pin*): The pin to use LEDC on. Can only be GPIO0-GPIO33.
- **id** (**Required**, *ID*): The id to use for this output component.
- **frequency** (*Optional*, float): At which frequency to run the LEDC channel's timer. Two LEDC channels always share the same timer and therefore also the same frequency. Defaults to 1000Hz.
- **bit_depth** (*Optional*, int): The bit depth to use for the LEDC channel. Defaults to 12.
- **channel** (*Optional*, int): Manually set the LEDC channel to use. Two adjacent channels share the same timer. Defaults to an automatic selection.
- All other options from *Output*.

See Also

- [Output Component](#)
- [ESP8266 Software PWM Output](#)
- [Monochromatic Light](#)
- [Speed Fan](#)
- [Power Supply Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

PCA9685 PWM Output

The PCA9685 output component exposes a PCA9685 PWM channel of a global *PCA9685 hub* as a float output.

```
# Example configuration entry
pca9685:
  - frequency: 500

# Individual outputs
output:
```

(continues on next page)

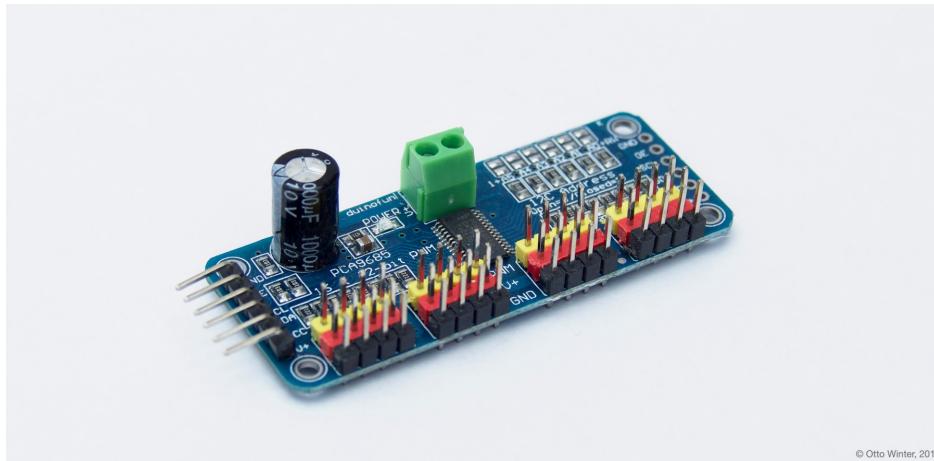


Fig. 3: PCA9685 16-Channel PWM Driver.

(continued from previous page)

```
- platform: pca9685
  id: 'pca9685_output1'
  channel: 0
```

Configuration variables:

- **id (Required, *ID*)**: The id to use for this output component.
- **channel (Required, int)**: Chose the channel of the PCA9685 of this output component. Must be in range from 0 to 15.
- **pca9685_id (Optional, *ID*)**: Manually specify the ID of the *PCA9685 hub*. Use this if you have multiple PCA9685s you want to use at the same time.
- All other options from *Output*.

See Also

- *PCA9685 PWM Component*
- *Output Component*
- *ESP8266 Software PWM Output*
- *ESP32 LEDC Output*
- *Monochromatic Light*
- *Speed Fan*
- *Power Supply Component*
- *API Reference*
- Edit this page on GitHub

Sensor Component

esphomelib has support for many different sensors. Each of them is a platform of the `sensor` domain and each sensor has several base configuration options.

Base Sensor Configuration

All sensors in esphomeyaml/esphomelib have a name and some other optional configuration options. By default, the sensor platform will chose appropriate values for all of these by default, but you can always override them if you want to.

```
# Example sensor configuration
name: Livingroom Temperature

# Optional variables:
unit_of_measurement: "°C"
icon: "mdi:water-percent"
accuracy_decimals: 1
expire_after: 30s
filters:
  - sliding_window_moving_average:
    window_size: 15
    send_every: 15
```

Configuration variables:

- **name** (**Required**, string): The name for the sensor.
- **unit_of_measurement** (*Optional*, string): Manually set the unit of measurement the sensor should advertise its values with. This does not actually do any maths (conversion between units).
- **icon** (*Optional*, icon): Manually set the icon to use for the sensor in the frontend.
- **accuracy_decimals** (*Optional*, int): Manually set the accuracy of decimals to use when reporting values.
- **expire_after** (*Optional*, *Time*): Manually set the time in which the sensor values should be marked as “expired”/“unknown”. Not providing any value means no expiry.
- **filters** (*Optional*): Specify filters to use for some basic transforming of values. Defaults to a basic sliding window moving average over the last few values. See [Sensor Filters](#) for more information.
- **on_value** (*Optional*, [Automation](#)): An automation to perform when a new value is published. See [on_value](#).
- **on_value_range** (*Optional*, [Automation](#)): An automation to perform when a published value transition from outside to a range to inside. See [on_value_range](#).
- **on_raw_value** (*Optional*, [Automation](#)): An automation to perform when a raw value is received that hasn’t passed through any filters. See [on_raw_value](#).
- All other options from [MQTT Component](#).

Sensor Filters

esphomeyaml/esphomelib allow you to do some basic preprocessing of sensor values before they’re sent to Home Assistant. This is for example useful if you want to apply some average over the last few values to

relief Home Assistant's state machine and keep the history graphs in the front-end a bit more clean. More sophisticated filters should be done with Home Assistant [filter sensor](#).

```
# Example filters:
filters:
- offset: 2.0
- multiply: 1.2
- filter_out: 42.0
- filter_nan:
- sliding_window_moving_average:
  window_size: 15
  send_every: 15
- exponential_moving_average:
  alpha: 0.1
  send_every: 15
- throttle: 1s
- heartbeat: 5s
- debounce: 0.1s
- delta: 5.0
- unique:
- or:
  - throttle: 1s
  - delta: 5.0
- lambda: return x * (9.0/5.0) + 32.0;
```

Above example configuration entry is probably a bit useless, but shows every filter there is currently:

- **offset**: Add an offset to every sensor value.
- **multiply**: Multiply each sensor value by this number.
- **filter_out**: Remove every sensor value that equals this number.
- **filter_nan**: Remove every value that is considered NAN (not a number) in C.
- **sliding_window_moving_average**: A [simple moving average](#) over the last few values.
 - **window_size**: The number of values over which to perform an average when pushing out a value.
 - **send_every**: How often a sensor value should be pushed out. For example, in above configuration the weighted average is only pushed out on every 15th received sensor value.
- **exponential_moving_average**: A simple exponential moving average over the last few values.
 - **alpha**: The forget factor/alpha value of the filter.
 - **send_every**: How often a sensor value should be pushed out.
- **throttle**: Throttle the incoming values. When this filter gets an incoming value, it checks if the last incoming value is at least **specified time period** old. If it is not older than the configured value, the value is not passed forward.
- **heartbeat**: Send the last value that this sensor in the specified time interval. So a value of **10s** will cause the filter to output values every 10s regardless of the input values.
- **debounce**: Only send values if the last incoming value is at least **specified time period** old. For example if two values come in at almost the same time, this filter will only output the last value and only after the specified time period has passed without any new incoming values.
- **delta**: This filter stores the last value passed through this filter and only passes incoming values through if the absolute difference is greater than the configured value. For example if a value of 1.0

first comes in, it's passed on. If the delta filter is configured with a value of 5, it will now not pass on an incoming value of 2.0, only values that are at least 6.0 big or -4.0.

- **unique**: This filter has no parameter and does one very simple thing: It only passes forward values if they are different from the last one that got through the pipeline.
- **or**: Pass forward a value with the first child filter that returns. Above example will only pass forward values that are *either* at least 1s old or are if the absolute difference is at least 5.0.
- **lambda**: Perform a simple mathematical operation over the sensor values. The input value is `x` and the result of the lambda is used as output. Each floating point operation should have `.0` attached as in above configuration. This will be copied over to the C++ code as a raw string.

Example: Converting Celsius to Fahrenheit

While I personally don't like the Fahrenheit temperature scale, I do understand that having temperature values appear in the fahrenheit unit is quite useful to some users. esphomelib uses the celsius temperature unit internally, and I'm not planning on making converting between the two simple (), but you can use this filter to convert celsius values to fahrenheit.

```
filters:  
  - lambda: return x * (9.0/5.0) + 32.0;  
unit_of_measurement: °F
```

Default Filter

By default, esphomelib takes an average over the last 15 values before publishing updates. This was done in order to automatically decrease sensor noise. Therefore if you have an `update_interval` of 15 seconds, you will only see the values every 3 and a half minutes or so. To disable the default filter and publish all raw values directly, put an empty `filters:` block in your configuration:

```
sensor:  
  - platform: ...  
    filters:
```

Sensor Automation

You can access the most recent state of the sensor in `lambda`s using `id(sensor_id).value` and the most recent raw state using `id(sensor_id).raw_value`.

on_value

This automation will be triggered when a new value that has passed through all filters is published. In `Lambdas` you can get the value from the trigger with `x`.

```
sensor:  
  - platform: dallas  
    # ...  
    on_value:  
      then:  
        - light.turn_on:
```

(continues on next page)

(continued from previous page)

```
id: light_1
red: !lambda "return x/255;"
```

Configuration variables: See [Automation](#).

`on_value_range`

With this automation you can observe if a sensor value passes from outside a defined range of values to inside a range. For example you can have an automation that triggers when a humidity crosses a threshold, and then turns on a dehumidifier. This trigger will only trigger when the new value is inside the range and the previous value was outside the range. It will also trigger on startup if the first value received is inside the range.

Define the range with `above` and `below`. If only one of them is defined, the interval is half-open. So for example `above: 5` with no `below` would mean the range from 5 to positive infinity.

```
sensor:
- platform: dallas
# ...
on_value_range:
  above: 5
  below: 10
  then:
    - switch.turn_on:
      id: relay_1
```

Configuration variables:

- **above** (*Optional*, float): The minimum for the trigger.
- **below** (*Optional*, float): The maximum for the trigger.
- See [Automation](#).

`on_raw_value`

This automation will be triggered when a new value that has passed through all filters is published. In [Lambdas](#) you can get the value from the trigger with `x`.

```
sensor:
- platform: dallas
# ...
on_value:
  then:
    - light.turn_on:
      id: light_1
      red: !lambda "return x/255;"
```

Configuration variables: See [Automation](#).

lambda calls

From [lambdas](#), you can call several methods on all sensors to do some advanced stuff (see the full [API Reference](#) for more info).

- `push_new_value()`: Manually cause the sensor to push out a value. It will then be processed by the sensor filters, and once done be published to MQTT.

```
// Within lambda, push a value of 42.0
id(my_binary_sensor).push_new_value(42.0);
```

- `value`: Retrieve the current value of the sensor that has passed through all sensor filters. Is NAN if no value has gotten through all filters yet.

```
// For example, create a custom log message when a value is received:
ESP_LOGI("main", "Value of my sensor: %f", id(my_sensor).value);
```

- `raw_value`: Retrieve the current value of the sensor that has not passed through any filters Is NAN if no value if no value has been pushed by the sensor itself yet.

```
// For example, create a custom log message when a value is received:
ESP_LOGI("main", "Raw Value of my sensor: %f", id(my_sensor).value);
```

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

Dallas Temperature Sensor

The `dallas` sensor allows you to use ds18b20 and similar sensors. First, you need to define a *dallas sensor component*. The dallas sensor component (or “hub”) is an internal model that defines which pins the ds18b20 sensors are connected to. This is because with these sensors you can actually connect multiple sensors to a single pin and use them all at once.

To initialize a sensor, first supply either `address` or `index` to identify the sensor.

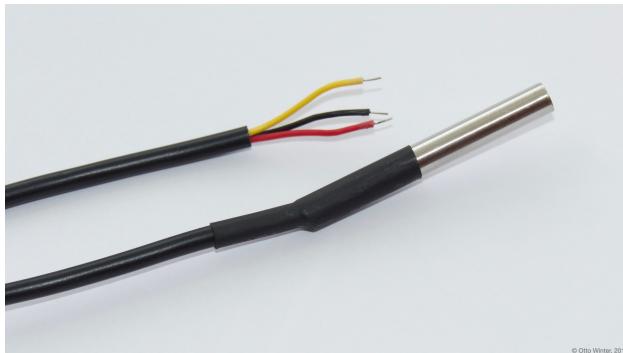


Fig. 4: Wired Version of the DS18b20 One-Wire Temperature Sensor.



Living Room Temperature

15.6 °C

```
# Example configuration entry
dallas:
  - pin: GPIO23

# Individual sensors
sensor:
  - platform: dallas
    address: 0x1C0000031EDD2A28
    name: "Living Room Temperature"
```

Configuration variables:

- **address** (**Required**, int): The address of the sensor. Use either this option or index.
- **index** (**Required**, int): The index of the sensor starting with 0. So the first sensor will for example have index 0. *It's recommended to use address instead.*
- **resolution** (*Optional*, int): An optional resolution from 8 to 12. Higher means more accurate. Defaults to the maximum for most dallas temperature sensors: 12.
- **dallas_id** (*Optional*, ID): The ID of the *dallas hub*. Use this if you have multiple dallas hubs.
- **id** (*Optional*, ID): Manually specify the ID used for code generation.
- All other options from *Sensor* and *MQTT Component*.

Getting Sensor IDs

It is highly recommended to use the `address` attribute for creating dallas sensors, because if you have multiple sensors on a bus and the automatic sensor discovery fails, all sensors indices will be shifted by one. In order to get the address, simply start the firmware on your device with a configured dallas hub and observe the log output (the `log level` must be set to at least `debug!`). Note that you don't need to define the individual sensors just yet, as the scanning will happen even with no sensors connected. For example with this configuration:

```
# Example configuration entry
dallas:
  - pin: GPIO23

# Note you don't have to add any sensors at this point
```

You will find something like this:

Now we can add the individual sensors to our configuration:

```
# Example configuration entry
dallas:
  - pin: GPIO23

sensor:
  - platform: dallas
    address: 0xA40000031F055028
    name: "Temperature #1"
  - platform: dallas
    address: 0xDD0000031EFB0428
```

(continues on next page)

```
[C][sensor :: dallas:setup:46]: Setting up DallasComponent ...
[C][sensor :: dallas:setup:47]:      Update Interval: 15000
[C][sensor :: dallas:setup:48]:      Want device count: 12
[D][sensor :: dallas:setup:55]: Found sensors:
[D][sensor :: dallas:setup:70]:      0xA40000031F055028
[D][sensor :: dallas:setup:70]:      0xDD0000031EFB0428
[D][sensor :: dallas:setup:70]:      0x790000031EE1DC28
[D][sensor :: dallas:setup:70]:      0xBA0000031F0E5228
[D][sensor :: dallas:setup:70]:      0x710000031F0E7E28
[D][sensor :: dallas:setup:70]:      0xFE0000031F1EAF28
[D][sensor :: dallas:setup:70]:      0x2C04173159F4FF28
[D][sensor :: dallas:setup:70]:      0x92041703081AFF28
[D][sensor :: dallas:setup:70]:      0x7D04173139EEFF28
[D][sensor :: dallas:setup:70]:      0x3204166398A5FF28
[D][sensor :: dallas:setup:70]:      0xD10417315BABFF28
[D][sensor :: dallas:setup:70]:      0x6C0517024A17FF28
```

(continued from previous page)

```
name: "Temperature #2"
- platform: dallas
# ...
```

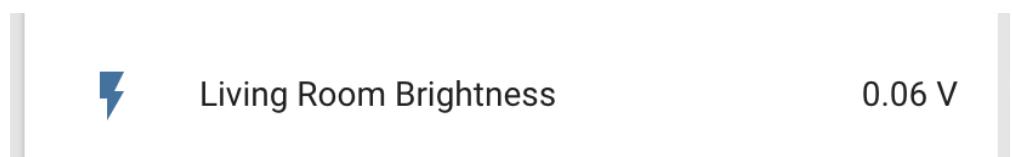
Next, individually warm up or cool down the sensors and observe the log again. You will see the outputted sensor values changing when they're being warmed. When you're finished mapping each address to a name, just change the `Temperature #1` to your assigned names and you should be ready.

See Also

- [Sensor Filters](#)
- [Dallas Temperature Component](#)
- [MAX6675 K-Type Thermocouple Temperature Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Analog To Digital Sensor

The Analog To Digital (adc) Sensor allows you to use the built-in ADC in your device to measure a voltage on certain pins. On the ESP8266 only pin A0 (GPIO17) supports this. On the ESP32 pins GPIO32 through GPIO39 can be used.



```
# Example configuration entry
sensor:
- platform: adc
  pin: A0
  name: "Living Room Brightness"
  update_interval: 15s
```

Configuration variables:

- **pin** (**Required**, *Pin*): The pin to measure the voltage on. Or on the ESP8266 alternatively also VCC, see [ESP8266 Measuring VCC](#).
- **name** (**Required**, string): The name of the voltage sensor.
- **attenuation** (*Optional*): Only on ESP32. Specify the ADC attenuation to use. See [ESP32 Attenuation](#).
- **update_interval** (*Optional*, *Time*): The interval to check the sensor. Defaults to **15s**. See [Default Filter](#).
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from [Sensor](#) and [MQTT Component](#).

Note: On the ESP8266, the voltage range is 0 to 1.0V - so to measure any higher voltage you need to scale the voltage down using, for example, a voltage divider circuit.

ESP32 Attenuation

On the ESP32, the voltage measured with the ADC caps out at 1.1V by default as the sensing range or the attenuation of the ADC is set to 0db by default.

To measure voltages higher than 1.1V, set **attenuation** to one of the following values:

- 0db for a full-scale voltage of 1.1V (default)
- 2.5db for a full-scale voltage of 1.5V
- 6db for a full-scale voltage of 2.2V
- 11db for a full-scale voltage of 3.9V

ESP8266 Measuring VCC

On the ESP8266 you can even measure the voltage the chip is getting. This can be useful in situations where you want to shut down the chip if the voltage is low when using a battery.

To measure the VCC voltage, set **pin:** to **VCC** and make sure nothing is connected to the **A0** pin.

```
sensor:
- platform: adc
  pin: VCC
  name: "VCC Voltage"
```

Next, you need to add a line at the top of your C++ project source code. Unfortunately, esphomelib can't do this automatically for you because of how the compiler is linking the esphomelib library. Open up the <NODE_NAME>/src/main.cpp file and insert the ADC_MODE line like this:

```
using namespace esphomelib;

// Enable measuring VCC
ADC_MODE(ADC_VCC);

void setup() {
    // ...
```

See Also

- [Sensor Filters](#)
- [ADS1115 Sensor](#)
- [MAX6675 K-Type Thermocouple Temperature Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

ADS1115 Sensor

Warning: This integration currently doesn't seem to work with certain chips and I'm waiting for my own ADS1115 to arrive to diagnose the issue. If you're experiencing issues too and want to help out, please set the [log level](#) to `VERY_VERBOSE` and send me some logs. Thanks!

The `ads1115` sensor allows you to use your ADS1115 sigma-delta ADC sensors ([datasheet](#), Adafruit) with esphomelib. First, setup a [ADS1115 Hub](#) for your ADS1115 sensor and then use this sensor platform to create individual sensors that will report the voltage to Home Assistant.

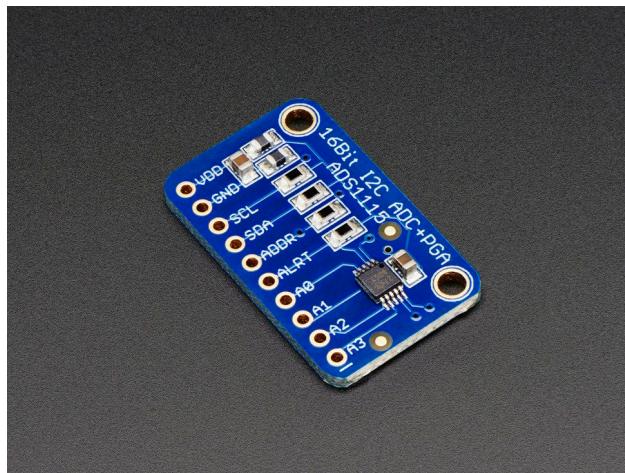


Fig. 5: ADS1115 16-Bit ADC. Image by Adafruit.



A screenshot of a Home Assistant card for a "Living Room Brightness" sensor. The card has a blue lightning bolt icon on the left. In the center, it says "Living Room Brightness". To the right, it shows a value of "0.06 V". There are vertical grey bars on either side of the card.

```
ads1115:
  - address: 0x48
sensor:
  - platform: ads1115
    multiplexer: 'A0_A1'
    gain: 1.024
    name: "Living Room Brightness"
```

Configuration variables:

- **multiplexer (Required)**: The multiplexer channel of this sensor. Effectively means between which pins you want to measure voltage.
- **gain (Required, float)**: The gain of this sensor.
- **name (Required, string)**: The name for this sensor.
- **ads1115_id (Optional, ID)**: Manually specify the ID of the *ADS1115 Hub* you want to use this sensor.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to 15s. See *Default Filter*.
- **id (Optional, ID)**: Manually specify the ID used for code generation.

Multiplexer And Gain

The ADS1115 has a multiplexer that can be configured to measure voltage between several pin configurations. These are:

- A0_A1 (between Pin 0 and Pin 1)
- A0_A3 (between Pin 0 and Pin 3)
- A1_A3 (between Pin 1 and Pin 3)
- A2_A3 (between Pin 2 and Pin 3)
- A0_GND (between Pin 0 and GND)
- A1_GND (between Pin 1 and GND)
- A2_GND (between Pin 2 and GND)
- A3_GND (between Pin 3 and GND)

Additionally, the ADS1115 has a Programmable Gain Amplifier (PGA) that can help you measure voltages in different ranges, these are:

- 6.144 (measures up to 6.144V)
- 4.096 (measures up to 4.096V)
- 2.048 (measures up to 2.048V)

- 1.024 (measures up to 1.024V)
- 0.512 (measures up to 0.512V)
- 0.256 (measures up to 0.256V)

See Also

- [Sensor Filters](#)
- [Analog To Digital Sensor](#)
- [MAX6675 K-Type Thermocouple Temperature Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

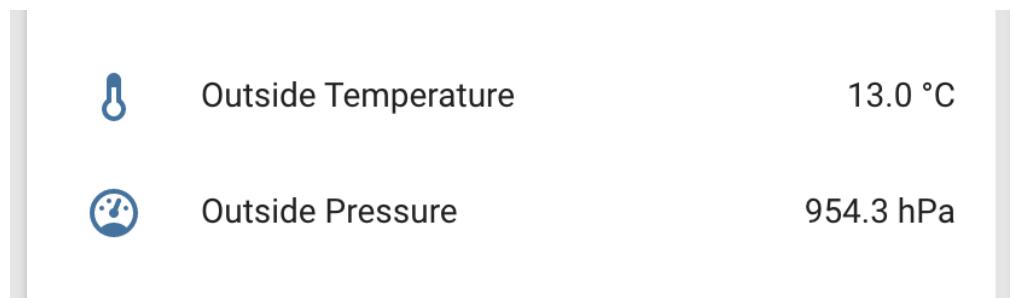
BMP085 Temperature+Pressure Sensor

The BMP085 sensor platform allows you to use your BMP085 ([datasheet](#), [adafruit](#)), BMP180 ([datasheet](#), [adafruit](#)) and BMP280 ([datasheet](#), [adafruit](#)) temperature and pressure sensors with esphomelib. The I^2C is required to be set up in your configuration for this sensor to work.



© Otto Winter, 2018

Fig. 6: BMP180 Temperature & Pressure Sensor..



```
# Example configuration entry
sensor:
  - platform: bmp085
    temperature:
      name: "Outside Temperature"
    pressure:
```

(continues on next page)

(continued from previous page)

```
name: "Outside Pressure"
update_interval: 15s
```

Configuration variables:

- **temperature (Required)**: The information for the temperature sensor.
 - **name (Required, string)**: The name for the temperature sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **pressure (Required)**: The information for the pressure sensor.
 - **name (Required, string)**: The name for the pressure sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **address (Optional, int)**: Manually specify the I²C address of the sensor. Defaults to 0x77.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to 15s. See *Default Filter*.

See Also

- *Sensor Filters*
- *BME280 Temperature+Pressure+Humidity Sensor*
- *BME680 Temperature+Pressure+Humidity+Gas Sensor*
- *API Reference*
- [Edit this page on GitHub](#)

DHT Temperature+Humidity Sensor

The DHT Temperature+Humidity sensor allows you to use your DHT11 ([datasheet](#), [adafruit](#)), DHT22 ([datasheet](#), [adafruit](#)), AM2302 ([datasheet](#), [adafruit](#)) and RHT03 ([datasheet](#), [sparkfun](#)) sensors with esphomelib.

The DHT22 and DHT11 require external pull up resistors on the data line. To do this, solder a resistor with *about* 4.7kΩ (anything in the range from 1kΩ to 10kΩ probably works fine, but if you’re having issues try the 4.7kΩ recommended by the manufacturer) between DATA and 3.3V.

```
# Example configuration entry
sensor:
  - platform: dht
    pin: D2
    temperature:
      name: "Living Room Temperature"
    humidity:
      name: "Living Room Humidity"
    update_interval: 15s
```

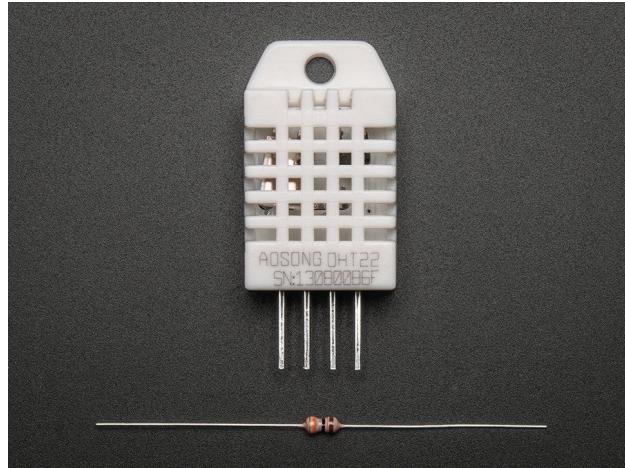
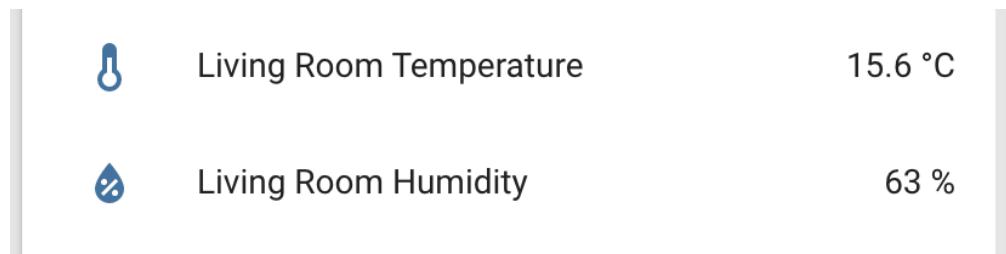


Fig. 7: DHT22 Temperature & Humidity Sensor. Image by Adafruit.



Configuration variables:

- **pin (Required, *Pin*):** The pin where the DHT bus is connected.
- **temperature (Required):** The information for the temperature sensor.
 - **name (Required, string):** The name for the temperature sensor.
 - **id (Optional, *ID*):** Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **humidity (Required):** The information for the humidity sensor
 - **name (Required, string):** The name for the humidity sensor.
 - **id (Optional, *ID*):** Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **model (Optional, int):** Manually specify the DHT model, can be one of AUTO_DETECT, DHT11, DHT22, AM2302, RHT03 and helps with some connection issues. Defaults to AUTO_DETECT.
- **update_interval (Optional, *Time*):** The interval to check the sensor. Defaults to 15s. See *Default Filter*.

Note: If you're seeing lots of invalid temperature/humidity warnings in the logs, try manually setting the DHT model with the `model`: configuration variable. Other problems could be wrong pull-up resistor values on the DATA pin or too long cables.

See Also

- [Sensor Filters](#)
- [DHT12 Temperature+Humidity Sensor](#)
- [HDC1080 Temperature+Humidity Sensor](#)
- [HTU21D Temperature+Humidity Sensor](#)
- [SHT3X-D Temperature+Humidity Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

HDC1080 Temperature+Humidity Sensor

The HDC1080 Temperature+Humidity sensor allows you to use your HDC1080 ([datasheet](#), [adafruit](#)) sensors with esphomelib. The *I²C Bus* is required to be set up in your configuration for this sensor to work.

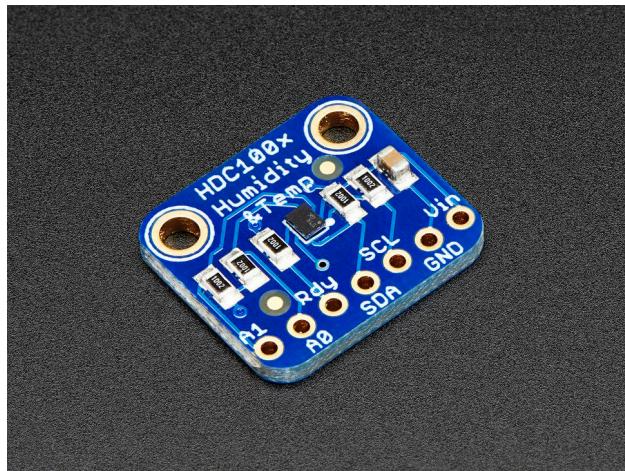


Fig. 8: HDC1080 Temperature & Humidity Sensor. Image by Adafruit.

	Living Room Temperature	15.6 °C
	Living Room Humidity	63 %

```
# Example configuration entry
sensor:
  - platform: hdc1080
    temperature:
      name: "Living Room Temperature"
    humidity:
      name: "Living Room Pressure"
    update_interval: 15s
```

Configuration variables:

- **temperature (Required)**: The information for the temperature sensor.
 - **name (Required, string)**: The name for the temperature sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **humidity (Required)**: The information for the humidity sensor
 - **name (Required, string)**: The name for the humidity sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to **15s**. See *Default Filter*.

Currently, the platform doesn't support activating the built-in heater, as it seems to only be rarely of use. If you need it, please open an issue.

See Also

- *Sensor Filters*
- *DHT Temperature+Humidity Sensor*
- *DHT12 Temperature+Humidity Sensor*
- *HTU21D Temperature+Humidity Sensor*
- *SHT3X-D Temperature+Humidity Sensor*
- *API Reference*
- Edit this page on GitHub

HTU21D Temperature+Humidity Sensor

The HTU21D Temperature+Humidity sensor allows you to use your HTU21D ([datasheet](#), [adafruit](#)) sensors with esphomelib. The *I²C Bus* is required to be set up in your configuration for this sensor to work.

```
# Example configuration entry
sensor:
  - platform: htu21d
    temperature:
      name: "Living Room Temperature"
    humidity:
      name: "Living Room Humidity"
    update_interval: 15s
```

Configuration variables:

- **temperature (Required)**: The information for the temperature sensor.
 - **name (Required, string)**: The name for the temperature sensor.

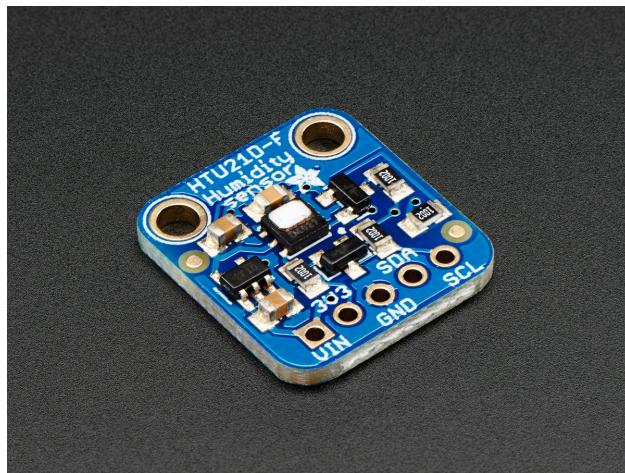
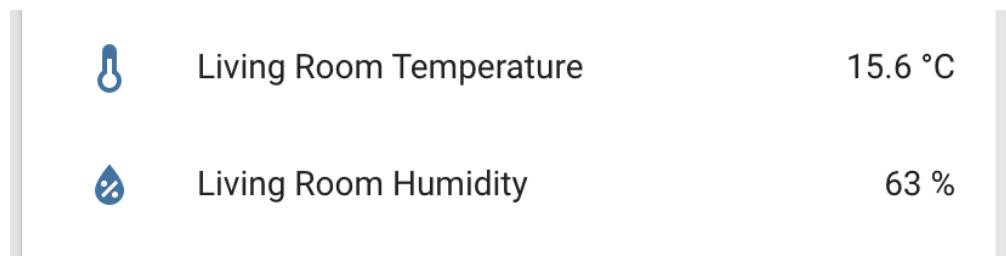


Fig. 9: HTU21D Temperature & Humidity Sensor. Image by Adafruit.



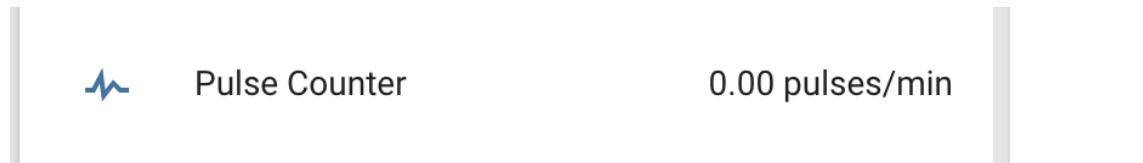
- **id** (*Optional, ID*): Set the ID of this sensor for use in lambdas.
- All other options from *Sensor* and *MQTT Component*.
- **humidity (Required)**: The information for the humidity sensor.
 - **name (Required, string)**: The name for the humidity sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to 15s. See *Default Filter*.

See Also

- *Sensor Filters*
- *DHT Temperature+Humidity Sensor*
- *DHT12 Temperature+Humidity Sensor*
- *HDC1080 Temperature+Humidity Sensor*
- *SHT3X-D Temperature+Humidity Sensor*
- *API Reference*
- Edit this page on GitHub

ESP32 Pulse Counter Sensor

The pulse counter sensor allows you to count the number of pulses on a specific pin using the [pulse counter peripheral](#) on the ESP32.



```
# Example configuration entry
sensor:
  - platform: pulse_counter
    pin: 12
    name: "Pulse Counter"
```

Configuration variables:

- **pin (Required, *Pin*)**: The pin to count pulses on.
- **name (Required, string)**: The name of the sensor.
- **pull_mode (Optional)**: The [pull mode](#) of the pin. One of PULLUP, PULLDOWN, PULLUP_PULLDOWN, FLOATING. Defaults to FLOATING.
- **count_mode (Optional)**: Configure how the counter should behave on a detected rising edge/falling edge.
 - **rising_edge (Optional)**: What to do when a rising edge is detected. One of DISABLE, INCREMENT and DECREMENT. Defaults to INCREMENT.
 - **falling_edge (Optional)**: What to do when a falling edge is detected. One of DISABLE, INCREMENT and DECREMENT. Defaults to DISABLE.
- **internal_filter (Optional, int)**: If a pulse shorter than this number of APB clock pulses (each 12.5 ns) is detected, it's discarded. See [esp-idf Filtering Pulses](#) for more information. Defaults to the max value 1023 or about 13 µs.
- **update_interval (Optional, *Time*)**: The interval to check the sensor. Defaults to 15s. See [Default Filter](#).
- **id (Optional, *ID*)**: Manually specify the ID used for code generation.
- All other options from [Sensor](#) and [MQTT Component](#).

Converting units

The sensor defaults to measuring its values using a unit of measurement of “pulses/min”. You can change this by using [Sensor Filters](#). For example, if you’re using the pulse counter with a photodiode to count the light pulses on a power meter, you can do the following:

```
# Example configuration entry
sensor:
  - platform: pulse_counter
```

(continues on next page)

(continued from previous page)

```
pin: 12
unit_of_measurement: 'kW'
name: 'Power Meter House'
filters:
  - multiply: 0.06
```

See Also

- *Sensor Filters*
- *Rotary Encoder Sensor*
- esp-idf Pulse Counter API.
- *API Reference*
- Edit this page on GitHub

Ultrasonic Distance Sensor

The ultrasonic distance sensor allows you to use simple ultrasonic sensors like the HC-SR04 ([datasheet](#), [sparkfun](#)) with esphomelib to measure distances. These sensors usually can't measure anything more than about two meters and may sometimes make some annoying clicking sounds.

This sensor platform expects a sensor that can be sent a **trigger pulse** on a specific pin and will send out a **echo pulse** once a measurement has been taken. Because sometimes (for example if no object is detected) the echo pulse is never returned, this sensor also has a timeout option which specifies how long to wait for values. During this timeout period the whole core will be blocked and therefore shouldn't be set too high.

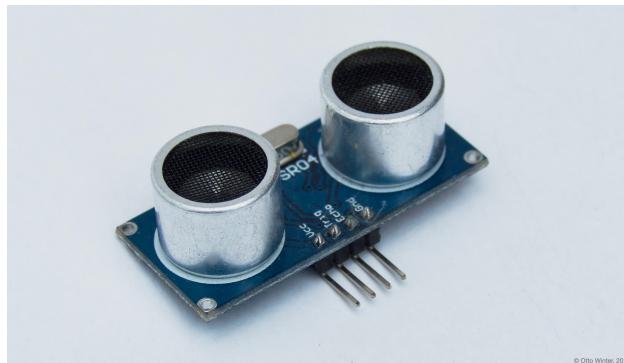


Fig. 10: HC-SR04 Ultrasonic Distance Sensor.



Ultrasonic Sensor

nan m

```
# Example configuration entry
sensor:
  - platform: ultrasonic
```

(continues on next page)

(continued from previous page)

```
trigger_pin: D1
echo_pin: D2
name: "Ultrasonic Sensor"
```

Configuration variables:

- **trigger_pin** (**Required**, *Pin Schema*): The output pin to periodically send the trigger pulse to.
- **echo_pin** (**Required**, *Pin Schema*): The input pin on which to wait for the echo.
- **name** (**Required**, string): The name of the sensor.
- **timeout_meter** (*Optional*, float): The number of meters for the timeout. Use either this or `timeout_time`. Defaults to 2 meters.
- **timeout_time** (*Optional*, int): The number of microseconds for the timeout. Use either this or `timeout_meter`. Defaults to 11662 μ s.
- **update_interval** (*Optional*, *Time*): The interval to check the sensor. Defaults to 15s. See *Default Filter*.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Sensor* and *MQTT Component*.

See Also

- *Sensor Filters*
- *Template Sensor*
- *API Reference*
- [Edit this page on GitHub](#)

MPU6050 Accelerometer/Gyroscope Sensor

The MPU6050 Accelerometer/Gyroscope sensor allows you to use your MPU6050 ([datasheet](#), [Sparkfun](#)) sensors with esphomelib. The [I²C Bus](#) is required to be set up in your configuration for this sensor to work.

This component only does some basic filtering and no calibration. Due to the complexity of this sensor and the amount of possible configuration options, you should probably create a custom component by copying and modifying the existing code if you want a specific new feature. Supporting all possible use-cases would be quite hard.

```
# Example configuration entry
sensor:
  - platform: mpu6050
    address: 0x68
    accel_x:
      name: "MPU6050 Accel X"
    accel_y:
      name: "MPU6050 Accel Y"
    accel_z:
      name: "MPU6050 Accel z"
```

(continues on next page)

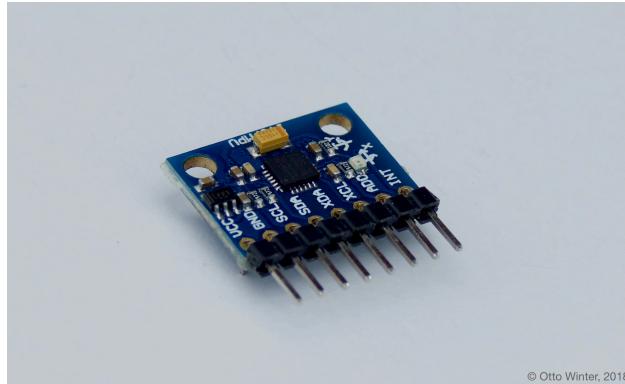


Fig. 11: MPU6050 Accelerometer/Gyroscope Sensor.

MPU6050

	MPU6050 Accel X	9.87 m/s ²
	MPU6050 Accel Y	0.41 m/s ²
	MPU6050 Accel Z	-0.40 m/s ²
	MPU6050 Gyro X	-2.62 °/s
	MPU6050 Gyro Y	0.43 °/s
	MPU6050 Gyro Z	-1.46 °/s
	MPU6050 Temperature	27.6 °C

(continued from previous page)

```
gyro_x:  
    name: "MPU6050 Gyro X"  
gyro_y:  
    name: "MPU6050 Gyro Y"  
gyro_z:  
    name: "MPU6050 Gyro z"  
temperature:  
    name: "MPU6050 Temperature"
```

Configuration variables:

- **address** (*Optional, int*): Manually specify the I²C address of the sensor. Defaults to 0x68.
- **accel_x** (*Optional*): Use the X-Axis of the Accelerometer. All options from [Sensor](#) and [MQTT Component](#).
- **accel_y** (*Optional*): Use the Y-Axis of the Accelerometer. All options from [Sensor](#) and [MQTT Component](#).
- **accel_z** (*Optional*): Use the Z-Axis of the Accelerometer. All options from [Sensor](#) and [MQTT Component](#).
- **gyro_x** (*Optional*): Use the X-Axis of the Gyroscope. All options from [Sensor](#) and [MQTT Component](#).
- **gyro_y** (*Optional*): Use the Y-Axis of the Gyroscope. All options from [Sensor](#) and [MQTT Component](#).
- **gyro_z** (*Optional*): Use the Z-Axis of the Gyroscope. All options from [Sensor](#) and [MQTT Component](#).
- **temperature** (*Optional*): Use the internal temperature of the sensor. All options from [Sensor](#) and [MQTT Component](#).
- **update_interval** (*Optional, Time*): The interval to check the sensor. Defaults to 15s. See [Default Filter](#).
- **id** (*Optional, ID*): Manually specify the ID used for code generation.

See Also

- [Sensor Filters](#)
- [Template Sensor](#)
- [Ultrasonic Distance Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

BH1750 Ambient Light Sensor

The bh1750 sensor platform allows you to use your BH1750 (datasheet, Aliexpress, mklec) ambient light sensor with esphomelib. The I²C bus is required to be set up in your configuration for this sensor to work.

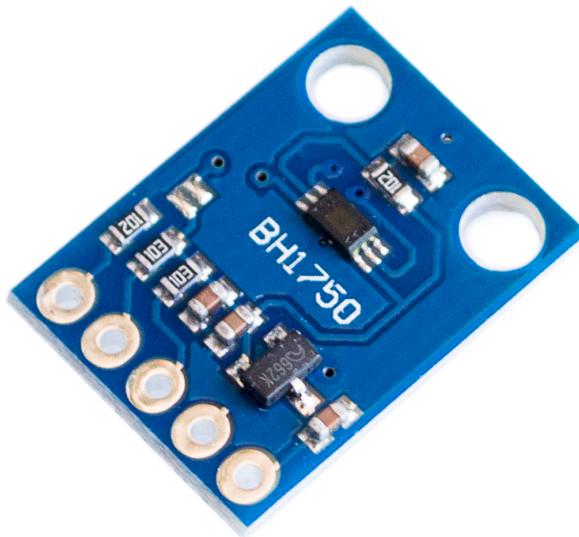


Fig. 12: BH1750 Ambient Light Sensor. Images from Aliexpress and mklec.

```
# Example configuration entry
sensor:
- platform: bh1750
  name: "Living Room Brightness"
  address: 0x23
  update_interval: 15s
```

Configuration variables:

- **name** (**Required**, string): The name for the sensor.
- **address** (*Optional*, int): Manually specify the I²C address of the sensor. Defaults to 0x23 (address if address pin is pulled low). If the address pin is pulled high, the address is 0x5C.
- **resolution** (*Optional*, string): The resolution of the sensor in lx. One of 4.0, 1.0, 0.5. Defaults to 0.5 (the maximum resolution).
- **update_interval** (*Optional*, Time): The interval to check the sensor. Defaults to 15s. See *Default Filter*.
- **id** (*Optional*, ID): Manually specify the ID used for code generation.
- All other options from *Sensor* and *MQTT Component*.

See Also

- *Sensor Filters*
- *TSL2561 Ambient Light Sensor*
- *API Reference*

- Edit this page on GitHub

BME280 Temperature+Pressure+Humidity Sensor

The `bme280` sensor platform allows you to use your BME280 ([datasheet](#), [Adafruit](#)) temperature, pressure and humidity sensors with esphomelib. The I^2C is required to be set up in your configuration for this sensor to work.

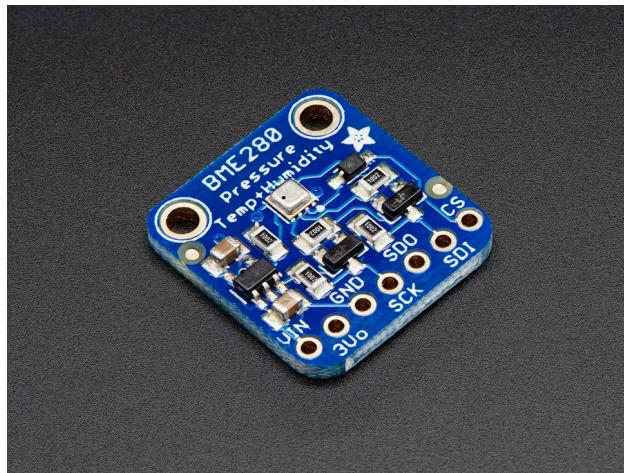


Fig. 13: BME280 Temperature, Pressure & Humidity Sensor. Image by [Adafruit](#).

```
# Example configuration entry
sensor:
  - platform: bme280
    temperature:
      name: "Outside Temperature"
      oversampling: 16x
    pressure:
      name: "Outside Pressure"
    humidity:
      name: "Outside Humidity"
    address: 0x77
    update_interval: 15s
```

Configuration variables:

- **temperature (Required)**: The information for the temperature. sensor
 - **name (Required, string)**: The name for the temperature sensor.
 - **oversampling (Optional)**: The oversampling parameter for the temperature sensor. See [Over-sampling Options](#).
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **pressure (Required)**: The information for the pressure sensor.
 - **name (Required, string)**: The name for the pressure sensor.

- **oversampling** (*Optional*): The oversampling parameter for the temperature sensor. See [Oversampling Options](#).
- **id** (*Optional, ID*): Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **humidity (Required)**: The information for the pressure sensor.
 - **name (Required, string)**: The name for the humidity sensor.
 - **oversampling** (*Optional*): The oversampling parameter for the temperature sensor. See [Oversampling Options](#).
 - **id** (*Optional, ID*): Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **address** (*Optional, int*): Manually specify the i²c address of the sensor. Defaults to 0x77. Another address can be 0x76.
- **iir_filter** (*Optional*): Set up an Infinite Impulse Response filter to increase accuracy. One of OFF, 2x, 4x, 16x. Defaults to OFF.
- **update_interval** (*Optional, Time*): The interval to check the sensor. Defaults to 15s. See [Default Filter](#).

Oversampling Options

By default, the BME280 sensor measures each value 16 times when requesting a new value. You can, however, configure this amount. Possible oversampling values:

- NONE
- 1x
- 2x
- 4x
- 8x
- 16x (default)

See Also

- [Sensor Filters](#)
- [BME680 Temperature+Pressure+Humidity+Gas Sensor](#)
- [BMP085 Temperature+Pressure Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

BME680 Temperature+Pressure+Humidity+Gas Sensor

Warning: This sensor is experimental has not been fully tested yet as I do not own all sensors. If you can verify it works (or if it doesn't), please notify me on [discord](#).

The `bme680` sensor platform allows you to use your BME680 ([datasheet](#), [Adafruit](#)) temperature, pressure and humidity sensors with esphomelib. The I^2C is required to be set up in your configuration for this sensor to work.

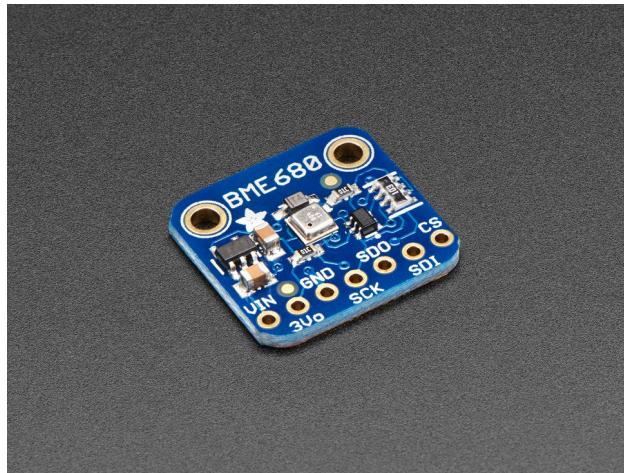


Fig. 14: BME680 Temperature, Pressure & Humidity Sensor. Image by [Adafruit](#).

```
# Example configuration entry
sensor:
  - platform: bme680
    temperature:
      name: "Outside Temperature"
      oversampling: 16x
    pressure:
      name: "Outside Pressure"
    humidity:
      name: "Outside Humidity"
    gas_resistance:
      name: "Outside Gas Sensor"
      address: 0x77
      update_interval: 15s
```

Configuration variables:

- **temperature (Required):** The information for the temperature sensor.
 - **name (Required, string):** The name for the temperature sensor.
 - **oversampling (Optional):** The oversampling parameter for the temperature sensor. See [Over-sampling Options](#).
 - **id (Optional, ID):** Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).

- **pressure (Required)**: The information for the pressure sensor.
 - **name (Required, string)**: The name for the pressure sensor.
 - **oversampling (Optional)**: The oversampling parameter for the temperature sensor. See [Oversampling Options](#).
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **humidity (Required)**: The information for the pressure sensor.
 - **name (Required, string)**: The name for the humidity sensor.
 - **oversampling (Optional)**: The oversampling parameter for the temperature sensor. See [Oversampling Options](#).
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **gas_resistance (Required)**: The information for the gas sensor.
 - **name (Required, string)**: The name for the gas resistance sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **address (Optional, int)**: Manually specify the i²c address of the sensor. Defaults to 0x77. Another address can be 0x76.
- **iir_filter (Optional)**: Set up an Infinite Impulse Response filter to increase accuracy. One of OFF, 2x, 4x, 16x. Defaults to OFF.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to 15s. See [Default Filter](#).

Oversampling Options

By default, the BME680 sensor measures each value 16 times when requesting a new value. You can, however, configure this amount. Possible oversampling values:

- NONE
- 1x
- 2x
- 4x
- 8x
- 16x (default)

See Also

- [Sensor Filters](#)
- [BME280 Temperature+Pressure+Humidity Sensor](#)
- [BMP085 Temperature+Pressure Sensor](#)

- [API Reference](#)
- [Edit this page on GitHub](#)

TSL2561 Ambient Light Sensor

Warning: This sensor is experimental has not been fully tested yet as I do not own all sensors. If you can verify it works (or if it doesn't), please notify me on [discord](#).

The `tsl2561` sensor platform allows you to use your BH1750 ([datasheet](#), [Adafruit](#)) ambient light sensor with esphomelib. The I^2C is required to be set up in your configuration for this sensor to work.

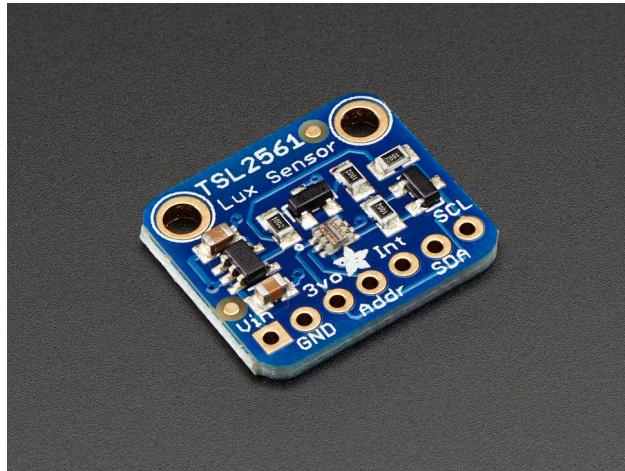


Fig. 15: TSL2561 Ambient Light Sensor. Image by [Adafruit](#).

```
# Example configuration entry
sensor:
  - platform: tsl2561
    name: "TSL2561 Ambient Light"
    address: 0x39
    update_interval: 15s
```

Configuration variables:

- **name** (**Required**, string): The name for the sensor.
- **address** (*Optional*, int): Manually specify the I^2C address of the sensor. Defaults to 0x39.
- **integration_time** (*Optional*, [Time](#)): The time the sensor will take for each measurement. Longer means more accurate values. One of 14ms, 101ms, 402ms. Defaults to 402ms.
- **gain** (*Optional*, string): The gain of the sensor. Higher values are better in low-light conditions. One of 1x and 16x. Defaults to 1x.
- **is_cs_package** (*Optional*, boolean): The “CS” package of this sensor has a slightly different formula for calculating the illuminance in lx. Set this to `true` if you’re working with a CS package. Defaults to `false`.

- **update_interval** (*Optional, Time*): The interval to check the sensor. Defaults to 15s. See [Default Filter](#).
- **id** (*Optional, ID*): Manually specify the ID used for code generation.
- All other options from [Sensor](#) and [MQTT Component](#).

See Also

- [Sensor Filters](#)
- [BH1750 Ambient Light Sensor](#)
- [Analog To Digital Sensor](#)
- [API Reference](#)
- Edit this page on [GitHub](#)

SHT3X-D Temperature+Humidity Sensor

Warning: This sensor is experimental has not been fully tested yet as I do not own all sensors. If you can verify it works (or if it doesn't), please notify me on [discord](#).

The `sht3xd` sensor platform Temperature+Humidity sensor allows you to use your Sensiron SHT31-D ([datasheet](#), [Adafruit](#)) sensors with esphomelib. The I^2C Bus is required to be set up in your configuration for this sensor to work.

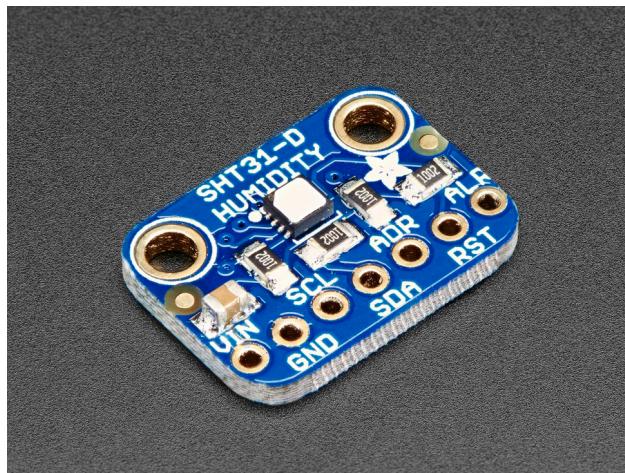
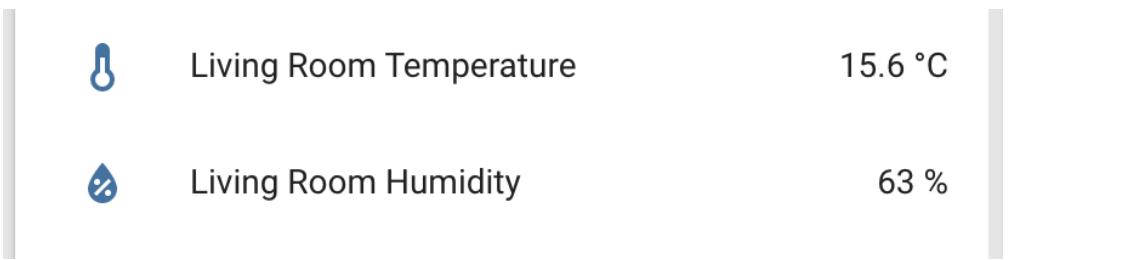


Fig. 16: SHT3X-D Temperature & Humidity Sensor. Image by Adafruit.

```
# Example configuration entry
sensor:
- platform: sht3xd
  temperature:
    name: "Living Room Temperature"
  humidity:
    name: "Living Room Humidity"
```

(continues on next page)



(continued from previous page)

```
accuracy: high
address: 0x44
update_interval: 15s
```

Configuration variables:

- **temperature (Required)**: The information for the temperature sensor.
 - **name (Required, string)**: The name for the temperature sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **humidity (Required)**: The information for the humidity sensor.
 - **name (Required, string)**: The name for the humidity sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from *Sensor* and *MQTT Component*.
- **address (Optional, int)**: Manually specify the i²c address of the sensor. Defaults to 0xff.
- **accuracy (Optional, string)**: The accuracy of the sensor. One of `low`, `medium` and `high`. Lower accuracies allow for faster update intervals. Defaults to `high`.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to `15s`. See *Default Filter*.

See Also

- *Sensor Filters*
- *DHT Temperature+Humidity Sensor*
- *DHT12 Temperature+Humidity Sensor*
- *HDC1080 Temperature+Humidity Sensor*
- *HTU21D Temperature+Humidity Sensor*
- *API Reference*
- Edit this page on GitHub

DHT12 Temperature+Humidity Sensor

The `dht12` Temperature+Humidity sensor allows you to use your DHT12 ([datasheet](#), [electrodragon](#)) i2c-based sensor with esphomelib.

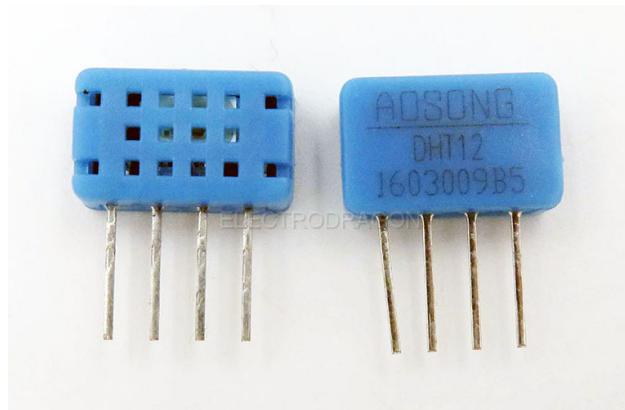
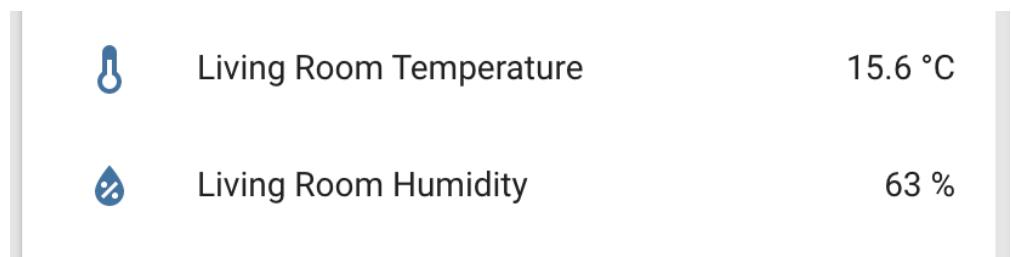


Fig. 17: DHT12 Temperature & Humidity Sensor. Image by [electrodragon](#).



```
# Example configuration entry
sensor:
- platform: dht12
  temperature:
    name: "Living Room Temperature"
  humidity:
    name: "Living Room Humidity"
  update_interval: 15s
```

Configuration variables:

- **temperature (Required)**: The information for the temperature sensor.
 - **name (Required, string)**: The name for the temperature sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).
- **humidity (Required)**: The information for the humidity sensor
 - **name (Required, string)**: The name for the humidity sensor.
 - **id (Optional, ID)**: Set the ID of this sensor for use in lambdas.
 - All other options from [Sensor](#) and [MQTT Component](#).

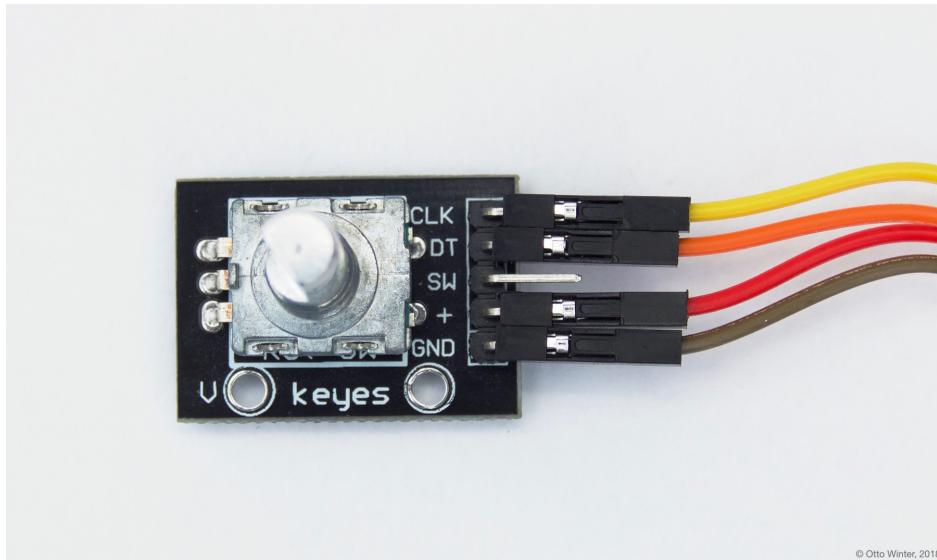
- **update_interval** (*Optional, Time*): The interval to check the sensor. Defaults to 15s. See [Default Filter](#).

See Also

- [Sensor Filters](#)
- [DHT Temperature+Humidity Sensor](#)
- [HDC1080 Temperature+Humidity Sensor](#)
- [HTU21D Temperature+Humidity Sensor](#)
- [SHT3X-D Temperature+Humidity Sensor](#)
- [API Reference](#)
- Edit this page on GitHub

Rotary Encoder Sensor

The `rotary_encoder` sensor platform allows you to use any continuous-rotation rotary encoders with esphomeyaml. These devices usually have two pins with which they encode the rotation. Every time the knob of the rotary encoder is turned, the signals of the two pins go HIGH and LOW in turn. See [this Arduino article](#) to gain a better understanding of these sensors.



© Otto Winter, 2018

Fig. 18: Example of a continuous rotary encoder. Pin + is connected to 3.3V, GND is connected to GND, and CLK & DT are A & B.



Rotary Encoder

-3 steps

To use rotary encoders in esphomeyaml, first identify the two pins encoding the step value. These are often called CLK and DT as in above image. Note if the values this sensor outputs go in the wrong direction, you can just swap these two pins.

```
# Example configuration entry
sensor:
  - platform: rotary_encoder
    name: "Rotary Encoder"
    pin_a: D1
    pin_b: D2
```

Configuration variables:

- **pin_a** (**Required**, *Pin Schema*): The first pin for determining the step value. Must not be a pin from an external I/O expander.
- **pin_b** (**Required**, *Pin Schema*): The second pin for determining the step value. Must not be a pin from an external I/O expander.
- **name** (**Required**, string): The name of the rotary encoder sensor.
- **pin_reset** (*Optional*, *Pin Schema*): An optional pin that resets the step value. This is useful with rotary encoders that have have a third pin. Defaults to no reset pin.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Sensor* and *MQTT Component*.

Debouncing Output

This sensor can output a lot of values in a short period of time when turning the knob. In order to not put too much stress on your network connection, you can leverage esphomelib's sensor filters. The following will only send out values if the last input value is at least 0.1s seconds old *or* if the new rotary encoder value has changed by 10 from the previous value.

```
# Example configuration entry
sensor:
  - platform: rotary_encoder
    name: "Rotary Encoder"
    pin_a: D1
    pin_b: D2
    filters:
      - or:
        - debounce: 0.1s
        - delta: 10
```

See Also

- *Sensor Filters*
- *ESP32 Pulse Counter Sensor*
- *Template Sensor*
- *API Reference*
- Edit this page on GitHub

Template Sensor

The `template` sensor platform allows you to create a sensor with templated values using *lambdas*.

```
# Example configuration entry
sensor:
  - platform: template
    name: "Template Sensor"
    lambda: >-
      if (id(some_binary_sensor).value) {
        return 42.0;
      } else {
        return 0.0;
      }
    update_interval: 15s
```

Possible return values for the lambda:

- `return <FLOATING_POINT_NUMBER>`; the new value for the sensor.
- `return NAN`; if the state should be considered invalid to indicate an error (advanced).
- `return {}`; if you don't want to publish a new state (advanced).

Configuration variables:

- **name** (**Required**, string): The name of the binary sensor.
- **lambda** (*Optional*, `lambda`): Lambda to be evaluated every update interval to get the new value of the sensor
- **update_interval** (*Optional*, `Time`): The interval to check the sensor. Defaults to `15s`. See *Default Filter*.
- **id** (*Optional*,`:ref:config-id`): Manually specify the ID used for code generation.
- All other options from *Binary Sensor* and *MQTT Component*.

See Also

- *Sensor Filters*
- *Automations And Templates*
- *API Reference*
- Edit this page on GitHub

MAX6675 K-Type Thermocouple Temperature Sensor

The `max6675` temperature sensor allows you to use your max6675 thermocouple temperature sensor ([datasheet](#), [sainsmart](#)) with esphomelib

Connect GND to GND, VCC to 3.3V and the other three MISO (or S0 for short), CS and CLOCK (or SCK) to free GPIO pins.

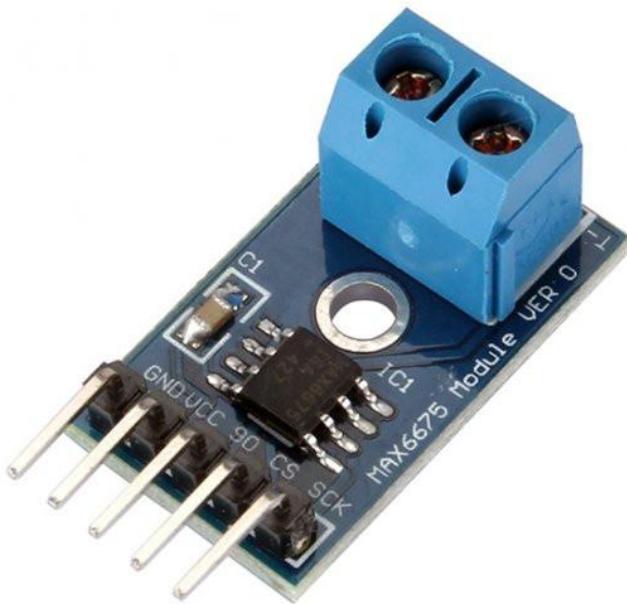


Fig. 19: MAX6675 K-Type Thermocouple Temperature Sensor. Image by [sainsmart](#).



Living Room Temperature

15.6 °C

```
# Example configuration entry
sensor:
- platform: max6675
  name: "Living Room Temperature"
  pin_cs: D0
  pin_clock: D1
  pin_miso: D2
  update_interval: 15s
```

Configuration variables:

- **name (Required)**: The name for the temperature sensor.
- **pin_cs (Required)**: The Chip Select pin of the SPI interface.
- **pin_clock (Required)**: The Clock pin of the SPI interface.
- **pin_miso (Required)**: The Master-In/Slave-Out pin of the SPI interface.
- **update_interval (Optional, Time)**: The interval to check the sensor. Defaults to 15s. See [Default Filter](#).
- **id (Optional, ID)**: Manually specify the ID used for codegeneration.
- All other options from [Sensor](#) and [MQTT Component](#).

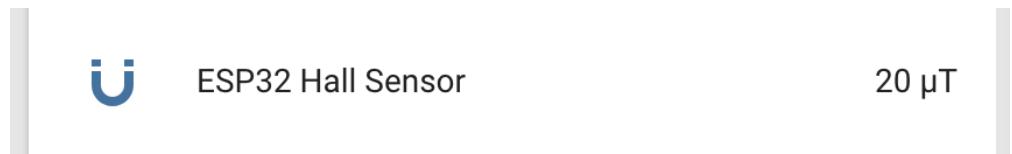
See Also

- [Sensor Filters](#)
- [Dallas Temperature Sensor](#)
- [DHT Temperature+Humidity Sensor](#)
- [Analog To Digital Sensor](#)
- [SHT3X-D Temperature+Humidity Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

ESP32 Hall Sensor

The `esp32_hall` sensor platform allows you to use the integrated hall effect sensor of the ESP32 chip to measure the magnitude and direction of magnetic field around the chip (with quite poor accuracy).

Please make sure that nothing is connected to pins GPIO36 and GPIO39 if this component is enabled, as those pins are used for the internal low-noise amplifier used by the hall sensor.



```
# Example configuration entry
sensor:
- platform: esp32_hall
  name: "ESP32 Hall Sensor"
  update_interval: 15s
```

Configuration variables:

- **name** (**Required**, string): The name of the hall effect sensor.
- **update_interval** (*Optional*, *Time*): The interval to check the sensor. Defaults to 15s. See *Default Filter*.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Sensor* and *MQTT Component*.

Warning: The values this sensor outputs were only calibrated with a few magnets and no real “truth” sensor. Therefore the values could very well be off by orders of magnitude. Besides, this sensor should only be used to detect sudden high changes in the magnetic field.

If you have a real magnetic field calibration setup and want to contribute your values to esphomelib, please feel free to do so .

See Also

- *Sensor Filters*
- *Analog To Digital Sensor*
- *Generic ESP32*
- *API Reference*
- [Edit this page on GitHub](#)

Custom Sensor Component

Warning: While I do try to keep the esphomeyaml configuration options as stable as possible and backport them, the esphomelib API is less stable. If something in the APIs needs to be changed in order for something else to work, I will do so.

So, you just set up esphomelib for your ESP32/ESP8266, but sadly esphomelib is missing a sensor integration you’d really like to have . It’s pretty much impossible to support every single sensor, as there are simply too many. That’s why esphomelib has a really simple API for you to create your own **custom sensors**

In this guide, we will go through creating a custom sensor component for the [BMP180](#) pressure sensor (we will only do the pressure part, temperature is more or less the same). During this guide, you will learn how to 1. define a custom sensor esphomelib can use 2. go over how to register the sensor so that it will be shown inside Home Assistant and 3. leverage an existing arduino library for the BMP180 with esphomelib.

Note: Since the creation of this guide, the BMP180 has been officially supported by the *BMP085 component*. The code still applies though.

This guide will require at least a bit of knowledge of C++, so be prepared for that. If you have any problems, I'm here to help :) <https://discord.gg/KhAMKrd>

Step 1: Custom Sensor Definition

At this point, you might have a main source file like this:

```
// ...
using namespace esphomelib;

void setup() {
    // ===== DO NOT EDIT ANYTHING BELOW THIS LINE =====
    // ===== AUTO GENERATED CODE BEGIN =====
    App.set_name("livingroom");
    App.init_log();
    // ...
    // ===== AUTO GENERATED CODE END =====
    // ===== YOU CAN EDIT AFTER THIS LINE =====
    App.setup();
}

void loop() {
    App.loop();
}
```

To create your own custom sensor, you just have define a C++ class that extends `Component` and `Sensor` like this:

```
using namespace esphomelib;

class CustomSensor : public Component, public sensor::Sensor {
public:
    void setup() override {
        // This will be called by App.setup()
    }
    void loop() override {
        // This will be called by App.loop()
    }
};

void setup() {
    // ...
```

You've just created your first esphomelib sensor . It doesn't do very much right now and is never registered, but it's a first step.

Let's now take a look at how a sensor works in esphomelib: A sensor is some hardware device (like a BMP180) that sends out new values like temperatures.

Like any Component in esphomelib, if it's registered in the Application, `setup()` will be called for you when `App.setup()` is run. `setup()` is also the place where you should do hardware initialization like setting `pinMode()`. Next, every time `App.loop()` is called, your component will also receive a `loop()` call. This is

the place where you should do stuff like querying a sensor for a new value like you might be used to do in an Arduino sketch.

Let's now also take a closer look at this line, which you might not be too used to when writing pure C code:

```
class CustomSensor : public Component, public sensor::Sensor {
```

What this line is essentially saying is that we're defining our own class that's called `CustomSensor` which is also a subclass of `Component` and `Sensor` (in the namespace `sensor::`). `Component` is there so that we can register it in our application and so that we will receive `setup()` and `loop()` calls. We're also inheriting from the `Sensor` class so that our custom sensor can be used by the MQTT sensor to automatically display it in the Home Assistant frontend.

As most sensors really just setup some pins and then check the sensor every x seconds, there's another abstraction that we'll use to simplify our code: `PollingSensorComponent`.

```
class CustomSensor : public sensor::PollingSensorComponent {
public:
    CustomSensor(uint32_t update_interval) : sensor::PollingSensorComponent(update_interval) {}

    void setup() override {
        // This will be called by App.setup()
    }
    void update() override {
        // This will be called every `update_interval` milliseconds.
    }
};
```

What `PollingSensorComponent` (and `PollingComponent`) does is essentially just replace the `loop()` method and will call `update()` instead every `update_interval` milliseconds. Because with most sensors, you really don't need to get the latest values with every single `loop()` call (which can be called many times per second). If we forward the `update_interval` in our *constructor* (line 3), `PollingSensorComponent` will call `update()` for us every `update_interval` milliseconds, so that we don't have to do time checking ourself. You don't really need to know about constructors for now, but I would definitely recommend it.

Let's also now make our sensor actually *output* something (42 for now):

```
// class CustomSensor ...
// ... previous code
void update() override {
    push_new_value(42.0); // 42°C
}

std::string unit_of_measurement() override { return "°C"; }
int8_t accuracy_decimals() override { return 2; } // 2 decimal places of accuracy.
};
```

Every time `update` is called we will now **push** a new value to the MQTT sensor component that's listening to our events. Additionally, we created a function that tells the sensor what unit of measurement the value is in, this is not strictly required and only used for a nice output in Home Assistant.

Step 2: Registering the custom sensor

Now we have our Custom Sensor set up, but unfortunately it doesn't do much right now. Actually ... it does nothing because it's never registered in the App, so esphomelib can't know about it. Let's change that.

In your global `setup()` method, after you've setup all other components, do the following:

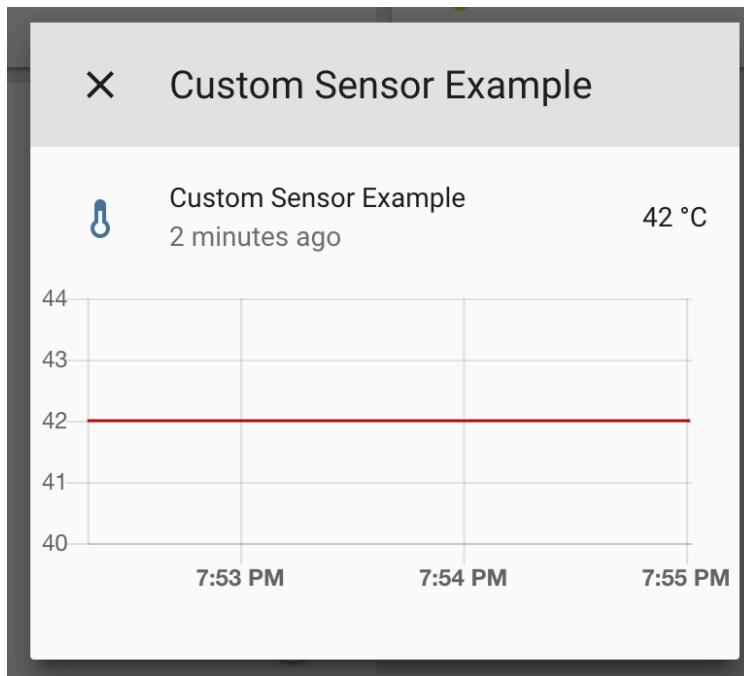
```
void setup() {
    // ...
    App.init_mqtt(...);

    // ... other stuff

    auto *custom_sensor = new CustomSensor(5000); // update every 5000ms or every 5 seconds.
    App.register_component(custom_sensor);
    App.make_mqtt_sensor_for(custom_sensor, "Custom Sensor Example");

    App.setup();
}
```

If you have Home Assistant MQTT discovery setup, it will even automatically show up in the frontend (with the entity id `sensor.custom_sensor_example`)



Let's go through the code for registering our custom sensor. First, we're creating a new `CustomSensor` instance with the update interval of 5000ms using the new C++ syntax (important!) and assigning it to a variable `custom_sensor` (using C++11 `auto` type specifier to make it simpler).

```
auto *custom_sensor = new CustomSensor(5000);
```

Next, we *register* the component in esphomelib's Application instance so that it can call the component's `setup()` and `loop()`.

```
App.register_component(custom_sensor);

// you could also write this, it's a bit shorter and works the same way.
// auto *custom_sensor = App.register_component(new CustomSensor(5000));
```

Lastly, we're setting up a `MQTTSensorComponent` for our sensor, this mqtt component will automatically set up a bunch of callbacks so that it can publish state changes to MQTT when you call `publish_new_value()`, create automatic MQTT discovery messages and setup a moving average over the sensor values (adjust these as you would with any other sensor).

Step 3: BMP180 support

Let's finally make this custom sensor useful by adding the BMP180 aspect into it! A great feature of esphomelib is that you can just use all existing arduino libraries, amazing right? Now for this example we'll use the [Adafruit BMP085 Library](#) library by Adafruit.

First we'll need to add the library to our platformio dependencies. To do so, put the following in the `common` section of your `platformio.ini`:

```
[common]
lib_deps = Adafruit BMP085 Library
build_flags =
upload_flags =
```

Next, include the library in your main sketch file:

```
#include "esphomelib/application.h"
#include <Adafruit_BMP085.h>

using namespace esphomelib;
```

Then update our sensor for BMP180 support:

```
class BMP180Sensor : public sensor::PollingSensorComponent {
public:
    Adafruit_BMP085 bmp;

    BMP180Sensor(uint32_t update_interval) : sensor::PollingSensorComponent(update_interval) {}

    void setup() override {
        bmp.begin();
    }

    void update() override {
        int pressure = bmp.readPressure(); // in Pa, or 1/100 hPa
        push_new_value(pressure / 100.0); // convert to hPa
    }

    std::string unit_of_measurement() override { return "hPa"; }
    int8_t accuracy_decimals() override { return 2; } // 2 decimal places of accuracy.
};
```

There's not too much going on there. First, we define the variable `bmp` of type `Adafruit_BMP085` inside our class as a class member. In `setup()` we initialize the library and in `update()` we read the pressure and send it out to MQTT.

You've now successfully created your first custom sensor component Happy coding!

See Also

- [Full source code](#)
- [Edit this page on GitHub](#)

Switch Component

The `switch` domain includes all platforms that should show up like a switch and can only be turned ON or OFF.

Base Switch Configuration

```
switch:
  - platform: ...
    name: "Switch Name"
    icon: "mdi:restart"
```

Configuration variables:

- **name** (**Required**, string): The name of the switch.
- **icon** (*Optional*, icon): Manually set the icon to use for the sensor in the frontend.
- **inverted** (*Optional*, boolean): Whether to invert the binary state, i.e. report ON states as OFF and vice versa. Defaults to `False`.
- All other options from [MQTT Component](#).

switch.toggle Action

This action toggles a switch with the given ID when executed.

```
on_...:
  then:
    - switch.toggle:
        id: relay_1
```

switch.turn_on Action

This action turns a switch with the given ID on when executed.

```
on_...:
  then:
    - switch.turn_on:
        id: relay_1
```

switch.turn_off Action

This action turns a switch with the given ID off when executed.

```
on_...:
  then:
    - switch.turn_off:
        id: relay_1
```

lambda calls

From `lambdas`, you can call several methods on all covers to do some advanced stuff (see the full [API Reference](#) for more info).

- `publish_state()`: Manually cause the switch to publish a new state and store it internally. If it's different from the last internal state, it's additionally published to the frontend.

```
// Within lambda, make the switch report a specific state
id(my_switch).publish_state(false);
id(my_switch).publish_state(true);
```

- `state`: Retrieve the current state of the switch.

```
// Within lambda, get the switch state and conditionally do something
if (id(my_switch).value) {
    // Switch is ON, do something here
} else {
    // Switch is OFF, do something else here
}
```

- `write_state()`: Manually cause the cover to go into an OFF/ON state from code. Similar to the `switch.turn_on` and `switch.turn_off` actions, but can be used in complex lambda expressions.

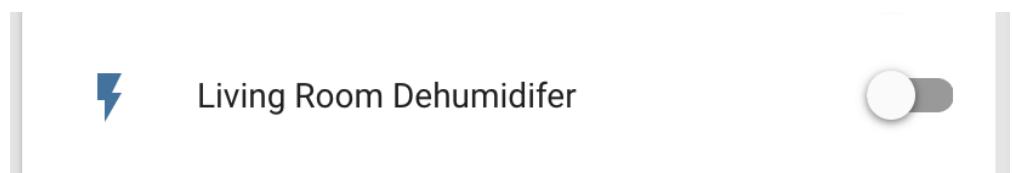
```
id(my_switch).write_state(false);
id(my_switch).write_state(true);
// Toggle the switch
id(my_switch).write_state(!id(my_switch).state);
```

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

GPIO Switch

The `gpio` switch platform allows you to use any pin on your node as a switch. You can for example hook up a relay to a GPIO pin and use it through this platform.



```
# Example configuration entry
switch:
  - platform: gpio
    pin: 25
    name: "Living Room Dehumidifier"
```

Configuration variables:

- **pin** (**Required**, *Pin Schema*): The GPIO pin to use for the switch.
- **name** (**Required**, string): The name for the switch.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from *Switch* and *MQTT Component*.

Note: If you want the pin to default to HIGH on startup, you can use the inverted property of the *Pin Schema*:

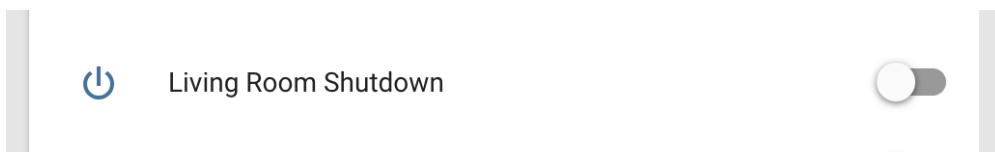
```
# Example configuration entry
switch:
  - platform: gpio
    pin:
      number: 25
      inverted: True
    name: "Living Room Dehumidifier"
```

See Also

- *Switch Component*
- *GPIO Output*
- *API Reference*
- Edit this page on GitHub

Shutdown Switch

The **shutdown** switch platform allows you to shutdown your node remotely through Home Assistant. It does this by putting the node into deep sleep mode with no wakeup source selected. After enabling, the only way to startup the ESP again is by pressing the reset button or restarting the power supply.



```
# Example configuration entry
switch:
  - platform: shutdown
    name: "Living Room Shutdown"
```

Configuration variables:

- **name** (**Required**, string): The name for the switch.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.

- All other options from [Switch](#) and [MQTT Component](#).

See Also

- [Restart Switch](#)
- [Template Switch](#)
- [API Reference](#)
- Edit this page on GitHub

Generic Output Switch

The output switch platform allows you to use any output component as a switch.



```
# Example configuration entry
output:
  - platform: gpio
    pin: 25
    id: 'generic_out'
switch:
  - platform: output
    name: "Generic Output"
    output: 'generic_out'
```

Configuration variables:

- **output (Required, ID)**: The ID of the output component to use.
- **name (Required, string)**: The name for the switch.
- **id (Optional, ID)**: Manually specify the ID used for code generation.
- All other options from [Switch](#) and [MQTT Component](#).

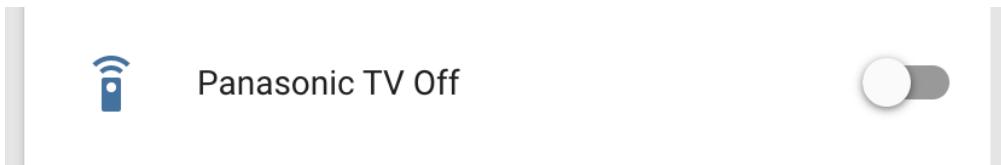
See Also

- [Output Component](#)
- [API Reference](#)
- Edit this page on GitHub

IR Transmitter Switch

The `ir_transmitter` switch platform allows you to create switches that send an IR code using the [IR Transmitter Component](#). Every time the switch is turned on, the IR signal with the provided IR code is sent out.

Theoretically this platform can also be used to create arbitrary output signals on any pin by using the `raw:` option.



```
# Example configuration entry
ir_transmitter:
  pin: 32

# Individual switches
switch:
  - platform: ir_transmitter
    name: "Panasonic TV Off"
    panasonic:
      address: 0x4004
      command: 0x100BCBD
    repeat: 25
```

Configuration variables:

- **name (Required, string):** The name for the switch.
- The IR code, see [Defining IR Codes](#). Only one of them can be specified per switch.
- **repeat (Optional, int):** How often the command should be repeated. Additionally, an `wait_time` option can be specified in the `repeat` section to set how long to wait in between repeats. Defaults to 1 (code is sent once). Example: `repeat\:\: \{times\:\: 10, wait_time\:\: 20us\}`
- **ir_transmitter_id (Optional, ID):** The id of the [IR Transmitter Component](#). Defaults to the first hub specified.
- **id (Optional, ID):** Manually specify the ID used for code generation.
- All other options from [Switch](#) and [MQTT Component](#).

Defining IR Codes

To get the IR codes, first use an Arduino or similar device with an IR Receiver and upload the Aruino-IRRemote IRRecvDump sketch. Then press the buttons on the remote and observe the serial monitor. It should include all the information needed to setup these IR codes.

```
nec:
  address: 0x4242
  command: 0x8484
```

(continues on next page)

(continued from previous page)

```

lg:
  data: 0x01234567890ABC
  nbits: 28
sony:
  data: 0xABCD
  nbits: 12
panasonic:
  address: 0x4004
  command: 0x1000BCD
raw:
  carrier_frequency: 35kHz
  data:
    - 1000
    - -1000

```

Configuration variables:

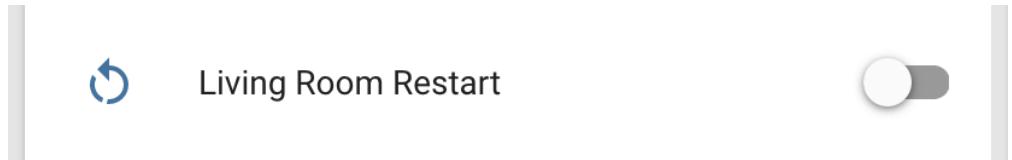
- **nec**: Send a NEC IR code.
 - **address**: The address of the device.
 - **command**: The command to send.
- **lg**: Send an LG IR code.
 - **data**: The data bytes to send.
 - **nbits**: The number of bits to send, defaults to 28.
- **sony**: Send an Sony IR code.
 - **data**: The data bytes to send.
 - **nbits**: The number of bits to send, defaults to 12.
- **panasonic**: Send an Panasonic IR code.
 - **address**: The address of the device.
 - **command**: The command to send.
- **raw**: Send an arbitrary signal.
 - **carrier_frequency**: The frequency to use for the carrier. A lot of IR sensors only respond to a very specific frequency.
 - **data**: List containing integers describing the signal to send. Each value is a time in μ s declaring how long the carrier should be switched on or off. Positive values mean ON, negative values mean OFF.

See Also

- [Switch Component](#)
- [IR Transmitter Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Restart Switch

The `restart` switch platform allows you to restart your node remotely through Home Assistant.



```
# Example configuration entry
switch:
  - platform: restart
    name: "Living Room Restart"
```

Configuration variables:

- **name** (**Required**, string): The name for the switch.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.
- All other options from [Switch](#) and [MQTT Component](#).

See Also

- [Shutdown Switch](#)
- [Template Switch](#)
- [API Reference](#)
- Edit this page on GitHub

Template Switch

The `template` switch platform allows you to create simple switches out of just actions and an optional value lambda. Once defined, it will automatically appear in Home Assistant as a switch and can be controlled through the frontend.

```
# Example configuration entry
switch:
  - platform: template
    name: "Template Switch"
    lambda: >-
      if (id(some_binary_sensor).value) {
        return true;
      } else {
        return false;
      }
    turn_on_action:
      - switch.turn_on:
          id: switch2
    turn_off_action:
```

(continues on next page)

(continued from previous page)

```
- switch.turn_on:
  id: switch1
optimistic: true
```

Possible return values for the optional lambda:

- `return true;` if the switch should be reported as ON.
- `return false;` if the switch should be reported as OFF.
- `return {};` if the last state should be repeated.

Configuration variables:

- **`name`** (**Required**, string): The name of the switch.
- **`lambda`** (*Optional*, `lambda`): Lambda to be evaluated repeatedly to get the current state of the switch. Only state *changes* will be published to MQTT.
- **`optimistic`** (*Optional*, boolean): Whether to operate in optimistic mode - when in this mode, any command sent to the template cover will immediately update the reported state and no lambda needs to be used. Defaults to `false`.
- **`turn_on_action`** (*Optional*, `Action`): The action that should be performed when the remote (like Home Assistant's frontend) requests the switch to be turned on.
- **`turn_off_action`** (*Optional*, `Action`): The action that should be performed when the remote (like Home Assistant's frontend) requests the switch to be turned off.
- **`id`** (*Optional*, `ID`): Manually specify the ID used for code generation.
- All other options from [Binary Sensor](#) and [MQTT Component](#).

Note: esphomelib will automatically try to restore the last state from flash on boot, it will then also call the turn on/off actions for you.

See Also

- [Automations And Templates](#)
- [Switch Component](#)
- [Binary Sensor Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

ADS1115 Hub

The `ads1115` domain creates a global hub so that you can later create individual sensors using the [ADS1115 Sensor Platform](#). To use this hub, first setup the [I²C Bus](#) and connect the sensor to the pins specified there.

```
ads1115:
- address: 0x48
```

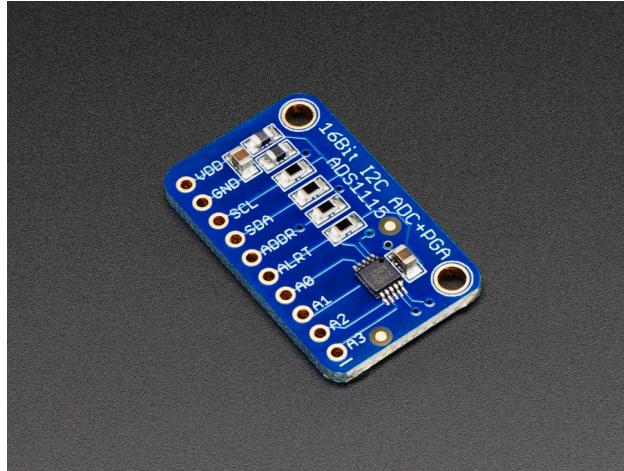


Fig. 20: ADS1115 16-Bit ADC. Image by Adafruit.

Configuration variables:

- **address** (**Required**, int): The i²c address of the sensor. See [I²C Addresses](#) for more information.
- **id** (*Optional*, [ID](#)): Manually specify the ID for this ADS1115 Hub. Use this if you want to use multiple ADS1115 hubs at once.

I²C Addresses

In order to allow multiple sensors to be connected to the same i²c bus, the creators of this sensor hardware have included some options to change the i²c address.

- If the address pin is pulled to GND, the address is 0x48 (Default).
- If the address pin is pulled to VCC, the address is 0x49.
- If the address pin is tied to SDA, the address is 0x4a.
- If the address pin is tied to SCL, the address is 0x4B.

See Also

- [ADS1115 Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Dallas Temperature Component

The `dallas` component allows you to use your DS18b20 (datasheet) and similar One-Wire temperature sensors.

To use your dallas sensor, first define a dallas “hub” with a pin and id, which you will later use to create the sensors. The 1-Wire bus the sensors are connected to should have an external pullup resistor of about 4.7KΩ.

For this, connect a resistor of *about* $4.7\text{K}\Omega$ (values around that like 1Ω will, if you don't have massively long wires, work fine in most cases) between 3.3V and the data pin.

```
# Example configuration entry
dallas:
  - pin: 23

# Individual sensors
sensor:
  - platform: dallas
    address: 0x1c0000031edd2a28
    name: "Livingroom Temperature"
```

Configuration variables:

- **pin** (*Required*, number): The pin the sensor bus is connected to.
- **update_interval** (*Optional*, *Time*): The interval that the sensors should be checked. Defaults to 15 seconds. See *Default Filter*.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.

See Also

- *Dallas Temperature Sensor*
- *API Reference*
- [Edit this page on GitHub](#)

Debug Component

The debug component can be used to debug problems with esphomelib. At startup, it prints a bunch of useful information like reset reason, free heap size, esphomelib version and so on.

```
[17:15:46][D][debug:setup:33]: esphomelib version 1.2.1-dev
[17:15:46][D][debug:setup:35]: Free Heap Size: 78340 bytes
[17:15:46][D][debug:setup:52]: Flash Chip: Size=4096kB Speed=40MHz Mode=DIO
[17:15:46][D][debug:setup:82]: Chip: Model=UNKNOWN, Features=WIFI_BGN,BLE,BT, Cores=2, Revision=1
[17:15:46][D][debug:setup:84]: ESP-IDF Version: v3.1-dev-239-g1c3dd23f-dirty
[17:15:46][D][debug:setup:87]: EFuse MAC: 0x3FFD46E03FFE915C
[17:15:46][D][debug:setup:108]: Reset Reason: Software Reset CPU
[17:15:46][D][debug:setup:126]: Wakeup Reason: Unknown
```

Fig. 21: Example debug component output.

```
# Example configuration entry
debug:

# Logger must be at least debug (default)
logger:
  level: debug
```

There are no configuration variables for this component.

See Also

- [Logger Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Deep Sleep Component

The `deep_sleep` component can be used to automatically enter a deep sleep mode on the ESP8266/ESP32 after a certain amount of time. This is especially useful with nodes that operate on batteries and therefore need to conserve as much energy as possible.

To use `deep_sleep` first specify how long the node should be active, i.e. how long it should check sensor values and report them, using the `run_duration` and `run_cycles` options. If you use both in your configuration, any time either one of them is finished, the node will go into deep sleep mode.

Next, tell the node how it should wakeup. On the ESP8266, you can only put the node into deep sleep for a duration using `sleep_duration`, note that on the ESP8266 GPIO16 must be connected to the RST pin so that it will wake up again. On the ESP32, you additionally have the option to wake up on any RTC pin (0, 2, 4, 12, 13, 14, 15, 25, 26, 27, 32, 39).

While in deep sleep mode, the node will not do any work and not respond to any network traffic, even Over The Air updates.

```
# Example configuration entry
deep_sleep:
    run_duration: 10s
    sleep_duration: 10min
```

Configuration variables:

- **`run_duration`** (*Optional, Time*): The time duration the node should be active, i.e. run code.
- **`run_cycles`** (*Optional, int*): The number of `loop()` cycles to go through before entering deep sleep mode.
- **`sleep_duration`** (*Optional, Time*): The time duration to stay in deep sleep mode.
- **`wakeup_pin`** (*Optional, Pin Schema*): Only on ESP32. A pin to wake up to once in deep sleep mode. Use the inverted property to wake up to LOW signals.
- **`id`** (*Optional, ID*): Manually specify the ID used for code generation.

See Also

- [Shutdown Switch](#)
- [Automations And Templates](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

ESP32 Bluetooth Low Energy Hub

The `esp32_ble` component creates a global hub so that you can track bluetooth low energy devices using your ESP32 node.

Currently this component only works with few supported bluetooth devices (most of them being BLE “beacons”) and currently only is capable of creating binary sensors indicating whether a specific BLE MAC Address can be found or not.

In the future, this integration will be expanded to support reading RSSI values and hopefully support lots more devices like tracking smartphones and reading temperature values from BLE sensors.

Note: Be warned: This integration is currently not very stable and sometimes causes sporadic restarts of the node. Additionally, using this integration will increase the required flash memory size by up to 500kB.

See [Setting up devices](#) for information on how you can find out the MAC address of a device and track it using esphomelib.

```
# Example configuration entry
esp32_ble:
  scan_interval: 300s

binary_sensor:
  - platform: esp32_ble
    mac_address: AC:37:43:77:5F:4C
    name: "ESP32 BLE Tracker Google Home Mini"
```

Configuration variables:

- **scan_interval** (*Optional, Time*): The length of each scan. If a device is not found within this time window, it will be marked as not present. Defaults to 300s.
- **id** (*Optional, ID*): Manually specify the ID for this ESP32 BLE Hub.

See Also

- [ESP32 Bluetooth Low Energy Device](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

ESP32 Touch Pad Hub

The `esp32_touch` component creates a global hub for detecting touches on the eight touch pads of the ESP32 as *binary sensors*.

```
# Example configuration entry
esp32_touch:
  setup_mode: False

binary_sensor:
```

(continues on next page)

(continued from previous page)

```
- platform: esp32_touch
  name: "ESP32 Touch Pad GPIO27"
  pin: GPIO27
  threshold: 1000
```

Configuration variables:

- **setup_mode** (*Optional*, boolean): Whether debug messages with the touch pad value should be displayed in the logs. Useful for finding out suitable thresholds for the binary sensors, but spam the logs. See [setting up touch pads](#) for more information. Defaults to false.
- **id** (*Optional*, *ID*): Manually specify the ID for code generation.

Advanced options (the defaults are usually quite good, but if you're having accuracy issues, use these):

- **iir_filter** (*Optional*, *Time*): Optionally set up an [Infinite Impulse Response](#) filter should be applied to all touch pads. This can increase the accuracy of the touch pads a lot, but higher values decrease the response time. A good value to start with is 10ms. Default is no IIR filter.
- **sleep_duration** (*Optional*, *Time*): Set a time period denoting the amount of time the touch peripheral should sleep between measurements. This can decrease power usage but make the sensor slower. Default is about 27 milliseconds.
- **measurement_duration** (*Optional*, *Time*): Set the conversion time for all touch pads. A longer conversion time means that more charge/discharge cycles of the touch pad can be performed, therefore increasing accuracy. Default is about 8ms, the maximum amount.
- **low_voltage_reference** (*Optional*): The low voltage reference to use for the charge cycles. See the [esp-idf docs](#) for a nice explanation of this. One of 0.5V, 0.6V, 0.7V, 0.8V. Default is 0.5V.
- **high_voltage_reference** (*Optional*): The high voltage reference to use for the charge cycles. See the [esp-idf docs](#) for a nice explanation of this. One of 2.4V, 2.5V, 2.6V, 2.7V. Default is 2.7V.
- **voltage_attenuation** (*Optional*): The voltage attenuation to use for the charge cycles. See the [esp-idf docs](#) for a nice explanation of this. One of 1.5V, 1V, 0.5V, 0V. Default is 0V.

See Also

- [ESP32 Touch Pad Binary Sensor](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

esphomeyaml Core Configuration

Here you specify some core information that esphomeyaml needs to create firmwares. Most importantly, this is the section of the configuration where you specify the **name** of the node, the **platform** and **board** you're using.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP32
  board: nodemcu-32s
```

Configuration variables:

- **name** (**Required**, string): This is the name of the node. It should always be unique to the node and no other node in your system can use the same name. It can also only contain upper/lowercase characters, digits and underscores.
- **platform** (**Required**, string): The platform your board is on, either ESP32 or ESP8266. See [Using the latest Arduino framework version](#).
- **board** (**Required**, string): The board esphomeyaml should specify for platformio. For the ESP32, choose the appropriate one from [this list](#) and use [this list](#) for ESP8266-based boards.
- **library_uri** (*Optional*, string): You can manually specify the [version of esphomelib](#) to use here. Accepts all parameters of [platformio lib install](#). Use <https://github.com/OttoWinter/esphomelib.git> for the latest (unstable) build. Defaults to the latest stable version.
- **simplify** (*Optional*, boolean): Whether to simplify the auto-generated code, i.e. whether to remove unused variables, use `auto` types and so on. Defaults to `true`.
- **use_build_flags** (*Optional*, boolean): If esphomeyaml should manually set build flags that specifically set what should be included in the binary. Most of this is already done automatically by the linker but this option can help with shrinking the firmware size while slowing down compilation. Defaults to `true`.

Using the latest Arduino framework version

The default version of the arduino framework distributed through platformio is usually quite old and the latest staging versions of the framework can in some cases increase stability a lot.

To use the latest version of the arduino framework with esphomeyaml, specify an URL pointing to the staging version using the `platform:` parameter.

For the ESP32, this URL is <https://github.com/platformio/platform-espressif32.git#feature/stage>. And for the ESP8266, the URL is <https://github.com/platformio/platform-espressif8266.git#feature/stage>.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: 'https://github.com/platformio/platform-espressif32.git#feature/stage'
  board: nodemcu-32s
```

See Also

- [Edit this page on GitHub](#)

I²C Bus

This component sets up the I²C bus for your ESP32 or ESP8266. In order for those components to work correctly, you need to define the I²C bus in your configuration.

```
# Example configuration entry
i2c:
  sda: 21
  scl: 22
  scan: False
```

Configuration variables:

- **sda** (*Optional, Pin*): The pin for the data line of the i²c bus. Defaults to the default of your board (usually GPIO21 for ESP32 and GPIO4 for ESP8266).
- **scl** (*Optional, Pin*): The pin for the clock line of the i²c bus. Defaults to the default of your board (usually GPIO22 for ESP32 and GPIO5 for ESP8266).
- **scan** (*Optional, boolean*): If esphomelib should do a search of the i2c address space on startup. Note that this can slow down startup and is only recommended for when setting up new sensors. Defaults to `False`.
- **frequency** (*Optional, float*): Set the frequency the i²c bus should operate on. Defaults to “100kHz”.
- **receive_timeout** (*Optional, Time*): Advanced: Set a timeout for operations on the i2c bus. Defaults to 100ms.

Note: If you’re using the ESP32 and i2c frequently is showing errors in the logs, try with the latest version of the Arduino framework. See [Using the latest Arduino framework version](#) for information on how to do this.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

IR Transmitter Component

The IR transmitter component lets you send infrared messages to control devices in your home. First, you need to setup a global hub that specifies which pin your IR led is connected to. Afterwards you can create *individual switches* that each send a pre-defined IR code to a device.

Note: This component is *much* more accurate on the ESP32, since that chipset has a dedicated peripheral for sending exact signal sequences.

```
# Example configuration entry
ir_transmitter:
  - id: 'ir_hub1'
    pin: 32

# Individual switches
switch:
  - platform: ir_transmitter
    ir_transmitter_id: 'ir_hub1'
    name: "Panasonic TV Off"
    panasonic:
      address: 0x4004
      command: 0x100BCBD
```

Configuration variables:

- **pin** (**Required**, *Pin*): The pin of the IR LED.
- **carrier_duty_percent** (*Optional*, int): The duty percentage of the carrier. 50 for example means that the LED will be on 50% of the time. Must be in range from 1 to 100. Defaults to 50.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation. Use this if you have multiple IR transmitters.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

Logger Component

The logger component automatically logs all log messages through the serial port and through MQTT topics. By default, all logs with a severity higher than `DEBUG` will be shown. Decreasing the log level can help with the performance of the application and memory size.

```
# Example configuration entry
logger:
  level: DEBUG
```

Configuration variables:

- **baud_rate** (*Optional*, int): The baud rate to use for the serial UART port. Defaults to 115200.
- **tx_buffer_size** (*Optional*, string): The size of the buffer used for log messages. Decrease this if you're having memory problems. Defaults to 512.
- **level** (*Optional*, string): The global log level. Any log message with a lower severity will not be shown. Defaults to `DEBUG`.
- **logs** (*Optional*, mapping): Manually set the log level for a specific component or tag. See [Manual Log Levels for more information](#).
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.

Log Levels

Possible log levels are (sorted by severity):

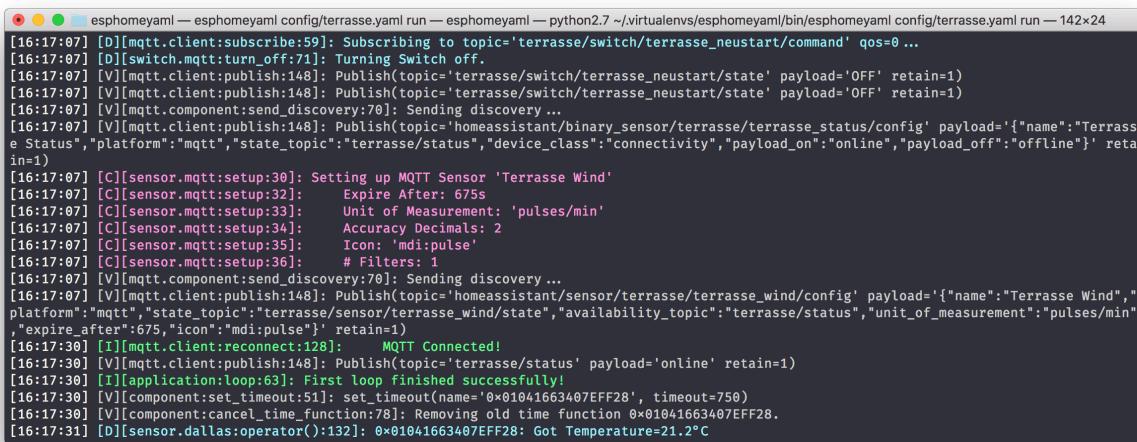
- `NONE`
- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `VERBOSE`

- VERY_VERBOSE

Manual Tag-Specific Log Levels

If some component is spamming the logs and you want to manually set the log level for it, first identify the tag of the log messages in question and then disable them in your configuration.

Suppose we want to have verbose log messages globally, but the MQTT client spams too much. In the following example, we'd first see that the tag of the MQTT client is `mqtt.client` (before the first colon) and the tag for MQTT components is `mqtt.component`.



```
[16:17:07] [D][mqtt.client:subscribe:59]: Subscribing to topic='terrasse/switch/terrasse_neustart/command' qos=0 ...
[16:17:07] [D][switch.mqtt:turn_off:71]: Turning Switch off.
[16:17:07] [V][mqtt.client:publish:148]: Publish(topic='terrasse/switch/terrasse_neustart/state' payload='OFF' retain=1)
[16:17:07] [V][mqtt.client:publish:148]: Publish(topic='terrasse/switch/terrasse_neustart/state' payload='OFF' retain=1)
[16:17:07] [V][mqtt.component:send_discovery:70]: Sending discovery ...
[16:17:07] [V][mqtt.client:publish:148]: Publish(topic='homeassistant/binary_sensor/terrasse_terrasse_status/config' payload='{"name": "Terrasse Status", "platform": "mqtt", "state_topic": "terrasse/status", "device_class": "connectivity", "payload_on": "online", "payload_off": "offline"}' retain=1)
[16:17:07] [C][sensor.mqtt:setup:30]: Setting up MQTT Sensor 'Terrasse Wind'
[16:17:07] [C][sensor.mqtt:setup:32]:   Expire After: 675s
[16:17:07] [C][sensor.mqtt:setup:33]:   Unit of Measurement: 'pulses/min'
[16:17:07] [C][sensor.mqtt:setup:34]:   Accuracy Decimals: 2
[16:17:07] [C][sensor.mqtt:setup:35]:   Icon: 'mdi:pulse'
[16:17:07] [C][sensor.mqtt:setup:36]:   # Filters: 1
[16:17:07] [V][mqtt.component:send_discovery:70]: Sending discovery ...
[16:17:07] [V][mqtt.client:publish:148]: Publish(topic='homeassistant/sensor/terrasse_terrassewind/config' payload='{"name": "Terrasse Wind", "platform": "mqtt", "state_topic": "terrasse/sensor/terrasse_wind/state", "availability_topic": "terrasse/status", "unit_of_measurement": "pulses/min", "expire_after": 675, "icon": "mdi:pulse"}' retain=1)
[16:17:30] [I][mqtt.client:reconnect:128]:   MQTT Connected!
[16:17:30] [V][mqtt.client:publish:148]: Publish(topic='terrasse/status' payload='online' retain=1)
[16:17:30] [I][application:loop:63]: First loop finished successfully!
[16:17:30] [V][component:set_timeout:51]: set_timeout(name='0x01041663407EFF28', timeout=750)
[16:17:30] [V][component:cancel_time_function:78]: Removing old time function 0x01041663407EFF28.
[16:17:31] [D][sensor.dallas:operator():132]: 0x01041663407EFF28: Got Temperature=21.2°C
```

Next, we can manually set the log levels in the configuration like this:

```
logger:
  level: VERBOSE
  logs:
    mqtt.component: DEBUG
    mqtt.client: ERROR
```

Please note that the global log level determines what log messages are saved in the binary. So for example a `INFO` global log message will purge all `DEBUG` log statements from the binary in order to conserve space. This however means that you cannot set tag-specific log levels that have a lower severity than the global log level.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

MQTT Client Component

The MQTT Client Component sets up the MQTT connection to your broker and is currently required for esphomelib to work. In most cases, you will just be able to copy over the [MQTT section](#) of your Home Assistant configuration.

```
# Example configuration entry
mqtt:
  broker: 10.0.0.2
  username: livingroom
  password: MyMQTTPassword
```

Configuration variables:

- **broker** (**Required**, string): The host of your MQTT broker.
- **port** (*Optional*, int): The port to connect to. Defaults to 1883.
- **username** (*Optional*, string): The username to use for authentication. Empty (the default) means no authentication.
- **password** (*Optional*, string): The password to use for authentication. Empty (the default) means no authentication.
- **client_id** (*Optional*, string): The client id to use for opening connections. See [Defaults](#) for more information.
- **discovery** (*Optional*, boolean): If Home Assistant automatic discovery should be enabled. Defaults to `True`.
- **discovery_retain** (*Optional*, boolean): Whether to retain MQTT discovery messages so that entities are added automatically on Home Assistant restart. Defaults to `True`.
- **discovery_prefix** (*Optional*, string): The prefix to use for Home Assistant's MQTT discovery. Should not contain trailing slash. Defaults to `homeassistant`.
- **topic_prefix** (*Optional*, string): The prefix used for all MQTT messages. Should not contain trailing slash. Defaults to `<APP_NAME>`.
- **log_topic** (*Optional*, [MQTTMessage](#)): The topic to send MQTT log messages to.
- **birth_message** (*Optional*, [MQTTMessage](#)): The message to send when a connection to the broker is established. See [Last Will And Birth Messages](#) for more information.
- **will_message** (*Optional*, [MQTTMessage](#)): The message to send when the MQTT connection is dropped. See [Last Will And Birth Messages](#) for more information.
- **ssl_fingerprints** (*Optional*, list): Only on ESP8266. A list of SHA1 hashes used for verifying SSL connections. See [SSL Fingerprints](#) for more information.
- **keepalive** (*Optional*, [Time](#)): The time to keep the MQTT socket alive, decreasing this can help with overall stability due to more WiFi traffic with more pings. Defaults to 15 seconds.
- **on_message** (*Optional*, [Automation](#)): An action to be performed when a message on a specific MQTT topic is received. See [on_message](#).
- **id** (*Optional*, [ID](#)): Manually specify the ID used for code generation.

MQTTMessage

With the MQTT Message schema you can tell esphomeyaml how a specific MQTT message should be sent. It is used in several places like last will and birth messages or MQTT log options.

```
# Simple:  
some_option: topic/to/send/to  
  
# Disable:  
some_option:  
  
# Advanced:  
some_option:  
  topic: topic/to/send/to  
  payload: online  
  qos: 0  
  retain: True
```

Configuration options:

- **topic** (*Required*, string): The MQTT topic to publish the message.
- **payload** (*Required*, string): The message content. Will be filled by the actual payload with some options, like log_topic.
- **qos** (*Optional*, int): The [Quality of Service](#) level of the topic. Defaults to 0.
- **retain** (*Optional*, boolean): If the published message should have a retain flag on or not. Defaults to `True`.

Using with Home Assistant

Using esphomelib with Home Assistant is easy, simply setup an MQTT broker (like [mosquitto](#)) and point both your Home Assistant installation and esphomelib to that broker. Next, enable discovery in your Home Assistant configuration with the following:

```
# Example Home Assistant configuration.yaml entry  
mqtt:  
  broker: ...  
  discovery: True
```

And that should already be it All devices defined through esphomelib/esphomeyaml should show up automatically in the entities section of Home Assistant.

When adding new entities, you might run into trouble with old entities still appearing in Home Assistant's front-end. This is because in order to have Home Assistant "discover" your devices on restart, all discovery MQTT messages need to be retained. Therefore the old entities will also re-appear on every Home Assistant restart even though they're in esphomeyaml anymore.

To fix this, esphomeyaml has a simple helper script that purges stale retained messages for you:

```
esphomeyaml configuration.yaml clean-mqtt
```

This will remove all retained messages with the topic <DISCOVERY_PREFIX>/+/NODE_NAME/#. If you want to purge on another topic, simply add --topic <your_topic> to the command.

Defaults

By default, esphomelib will prefix all messages with your node name or `topic_prefix` if you have specified it manually. The client id will automatically be generated by using your node name and adding the MAC

address of your device to it. Next, discovery is enabled by default with Home Assistant's default prefix `homeassistant`.

If you want to prefix all MQTT messages with a different prefix, like `home/living_room`, you can specify a custom `topic_prefix` in the configuration. That way, you can use your existing wildcards like `home/+/#` together with esphomelib. All other features of esphomelib (like availability) should still work correctly.

Last Will And Birth Messages

esphomelib (and esphomeyaml) uses the `last will testament` and birth message feature of MQTT to achieve availability reporting for Home Assistant. If the node is not connected to MQTT, Home Assistant will show all its entities as unavailable (a feature `available`).

Wohnzimmer

	Heizkörper Ventilator	Unavailable
	Heizkörpertemperatur	Unavailable
	Raumtemperatur	Unavailable

By default, esphomelib will send a retained MQTT message to `<TOPIC_PREFIX>/status` with payload `online`, and will tell the broker to send a message `<TOPIC_PREFIX>/status` with payload `offline` if the connection drops.

You can change these messages by overriding the `birth_message` and `will_message` with the following options.

```
mqtt:
  # ...
  birth_message:
    topic: myavailability/topic
    payload: online
  will_message:
    topic: myavailability/topic
    payload: offline
```

- `birth_message` (*Optional, MQTTMessage*)
- `will_message` (*Optional, MQTTMessage*)

If the birth message and last will message have empty topics or topics that are different from each other, availability reporting will be disabled.

SSL Fingerprints

On the ESP8266 you have the option to use SSL connections for MQTT. This feature will get expanded to the ESP32 once the base library, AsyncTCP, supports it. Please note that the SSL feature only checks the SHA1 hash of the SSL certificate to verify the integrity of the connection, so every time the certificate

changes, you'll have to update the fingerprints variable. Additionally, SHA1 is known to be partially insecure and with some computing power the fingerprint can be faked.

To get this fingerprint, first put the broker and port options in the configuration and then run the `mqtt-fingerprint` script of esphomeyaml to get the certificate:

```
esphomeyaml livingroom.yaml mqtt-fingerprint
> SHA1 Fingerprint: a502ff13999f8b398ef1834f1123650b3236fc07
> Copy above string into mqtt.ssl_fingerprints section of livingroom.yaml
```

```
mqtt:
  # ...
  ssl_fingerprints:
    - a502ff13999f8b398ef1834f1123650b3236fc07
```

MQTT Component Base Configuration

All components in esphomelib that do some sort of communication through MQTT can have some overrides for specific options.

```
name: "Component Name"
# Optional variables:
retain: True
discovery: True
availability:
  topic: livingroom/status
  payload_available: online
  payload_not_available: offline
state_topic: livingroom/custom_state_topic
command_topic: livingroom/custom_command_topic
```

Configuration variables:

- **name** (**Required**, string): The name to use for the MQTT Component.
- **retain** (*Optional*, boolean): If all MQTT state messages should be retained. Defaults to `True`.
- **discovery** (*Optional*, boolean): Manually enable/disable discovery for a component. Defaults to the global default.
- **availability** (*Optional*): Manually set what should be sent to Home Assistant for showing entity availability. Default derived from *global birth/last will message*.
- **state_topic** (*Optional*, string): The topic to publish state updates to. Defaults to `<TOPIC_PREFIX>/<COMPONENT_TYPE>/<COMPONENT_NAME>/state`.
- **command_topic** (*Optional*, string): The topic to subscribe to for commands from the remote. Defaults to `<TOPIC_PREFIX>/<COMPONENT_TYPE>/<COMPONENT_NAME>/command`.

on_message

With this configuration option you can write complex automations whenever an MQTT message on a specific topic is received. To use the message content, use a *lambda* template, the message payload is available under the name `x` inside that lambda.

```
mqtt:
  # ...
  on_message:
    topic: my/custom/topic
    qos: 0
    then:
      - switch.turn_on:
          id: some_switch
```

Configuration variables:

- **topic** (**Required**, string): The MQTT topic to subscribe to and listen for MQTT messages on. Every time a message with **this exact topic** is received, the automation will trigger.

Note: Currently the topic does not support MQTT wildcards like + or #.

-
- **qos** (*Optional*, integer): The MQTT Quality of Service to subscribe to the topic with. Defaults to 0.

Note: You can even specify multiple `on_message` triggers by using a YAML list:

```
mqtt:
  on_message:
    - topic: some/topic
      then:
        - # ...
    - topic: some/other/topic
      then:
        - # ...
```

mqtt.publish Action

Publish an MQTT message on a topic using this action in automations.

```
on_...:
  then:
    - mqtt.publish:
        topic: some/topic
        payload: "Something happened!"

    # Templated:
    - mqtt.publish:
        topic: !lambda >-
          if (id(reed_switch).value) return "topic1";
          else return "topic2";
        payload: !lambda >-
          return id(reed_switch).value ? "YES" : "NO";
```

Configuration options:

- **topic** (*Required*, string, *templatable*): The MQTT topic to publish the message.
- **payload** (*Required*, string, *templatable*): The message content.
- **qos** (*Optional*, int, *templatable*): The *Quality of Service* level of the topic. Defaults to 0.

- **retain** (*Optional*, boolean, *templatable*): If the published message should have a retain flag on or not. Defaults to `False`.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

OTA Update Component

With the OTA (Over The Air) update component you can upload your firmware binaries to your node without having to use an USB cable for uploads. esphomeyaml natively supports this through its `run` and `upload` helper scripts.

Optionally, you can also define a password to use for OTA updates so that an intruder isn't able to upload any firmware to the ESP without having hardware access to it. This password is also hashed automatically, so an intruder can't extract the password from the binary.

esphomelib also supports an “OTA safe mode”. If for some reason your node gets into a boot loop, esphomelib will automatically try to detect this and will go over into a safe mode after 10 unsuccessful boot attempts. In that mode, all components are disabled and only Serial Logging+WiFi+OTA are initialized, so that you can upload a new binary.

```
# Example configuration entry
ota:
  safe_mode: True
  password: VERYSECURE
```

Configuration variables:

- **safe_mode** (*Optional*, boolean): Whether to enable safe mode. Defaults to `True`.
- **password** (*Optional*, string): The password to use for updates.
- **port** (*Optional*, int): The port to use for OTA updates. Defaults to 3232 for the ESP32 and 8266 for the ESP8266.
- **id** (*Optional*, *ID*): Manually specify the ID used for code generation.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

PCA9685 PWM Component

The PCA9685 component represents a PCA9685 12-bit PWM driver ([datasheet](#), [adafruit](#)) in esphomelib. It uses *I²C Bus* for communication.

To use the channels of this components, you first need to setup the global `pca9685` hub and give it an id, and then define the *individual output channels*.

```
# Example configuration entry
pca9685:
  frequency: 500

# Individual outputs
output:
  - platform: pca9685
    pca9685_id: 'pca9685_hub1'
    channel: 0
```

Configuration variables:

- **frequency** (**Required**, float): The frequency to let the component drive all PWM outputs at. Must be in range from 24Hz to 1526Hz.
- **address** (*Optional*, int): The I²C address of the driver. Defaults to 0x00.
- **id** (*Optional*, ID): The id to use for this pca9685 component. Use this if you have multiple PCA9685s connected at the same time

See Also

- *PCA9685 PWM Output*
- *API Reference*
- [Edit this page on GitHub](#)

PCF8574 I/O Expander

The PCF8574 component allows you to use PCF8574 or PCF8575 I/O expanders ([datasheet](#), [Sparkfun](#)) in esphomeyaml. It uses *I²C Bus* for communication.

Once configured, you can use any of the 8 pins (PCF8574) or 16 pins (PCF8575) as pins for your projects. Within esphomelib they emulate a real internal GPIO pin and can therefore be used with many of esphomelib's components such as the GPIO binary sensor or GPIO switch.

Any option accepting a *Pin Schema* can theoretically be used, but some more complicated components that do communication through this I/O expander will not work.

```
# Example configuration entry
pcf8574:
  - id: 'pcf8574_hub'
    address: 0x21
    pcf8575: False

# Individual outputs
switch:
  - platform: gpio
    name: "PCF8574 Pin #0"
    pin:
      pcf8574: pcf8574_hub
      # Use pin number 0
      number: 0
```

(continues on next page)

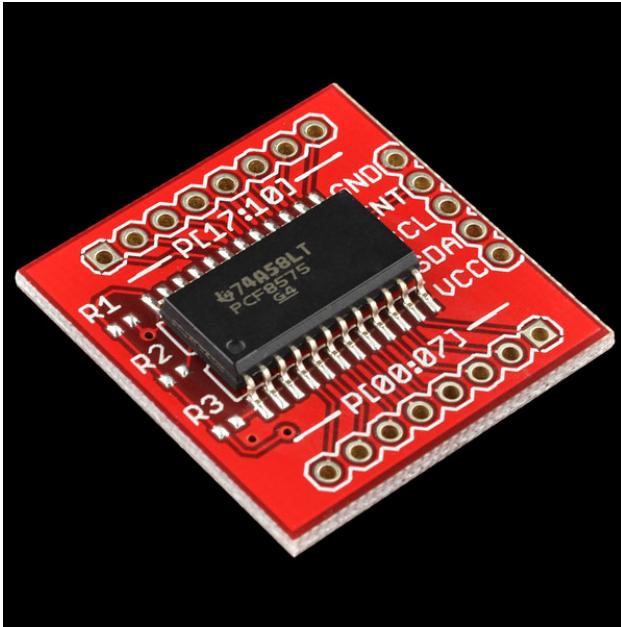


Fig. 22: PCF8574 I/O Expander. Image by Sparkfun.

(continued from previous page)

```
# One of INPUT, INPUT_PULLUP or OUTPUT
mode: OUTPUT
inverted: False
```

Configuration variables:

- **id (Required, *ID*)**: The id to use for this PCF8574 component.
- **address (Optional, int)**: The I²C address of the driver. Defaults to 0x21.
- **pcf8575 (Optional, boolean)**: Whether this is a 16-pin PCF8575. Defaults to False.

See Also

- *I²C Bus*
- *GPIO Switch*
- *GPIO Binary Sensor*
- *API Reference*
- Edit this page on GitHub

Power Supply Component

The `power_supply` component allows you to have a high power mode for certain outputs. For example, if you're using an `ATX power supply` to power your LED strips, you usually don't want to have the power supply on all the time while the output is not on. The power supply component can be attached to any

Output Component and will automatically switch on if any of the outputs are on. Furthermore, it also has a cooldown time that keeps the power supply on for a while after the last output has been disabled.

```
# Example configuration entry
power_supply:
- id: 'power_supply1'
  pin: 13
```

Configuration variables:

- **id (Required, ID)**: The id of the power supply so that it can be used by the outputs.
- **pin (Required, Pin Schema)**: The GPIO pin to control the power supply on.
- **enable_time (Optional, Time)**: The time to that the power supply needs for startup. The output component will wait for this period of time after turning on the PSU and before switching the output on. Defaults to 20ms.
- **keep_on_time (Optional, Time)**: The time the power supply should be kept enabled after the last output that used it has been switch off. Defaults to 10s.

See the [output component base configuration](#) for information on how to apply the power supply for a specific output.

ATX Power Supplies



© Otto Winter, 2018

The power supply component will default to pulling the specified GPIO pin up when high power mode is needed. Most ATX power supplies however operate with an active-low configuration. Therefore their output needs to be inverted.

```
power_supply:
  - id: 'atx_power_supply'
    pin:
      number: 13
      inverted: true
```

Then simply connect the green control wire from the ATX power supply to your specified pin. It's recommended to put a small resistor (about $1\text{k}\Omega$) in between to protect the ESP board.

See Also

- [Output Component](#)
- [API Reference](#)
- [Edit this page on GitHub](#)

Web Server Component

The `web_server` component creates a simple web server on the node that can be accessed through any browser and a simple REST API. Please note that enabling this component will take up *a lot* of memory and can lead to problems, especially on the ESP8266.

To navigate to the web server in your browser, either use the IP address of the node or use `<node_name>.local/` (note the trailing forward slash) via mDNS.

To conserve flash size, the CSS and JS files used on the root page to show a simple user interface are hosted by esphomelib.com. If you want to use your own service, use the `css_url` and `js_url` options in your configuration.

```
# Example configuration entry
web_server:
  port: 80
```

Configuration variables:

- **port** (*Optional, int*): The port the web server should open its socket on.
- **css_url** (*Optional, url*): The URL that should be used for the CSS stylesheet. Defaults to `https://esphomelib.com/_static/webserver-v1.min.css` (updates will go to v2, v3, etc).
- **js_url** (*Optional, url*): The URL that should be used for the JS script. Defaults to `https://esphomelib.com/_static/webserver-v1.min.js`.
- **id** (*Optional, ID*): Manually specify the ID used for code generation.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

terrasse Web Server

States

Name	State	Actions
Terrasse Wind	0.00 pulses/min	
Terrasse Regen	0.00 pulses/min	
Terrasse Temperatur	45.1 °C	
Terrasse Neustart	OFF	<input type="button" value="Toggle"/>
Terrasse Status	ON	

See [esphomelib Web API](#) for REST API documentation.

Debug Log

```
[V] [component:set_timeout:53]: set_timeout(name='0x01041663407EFF28', timeout=750)
[D] [sensor.pulse_counter:update:125]: 0: Retrieved counter (raw=0): 0.00 pulses/min
[V] [sensor.sensor:push_new_value:22]: 'Terrasse Wind': Received new value 0.000000
[V] [sensor.sensor:push_new_value:28]: 'Terrasse Wind': Filter #0 aborted chain
[D] [sensor.pulse_counter:update:125]: 1: Retrieved counter (raw=0): 0.00 pulses/min
[V] [sensor.sensor:push_new_value:22]: 'Terrasse Regen': Received new value 0.000000
[V] [sensor.sensor:push_new_value:28]: 'Terrasse Regen': Filter #0 aborted chain
[D] [sensor.dallas:operator():127]: 'Terrasse Temperatur': Got Temperature=44.9°C
[V] [sensor.sensor:push_new_value:22]: 'Terrasse Temperatur': Received new value 44.937500
[V] [sensor.sensor:push_new_value:28]: 'Terrasse Temperatur': Filter #0 aborted chain
```

Fig. 23: Example web server frontend.

WiFi Component

This core esphomelib component sets up WiFi connections to access points for you. It needs to be in your configuration or otherwise esphomeyaml will fail in the config validation stage.

It's recommended to provide a static IP for your node, as it can dramatically improve connection times.

```
# Example configuration entry
wifi:
  ssid: MyHomeNetwork
  password: VerySafePassword

  # Optional manual IP
  manual_ip:
    static_ip: 10.0.0.42
    gateway: 10.0.0.1
    subnet: 255.255.255.0
```

Configuration variables:

- **ssid** (*Optional*, string): The name (or service set identifier) of the WiFi access point your device should connect to.
- **password** (*Optional*, string): The password (or PSK) for your WiFi network. Leave empty for no password.
- **manual_ip** (*Optional*): Manually configure the static IP of the node.
 - **static_ip** (*Required*, IPv4 address): The static IP of your node.
 - **gateway** (*Required*, IPv4 address): The gateway of the local network.
 - **subnet** (*Required*, IPv4 address): The subnet of the local network.
 - **dns1** (*Optional*, IPv4 address): The main DNS server to use.
 - **dns2** (*Optional*, IPv4 address): The backup DNS server to use.
- **hostname** (*Optional*, string): Manually set the hostname of the node. Can only be 63 long at max and must only contain alphanumeric characters plus dashes and underscores.
- **ap** (*Optional*): Enable an access point mode on the node.
 - **ssid** (*Required*, string): The name of the access point to create.
 - **password** (*Optional* string): The password for the access point. Leave empty for no password.
 - **channel** (*Optional*, int): The channel the AP should operate on from 1 to 14. Defaults to 1.
 - **manual_ip** (*Optional*): Manually set the IP options for the AP. Same options as manual_ip for station mode.
- **domain** (*Optional*, string): Set the domain of the node hostname used for uploading. For example, if it's set to .local, all uploads will be sent to <HOSTNAME>.local. Defaults to .local.
- **id** (*Optional*, ID): Manually specify the ID used for code generation.

Access Point Mode

Since version 1.3, esphomelib has an optional “Access Point Mode”. If you include `ap:` in your wifi configuration, esphomelib will automatically set up an access point that you can connect to. Additionally, you can specify both a “normal” station mode and AP mode at the same time. This will cause esphomelib to only enable the access point when no connection to the wifi router can be made.

See Also

- [API Reference](#)
- [Edit this page on GitHub](#)

Cookbook

Simple Garage Door

The following is a possible configuration file for garage doors that are controlled by two relays: One for opening and another one for closing the garage door. When either one of them is turned on for a short period of time, the close/open action begins.

```
switch:
  - platform: gpio
    pin: D3
    name: "Garage Door Open Switch"
    id: open_switch
  - platform: gpio
    pin: D4
    name: "Garage Door Close Switch"
    id: close_switch
cover:
  - platform: template
    name: "Garage Door"
    open_action:
      # Cancel any previous action
      - switch.turn_off:
          id: close_switch
      # Turn the OPEN switch on briefly
      - switch.turn_on:
          id: open_switch
      - delay: 0.1s
      - switch.turn_off:
          id: open_switch
    close_action:
      - switch.turn_off:
          id: open_switch
      - switch.turn_on:
          id: close_switch
      - delay: 0.1s
      - switch.turn_off:
          id: close_switch
    stop_action:
      - switch.turn_off:
          id: close_switch
```

(continues on next page)

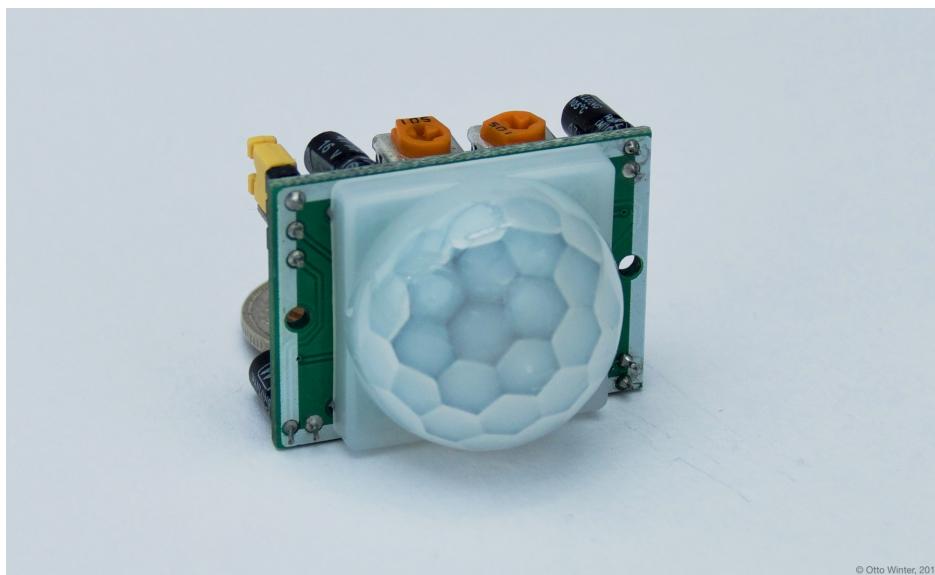
(continued from previous page)

```
- switch.turn_off:  
  id: open_switch  
optimistic: true
```

See Also

- [Automations And Templates](#)
- [Template Cover](#)
- [Edit this page on GitHub](#)

Passive Infrared Sensor

© Otto Winter, 2018

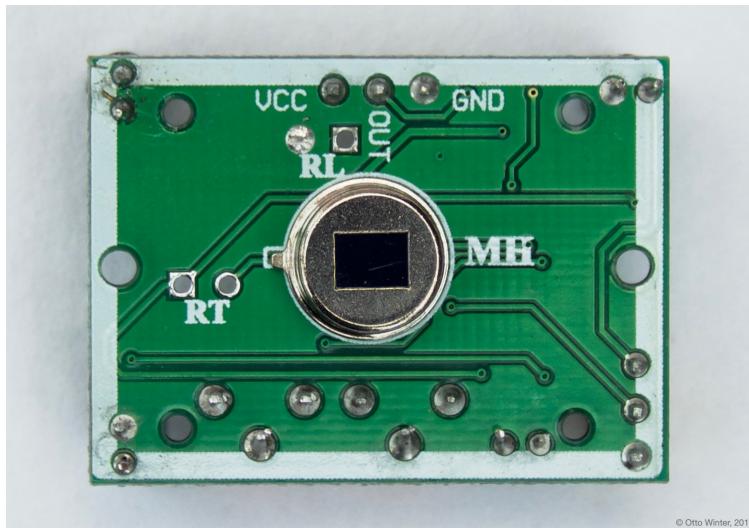
Passive Infrared Sensors (or PIR sensors for short) are completely supported by esphomelib. These sensors measure the infrared light emitted from objects in its field of view, and if it detects a sudden change between different parts of the sensing area, the signal is pulled high.

Connecting the PIR sensor is also quite simple. You need to connect GND to a GND pin on your board and VCC to a 5V or 12V pin. Technically you can also connect VCC to 3.3V, but the sensor measurements won't be as stable.

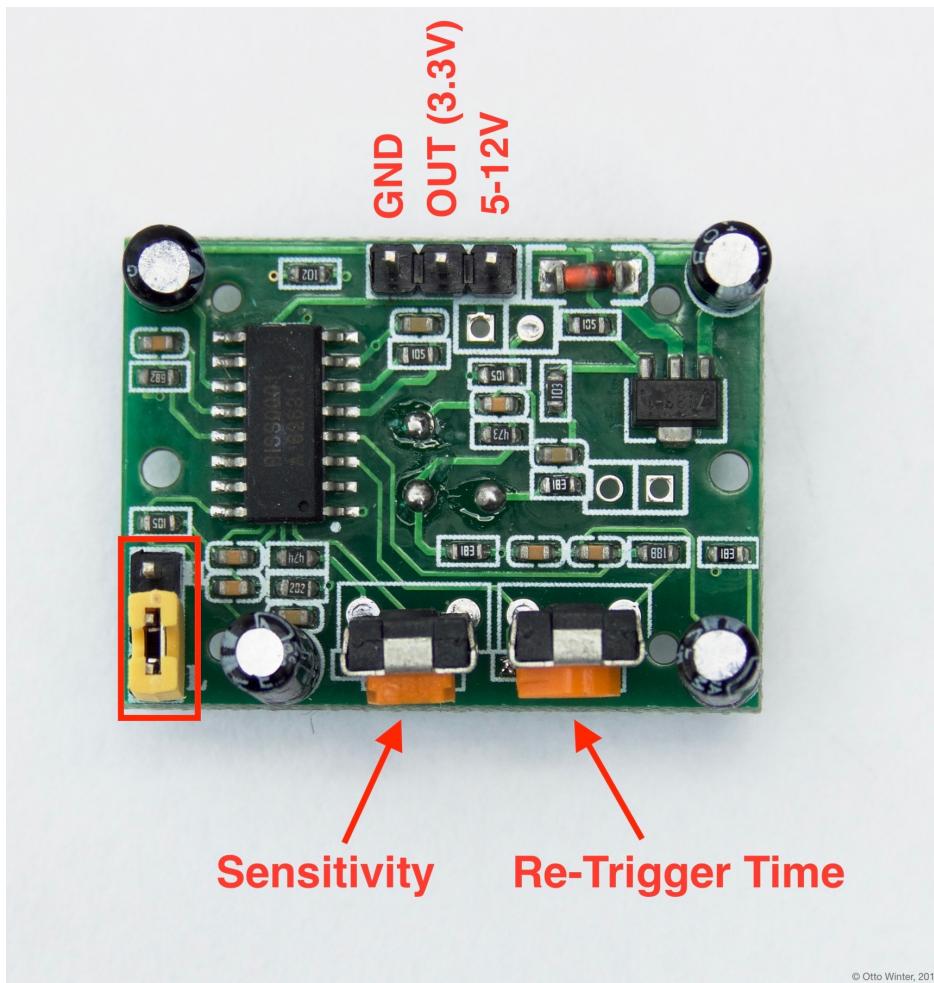
Next you need to connect the signal pin (OUT). Fortunately, the sensor signal has a voltage of 3.3V max, so we can directly connect it to a free GPIO pin on the ESP board. Otherwise, we would need to step down the voltage in order to not damage the ESP.

Warning: Some PIR sensors have the GND and power supply pins swapped, please open the front cover to see which pin mapping your PIR sensor is using to make sure.

On the back side you will additionally find two knobs that you can turn to change the sensor sensitivity and time the signal will stay active for once motion has been detected. Turning these clockwise will increase sensitivity/re-trigger time.



© Otto Winter, 2018



© Otto Winter, 2018

To configure esphomeyaml for use with the PIR sensor, use a [GPIO Binary Sensor](#). It can detect if a pin is pulled HIGH/LOW and reports those values to Home Assistant. Optionally also set a `device_class` so that Home Assistant uses a nice icon for the binary sensor.

```
binary_sensor:  
- platform: gpio  
  pin: <PIN_PIR_SENSOR_IS_CONNECTED_TO>  
  name: "PIR Sensor"  
  device_class: motion
```



See Also

- Awesome article explaining how PIR Sensors work.
- [GPIO Binary Sensor](#)
- [BRUH Multisensor](#)
- Edit this page on GitHub

BRUH Multisensor

The BRUH Multisensor is a great introductory project into Home Automation with an amazing setup tutorial. And fortunately esphomelib has complete support for all the stuff used by the Multisensor

Thank you very much to [@jackjohnsonuk](#) for providing this configuration file

```
esphomeyaml:  
  name: <NODE_NAME>  
  platform: ESP8266  
  board: nodemcuv2  
  
  wifi:  
    ssid: <SSID>  
    password: <PASSWORD>  
  
  mqtt:  
    broker: <MQTT_BROKER>  
    username: <USER>  
    password: <PASSWORD>  
  
  # Enable logging  
  logger:  
  
  ota:  
  
  sensor:  
    - platform: dht  
      pin: D7  
      temperature:  
        name: "Multisensor Temperature"
```

(continues on next page)

(continued from previous page)

```
humidity:
  name: "Multisensor Humidity"
- platform: adc
  pin: A0
  name: "Multisensor Brightness"
  unit_of_measurement: lux
  filters:
    - lambda: >-
      return (x / 10000.0) * 2000000.0;

binary_sensor:
- platform: gpio
  pin: D5
  name: "Multisensor Motion"
  device_class: motion

output:
- platform: esp8266_pwm
  pin: D1
  id: redgpio
- platform: esp8266_pwm
  pin: D2
  id: greengpio
- platform: esp8266_pwm
  pin: D3
  id: bluegpio

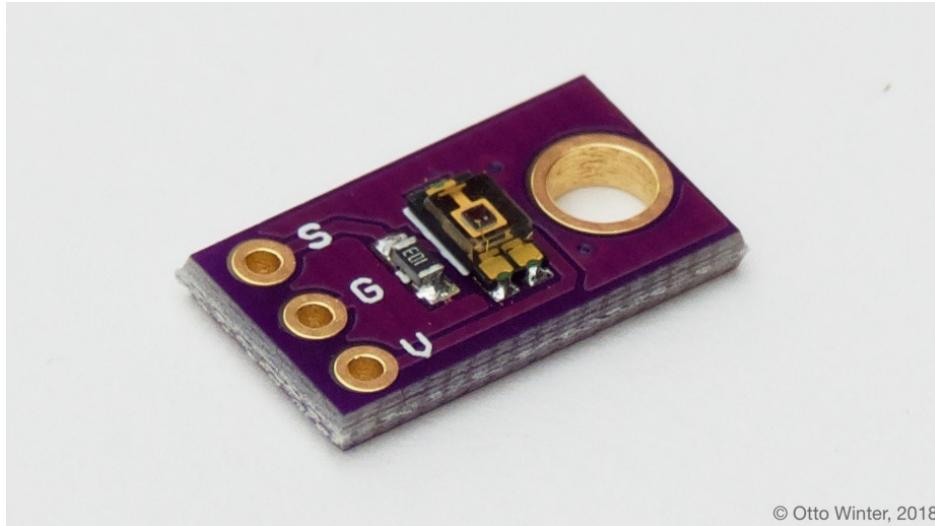
light:
- platform: rgb
  name: "Multisensor Light"
  red: redgpio
  green: greengpio
  blue: bluegpio
```

See Also

- [NodeMCU ESP8266](#)
- [Passive Infrared Sensor](#)
- [TEMT6000](#)
- [RGB Light](#)
- [ESP8266 Software PWM Output](#)
- [DHT Temperature+Humidity Sensor](#)
- [Analog To Digital Sensor](#)
- [GPIO Binary Sensor](#)
- [Edit this page on GitHub](#)

TEMT6000

The TEMT6000 is a simple and cheap ambient light sensor. The sensor itself changes its resistance based on how much light hits the sensor. In order for us to read this resistance the breakout boards you can buy these chips on often have a small constant value resistor and three pins: GND, VCC and SIG. Connect VCC to 3.3V, GND to GND and SIG to any available *analog pin*.



To get the brightness the sensor measures, we then simply have to measure the voltage on the **SIG** (also called **OUT**) pin using the *Analog To Digital Sensor* and convert those voltage measurements to illuminance values in lux using a formula:

```
sensor:  
  - platform: adc  
    pin: A0  
    name: "TEMT6000 Illuminance"  
    unit_of_measurement: lx  
    filters:  
      - lambda: >-  
        return (x / 10000.0) * 2000000.0;
```

Formula Explanation:

To get the illuminance in lux, we first need to convert the measured voltage to the current flowing across the TEMT6000 sensor. This current is also equal to the current flowing across the $10\text{k}\Omega$ resistor in the voltage divider circuit, which is $I = \text{adc_value}/10000\text{k}\Omega$.

The [datasheet](#) for the TEMT6000 specifies a proportional correlation between current and illuminance: Every 2 μA of current correlates to 1 lx in the illuminance.

Note: The default voltage range of the ADC for the ESP8266 and ESP32 are from 0 to 1.0V. So you won't be able to measure any value above 200 lx using the default setup.

For the ESP32, you have the option of setting a *Voltage Attenuation* (note that the formula doesn't need to be adjusted if you set an attenuation, as the value x is automatically converted to volts).

For the ESP8266, you unfortunately need to tinker with the hardware a bit to decrease the voltage a bit. So one option would be to create another voltage divider on the **SIG** pin to divide the analog voltage by a

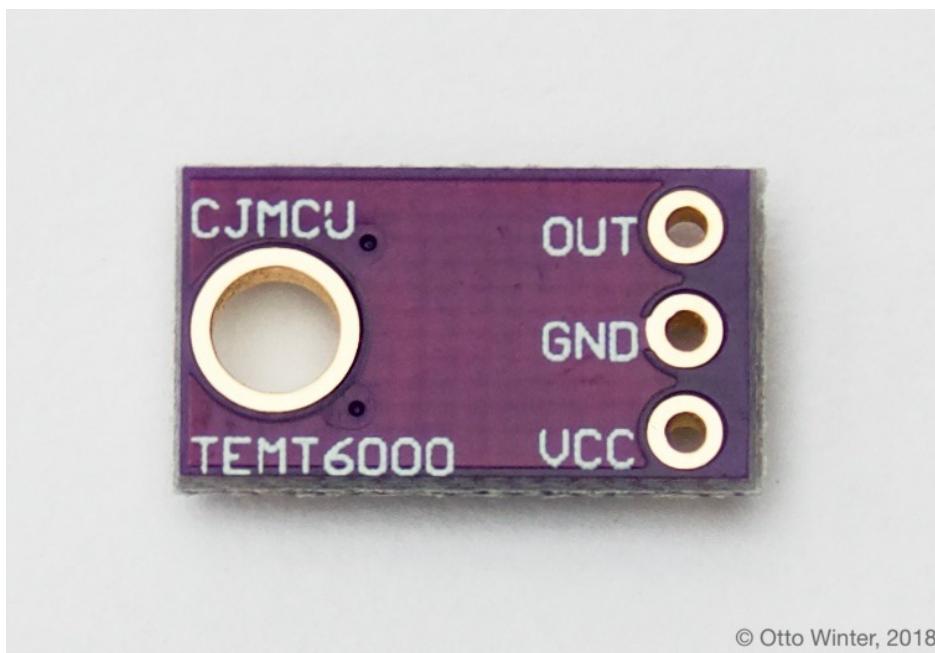


Fig. 24: Pins on the TEMT6000. Connect OUT to an ADC pin, GND to GND, and VCC to 3.3V

constant value.

See Also

- [Analog To Digital Sensor](#)
- [TEMT6000 datasheet](#)
- [BRUH Multisensor](#)
- [Edit this page on GitHub](#)

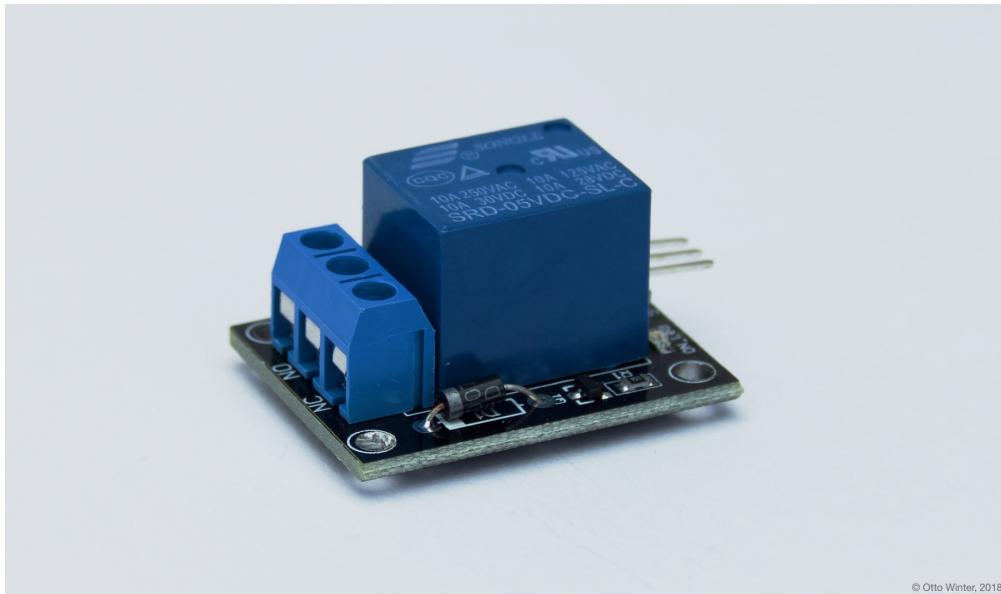
Relay

Relays are simple electronics components that allow you to switch a high load using only a single pin on your board.

On the inside of a relay there is a coil that's connected to the input signal. Every time the input signal switches on, the current flowing through the coil induces a magnetic field that closes a contact.

On the output side you have three pins: common (COM), normally open (NO) and normally closed (NC). Connect COM and one of NO and NC in series with your load. When the input turns on, NO is connected to COM and when the input turns off, NC is connected to COM.

Warning: Relays are a very common component in electronics, so there are *many* types of them and I won't attempt to describe all of them (partly due to my own lack of knowledge). Please check with the datasheet of your relay to see its current limits, what voltage it requires and so on. Please also note that



relays are physical components - as such, they have a limited number of switches that can happen before they break.

To use a relay with esphomelib, use a [GPIO Switch](#).

```
switch:  
  - platform: gpio  
    name: "Relay"  
    pin: <PIN_RELAY_IS_CONNECTED_TO>
```

See Also

- [GPIO Switch](#)
- [GPIO Output](#)
- How do relays work.
- Edit this page on GitHub

Non-Invasive Power Meter

So an essential part of making your home smart is knowing how much power it uses over the day. Tracking this can be difficult, often you need to install a completely new power meter which can often cost a bunch of money. However, quite a few power meters have a red LED on the front that blinks every time that one Wh has been used.

The simple idea therefore is: Why don't we just abuse that functionality to make the power-meter IoT enabled? We just have to hook up a simple photoresistor in front of that aforementioned LED and track the amount of pulses we receive. Then using esphomelib we can instantly have the power meter show up in Home Assistant

Note: This guide currently only works with the ESP32, and even if it is ported back to the ESP8266 at some point, the ESP32 will still achieve a much higher accuracy because it has a hardware-based pulse counter.

Hooking it all up is quite easy: Just buy a suitable photoresistor (make sure the wave length approximately matches the one from your power meter). Then connect it using a simple variable resistor divider (see [this article](#) for inspiration). And... that should already be it :)



For esphomelib, you can then use the [pulse counter sensor](#) using below configuration:

```
sensor:
  - platform: pulse_counter
    pin: GPIO12
    unit_of_measurement: 'kW'
    name: 'Power Meter'
    filters:
      - multiply: 0.06
```

Adjust GPIO12 to match your set up of course. The output from the pulse counter sensor is in `pulses/min` and we also know that 1000 pulses from the LED should equal 1kWh of power usage. Thus, rearranging the expression yields a proportional factor of 0.06 from `pulses/min` to `kW`.

And if a technician shows up and he looks confused about what the heck you have done to your power meter, tell them about esphomelib

Cookbook

Generic ESP32

All ESP32-based devices are supported by esphomeyaml. Simply select **ESP32** when the esphomeyaml wizard asks you for your platform and choose a board type from [this link](#) when the wizard asks you for the board

type.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP32
  board: <BOARD_TYPE>
```

The ESP32 boards often use the internal GPIO pin numbering on the board, this means that you don't have to worry about other kinds of pin numberings, yay!

Some notes about the pins on the ESP32:

- GPIO00 is used to determine the boot mode on startup. It should therefore not be pulled LOW on startup to avoid booting into flash mode. You can, however, still use this as an output pin.
- GPIO34-GPIO39 can not be used as outputs (even though GPIO stands for “general purpose input output”...)
- GPIO32-GPIO39: These pins can be used with the *Analog To Digital Sensor* to measure voltages.
- GPIO2: This pin is connected to the blue LED on the board as seen in above picture. It also supports the *touch pad binary sensor* like some other pins.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP32
  board: <BOARD_TYPE>

binary_sensor:
  - platform: gpio
    name: "Pin GPIO023"
    pin: GPIO023
```

See Also

- *NodeMCU ESP32*
- Edit this page on GitHub

Generic ESP8266

All ESP8266-based devices are supported by esphomeyaml. Simply select **ESP8266** when the esphomeyaml wizard asks you for your platform and choose a board type from [this link](#) when the wizard asks you for the board type.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP8266
  board: <BOARD_TYPE>
```

Many boards have a pin numbering for the exposed pins that is different from the internally used ones. esphomeyaml tries to map the silk-screen pin numbers into the internal pin numbers with a few boards, but for generic ESP8266 boards it is often required to just use the internal pin numbers. To do this, just prefix all pins with **GPIO**, for example **GPIO0** for the pin with the internal pin number 0.

Some notes on the pins:

- GPIO6 - GPIO11, GPIO0, GPIO2 and GPIO15 are often already used by the internal flash interface and boot mode detection. So it's best to avoid using these pins.
- GPIO17 additionally has an ADC connected to it. See the *Analog To Digital Sensor* to read out voltages (in the range from 0 to 1.0V) on this pin.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP8266
  board: nodemcuv2

binary_sensor:
  - platform: gpio
    name: "Pin GPIO17"
    pin: GPIO17
```

See Also

- [NodeMCU ESP8266](#)
- [Edit this page on GitHub](#)

NodeMCU ESP32

The NodeMCU ESP32 board (in some cases also known as ESP32-DevkitC) is fully supported by esphomeyaml. Simply select **ESP32** when the esphomeyaml wizard asks you for your platform and **nodemcu-32s** as the board type.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP32
  board: nodemcu-32s
```

The ESP32 boards often use the internal GPIO pin numbering on the board, this means that you don't have to worry about other kinds of pin numberings, yay!

Note that in certain conditions you *can* use the pins marked as **INTERNAL** in above image.

- GPIO0 is used to determine the boot mode on startup. It should therefore not be pulled LOW on startup to avoid booting into flash mode. You can, however, still use this as an output pin.
- GPIO34-GPIO39 can not be used as outputs (even though GPIO stands for “general purpose input **output**”...)
- GPIO32-GPIO39: These pins can be used with the *Analog To Digital Sensor* to measure voltages.
- GPIO2: This pin is connected to the blue LED on the board as seen in above picture. It also supports the *touch pad binary sensor* like some other pins.
- 5V is connected to the 5V rail from the USB bus and can be used to power the board. Note that the UART chip is directly connected to this rail and you therefore **cannot** supply other voltages into this pin.

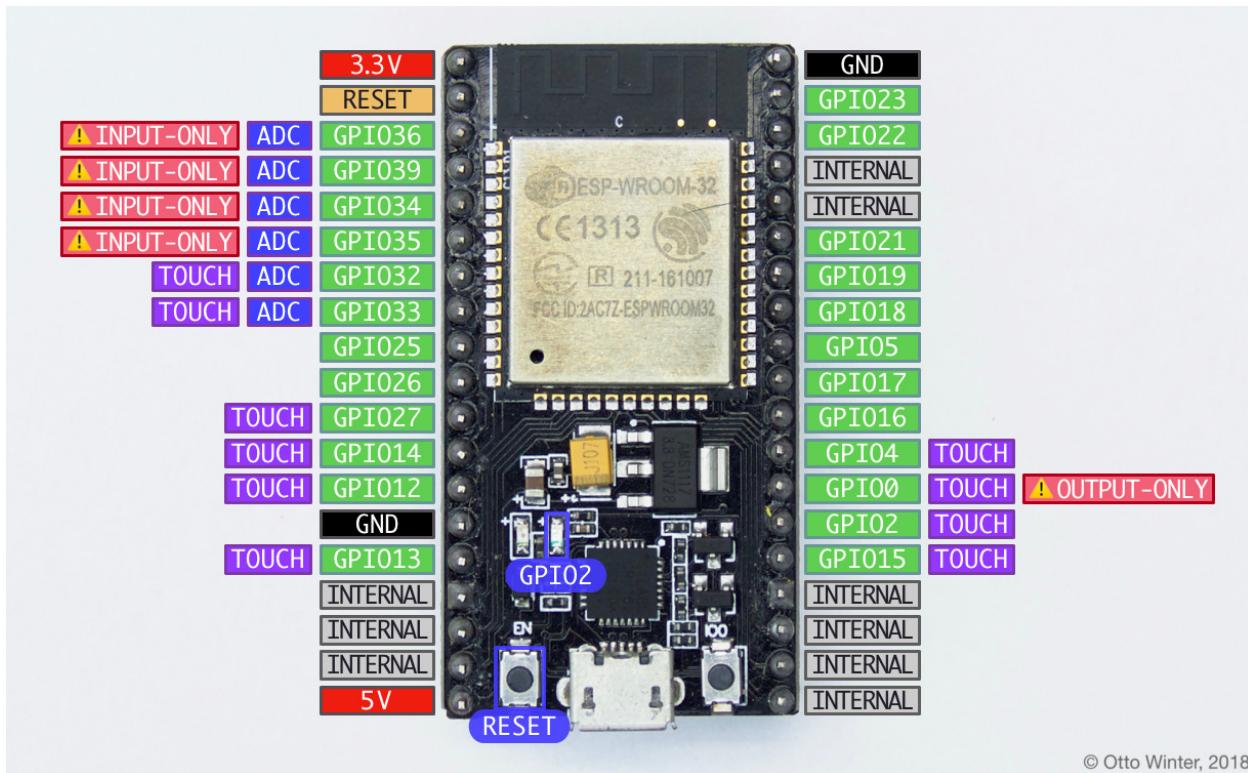


Fig. 25: Pins on the NodeMCU ESP32 development board.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP32
  board: nodemcu-32s

binary_sensor:
  - platform: gpio
    name: "Pin GPIO23"
    pin: GPIO23
```

See Also

- [Generic ESP32](#)
- [NodeMCU ESP8266](#)
- [Edit this page on GitHub](#)

NodeMCU ESP8266

The NodeMCU board is fully supported by esphomeyaml. Simply select `ESP8266` when the esphomeyaml wizard asks you for your platform and `nodemcuv2` as the board type.

Note: Most NodeMCU that can be purchased now are version 2 or upwards, if you're using an original v1 board, set the board type to `nodemcu`.

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP8266
  board: nodemcuv2
```

The NodeMCU's pin numbering as seen on the board (the D0 etc pins) is different from the internal pin numbering. For example, the D8 pin number maps to the internal GPIO0 pin. Fortunately esphomeyaml knows the mapping from the on-board pin numbers to the internal pin numbering, but you need to prefix the pin numbers with D as in below image in order for this automatic mapping to occur.

In general, it is best to just use the D0, D1, ... pin numbering to avoid confusion

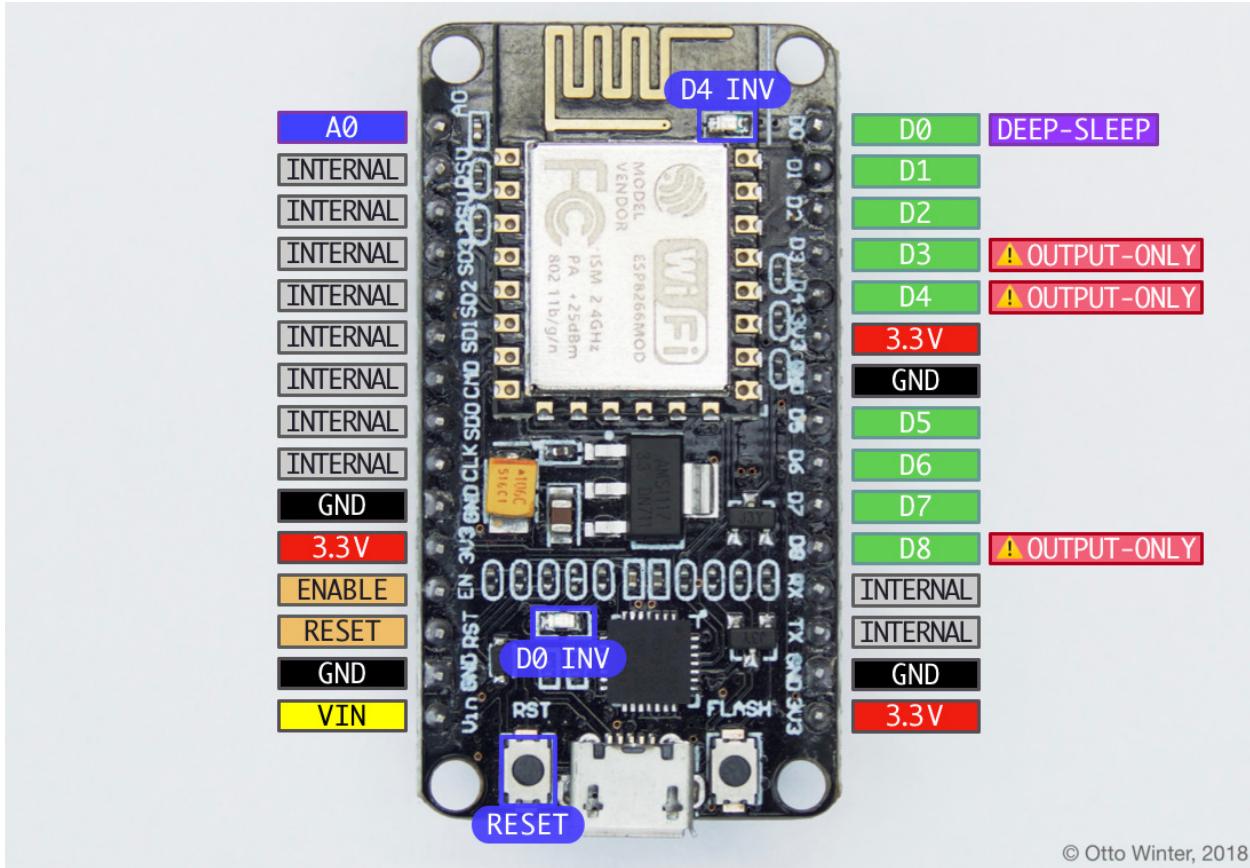


Fig. 26: Pins on the NodeMCU ESP8266 development board.

Note that in certain conditions you *can* use the pins marked as INTERNAL in above image.

- D0 also can be used to wake the device up from *deep sleep* if the pin is connected to the RESET pin. D0 is additionally connected to the LED next to the UART chip, but in an inverted mode.
- D3, D4 and D8 are used on startup to determine the boot mode, therefore these pins should not be pulled low on startup. You can, however, still use them as output pins.

- A0: This pin can be used as a normal GPIO pin (like D1 etc) but additionally can measure voltages from 0 to 1.0V using the *Analog To Digital Sensor*.
- VIN: This pin can be used to use an external power supply with the board. Supply a voltage from 3.3V to 12V to this pin and the linear voltage regulator on the board will power the board.
- ENABLE/RESET: When these pins are triggered, the board resets. The difference between the pins is how they can handle voltages above 3.3V

```
# Example configuration entry
esphomeyaml:
  name: livingroom
  platform: ESP8266
  board: nodemcuv2

binary_sensor:
  - platform: gpio
    name: "Pin D0"
    pin: D0
```

See Also

- *Generic ESP8266*
- *NodeMCU ESP32*
- [Edit this page on GitHub](#)

Generic Sonoff

In principle esphomelib supports all Sonoff devices, but as these devices are quite expensive and shipping from China takes a long time, I've only set up dedicated guides for the *Sonoff S20* and *Sonoff 4CH*.

To use sonoff devices with esphomeyaml, set the `board` in the *esphomeyaml section* to `esp01_1m` and set `board_flash_mode` to `dout`.

```
esphomeyaml:
  name: <NAME_OF_NODE>
  platform: ESP8266
  board: esp01_1m
  board_flash_mode: dout
```

After that use the following list of pin to function mappings to set up your Sonoff device. This list has been compiled from the Sonoff Tasmota pin source file which can be found here: https://github.com/arendst/Sonoff-Tasmota/blob/development/sonoff/sonoff_template.h

Sonoff RF

GPIO0	Button (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO4	Optional Sensor
GPIO12	Relay and Red LED
GPIO13	Green LED (inverted)
GPIO14	Optional Sensor

Sonoff SV

GPIO0	Button (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO4	Optional Sensor
GPIO5	Optional Sensor
GPIO12	Relay and Red LED
GPIO13	Green LED (inverted)
GPIO14	Optional Sensor
GPIO17	Analog Input

Sonoff TH

GPIO0	Button (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO4	Optional Sensor
GPIO12	Relay and Red LED
GPIO13	Green LED (inverted)
GPIO14	Optional Sensor

Slampher

GPIO0	Button (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO12	Relay and Red LED
GPIO13	Blue LED (inverted)

Sonoff Touch

GPIO0	Button (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO12	Relay and Red LED
GPIO13	Blue LED (inverted)

Sonoff LED

GPIO0	Button (inverted)
GPIO4	Green Channel
GPIO5	Red Channel
GPIO12	Cold White Channel
GPIO13	Blue LED (inverted)
GPIO14	Warm White Channel
GPIO15	Blue Channel

Sonoff T1, Sonoff T2, Sonoff T3

GPIO0	Button #1 (inverted)
GPIO1	RX pin (for external sensors)
GPIO2	Optional Sensor
GPIO3	TX pin (for external sensors)
GPIO4	Relay #3 and Blue LED
GPIO5	Relay #2 and Blue LED
GPIO9	Button #2 (inverted)
GPIO10	Button #3 (inverted)
GPIO12	Relay #1 and Blue LED
GPIO13	Blue LED (inverted)

Arilux LC01

GPIO0	Optional Button
GPIO1	RX pin (for external sensors)
GPIO2	RF receiver (unsupported yet)
GPIO3	TX pin (for external sensors)
GPIO5	Red Channel
GPIO12	Green Channel
GPIO13	Blue Channel
GPIO14	White Channel

Arilux LC11

GPIO0	Optional Button
GPIO1	RX pin (for external sensors)
GPIO2	RF receiver (unsupported yet)
GPIO3	TX pin (for external sensors)
GPIO4	Green Channel
GPIO5	Red Channel
GPIO12	Warm White Channel
GPIO13	Cold White Channel
GPIO14	Blue Channel
GPIO15	RF Receiver (unsupported yet)

Arilux LC06

GPIO0	Optional Button
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO12	Green Channel
GPIO13	Blue Channel
GPIO14	Red Channel
GPIO15	White Channel

Sonoff Dual R2

GPIO0	Button 0 on header (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO5	Relay #2
GPIO9	Button 1 on header (inverted)
GPIO10	Button on casing (inverted)
GPIO12	Relay #1
GPIO13	Blue LED (inverted)

Sonoff S31

GPIO0	Button (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)
GPIO5	Relay #2
GPIO12	Relay and red LED
GPIO13	Green LED (inverted)

See Also

- [Using With Sonoff S20](#)
- [Using With Sonoff 4CH](#)
- [Using With Sonoff Basic](#)
- [Generic ESP8266](#)
- [Edit this page on GitHub](#)

Using With Sonoff 4CH

esphomeyaml can also be used with Sonoff 4CH wireless switches. These devices are basically just an ESP8266 chip with 4 relays to control power output, a few buttons on the top and a few status LEDs.



Fig. 27: Sonoff 4CH WiFi switch.

This guide will step you through setting up your Sonoff 4CH and flashing the first esphomeyaml firmware with the serial interface. After that, you will be able to upload all future firmwares with the remote Over-The-Air update process.

Note: If you've previously installed Sonoff-Tasmota on your Sonoff 4CH, you're in luck – esphomeyaml can generate a firmware binary which you can then upload via the Tasmota web interface. To see how to create this binary, skip to [Step 3: Creating Firmware](#).

Since firmware version 1.6.0, iTead (the creator of this device) has removed the ability to upload a custom firmware through their own upload process. Unfortunately, that means that the only way to flash the initial esphomeyaml firmware is by physically opening the device up and using the UART interface.

Warning: Opening up this device can be very dangerous if not done correctly. While the device is open, you will be a single touch away from being electrocuted if the device is plugged in.

So, during this *entire* guide **never ever** plug the device in. Also, you should only do this if you know what you're doing. If you, at any step, feel something is wrong or are uncomfortable with continuing, it's best to just stop for your own safety.

It's your own responsibility to make sure everything you do during this setup process is safe.

For this guide you will need:

- Sonoff 4CH
- An USB to UART Bridge for flashing the device. These can be bought on Amazon for less than 5 dollars. Note that the bridge *must* be 3.3V compatible. Otherwise you will destroy your Sonoff.
- Jumper wires to connect the UART bridge to the header pins.
- Computer running esphomeyaml or HassIO add-on.
- Screwdriver to open up the Sonoff 4CH.

Have everything? Great! Then you can start.

Step 1: Opening up the Sonoff 4CH

The first step is to open up the Sonoff 4CH. Note that you do not have to run the original firmware supplied with the Sonoff 4CH before doing this step.

Warning: Just to repeat this: Make **absolutely sure** the device is not connected to any appliance or plugged in before doing this step.

While the device is not plugged in, turn the device so you are viewing it from the top, then unscrew the long screws in the four corners of the top cover.

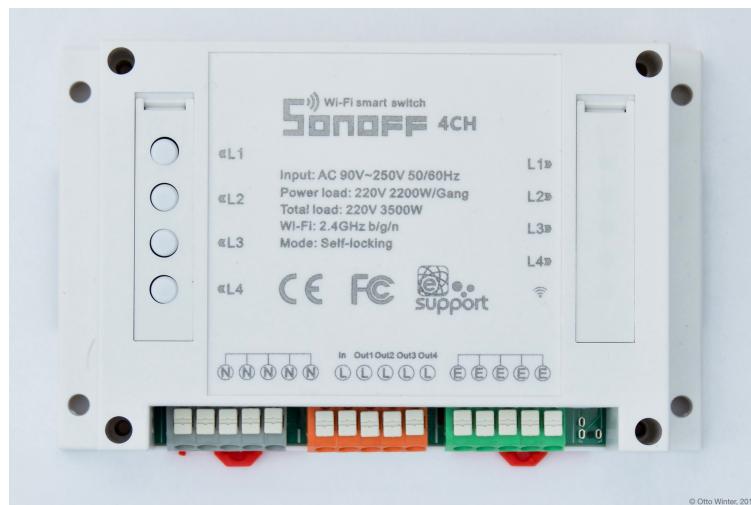


Fig. 28: There are four screws on the front of the Sonoff 4CH.

After that, you should be able to remove the front cover and should be greeted by the main board. The chip we're interested in here is the “big” one encased in an aluminium cover.

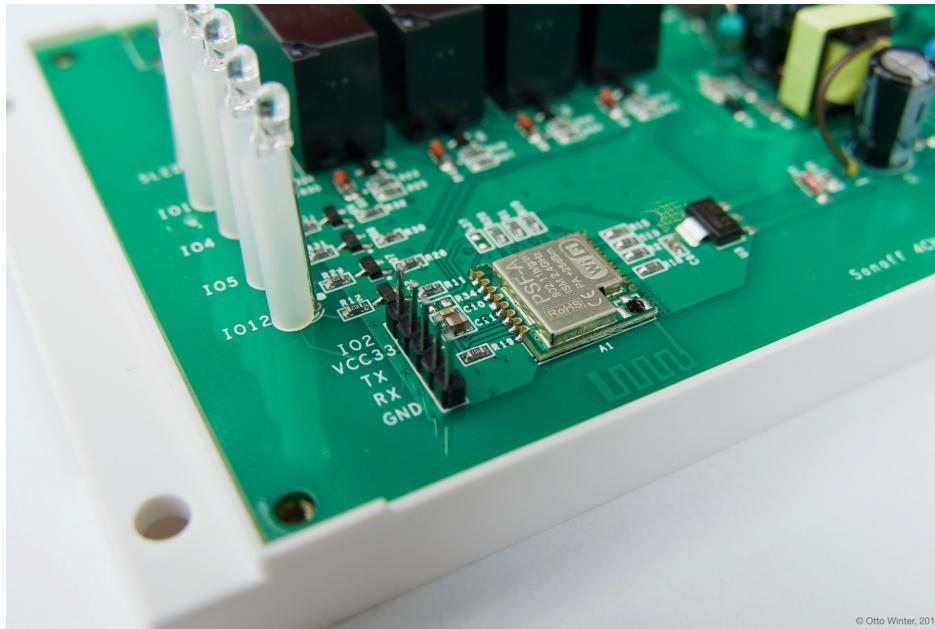


Fig. 29: The main chip of the Sonoff 4CH and the header pins we’re going to use to flash our custom firmware.

Step 2: Connecting UART

Now we need our computer to somehow establish a data connection to the board. For this we will have to connect the four wires on the UART to USB bridge to the UART pins of the Sonoff 4Ch.

Fortunately for us, exactly these pins come pre-populated with a few header pins. You can identify these by the **VCC33**, **RX**, **TX** and **GND** markings on the silk-screen.

Now go ahead and connect these pins to your UART to USB bridge as seen in below image. Make sure that you connect these correctly, especially the **VCC33** and **GND** parts as you can otherwise destroy the chip.

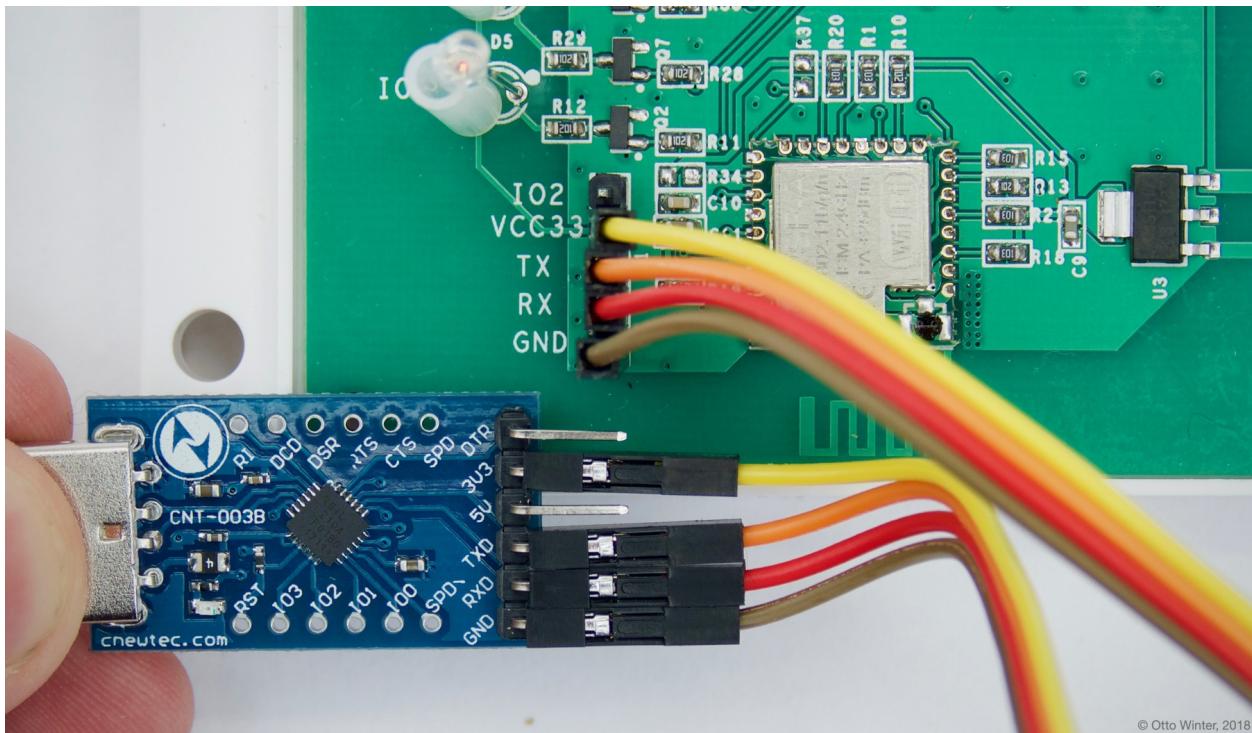
VCC33 should be connected to the **3V3 (not 5V)** pin of the UART bridge, **GND** to **GND** and the same with **RX/TX**.

When you’re done, it should look something like this:

Note: On some older 4CHs, the **RX** and **TX** pins are swapped (sometimes even the written silkscreen is wrong). If your upload fails with a `error: espcomm_upload_mem failed` message it’s most likely due to the pins being swapped. In that case, just swap **RX** and **TX** and try again - you won’t break anything if they’re swapped.

Step 3: Creating Firmware

The Sonoff 4CH is based on the **ESP8266** platform (technically it’s the **ESP8285**, but for our purposes they’re the same) and is a subtype of the **esp01_1m** board. With this information, you can step through



the esphomeyaml wizard (`esphomeyaml sonoff_4ch.yaml wizard`), or alternatively, you can just take the below configuration file and modify it to your needs.

If you go through the wizard, please make sure you manually set `board_flash_mode` to `dout` as seen below. The version of the uploader used by esphomeyaml should automatically detect that the Sonoff 4CH uses the `dout` SPI flash chip mode. But, as some users of other firmwares have said that other flash modes can brick the device, it's always good to specify it explicitly.

```
esphomeyaml:
  name: <NAME_OF_NODE>
  platform: ESP8266
  board: esp01_1m
  board_flash_mode: dout

  wifi:
    ssid: <YOUR_SSID>
    password: <YOUR_PASSWORD>

  mqtt:
    broker: <YOUR_MQTT_BROKER>
    username: <YOUR_USERNAME>
    password: <YOUR_PASSWORD>

  logger:

  ota:
```

Now run `esphomeyaml sonoff_4ch.yaml compile` to validate the configuration and pre-compile the firmware.

Note: After this step, you will be able to find the compiled binary under `<NAME_OF_NODE>/pioenvs/`

`<NAME_OF_NODE>/firmware.bin`. If you're having trouble with uploading, you can also try uploading this file directly with other tools.

Step 4: Uploading Firmware

In order to upload the firmware, you're first going to need to get the chip into a flash mode, otherwise the device will start up without accepting any firmware flash attempts. To do this, while the device is UART bridge is not connected to your USB port, start pressing the bottom-left push button labeled FW/I00 and continue to do so while plugging in the UART bridge into your computer. Keep holding the button for another 2-4 seconds. The 4CH should now be in a flash mode and should not blink with any LED.



© Otto Winter, 2018

Fig. 30: You need to press the button labeled FW/I00 during startup.

Now you can finally run the upload command:

```
esphomeyaml sonoff_4ch.yaml run
```

If successful, you should see something like this:

Hooray ! You've now successfully uploaded the first esphomeyaml firmware to your Sonoff 4CH. And in a moment, you will be able to use all of esphomeyaml's great features with your Sonoff 4CH.

If above step does, however, not work, here are some steps that can help:

- Sometimes the UART bridge cannot supply enough current to the chip to operate, in this case use a 3.3V supply you have lying around. A nice hack is to use the power supply of NodeMCU boards.

```

Looking for upload port...
Use manually specified: /dev/cu.SLAB_USBtoUART
Uploading .pioenvs/sonoff_4ch/firmware.bin
Uploading 360704 bytes from .pioenvs/sonoff_4ch/firmware.bin to flash at 0x00000000
.... [ 22% ]
.... [ 45% ]
.... [ 67% ]
.... [ 90% ]
.... [ 100% ]
===== [SUCCESS] Took 42.06 seconds =====
INFO [esphomeyaml.__main__] Successfully uploaded program.
INFO [esphomeyaml.__main__] Starting log output from /dev/cu.SLAB_USBtoUART with baud rate 115200
[11:46:08][I][wifi:280]: WiFi connected.
[11:46:08][C][wifi:281]:     IP Address: 192.168.178.128
[11:46:08][C][wifi:282]:     Subnet: 255.255.255.0
[11:46:08][C][wifi:283]:     Gateway: 192.168.178.1
[11:46:08][C][wifi:284]:     DNS1: 192.168.178.1
[11:46:08][C][wifi:285]:     DNS2: 0.0.0.0
[11:46:08][C][mqtt.client:024]: Setting up MQTT ...
[11:46:08][C][mqtt.client:026]:     Server Address: 192.168.178.84:1883
[11:46:08][C][mqtt.client:027]:     Username: 'debug'

```

Simply connect the NodeMCU's 3.3V to VCC and GND to GND. **Do not attempt to plug the device into a socket to overcome this problem while troubleshooting.**

- In other cases the TX and RX pin are reversed. Simple disconnect the device, swap the two pins and put it into flash mode again.

Step 5: Adding the Button, Relay and LEDs

Now we would like the 4CH to actually do something, not just connect to WiFi and pretty much sit idle.

Below you will find a table of all usable GPIO pins of the Sonoff 4CH and a configuration file that exposes all of the basic functions.

GPIO0	Button #1 (inverted)
GPIO9	Button #2 (inverted)
GPIO10	Button #3 (inverted)
GPIO14	Button #4 (inverted)
GPIO12	Relay #1 and red LED
GPIO5	Relay #2 and red LED
GPIO4	Relay #3 and red LED
GPIO15	Relay #4 and red LED
GPIO13	Blue LED (inverted)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)

```

esphomeyaml:
  name: <NAME_OF_NODE>
  platform: ESP8266
  board: esp01_1m
  board_flash_mode: dout

  wifi:
    ssid: <YOUR_SSID>
    password: <YOUR_PASSWORD>

```

(continues on next page)

(continued from previous page)

```

mqtt:
  broker: <YOUR_MQTT_BROKER>
  username: <YOUR_USERNAME>
  password: <YOUR_PASSWORD>

logger:

ota:

binary_sensor:
  - platform: gpio
    pin:
      number: GPIO00
      mode: INPUT_PULLUP
      inverted: True
    name: "Sonoff 4CH Button 1"
  - platform: gpio
    pin:
      number: GPIO09
      mode: INPUT_PULLUP
      inverted: True
    name: "Sonoff 4CH Button 2"
  - platform: gpio
    pin:
      number: GPIO010
      mode: INPUT_PULLUP
      inverted: True
    name: "Sonoff 4CH Button 3"
  - platform: gpio
    pin:
      number: GPIO014
      mode: INPUT_PULLUP
      inverted: True
    name: "Sonoff 4CH Button 4"
  - platform: status
    name: "Sonoff 4CH Status"

switch:
  - platform: gpio
    name: "Sonoff 4CH Relay 1"
    pin: GPIO012
  - platform: gpio
    name: "Sonoff 4CH Relay 2"
    pin: GPIO05
  - platform: gpio
    name: "Sonoff 4CH Relay 3"
    pin: GPIO04
  - platform: gpio
    name: "Sonoff 4CH Relay 4"
    pin: GPIO015

output:
  # Register the blue LED as a dimmable output . . .
  - platform: esp8266_pwm
    id: blue_led

```

(continues on next page)

(continued from previous page)

```

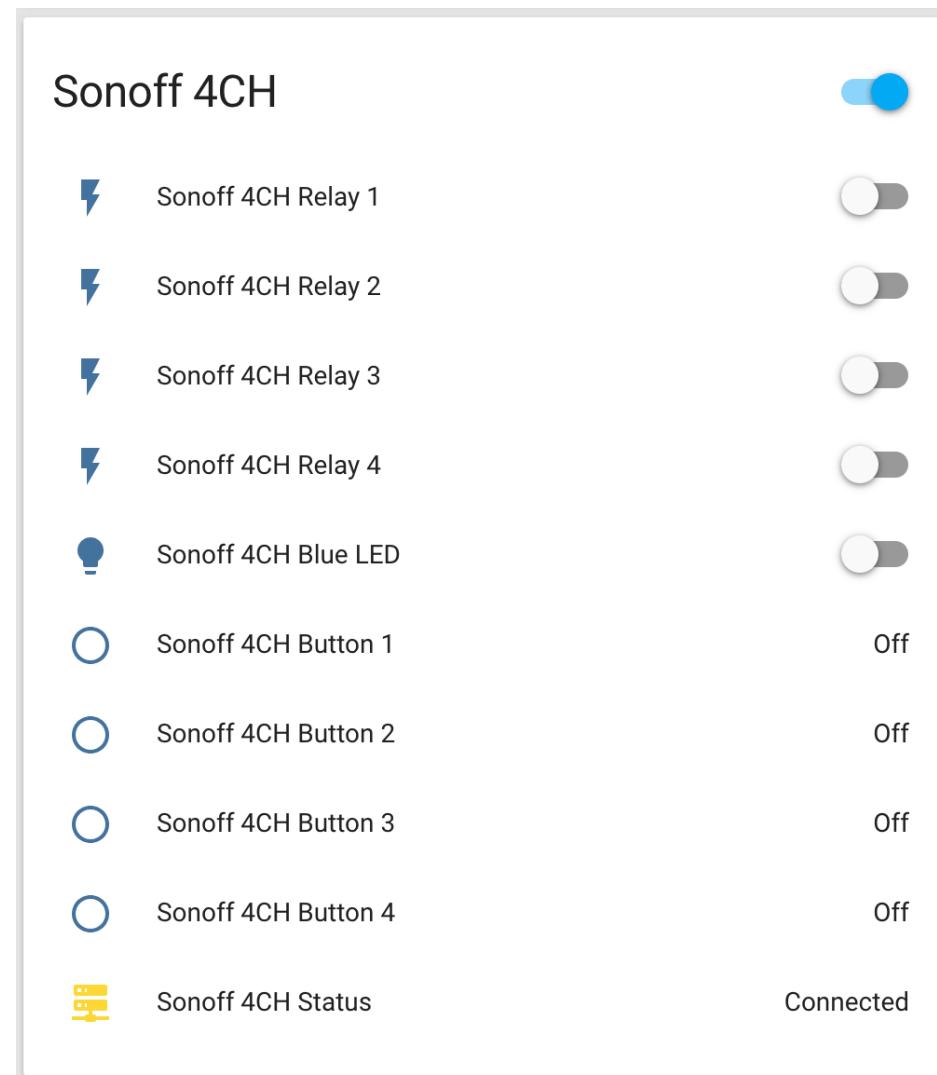
pin: GPIO13
inverted: True

light:
# ... and then make a light out of it.
- platform: monochromatic
  name: "Sonoff 4CH Blue LED"
  output: blue_led

```

Above example also showcases an important concept of esphomeyaml: IDs and linking. In order to make all components in esphomeyaml as much “plug and play” as possible, you can use IDs to define them in one area, and simply pass that ID later on. For example, above you can see an PWM (dimmer) output being created with the ID `blue_led` for the blue LED. Later on it is then transformed into a *monochromatic light*. If you additionally want the buttons to control the relays, look at [the complete Sonoff 4CH with automation example](#).

Upload the firmware again (through OTA or Serial) and you should immediately see something like this in Home Assistant because of esphomeyaml’s automatic MQTT discovery. (You’ll of course have to add them to groups if you have a `default_view` set):



Step 6: Finishing Up

If you're sure everything is done with the 4CH and have double checked there's nothing that could cause a short in the case, you can put the front cover with the button on the base again and screw everything together.

Now triple or even quadruple check the UART bridge is not connected to the 4CH, then comes the time when you can connect it.

Happy hacking!

See Also

- [Generic Sonoff](#)
- [Using With Sonoff S20](#)
- [Edit this page on GitHub](#)

Using With Sonoff S20

esphomeyaml can also be used with Sonoff S20 smart sockets. These devices are basically just an ESP8266 chip with a relay to control the socket, a small button on the front and a blue and green LED light.



Fig. 31: Sonoff S20 Smart Socket.

This guide will step you through setting up your Sonoff S20 and flashing the first esphomeyaml firmware with the serial interface. After that, you will be able to upload all future firmwares with the remote Over-The-Air update process.

Note: If you've previously installed Sonoff-Tasmota on your Sonoff S20, you're in luck – esphomeyaml can generate a firmware binary which you can then upload via the Tasmota web interface. To see how to create

this binary, skip to [Step 3: Creating Firmware](#).

Since firmware version 1.6.0, iTea (the creator of this device) has removed the ability to upload a custom firmware through their own upload process. Unfortunately, that means that the only way to flash the initial esphomeyaml firmware is by physically opening the device up and using the UART interface.

Warning: Opening up this device can be very dangerous if not done correctly. While the device is open, you will be a single touch away from being electrocuted if the device is plugged in.

So, during this *entire* guide **never ever** plug the device in. Also, you should only do this if you know what you're doing. If you, at any step, feel something is wrong or are uncomfortable with continuing, it's best to just stop for your own safety.

It's your own responsibility to make sure everything you do during this setup process is safe.

For this guide you will need:

- Sonoff S20
- An USB to UART Bridge for flashing the device. These can be bought on Amazon for less than 5 dollars. Note that the bridge *must* be 3.3V compatible. Otherwise you will destroy your S20.
- Computer running esphomeyaml HassIO add-on.
- Screwdriver to open up the S20.
- Soldering iron and a few header pins to connect the UART interface.

Have everything? Great! Then you can start.

Step 1: Opening up the Sonoff S20

The first step is to open up the Sonoff S20. Note that you do not have to run the original firmware supplied with the Sonoff S20 before doing this step.

Warning: Just to repeat this: Make **absolutely sure** the device is not connected to any appliance or plugged in before doing this step.

While the device is not plugged in, turn the back side so it's facing you and unscrew the three black screws holding the back of the case together with the front.

After that, you should be able to remove the front cover and should be greeted by a bunch of parts.

Step 2: Connecting UART

We're interested in the main part of the S20 with the green PCB. On the bottom of the PCB, you will find four unpopulated holes. These pins have the UART interface used to flash firmwares onto the device and debug issues.

So, in order to flash our own custom firmware, we're going to need to somehow connect the UART to USB bridge to these pins. The only way to make a good connection here is by using a soldering iron and soldering on some pin headers. On older models of the Sonoff S20, you were able to get the whole PCB out. Newer versions, however, glue the PCB onto the case to avoid people flashing custom firmwares. If the latter is the case, you will need to just solder the pin headers from above - it's a bit difficult, but possible.



Fig. 32: There are three screws on the back of the Sonoff S20.



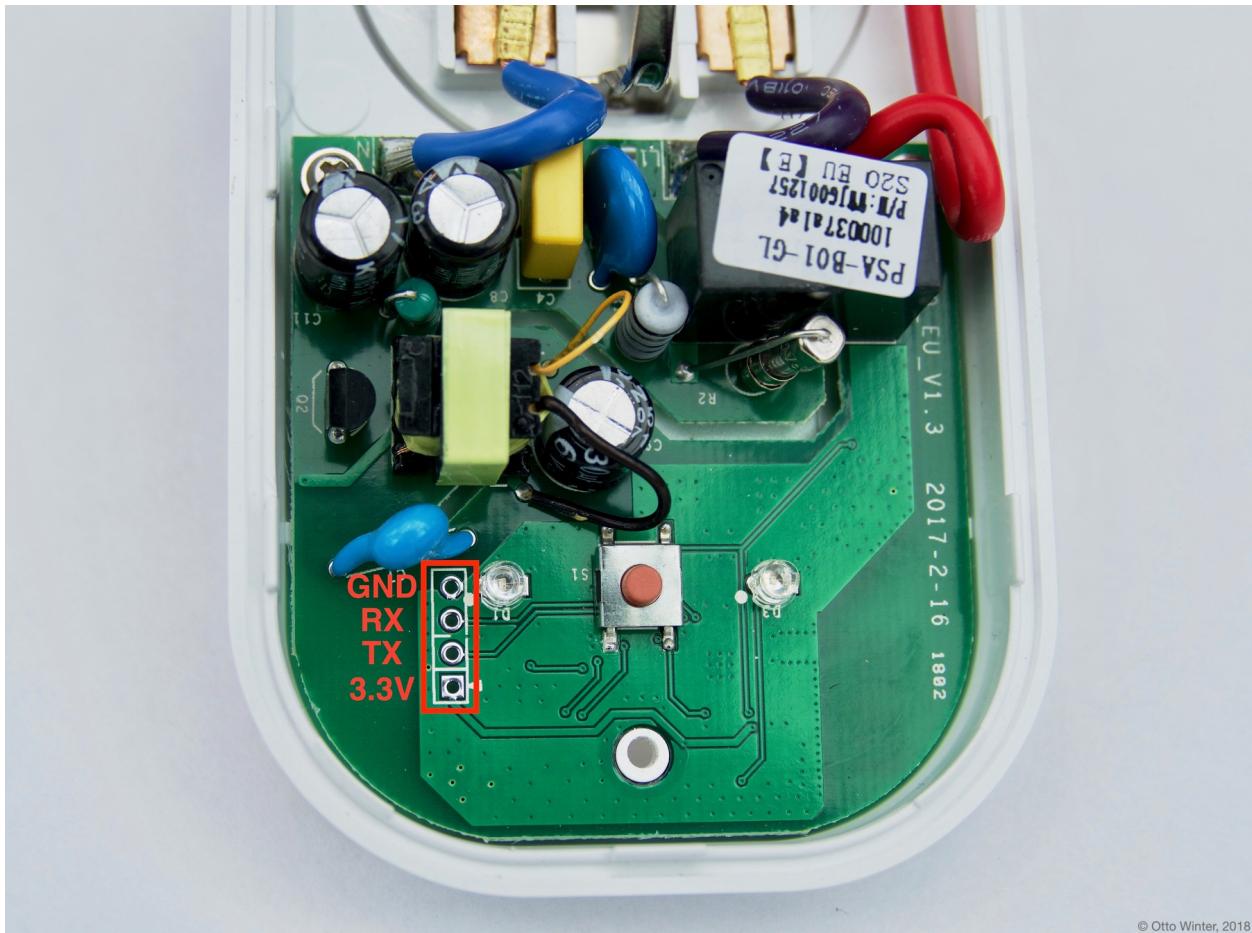
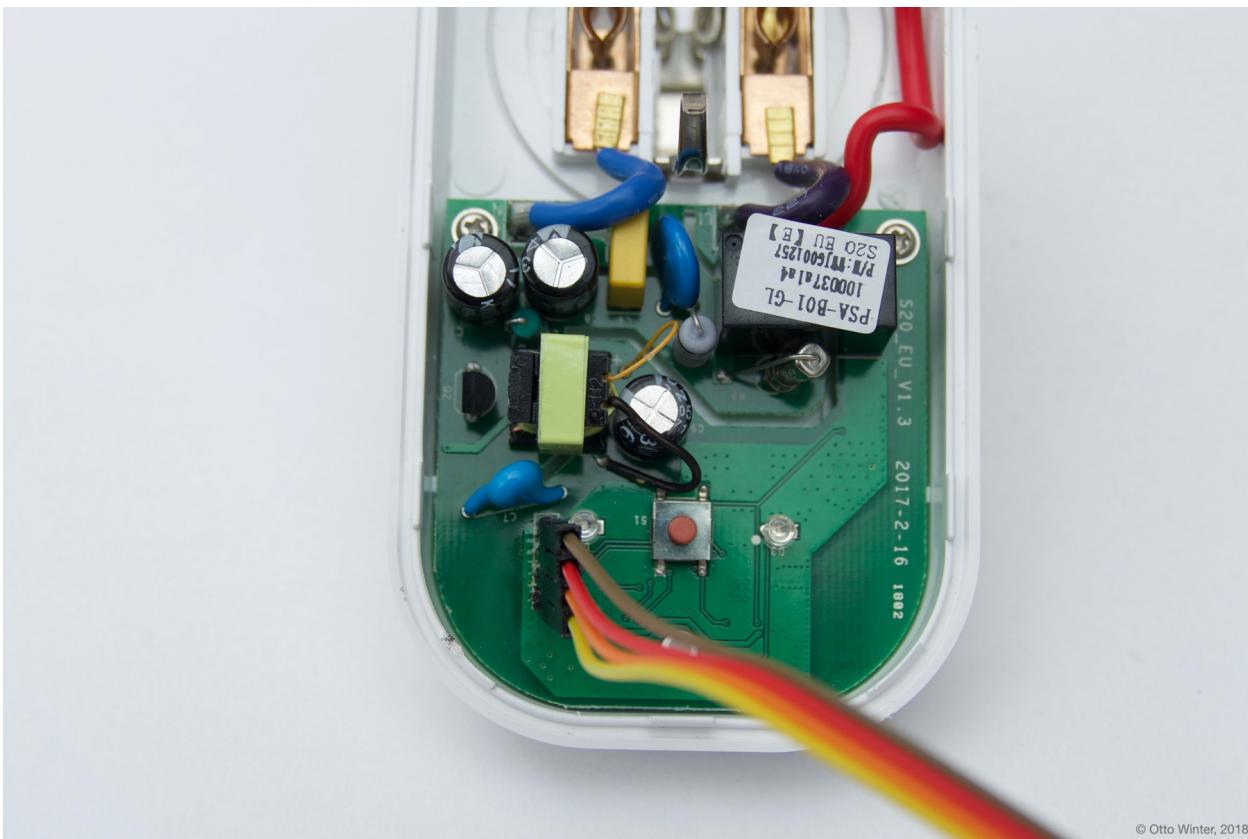


Fig. 33: The UART interface of the Sonoff S20.

© Otto Winter, 2018

When you're done, it should look something like this:



Now go ahead and connect the pins to your UART bridge, making sure the S20 is not plugged in as before. Also beware that some UART to USB bridges supply 5V on the VCC pin if it's not explicitly labeled 3.3V. It's best to just use a multimeter and double check if it's unclear.

Note: On some older S20s, the RX and TX pins are swapped (sometimes even the written silkscreen is wrong). If your upload fails with a `error: espcomm_upload_mem failed` message it's most likely due to the pins being swapped. In that case, just swap RX and TX and try again - you won't break anything if they're swapped.

Step 3: Creating Firmware

The Sonoff S20 is based on the ESP8266 platform and is a subtype of the `esp01_1m` board. With this information, you can step through the esphomeyaml wizard (`esphomeyaml sonoff_s20.yaml wizard`), or alternatively, you can just take the below configuration file and modify it to your needs.

If you go through the wizard, please make sure you manually set `board_flash_mode` to `dout` as seen below. The version of the uploader used by esphomeyaml should automatically detect that the Sonoff S20 uses the `dout` SPI flash chip mode. But, as some users of other firmwares have said that other flash modes can brick the device, it's always good to specify it explicitly.

```
esphomeyaml:  
  name: <NAME_OF_NODE>  
  platform: ESP8266
```

(continues on next page)

(continued from previous page)

```

board: esp01_1m
board_flash_mode: dout

wifi:
  ssid: <YOUR_SSID>
  password: <YOUR_PASSWORD>

mqtt:
  broker: <YOUR_MQTT_BROKER>
  username: <YOUR_USERNAME>
  password: <YOUR_PASSWORD>

logger:

ota:

```

Now run `esphomeyaml sonoff_s20.yaml compile` to validate the configuration and pre-compile the firmware.

Note: After this step, you will be able to find the compiled binary under `<NAME_OF_NODE>/pioenvs/<NAME_OF_NODE>/firmware.bin`. If you're having trouble with uploading, you can also try uploading this file directly with other tools.

Step 4: Uploading Firmware

In order to upload the firmware, you're first going to need to get the chip into a flash mode, otherwise the device will start up without accepting any firmware flash attempts. To do this, while the device is UART bridge is not connected to your USB port, start pressing the small push button in the middle of the PCB. Then plug in the UART bridge into your computer and just keep holding the button pressed for 2-4 seconds. The S20 should now be in a flash mode and should not blink with any LED.

Now you can finally run the upload command:

```
esphomeyaml sonoff_s20.yaml run
```

If successful, you should see something like this:

```

Uploading .pioenvs/sonoff_s20/firmware.bin
Uploading 344320 bytes from .pioenvs/sonoff_s20/firmware.bin to flash at 0x00000000
..... [ 23% ]
..... [ 47% ]
..... [ 71% ]
..... [ 94% ]
..... [ 100% ]
===== [SUCCESS] Took 40.87 seconds ==
=====
INFO [esphomeyaml.__main__] Successfully uploaded program.
--- Miniterm on /dev/cu.SLAB_USBtoUART 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
[I][log_component:056]: Log initialized
[C][ota:163]: There have been 0 suspected unsuccessful boot attempts.
[I][application:041]: Application::setup()

```

Hooray ! You've now successfully uploaded the first esphomeyaml firmware to your Sonoff S20. And in a moment, you will be able to use all of esphomeyaml's great features with your Sonoff S20.

If above step does, however, not work, here are some steps that can help:

- Sometimes the UART bridge cannot supply enough current to the chip to operate, in this case use a 3.3V supply you have lying around. A nice hack is to use the power supply of NodeMCU boards. Simply connect 3.3V to VCC and GND to GND on the pins. **Do not attempt to plug the device into a socket to overcome this problem while troubleshooting.**
- In other cases the TX and RX pin are reversed. Simple disconnect the device, swap the two pins and put it into flash mode again.

Step 5: Adding the Button, Relay and LEDs

Now we would like the S20 to actually do something, not just connect to WiFi and pretty much sit idle.

Below you will find a table of all usable GPIO pins of the S20 and a configuration file that exposes all of the basic functions.

GPIO0	Push Button (HIGH = off, LOW = on)
GPIO12	Relay and its status LED
GPIO13	Green LED (HIGH = off, LOW = on)
GPIO1	RX pin (for external sensors)
GPIO3	TX pin (for external sensors)

```
esphomeyaml:  
  name: <NAME_OF_NODE>  
  platform: ESP8266  
  board: esp01_1m  
  board_flash_mode: dout  
  
  wifi:  
    ssid: <YOUR_SSID>  
    password: <YOUR_PASSWORD>  
  
  mqtt:  
    broker: <YOUR_MQTT_BROKER>  
    username: <YOUR_USERNAME>  
    password: <YOUR_PASSWORD>  
  
  logger:  
  
  ota:  
  
  binary_sensor:  
    - platform: gpio  
      pin:  
        number: GPIO0  
        mode: INPUT_PULLUP  
        inverted: True  
        name: "Sonoff S20 Button"  
    - platform: status  
      name: "Sonoff S20 Status"
```

(continues on next page)

(continued from previous page)

```

switch:
- platform: gpio
  name: "Sonoff S20 Relay"
  pin: GPIO12

output:
# Register the green LED as a dimmable output ....
- platform: esp8266_pwm
  id: s20_green_led
  pin: GPIO13
  inverted: True

light:
# ... and then make a light out of it.
- platform: monochromatic
  name: "Sonoff S20 Green LED"
  output: s20_green_led

```

Above example also showcases an important concept of esphomeyaml: IDs and linking. In order to make all components in esphomeyaml as much “plug and play” as possible, you can use IDs to define them in one area, and simply pass that ID later on. For example, above you can see an PWM (dimmer) output being created with the ID `s20_green_led` for the green LED. Later on it is then transformed into a *monochromatic light*.

And if you want the thing that’s connected through the output of the S20 to appear as a light in Home Assistant, replace the last part with this:

```

switch:
- platform: restart
  name: "Sonoff S20 Restart"

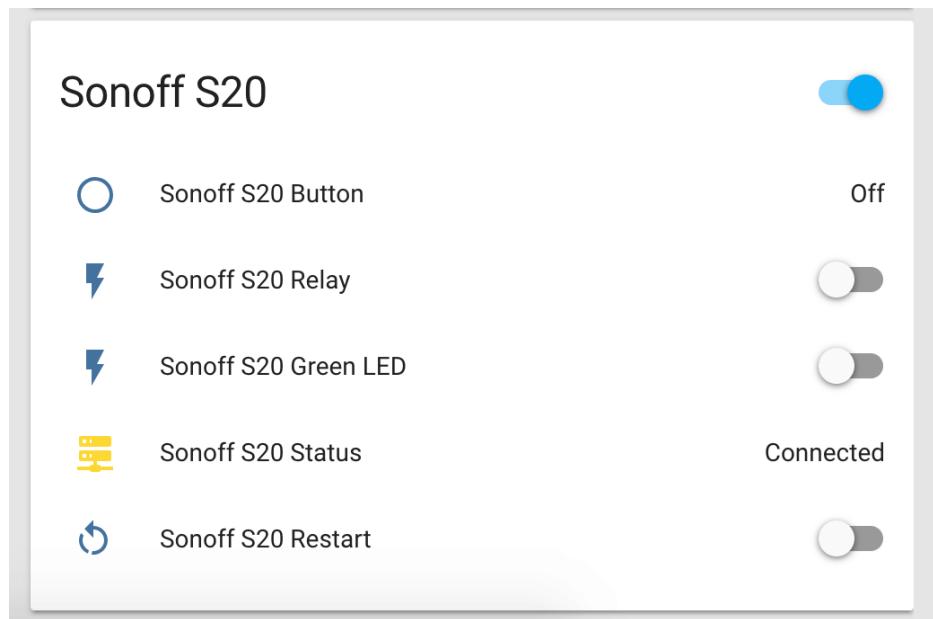
output:
- platform: esp8266_pwm
  id: s20_green_led
  pin:
    number: GPIO13
  inverted: True
# Note: do *not* make the relay a dimmable (PWM) signal, relays cannot handle that
- platform: binary
  id: s20_relay
  pin: GPIO12

light:
- platform: monochromatic
  name: "Sonoff S20 Green LED"
  output: s20_green_led
- platform: binary
  name: "Sonoff S20 Relay"
  output: s20_relay

```

To make pressing the button on the front toggle the relay, have a look at the [the complete Sonoff S20 with automation example](#).

Upload the firmware again (through OTA or Serial) and you should immediately see something like this in Home Assistant because of esphomeyaml’s automatic MQTT discovery. (You’ll of course have to add them to groups if you have a `default_view` set):



Step 6: Finishing Up

Now you're pretty much done with setting up the Sonoff S20. The only steps left are to remove any cables within the housing and make sure everything in there is clean. If, for example, you used wires to connect the UART console, you should definitely remove them to avoid a short with mains.

Sometimes the soldered-on header pins can also interfere with the button. It's best to remove the header pins again, as you will hopefully not need to use them again because of esphomeyaml's Over-The-Air Update features (+ the OTA safe mode; if your node reboots more than 10 times in a row, it will automatically enter an OTA-only safe mode).

If you're sure everything is done with the S20 and have double checked there's nothing that could cause a short in the case, you can put the front cover with the button on the base again and screw everything together.

Now triple or even quadruple check the UART bridge is not connected to the S20, then comes the time when you can plug it into the socket.

Happy hacking!

See Also

- [Generic Sonoff](#)
- [Using With Sonoff 4CH](#)
- [Edit this page on GitHub](#)

Using With Sonoff Basic

esphomeyaml can be used with Sonoff Basic. These devices are basically just an ESP8266 chip with a relay to control the connection, a small button on the front and a green LED light.



Fig. 34: Sonoff Basic

This guide will not guide you through setting up your Sonoff Basic step-by-step. It just provide a sample configuration. For detailed instructions, see [Using With Sonoff S20](#), as those devices are pretty similar.

Note: If you've previously installed Sonoff-Tasmota or ESPPurna on your Sonoff Basic, you're in luck esphomeyaml can generate a firmware binary which you can then upload via the web interface.

Sample configuration

The Sonoff Basic is based on the `ESP8266` platform and is a subtype of the `esp01_1m` board. With this information, you can also step through the esphomeyaml wizard (`esphomeyaml sonoff_basic.yaml wizard`) if you don't want to use the sample configuration file from below.

If you go through the wizard, please make sure you manually set `board_flash_mode` to `dout` as seen below. The version of the uploader used by esphomeyaml should automatically detect that the Sonoff Basic uses the `dout` SPI flash chip mode. But, as some users of other firmwares have said that other flash modes can brick the device, it's always good to specify it explicitly.

Below you will find a table of all usable GPIO pins of the Sonoff Basic and a configuration file that exposes all of the basic functions.

<code>GPIO0</code>	Button (inverted)
<code>GPIO1</code>	RX pin (for external sensors)
<code>GPIO3</code>	TX pin (for external sensors)
<code>GPIO4</code>	Optional Sensor
<code>GPIO12</code>	Relay and Red LED
<code>GPIO13</code>	Green LED (inverted)
<code>GPIO14</code>	Optional Sensor
<code>GPIO17</code>	Analog Input

```
esphomeyaml:
  name: <NAME_OF_NODE>
```

(continues on next page)

(continued from previous page)

```
platform: ESP8266
board: esp01_1m
board_flash_mode: dout

wifi:
  ssid: <YOUR_SSID>
  password: <YOUR_PASSWORD>

mqtt:
  broker: <YOUR_MQTT_BROKER>
  username: <YOUR_USERNAME>
  password: <YOUR_PASSWORD>

logger:

ota:

binary_sensor:
  - platform: gpio
    pin:
      number: GPIO00
      mode: INPUT_PULLUP
      inverted: True
      name: "Sonoff Basic Button"
  - platform: status
    name: "Sonoff Basic Status"

switch:
  - platform: gpio
    name: "Sonoff Basic Relay"
    pin: GPIO12

output:
  - platform: esp8266_pwm
    id: basic_green_led
    pin: GPIO13
    inverted: True

light:
  - platform: monochromatic
    name: "Sonoff Basic Green LED"
    output: basic_green_led
```

Now run `esphomeyaml sonoff_basic.yaml` compile to validate the configuration and pre-compile the firmware.

Note: After this step, you will be able to find the compiled binary under `<NAME_OF_NODE>/pioenvs/<NAME_OF_NODE>/firmware.bin`. If you're having trouble with uploading, you can also try uploading this file directly with other tools.

Or run the upload command if your device is connected to the serial interface:

```
esphomeyaml sonoff_basic.yaml run
```

See Also

- [Generic Sonoff](#)
- [Using With Sonoff 4CH](#)
- [Using With Sonoff S20](#)
- [Edit this page on GitHub](#)

Guides

Automations And Templates

Automations and templates are two very powerful concepts of esphomelib/yaml. Automations allow you to perform actions under certain conditions and templates are a way to easily customize everything about your node without having to dive into the full esphomelib C++ API.

Let's begin with an example to explain these concepts. Suppose you have this configuration file:

```
switch:
- platform: gpio
  pin: GPIO3
  name: "Living Room Dehumidifier"

binary_sensor:
- platform: gpio
  pin: GPIO4
  name: "Living Room Dehumidifier Toggle Button"
```

With this file you can already perform some basic tasks. You can control the ON/OFF state of the dehumidifier in your livingroom from Home Assistant's front-end. But in many cases, controlling everything strictly from the frontend is quite a pain. That's why you have decided to also install a simple push button next to the dehumidifier on pin GPIO4. A simple push on this button should toggle the state of the dehumidifier.

You *could* write an automation to do this task in Home Assistant's automation engine, but ideally the IoT should work without an internet connection and should not break without the MQTT server being online.

That's why, starting with esphomelib 1.7.0, there's a new automation engine. With it, you can write some basic (and also some more advanced) automations using a syntax that is hopefully a bit easier to read and understand than Home Assistant's.

For example, this configuration would achieve your desired behavior:

```
switch:
- platform: gpio
  pin: GPIO3
  name: "Living Room Dehumidifier"
  id: dehumidifier1

binary_sensor:
- platform: gpio
  pin: GPIO4
  name: "Living Room Dehumidifier Toggle Button"
  on_press:
    then:
      - switch.toggle:
          id: dehumidifier1
```

Woah, hold on there. Please explain what's going on here! Sure :) Let's step through what's happening here.

```
switch:  
  - platform: gpio  
    # ...  
    id: dehumidifier1
```

First, we have to give the dehumidifier an *ID* so that we can later use it inside our awesome automation.

```
binary_sensor:  
  - platform: gpio  
    # ...  
    on_press:
```

We now attach a special attribute `on_press` to the toggle button. This part is called a “trigger”. In this example, the automation in the next few lines will execute whenever someone *begins* to press the button. Note the terminology follows what you would call these events on mouse buttons. A *press* happens when you begin pressing the button/mouse. There are also other triggers like `on_release`, `on_click` or `on_double_click` available.

```
# ...  
on_press:  
  then:  
    - switch.toggle:  
      id: dehumidifier1
```

Actions

Now comes the actual automation block. With `then`, you tell esphomeyaml what should happen when the press happens. Within this block, you can define several “actions”. For example, `switch.toggle` and the line after that form an action. Each action is separated by a dash and multiple actions can be executed in series by just adding another – like so:

```
# ...  
on_press:  
  then:  
    - switch.toggle:  
      id: dehumidifier1  
    - delay: 2s  
    - switch.toggle:  
      id: dehumidifier1
```

With this automation, a press on the push button would cause the dehumidifier to turn on/off for 2 seconds, and then cycle back to its original state. Similarly you can have a single trigger with multiple automations:

```
# ...  
on_press:  
  - then:  
    - switch.toggle:  
      id: dehumidifier1  
  - then:  
    - light.toggle:  
      id: dehumidifier_indicator_light
```

(continues on next page)

(continued from previous page)

```
# Same as:
on_press:
  then:
    - switch.toggle:
        id: dehumidifier1
    - light.toggle:
        id: dehumidifier_indicator_light
```

As a last example, let's make our dehumidifier smart: Let's make it turn on automatically when the humidity a sensor reports is above 65% and make it turn off again when it reaches 50%

```
sensor:
  - platform: dht
    humidity:
      name: "Living Room Humidity"
      on_value_range:
        - above: 65.0
          then:
            - switch.turn_on:
                id: dehumidifier1
        - below: 50.0
          then:
            - switch.turn_off:
                id: dehumidifier1
    temperature:
      name: "Living Room Temperature"
```

That's a lot of indentation `on_value_range` is a special trigger for sensors that triggers when the value output of the sensor is within a certain range. In the first example, this range is defined as “any value above or including 65.0”, and the second one refers to once the humidity reaches 50% or below.

Now that concludes the introduction into automations in esphomeyaml. They're a powerful tool to automate almost everything on your device with an easy-to-use syntax. For the cases where the “pure” YAML automations don't work, esphomelib has another extremely powerful tool to offer: Templates.

Templates (Lambdas)

With templates inside esphomelib, you can do almost *everything*. If for example you want to only perform a certain automation if a certain complex formula evaluates to true, you can do that with templates. Let's look at an example first:

```
binary_sensor:
  - platform: gpio
    name: "Cover End Stop"
    id: top_end_stop
cover:
  - platform: template
    name: Living Room Cover
    lambda: !lambda >-
      if (id(top_end_stop).value) {
        return cover::COVER_OPEN;
      } else {
        return cover::COVER_CLOSED;
    }
```

What's happening here? First, we define a binary sensor (with the id `top_end_stop`) and then a *template cover*. The *state* of the template cover is controlled by a template, or "lambda". In lambdas you're effectively writing C++ code and therefore the name lambda is used instead of Home Assistant's "template" lingo to avoid confusion. But before you go shy away from using lambdas because you just hear C++ and think oh noes, I'm not going down *that* road: Writing lambdas is not that hard! Here's a bit of a primer:

First, you might have already wondered what the `lambda: !lambda >-` part is supposed to mean. `!lambda` tells esphomeyaml that the following block is supposed to be interpreted as a lambda, or C++ code. Note that here, the `lambda:` key would actually implicitly make the following block a lambda so in this context, you could have just written `lambda: >-`.

Next, there's the weird `>-` character combination. This effectively tells the YAML parser to treat the following **indented** (!) block as plaintext. Without it, the YAML parser would attempt to read the following block as if it were made up of YAML keys like `cover:` for example. (You may also have seen variations of this like `|-` or just `|` or `>`. There's a slight difference in how these different styles deal with whitespace, but for our purposes we can ignore that).

With `if (...) { ... } else { ... }` we create a *condition*. What this effectively says is that if the thing inside the first parentheses evaluates to `true` then execute the first block (in this case `return cover::COVER_OPEN;`, or else evaluate the second block. `return ...;` makes the code block give back a value to the template. In this case, we're either *returning* `cover::COVER_OPEN` or `cover::COVER_CLOSED` to indicate that the cover is closed or open.

Finally, `id(...)` is a helper function that makes esphomeyaml fetch an object with the supplied ID (which you defined somewhere else, like `top_end_stop`) and let's you call any of esphomelib's many APIs directly. For example, here we're retrieving the current state of the end stop using `.value` and using it to construct our cover state.

Note: esphomeyaml (currently) does not check the validity of lambda expressions you enter and will blindly copy them into the generated C++ code. If compilation fails or something else is not working as expected with lambdas, it's always best to look at the generated C++ source file under `<NODE_NAME>/src/main.cpp`.

Bonus: Templating Actions

Another feature of esphomeyaml is that you can template almost every parameter for actions in automations. For example if you have a light and want to set it to a pre-defined color when a button is pressed, you can do this:

```
on_press:
  then:
    - light.turn_on:
        id: some_light_id
        transition_length: 0.5s
        red: 0.8
        green: 1.0
        blue: !lambda >-
          # The sensor outputs values from 0 to 100. The blue
          # part of the light color will be determined by the sensor value.
          return id(some_sensor).value / 100.0;
```

Every parameter in actions that has the label "templatable" in the docs can be templated like above, using all of the usual lambda syntax.

All Triggers

- `mqtt.on_message`
- `sensor.on_value`
- `sensor.on_value_range`
- `sensor.on_raw_value`
- `binary_sensor.on_press`
- `binary_sensor.on_release`
- `binary_sensor.on_click`
- `binary_sensor.on_double_click`

All Actions

- `delay`
- `lambda`
- `mqtt.publish`
- `switch.toggle`
- `switch.turn_off`
- `switch.turn_on`
- `light.toggle`
- `light.turn_off`
- `light.turn_on`
- `cover.open`
- `cover.close`
- `cover.stop`
- `fan.toggle`
- `fan.turn_off`
- `fan.turn_on`

Delay Action

This action delays the execution of the next action in the action list by a specified time period.

```
on_...:
  then:
    - switch.turn_on:
        id: relay_1
    - delay: 2s
    - switch.turn_off:
        id: relay_1
    # Templated, waits for 1s (1000ms) only if a reed switch is active
    - delay: !lambda "if (id(reed_switch).value) return 1000; else return 0;"
```

Note: This is a “smart” asynchronous delay - other code will still run in the background while the delay is happening.

Lambda Action

This action executes an arbitrary piece of C++ code (see [Lambda](#)).

```
on_...:
  then:
    - lambda: >-
      id(some_binary_sensor).publish_state(false);
```

See Also

- [Configuration Types](#)
- [Frequently Asked Questions](#)
- [Edit this page on GitHub](#)

Configuration Types

esphomeyaml’s configuration files have several configuration types. This page describes them.

ID

Quite an important aspect of esphomeyaml are “ids”. They are used to connect components from different domains. For example, you define an output component together with an id and then later specify that same id in the light component. IDs should always be unique within a configuration and esphomeyaml will warn you if you try to use the same ID twice.

Because esphomeyaml converts your configuration into C++ code and the ids are in reality just C++ variable names, they must also adhere to C++’s naming conventions. [C++ Variable names ...](#)

- ... must start with a letter and can end with numbers.
- ... must not have a space in the name.
- ... can not have special characters except the underscore (“_”).
- ... must not be a keyword.

Pin

esphomeyaml always uses the **chip-internal GPIO numbers**. These internal numbers are always integers like 16 and can be prefixed by GPIO. For example to use the pin with the **internal** GPIO number 16, you could type `GPIO16` or just `16`.

Most boards however have aliases for certain pins. For example the NodeMCU ESP8266 uses pin names D0 through D8 as aliases for the internal GPIO pin numbers. Each board (defined in [esphomeyaml section](#)) has their own aliases and so not all of them are supported yet. For example, for the D0 (as printed on the PCB

silkscreen) pin on the NodeMCU ESP8266 has the internal GPIO name `GPIO16`, but also has an alias `D0`. So using either one of these names in your configuration will lead to the same result.

```
some_config_option:
  pin: GPIO16

some_config_option:
  # alias on the NodeMCU ESP8266:
  pin: D0
```

Pin Schema

In some places, esphomeyaml also supports a more advanced “pin schema”.

```
some_config_option:
  # Basic:
  pin: D0

  # Advanced:
  pin:
    number: D0
    inverted: True
    mode: INPUT_PULLUP
```

Configuration variables:

- **number** (**Required**, pin): The pin number.
- **inverted** (*Optional*, boolean): If all read and written values should be treated as inverted. Defaults to `False`.
- **mode** (*Optional*, string): A pin mode to set for the pin at startup, corresponds to Arduino’s `pinMode` call.

Available Pin Modes:

- `INPUT`
- `OUTPUT`
- `OUTPUT_OPEN_DRAIN`
- `ANALOG` (only on ESP32)
- `INPUT_PULLUP`
- `INPUT_PULLDOWN` (only on ESP32)
- `INPUT_PULLDOWN_16` (only on ESP8266 and only on `GPIO16`)

More exotic Pin Modes are also supported, but rarely used:

- `WAKEUP_PULLUP` (only on ESP8266)
- `WAKEUP_PULLDOWN` (only on ESP8266)
- `SPECIAL`
- `FUNCTION_0` (only on ESP8266)
- `FUNCTION_1`
- `FUNCTION_2`

- FUNCTION_3
- FUNCTION_4
- FUNCTION_5 (only on ESP32)
- FUNCTION_6 (only on ESP32)

Time

In lots of places in esphomeyaml you need to define time periods. There are several ways of doing this. See below examples to see how you can specify time periods:

```
some_config_option:  
    some_time_option: 1000us # 1000 microseconds = 1ms  
    some_time_option: 1000ms # 1000 milliseconds  
    some_time_option: 1.5s # 1.5 seconds  
    some_time_option: 0.5min # half a minute  
    some_time_option: 2h # 2 hours  
    some_time_option: 2:01 # 2 hours 1 minute  
    some_time_option: 2:01:30 # 2 hours 1 minute 30 seconds  
    # 10ms + 30s + 25min + 3h  
    some_time_option:  
        milliseconds: 10  
        seconds: 30  
        minutes: 25  
        hours: 3  
        days: 0
```

See Also

- [esphomeyaml index](#)
- [Getting Started with esphomeyaml](#)
- [Frequently Asked Questions](#)
- [Edit this page on GitHub](#)

Frequently Asked Questions

Tips for using esphomeyaml

1. esphomeyaml supports (most of) Home Assistant's YAML configuration directives like `!include`, `!secret`. So you can store all your secret WiFi passwords and so on in a file called `secrets.yaml` within the directory where the configuration file is.
2. If you want to see how esphomeyaml interprets your configuration, run

```
esphomeyaml livingroom.yaml config
```

3. To view the logs from your node without uploading, run

```
esphomeyaml livingroom.yaml logs
```

4. If you have changed the name of the node and want to update over-the-air, just specify `--upload-port` when running `esphomeyaml`. For example:

```
esphomeyaml livingroom.yaml run --upload-port 192.168.178.52
```

5. You can always find the source `esphomeyaml` generates under `<NODE_NAME>/src/main.cpp`. It's even possible to edit anything outside of the `AUTO GENERATED CODE BEGIN/END` lines for creating *custom sensors*.

What's the difference between esphomelib and esphomeyaml?

`esphomelib` is a C++ framework around Arduino for creating custom firmwares for ESP8266/ESP32 devices. So with `esphomelib`, you need to write C++ code.

`esphomeyaml` is a tool, written in python, that creates source code that uses the `esphomelib` framework. It does this by parsing in a YAML file and generating a C++ source file, compiling it and uploading the binary to the device. It is meant to be a powerful yet user-friendly engine for creating custom firmwares for ESP8266/ESP32 devices. Ideally, it should enable users to use a single command to do everything they want to do with their device without messing around with build systems and so on.

The nice part of the `esphomelib/esphomeyaml` combo is that you can easily edit the source code `esphomeyaml` generates and insert your own custom components such as sensors in it. So, if for example a sensor you really want to use, is not supported, you can easily [create a custom component](#) for it.

Because `esphomeyaml` runs on a host with lots of resources (as opposed to the ESP node itself), `esphomeyaml` will in the future also be able to do some really powerful stuff. I have some ideas like having an automatic schematic creator or a simple `blockly-like` in mind that will hopefully make the user-experience of using ESP32/ESP8266 nodes a lot easier.

Help! Something's not working!!

That's no good. Here are some steps that resolve some problems:

- **Update platformio** Some errors are caused by `platformio` not having the latest version. Try running `platformio update` in your terminal.
- **Clean the platformio cache:** Sometimes the build cache leaves behind some weird artifacts. Try running `platformio run -d <NAME_OF_NODE> -t clean`.
- **Try with the latest Arduino framework version:** See [this](#).
- **Still an error?** Please file a bug report over in the `esphomelib` issue tracker. I will take a look at it as soon as I can. Thanks!

How to submit an issue report

First of all, thank you very much to everybody submitting issue reports! While I try to test `esphomelib/yaml` as much as I can using my own hardware, I don't own every single device type and mostly only do tests with my own home automation system. When doing some changes in the core, it can quickly happen that something somewhere breaks. Issue reports are a great way for me to track and (hopefully) fix issues, so thank you!

For me to fix the issue the quickest, there are some things that would be really helpful:

1. How do you use `esphomelib`? Through `esphomeyaml` or directly through C++ code?

2. If it's a build/upload issue: What system are you compiling/uploading things from? Windows, POSIX, from docker?
3. A snippet of the code/configuration file used is always great for a better understanding of the issue.
4. If it's an i2c or hardware communication issue please also try setting the *log level* to `VERY_VERBOSE` as it provides helpful information about what is going on.

You can find the issue tracker here <https://github.com/OttoWinter/esphomelib/issues>

How do I update to the latest version?

It's simple. Run:

```
pip2 install -U esphomeyaml
# From docker:
docker pull ottowinter/esphomeyaml:latest
```

And in HassIO, there's a simple UPDATE button when there's an update available as with all add-ons

How do I use the latest bleeding edge version?

First, a fair warning that the latest bleeding edge version is not always stable and might have issues. If you find some, please do however report them if you have time :)

Installing the latest bleeding edge version of esphomelib is also quite easy. It's most often required if there was a bug somewhere and I didn't feel like building & pushing a whole new release out (this often takes up to 2 hours!). To install the dev version of esphomeyaml:

- In HassIO: In the esphomeyaml add-on repository there's also a second add-on called `esphomeyaml-edge`. Install that and stop the stable version (both can't run at the same time without port collisions).
- From pip: Run `pip install git+git://github.com/OttoWinter/esphomeyaml.git`
- From docker: Run `docker pull ottowinter/esphomeyaml:dev` and use `ottowinter/esphomeyaml:dev` in all commands.

Next, if you want to use the latest version of esphomelib too:

```
# Sample configuration entry
esphomeyaml:
  name: ...
  library_uri: 'https://github.com/OttoWinter/esphomelib.git'
  # ...
```

In some cases it's also a good idea to use the latest Arduino framework version. See [this](#).

Does esphomelib support [this device/feature]?

If it's not in [the docs](#), it's probably sadly not supported. However, I'm always trying to add support for new features, so feel free to create a feature request in the [esphomelib issue tracker](#). Thanks!

I have a question... How can I contact you?

Sure! I'd be happy to help :) You can contact me here:

- Discord
- Home Assistant Community Forums
- esphomelib and esphomeyaml issue trackers. Preferably only for issues and feature requests.
- Alternatively, also under my e-mail address contact (at) otto-winter.com

My node keeps reconnecting randomly

Jep, that's a known issue. However, it seems to be very low-level and I don't really know how to solve it. I'm working on possible work-arounds for the issue but currently I do not have a real solution.

Some steps that can help with the issue:

- Use the most recent version of th arduino framework. The platformio arduino package always takes some time to update and the most recent version often includes some awesome patches. See [Using the latest Arduino framework version](#).
- The issue seems to be happen with cheap boards more frequently. Especially the "cheap" NodeMCU boards from eBay sometimes have quite bad antennas.
- Play around with the `keepalive` option of the [MQTT client](#), sometimes increasing this value helps (because it's causing more pings in the background), some other times a higher keepalive works better.

Devices that will (hopefully) be supported soon:

Devices/Sensors that I've bought and will be supported at some point (ordered by priority):

- 433MHz Transmitter Component
- PN532 NFC Board
- INA219/INA3221 Current Sensor
- GP2Y10 Dust Sensor
- TCS34725 RGB Light Sensor
- APDS-9960 RGB Gesture Sensor
- MCP2301 16-Channel I/O Expander
- MH-Z19 CO² Sensor
- HMC5883L Compass Sensor
- SPI E-Ink Display Module
- 8-Segment Display
- I²C LCD Display (4 rows, 20 characters)
- I²C/SPI SSD1306 OLED Display
- Nextion TFT LCD Display
- MLX90614 Infrared Thermometer
- MS5611 Pressure Sensor

- PCF8591 ADC
- OV2640 Camera
- L298N H-Bridge Motor Driver
- A4988 Stepper Motor Driver
- MQ-2 Gas Sensor

Other features that I'm working on:

- ESP32 IR/433MHz Receiver
- Pulse Counter for the ESP8266 (using interrupts)
- Multiple WiFi Networks to connect to
- Improve “Restart due to WiFi/MQTT disconnect” logic and make the timeouts more configurable
- Color Temperature for Lights
- Status LED
- More FastLED effects
- Support for displays (like LCD/OLED/E-Ink displays)
- Cameras (probably through ArduCAM)

Devices which are supported, but not tested yet. I'm still waiting for these to arrive from China:

- ADS1115 Voltage Sensor
- TSL2561 Brightness Sensor
- HDC1080 Temperature+Humidity Sensor
- SHT31-D Temperature+Humidity Sensor
- BME280
- BME680

Devices that are technically already supported, but for which guides will be created soon-ish:

- Sonoff Basic
- Sonoff Basic RF
- Soil Moisture Sensor

Anything missing? I'd be happy to chat about more integrations over on the [discord channel](#) - no guarantees that everything will be supported though!

See Also

- [*esphomeyaml index*](#)
- [*Contributing*](#)
- [Edit this page on GitHub](#)

Getting Started with esphomeyaml

esphomeyaml is the perfect solution for creating custom firmwares for your ESP8266/ESP32 boards. In this guide we'll go through how to setup a basic “node” in a few simple steps.

Installation

Installing esphomeyaml is very easy. All you need to do is have [Python 2.7](#) installed (because of platformio) and install the console script through pip.

```
pip install esphomeyaml
```

Alternatively, there's also a docker image available for easy installation (the docker hub image is only available for amd64 right now, if you have an RPi, please install esphomelib through pip or use [the HassIO add-on](#):

```
docker pull ottowinter/esphomeyaml
```

Creating A Project

Now let's setup a configuration file. Fortunately, esphomeyaml has a friendly setup wizard that will guide you through creating your first configuration file. For example, if you want to create a configuration file called `livingroom.yaml`:

```
esphomeyaml livingroom.yaml wizard
# On Docker:
docker run --rm -v "`pwd`:/config -it ottowinter/esphomeyaml livingroom.yaml wizard
```

Note: If you have trouble with installing esphomeyaml and see errors like “command not found”, try running `python2 -m esphomeyaml livingroom.yaml wizard`.

At the end of this step, you will have your first YAML configuration file ready. It doesn't do much yet and only makes your device connect to the WiFi network and MQTT broker, but still it's a first step.

Adding some features

So now you should have a file called `livingroom.yaml` (or similar). Go open that file in an editor of your choice and let's add a [simple GPIO switch](#) to our app.

```
switch:
- platform: gpio
  name: "Living Room Dehumidifier"
  pin: 5
```

The configuration format should hopefully immediately seem similar to you. esphomeyaml has tried to keep it as close to Home Assistant's `configuration.yaml` schema as possible. In above example, we're simply adding a switch that's called “Living Room Relay” (could control anything really, for example lights) and is connected to pin `GPIO5`. The nice thing about esphomeyaml is that it will automatically also try to translate pin numbers for you based on the board. For example in above configuration, if using a NodeMCU board, you could have just as well set `D1` as the `pin:` option.

First Uploading

Now you can go ahead and add some more components. Once you feel like you have something you want to upload to your ESP board, simply plug in the device via USB and type the following command (replacing `livingroom.yaml` with your configuration file):

```
esphomeyaml livingroom.yaml run
```

You should see `esphomeyaml` validating the configuration and telling you about potential problems. Then `esphomeyaml` will proceed to compile and upload the custom firmware. You will also see that `esphomeyaml` created a new folder with the name of your node. This is a new platformio project that you can modify afterwards and play around with.

On docker, the first upload is a bit more complicated, either you manage to map the serial device into docker with the `-v` option, or you just call `compile` within the container and let platformio do the uploading on the host system:

```
docker run --rm -v "$(pwd)":/config -it ottowinter/esphomeyaml livingroom.yaml compile  
platformio run -d livingroom -t upload
```

Now if you have [MQTT Discovery](#) enabled in your Home Assistant configuration, the switch should already be automatically be added (Make sure you've added it to a view too.)



Living Room Dehumidifier



After the first upload, you will probably never need to use the USB cable again, as all features of esphomelib are enabled remotely as well. No more opening hidden boxes stowed in places hard to reach. Yay!

Adding A Binary Sensor

Next, we're going to add a very simple binary sensor that periodically checks a GPIO pin whether it's pulled high or low - the [GPIO Binary Sensor](#).

```
binary_sensor:  
  - platform: gpio  
    name: "Living Room Window"  
    pin:  
      number: 16  
      inverted: True  
      mode: INPUT_PULLUP
```

This is an advanced feature of `esphomeyaml`, almost all pins can optionally have a more complicated configuration schema with options for inversion and `pinMode` - the [Pin Schema](#).

This time when uploading, you don't need to have the device plugged in through USB again. The upload will magically happen “over the air”. Using `esphomeyaml` directly, this is the same as from a USB cable, but for docker you need to supply an additional parameter:

```
esphomeyaml livingroom.yaml run
# On docker
docker run --rm -p 6123:6123 -v "`pwd`:/config" -it ottowinter/esphomeyaml livingroom.yaml run
```



Living Room Window

Off

Where To Go Next

Great ! You've now successfully setup your first esphomeyaml project and uploaded your first esphomelib custom firmware to your node. You've also learned how to enable some basic components via the configuration file.

So now is a great time to go take a look at the [Components Index](#), hopefully you'll find all sensors/outputs/... you'll need in there. If you're having any problems or want new features, please either create a new issue on the [GitHub issue tracker](#) or contact me via the [Discord chat](#).

Bonus: esphomeyaml dashboard

Starting with version 1.6.0 esphomeyaml features a dashboard that you can use to easily manage your nodes from a nice web interface. It was primarily designed for [the HassIO add-on](#), but also works with a simple command.

To start the esphomeyaml dashboard, simply start esphomeyaml with the following command (with `config/` pointing to a directory where you want to store your configurations)

```
# Install dashboard dependencies
pip2 install tornado esptool
esphomeyaml config/ dashboard

# On docker
docker run --rm -p 6052:6052 -p 6123:6123 -v "`pwd`:/config" -it ottowinter/esphomeyaml /config
  ↵ dashboard
```

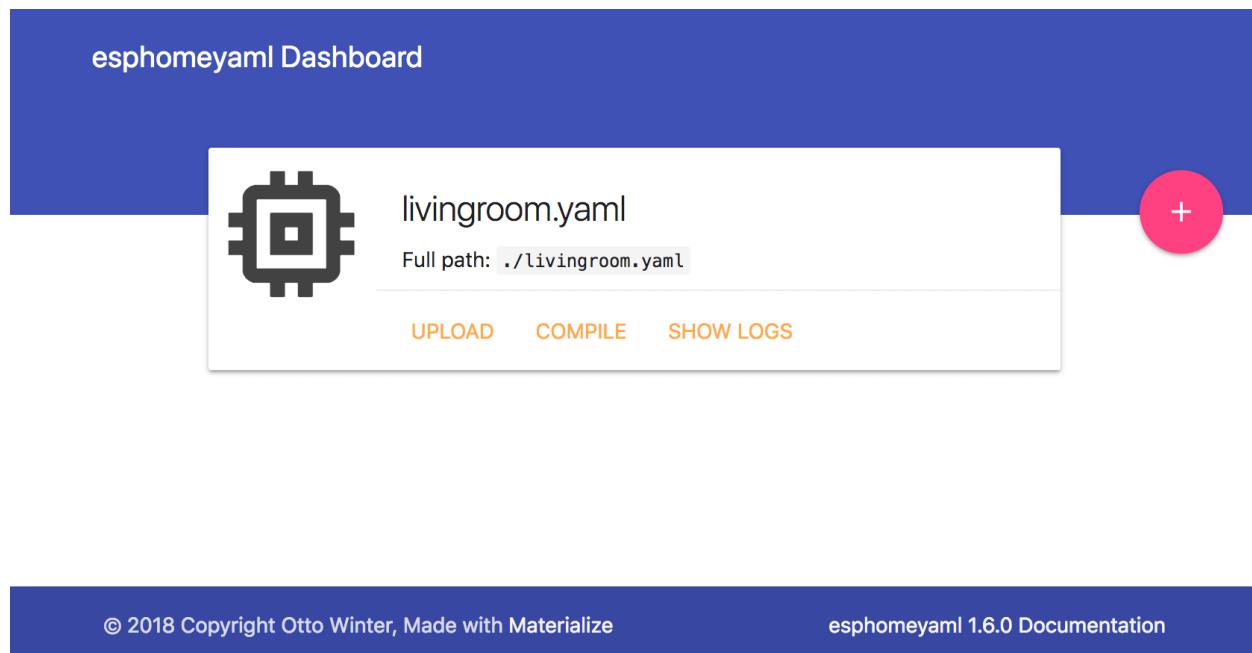
After that, you will be able to access the dashboard through `localhost:6052`.

Using Custom components

esphomelib's powerful core makes it easy to create own custom sensors. Please first follow the [Custom Sensor Component Guide](#) to see how this can be done. For using custom components with esphomeyaml you only need to open up the auto-generated `src/main.cpp` file in the platformio project folder. The lines in between `AUTO GENERATED CODE BEGIN` and `AUTO GENERATED CODE END` should not be edited and all changes in there will be overridden, but outside of those comments you can safely create custom sensors while still using esphomeyaml's great configuration options.

```
// Auto generated code by esphomeyaml
#include "esphomelib/application.h"
```

(continues on next page)



© 2018 Copyright Otto Winter, Made with Materialize

esphomeyaml 1.6.0 Documentation

(continued from previous page)

```
using namespace esphomelib;

void setup() {
    // ===== DO NOT EDIT ANYTHING BELOW THIS LINE =====
    // ===== AUTO GENERATED CODE BEGIN =====
    App.set_name("cabinet");
    // ...
    // ===== AUTO GENERATED CODE END =====
    // ===== YOU CAN EDIT AFTER THIS LINE =====
    App.setup();
}

void loop() {
    App.loop();
    delay(20);
}
```

See Also

- [esphomeyaml index](#)
- [Getting Started with esphomeyaml through HassIO](#)
- [Edit this page on GitHub](#)

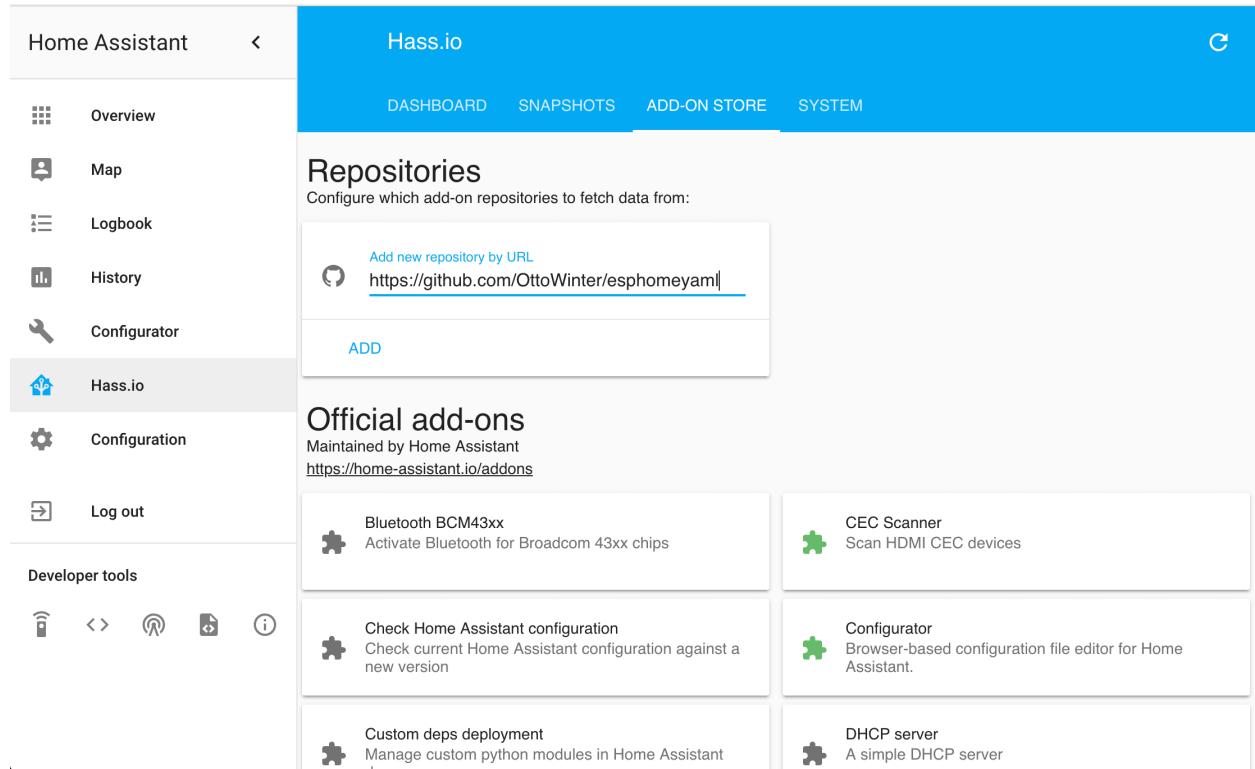
Getting Started with esphomeyaml through HassIO

esphomeyaml is the perfect solution for creating custom firmwares for your ESP8266/ESP32 boards. In this guide we'll go through how to setup a basic “node” by use of the HassIO add-on.

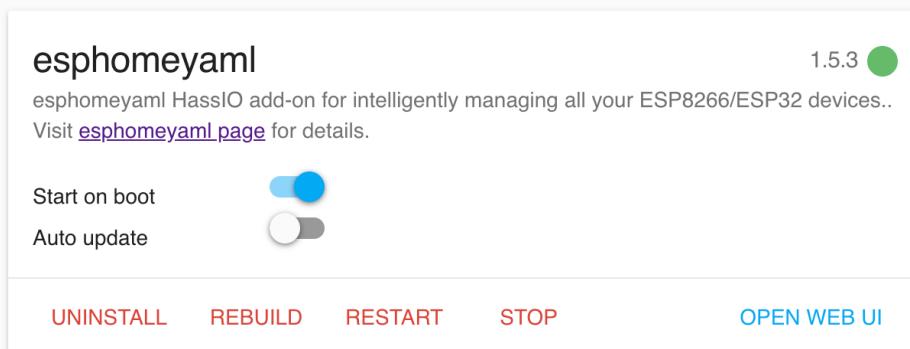
But first, here's a very quick introduction of how esphomeyaml works: esphomeyaml is a *tool* which reads in YAML configuration files (just like Home Assistant) and creates custom firmwares. The tool also has many helpers that simplify flashing devices and aims to make managing your ESP boards as simple as possible. Once you have added devices or sensors in esphomeyaml's configuration, they will even automatically show up in Home Assistant's UI.

Installation

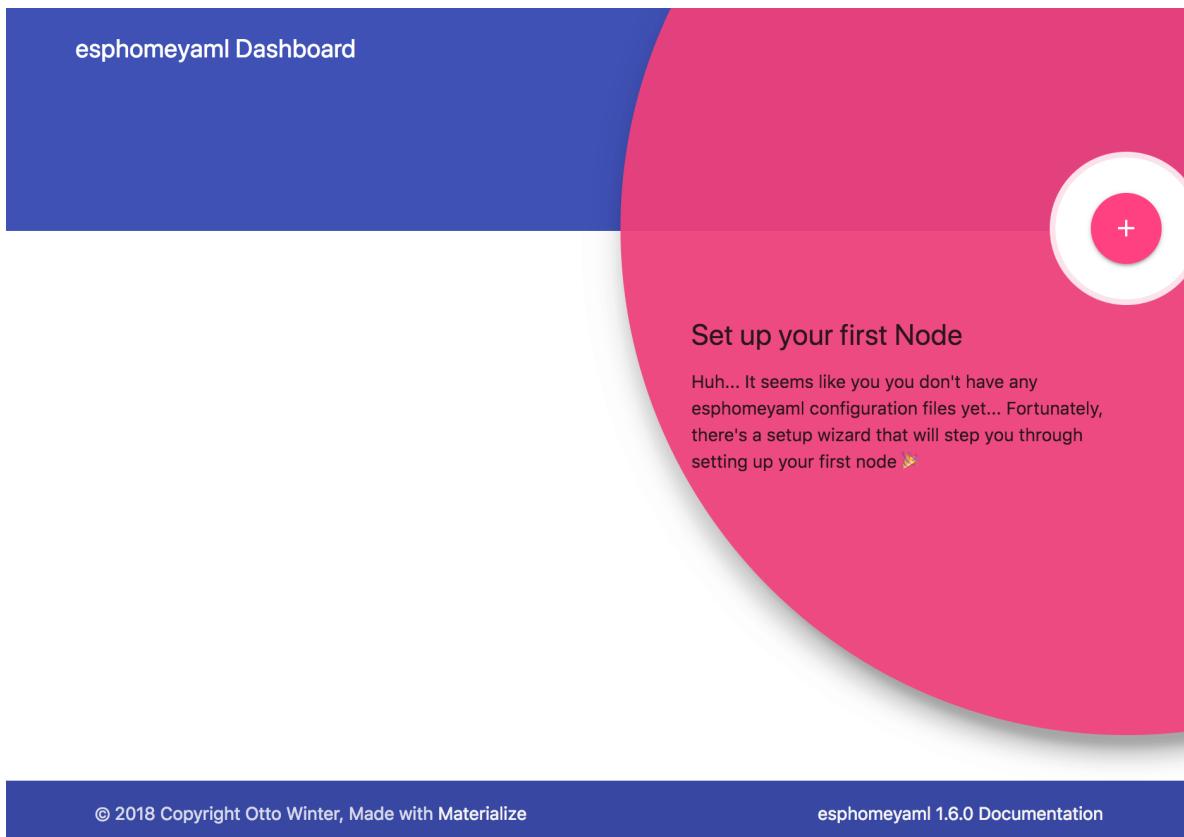
Installing the esphomeyaml HassIO add-on is easy. Just navigate to the HassIO panel in your Home Assistant frontend and add the esphomeyaml add-on repository: <https://github.com/OttoWinter/esphomeyaml>



After that, wait a bit until the add-on is installed (this can take a while) and go to the add-on page. Start the add-on and then click “Open Web UI”.



You should now be greeted by a nice introduction wizard which will step you through creating your first configuration.



© 2018 Copyright Otto Winter, Made with Materialize

esphomeyaml 1.6.0 Documentation

Note: If the UI isn't showing up correctly, it's probably because your browser isn't supported. Please try using Google Chrome in that case.

Dashboard Interface

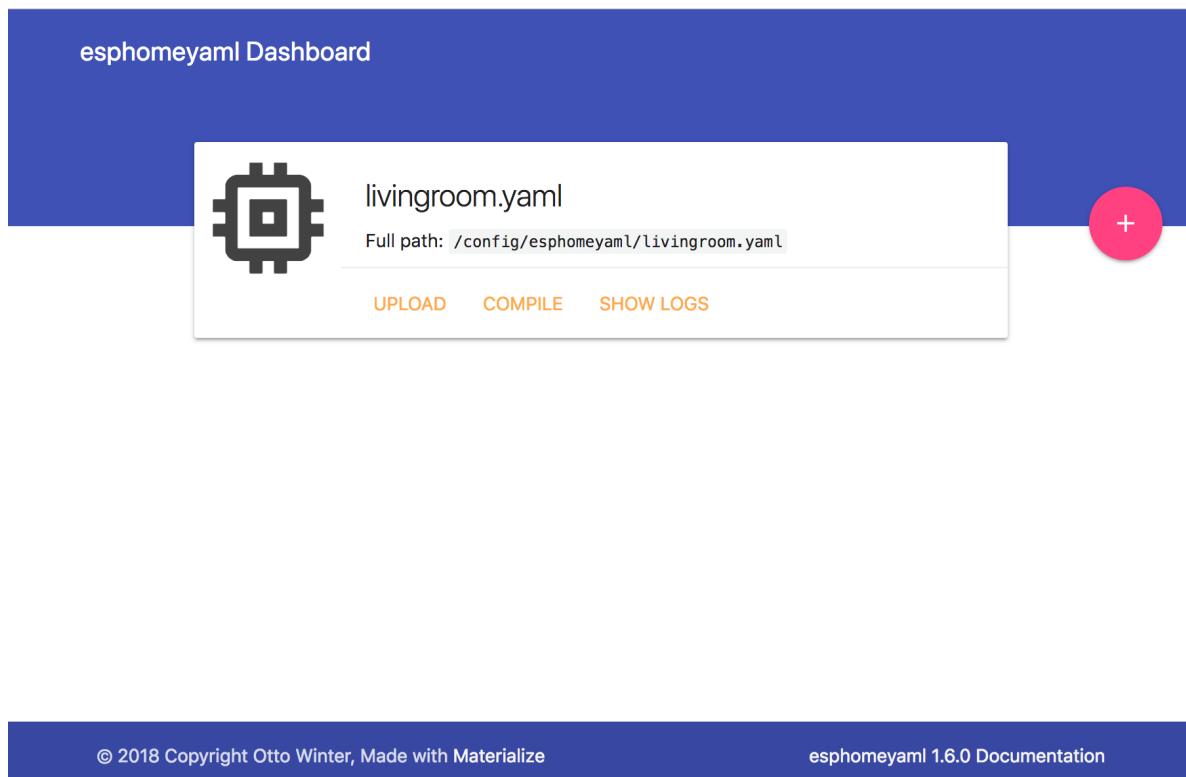
Assuming you created your first configuration file with the wizard, let's take a quick tour of the esphomeyaml dashboard interface.

On the front page you will see all configurations for nodes you created. For each file, there are three basic actions you can perform:

- **UPLOAD:** This compiles the firmware for your node and uploads it using any connected USB device or, if no USB devices are connected, over-the-air using the *OTA Update Component*.

Warning: The HassIO Add-On is currently not capable of discovering new USB ports after the add-on has started due to some docker restrictions. Please go to the add-on details page and restart the add-on if a new USB device is not automatically found.

- **COMPILE:** This command compiles the firmware and gives you the option of downloading the generated binary so that you can upload it yourself from your computer.
- **SHOW LOGS:** With this command you can view all the logs the node is outputting. If a USB device is connected, it will attempt to use the serial connection. Otherwise it will use the built-in MQTT



© 2018 Copyright Otto Winter, Made with Materialize

esphomeyaml 1.6.0 Documentation

logs.

The configuration files for esphomeyaml can be found and edited under `<HOME_ASSISTANT_CONFIG>/esphomeyaml/`. For example the configuration for the node in above picture can be found in `/config/esphomeyaml/livingroom.yaml`.

Tip: Use the awesome [HASS Configurator Add-On](#) to edit your esphomeyaml configuration files.

Now go ahead and use one of the [devices guides](#) to extend your configuration for the device you intend to flash an esphomeyaml firmware onto. Then proceed with uploading the first firmware using the upload button.

Note: Currently the build toolchain for the ESP32 does not work on RPis. If you need to compile software for ESP32 (**not** ESP8266) boards, please install esphomeyaml on your computer.

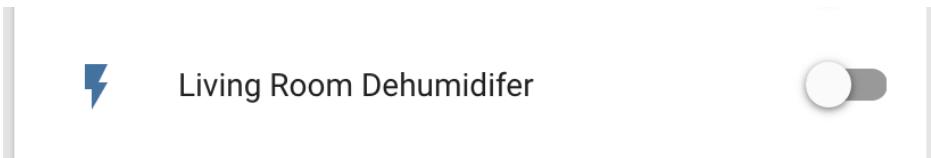
Adding some (basic) features

So now you should have a file called `/config/esphomeyaml/livingroom.yaml` (or similar). Go open that file in and add a [simple GPIO switch](#) to the configuration like this:

```
switch:
- platform: gpio
  name: "Living Room Dehumidifier"
  pin: 5
```

In above example, we're simply adding a switch that's called "Living Room Relay" (could control anything really, for example lights) and is connected to the pin GPIO5.

Now if you have [MQTT Discovery](#) enabled in your Home Assistant configuration, the switch should already be automatically be added (Make sure you've added it to a view too.)



After the first upload, you will probably never need to use the USB cable again, as all features of esphomelib are enabled remotely as well. No more opening hidden boxes stowed in places hard to reach. Yay!

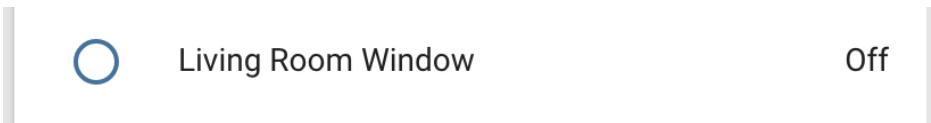
Adding A Binary Sensor

Next, we're going to add a very simple binary sensor that periodically checks a GPIO pin whether it's pulled high or low - the [GPIO Binary Sensor](#).

```
binary_sensor:  
  - platform: gpio  
    name: "Living Room Window"  
    pin:  
      number: 16  
      inverted: True  
      mode: INPUT_PULLUP
```

This is an advanced feature of esphomeyaml, almost all pins can optionally have a more complicated configuration schema with options for inversion and pinMode - the [Pin Schema](#).

This time when uploading, you don't need to have the device plugged in through USB again. The upload will magically happen "[over the air](#)".



Where To Go Next

Great ! You've now successfully setup your first esphomeyaml project and uploaded your first esphomelib custom firmware to your node. You've also learned how to enable some basic components via the configuration file.

So now is a great time to go take a look at the [Components Index](#), hopefully you'll find all sensors/outputs/... you'll need in there. If you're having any problems or want new features, please either create a new issue on the [GitHub issue tracker](#) or contact me via the [Discord chat](#).

Using Custom components

esphomelib's powerful core makes it easy to create own custom sensors. Please first follow the [Custom Sensor Component Guide](#) to see how this can be done. For using custom components with esphomeyaml you only

need to open up the auto-generated <NODE_NAME>/src/main.cpp file in the platformio project folder. The lines in between AUTO GENERATED CODE BEGIN and AUTO GENERATED CODE END should not be edited and all changes in there will be overridden, but outside of those comments you can safely create custom sensors while still using esphomeyaml's great configuration options.

```
// Auto generated code by esphomeyaml
#include "esphomelib/application.h"

using namespace esphomelib;

void setup() {
    // ===== DO NOT EDIT ANYTHING BELOW THIS LINE =====
    // ===== AUTO GENERATED CODE BEGIN =====
    App.set_name("cabinet");
    // ...
    // ===== AUTO GENERATED CODE END =====
    // ===== YOU CAN EDIT AFTER THIS LINE =====
    App.setup();
}

void loop() {
    App.loop();
    delay(20);
}
```

See Also

- [esphomeyaml index](#)
- [Getting Started with esphomeyaml](#)
- [Edit this page on GitHub](#)

Migrating from ESPEasy

Migrating from previous ESPEasy setups is very easy. You just need to have esphomeyaml create a binary for you and then upload that in the ESPEasy web interface.

Getting Binary

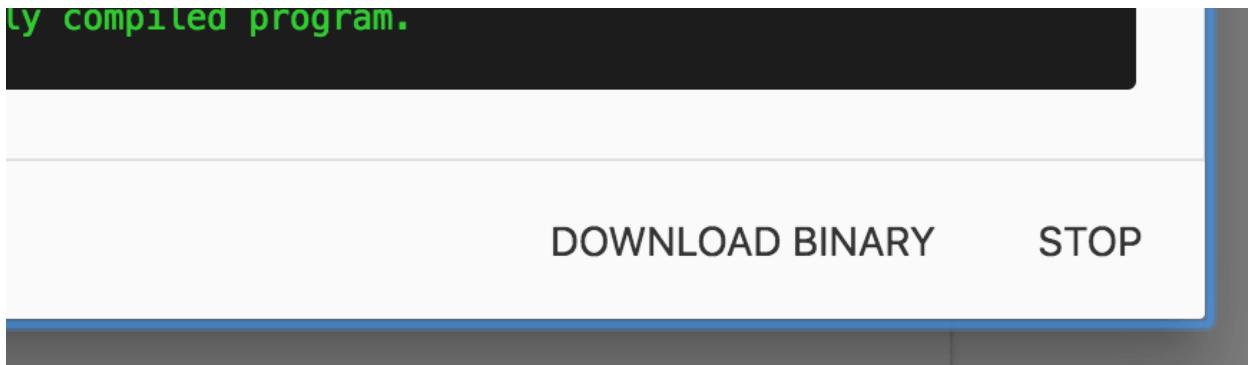
First follow the guides for the [different supported devices](#) and create a configuration file. Then, generate and download the binary:

- **Using the HassIO add-on/dashboard:** Just click the COMPILE button, wait for the compilation to end and press the DOWNLOAD BINARY button.
- **Using the command line:** run `esphomeyaml livingroom.yaml compile` (replacing `livingroom.yaml` with your configuration file of course) and navigate to the <NODE_NAME>/.pioenvs/<NODE_NAME>/ folder. There you will find a `firmware.bin` file, this is the binary you will upload.

Uploading Binary

To upload the binary, navigate to the ESPEasy web interface and enter the “Tools” section.

ly compiled program.



Welcome to ESP Easy: livingroom

Main Config Hardware Devices Tools

The Tools section of the ESP Easy interface. It features a navigation bar with tabs: System, Reboot, Log, and Advanced. Under System, there are buttons for Wifi (with Connect, Disconnect, Scan), Interfaces (with I2C Scan), Settings (with Load and Save), and Firmware (with Load and a question mark). A Command input field with a Submit button is also present.

Tools	
System	Reboot Log Advanced
Wifi	Connect Disconnect Scan
Interfaces	I2C Scan
Settings	Load Save
Firmware	Load ?
Command	<input type="text"/>
Submit	

Press “Load” under Firmware, then select the binary you previously downloaded and upload the binary. If everything succeeds, you should now have esphomelib on your node .

Note: With esphomelib, you in most cases won’t need to worry about the available flash size, as the binary only ever includes the code that you are actually using.

Happy Hacking!

See Also

- [NodeMCU ESP8266](#)
- [NodeMCU ESP32](#)
- [Generic ESP8266](#)
- [Generic ESP32](#)
- [Using With Sonoff S20](#)
- [Using With Sonoff 4CH](#)
- [Migrating from ESPurna](#)
- [Migrating from Sonoff Tasmota](#)
- [Edit this page on GitHub](#)

Migrating from ESPurna

Migrating from previous ESPurna setups is very easy. You just need to have esphomeyaml create a binary for you and then upload that in the ESPurna web interface.

Getting Binary

First follow the guides for the [different supported devices](#) and create a configuration file. Then, generate and download the binary:

- **Using the HassIO add-on/dashboard:** Just click the COMPILE button, wait for the compilation to end and press the DOWNLOAD BINARY button.

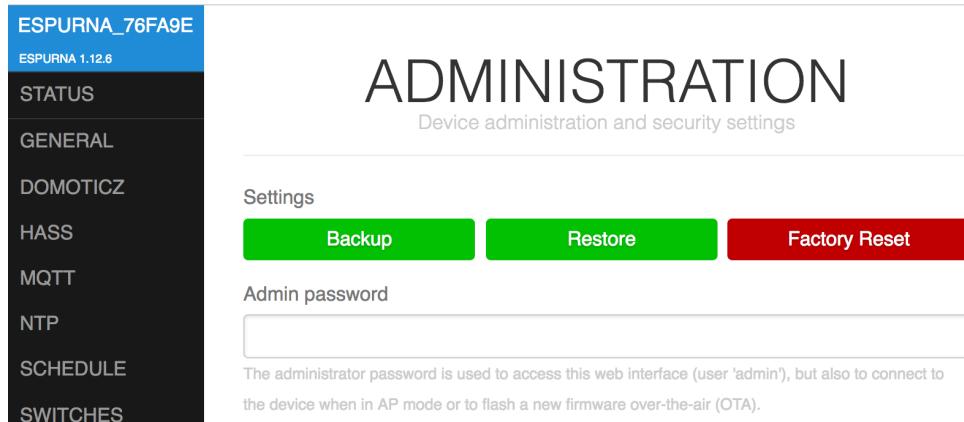
ly compiled program.

DOWNLOAD BINARY STOP

- **Using the command line:** run `esphomeyaml livingroom.yaml compile` (replacing `livingroom.yaml` with your configuration file of course) and navigate to the `<NODE_NAME>/pioenvs/<NODE_NAME>/` folder. There you will find a `firmware.bin` file, this is the binary you will upload.

Uploading Binary

To upload the binary, navigate to the ESPurna web interface and enter the “General” section.



In the “Upgrade” section, choose the binary you previously downloaded and press “Upgrade”. If everything succeeds, you should now have esphomelib on your node

Note: With esphomelib, you in most cases won’t need to worry about the available flash size, as the binary only ever includes the code that you are actually using.

Upgrade

livingroom.bin

The device has 2670592 bytes available for OTA updates. If your image is larger than this consider doing a [two-step update](#).

Happy Hacking!

See Also

- [NodeMCU ESP8266](#)
- [NodeMCU ESP32](#)
- [Generic ESP8266](#)
- [Generic ESP32](#)
- [Using With Sonoff S20](#)
- [Using With Sonoff 4CH](#)
- [Migrating from ESPEasy](#)

- *Migrating from Sonoff Tasmota*
- Edit this page on GitHub

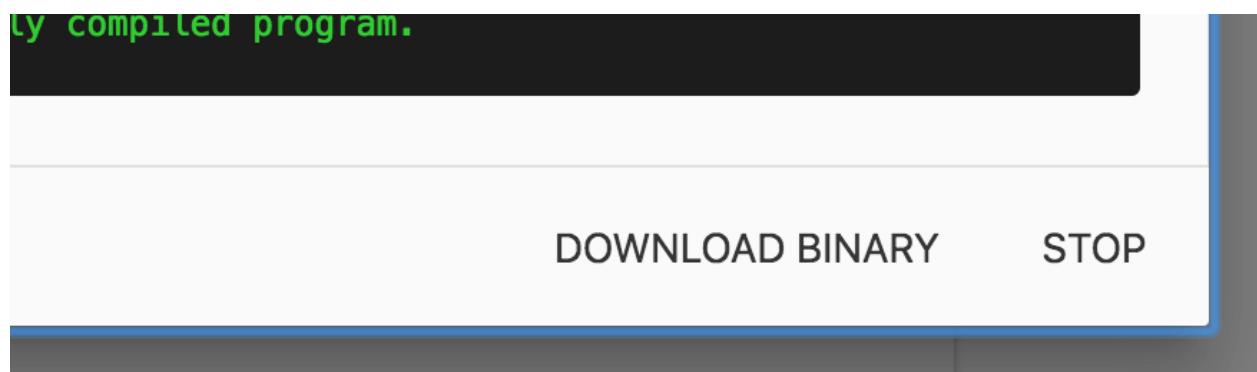
Migrating from Sonoff Tasmota

Migrating from previous Sonoff Tasmota setups is very easy. You just need to have esphomeyaml create a binary for you and then upload that in the Tasmota web interface.

Getting Binary

First follow the guides for the *different supported devices* and create a configuration file. Then, generate and download the binary:

- **Using the HassIO add-on/dashboard:** Just click the COMPILE button, wait for the compilation to end and press the DOWNLOAD BINARY button.



- **Using the command line:** run `esphomeyaml livingroom.yaml compile` (replacing `livingroom.yaml` with your configuration file of course) and navigate to the `<NODE_NAME>/pioenvs/<NODE_NAME>/` folder. There you will find a `firmware.bin` file, this is the binary you will upload.

Uploading Binary

To upload the binary, navigate to the tasmota web interface and enter the “Firmware Upgrade” section.

In the “Upgrade by file upload” section, choose the binary you previously downloaded

If everything succeeds, you will see an “Upload Successful” message and esphomelib will connect to the configured WiFi network

Happy Hacking!

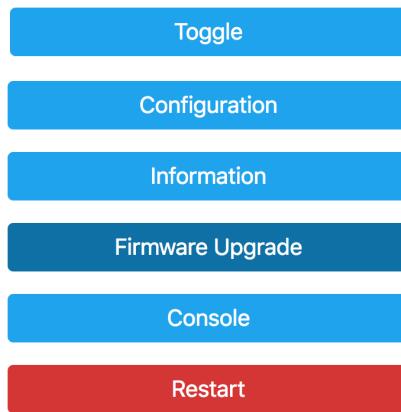
See Also

- *Using With Sonoff S20*
- *Using With Sonoff 4CH*
- *Generic Sonoff*
- *NodeMCU ESP8266*

Sonoff Basic Module

Sonoff

OFF



Sonoff Basic Module

Sonoff

Upgrade by web server

OTA Url
<http://sonoff.maddox.co.uk/tasmota/sonoff.bin>

Start upgrade

Upgrade by file upload

No file chosen

Start upgrade

Main Menu

Sonoff Basic Module

Sonoff

Upload Successful

Device will restart in a few seconds

Main Menu

- [NodeMCU ESP32](#)
- [Generic ESP8266](#)
- [Generic ESP32](#)
- [Migrating from ESPurna](#)
- [Migrating from ESPEasy](#)
- [Edit this page on GitHub](#)

Contributing

Contributions to the esphomelib suite are very welcome! All the code for the projects is hosted on GitHub and you can find the sources here:

- [esphomelib](#) (The C++ framework)
- [esphomeyaml](#) (The Python YAML to C++ transpiler)
- [esphomedocs](#) (The documentation which you're reading here)

Just clone the repository locally, do the changes for your new feature/bugfix and submit a pull request. I will try to take a look at your PR as soon as possible.

Contributing to esphomedocs

One of the areas of esphomelib that can always be improved is the documentation. If you see an issue somewhere, or spelling mistakes or if you want to share your awesome setup, please feel free to submit a pull request.

The esphomelib documentation is built using [sphinx](#) and uses [reStructuredText](#) for all source files.

In my opinion, markdown would have been the much better choice in hindsight, but at the time I was setting up the documentation good doxygen integration was key to me. Anyway, here's a quick RST primer:

- **Headers:** You can write titles like this:

```
My Title  
=====
```

and sub-titles like this:

```
My Sub Title  
~~~~~
```

- **Links:** To create a link to an external resource (for example <https://www.google.com>), use `Link text <link_url>`__. For example:

```
`Google.com <https://www.google.com>`__
```

[Google.com](https://www.google.com)

- **References:** To reference another document, use the :doc: and :ref: roles (references are set up globally and can be used between documents):

```
.. _my-reference-label:  
  
Section to cross-reference  
~~~~~  
  
See :ref:`my-reference-label`, also see :doc:`/esphomeyaml/components/switch/gpio`.  
:doc:`Using custom text </esphomeyaml/components/switch/gpio>`.
```

See [Devices](#), also see [GPIO Switch](#). [Using custom text](#).

- **Inline code:** To have text appear like this, use double backticks:

```
To have text appear ``like this``, use double backticks.
```

To have text appear like this, use double backticks.

- **Code blocks:** To show a sample configuration file, use the code directive:

```
.. code:: yaml  
  
# Sample configuration entry  
switch:  
  - platform: gpio  
    name: "Relay #42"  
    pin: GPIO13
```

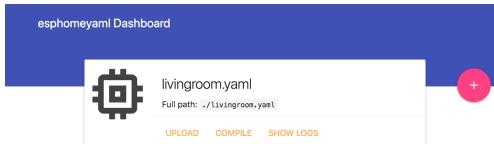
```
# Sample configuration entry  
switch:  
  - platform: gpio  
    name: "Relay #42"  
    pin: GPIO13
```

Note: The YAML syntax highlighter is currently broken. Somehow sphinx thinks this should be C++ code . If you know how to fix this, it would be very much appreciated

- **Images:** To show images, use the figure directive:

```
.. figure:: images/dashboard.png
    :align: center
    :width: 40.0%
```

Optional figure caption.



© 2018 Copyright Otto Winter, Made with Materialize esphomeyaml 1.6.0 Documentation

Fig. 35: Optional figure caption.

- **Notes and warnings:** You can create simple notes and warnings using the `note` and `warning` directives:

```
.. note::

    This is a note.

.. warning::

    This is a warning.
```

Note: This is a note.

Warning: This is a warning.

- **Italic and boldface font families:** To *italicize* text, use one asterisk around the text. To put a **strong emphasis** on a piece of text, put two asterisks around it.

```
*This is italicized.* (A weird word...)
**This is very important.**
```

This is italicized. (A weird word...) **This is very important.**

- **Ordered and unordered list:** The syntax for lists in RST is more or less the same as in markdown:

```
- Unordered Item
  - Unordered Sub-Item
  - Item with a very long text so that it does not fully fit in a single line and
    must be split up into multiple lines.

1. Ordered Item #1
2. Ordered Item #2
```

– Unordered Item

- * Unordered Sub-Item
- Item with a very long text so that it does not fully fit in a single line and must be split up into multiple lines.
- 1. Ordered Item #1
- 2. Ordered Item #2

reStructured text can do a lot more than this, so if you're looking for a more complete guide please have a look at the [Sphinx reStructuredText Primer](#).

To check your documentation changes locally, you first need install sphinx (**with Python 3**) and [doxygen](#).

```
pip3 install sphinx breathe
```

Next, you will also need to clone the [esphomelib](#) repository into the folder where `esphomedocs` sits like this:

```
esphomedocs/
    api/
    esphomeyaml/
    Doxygen
    Makefile
    index.rst
    ...
esphomelib/
    src/
    examples/
    library.json
    platformio.ini
    ...
```

Then, use the provided Makefile to build the changes and start a simple web server:

```
# Update doxygen API docs
make doxyg
# Start web server on port 8000
make webserver

# Updates then happen via:
make html
```

Some notes about the docs:

- Use the english language (duh...)
- An image tells a thousand words, please use them wherever possible. But also don't forget to shrink them, for example I often use <https://tinypng.com/>
- Try to use examples as often as possible (also while it's great to use highly accurate, and domain-specific lingo, it should not interfere with new users understanding the content)
- When adding new files, please also add them to the `index.rst` file in the directory you're editing.
- Fixes/improvements for the docs themselves should go to the `current` branch of the `esphomedocs` repository. New features should be added against the `next` branch.

Contributing to esphomelib

esphomelib is the engine behind all the esphomeyaml stuff. The framework is also designed to be used on its own - i.e. without esphomeyaml. To contribute code to esphomelib to fix a bug or add a new integration/feature, clone the repository, make your changes and create a pull request.

At some point, I will create a dedicated guide for the exact setup used, but for now just look around the code base a bit and see how other components are doing stuff.

To initialize the development environment, navigate to the repository and execute:

```
# View available IDEs:  
pio init --help  
# Initialize for IDE  
pio init --ide {YOUR_IDE}
```

Standard for the esphomelib codebase:

- All features should at least have a bit of documentation using the doxygen documentation style (see other source files for reference)
- The code style is based on the [Google C++ Style Guide](#) with a few modifications:
 - function, method and variable names are `lower_snake_case`
 - class/struct/enum names should be `UpperCamelCase`
 - constants should be `UPPER_SNAKE_CASE`
 - fields should be `protected` and `lowe_snake_case_with_trailing_underscore_`.
 - It's preferred to use long variable/function names over short and non-descriptive ones.
- Use two spaces, not tabs.
- Using `#define`s is discouraged and should be replaced by constants.
- Use `using type_t = int;` instead of `typedef int type_t;`
- Be careful with including large standard library headers, they can considerably increase the code size.
- All features should only be compiled if a user explicitly defined so using `-DUSE_<FEATURE>` (see `esphomeyaml/defines.h`)
- Header files `.h` should not include source code. All code should sit in C++ `.cpp` files. (except for templates)
- Using explicit int sizes is like `int64_t` is preferred over standard types like `long long`.
- All new features should have at least one example usage in the examples directory.
- New components should dump their configuration using `ESP_LOGCONFIG` at startup in `setup()`
- The number of external libraries should be kept to a minimum. If the component you're developing has a simple communication interface, please consider implementing the library natively in esphomelib.
- Implementations for new devices should contain reference links for the datasheet and other sample implementations.
- Please test your changes :)

Contributing to esphomeyaml

esphomeyaml primarily does two things: It validates the configuration and creates C++ code.

The configuration validation should always be very strict with validating user input - it's always better to fail quickly if a configuration isn't right than to have the user find out the issue after a few hours of debugging.

Preferably, the configuration validation messages should explain the exact validation issue (and not "invalid name!") and try to suggest a possible fix.

The C++ code generation engine is 99% syntactic sugar and unfortunately not too well documented yet. Have a look around other components and you will hopefully quickly get the gist of how to interact with the code generation engine.

The python source code of your component will automatically be loaded if the user uses it in the configuration. Specifically, it may contain these fields:

- **CONFIG_SCHEMA**: for *components* like `dallas`. This is the configuration schema that will be validated against the user configuration.
- **PLATFORM_SCHEMA**: for *platforms* like `sensor.dallas`. This is the configuration schema that will be validated against every `platform`: definition in the config of your platform name.
- **to_code**: The "workhorse" of esphomeyaml. This will be called with the configuration of your component/platform and you can add code to the global code index in here.
 - Call an `Application` method like this `App.make_dallas_component()`
 - Register a variable using `variable(<TYPE>, <VAR_ID>, rhs)`. This will generate an assignment expression and add it to the global expression index. The return value is the left hand side variable which you can use for further calls.

```
<TYPE> <VAR_ID> = <rhs>;
```

- Register a variable of a pointer type using `Pvariable(<TYPE>, <VAR_ID>, rhs)`.

```
<TYPE> *<VAR_ID> = <rhs>;  
  
// rhs = App.make_dallas_component(12, 15000)  
// var = Pvariable(DallasComponent, "dallas_id", rhs)  
// add(var.hello_world())  
DallasComponent *dallas_id = App.make_dallas_component(12, 15000)  
dallas_id->hello_world()
```

- Expressions like `var.hello_world()` are not automatically added to the code and need to be added to the global expression index using `add()`.
- Access variables using `get_variable()`. The variable will automatically know if it is a pointer and use the correct operator. Additionally, you can pass a type as the second argument to `get_variable`. This will cause esphomeyaml to use the first variable of that type.

```
hub = get_variable(config.get(CONF_DALLAS_ID), DallasComponent)
```

- Pass configuration arguments to mock function calls (like `App.make_dallas_component`) using normal python :)

```
rhs = App.make_dallas_component(config[CONF_PIN], config.get(CONF_UPDATE_INTERVAL))
```

Note the `config.get()`: Trailing `None` values in function calls are stripped.

- **BUILD_FLAGS**: Pass build flags that should be provided if your component is loaded.

```
BUILD_FLAGS = '-DUSE_DALLAS_SENSOR'
```

- REQUIRED_BUILD_FLAGS: Like BUILD_FLAGS, but also uses these build flags if the user has disabled build flags in the *esphomeyaml* section.
- DEPENDENCIES: Other components that are required to be in the user's configuration if this platform/component is loaded:

```
DEPENDENCIES = ['i2c']
```

- ESP_PLATFORMS: Provide a whitelist of platforms this integration works on. Default is work on all platforms.

```
ESP_PLATFORMS = [ESP_PLATFORM_ESP32]
```

Run `pip2 install -e .` to install a development version of esphomeyaml.

See Also

- *esphomeyaml index*
- *Frequently Asked Questions*
- [Edit this page on GitHub](#)

3.2 Web Server API

Since version 1.3, esphomelib includes a built-in web server that can be used to view states and send commands. In addition to visible the web-frontend available under the root index of the web server, there's also two other features the web server currently offers: A real time event source and REST API.

Note that the web server is only and will only ever be intended to view and edit states. Specifically not something like configuring the node, as that would quickly blow up the required flash and memory size.

First up, to use the web server enable it using `App.init_web_server()` directly from code or using the *Web Server Section* in esphomeyaml. Then, navigate to the front end interface with the IP of the node or alternatively using mDNS with `<name>.local/`. So for example to navigate to the web server of a node called `livingroom`, you would enter `livingroom.local/` in your browser.

While it's currently recommended to use esphomelib directly through Home Assistant, if you want to integrate esphomelib with an external or self-built application you can use two available APIs: the real-time event source API and REST API.

3.2.1 Event Source API

If you want to receive real-time updates for sensor state updates, it's recommended to use the Event Source Web API. With the URL `/events`, you can create an `Event Source` that receives real-time updates of states and the debug log using server-sent events. Event sources are easy to implement in many languages and already have many libraries available. For example `eventsource` for `node.js` and `eventsource` for `python`.

Currently, there are three types of events sent: `ping`, `state` and `log`. The first one is repeatedly sent out to keep the connection alive. `log` events are sent every time a log message is triggered and is used to show the debug log on the index page. `state` is where the real magic happens. All events with this type have a JSON payload that describes the state of a component. Each of these JSON payloads have two mandatory

fields: `id` and `state`. `ID` is the unique identifier of the component and is prefixed with the domain of the component, for example `sensor`. `state` contains a simple text-based representation of the state of the underlying component, for example ON/OFF or 21.4 °C. Several components also have additional fields in this payload, for example lights have a `brightness` attribute.

1711210	state	{"id": "sensor-wrmepumpe_verdampfer", "state": "20.4 °C", "value": 20.3625}	19:00:52.333
1711210	state	{"id": "switch-heizung_neustart", "state": "OFF", "value": false}	19:00:52.333
1711210	state	{"id": "binary_sensor-heizung_status", "state": "ON", "value": true}	19:00:52.333
1718048	log	[0;36m[D][sensor.dht:update:53]: Got Temperature=20.0°C Humidity=52.3%[0m	19:00:59.206
1718052	log	[0;37m[V][sensor.sensor:push_new_value:22]: 'Outside Temperature': Received new value 20.000...	19:00:59.219
1718060	log	[0;37m[V][sensor.sensor:push_new_value:28]: 'Outside Temperature': Filter #0 aborted chain[0m	19:00:59.219
1718069	log	[0;37m[V][sensor.sensor:push_new_value:22]: 'Outside Humidity': Received new value 52.299999[...	19:00:59.220
1718077	log	[0;37m[V][sensor.sensor:push_new_value:28]: 'Outside Humidity': Filter #0 aborted chain[0m	19:00:59.220
1719698	ping		19:01:00.844
1721847	log	[0;37m[V][component:set_timeout:53]: set_timeout(name='0xFE0000031F1EAF28', timeout=750)[0m	19:01:02.994
1721851	log	[0;37m[V][component:set_timeout:53]: set_timeout(name='0xBA0000031F0E5228', timeout=750)[...]	19:01:02.999
1721860	log	[0;37m[V][component:set_timeout:53]: set_timeout(name='0xA40000031F055028', timeout=750)[0m	19:01:02.999
1721868	log	[0;37m[V][component:set_timeout:53]: set_timeout(name='0x790000031EE1DC28', timeout=750)[...]	19:01:02.999

Fig. 36: Example payload of the event source API.

Additionally, each time a client connects to the event source the server sends out all current states so that the client can catch up with reality.

The payloads of these state events are also the same as the payloads of the REST API GET calls. I would recommend just opening the network debug panel of your web browser to see what's sent.

REST API

There's also a simple REST API available which can be used to get and set the current state. All calls to this API follow the URL schema `/<domain>/<id>[/<method>?<param>=<value>]`. The `domain` is the type of the component, for example `sensor` or `light`. `id` refers to the id of the component - this ID is created by taking the name of the component, stripping out all non-alphanumeric characters, making everything lowercase and replacing all spaces by underscores.

By creating a simple GET request for a URL of the form `/<domain>/<id>` you will get a JSON payload describing the current state of the component. This payload is equivalent to the ones sent by the event source API.

To actually *control* the state of a component you need to send a POST request with a `method` like `turn_on`. For example, to turn on a light, you would send a POST request to `/light/livingroom_lights/turn_on`. Some components also optionally accept URL parameters to control some other aspects of a component, for example the brightness of a light.

Sensor

Sensors only support GET requests by sending a request to `/sensor/<id>`. For example sending a GET request to `/sensor/outside_temperature` could yield this payload:

```
{
  "id": "sensor-outside_temperature",
  "state": "19.8 °C",
  "value": 19.76666
}
```

- **id:** The id of the sensor. Prefixed by `sensor-`.
- **state:** The text-based state of the sensor as a string.
- **value:** The floating point (filtered) value of the sensor.

Binary Sensor

Binary sensors have a similar payload and also only support GET requests. For example requesting the current state of a binary sensor using the URL `binary_sensor/living_room_status` could result in following payload:

```
{
  "id": "binary_sensor-living_room_status",
  "state": "ON",
  "value": true
}
```

- **id:** The id of the binary sensor. Prefixed by `binary_sensor-`.
- **state:** The text-based state of the binary sensor as a string.
- **value:** The binary (`true/false`) state of the binary sensor.

Switch

Switches have the exact same properties as a binary sensor in the state reporting aspect, but they additionally support setting states with the `turn_on`, `turn_off` and `toggle` methods.

Each of these is quite self explanatory. Creating a POST request to `/switch/dehumidifier/turn_on` would for example result in the component called “Dehumidifier” to be turned on. The server will respond with a 200 OK HTTP return code if the call succeeded.

Light

Lights support quite a few more complicated options, like brightness or color. But first, to get the state of a light, send a GET request to `/light/<id>`, for example `light/living_room_lights`.

```
{
  "id": "light-living_room_lights",
  "state": "ON",
  "brightness": 255,
  "color": {
    "r": 255,
    "g": 255,
    "b": 255
  },
  "effect": "None",
  "white_value": 255
}
```

- **id:** The id of the light. Prefixed by `light-`.
- **state:** The text-based state of the light as a string.

- **brightness**: The brightness of the light from 0 to 255. Only if the light supports brightness. If **state** is OFF, this can still report values like 255 in order to send the full state.
- **color**: The color of this light, only if it supports color.
 - **r**: The red channel of this light. From 0 to 255.
 - **g**: The green channel of this light. From 0 to 255.
 - **b**: The blue channel of this light. From 0 to 255.
- **effect**: The currently active effect, only if the light supports effects.
- **white_value**: The white value of RGBW lights. From 0 to 255.

Setting light state can happen through three POST method calls: `turn_on`, `turn_off` and `toggle`. Turn on and off have additional URL encoded parameters that can be used to set other properties. For example creating a POST request at `/light/<id>/turn_on?brightness=128&transition=2` will create transition with length 2s to the brightness 128 while retaining the color of the light.

`turn_on` optional URL parameters:

- **brightness**: The brightness of the light, from 0 to 255.
- **r**: The red color channel of the light, from 0 to 255.
- **g**: The green color channel of the light, from 0 to 255.
- **b**: The blue color channel of the light, from 0 to 255.
- **white_value**: The white channel of RGBW lights, from 0 to 255.
- **flash**: Flash the color provided by the other properties for a duration in seconds.
- **transition**: Transition to the specified color values in this duration in seconds.
- **effect**: Set an effect for the light.

`turn_off` optional URL parameters:

- **transition**: Transition to off in this duration in seconds.

Fan

Fans are similar to switches as they can be turned on/off and toggled. In addition, if the underlying fan supports it, fans in the web server also support the speed settings “low”, “medium” and “high” and an oscillation setting. To get the current state of a fan, create a GET request to `/fan/<id>`.

```
{  
    "id": "fan-living_room_fan",  
    "state": "ON",  
    "value": true,  
    "speed": "high",  
    "oscillation": false  
}
```

- **id**: The id of the fan. Prefixed by `fan-`.
- **state**: The text-based state of the fan as a string.
- **value**: The binary (`true/false`) state of the fan.
- **speed**: The speed setting of the fan if it’s supported. Either “off”, “low”, “medium” or “high”.
- **oscillation**: Whether the oscillation setting of the fan is on. Only sent if the fan supports it.

To control the state of the fan, send POST requests to `/fan/<id>/turn_on`, `/fan/<id>/turn_off` and `/fan/<id>/toggle`. Turn on additionally supports these optional parameters:

- **speed**: The new speed setting of the fan. Values as above.
- **oscillation**: The new oscillation setting of the fan. Values as above.

3.3 API Reference

3.3.1 Core

Application

The `Application` class has two objectives: 1. handle all of your `Component`s and 2. provide helper methods to simplify creating a component.

In itself, an Application instance doesn't do and store very much. It mostly only keeps track of all Components (using an internal `std::vector()` containing pointers), so that when the user calls the `setup()` or `loop()` methods, it can relay those calls to the components.

In order for the Application to know about your components, each one of them should be registered using the `register_component()` call. And for subclasses of `MQTTComponent`, `register_mqtt_component()` shall be used.

Apart from the helpers, only one function is really necessary for setting up the Application instance: `set_name()`. The name provided by this is used throughout the code base to construct various strings such as MQTT topics or WiFi hostnames. Additionally, some one-time-setup components such as MQTT or WiFi can be initialized with the `init_` calls. Next, this class provides a bunch of helper methods to create and register components easily using the `make_` calls.

Each component can tell the Application with which *priority* it wishes to be called using the `get_loop_priority()` and `get_setup_priority()` overrides. The Application will then automatically order the components before execution.

API Reference

`class Application`

This is the class that combines all components.

Public Functions

`void Application::set_name(const std::string &name)`

Set the name of the item that is running this app.

Note: This will automatically be converted to lowercase_underscore.

Parameters

- **name**: The name of your app.

`LogComponent *Application::init_log(uint32_t baud_rate = 115200, size_t tx_buffer_size = 512)`

Initialize the logging engine.

Return The created and initialized *LogComponent*.

Parameters

- **baud_rate**: The serial baud rate. Set to 0 to disable UART debugging.
- **tx_buffer_size**: The size of the printf buffer.

`WiFiComponent *Applicationinit_wifi(const std::string &ssid, const std::string &password = "")`

Initialize the WiFi engine in client mode.

Note: for advanced options, such as manual ip, use the return value.

Return The *WiFiComponent*.

Parameters

- **ssid**: The ssid of the network you want to connect to.
- **password**: The password of your network. Leave empty for no password

`WiFiComponent *Applicationinit_wifi()`

Initialize the WiFi engine with no initial mode. Use this if you just want an Access Point.

`OTACOMPONENT *Applicationinit_ota()`

Initialize Over-the-Air updates.

Return The *OTACOMPONENT*. Use this to set advanced settings.

`MQTTClientComponent *Applicationinit_mqtt(const std::string &address, uint16_t port, const std::string &username, const std::string &password)`

Initialize the MQTT client.

Return The *MQTTClient*. Use this to set advanced settings.

Parameters

- **address**: The address of your server.
- **port**: The port of your server.
- **username**: The username.
- **password**: The password. Empty for no password.

`MQTTClientComponent *Applicationinit_mqtt(const std::string &address, const std::string &username, const std::string &password)`

Initialize the MQTT client.

Return The *MQTTClient*. Use this to set advanced settings.

Parameters

- **address**: The address of your server.
- **username**: The username.
- **password**: The password. Empty for no password.

*I2CComponent *Application***init_i2c**(uint8_t *sda_pin* = SDA, uint8_t *scl_pin* = SCL, bool *scan* = false)

Initialize the i2c bus on the provided SDA and SCL pins for use with other components.

SDA/SCL pins default to the values defined by the Arduino framework and are usually GPIO4 and GPIO5 on the ESP8266 (D2 and D1 on NodeMCU). And for the ESP32 it defaults to GPIO21 and GPIO22 for SDA and SCL, respectively.

Parameters

- **sda_pin**: The SDA pin the i2c bus is connected to.
- **scl_pin**: The SCL pin the i2c bus is connected to.
- **scan**: If a scan of connected i2c devices should be done at startup.

*WebServer *Application***init_web_server**(uint16_t *port* = 80)

Initialize the web server.

Note that this will take up quite a bit of flash space and RAM of the node. Especially on ESP8266 boards this can quickly cause memory problems.

Return The *WebServer* object, use this for advanced settings.

Parameters

- **port**: The port of the web server, defaults to 80.

*ESP32BLETracker *Application***make_esp32_ble_tracker**()

Setup an ESP32 BLE Tracker Hub.

Only one of these should be created. Individual sensors can be created using `make_device()` on the return value.

Note that this component is currently not completely stable (probably due to some issues in the esp-idf core that are being worked on). Enabling this integration will increase the created binary size by up to 500kB, can cause seemingly random reboots/resets and only can discover a small number of devices at the moment.

Return An *ESP32BLETracker* instance, use this to create individual trackers as binary sensors.

template <typename T>

Automation<T> **Application***make_automation**(*Trigger*<T> **trigger*)

mqtt::MQTTMessageTrigger **Application***make_mqtt_message_trigger**(const std::string &*topic*, uint8_t *qos* = 0)

*StartupTrigger *Application***make_startup_trigger**()

*ShutdownTrigger *Application***make_shutdown_trigger**()

MQTTBinarySensorComponent **Application***register_binary_sensor**(binary_sensor::BinarySensor **binary_sensor*)

Register a binary sensor and set it up for the front-end.

```
Application::MakeGPIOBinarySensor Applicationmake_gpio_binary_sensor(const std::string  
                      &friendly_name,  
                      const GPIOIn-  
                      putPin &pin,  
                      const std::string  
                      &device_class =  
                      "")
```

Create a simple GPIO binary sensor.

Note: advanced options such as inverted input are available in the return value.

Parameters

- **friendly_name**: The friendly name that should be advertised. Leave empty for no automatic discovery.
- **pin**: The GPIO pin.
- **device_class**: The Home Assistant `device_class`.

```
Application::MakeStatusBinarySensor Applicationmake_status_binary_sensor(const  
                                         std::string  
                                         &friendly_name)
```

Create a simple binary sensor that reports the online/offline state of the node.

Uses the MQTT last will and birth message feature. If the values for these features are custom, you need to override them using the return value of this function.

Return A MQTTBinarySensorComponent. Use this to set custom status messages.

Parameters

- **friendly_name**: The friendly name advertised via MQTT discovery.

```
ESP32TouchComponent *Applicationmake_esp32_touch_component()
```

Setup an ESP32TouchComponent to detect touches on certain pins of the ESP32 using the built-in touch peripheral.

First set up the global hub using this method. Then create individual binary sensors using the `make_touch_pad` function on the return type.

Return The new ESP32TouchComponent. Use this to create the binary sensors.

```
Application::MakeTemplateBinarySensor Applicationmake_template_binary_sensor(const  
                                         std::string  
                                         &name,  
                                         std::function<optional<bool>>  
> &&f
```

```
sensor::MQTTSensorComponent *Applicationregister_sensor(sensor::Sensor *sensor)
```

Register a sensor and create a MQTT Sensor if the MQTT client is set up.

```
Application::MakeDHTSensor Applicationmake_dht_sensor(const std::string &tem-  
                                         perature_friendly_name,  
                                         const std::string &humid-  
                                         ity_friendly_name, const GPI-  
                                         OOutputPin &pin, uint32_t  
                                         update_interval = 15000)
```

Create a DHT sensor component.

Note: This method automatically applies a SlidingWindowMovingAverageFilter.

Return The components. Use this for advanced settings.

Parameters

- **temperature_friendly_name**: The name the temperature sensor should be advertised as. Leave empty for no automatic discovery.
- **humidity_friendly_name**: The name the humidity sensor should be advertised as. Leave empty for no automatic discovery.
- **pin**: The pin the DHT sensor is connected to.
- **update_interval**: The interval (in ms) the sensor should be checked.

```
DallasComponent *Applicationmake_dallas_component(ESPOneWire *one_wire, uint32_t update_interval = 15000)
```

```
DallasComponent *Applicationmake_dallas_component(const GPIOOutputPin &pin, uint32_t update_interval = 15000)
```

```
Application::MakePulseCounterSensor Applicationmake_pulse_counter_sensor(const std::string &friendly_name, uint8_t pin, uint32_t update_interval = 15000)
```

Create an ESP32 Pulse Counter component.

The pulse counter peripheral will automatically all pulses on pin in the background. Every check_interval ms the amount of pulses will be retrieved and the difference to the last value will be reported via MQTT as a sensor.

Return The components. Use this for advanced settings.

Parameters

- **pin**: The pin the pulse counter should count pulses on.
- **friendly_name**: The name the sensor should be advertised as.
- **update_interval**: The interval in ms the sensor should be checked.

```
Application::MakeADCSensor Applicationmake_adc_sensor(const std::string &friendly_name, uint8_t pin, uint32_t update_interval = 15000)
```

Create an ADC Sensor component.

Every check_interval ms, the value from the specified pin (only A0 on ESP8266, 32-39 for ESP32), and converts it into the volt unit. On the ESP32 you can additionally specify a channel attenuation using the return value of this function. pinMode can also be set using the return value.

Return The components. Use this for advanced settings.

Parameters

- **pin**: The pin the ADC should sense on.
- **friendly_name**: The name the sensor should be advertised as.

- `update_interval`: The interval in ms the sensor should be checked.

`ADS1115Component *Applicationmake_ads1115_component(uint8_t address)`
Create an ADS1115 component hub.

From this hub you can then create individual sensors using `get_sensor()`.

Note that you should have i2c setup for this component to work. To setup i2c call `App.init_i2c(SDA_PIN, SCL_PIN);` before `App.setup()`.

Return The `ADS1115Component` hub. Use this to set advanced setting and create the actual sensors.

Parameters

- `address`: The i2c address of the ADS1115. See `ADS1115Component::set_address` for possible values.

Application::MakeBMP085Sensor Applicationmake_bmp085_sensor(const std::string &temperature_friendly_name, const std::string &pressure_friendly_name, uint32_t update_interval = 30000)

Create an BMP085/BMP180/BMP280 i2c temperature+pressure sensor.

Be sure to initialize i2c before calling `App.setup()` in order for this to work. Do so with `App.init_i2c(SDA_PIN, SCL_PIN);`.

Return A `MakeBMP085Component` object, use this to set advanced settings.

Parameters

- `temperature_friendly_name`: The friendly name the temperature should be advertised as.
- `pressure_friendly_name`: The friendly name the pressure should be advertised as.
- `update_interval`: The interval in ms to update the sensor values.

Application::MakeHTU21DSensor Applicationmake_htu21d_sensor(const std::string &temperature_friendly_name, const std::string &humidity_friendly_name, uint32_t update_interval = 15000)

Create a HTU21D i2c-based temperature+humidity highly accurate sensor.

Be sure to initialize i2c before calling `App.setup` in order for this to work. Do so with `App.init_i2c(SDA_PIN, SCL_PIN);`.

Return A `MakeHTU21DSensor`, use this to set advanced settings.

Parameters

- `temperature_friendly_name`: The friendly name the temperature sensor should be advertised as.
- `humidity_friendly_name`: The friendly name the humidity sensor should be advertised as.

- `update_interval`: The interval in ms to update the sensor values.

```
Application::MakeHDC1080Sensor Application make_hdc1080_sensor(const std::string &temperature_friendly_name,
                                                               const std::string &humidity_friendly_name,
                                                               uint32_t update_interval
                                                               = 15000)
```

Create a HDC1080 i2c-based temperature+humidity sensor.

Be sure to initialize i2c before calling `App.setup` in order for this to work. Do so with `App.init_i2c(SDA_PIN, SCL_PIN);`.

Return A `MakeHDC1080Sensor`, use this to set advanced settings.

Parameters

- `temperature_friendly_name`: The friendly name the temperature sensor should be advertised as.
- `humidity_friendly_name`: The friendly name the humidity sensor should be advertised as.
- `update_interval`: The interval in ms to update the sensor values.

```
Application::MakeUltrasonicSensor Application make_ultrasonic_sensor(const std::string
&friendly_name,
const GPIOOutputPin &trigger_pin,
const GPIOInputPin &echo_pin, uint32_t
update_interval
= 5000)
```

Create an Ultrasonic range sensor.

This can for example be an HC-SR04 ultrasonic sensor. It sends out a short ultrasonic wave and listens for an echo. The time between the sending and receiving is then (with some maths) converted to a measurement in meters. You need to specify the trigger pin (where to short pulse will be sent to) and the echo pin (where we're waiting for the echo). Note that in order to not block indefinitely if we don't receive an echo, this class has a default timeout of around 2m. You can change that using the return value of this function.

Return The Ultrasonic sensor + MQTT sensor pair, use this for advanced settings.

Parameters

- `friendly_name`: The friendly name for this sensor advertised to Home Assistant.
- `trigger_pin`: The pin the short pulse will be sent to, can be integer or `GPIOOutputPin`.
- `echo_pin`: The pin we wait that we wait on for the echo, can be integer or `GPIOInputPin`.
- `update_interval`: The time in ms between updates, defaults to 5 seconds.

```
sensor::MPU6050Component * Application::make_mpu6050_sensor(uint8_t address = 0x68, uint32_t update
```

Create a MPU6050 Accelerometer+Gyroscope+Temperature sensor hub.

This integration can be used to get accurate accelerometer readings and uncalibrated gyroscope values (in degrees per second). If you need the latter with calibration applied, your best bet it to just copy the source and do it yourself, as calibration must be performed while the sensor is at rest and this property can not be asserted for all use cases.

Return An MPU6050Component, use this to create the individual sensors and register them with `register_sensor`.

Parameters

- `address`: The address of the device, defaults to 0x68.
- `update_interval`: The interval in ms to update the sensor values.

```
Application::MakeTSL2561Sensor Applicationmake tsl2561_sensor(const std::string &name,  
                           uint8_t address = 0x23,  
                           uint32_t update_interval  
                           = 15000)
```

Create a TSL2561 accurate ambient light sensor.

This i2c-based device can provide very precise illuminance readings with great accuracy regarding the human eye response to the brightness level. By default, this sensor uses the i2c address 0x39, but you can change it with `set_address` later using the return value of this function (address 0x29 if '0' shorted on board, address 0x49 if '1' shorted on board).

The sensor values that are pushed out will be the transformed illuminance values in lx taking using the internal IR and Full Spectrum photodiodes.

Additionally, you can specify the time the sensor takes for accumulating the values (higher is better for lower light conditions, defaults to 402ms - the max) and a gain that should be used for the ADCs (defaults to 1x). Finally, this integration is energy efficient and only turns on the sensor when the values are read out.

Return The TSL2561Sensor + MQTT sensor pair, use this for advanced settings.

Parameters

- `name`: The friendly name how the sensor should be advertised.
- `address`: The address of this i2c device.
- `update_interval`: The interval in ms to update the sensor values.

```
Application::MakeBH1750Sensor Applicationmake_bh1750_sensor(const std::string &name,  
                           uint8_t address = 0x23,  
                           uint32_t update_interval =  
                           15000)
```

Create a BH1750 ambient light sensor.

This i2c-based provides ambient light readings in lx with resolutions of 4LX, 1LX and 0.5LX (the default). To change the resolution, call `set_resolution` on the return value of this function.

By default, this sensor uses the i2c address 0x23 (the default if ADDR is pulled low). If the ADDR pin is pulled high (above 0.7VCC), then you can manually set the address to 0x5C using `set_address`.

Return The BH1750Sensor + MQTT sensor pair, use this for advanced settings.

Parameters

- `name`: The friendly name that this sensor should be advertised as.
- `address`: The address of this i2c device.
- `update_interval`: The interval in ms to update the sensor values.

```
Application::MakeBME280Sensor Applicationmake_bme280_sensor(const std::string &temperature_name, const std::string &pressure_name, const std::string &humidity_name, uint8_t address = 0x77, uint32_t update_interval = 15000)
```

Create a BME280 Temperature+Pressure+Humidity i2c sensor.

Return The BME280Component + MQTT sensors tuple, use this for advanced settings.

Parameters

- **temperature_name**: The friendly name the temperature sensor should be advertised as.
- **pressure_name**: The friendly name the pressure sensor should be advertised as.
- **humidity_name**: The friendly name the humidity sensor should be advertised as.
- **address**: The i2c address of the sensor. Defaults to 0x77 (SDO to V_DDIO), can also be 0x76.
- **update_interval**: The interval in ms to update the sensor values.

```
Application::MakeBME680Sensor Applicationmake_bme680_sensor(const std::string &temperature_name, const std::string &pressure_name, const std::string &humidity_name, const std::string &gas_resistance_name, uint8_t address = 0x76, uint32_t update_interval = 15000)
```

```
Application::MakeSHT3XDSensor Applicationmake_sht3xd_sensor(const std::string &temperature_name, const std::string &humidity_name, uint8_t address = 0x44, uint32_t update_interval = 15000)
```

```
Application::MakeDHT12Sensor Applicationmake_dht12_sensor(const std::string &temperature_name, const std::string &humidity_name, uint32_t update_interval = 15000)
```

```
Application::MakeRotaryEncoderSensor Applicationmake_rotary_encoder_sensor(const std::string &name, const GPIOInputPin &pin_a, const GPIOInputPin &pin_b)
```

Create a continuous rotary encoder sensor with a digital signal.

It will keep track of how far the encoder has been turned using the signals from the two required pins A & B. There's also support for a third "index" pin. Each time this pin is pulled high, the counter will reset to 0.

Additionally, you can specify a resolution for the rotary encoder. By default, the encoder will only increment the counter once a full cycle of A&B signals has been detected to prevent triggers from noise. You can change this behavior using the set_resolution method.

The output value of this rotary encoder is a raw integer step value. Use filters to convert this raw value to something sensible like degrees. Next, this sensor pushes its state on every detected counter change.

Read <https://playground.arduino.cc/Main/RotaryEncoders> to see how they work.

Return A *MakeRotaryEncoderSensor*, use this for advanced settings.

Parameters

- **name:** The name of the rotary encoder.
- **pin_a:** The first pin of the sensor.
- **pin_b:** The second pin of the sensor.

```
Application::MakeTemplateSensor Applicationmake_template_sensor(const std::string &name,
                                                               std::function<optional<float>>
                                                               > &&uint32_t update_interval = 15000)
```

```
Application::MakeMAX6675Sensor Applicationmake_max6675_sensor(const std::string &name,
                                                               const GPIOOutputPin
                                                               &cs, const GPIOOut-
                                                               putPin &clock, const
                                                               GPIOInputPin &miso,
                                                               uint32_t update_interval
                                                               = 15000)
```

```
Application::MakeESP32HallSensor Applicationmake_esp32_hall_sensor(const std::string
                                                               &name, uint32_t
                                                               update_interval =
                                                               15000)
```

```
PowerSupplyComponent *Applicationmake_power_supply(const GPIOOutputPin &pin,
                                                 uint32_t enable_time = 20, uint32_t
                                                 keep_on_time = 10000)
```

Create a power supply component that will automatically switch on and off.

Return The *PowerSupplyComponent*.

Parameters

- **pin:** The pin the power supply is connected to.
- **enable_time:** The time (in ms) the power supply needs until it can provide high power when powering on.
- **keep_on_time:** The time (in ms) the power supply should stay on when it is not used.

```
LEDOutputComponent *Applicationmake_ledc_output(uint8_t pin, float frequency = 1000.0f,
                                                uint8_t bit_depth = 12)
```

Create a ESP32 LEDC channel.

Return The LEDC component. Use this for advanced settings.

Parameters

- **pin**: The pin.
- **frequency**: The PWM frequency.
- **bit_depth**: The LEDC bit depth.

`PCA9685OutputComponent *Applicationmake_pca9685_component(float frequency)`
Create a PCA9685 component.

Return The PCA9685 component. Use this for advanced settings.

Parameters

- **frequency**: The PWM frequency.

`output::GPIOBinaryOutputComponent *Applicationmake_gpio_output(const GPIOOutputPin &pin)`

Create a simple binary GPIO output component.

Note: This is *only* a binary output component, not a switch that will be exposed in Home Assistant. See `make_simple_gpio_switch` for a switch.

Return The GPIOBinaryOutputComponent. Use this for advanced settings.

Parameters

- **pin**: The GPIO pin.

`ESP8266PWMOutput *Applicationmake_esp8266_pwm_output(GPIOOutputPin pin)`
Create an ESP8266 software PWM channel.

Warning: This is a *software* PWM and therefore can have noticeable flickering. Additionally, this software PWM can't output values higher than 80%.

Return The PWM output channel, use this for advanced settings and using it with lights.

Parameters

- **pin**: The pin for this PWM output, supported pins are 0-16.

`MQTTJSONLightComponent *Applicationregister_light(light::LightState *state)`
Register a light within esphomelib.

`Application::MakeLight Applicationmake_light_for_light_output(const std::string &name, light::LightOutput *output)`

`Application::MakeLight Applicationmake_binary_light(const std::string &friendly_name, output::BinaryOutput *binary)`

Create a binary light.

Return The components for this light. Use this for advanced settings.

Parameters

- **friendly_name**: The name the light should be advertised as. Leave empty for no automatic discovery.

- **binary**: The binary output channel.

```
Application::MakeLight Application::make_monochromatic_light(const std::string &friendly_name, std::string out-put::FloatOutput *mono)
```

Create a monochromatic light.

Return The components for this light. Use this for advanced settings.

Parameters

- **friendly_name**: The name the light should be advertised as. Leave empty for no automatic discovery.
- **mono**: The output channel.

```
Application::MakeLight Application::make_rgb_light(const std::string &friendly_name, std::string out-put::FloatOutput *red, std::string out-put::FloatOutput *green, std::string out-put::FloatOutput *blue)
```

Create a RGB light.

Return The components for this light. Use this for advanced settings.

Parameters

- **friendly_name**: The name the light should be advertised as. Leave empty for no automatic discovery.
- **red**: The red output channel.
- **green**: The green output channel.
- **blue**: The blue output channel.

```
Application::MakeLight Application::make_rgbw_light(const std::string &friendly_name, std::string out-put::FloatOutput *red, std::string out-put::FloatOutput *green, std::string out-put::FloatOutput *blue, std::string out-put::FloatOutput *white)
```

Create a RGBW light.

Return The components for this light. Use this for advanced settings.

Parameters

- **friendly_name**: The name the light should be advertised as. Leave empty for no automatic discovery.
- **red**: The red output channel.
- **green**: The green output channel.
- **blue**: The blue output channel.
- **white**: The white output channel.

```
Application::MakeFastLEDLight Application::make_fast_led_light(const std::string &name)
```

Create an FastLED light.

```
MQTTSwitchComponent *Application::register_switch(switch_::Switch *switch_)
```

Register a Switch internally, creating a MQTT Switch if the MQTT client is set up.

```
IRTransmitterComponent *Applicationmake_ir_transmitter(const GPIOOutputPin &pin,
                                                       uint8_t carrier_duty_percent =
                                                       50)
```

Create an IR transmitter.

Return The IRTransmitterComponent. Use this for advanced settings.

Parameters

- **pin:** The pin the IR led is connected to.
- **carrier_duty_percent:** The duty cycle of the IR output. Decrease this if your LED gets hot.

```
Application::MakeGPIOSwitch Applicationmake_gpio_switch(const std::string
                                                       &friendly_name, const GPIOOutputPin &pin)
```

Create a simple GPIO switch that can be toggled on/off and appears in the frontend.

Return A GPIOSwitchStruct, use this to set advanced settings.

Parameters

- **pin:** The pin used for this switch. Can be integer or *GPIOOutputPin*.
- **friendly_name:** The friendly name advertised to Home Assistant for this switch-

```
Application::MakeRestartSwitch Applicationmake_restart_switch(const std::string
                                                               &friendly_name)
```

Make a simple switch that restarts the device with the provided friendly name.

```
Application::MakeShutdownSwitch Applicationmake_shutdown_switch(const std::string
                                                               &friendly_name)
```

Make a simple switch that shuts the node down indefinitely.

```
Application::MakeSimpleSwitch Applicationmake_simple_switch(const std::string
                                                          &friendly_name, output::BinaryOutput *output)
```

Make a simple switch that exposes a binary output as a switch.

```
Application::MakeTemplateSwitch Applicationmake_template_switch(const std::string &name)
```

```
fan::MQTTFanComponent *Applicationregister_fan(fan::FanState *state)
Register a fan internally.
```

```
Application::MakeFan Applicationmake_fan(const std::string &friendly_name)
```

Create and connect a Fan with the specified friendly name.

Return A FanStruct, use the output field to set your output channels.

Parameters

- **friendly_name:** The friendly name of the Fan to advertise.

```
cover::MQTTCoverComponent *Applicationregister_cover(cover::Cover *cover)
```

```
Application::MakeTemplateCover Applicationmake_template_cover(const std::string &name)
```

```
DebugComponent *Applicationmake_debug_component()
```

```
DeepSleepComponent *Applicationmake_deep_sleep_component()
PCF8574Component *Applicationmake_pcf8574_component(uint8_t address = 0x21, bool
                                                       pcf8575 = false)
```

Create a PCF8574/PCF8575 port expander component.

This component will allow you to emulate *GPIOInputPin* and *GPIOOutputPin* instances that are used within esphomelib. You can therefore simply pass the result of calling `make_pin` on the component to any method accepting *GPIOInputPin* or *GPIOOutputPin*.

Optionally, this component also has support for the 16-channel PCF8575 port expander. To use the PCF8575, set the `pcf8575` in this helper function.

Return The PCF8574Component instance to get individual pins.

Parameters

- `address`: The i2c address to use for this port expander. Defaults to 0x21.
- `pcf8575`: If this is an PCF8575. Defaults to PCF8574.

```
template <class C>
C *Applicationregister_component(C *c)
```

Register the component in this *Application* instance.

```
template <class C>
C *Applicationregister_controller(C *c)
```

```
void Applicationsetup()
```

Set up all the registered components. Call this at the end of your `setup()` function.

```
void Applicationloop()
```

Make a loop iteration. Call this in your `loop()` function.

```
WiFiComponent *Applicationget_wifi() const
```

```
MQTTClientComponent *Applicationget_mqtt_client() const
```

```
const std::string &Applicationget_name() const
```

Get the name of this *Application* set by `set_name()`.

Protected Attributes

```
std::vector<Component *> Applicationcomponents_ = {}
std::vector<Controller *> Applicationcontrollers_ = {}
mqtt::MQTTClientComponent *Applicationmqtt_client_ = {nullptr}
WiFiComponent *Applicationwifi_ = {nullptr}
std::string Applicationname_
Component::ComponentState Applicationapplication_state_ = {Component::CONSTRUCTION}
I2CCComponent *Applicationi2c_ = {nullptr}

struct ApplicationMakeADCsensor
```

Public Members

```
sensor::ADCSensorComponent *Application::MakeADCSensoradc  
sensor::MQTTSensorComponent *Application::MakeADCSensormqtt  
struct ApplicationMakeBH1750Sensor
```

Public Members

```
sensor::BH1750Sensor *Application::MakeBH1750Sensorbh1750  
sensor::MQTTSensorComponent *Application::MakeBH1750Sensormqtt  
struct ApplicationMakeBME280Sensor
```

Public Members

```
sensor::BME280Component *Application::MakeBME280Sensorbme280  
sensor::MQTTSensorComponent *Application::MakeBME280Sensormqtt_temperature  
sensor::MQTTSensorComponent *Application::MakeBME280Sensormqtt_pressure  
sensor::MQTTSensorComponent *Application::MakeBME280Sensormqtt_humidity  
struct ApplicationMakeBME680Sensor
```

Public Members

```
sensor::BME680Component *Application::MakeBME680Sensorbme680  
sensor::MQTTSensorComponent *Application::MakeBME680Sensormqtt_temperature  
sensor::MQTTSensorComponent *Application::MakeBME680Sensormqtt_pressure  
sensor::MQTTSensorComponent *Application::MakeBME680Sensormqtt_humidity  
sensor::MQTTSensorComponent *Application::MakeBME680Sensormqtt_gas_resistance  
struct ApplicationMakeBMP085Sensor
```

Public Members

```
sensor::BMP085Component *Application::MakeBMP085Sensorbmp  
sensor::MQTTSensorComponent *Application::MakeBMP085Sensormqtt_temperature  
sensor::MQTTSensorComponent *Application::MakeBMP085Sensormqtt_pressure  
struct ApplicationMakeDHT12Sensor
```

Public Members

```
sensor::DHT12Component *Application::MakeDHT12Sensordht12
sensor::MQTTSensorComponent *Application::MakeDHT12Sensormqtt_temperature
sensor::MQTTSensorComponent *Application::MakeDHT12Sensormqtt_humidity

struct ApplicationMakeDHTSensor
```

Public Members

```
sensor::DHTComponent *Application::MakeDHTSensordht
sensor::MQTTSensorComponent *Application::MakeDHTSensormqtt_temperature
sensor::MQTTSensorComponent *Application::MakeDHTSensormqtt_humidity

struct ApplicationMakeESP32HallSensor
```

Public Members

```
sensor::ESP32HallSensor *Application::MakeESP32HallSensorhall
sensor::MQTTSensorComponent *Application::MakeESP32HallSensormqtt

struct ApplicationMakeFan
```

Public Members

```
fan::BasicFanComponent *Application::MakeFanoutput
fan::FanState *Application::MakeFanstate
fan::MQTTFanComponent *Application::MakeFanmqtt

struct ApplicationMakeFastLEDDLight
```

Public Members

```
light::FastLEDDLightOutputComponent *Application::MakeFastLEDDLightfast_led
light::LightState *Application::MakeFastLEDDLightstate
light::MQTTJSONLightComponent *Application::MakeFastLEDDLightmqtt

struct ApplicationMakeGPIOBinarySensor
```

Public Members

```
binary_sensor::GPIOBinarySensorComponent *Application::MakeGPIOBinarySensorgpio
binary_sensor::MQTTBinarySensorComponent *Application::MakeGPIOBinarySensormqtt

struct ApplicationMakeGPIOSwitch
```

Public Members

```
output::GPIOBinaryOutputComponent *Application::MakeGPIOSwitchgpio
switch_::SimpleSwitch *Application::MakeGPIOSwitchswitch_
switch_::MQTTSwitchComponent *Application::MakeGPIOSwitchmqtt

struct ApplicationMakeHDC1080Sensor
```

Public Members

```
sensor::HDC1080Component *Application::MakeHDC1080Sensorhdc1080
sensor::MQTTSensorComponent *Application::MakeHDC1080Sensormqtt_temperature
sensor::MQTTSensorComponent *Application::MakeHDC1080Sensormqtt_humidity

struct ApplicationMakeHTU21DSensor
```

Public Members

```
sensor::HTU21DComponent *Application::MakeHTU21DSensorhtu21d
sensor::MQTTSensorComponent *Application::MakeHTU21DSensormqtt_temperature
sensor::MQTTSensorComponent *Application::MakeHTU21DSensormqtt_humidity

struct ApplicationMakeLight
```

Public Members

```
light::LightOutput *Application::MakeLightoutput
light::LightState *Application::MakeLightstate
light::MQTTJSONLightComponent *Application::MakeLightmqtt

struct ApplicationMakeMAX6675Sensor
```

Public Members

```
sensor::MAX6675Sensor *Application::MakeMAX6675Sensormax6675
sensor::MQTTSensorComponent *Application::MakeMAX6675Sensormqtt

struct ApplicationMakePulseCounterSensor
```

Public Members

```
sensor::PulseCounterSensorComponent *Application::MakePulseCounterSensorpcnt
sensor::MQTTSensorComponent *Application::MakePulseCounterSensormqtt

struct ApplicationMakeRestartSwitch
```

Public Members

```
switch_::RestartSwitch *Application::MakeRestartSwitchrestart
switch_::MQTTSwitchComponent *Application::MakeRestartSwitchmqtt
struct ApplicationMakeRotaryEncoderSensor
```

Public Members

```
sensor::RotaryEncoderSensor *Application::MakeRotaryEncoderSensorrotary_encoder
sensor::MQTTSensorComponent *Application::MakeRotaryEncoderSensormqtt
struct ApplicationMakeSHT3XDSensor
```

Public Members

```
sensor::SHT3XDComponent *Application::MakeSHT3XDSensorsht3xd
sensor::MQTTSensorComponent *Application::MakeSHT3XDSensormqtt_temperature
sensor::MQTTSensorComponent *Application::MakeSHT3XDSensormqtt_humidity
struct ApplicationMakeShutdownSwitch
```

Public Members

```
switch_::ShutdownSwitch *Application::MakeShutdownSwitchshutdown
switch_::MQTTSwitchComponent *Application::MakeShutdownSwitchmqtt
struct ApplicationMakeSimpleSwitch
```

Public Members

```
switch_::SimpleSwitch *Application::MakeSimpleSwitchswitch_
switch_::MQTTSwitchComponent *Application::MakeSimpleSwitchmqtt
struct ApplicationMakeStatusBinarySensor
```

Public Members

```
binary_sensor::StatusBinarySensor *Application::MakeStatusBinarySensorstatus
binary_sensor::MQTTBinarySensorComponent *Application::MakeStatusBinarySensormqtt
struct ApplicationMakeTemplateBinarySensor
```

Public Members

```
binary_sensor::TemplateBinarySensor *Application::MakeTemplateBinarySensortemplate_
binary_sensor::MQTTBinarySensorComponent *Application::MakeTemplateBinarySensormqtt
struct ApplicationMakeTemplateCover
```

Public Members

```
cover::TemplateCover *Application::MakeTemplateCovertemplate_
cover::MQTTCoverComponent *Application::MakeTemplateCovermqtt
struct ApplicationMakeTemplateSensor
```

Public Members

```
sensor::TemplateSensor *Application::MakeTemplateSensortemplate_
sensor::MQTTSensorComponent *Application::MakeTemplateSensormqtt
struct ApplicationMakeTemplateSwitch
```

Public Members

```
switch_::TemplateSwitch *Application::MakeTemplateSwitchtemplate_
switch_::MQTTSwitchComponent *Application::MakeTemplateSwitchmqtt
struct ApplicationMakeTSL2561Sensor
```

Public Members

```
sensor::TSL2561Sensor *Application::MakeTSL2561Sensortsl2561
sensor::MQTTSensorComponent *Application::MakeTSL2561Sensormqtt
struct ApplicationMakeUltrasonicSensor
```

Public Members

```
sensor::UltrasonicSensorComponent *Application::MakeUltrasonicSensorultrasonic
sensor::MQTTSensorComponent *Application::MakeUltrasonicSensormqtt
```

Application App

Global storage of *Application* pointer - only one *Application* can exist.

Component

Every object that should be handled by the Application instance and receive *setup()* and *loop()* calls must be a subclass of *Component*.

API Reference

Component

`class Component`

The base class for all esphomelib components.

esphomelib uses components to separate code for self-contained units such as peripheral devices in order to keep the library clean and simple. Each component should be registered in the [Application](#) instance via `add_component`. The application will then call the component's `setup()` and `loop()` methods at the appropriate time. Each component should save all the information required for setup in the constructor, and only do the actual hardware initialization, such as `pinMode`, in `setup()`.

Additionally, the `get_setup_priority()` and `get_loop_priority()` can be overwritten in order to force the `setup()` or `loop()` methods to be executed earlier than other components. For example, setting up a GPIO pin to be output-only should be done before WiFi and MQTT are initialized - so the component can assign itself a high priority via `get_setup_priority()`.

See `Application::add_component()`

Subclassed by `binary_sensor::ESP32TouchComponent`, `binary_sensor::GPIOBinarySensorComponent`, `binary_sensor::TemplateBinarySensor`, `cover::TemplateCover`, `DebugComponent`, `DeepSleepComponent`, `DelayAction< T >`, `ESP32BLETracker`, `fan::BasicFanComponent`, `I2CComponent`, `io::PCF8574Component`, `light::FastLEDDOutputComponent`, `light::LightState`, `LogComponent`, `mqtt::MQTTClientComponent`, `mqtt::MQTTComponent`, `OTAClientComponent`, `output::ESP8266PWMOutput`, `output::GPIOBinaryOutputComponent`, `output::LEDCOutputComponent`, `output::PCA9685OutputComponent`, `PollingComponent`, `PowerSupplyComponent`, `sensor::ADS1115Component`, `sensor::DebounceFilter`, `sensor::HeartbeatFilter`, `sensor::RotaryEncoderSensor`, `StartupTrigger`, `switch_::IRTransmitterComponent`, `switch_::Switch`, `WebServer`, `WiFiComponent`

Public Types

`enum ComponentComponentState`

Values:

`ComponentCONSTRUCTION` = 0

`ComponentSETUP` = 1

`ComponentLOOP` = 2

`ComponentFAILED` = 3

Public Functions

`void Componentsetup()`

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing nothing.

`void Componentloop()`

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

```
float Component::get_setup_priority() const
    priority of setup().
```

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
float Component::get_loop_priority() const
    priority of loop().
```

higher -> executed earlier

Defaults to 0.

Return The loop priority of this component

```
void Component::loop_()
    Public loop() functions.
```

These will be called by the `Application` instance.

Note: This should normally not be overridden, unless you know what you're doing. They're basically to make creating custom components easier. For example the SensorComponent can override these methods to not have the user call some super methods within their custom sensors. These methods should ALWAYS call the `loop_internal()` and `setup_internal()` methods.

Basically, it handles stuff like interval/timeout functions and eventually calls `loop()`.

```
void Component::setup_()
```

```
Component::ComponentState Component::get_component_state() const
```

```
void Component::mark_failed()
```

Mark this component as failed.

Any future timeouts/intervals/setup/loop will no longer be called.

This might be useful if a component wants to indicate that a connection to its peripheral failed. For example, i2c based components can check if the remote device is responding and otherwise mark the component as failed. Eventually this will also enable smart status LEDs.

```
bool Component::is_failed()
```

Protected Types

```
using Component::time_func_t = std::function<void()>
    Simple typedef for interval/timeout functions.
```

See `set_interval()`

See `set_timeout()`

Protected Functions

void *Componentloop_internal()*

void *Componentssetup_internal()*

void *Componentset_interval(uint32_t interval, time_func_t &&f)*

void *Componentset_interval(const std::string &name, uint32_t interval, time_func_t &&f)*

Set an interval function with a unique name.

Empty name means no cancelling possible.

This will call f every interval ms. Can be cancelled via *CancelInterval()*. Similar to javascript's *setInterval()*.

IMPORTANT: Do not rely on this having correct timing. This is only called from *loop()* and therefore can be significantly delay. If you need exact timing please use hardware timers.

See *cancel_interval()*

Parameters

- **name:** The identifier for this interval function.
- **interval:** The interval in ms.
- **f:** The function (or lambda) that should be called

bool *Componentcancel_interval(const std::string &name)*

Cancel an interval function.

Return Whether an interval functions was deleted.

Parameters

- **name:** The identifier for this interval function.

void *Componentset_timeout(uint32_t timeout, time_func_t &&f)*

void *Componentset_timeout(const std::string &name, uint32_t timeout, time_func_t &&f)*

Set a timeout function with a unique name.

Similar to javascript's *setTimeout()*. Empty name means no cancelling possible.

IMPORTANT: Do not rely on this having correct timing. This is only called from *loop()* and therefore can be significantly delay. If you need exact timing please use hardware timers.

See *cancel_timeout()*

Parameters

- **name:** The identifier for this timeout function.
- **timeout:** The timeout in ms.
- **f:** The function (or lambda) that should be called

bool *Componentcancel_timeout(const std::string &name)*

Cancel a timeout function.

Return Whether a timeout functions was deleted.

Parameters

- `name`: The identifier for this timeout function.

void `Componentdefer(const std::string &name, time_func_t &&f)`

Defer a callback to the next `loop()` call.

If name is specified and a `defer()` object with the same name exists, the old one is first removed.

Parameters

- `name`: The name of the defer function.
- `f`: The callback.

void `Componentdefer(time_func_t &&f)`

Defer a callback to the next `loop()` call.

bool `Componentcancel_defer(const std::string &name)`

Cancel a defer callback using the specified name, name must not be empty.

bool `Componentcancel_time_function(const std::string &name, TimeFunction::Type type)`

Cancel an only time function. If name is empty, won't do anything.

Protected Attributes

`std::vector<TimeFunction> Componenttime_functions_`

Storage for interval/timeout functions.

Intentionally a vector despite its map-like nature, because of the memory overhead.

`ComponentState Componentcomponent_state_ = {CONSTRUCTION}`

State of this component.

`struct ComponentTimeFunction`

Internal struct for storing timeout/interval functions.

Public Types

`enum ComponentType`

Values:

`ComponentTIMEOUT`

`ComponentINTERVAL`

`ComponentDEFER`

Public Functions

bool `Component::TimeFunctionshould_run(uint32_t now) const`

Public Members

`std::string Component::TimeFunctionname`

The name/id of this `TimeFunction`.

`Component::TimeFunction::Type Component::TimeFunctiontype`

The type of this `TimeFunction`. Either TIMEOUT, INTERVAL or DEFER.

`uint32_t Component::TimeFunctioninterval`

The interval/timeout of this function.

`uint32_t Component::TimeFunctionlast_execution`

The last execution for interval functions and the time, SetInterval was called, for timeout functions.

`time_func_t Component::TimeFunctionf`

The function (or callback) itself.

`bool Component::TimeFunctionremove`

PollingComponent

`class PollingComponent`

This class simplifies creating components that periodically check a state.

You basically just need to implement the update() function, it will be called every update_interval ms after startup. Note that this class cannot guarantee a correct timing, as it's not using timers, just a software polling feature with set_interval() from `Component`.

Inherits from `Component`

Subclassed by `sensor::BME280Component`, `sensor::BME680Component`, `sensor::BMP085Component`, `sensor::DallasComponent`, `sensor::DHT12Component`, `sensor::DHTComponent`, `sensor::HDC1080Component`, `sensor::HTU21DComponent`, `sensor::MPU6050Component`, `sensor::PollingSensorComponent`, `sensor::SHT3XDCComponent`

Public Functions

`PollingComponent::PollingComponent(uint32_t update_interval)`

Initialize this polling component with the given update interval in ms.

Parameters

- `update_interval`: The update interval in ms.

`void PollingComponent::set_update_interval(uint32_t update_interval)`

Manually set the update interval in ms for this polling object.

Override this if you want to do some validation for the update interval.

Parameters

- `update_interval`: The update interval in ms.

`virtual void PollingComponent::update() = 0`

`void PollingComponent::setup_()`

```
uint32_t PollingComponentget_update_interval() const
Get the update interval in ms of this sensor.
```

Protected Attributes

```
uint32_t PollingComponentupdate_interval_
```

Setup Priorities

```
namespace setup_priority
default setup priorities for components of different types.
```

Variables

```
const float setup_priorityHARDWARE = 100.0f
for hardware initialization, but only where it's really necessary (like outputs)

const float setup_priorityWIFI = 10.0f
for WiFi initialization

const float setup_priorityMQTT_CLIENT = 7.5f
for the MQTT client initialization

const float setup_priorityHARDWARE_LATE = 0.0f

const float setup_priorityMQTT_COMPONENT = -5.0f
for MQTT component initialization

const float setup_priorityLATE = -10.0f
```

WiFi

Example Usage

```
// Basic
App.init_wifi("YOUR_SSID", "YOUR_PASSWORD");
// Manual IP
auto *wifi = App.init_wifi("YOUR_SSID", "YOUR_PASSWORD");
wifi->set_sta_manual_ip(ManualIP{
    .static_ip = IPAddress(192, 168, 178, 42),
    .gateway = IPAddress(192, 168, 178, 1),
    .subnet = IPAddress(255, 255, 255, 0)
});
// AP
wifi->set_ap("AP SSID", "Optional AP Password");
```

API Reference

WiFiComponent

```
class WiFiComponent
This component is responsible for managing the ESP WiFi interface.
```

Inherits from [Component](#)

Public Functions

`WiFiComponent WiFiComponent()`

Construct a [WiFiComponent](#).

`void WiFiComponent::set_sta(const std::string &ssid, const std::string &password)`

Setup the STA (client) mode. The parameters define which station to connect to.

`void WiFiComponent::set_sta_manual_ip(ManualIP manual_ip)`

Manually set a static IP for this WiFi interface.

`void WiFiComponent::set_ap(const std::string &ssid, const std::string &password = "", uint8_t channel = 1)`

Setup an Access Point that should be created if no connection to a station can be made.

This can also be used without `set_sta()`. Then the AP will always be active.

If both STA and AP are defined, then both will be enabled at startup, but if a connection to a station can be made, the AP will be turned off again.

`void WiFiComponent::set_ap_manual_ip(ManualIP manual_ip)`

Manually set the manual IP for the AP.

`void WiFiComponent::set_hostname(std::string &&hostname)`

Set the advertised hostname, defaults to the App name.

`const std::string & WiFiComponent::get_hostname()`

`void WiFiComponent::setup()`

Setup WiFi interface.

`float WiFiComponent::get_setup_priority() const`

WIFI setup priority.

`float WiFiComponent::get_loop_priority() const`

priority of `loop()`.

higher -> executed earlier

Defaults to 0.

Return The loop priority of this component

`void WiFiComponent::loop()`

Reconnect WiFi if required.

`bool WiFiComponent::has_sta() const`

`bool WiFiComponent::has_ap() const`

Protected Functions

```
void WiFiComponentsetup_sta_config(bool show_config = true)  
void WiFiComponentsetup_ap_config()  
void WiFiComponentwait_for_sta()  
    Waits for the WiFi class to return a connected state.  
void WiFiComponentsta_connected()
```

Protected Attributes

```
std::string WiFiComponenthostname_  
bool WiFiComponentsta_on_  
std::string WiFiComponentsta_ssid_  
std::string WiFiComponentsta_password_  
optional<ManualIP> WiFiComponentsta_manual_ip_  
bool WiFiComponentap_on_  
std::string WiFiComponentap_ssid_  
std::string WiFiComponentap_password_  
uint8_t WiFiComponentap_channel_  
optional<ManualIP> WiFiComponentap_manual_ip_
```

Protected Static Functions

```
void WiFiComponenton_wifi_event(WiFiEvent_t event)  
    Used for logging WiFi events.
```

```
struct ManualIP  
Struct for setting static IPs in WiFiComponent.
```

Public Members

```
IPAddress ManualIPstatic_ip  
IPAddress ManualIPgateway  
IPAddress ManualIPsubnet  
IPAddress ManualIPdns1  
    The first DNS server. 0.0.0.0 for default.  
IPAddress ManualIPdns2  
    The second DNS server. 0.0.0.0 for default.
```

MQTT Client

API Reference

MQTTClientComponent

```
class mqtt::MQTTClientComponent  
Inherits from Component
```

Public Functions

```
mqtt::MQTTClientComponent::MQTTClientComponent(const MQTTCredentials &credentials)
```

```
void mqtt::MQTTClientComponent::set_last_will(MQTTMessage &&message)  
Set the last will testament message.
```

```
void mqtt::MQTTClientComponent::disable_last_will()  
Remove the last will testament message.
```

```
void mqtt::MQTTClientComponent::set_birth_message(MQTTMessage &&message)  
Set the birth message.
```

```
void mqtt::MQTTClientComponent::disable_birth_message()  
Remove the birth message.
```

```
void mqtt::MQTTClientComponent::set_keep_alive(uint16_t keep_alive_s)  
Set the keep alive time in seconds, every 0.7*keep_alive a ping will be sent.
```

```
void mqtt::MQTTClientComponent::set_discovery_info(std::string &&prefix, bool retain)  
Set the Home Assistant discovery info.
```

See [MQTT Discovery](#).

Parameters

- **prefix**: The Home Assistant discovery prefix.
- **retain**: Whether to retain discovery messages.

```
const MQTTDiscoveryInfo &mqtt::MQTTClientComponent::get_discovery_info() const  
Get Home Assistant discovery info.
```

```
void mqtt::MQTTClientComponent::disable_discovery()  
Globally disable Home Assistant discovery.
```

```
bool mqtt::MQTTClientComponent::is_discovery_enabled() const
```

```
void mqtt::MQTTClientComponent::set_client_id(std::string client_id)  
Manually set the client id, by default it's <name>-<MAC>, it's automatically truncated to 23  
chars.
```

```
void mqtt::MQTTClientComponent::add_ssl_fingerprint(const std::array<uint8_t,  
SHA1_SIZE> &fingerprint)  
Add a SSL fingerprint to use for TCP SSL connections to the MQTT broker.
```

To use this feature you first have to globally enable the `ASYNC_TCP_SSL_ENABLED` define flag. This function can be called multiple times and any certificate that matches any of the provided fingerprints will match. Calling this method will also automatically disable all non-ssl connections.

Warning This is *not* secure and *not* how SSL is usually done. You'll have to add a separate fingerprint for every certificate you use. Additionally, the hashing algorithm used here due to the constraints of the MCU, SHA1, is known to be insecure.

Parameters

- **fingerprint:** The SSL fingerprint as a 20 value long std::array.

```
const Availability &mqtt::MQTTClientComponentget_availability()
```

```
void mqtt::MQTTClientComponentset_topic_prefix(std::string topic_prefix)
```

Set the topic prefix that will be prepended to all topics together with “/”.

This will, in most cases, be the name of your *Application*.

For example, if “livingroom” is passed to this method, all state topics will, by default, look like “livingroom/.../state”

Parameters

- **topic_prefix:** The topic prefix. The last “/” is appended automatically.

```
const std::string &mqtt::MQTTClientComponentget_topic_prefix() const
```

Get the topic prefix of this device, using default if necessary.

```
void mqtt::MQTTClientComponentset_log_message_template(MQTTMessage &&message)
```

Manually set the topic used for logging.

```
void mqtt::MQTTClientComponentdisable_log_message()
```

Get the topic used for logging. Defaults to “<topic_prefix>/debug” and the value is cached for speed.

```
bool mqtt::MQTTClientComponentis_log_message_enabled() const
```

```
void mqtt::MQTTClientComponentsubscribe(const std::string &topic, mqtt_callback_t callback, uint8_t qos = 0)
```

Subscribe to an MQTT topic and call callback when a message is received.

Parameters

- **topic:** The topic. Wildcards are currently not supported.
- **callback:** The callback function.
- **qos:** The QoS of this subscription.

```
void mqtt::MQTTClientComponentsubscribe_json(const std::string &topic, json_parse_t callback, uint8_t qos = 0)
```

Subscribe to a MQTT topic and automatically parse JSON payload.

If an invalid JSON payload is received, the callback will not be called.

Parameters

- **topic:** The topic. Wildcards are currently not supported.
- **callback:** The callback with a parsed JsonObject that will be called when a message with matching topic is received.
- **qos:** The QoS of this subscription.

```
void mqtt::MQTTClientComponentpublish(const MQTTMessage &message)
Publish a MQTTMessage.
```

Parameters

- **message**: The message.

```
void mqtt::MQTTClientComponentpublish(const std::string &topic, const std::string &payload, uint8_t qos, bool retain)
Publish a MQTT message.
```

Parameters

- **topic**: The topic.
- **payload**: The payload.
- **retain**: Whether to retain the message.

```
void mqtt::MQTTClientComponentpublish_json(const std::string &topic, const json_build_t &f, uint8_t qos, bool retain)
Construct and send a JSON MQTT message.
```

Parameters

- **topic**: The topic.
- **f**: The Json Message builder.
- **retain**: Whether to retain the message.

```
bool mqtt::MQTTClientComponentis_connected()
Return whether this client is currently connected to the MQTT server.
```

```
void mqtt::MQTTClientComponentadd_on_connect_callback(std::function<void> &&callback)
Add a callback that will be called every time the MQTT client reconnects.
```

```
void mqtt::MQTTClientComponentsetup()
Setup the MQTT client, registering a bunch of callbacks and attempting to connect.
```

```
void mqtt::MQTTClientComponentloop()
Reconnect if required.
```

```
float mqtt::MQTTClientComponentget_setup_priority() const
MQTT client setup priority.
```

```
void mqtt::MQTTClientComponenton_message(const std::string &topic, const std::string &payload)
```

```
MQTTMessageTrigger *mqtt::MQTTClientComponentmake_message_trigger(const std::string &topic, uint8_t qos = 0)
template <typename T>
MQTTPublishAction<T> *mqtt::MQTTClientComponentmake_publish_action()
```

Protected Functions

```
void mqtt::MQTTClientComponentreconnect()
    Reconnect to the MQTT broker if not already connected.

void mqtt::MQTTClientComponentrecalculate_availability()
    Re-calculate the availability property.
```

Protected Attributes

```
MQTTCredentials mqtt::MQTTClientComponentcredentials_
MQTTMessage mqtt::MQTTClientComponentlast_will_
    The last will message.
    Disabled optional denotes it being default and an empty topic denotes the the feature being
    disabled.

MQTTMessage mqtt::MQTTClientComponentbirth_message_
    The birth message (e.g.
    the message that's send on an established connection. See last_will_ for what different values
    denote.

Availability mqtt::MQTTClientComponentavailability_ = {}
    Caches availability.

MQTTDiscoveryInfo mqtt::MQTTClientComponentdiscovery_info_ = { .prefix = "homeassistant", .retain = true }
    The discovery info options for Home Assistant.
    Undefined optional means default and empty prefix means disabled.

std::string mqtt::MQTTClientComponenttopic_prefix_ = {}
MQTTMessage mqtt::MQTTClientComponentlog_message_
std::vector<MQTTSubscription> mqtt::MQTTClientComponentsubscriptions_
AsyncMqttClient mqtt::MQTTClientComponentmqtt_client_
CallbackManager<void()> mqtt::MQTTClientComponenton_connect_ = {}

using mqtt::mqtt_callback_t = typedef std::function<void(const std::string &)>
    Callback for MQTT subscriptions.

First parameter is the topic, the second one is the payload.

struct mqtt::MQTTMessage
    internal struct for MQTT messages.
```

Public Members

```
std::string mqtt::MQTTMessagetopic
std::string mqtt::MQTTMessagepayload
uint8_t mqtt::MQTTMessageqos
    QoS. Only for last will testaments.

bool mqtt::MQTTMessageretain
```

```
struct mqtt::MQTTSubscription
internal struct for MQTT subscriptions.
```

Public Members

```
std::string mqtt::MQTTSubscriptiontopic
uint8_t mqtt::MQTTSubscriptionqos
mqtt_callback_t mqtt::MQTTSubscriptioncallback

struct mqtt::MQTTCredentials
internal struct for MQTT credentials.
```

Public Members

```
std::string mqtt::MQTTCredentialsaddress
The address of the server without port number.

uint16_t mqtt::MQTTCredentialsport
The port number of the server.

std::string mqtt::MQTTCredentialsusername
std::string mqtt::MQTTCredentialspassword
std::string mqtt::MQTTCredentialsclient_id
The client ID. Will automatically be truncated to 23 characters.

struct mqtt::Availability
Simple data struct for Home Assistant component availability.
```

Public Members

```
std::string mqtt::Availabilitytopic
Empty means disabled.

std::string mqtt::Availabilitypayload_available
std::string mqtt::Availabilitypayload_not_available
```

```
struct mqtt::MQTTDiscoveryInfo
Internal struct for MQTT Home Assistant discovery.
```

See [MQTT Discovery](#).

Public Members

```
std::string mqtt::MQTTDiscoveryInfoprefix
The Home Assistant discovery prefix. Empty means disabled.

bool mqtt::MQTTDiscoveryInforetain
Whether to retain discovery messages.
```

MQTTClientComponent *mqtt::global_mqtt_client = nullptr

MQTTComponent

```
class mqtt::MQTTComponent
```

MQTTComponent is the base class for all components that interact with MQTT to expose certain functionality or data from actuators or sensors to clients.

Although this class should work with all MQTT solutions, it has been specifically designed for use with Home Assistant. For example, this class supports Home Assistant MQTT discovery out of the box.

In order to implement automatic Home Assistant discovery, all sub-classes should:

1. Implement `send_discovery` that creates a Home Assistant discovery payload.
2. Override `component_type()` to return the appropriate component type such as “light” or “sensor”.
3. Subscribe to command topics using `subscribe()` or `subscribe_json()` during `setup()`.

In order to best separate the front- and back-end of esphomelib, all sub-classes should only parse/send MQTT messages and interact with back-end components via callbacks to ensure a clean separation.

Inherits from *Component*

Subclassed by `binary_sensor::MQTTBinarySensorComponent`, `cover::MQTTCoverComponent`, `fan::MQTTFanComponent`, `light::MQTTJSONLightComponent`, `sensor::MQTTSensorComponent`

Public Functions

```
mqtt::MQTTComponent::MQTTComponent()
```

Constructs a *MQTTComponent*.

```
void mqtt::MQTTComponent::setup_()
```

Override `setup_` so that we can call `send_discovery()` when needed.

```
void mqtt::MQTTComponent::loop_()
```

Public `loop()` functions.

These will be called by the *Application* instance.

Note: This should normally not be overridden, unless you know what you’re doing. They’re basically to make creating custom components easier. For example the SensorComponent can override these methods to not have the user call some super methods within their custom sensors. These methods should ALWAYS call the `loop_internal()` and `setup_internal()` methods.

Basically, it handles stuff like interval/timeout functions and eventually calls `loop()`.

```
virtual void mqtt::MQTTComponent::send_discovery(JsonBuffer &buffer, JsonObject &root,
                                                SendDiscoveryConfig &config) = 0
```

Send discovery info to the Home Assistant, override this.

```
void mqtt::MQTTComponent::set_retain(bool retain)
```

Set whether state message should be retained.

```
bool mqtt::MQTTComponent::get_retain() const
```

```
void mqtt::MQTTComponent::disable_discovery()
```

Disable discovery. Sets friendly name to “”.

```
bool mqtt::MQTTComponent::is_discovery_enabled() const
```

```
virtual std::string mqtt::MQTTComponentcomponent_type() const = 0
    Override this method to return the component type (e.g. "light", "sensor", ...)

void mqtt::MQTTComponentset_custom_state_topic(const std::string &custom_state_topic)
    Set a custom state topic. Set to "" for default behavior.

void mqtt::MQTTComponentset_custom_command_topic(const std::string &custom_command_topic)
    Set a custom command topic. Set to "" for default behavior.

void mqtt::MQTTComponentset_custom_topic(const std::string &key, const std::string &custom_topic)
float mqtt::MQTTComponentget_setup_priority() const
    MQTT_COMPONENT setup priority.

void mqtt::MQTTComponentset_availability(std::string topic, std::string payload_available,
                                         std::string payload_not_available)
    Set the Home Assistant availability data.

See See Home Assistant for more info.

void mqtt::MQTTComponentdisable_availability()
```

Protected Functions

```
std::string mqtt::MQTTComponentget_discovery_topic(const MQTTDiscoveryInfo &discovery_info) const
    Helper method to get the discovery topic for this component.
```

```
std::string mqtt::MQTTComponentget_default_topic_for(const std::string &suffix) const
    Get this components state/command/...
topic.
```

Return The full topic.

Parameters

- **suffix:** The suffix/key such as "state" or "command".

```
virtual std::string mqtt::MQTTComponentfriendly_name() const = 0
    Get the friendly name of this MQTT component.
```

```
const std::string mqtt::MQTTComponentget_state_topic() const
    Get the MQTT topic that new states will be shared to.
```

```
const std::string mqtt::MQTTComponentget_command_topic() const
    Get the MQTT topic for listening to commands.
```

```
const std::string mqtt::MQTTComponentget_topic_for(const std::string &key) const
    Get the MQTT topic for a specific suffix/key, if a custom topic has been defined, that one will be used.

Otherwise, one will be generated with get\_default\_topic\_for\(\).
```

```
void mqtt::MQTTComponentsend_discovery_()
    Internal method to start sending discovery info, this will call send\_discovery\(\).
```

```
void mqtt::MQTTComponent::send_message(const std::string &topic, const std::string &payload, const optional<uint8_t> &qos = {}, const optional<bool> &retain = {})
```

Send a MQTT message.

Parameters

- **topic:** The topic.
- **payload:** The payload.
- **retain:** Whether to retain the message. If not set, defaults to get_retain.

```
void mqtt::MQTTComponent::send_json_message(const std::string &topic, const json_build_t &f, const optional<uint8_t> &qos = {}, const optional<bool> &retain = {})
```

Construct and send a JSON MQTT message.

Parameters

- **topic:** The topic.
- **f:** The Json Message builder.
- **retain:** Whether to retain the message. If not set, defaults to get_retain.

```
void mqtt::MQTTComponent::subscribe(const std::string &topic, mqtt_callback_t callback, uint8_t qos = 0)
```

Subscribe to a MQTT topic.

Parameters

- **topic:** The topic. Wildcards are currently not supported.
- **callback:** The callback that will be called when a message with matching topic is received.
- **qos:** The MQTT quality of service. Defaults to 0.

```
void mqtt::MQTTComponent::subscribe_json(const std::string &topic, json_parse_t callback, uint8_t qos = 0)
```

Subscribe to a MQTT topic and automatically parse JSON payload.

If an invalid JSON payload is received, the callback will not be called.

Parameters

- **topic:** The topic. Wildcards are currently not supported.
- **callback:** The callback with a parsed JsonObject that will be called when a message with matching topic is received.
- **qos:** The MQTT quality of service. Defaults to 0.

```
std::string mqtt::MQTTComponent::get_default_object_id() const
```

Generate the Home Assistant MQTT discovery object id by automatically transforming the friendly name.

Protected Attributes

```
std::map<std::string, std::string> mqtt::MQTTComponentcustom_topics_ = {}
bool mqtt::MQTTComponentretain_ = {true}
bool mqtt::MQTTComponentdiscovery_enabled_ = {true}
Availability *mqtt::MQTTComponentavailability_ = {nullptr}
bool mqtt::MQTTComponentnext_send_discovery_ = {true}

struct mqtt::SendDiscoveryConfig
    Simple Helper struct used for Home Assistant MQTT send_discovery().
```

Public Members

```
bool mqtt::SendDiscoveryConfigstate_topic = {true}
    If the state topic should be included. Defaults to true.
bool mqtt::SendDiscoveryConfigcommand_topic = {true}
    If the command topic should be included. Default to true.
const char *mqtt::SendDiscoveryConfigplatform = {"mqtt"}
    The platform of this component. Defaults to "mqtt".
```

Over-The-Air Updates

Example Usage

```
// Setup basic OTA
App.init_ota();
// Enable safe mode.
App.init_ota()->start_safe_mode();
// OTA password
auto *ota = App.init_ota();
ota->set_auth_plaintext_password("VERY_SECURE");
ota->start_safe_mode();
// OTA MD5 password
auto *ota = App.init_ota();
ota->set_auth_password_hash("761d3a8c46989f1d357842e8dedf7712");
ota->start_safe_mode();
```

API Reference

OTACOMPONENT

```
class OTACOMPONENT
```

OTACOMPONENT provides a simple way to integrate Over-the-Air updates into your app using ArduinoOTA.

Inherits from *Component*

Public Functions

`OTACOMPONENT(OTACOMPONENT(uint16_t port = OTA_DEFAULT_PORT, std::string hostname
= ""))`

Construct an `OTACOMPONENT`.

Defaults to no authentication.

Parameters

- `port`: The port ArduinoOTA will listen on.
- `hostname`: The hostname ArduinoOTA will advertise.

`void OTACOMPONENTset_auth_open()`

Set ArduinoOTA to accept updates without authentication.

`void OTACOMPONENTset_auth_plaintext_password(const std::string &password)`

Set a plaintext password that ArduinoOTA will use for authentication.

Note: theoretically this password can be read from ROM by an intruder.

Parameters

- `password`: The plaintext password.

`void OTACOMPONENTset_auth_password_hash(const std::string &hash)`

Set a MD5 password hash that ArduinoOTA will use for authentication.

Parameters

- `hash`: The MD5 hash of the password.

`void OTACOMPONENTset_port(uint16_t port)`

Manually set the port OTA should listen on.

`void OTACOMPONENTset_hostname(const std::string &hostname)`

Set the hostname advertised with mDNS. Empty for default hostname.

`void OTACOMPONENTstart_safe_mode(uint8_t num_attempts = 10, uint32_t enable_time =
120000)`

Start OTA safe mode.

When called at startup, this method will automatically detect boot loops.

If a boot loop is detected (by `num_attempts` boots which each lasted less than `enable_time`), this method will block startup and enable just WiFi and OTA so that users can flash a new image.

When in boot loop safe mode, if no OTA attempt is made within `enable_time` milliseconds, the device is restarted. If the device has stayed on for more than `enable_time` milliseconds, the boot is considered successful and the `num_attempts` counter is reset.

Parameters

- `num_attempts`: The number of failed boot attempts until which time safe mode should be enabled.
- `enable_time`: The time in ms safe mode should be on before restarting.

```
void OTACOMPONENTsetup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing nothing.

```
float OTACOMPONENTget_setup_priority() const
```

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void OTACOMPONENTloop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

```
const std::string &OTACOMPONENTget_hostname() const
```

```
uint16_t OTACOMPONENTget_port() const
```

```
void OTACOMPONENTclean_rtc()
```

Protected Types

```
enum [anonymous] __anonymous1
```

Values:

```
OTACOMPONENTOPEN
```

```
OTACOMPONENTPLAINTEXT
```

```
OTACOMPONENTHASH
```

Protected Functions

```
void OTACOMPONENTwrite_rtc_(uint8_t val)
```

```
uint8_t OTACOMPONENTread_rtc_()
```

Protected Attributes

```
OTACOMPONENT::@1 OTACOMPONENT::auth_type_
```

```
std::string OTACOMPONENTpassword_
```

```
uint16_t OTACOMPONENTport_
```

```
std::string OTACOMPONENThostname_
```

```
WiFiServer *OTACOMPONENTserver_
```

```
bool OTACOMPONENTota_triggered_ = {false}
```

stores whether OTA is currently active.

```

bool OTACComponenthas_safe_mode_ = {false}
    stores whether safe mode can be enabled.

uint32_t OTACComponentsafe_mode_start_time_
    stores when safe mode was enabled.

uint32_t OTACComponentsafe_mode_enable_time_ = {60000}
    The time safe mode should be on for.

uint8_t OTACComponentsafe_mode_rtc_value_
uint8_t OTACComponentsafe_mode_num_attempts_
uint8_t OTACComponentat_ota_progress_message_ = {0}
    store OTA progress message index so that we don't spam logs

```

Helpers

esphomelib uses a bunch of helpers to make the library easier to use.

API Reference

helpers.h

Typedefs

```

using json_parse_t = std::function<void(JsonObject&)>
    Callback function typedef for parsing JsonObjects.

using json_build_t = std::function<void(JsonBuffer&, JsonObject&)>
    Callback function typedef for building JsonObjects.

```

Functions

```

std::string get_mac_address()
    Gets the MAC address as a string, this can be used as way to identify this ESP32.

std::string generate_hostname(const std::string &base)
    Constructs a hostname by concatenating base, a hyphen, and the MAC address.

std::string sanitize_hostname(const std::string &hostname)
    Sanitize the hostname by removing characters that are not in the whitelist and truncating it to 63
    chars.

std::string truncate_string(const std::string &s, size_t length)
    Truncate a string to a specific length.

bool is_empty(const IPAddress &address)
    Checks whether the provided IPAddress is empty (is 0.0.0.0).

void reboot(const char *cause)
    Force a shutdown (and reboot) of the ESP, calling any registered shutdown hooks.

void add_shutdown_hook(std::function<void>const char *> &&fAdd a shutdown callback.

```

```
void safe_reboot(const char *cause)
Create a safe shutdown (and reboot) of the ESP, calling any registered shutdown and safe shutdown hooks.
```

```
void run_shutdown_hooks(const char *cause)
Run shutdown hooks.
```

```
void add_safe_shutdown_hook(std::function<void> const char *
> &&f)Add a safe shutdown callback that will be called if the device is shut down intentionally.
```

```
void run_safe_shutdown_hooks(const char *cause)
Run safe shutdown and force shutdown hooks.
```

```
std::string to_lowercase_underscore(std::string s)
Convert the string to lowercase_underscore.
```

```
std::string build_json(const json_build_t &f)
Build a JSON string with the provided json build function.
```

```
void parse_json(const std::string &data, const json_parse_t &f)
Parse a JSON string and run the provided json parse function if it's valid.
```

```
template <typename T>
T clamp(T min, T max, T val)
Clamp the value between min and max.
```

Return val clamped in between min and max.

Template Parameters

- T: The input/output typename.

Parameters

- min: The minimum value.
- max: The maximum value.
- val: The value.

```
template <typename T>
T lerp(T start, T end, T completion)
Linearly interpolate between end start and end by completion.
```

Return The linearly interpolated value.

Template Parameters

- T: The input/output typename.

Parameters

- start: The start value.
- end: The end value.
- completion: The completion. 0 is start value, 1 is end value.

```
template <typename T, typename ... Args>
std::unique_ptr<T> make_unique(Args&&... args)
std::make_unique
```

```
uint32_t random_uint32()
Return a random 32 bit unsigned integer.
```

```
double random_double()
    Returns a random double between 0 and 1.

    Note: This function probably doesn't provide a truly uniform distribution.

float random_float()
    Returns a random float between 0 and 1. Essentially just casts random_double() to a float.

float gamma_correct(float value, float gamma)
    Applies gamma correction with the provided gamma to value.

std::string value_accuracy_to_string(float value, int8_t accuracy_decimals)
    Create a string from a value and an accuracy in decimals.

std::string uint64_to_string(uint64_t num)
    Convert a uint64_t to a hex string.

std::string uint32_to_string(uint32_t num)
    Convert a uint32_t to a hex string.

std::string sanitize_string_whitelist(const std::string &s, const std::string &whitelist)
    Sanitizes the input string with the whitelist.

void disable_interrupts()
    Cross-platform method to disable interrupts.

    Useful when you need to do some timing-dependent communication.
```

See Do not forget to call `enable_interrupts()` again or otherwise things will go very wrong.

```
void enable_interrupts()
    Cross-platform method to enable interrupts after they have been disabled.

uint8_t crc8(uint8_t *data, uint8_t len)
    Calculate a crc8 of data with the provided data length.

optional<bool> parse_on_off(const char *str, const char *payload_on = "on", const char *payload_off = "off")
```

Variables

```
const char *HOSTNAME_CHARACTER_WHITELIST
    The characters that are allowed in a hostname.

CallbackManager<void(const char *)> shutdown_hooks
CallbackManager<void(const char *)> safe_shutdown_hooks
template <typename T>
class SlidingWindowMovingAverage
    #include <helpers.h> Helper class that implements a sliding window moving average.
```

Public Functions

SlidingWindowMovingAverage `SlidingWindowMovingAverage(size_t max_size)`
Create the *SlidingWindowMovingAverage*.

Parameters

- `max_size`: The window size.

T *SlidingWindowMovingAverage***next_value**(T *value*)
Add value to the interval buffer.

Return The new average.

Parameters

- **value:** The value.

T *SlidingWindowMovingAverage***calculate_average**()
Return the average across the sliding window.

size_t *SlidingWindowMovingAverage***get_max_size**() const
void *SlidingWindowMovingAverage***set_max_size**(size_t *max_size*)

Protected Attributes

```
std::queue<T> SlidingWindowMovingAveragequeue_
size_t SlidingWindowMovingAveragemax_size_
T SlidingWindowMovingAveragesum_
class ExponentialMovingAverage
#include <helpers.h> Helper class that implements an exponential moving average.
```

Public Functions

```
ExponentialMovingAverageExponentialMovingAverage(float alpha)
float ExponentialMovingAveragenext_value(float value)
float ExponentialMovingAveragecalculate_average()
void ExponentialMovingAverageset_alpha(float alpha)
float ExponentialMovingAverageget_alpha() const
```

Protected Attributes

```
float ExponentialMovingAveragealpha_
float ExponentialMovingAverageaccumulator_
template <typename... Ts>
template<>
class CallbackManager<void(Ts...)>
#include <helpers.h> Simple helper class to allow having multiple subscribers to a signal.
```

Template Parameters

- **Ts:** The arguments for the callback, wrapped in void().

Public Functions

```
void CallbackManageradd(std::function<void> Ts... > &&callbackAdd a callback to the internal callback list.
```

```
void CallbackManagercall(Ts... args) Call all callbacks in this manager.
```

Protected Attributes

```
std::vector<std::function<void(Ts...)>> CallbackManagercallbacks_
template <typename T, typename X>
class TemplatableValue
```

Public Functions

```
TemplatableValueTemplatableValue()
```

```
TemplatableValueTemplatableValue(T const &value)
```

```
TemplatableValueTemplatableValue(std::function<T>X > f
```

```
bool TemplatableValuehas_value()
```

```
T TemplatableValuevalue(X x)
```

Protected Types

```
enum [anonymous]
```

Values:

EMPTY

VALUE

LAMBDA

Protected Attributes

```
TemplatableValue::@0 TemplatableValue::type_
```

```
T TemplatableValuevalue_
```

```
std::function<T(X)> TemplatableValuef_
```

optional.h

Functions

```
const nullopt_t nullopt((nullopt_t::init()))
template <typename T, typename U>
```

```
bool operator==(optional<T> const &x, optional<U> const &y)
template <typename T, typename U>
bool operator!=(optional<T> const &x, optional<U> const &y)
template <typename T, typename U>
bool operator<(optional<T> const &x, optional<U> const &y)
template <typename T, typename U>
bool operator>(optional<T> const &x, optional<U> const &y)
template <typename T, typename U>
bool operator<=(optional<T> const &x, optional<U> const &y)
template <typename T, typename U>
bool operator>=(optional<T> const &x, optional<U> const &y)
template <typename T>
bool operator==(optional<T> const &x, nullopt_t)
template <typename T>
bool operator==(nullopt_t, optional<T> const &x)
template <typename T>
bool operator!=(optional<T> const &x, nullopt_t)
template <typename T>
bool operator!=(nullopt_t, optional<T> const &x)
template <typename T>
bool operator<(optional<T> const&, nullopt_t)
template <typename T>
bool operator<(nullopt_t, optional<T> const &x)
template <typename T>
bool operator<=(optional<T> const &x, nullopt_t)
template <typename T>
bool operator<=(nullopt_t, optional<T> const&)
template <typename T>
bool operator>(optional<T> const &x, nullopt_t)
template <typename T>
bool operator>(nullopt_t, optional<T> const&)
template <typename T>
bool operator>=(optional<T> const&, nullopt_t)
template <typename T>
bool operator>=(nullopt_t, optional<T> const &x)
template <typename T, typename U>
bool operator==(optional<T> const &x, U const &v)
template <typename T, typename U>
bool operator==(U const &v, optional<T> const &x)
template <typename T, typename U>
bool operator!=(optional<T> const &x, U const &v)
template <typename T, typename U>
bool operator!=(U const &v, optional<T> const &x)
template <typename T, typename U>
bool operator<(optional<T> const &x, U const &v)
template <typename T, typename U>
bool operator<(U const &v, optional<T> const &x)
template <typename T, typename U>
bool operator<=(optional<T> const &x, U const &v)
template <typename T, typename U>
bool operator<=(U const &v, optional<T> const &x)
template <typename T, typename U>
bool operator>(optional<T> const &x, U const &v)
template <typename T, typename U>
```

```

bool operator>(U const &v, optional<T> const &x)
template <typename T, typename U>
bool operator>=(optional<T> const &x, U const &v)
template <typename T, typename U>
bool operator>=(U const &v, optional<T> const &x)
template <typename T>
void swap(optional<T> &x, optional<T> &y)
template <typename T>
optional<T> make_optional(T const &v)

struct nullopt_t

```

Public Functions

```

nullopt_t nullopt_t::init()
template <typename T>
class optional

```

Public Types

```
typedef T optional::value_type
```

Public Functions

```

optional::optional()
optional::optional(nullopt_t)
optional::optional(T const &arg)
template <class U>
optional::optional(optional<U> const &other)

optional &optional::operator=(nullopt_t)
template <class U>
optional &optional::operator=(optional<U> const &other)

void optional::swap(optional &rhs)

value_type const *optional::operator->() const
value_type *optional::operator->()

value_type const &optional::operator*() const
value_type &optional::operator*()

optional::operator safe_bool() const

bool optional::has_value() const
value_type const &optional::value() const
value_type &optional::value()

```

```
template <class U>
value_type optional<U>::value_or(U const &v) const
void optional::reset()
```

Private Types

```
template<>
typedef void (optional::*optional<T>::safe_bool)() const
```

Private Functions

```
void optional::this_type_does_not_support_comparisons() const
template <typename V>
void optional::initialize(V const &value)
```

Private Members

```
bool optional::has_value_
value_type optional::value_
```

ESPPreferences

class ESPPreferences

Helper class to allow easy access to non-volatile storage like SPIFFS to save preferences.

Public Functions

```
void ESPPreferences::begin(const std::string &name)
    Start the preferences object with the specified app name.

size_t ESPPreferences::put_bool(const std::string &friendly_name, const std::string &key, bool value)
size_t ESPPreferences::put_int8(const std::string &friendly_name, const std::string &key, int8_t value)
size_t ESPPreferences::put_uint8(const std::string &friendly_name, const std::string &key, uint8_t value)
size_t ESPPreferences::put_int16(const std::string &friendly_name, const std::string &key, int16_t value)
size_t ESPPreferences::put_uint16(const std::string &friendly_name, const std::string &key, uint16_t value)
size_t ESPPreferences::put_int32(const std::string &friendly_name, const std::string &key, int32_t value)
size_t ESPPreferences::put_uint32(const std::string &friendly_name, const std::string &key, uint32_t value)
```

```

size_t ESPPreferencesput_int64(const std::string &friendly_name, const std::string &key,
                           int64_t value)
size_t ESPPreferencesput_uint64(const std::string &friendly_name, const std::string &key,
                                 uint64_t value)
size_t ESPPreferencesput_float(const std::string &friendly_name, const std::string &key,
                                float value)
size_t ESPPreferencesput_double(const std::string &friendly_name, const std::string &key,
                                 double value)
bool ESPPreferencesget_bool(const std::string &friendly_name, const std::string &key, bool
                            default_value)
int8_t ESPPreferencesget_int8(const std::string &friendly_name, const std::string &key,
                               int8_t default_value)
uint8_t ESPPreferencesget_uint8(const std::string &friendly_name, const std::string &key,
                                 uint8_t default_value)
int16_t ESPPreferencesget_int16(const std::string &friendly_name, const std::string &key,
                                 int16_t default_value)
uint16_t ESPPreferencesget_uint16(const std::string &friendly_name, const std::string &key,
                                    uint16_t default_value)
int32_t ESPPreferencesget_int32(const std::string &friendly_name, const std::string &key,
                                 int32_t default_value)
uint32_t ESPPreferencesget_uint32(const std::string &friendly_name, const std::string &key,
                                    uint32_t default_value)
int64_t ESPPreferencesget_int64(const std::string &friendly_name, const std::string &key,
                                 int64_t default_value)
uint64_t ESPPreferencesget_uint64(const std::string &friendly_name, const std::string &key,
                                    uint64_t default_value)
float ESPPreferencesget_float(const std::string &friendly_name, const std::string &key, float
                                default_value)
double ESPPreferencesget_double(const std::string &friendly_name, const std::string &key,
                                 double default_value)

```

Protected Functions

```
std::string ESPPreferencesget_preference_key(const std::string &friendly_name, const
                                              std::string &key)
```

Return a key for the nvs storage by hashing the friendly name and truncating the key to 7 characters.

Protected Attributes

Preferences *ESPPreferences***preferences_**

ESPPreferences **global_preferences**

esphal.h

This header should be used whenever you want to access some *digitalRead*, *digitalWrite*, ... methods.

`class GPIOPin`

A high-level abstraction class that can expose a pin together with useful options like `pinMode`.

Set the parameters for this at construction time and use `setup()` to apply them. The `inverted` parameter will automatically invert the input/output for you.

Use `read_value()` and `write_value()` to use `digitalRead()` and `digitalWrite()`, respectively.

Subclassed by `GPIOInputPin`, `GPIOOutputPin`

Public Functions

`GPIOPin::GPIOPin(uint8_t pin, uint8_t mode, bool inverted = false)`

Construct the `GPIOPin` instance.

Parameters

- `pin`: The GPIO pin number of this instance.
- `mode`: The Arduino `pinMode` that this pin should be put into at `setup()`.
- `inverted`: Whether all `digitalRead/digitalWrite` calls should be inverted.

`GPIOPin::GPIOPin()`

Default constructor so that it can be stored in optionals.

`GPIOPin *GPIOPin::copy() const`

`void GPIOPin::setup()`

Setup the pin mode.

`bool GPIOPin::digital_read()`

Read the binary value from this pin using `digitalRead` (and inverts automatically).

`void GPIOPin::digital_write(bool value)`

Write the binary value to this pin using `digitalWrite` (and inverts automatically).

`void GPIOPin::pin_mode(uint8_t mode)`

Set the pin mode.

`void GPIOPin::set_inverted(bool inverted)`

Set the inverted mode of this pin.

`void GPIOPin::set_pin(uint8_t pin)`

Set the GPIO pin number.

`void GPIOPin::set_mode(uint8_t mode)`

Set the `pinMode` of this pin.

`unsigned char GPIOPin::get_pin() const`

Get the GPIO pin number.

`unsigned char GPIOPin::get_mode() const`

Get the `pinMode` of this pin.

```
bool GPIOPinis_inverted() const
    Return whether this pin shall be treated as inverted. (for example active-low)
```

Protected Attributes

```
uint8_t GPIOPinpin_
uint8_t GPIOPinmode_
bool GPIOPininverted_
```

class *GPIOOutputPin*

Basically just a *GPIOPin*, but defaults to OUTPUT pinMode.

Note that theoretically you can still assign an INPUT pinMode to this - we intentionally don't check this.

The constructor also allows for easy usage because of it's lack of "explicit" constructor.

Inherits from *GPIOPin*

Subclassed by *io::PCF8574GPIOOutputPin*

Public Functions

```
GPIOOutputPinGPIOOutputPin()
```

```
GPIOOutputPinGPIOOutputPin(uint8_t pin, uint8_t mode = OUTPUT, bool inverted = false)
```

class *GPIOInputPin*

Basically just a *GPIOPin*, but defaults to INPUT pinMode.

Note that theoretically you can still assign an OUTPUT pinMode to this - we intentionally don't check this.

The constructor also allows for easy usage because of it's lack of "explicit" constructor.

Inherits from *GPIOPin*

Subclassed by *io::PCF8574GPIOInputPin*

Public Functions

```
GPIOInputPinGPIOInputPin()
```

```
GPIOInputPinGPIOInputPin(uint8_t pin, uint8_t mode = INPUT, bool inverted = false)
```

ESPOneWire

esphomelib has its own implementation of OneWire, because the implementation in the Arduino libraries seems to have lots of timing issues with the ESP8266/ESP32. That's why ESPOneWire was created.

class *ESPOneWire*

This is esphomelib's own (minimal) implementation of 1-Wire that improves timing for ESP boards.

It's more or less the same as Arduino's internal library but uses some fancy C++ and 64 bit unsigned integers to make our lives easier.

Public Functions

ESPHOMEINCLUDE_Namespace_BEGIN *ESPOneWire*
ESPOneWire(*GPIOPin* **pin*)

Construct a OneWire instance for the specified pin. There should only exist one instance per pin.

bool *ESPOneWire***reset**()

Reset the bus, should be done before all write operations.

Takes approximately 1ms.

Return Whether the operation was successful.

void *ESPOneWire***write_bit**(bool *bit*)

Write a single bit to the bus, takes about 70µs.

bool *ESPOneWire***read_bit**()

Read a single bit from the bus, takes about 70µs.

void *ESPOneWire***write8**(uint8_t *val*)

Write a word to the bus. LSB first.

void *ESPOneWire***write64**(uint64_t *val*)

Write a 64 bit unsigned integer to the bus. LSB first.

void *ESPOneWire***skip**()

Write a command to the bus that addresses all devices by skipping the ROM.

uint8_t *ESPOneWire***read8**()

Read an 8 bit word from the bus.

uint64_t *ESPOneWire***read64**()

Read an 64-bit unsigned integer from the bus.

void *ESPOneWire***select**(uint64_t *address*)

Select a specific address on the bus for the following command.

void *ESPOneWire***reset_search**()

Reset the device search.

uint64_t *ESPOneWire***search**()

Search for a 1-Wire device on the bus. Returns 0 if all devices have been found.

std::vector<uint64_t> *ESPOneWire***search_vec**()

Helper that wraps search in a std::vector.

Protected Functions

uint8_t **ESPOneWire***rom_number8**()

Helper to get the internal 64-bit unsigned rom number as a 8-bit integer pointer.

Protected Attributes

```
GPIOPin *ESPOneWirepin_
uint8_t ESPOneWirelast_discrepancy_ = {0}
uint8_t ESPOneWirelast_family_discrepancy_ = {0}
bool ESPOneWirelast_device_flag_ = {false}
uint64_t ESPOneWirerom_number = {0}
```

defines.h

Logging Engine

esphomelib will by default log to both Serial (with baudrate 115200).

API Reference

LogComponent

class LogComponent

A simple component that enables logging to Serial via the ESP_LOG* macros.

This component should optimally be setup very early because only after its setup log messages are actually sent. To do this, simply call [pre_setup\(\)](#) as early as possible.

Inherits from [Component](#)

Public Functions

```
LogComponent::LogComponent(uint32_t baud_rate = 11520, size_t tx_buffer_size = 512)
Construct the LogComponent.
```

Parameters

- **baud_rate**: The baud_rate for the serial interface. 0 to disable UART logging.
- **tx_buffer_size**: The buffer size (in bytes) used for constructing log messages.

```
void LogComponent::set_baud_rate(uint32_t baud_rate)
Manually set the baud rate for serial, set to 0 to disable.
```

```
void LogComponent::set_tx_buffer_size(size_t tx_buffer_size)
Set the buffer size that's used for constructing log messages. Log messages longer than this will be truncated.
```

```
void LogComponent::set_global_log_level(int log_level)
Set the global log level. Note: Use the ESPHOMELIB_LOG_LEVEL define to also remove the logs from the build.
```

```
void LogComponent::set_log_level(const std::string &tag, int log_level)
Set the log level of the specified tag.
```

```
void LogComponentpre_setup()
    Set up this component.

uint32_t LogComponentget_baud_rate() const
size_t LogComponentget_tx_buffer_size() const

int LogComponentlog_vprintf_(int level, const char *tag, const char *format, va_list args)

void LogComponentadd_on_log_callback(std::function<void(int, const char * > &&callbackRegister a callback that will be called for every log message sent.
```

Protected Attributes

```
uint32_t LogComponentbaud_rate_
std::vector<char> LogComponenttx_buffer_
int LogComponentglobal_log_level_ = {ESPHOMELOGLEVEL}
std::unordered_map<std::string, int> LogComponentlog_levels_
CallbackManager<void(int, const char *)> LogComponentlog_callback_ = {}
```

Power Supply

Example Usage

```
// Basic
auto *power_supply = App.make_power_supply(12);
// Inverted, for ATX
auto *atx = App.make_power_supply(GPIO0OutputPin(12, OUTPUT, true));
```

API Reference

PowerSupplyComponent

```
class PowerSupplyComponent
```

This class represents an power supply.

The power supply will automatically be turned on if a component requests high power and will automatically be turned off again keep_on_time (ms) after the last high-power request is cancelled. Additionally, an enable_time (ms) can be specified because many power supplies only actually provide high-power output after a few milliseconds.

Use the pin argument of the *Application* helper to enable inverted mode. For example most ATX power supplies operate in inverted mode, so to turn them on you have to pull the pin LOW.

To request high power mode, a component must use the *request_high_power()* function to register itself as needing high power mode. Once the high power mode is no longer required the component can use *unrequest_high_power()* to unregister its high power mode. IMPORTANT: An component should NOT hold multiple requests to the same power supply, as the *PowerSupplyComponent* only holds an internal counter of how many high power requests have been made.

Usually though, all this should actually be handled by `BinaryOutput` and `FloatOutput`, since using this class correctly is not too easy.

Inherits from `Component`

Public Functions

```
PowerSupplyComponentPowerSupplyComponent(GPIOPin *pin, uint32_t enable_time = 20,
                                         uint32_t keep_on_time = 10000)
```

Creates the `PowerSupplyComponent`.

Parameters

- `pin`: The pin of the power supply control wire.
- `enable_time`: The time in milliseconds the power supply requires for power up. The thread will block in the meantime
- `keep_on_time`: The time in milliseconds the power supply should be kept on after the last high-power request.

```
void PowerSupplyComponentset_keep_on_time(uint32_t keep_on_time)
```

Set the time in milliseconds the power supply should be kept on for after the last high-power request.

```
void PowerSupplyComponentset_enable_time(uint32_t enable_time)
```

Set the time in milliseconds the power supply needs for power-up.

```
bool PowerSupplyComponentis_enabled() const
```

Is this power supply currently on?

```
void PowerSupplyComponentrequest_high_power()
```

Request high power mode. Use `unrequest_high_power()` to remove this request.

```
void PowerSupplyComponentunrequest_high_power()
```

Un-request high power mode.

```
void PowerSupplyComponentsetup()
```

Register callbacks.

```
float PowerSupplyComponentget_setup_priority() const
```

Hardware setup priority (+1).

```
uint32_t PowerSupplyComponentget_keep_on_time() const
```

Get the keep on time.

```
uint32_t PowerSupplyComponentget_enable_time() const
```

Get the enable time.

Protected Attributes

```
GPIOPin *PowerSupplyComponentpin_
```

```
bool PowerSupplyComponentenabled_ = {false}
```

```
uint32_t PowerSupplyComponentenable_time_
```

```
uint32_t PowerSupplyComponentkeep_on_time_
int16_t PowerSupplyComponentactive_requests_ = {0}
```

Controller

API Reference

Controller

class Controller

Controllers allow an object to be notified of every component that's added to the *Application*.

Subclassed by *StoringController*

Public Functions

```
ESPHOMELIB_NAMESPACE_BEGIN void Controller::register_binary_sensor(binary_sensor::BinarySensor *obj)
void Controllerregister_fan(fan::FanState *obj)
void Controllerregister_light(light::LightState *obj)
void Controllerregister_sensor(sensor::Sensor *obj)
void Controllerregister_switch(switch_::Switch *obj)
void Controllerregister_cover(cover::Cover *cover)
```

StoringController

class StoringController

A *StoringController* is a controller that automatically stores all components internally in vectors.

Inherits from *Controller*

Subclassed by *WebServer*

Public Functions

```
void StoringControllerregister_binary_sensor(binary_sensor::BinarySensor *obj)
void StoringControllerregister_fan(fan::FanState *obj)
void StoringControllerregister_light(light::LightState *obj)
void StoringControllerregister_sensor(sensor::Sensor *obj)
void StoringControllerregister_switch(switch_::Switch *obj)
void StoringControllerregister_cover(cover::Cover *cover)
```

Protected Attributes

```
std::vector<binary_sensor::BinarySensor *> StoringControllerbinary_sensors_
std::vector<fan::FanState *> StoringControllerfans_
std::vector<light::LightState *> StoringControllerlights_
std::vector<sensor::Sensor *> StoringControllersensors_
std::vector<switch_::Switch *> StoringControllerswitches_
std::vector<cover::Cover *> StoringControllercovers_
```

Web Server

API Reference

`class WebServer`

This class allows users to create a web server with their ESP nodes.

Behind the scenes it's using AsyncWebServer to set up the server. It exposes 3 things: an index page under '/' that's used to show a simple web interface (the css/js is hosted by esphomelib.com by default), an event source under '/events' that automatically sends all state updates in real time + the debug log. Lastly, there's an REST API available under the '/light/...', '/sensor/...', ... URLs. A full documentation for this API can be found under <https://esphomelib.com/web-api/index.html>.

Additionally, the web server is advertised via mDNS.

Inherits from *StoringController*, *Component*, *AsyncWebHandler*

Public Functions

`WebServer::WebServer(uint16_t port)`

Initialize the web server with the specified port.

`void WebServer::set_css_url(const char *css_url)`

Set the URL to the CSS <link> that's sent to each client.

Defaults to https://esphomelib.com/_static/webserver-v1.min.css

Parameters

- `css_url`: The url to the web server stylesheet.

`void WebServer::set_js_url(const char *js_url)`

Set the URL to the script that's embedded in the index page.

Defaults to https://esphomelib.com/_static/webserver-v1.min.js

Parameters

- `js_url`: The url to the web server script.

`void WebServer::set_port(uint16_t port)`

Set the web server port.

```
void WebServersetup()  
    Setup the internal web server and register handlers.  
  
float WebServerget_setup_priority() const  
    MQTT setup priority.  
  
void WebServerhandle_index_request(AsyncWebServerRequest *request)  
    Handle an index request under '/'.  
  
void WebServerregister_sensor(sensor::Sensor *obj)  
    Internally register a sensor and set a callback on state changes.  
  
void WebServerhandle_sensor_request(AsyncWebServerRequest *request, UrlMatch match)  
    Handle a sensor request under '/sensor/<id>'.  
  
std::string WebServersensor_json(sensor::Sensor *obj, float value)  
    Dump the sensor state with its value as a JSON string.  
  
void WebServerregister_switch(switch_::Switch *obj)  
    Internally register a switch and set a callback on state changes.  
  
void WebServerhandle_switch_request(AsyncWebServerRequest *request, UrlMatch match)  
    Handle a switch request under '/switch/<id>/</turn_on/turn_off/toggle>'.  
  
std::string WebServerswitch_json(switch_::Switch *obj, bool value)  
    Dump the switch state with its value as a JSON string.  
  
void WebServerregister_binary_sensor(binary_sensor::BinarySensor *obj)  
    Internally register a binary sensor and set a callback on state changes.  
  
void WebServerhandle_binary_sensor_request(AsyncWebServerRequest *request, UrlMatch match)  
    Handle a binary sensor request under '/binary_sensor/<id>'.  
  
std::string WebServerbinary_sensor_json(binary_sensor::BinarySensor *obj, bool value)  
    Dump the binary sensor state with its value as a JSON string.  
  
void WebServerregister_fan(fan::FanState *obj)  
    Internally register a fan and set a callback on state changes.  
  
void WebServerhandle_fan_request(AsyncWebServerRequest *request, UrlMatch match)  
    Handle a fan request under '/fan/<id>/</turn_on/turn_off/toggle>'.  
  
std::string WebServerfan_json(fan::FanState *obj)  
    Dump the fan state as a JSON string.  
  
void WebServerregister_light(light::LightState *obj)  
    Internally register a light and set a callback on state changes.  
  
void WebServerhandle_light_request(AsyncWebServerRequest *request, UrlMatch match)  
    Handle a light request under '/light/<id>/</turn_on/turn_off/toggle>'.  
  
std::string WebServerlight_json(light::LightState *obj)  
    Dump the light state as a JSON string.  
  
bool WebServercanHandle(AsyncWebServerRequest *request)  
    Override the web handler's canHandle method.
```

```
void WebServerhandleRequest(AsyncWebServerRequest *request)
    Override the web handler's handleRequest method.
```

```
bool WebServerisRequestHandlerTrivial()
    This web handle is not trivial.
```

Protected Attributes

```
uint16_t WebServerport_
AsyncWebServer *WebServerserver_
AsyncEventSource WebServerevents_ = {"/events"}
const char *WebServercss_url_ = {nullptr}
const char *WebServerjs_url_ = {nullptr}

struct UrlMatch
Internal helper struct that is used to parse incoming URLs.
```

Public Members

```
std::string UrlMatchdomain
The domain of the component, for example "sensor".
std::string UrlMatchid
The id of the device that's being accessed, for example "living_room_fan".
std::string UrlMatchmethod
The method that's being called, for example "turn_on".
bool UrlMatchvalid
Whether this match is valid.
```

Deep Sleep

API Reference

DeepSleepComponent

```
class DeepSleepComponent
This component allows setting up the node to go into deep sleep mode to conserve battery.

To set this component up, first set when the deep sleep should trigger using set_run_cycles and set_run_duration, then set how long the deep sleep should last using set_sleep_duration and optionally on the ESP32 set_wakeup_pin.
```

Inherits from *Component*

Public Functions

```
void DeepSleepComponentset_sleep_duration(uint32_t time_ms)
Set the duration in ms the component should sleep once it's in deep sleep mode.
```

```
void DeepSleepComponentset_wakeup_pin(GPIOInputPin pin)
```

Set the pin to wake up to on the ESP32 once it's in deep sleep mode.

Use the inverted property to set the wakeup level.

```
void DeepSleepComponentset_wakeup_pin_mode(WakeupPinMode wakeup_pin_mode)
```

```
void DeepSleepComponentset_run_cycles(uint32_t cycles)
```

Set the number of loop cycles after which the node should go into deep sleep mode.

```
void DeepSleepComponentset_run_duration(uint32_t time_ms)
```

Set a duration in ms for how long the code should run before entering deep sleep mode.

```
void DeepSleepComponentsetup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing nothing.

```
void DeepSleepComponentloop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

```
float DeepSleepComponentget_loop_priority() const
```

priority of `loop()`.

higher -> executed earlier

Defaults to 0.

Return The loop priority of this component

```
float DeepSleepComponentget_setup_priority() const
```

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

Protected Functions

```
void DeepSleepComponentbegin_sleep()
```

Helper to enter deep sleep mode.

Protected Attributes

```
optional<uint64_t> DeepSleepComponentsleep_duration_
```

```
optional<GPIOInputPin> DeepSleepComponentwakeup_pin_
```

```
WakeupPinMode DeepSleepComponentwakeup_pin_mode_ = {WAKEUP_PIN_MODE_IGNORE}
```

```
bool DeepSleepComponentnext_enter_deep_sleep_ = {false}
```

```
optional<uint32_t> DeepSleepComponentloop_cycles_
```

```
uint32_t DeepSleepComponentat_loop_cycle_ = {0}
optional<uint32_t> DeepSleepComponentrun_duration_
```

I2CComponent

To make i2c devices easier to implement in esphomelib, there's a special I2CComponent implementing a bunch of i2c helper functions on top of the Arduino Wire library. It is also the preferred way of using i2c peripherals since it implements timeouts, verbose logs for debugging issues, and for the ESP32 the ability to have multiple i2c busses in operation at the same time.

API Reference

`class I2CComponent`

The `I2CComponent` is the base of esphomelib's i2c communication.

It handles setting up the bus (with pins, clock frequency) and provides nice helper functions to make reading from the i2c bus easier (see `read_bytes`, `write_bytes`) and safe (with read timeouts).

For the user, it has a few setters (see `set_sda_pin`, `set_scl_pin`, `set_frequency`, `set_receive_timeout`) to setup some parameters for the bus. Additionally, the i2c component has a scan feature that will scan the entire 7-bit i2c address range for devices that respond to transmissions to make finding the address of an i2c device easier.

On the ESP32, you can even have multiple I2C bus for communication, simply create multiple I2CComponents, each with different SDA and SCL pins and use `set_parent` on all I2CDevices that use the non-first I2C bus.

Inherits from `Component`

Public Functions

`I2CComponent(I2CComponent(uint8_t sda_pin, uint8_t scl_pin, bool scan = false)`

`void I2CComponentset_sda_pin(uint8_t sda_pin)`
Set the i2c SDA pin for this bus.

`void I2CComponentset_scl_pin(uint8_t scl_pin)`
Set the i2c SCL pin for this bus.

`void I2CComponentset_frequency(uint32_t frequency)`
Set the i2c clock frequency in Hz for this bus, defaults to 1000 Hz.

`void I2CComponentset_scan(bool scan)`
Set if a scan of the entire i2c address range should be done on startup.

`void I2CComponentset_receive_timeout(uint32_t receive_timeout)`
Set the timeout in ms for receiveing data, defaults to 100ms.

`bool I2CComponentread_bytes(uint8_t address, uint8_t register_, uint8_t *data, uint8_t len,`
`uint32_t conversion = 0)`
Read len amount of bytes from a register into data.

Optionally with a conversion time after writing the register value to the bus.

Return If the operation was successful.

Parameters

- **address:** The address to send the request to.
- **register_:** The register number to write to the bus before reading.
- **data:** An array to store len amount of 8-bit bytes into.
- **len:** The amount of bytes to request and write into data.
- **conversion:** The time in ms between writing the register value and reading out the value.

```
bool I2CComponentread_bytes_16(uint8_t address, uint8_t register_, uint16_t *data, uint8_t
```

```
len, uint32_t conversion = 0)
```

Read len amount of 16-bit words (MSB first) from a register into data.

Return If the operation was successful.

Parameters

- **address:** The address to send the request to.
- **register_:** The register number to write to the bus before reading.
- **data:** An array to store len amount of 16-bit words into.
- **len:** The amount of 16-bit words to request and write into data.
- **conversion:** The time in ms between writing the register value and reading out the value.

```
bool I2CComponentread_byte(uint8_t address, uint8_t register_, uint8_t *data, uint32_t con-
```

```
version = 0)
```

Read a single byte from a register into the data variable. Return true if successful.

```
bool I2CComponentread_byte_16(uint8_t address, uint8_t register_, uint16_t *data, uint32_t
```

```
conversion = 0)
```

Read a single 16-bit words (MSB first) from a register into the data variable. Return true if successful.

```
bool I2CComponentwrite_bytes(uint8_t address, uint8_t register_, const uint8_t *data,
```

```
uint8_t len)
```

Write len amount of 8-bit bytes to the specified register for address.

Return If the operation was successful.

Parameters

- **address:** The address to use for the transmission.
- **register_:** The register to write the values to.
- **data:** An array from which len bytes of data will be written to the bus.
- **len:** The amount of bytes to write to the bus.

```
bool I2CComponentwrite_bytes_16(uint8_t address, uint8_t register_, const uint16_t *data,
```

```
uint8_t len)
```

Write len amount of 16-bit words (MSB first) to the specified register for address.

Return If the operation was successful.

Parameters

- **address:** The address to use for the transmission.

- `register_`: The register to write the values to.
- `data`: An array from which len 16-bit words of data will be written to the bus.
- `len`: The amount of bytes to write to the bus.

`bool I2CComponentwrite_byte(uint8_t address, uint8_t register_, uint8_t data)`

Write a single byte of data into the specified register of address. Return true if successful.

`bool I2CComponentwrite_byte_16(uint8_t address, uint8_t register_, uint16_t data)`

Write a single 16-bit word of data into the specified register of address. Return true if successful.

`void I2CComponentbegin_transmission_(uint8_t address)`

Begin a write transmission to an address.

`bool I2CComponentend_transmission_(uint8_t address)`

End a write transmission to an address, return true if successful.

`void I2CComponentrequest_from_(uint8_t address, uint8_t len)`

Request data from an address with a number of (8-bit) bytes.

`bool I2CComponentread_(uint8_t address, uint8_t *data)`

Read a single byte from the bus into data.

Return true if no timeout happened.

Note: `request_from_` must be called before this.

`void I2CComponentwrite_(uint8_t address, const uint8_t *data, uint8_t len)`

Write len amount of bytes from data to address. `begin_transmission_` must be called before this.

`void I2CComponentwrite_16_(uint8_t address, const uint16_t *data, uint8_t len)`

Write len amount of 16-bit words from data to address. `begin_transmission_` must be called before this.

`bool I2CComponentreceive_(uint8_t address, uint8_t *data, uint8_t len)`

Request len amount of bytes from address and write the result it into data. Returns true iff was successful.

`bool I2CComponentreceive_16_(uint8_t address, uint16_t *data, uint8_t len)`

Request len amount of 16-bit words from address and write the result into data. Returns true iff was successful.

`void I2CComponentsetup()`

Setup the i2c. bus.

`void I2CComponentloop()`

Do an address range scan if necessary.

`float I2CComponentget_setup_priority() const`

Set a very high setup priority to make sure it's loaded before all other hardware.

Protected Attributes

`TwoWire *I2CComponentwire_`

`uint8_t I2CComponentsda_pin_`

`uint8_t I2CComponentscl_pin_`

```
bool I2CComponentscan_
uint32_t I2CComponentreceive_timeout_ = {100}
uint32_t I2CComponentfrequency_ = {1000}
```

class I2CDevice

All components doing communication on the I2C bus should subclass *I2CDevice*.

This class stores 1. the address of the i2c device and has a helper function to allow users to manually set the address and 2. stores a reference to the “parent” *I2CComponent*.

All this class basically does is to expose all helper functions from *I2CComponent*.

Subclassed by *io::PCF8574Component*, *output::PCA9685OutputComponent*, *sensor::ADS1115Component*, *sensor::BH1750Sensor*, *sensor::BME280Component*, *sensor::BME680Component*, *sensor::BMP085Component*, *sensor::DHT12Component*, *sensor::HDC1080Component*, *sensor::HTU21DComponent*, *sensor::MPU6050Component*, *sensor::SHT3XComponent*, *sensor::TSL2561Sensor*

Public Functions

```
I2CDevice I2CDevice(I2CComponent *parent, uint8_t address)
```

```
void I2CDeviceset_address(uint8_t address)
```

Manually set the i2c address of this device.

```
void I2CDeviceset_parent(I2CComponent *parent)
```

Manually set the parent i2c bus for this device.

Protected Functions

```
bool I2CDeviceread_bytes(uint8_t register_, uint8_t *data, uint8_t len, uint32_t conversion  
= 0)
```

Read len amount of bytes from a register into data.

Optionally with a conversion time after writing the register value to the bus.

Return If the operation was successful.

Parameters

- **register_**: The register number to write to the bus before reading.
- **data**: An array to store len amount of 8-bit bytes into.
- **len**: The amount of bytes to request and write into data.
- **conversion**: The time in ms between writing the register value and reading out the value.

```
bool I2CDeviceread_bytes_16(uint8_t register_, uint16_t *data, uint8_t len, uint32_t conversion  
= 0)
```

Read len amount of 16-bit words (MSB first) from a register into data.

Return If the operation was successful.

Parameters

- **register_**: The register number to write to the bus before reading.

- **data**: An array to store len amount of 16-bit words into.
- **len**: The amount of 16-bit words to request and write into data.
- **conversion**: The time in ms between writing the register value and reading out the value.

`bool I2CDeviceread_byte(uint8_t register_, uint8_t *data, uint32_t conversion = 0)`

Read a single byte from a register into the data variable. Return true if successful.

`bool I2CDeviceread_byte_16(uint8_t register_, uint16_t *data, uint32_t conversion = 0)`

Read a single 16-bit words (MSB first) from a register into the data variable. Return true if successful.

`bool I2CDevicewrite_bytes(uint8_t register_, const uint8_t *data, uint8_t len)`

Write len amount of 8-bit bytes to the specified register.

Return If the operation was successful.

Parameters

- **register_**: The register to write the values to.
- **data**: An array from which len bytes of data will be written to the bus.
- **len**: The amount of bytes to write to the bus.

`bool I2CDevicewrite_bytes_16(uint8_t register_, const uint16_t *data, uint8_t len)`

Write len amount of 16-bit words (MSB first) to the specified register.

Return If the operation was successful.

Parameters

- **register_**: The register to write the values to.
- **data**: An array from which len 16-bit words of data will be written to the bus.
- **len**: The amount of bytes to write to the bus.

`bool I2CDevicewrite_byte(uint8_t register_, uint8_t data)`

Write a single byte of data into the specified register. Return true if successful.

`bool I2CDevicewrite_byte_16(uint8_t register_, uint16_t data)`

Write a single 16-bit word of data into the specified register. Return true if successful.

Protected Attributes

`uint8_t I2CDeviceaddress`

`I2CComponent *I2CDeviceparent_`

Automation

API Reference

Typedefs

`using NoArg = bool`

```
template <typename T>
class Condition
    Subclassed by AndCondition< T >, LambdaCondition< T >, OrCondition< T >
```

Public Functions

```
virtual bool Conditioncheck(T x) = 0
template <typename T>
class AndCondition
    Inherits from Condition< T >
```

Public Functions

```
AndConditionAndCondition(const std::vector<Condition<T> *> &conditions)
bool AndConditioncheck(T x)
```

Protected Attributes

```
std::vector<Condition<T> *> AndConditionconditions_
template <typename T>
class OrCondition
    Inherits from Condition< T >
```

Public Functions

```
OrConditionOrCondition(const std::vector<Condition<T> *> &conditions)
bool OrConditioncheck(T x)
```

Protected Attributes

```
std::vector<Condition<T> *> OrConditionconditions_
template <typename T>
class LambdaCondition
    Inherits from Condition< T >
```

Public Functions

```
LambdaConditionLambdaCondition(std::function<bool> T
    > &&f
bool LambdaConditioncheck(T x)
```

Protected Attributes

```
std::function<bool(T)> LambdaConditionf_  

class RangeCondition  

    Inherits from Condition<float>
```

Public Functions

```
RangeConditionRangeCondition()  

bool RangeConditioncheck(float x)  

void RangeConditionset_min(std::function<float> float  

    > &&min)  

void RangeConditionset_min(float min)  

void RangeConditionset_max(std::function<float> float  

    > &&max)  

void RangeConditionset_max(float max)
```

Protected Attributes

```
TemplatableValue<float, float> RangeConditionmin_ = {NAN}  

TemplatableValue<float, float> RangeConditionmax_ = {NAN}  

template <typename T>  

class Trigger
```

Public Functions

```
void Triggeradd_on_trigger_callback(std::function<void()> T  

    > &&f)  

void Triggertrigger(T x)
```

Protected Attributes

```
CallbackManager<void(T)> Triggeron_trigger_  

template <>  

template<>  

class Trigger<NoArg>  

    Subclassed by binary_sensor::ClickTrigger, binary_sensor::DoubleClickTrigger,  

    binary_sensor::PressTrigger, binary_sensor::ReleaseTrigger, StartupTrigger
```

Public Functions

```
void Triggeradd_on_trigger_callback(std::function<void> NoArg
                                     > &&f
void Triggertrigger()
```

Protected Attributes

```
CallbackManager<void(NoArg)> Triggeron_trigger_
class StartupTrigger
Inherits from Trigger<NoArg>, Component
```

Public Functions

```
void StartupTriggersetup()
Where the component's initialization should happen.
Analogous to Arduino's setup(). This method is guaranteed to only be called once. Defaults to doing noting.
float StartupTriggerget_setup_priority() const
priority of setup().
higher -> executed earlier
Defaults to 0.
```

Return The setup priority of this component

```
class ShutdownTrigger
Inherits from Trigger<const char *>
```

Public Functions

```
ShutdownTriggerShutdownTrigger()
template <typename T>
class ActionList
```

Public Functions

```
Action<T> *ActionListadd_action(Action<T> *action)
void ActionListadd_actions(const std::vector<Action<T> *> &actions)
void ActionListplay(T x)
```

Protected Attributes

```
Action<T> *ActionListactions_begin_ = {nullptr}
Action<T> *ActionListactions_end_ = {nullptr}
template <typename T>
class Action
    Subclassed by cover::CloseAction< T >, cover::OpenAction< T >, cover::StopAction< T >, DelayAction< T >, fan::ToggleAction< T >, fan::TurnOffAction< T >, fan::TurnOnAction< T >, LambdaAction< T >, light::ToggleAction< T >, light::TurnOffAction< T >, light::TurnOnAction< T >, mqtt::MQTTPublishAction< T >, switch_::ToggleAction< T >, switch_::TurnOffAction< T >, switch_::TurnOnAction< T >
```

Public Functions

```
virtual void Actionplay(T x) = 0
void Actionplay_next(T x)
```

Protected Attributes

```
friend Action::ActionList< T >
Action<T> *Actionnext_ = nullptr
template <typename T>
class DelayAction
    Inherits from Action< T >, Component
```

Public Functions

```
DelayActionDelayAction()
void DelayActionset_delay(std::function<uint32_t> T
    > &&delay)
void DelayActionset_delay(uint32_t delay)
void DelayActionplay(T x)
```

Protected Attributes

```
TemplatableValue<uint32_t, T> DelayActiondelay_ = {0}
template <typename T>
class LambdaAction
    Inherits from Action< T >
```

Public Functions

```
LambdaActionLambdaAction(std::function<void> T
    > &&f)
```

```
void LambdaActionplay(T x)
```

Protected Attributes

```
std::function<void(T)> LambdaActionf_
template <typename T>
class ActionList
```

Public Functions

```
Action<T> *ActionListadd_action(Action<T> *action)
void ActionListadd_actions(const std::vector<Action<T> *> &actions)
void ActionListplay(T x)
```

Protected Attributes

```
Action<T> *ActionListactions_begin_ = {nullptr}
Action<T> *ActionListactions_end_ = {nullptr}
template <typename T>
class Automation
```

Public Functions

```
AutomationAutomation(Trigger<T> *trigger)
Condition<T> *Automationadd_condition(Condition<T> *condition)
void Automationadd_conditions(const std::vector<Condition<T> *> &conditions)
Action<T> *Automationadd_action(Action<T> *action)
void Automationadd_actions(const std::vector<Action<T> *> &actions)
```

Protected Functions

```
void Automationprocess_trigger_(T x)
```

Protected Attributes

```
Trigger<T> *Automationtrigger_
std::vector<Condition<T> *> Automationconditions_
ActionList<T> Automationactions_
```

3.3.2 Sensor

The `sensor` namespace contains all sensors.

See `Application::register_sensor()`.

ADC Sensor

Example Usage

```
// Basic
App.make_adc_sensor("Analog Voltage", 13);
// Custom update interval 30 seconds
App.make_adc_sensor("Analog Voltage", 13, 30000);
// Custom pinMode
App.make_adc_sensor("Analog Voltage", GPIOInputPin(13, INPUT_PULLUP));
// ESP32: Attenuation
auto adc = App.make_adc_sensor("Analog Voltage", 13);
adc.adc.set_attenuation(ADC_11db);
```

See `Application::make_adc_sensor()`.

API Reference

`class sensor::ADCSensorComponent`

This class allows using the integrated Analog to Digital converts of the ESP32 and ESP8266.

Internally it uses the existing `analogRead` methods for doing this.

The ESP8266 only has one pin where this can be used: A0

The ESP32 has multiple pins that can be used with this component: GPIO32-GPIO39 Note you can't use the ADC2 here on the ESP32 because that's already used by WiFi internally. Additionally on the ESP32 you can set an using `set_attenuation`.

Inherits from `sensor::PollingSensorComponent`

Public Functions

`sensor::ADCSensorComponent::ADCSensorComponent(const std::string &name, GPIOInputPin pin, uint32_t update_interval = 15000)`

Construct the ADCSensor with the provided pin and update interval in ms.

`void sensor::ADCSensorComponent::set_pin(const GPIOInputPin &pin)`

Manually set the pin used for this ADC sensor.

`void sensor::ADCSensorComponent::set_attenuation(adc_attenuation_t attenuation)`

Set the attenuation for this pin. Only available on the ESP32.

`GPIOInputPin &sensor::ADCSensorComponent::get_pin()`

Get the pin used for this ADC sensor.

`adc_attenuation_t sensor::ADCSensorComponent::get_attenuation() const`

Get the attenuation used for this sensor.

```
void sensor::ADCSensorComponentupdate()
    Update adc values.

void sensor::ADCSensorComponentsensor()
    Setup ADC.

std::string sensor::ADCSensorComponentunit_of_measurement()
    Unit of measurement: "V".

std::string sensor::ADCSensorComponenticon()
    Icon: "mdi:flash".

int8_t sensor::ADCSensorComponentaccuracy_decimals()
    Accuracy decimals: 2.

float sensor::ADCSensorComponentget_setup_priority() const
    HARDWARE_LATE setup priority.

std::string sensor::ADCSensorComponentunique_id()
    A unique ID for this sensor, empty for no unique id.
    See unique ID requirements: https://developers.home-assistant.io/docs/en/entity\_registry\_index.html#unique-id-requirements

Return The unique id as a string.
```

Protected Attributes

```
GPIOInputPin sensor::ADCSensorComponentpin_
adc_attenuation_t sensor::ADCSensorComponentattenuation_ = {ADC_0db}
```

Dallas Component

This class allows using Dallas (DS18b20) devices with esphomelib. You first have to create a hub or bus where all sensors are connected to (`DallasComponent`).

Next, use `get_sensor_by_address()` and `get_sensor_by_index()` to get individual sensors. You can get the addresses of dallas sensors by observing the log output at startup time.

Example Usage

```
// Bus setup
auto *dallas = App.make_dallas_component(15);

// By address
App.register_sensor(dallas->get_sensor_by_address("Ambient Temperature", 0xfe0000031f1eaf29));
// By index
App.register_sensor(dallas->get_sensor_by_index("Ambient Temperature", 0));
```

See `Application::make_dallas_component()` and `Application::register_sensor()`.

API Reference

`class sensor::DallasComponent`

Hub for dealing with dallas temperature sensor.

Uses a OneWire interface.

Get the individual sensors with `get_sensor_by_address` or `get_sensor_by_index`.

Inherits from `PollingComponent`

Public Functions

`sensor::DallasComponentDallasComponent(ESPOneWire *one_wire, uint32_t update_interval)`

Construct the `DallasComponent` hub with the given OneWire instance pointer.

`DallasTemperatureSensor *sensor::DallasComponentget_sensor_by_address(const std::string &name, uint64_t address, uint8_t resolution = 12)`

Get a `DallasTemperatureSensor` by address.

Return A pointer to a `DallasTemperatureSensor`, use this to setup MQTT.

Parameters

- `address`: 64-bit unsigned address for this sensor. Check debug logs for getting this.
- `resolution`: The resolution for this sensor, 8-12.

`DallasTemperatureSensor *sensor::DallasComponentget_sensor_by_index(const std::string &name, uint8_t index, uint8_t resolution = 12)`

Get a `DallasTemperatureSensor` by index.

Note: It is recommended to use sensors by address to avoid mixing up sensor values if one sensor can't be found (and therefore receives an incorrect index).

Return A pointer to a `DallasTemperatureSensor`, use this to setup MQTT.

Parameters

- `index`: The index of this sensor, starts with 0.
- `resolution`: The resolution for this sensor, 8-12.

`void sensor::DallasComponentset_one_wire(ESPOneWire *one_wire)`

Get the `ESPOneWire` instance used for this hub.

Manually set the `ESPOneWire` instance used for this hub.

`void sensor::DallasComponentsetup()`

Set up individual sensors and update intervals.

`float sensor::DallasComponentget_setup_priority() const`

HARDWARE_LATE setup priority.

`void sensor::DallasComponentupdate()`

```
ESPOneWire *sensor::DallasComponentget_one_wire() const
```

Protected Attributes

```
ESPOneWire *sensor::DallasComponentone_wire_
std::vector<DallasTemperatureSensor *> sensor::DallasComponents_
class sensor::DallasTemperatureSensor
Internal class that helps us create multiple sensors for one Dallas hub.
Inherits from sensor::EmptyPollingParentSensor< 1, ICON_EMPTY, UNIT_C, DallasComponent >
```

Public Functions

```
sensor::DallasTemperatureSensorDallasTemperatureSensor(const std::string &name, uint64_t
address, uint8_t resolution, Dallas-
Component *parent)
```

Construct the temperature sensor with the given address.

Parameters

- **address:** 64-bit unsigned address of the temperature sensor. Can be 0 to indicate using index.
- **resolution:** Resolution used for this sensor. Usually 8-12.
- **update_interval:** The interval in ms the sensor should be checked.

```
uint8_t *sensor::DallasTemperatureSensorget_address8()
```

Helper to get a pointer to the address as uint8_t.

```
const std::string &sensor::DallasTemperatureSensorget_address_name()
```

Helper to create (and cache) the name for this sensor. For example “0xfe0000031fleaf29”.

```
uint64_t sensor::DallasTemperatureSensorget_address() const
```

Get the 64-bit unsigned address for this sensor.

```
void sensor::DallasTemperatureSensorset_address(uint64_t address)
```

Set the 64-bit unsigned address for this sensor.

```
uint8_t sensor::DallasTemperatureSensorget_index() const
```

Get the index of this sensor. (0 if using address.)

```
void sensor::DallasTemperatureSensorset_index(uint8_t index)
```

Set the index of this sensor. If using index, address will be set after setup.

```
uint8_t sensor::DallasTemperatureSensorget_resolution() const
```

Get the set resolution for this sensor.

```
void sensor::DallasTemperatureSensorset_resolution(uint8_t resolution)
```

Set the resolution for this sensor.

```
uint16_t sensor::DallasTemperatureSensormillis_to_wait_for_conversion_() const
```

Get the number of milliseconds we have to wait for the conversion phase.

```
void sensor::DallasTemperatureSensorsetup_sensor_()
```

```
bool sensor::DallasTemperatureSensorread_scratch_pad_()
bool sensor::DallasTemperatureSensorcheck_scratch_pad_()
float sensor::DallasTemperatureSensorget_temp_c()
std::string sensor::DallasTemperatureSensorunique_id()
A unique ID for this sensor, empty for no unique id.
See unique ID requirements: https://developers.home-assistant.io/docs/en/entity\_registry\_index.html#unique-id-requirements
```

Return The unique id as a string.

Protected Attributes

```
uint64_t sensor::DallasTemperatureSensoraddress_
uint8_t sensor::DallasTemperatureSensorindex_
uint8_t sensor::DallasTemperatureSensorresolution_
std::string sensor::DallasTemperatureSensoraddress_name_
uint8_t sensor::DallasTemperatureSensorscratch_pad_[9] = {0,}
```

DHT Temperature/Humidity Sensor

Example Usage

```
// Basic
App.make_dht_sensor("Outside Temperature", "Outside Humidity", 12);
```

See *Application::make_dht_sensor()*.

API Reference

```
class sensor::DHTComponent
Component for reading temperature/humidity measurements from DHT11/DHT22 sensors.
Inherits from PollingComponent
```

Public Functions

```
sensor::DHTComponentDHTComponent(const std::string &temperature_name, const std::string
&humidity_name, GPIOPin *pin, uint32_t update_interval = 15000)
Construct a DHTComponent.
```

Parameters

- **pin:** The pin which DHT sensor is connected to.
- **update_interval:** The interval in ms the sensor should be checked.

```
void sensor::DHTComponentset_dht_model(DHTModel model)
Manually select the DHT model.
```

Valid values are:

- DHT_MODEL_AUTO_DETECT (default)
- DHT_MODEL_DHT11
- DHT_MODEL_DHT22
- DHT_MODEL_AM2302
- DHT_MODEL_RHT03

Parameters

- `model`: The DHT model.

```
DHTTemperatureSensor *sensor::DHTComponentget_temperature_sensor() const
```

```
DHTHumiditySensor *sensor::DHTComponentget_humidity_sensor() const
```

```
void sensor::DHTComponentsetup()
```

Set up the pins and check connection.

```
void sensor::DHTComponentupdate()
```

Update sensor values and push them to the frontend.

```
float sensor::DHTComponentget_setup_priority() const
```

HARDWARE_LATE setup priority.

Protected Functions

```
uint8_t sensor::DHTComponentread_sensor_(float *temperature, float *humidity)
```

```
uint8_t sensor::DHTComponentread_sensor_safe_(float *temperature, float *humidity)
```

Protected Attributes

```
GPIOPin *sensor::DHTComponentpin_
```

```
DHTModel sensor::DHTComponentmodel_ = {DHT_MODEL_AUTO_DETECT}
```

```
DHTTemperatureSensor *sensor::DHTComponenttemperature_sensor_
```

```
DHTHumiditySensor *sensor::DHTComponenthumidity_sensor_
```

DHT12 Temperature/Humidity Sensor

Example Usage

```
// Basic
App.make_dht12_sensor("Outside Temperature", "Outside Humidity");
```

See `Application::make_dht12_sensor()`.

API Reference

```
using sensor::DHT12TemperatureSensor = typedef EmptyPollingParentSensor<1, ICON_EMPTY, UNIT_C>
using sensor::DHT12HumiditySensor = typedef EmptyPollingParentSensor<0, ICON_WATER_PERCENT, UNIT_PERCENT>
class sensor::DHT12Component
    Inherits from PollingComponent, I2CDevice
```

Public Functions

```
sensor::DHT12Component(DHT12Component *parent, const std::string &temperature_name, const std::string &humidity_name, uint32_t update_interval = 15000)
```

void sensor::*DHT12Component***setup()**

Where the component's initialization should happen.

Analogous to Arduino's *setup()*. This method is guaranteed to only be called once. Defaults to doing noting.

```
float sensor::DHT12Componentget_setup_priority() const
    priority of setup().
```

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void sensor::DHT12Componentupdate()
```

```
DHT12TemperatureSensor *sensor::DHT12Componentget_temperature_sensor() const
```

```
DHT12HumiditySensor *sensor::DHT12Componentget_humidity_sensor() const
```

Protected Functions

```
bool sensor::DHT12Componentread_data_(uint8_t *data)
```

Protected Attributes

```
DHT12TemperatureSensor *sensor::DHT12Componenttemperature_sensor_
```

```
DHT12HumiditySensor *sensor::DHT12Componenthumidity_sensor_
```

ESP32 Pulse Counter Sensor

This component allows you to count pulses on a PIN using the internal *pulse* counter peripheral of the ESP32. By default, the values by this sensor are reported in "pulses/min". You can convert that to your own unit like in below example.

Example Usage

```
// Basic
App.make_pulse_counter_sensor("Stromverbrauch Wärmepumpe", 13);
// Unit conversion
auto strom_warme = App.make_pulse_counter_sensor("Stromverbrauch Wärmepumpe", 13);
strom_warme.mqtt->set_unit_of_measurement("kW");
strom_warme.mqtt->clear_filters();
strom_warme.mqtt->add_multiply_filter(0.06f); // convert from Wh pulse to kW
```

See [Application::make_pulse_counter_sensor\(\)](#).

API Reference

class sensor::PulseCounterSensorComponent

Pulse Counter - This is the sensor component for the ESP32 integrated pulse counter peripheral.

It offers 8 pulse counter units that can be setup in several ways to count pulses on a pin. Also allows for some simple filtering of short pulses using [set_filter\(\)](#), any pulse shorter than the value provided to that function will be discarded. The time is given in APB clock cycles, which usually amount to 12.5 ns per clock. Defaults to the max possible (about 13 ms). See <http://esp-idf.readthedocs.io/en/latest/api-reference/peripherals/pcnt.html> for more information.

The pulse counter defaults to reporting a value of the measurement unit “pulses/min”. To modify this behavior, use filters in MQTTSensor.

Inherits from [sensor::PollingSensorComponent](#)

Public Functions

```
sensor::PulseCounterSensorComponent::PulseCounterSensorComponent(const std::string
&name, uint8_t
pin, uint32_t update_interval = 30000)
```

Construct the Pulse Counter instance with the provided pin and update interval.

The pulse counter unit will automatically be set and the pulse counter is set up to increment the counter on rising edges by default.

Parameters

- **pin:** The pin.
- **update_interval:** The update interval in ms.

```
void sensor::PulseCounterSensorComponent::set_pin(uint8_t pin)
```

Manually set the pin for the pulse counter unit.

```
void sensor::PulseCounterSensorComponent::set_pull_mode(gpio_pull_mode_t pull_mode)
```

Manually set the pull mode of this pin, default to floating.

```
void sensor::PulseCounterSensorComponent::set_edge_mode(pcnt_count_mode_t
rising_edge_mode,
pcnt_count_mode_t
falling_edge_mode)
```

Set the pcnt_count_mode_t for the rising and falling edges. can be disable, increment and decrement.

```
void sensor::PulseCounterSensorComponentset_filter(uint16_t filter)
    Set a filter for this Pulse Counter unit.
```

See <http://esp-idf.readthedocs.io/en/latest/api-reference/peripherals/pcnt.html#filtering-pulses>

Filter is given in APB clock cycles, so a value of one would filter out any pulses shorter than 12.5 ns. This value can have 10-bit at maximum, so the maximum possible value is 1023, or about 12ms.

Parameters

- **filter:** The filter length in APB clock cycles.

```
void sensor::PulseCounterSensorComponentset_pcnt_unit(pcnt_unit_t pcnt_unit)
```

Manually set the pulse counter unit to be used. This is automatically set by the constructor.

```
uint16_t sensor::PulseCounterSensorComponentget_filter() const
```

Return the value from *set_filter()*.

```
gpio_pull_mode_t sensor::PulseCounterSensorComponentget_pull_mode() const
```

Return the pull mode from *set_pull_mode()*:

```
pcnt_unit_t sensor::PulseCounterSensorComponentget_pcnt_unit() const
```

Get the pulse counter unit of this component. Automatically set by constructor.

```
pcnt_count_mode_t sensor::PulseCounterSensorComponentget_rising_edge_mode() const
```

```
pcnt_count_mode_t sensor::PulseCounterSensorComponentget_falling_edge_mode() const
```

```
std::string sensor::PulseCounterSensorComponentunit_of_measurement()
```

Unit of measurement is “pulses/min”.

```
std::string sensor::PulseCounterSensorComponenticon()
```

Override this to set the Home Assistant icon for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “mdi:battery”.

```
int8_t sensor::PulseCounterSensorComponentaccuracy_decimals()
```

Return the accuracy in decimals for this sensor.

```
void sensor::PulseCounterSensorComponentsetup()
```

Where the component’s initialization should happen.

Analogous to Arduino’s *setup()*. This method is guaranteed to only be called once. Defaults to doing noting.

```
void sensor::PulseCounterSensorComponentupdate()
```

```
float sensor::PulseCounterSensorComponentget_setup_priority() const
```

priority of *setup()*.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
uint8_t sensor::PulseCounterSensorComponentget_pin() const
```

Protected Attributes

```
uint8_t sensor::PulseCounterSensorComponentpin_
gpio_pull_mode_t sensor::PulseCounterSensorComponentpull_mode_ = {GPIO_FLOATING}
pcnt_unit_t sensor::PulseCounterSensorComponentpcnt_unit_
pcnt_count_mode_t sensor::PulseCounterSensorComponentrising_edge_mode_ = {PCNT_COUNT_INC}
pcnt_count_mode_t sensor::PulseCounterSensorComponentfalling_edge_mode_ = {PCNT_COUNT_DIS}
uint16_t sensor::PulseCounterSensorComponentfilter_ = {1023}
int16_t sensor::PulseCounterSensorComponentlast_value_ = {0}
```

Ultrasonic Sensor

Example Usage

```
// Basic
App.make_ultrasonic("Ultrasonic", 12, 13); // trigger pin: 12, echo pin: 13
// Filter out timeouts
auto ultrasonic = App.make_ultrasonic("Ultrasonic", 12, 13);
ultrasonic.mqtt->set_filters(
    new sensor::FilterOutNANFilter(), // filter out timeouts
);
// Set timeout, 4 meters
ultrasonic.sensor->set_timeout_m(4.0f);
```

See *Application::make_ultrasonic_sensor()*.

API Reference

```
class sensor::UltrasonicSensorComponent
```

This sensor component allows you to use your ultrasonic distance sensors.

Instances of this component periodically send out a short pulse to an ultrasonic sensor, thus creating an ultrasonic sound, and check how long it takes until an “echo” comes back. With this we can measure the distance to an object.

Sometimes it can happen that we send a short ultrasonic pulse, but never hear the echo. In order to not wait for that signal indefinitely, this component has an additional parameter that allows setting a timeout in microseconds (or meters). If this timeout is reached, the sensor reports a “nan” (not a number) float value. The timeout defaults to 11662µs or 2m.

Usually these would be like HC-SR04 ultrasonic sensors: one trigger pin for sending the signal and another one echo pin for receiving the signal. Be very careful with that sensor though: it’s made for 5V VCC and doesn’t work very well with the ESP’s 3.3V, so you need to create a voltage divider in order to not damage your ESP.

Note: The MQTTSenorComponent will create a moving average over these values by default, to disable this behavior, call ultrasonic->mqtt->`clear_filters()` or similar like above.

Inherits from `sensor::PollingSensorComponent`

Public Functions

```
sensor::UltrasonicSensorComponent::UltrasonicSensorComponent(const std::string &name,
                                                               GPIOPin *trigger_pin, GPIOPin *echo_pin, uint32_t
                                                               update_interval = 5000)
```

Construct the ultrasonic sensor with the specified trigger pin and echo pin.

Parameters

- `trigger_pin`: The trigger pin where pulses are sent to.
- `echo_pin`: The echo pin where the echo is listened for.
- `update_interval`: The interval in ms the sensor should check for new values.

```
void sensor::UltrasonicSensorComponent::set_timeout_us(uint32_t timeout_us)
```

Set the timeout for waiting for the echo in μs .

```
void sensor::UltrasonicSensorComponent::set_timeout_m(float timeout_m)
```

Set the timeout for waiting for the echo in meter.

```
float sensor::UltrasonicSensorComponent::get_timeout_m() const
```

Get the timeout in meters for waiting for the echo.

```
uint32_t sensor::UltrasonicSensorComponent::get_timeout_us() const
```

Get the timeout in μs for waiting for the echo.

```
void sensor::UltrasonicSensorComponent::setup()
```

Set up pins and register interval.

```
void sensor::UltrasonicSensorComponent::update()
```

```
std::string sensor::UltrasonicSensorComponent::unit_of_measurement()
```

Override this to set the Home Assistant unit of measurement for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “°C”.

```
std::string sensor::UltrasonicSensorComponent::icon()
```

Override this to set the Home Assistant icon for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “mdi:battery”.

```
int8_t sensor::UltrasonicSensorComponent::accuracy_decimals()
```

Return the accuracy in decimals for this sensor.

```
float sensor::UltrasonicSensorComponent::get_setup_priority() const
```

Hardware setup priority, before MQTT and WiFi.

```
void sensor::UltrasonicSensorComponentset_pulse_time_us(uint32_t pulse_time_us)
    Set the time in µs the trigger pin should be enabled for in µs, defaults to 10µs (for HC-SR04)

uint32_t sensor::UltrasonicSensorComponentget_pulse_time_us() const
    Return the time in µs the trigger pin should be enabled for.
```

Protected Attributes

```
GPIOPin *sensor::UltrasonicSensorComponenttrigger_pin_
GPIOPin *sensor::UltrasonicSensorComponentecho_pin_
uint32_t sensor::UltrasonicSensorComponenttimeout_us_ = {11662}
uint32_t sensor::UltrasonicSensorComponentpulse_time_us_ = {10}
    2 meters.
```

Protected Static Functions

```
float sensor::UltrasonicSensorComponentus_to_m(uint32_t us)
    Helper function to convert the specified echo duration in µs to meters.

uint32_t sensor::UltrasonicSensorComponentm_to_us(float m)
    Helper function to convert the specified distance in meters to the echo duration in µs.
```

ADS1115 ADC Component

Warning: This sensor is experimental has not been tested yet. If you can verify it works, notify me on discord.

This class allows using ADS1115 Analog to Digital converters ([datasheet](#), [adafruit](#)) devices with esphomelib. Doing so requires some steps:

1. Initialize the i2c bus with the pins you have SDA and SCL connected to:

```
// inside setup()
App.init_i2c(SDA_PIN, SCL_PIN);
```

2. Create the “hub” or the ADS1115 device itself. The parameter you pass in here is the i2c address of the ADS1115. See `set_address()` for possible other addresses.

```
// after init_i2c
auto *ads1115 = App.make_ads1115_component(0x48);
```

This will create an ADS1115 component which you can now use to create individual sensors.

3. Create the sensors, you can have multiple of these. Do so by calling `get_sensor()` with the multiplexer channel you want (essentially between which pins you want to measure voltage) and the gain for that sensor and register that sensor.

```
auto *sensor = ads1115->get_sensor("ADS1115 Voltage #1", ADS1115_MUX_P0_N1, ADS1115_PGA_
    ↵6P144);
App.register_sensor(sensor);
```

See `Application::make_ads1115_component()`.

API Reference

`class sensor::ADS1115Component`

This class allows you to use your ADS1115 devices with esphomelib through i2c.

It is built of like the `DallasComponent`: A central hub (can be multiple ones) and multiple `Sensor` instances that all access this central hub.

Note that for this component to work correctly you need to have i2c setup. Do so with

```
App.init_i2c(SDA_PIN, SCL_PIN);
```

before the call to `App.setup()`.

Inherits from `Component`, `I2CDevice`

Public Functions

`sensor::ADS1115Component::ADS1115Component(I2CComponent *parent, uint8_t address)`

Construct the component hub for this ADS1115.

Parameters

- `address`: The i2c address for this sensor.

```
ADS1115Sensor *sensor::ADS1115Component::get_sensor(const std::string &name,
                                                    ADS1115Multiplexer multiplexer,
                                                    ADS1115Gain gain, uint32_t update_interval = 15000)
```

Get a sensor from this ADS1115 from the specified multiplexer and gain.

You can have one ADS1115 create multiple sensors with different multiplexers and/or gains.

Return An `ADS1115Sensor`, use this for advanced options.

Parameters

- `multiplexer`: The multiplexer, one of `ADS1115_MULTIPLEXER_` then `P0_N1`, `P0_N3`, `P1_N3`, `P2_N3`, `P0_NG`, `P1_NG`, `P2_NG`, `P3_NG`.
- `gain`: The gain, one of `ADS1115_GAIN_` then `6P144`, `4P096`, `2P048`, `1P024`, `0P512`, `0P256`` (B/C).
- `update_interval`: The interval in milliseconds the value for this sensor should be checked.

`void sensor::ADS1115Component::setup()`

Set up the internal sensor array.

`float sensor::ADS1115Component::get_setup_priority() const`

HARDWARE_LATE setup priority.

Protected Functions

```
void sensor::ADS1115Componentrequest_measurement_(ADS1115Sensor *sensor)  
    Helper method to request a measurement from a sensor.
```

Protected Attributes

```
std::vector<ADS1115Sensor *> sensor::ADS1115Componentsensors_  
  
class sensor::ADS1115Sensor  
    Internal holder class that is in instance of Sensor so that the hub can create individual sensors.  
    Inherits from sensor::EmptySensor< 3, ICON_FLASH, UNIT_V >
```

Public Functions

```
sensor::ADS1115SensorADS1115Sensor(const std::string &name, ADS1115Multiplexer multiplexer, ADS1115Gain gain, uint32_t update_interval)  
  
void sensor::ADS1115Sensorset_multiplexer(ADS1115Multiplexer multiplexer)  
  
void sensor::ADS1115Sensorset_gain(ADS1115Gain gain)  
  
uint32_t sensor::ADS1115Sensorupdate_interval()  
    Return with which interval the sensor is polled. Return 0 for non-polling mode.  
  
uint8_t sensor::ADS1115Sensorget_multiplexer() const  
  
uint8_t sensor::ADS1115Sensorget_gain() const
```

Protected Attributes

```
ADS1115Multiplexer sensor::ADS1115Sensormultiplexer_  
ADS1115Gain sensor::ADS1115Sensorgain_  
uint32_t sensor::ADS1115Sensorupdate_interval_
```

BMP085 Pressure/Temperature Sensor

The BMP085 component allows you get the temperature and pressure from your BMP085 ([datasheet](#), [adafruit](#)), BMP180 ([datasheet](#), [adafruit](#)) and BMP280 ([datasheet](#), [adafruit](#)) sensors with esphomelib. To use these i2c-based devices, first initialize the i2c bus using the pins you have for SDA and SCL and then create the sensors themselves as in below example.

Example Usage

```
// inside setup()  
App.init_i2c(SDA_PIN, SCL_PIN); // change these values for your pins.  
// create sensors  
App.make_bmp085_sensor("Outside Temperature", "Outside Pressure");
```

See *Application::make_bmp085_sensor()*.

API Reference

`class sensor::BMP085Component`

This *Component* represents a BMP085/BMP180/BMP280 Pressure+Temperature i2c sensor.

It's built up similar to the DHT component: a central hub that has two sensors.

Inherits from *PollingComponent*, *I2CDevice*

Public Functions

`sensor::BMP085Component::BMP085Component(I2CComponent *parent, const std::string &temperature_name, const std::string &pressure_name, uint32_t update_interval = 30000)`

Construct the *BMP085Component* using the provided address and update interval.

`BMP085TemperatureSensor *sensor::BMP085Component::get_temperature_sensor() const`
Get the internal temperature sensor used to expose the temperature as a sensor object.

`BMP085PressureSensor *sensor::BMP085Component::get_pressure_sensor() const`
Get the internal pressure sensor used to expose the pressure as a sensor object.

`void sensor::BMP085Component::update()`
Schedule temperature+pressure readings.

`void sensor::BMP085Component::setup()`
Setup the sensor and test for a connection.

Protected Functions

`void sensor::BMP085Component::read_temperature_()`

Internal method to read the temperature from the component after it has been scheduled.

`void sensor::BMP085Component::read_pressure_()`
Internal method to read the pressure from the component after it has been scheduled.

`bool sensor::BMP085Component::set_mode_(uint8_t mode)`

Protected Attributes

`BMP085TemperatureSensor *sensor::BMP085Component::temperature_ = {nullptr}`

`BMP085PressureSensor *sensor::BMP085Component::pressure_ = {nullptr}`

`CalibrationData sensor::BMP085Component::calibration_`

`struct sensor::BMP085Component::CalibrationData`

Public Members

`int16_t sensor::BMP085Component::CalibrationData::ac1`

`int16_t sensor::BMP085Component::CalibrationData::ac2`

```
int16_t sensor::BMP085Component::CalibrationDataac3
int16_t sensor::BMP085Component::CalibrationDataac4
int16_t sensor::BMP085Component::CalibrationDataac5
int16_t sensor::BMP085Component::CalibrationDataac6
int16_t sensor::BMP085Component::CalibrationDatab1
int16_t sensor::BMP085Component::CalibrationDatab2
int16_t sensor::BMP085Component::CalibrationDatamb
int16_t sensor::BMP085Component::CalibrationDatamc
int16_t sensor::BMP085Component::CalibrationDatamd
int32_t sensor::BMP085Component::CalibrationDatab5
```

HTU21D Temperature/Humidity Sensor

The HTU21D component allows you get accurate temperature and humidity readings from your HTU21D ([datasheet](#), [adafruit](#)) sensors with esphomelib. To use these i2c-based devices, first initialize the i2c bus using the pins you have for SDA and SCL and then create the sensors themselves as in below example.

Example Usage

```
// inside setup()
App.init_i2c(SDA_PIN, SCL_PIN); // change these values for your pins.
// create sensors
App.make_htu21d_sensor("Outside Temperature", "Outside Humidity");
```

See [Application::make_htu21d_sensor\(\)](#).

API Reference

class sensor::HTU21DComponent

This component represents the HTU21D i2c temperature+humidity sensor in esphomelib.

It's basically an i2c-based accurate temperature and humidity sensor. See <https://www.adafruit.com/product/1899> for more information.

Inherits from [PollingComponent](#), [I2CDevice](#)

Public Functions

```
sensor::HTU21DComponent::HTU21DComponent(I2CComponent *parent, const std::string &temperature_name, const std::string &humidity_name, uint32_t update_interval = 15000)
```

Construct the HTU21D with the given update interval.

```
HTU21DTemperatureSensor *sensor::HTU21DComponent::get_temperature_sensor() const
Get a pointer to the temperature sensor object used to expose temperatures as a sensor.
```

```
HTU21DHumiditySensor *sensor::HTU21DComponentget_humidity_sensor() const
    Get a pointer to the humidity sensor object used to expose humidities as a sensor.

void sensor::HTU21DComponentsetup()
    Setup (reset) the sensor and check connection.

void sensor::HTU21DComponentupdate()
    Update the sensor values (temperature+humidity).
```

Protected Attributes

```
HTU21DTemperatureSensor *sensor::HTU21DComponenttemperature_ = {nullptr}
HTU21DHumiditySensor *sensor::HTU21DComponenthumidity_ = {nullptr}
```

HDC1080 Temperature/Humidity Sensor

The HDC1080 component allows you get accurate temperature and humidity readings from your HDC1080 ([datasheet](#), [adafruit](#)) sensors with esphomelib. To use these i2c-based devices, first initialize the i2c bus using the pins you have for SDA and SCL and then create the sensors themselves as in below example.

Example Usage

```
// inside setup()
App.init_i2c(SDA_PIN, SCL_PIN); // change these values for your pins.
// create sensors
App.make_hdc1080_sensor("Outside Temperature", "Outside Humidity");
```

See [Application::make_hdc1080_sensor\(\)](#).

API Reference

```
class sensor::HDC1080Component
HDC1080 temperature+humidity i2c sensor integration.
```

Based off of implementation by ClosedCube: https://github.com/closedcube/ClosedCube_HDC1080_Arduino

Inherits from [PollingComponent](#), [I2CDevice](#)

Public Functions

```
sensor::HDC1080ComponentHDC1080Component(I2CComponent *parent, const std::string &temperature_name, const std::string &humidity_name, uint32_t update_interval)
Initialize the component with the provided update interval.
```

```
void sensor::HDC1080Componentsetup()
Setup the sensor and check for connection.
```

```
void sensor::HDC1080Componentupdate()
Retrieve the latest sensor values. This operation takes approximately 16ms.
```

```
HDC1080TemperatureSensor *sensor::HDC1080Componentget_temperature_sensor() const
    Get the internal temperature sensor.
```

```
HDC1080HumiditySensor *sensor::HDC1080Componentget_humidity_sensor() const
    Get the internal humidity sensor.
```

Protected Attributes

```
HDC1080TemperatureSensor *sensor::HDC1080Componenttemperature_
HDC1080HumiditySensor *sensor::HDC1080Componenthumidity_
```

MPU6050 Accelerometer/Gyroscope Component

The MPU6050 allows you to use your MPU6050 i2c-enabled accelerometers/gyroscopes with esphomelib (datasheet, Sparkfun). It requires i2c to be setup to work.

This component only supports reading in the measurements directly from the registers of the chip at the moment. If you do need some more complicated signal processing and/or configuration options, I would recommend copying the code over and creating your own custom component, as supporting every single possible configuration is not esphomelib's ultimate goal with a chip that supports such a variety of options.

Example Usage

```
// inside setup()
App.init_i2c(SDA_PIN, SCL_PIN); // change these values for your pins.
// create sensors
auto *mpu6050 = App.make_mpu6050_sensor();
App.register_sensor(mpu6050->make_accel_x_sensor("MPU6050 Accel X"));
App.register_sensor(mpu6050->make_accel_y_sensor("MPU6050 Accel Y"));
App.register_sensor(mpu6050->make_accel_z_sensor("MPU6050 Accel Z"));
App.register_sensor(mpu6050->make_gyro_x_sensor("MPU6050 Gyro X"));
App.register_sensor(mpu6050->make_gyro_y_sensor("MPU6050 Gyro Y"));
App.register_sensor(mpu6050->make_gyro_z_sensor("MPU6050 Gyro Z"));
App.register_sensor(mpu6050->make_temperature_sensor("MPU6050 Temperature"));
```

See `Application::make_mpu6050_sensor()`.

API Reference

```
class sensor::MPU6050Component
Inherits from PollingComponent, I2CDevice
```

Public Functions

```
sensor::MPU6050Component::MPU6050Component(I2CComponent * parent, uint8_t address = 0x68, uint32_t
```

```
void sensor::MPU6050Componentsetup()
Where the component's initialization should happen.
```

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing nothing.

```
void sensor::MPU6050Componentupdate()
float sensor::MPU6050Componentget_setup_priority() const
    priority of setup().
higher -> executed earlier
Defaults to 0.
```

Return The setup priority of this component

```
MPU6050AccelSensor *sensor::MPU6050Componentmake_accel_x_sensor(const std::string
&name)
MPU6050AccelSensor *sensor::MPU6050Componentmake_accel_y_sensor(const std::string
&name)
MPU6050AccelSensor *sensor::MPU6050Componentmake_accel_z_sensor(const std::string
&name)
MPU6050GyroSensor *sensor::MPU6050Componentmake_gyro_x_sensor(const std::string
&name)
MPU6050GyroSensor *sensor::MPU6050Componentmake_gyro_y_sensor(const std::string
&name)
MPU6050GyroSensor *sensor::MPU6050Componentmake_gyro_z_sensor(const std::string
&name)
MPU6050TemperatureSensor *sensor::MPU6050Componentmake_temperature_sensor(const std::string
&name)
```

Protected Attributes

```
MPU6050AccelSensor *sensor::MPU6050Componentaccel_x_sensor_ = {nullptr}
MPU6050AccelSensor *sensor::MPU6050Componentaccel_y_sensor_ = {nullptr}
MPU6050AccelSensor *sensor::MPU6050Componentaccel_z_sensor_ = {nullptr}
MPU6050TemperatureSensor *sensor::MPU6050Componenttemperature_sensor_ = {nullptr}
MPU6050GyroSensor *sensor::MPU6050Componentgyro_x_sensor_ = {nullptr}
MPU6050GyroSensor *sensor::MPU6050Componentgyro_y_sensor_ = {nullptr}
MPU6050GyroSensor *sensor::MPU6050Componentgyro_z_sensor_ = {nullptr}
```

TSL2561 Ambient Light Sensor

Warning: This sensor is experimental has not been tested yet. If you can verify it works (or if it doesn't), notify me on [discord](#).

The TSL2561 sensor allows you to use your TSL2561 i2c-enabled ambient light sensor with esphomelib ([datasheet](#), [Adafruit](#)). It requires i2c to be setup to work.

Example Usage

```
// Basic
auto tsl2561 = App.make_tsl2561_sensor("TSL2561 Illuminance Sensor");

// Advanced settings
// set the time the sensor will take for value accumulation, default: 402 ms
tsl2561.tsl2561->set_integration_time(sensor::TSL2561_INTEGRATION_14MS);
// set a higher gain for low light conditions, default: 1x
tsl2561.tsl2561->set_gain(sensor::TSL2561_GAIN_16X);
```

See [Application::make_tsl2561_sensor\(\)](#).

API Reference

class sensor::TSL2561Sensor

This class includes support for the TSL2561 i2c ambient light sensor.

Inherits from [sensor::PollingSensorComponent](#), [I2CDevice](#)

Public Functions

`sensor::TSL2561Sensor::TSL2561Sensor(I2CComponent * parent, const std::string & name, uint8_t addr)`

`void sensor::TSL2561Sensor::set_integration_time(TSL2561IntegrationTime integration_time)`

Set the time that sensor values should be accumulated for.

Longer means more accurate, but also mean more power consumption.

Possible values are:

- `sensor::TSL2561_INTEGRATION_14MS`
- `sensor::TSL2561_INTEGRATION_101MS`
- `sensor::TSL2561_INTEGRATION_402MS` (default)

Parameters

- `integration_time`: The new integration time.

`void sensor::TSL2561Sensor::set_gain(TSL2561Gain gain)`

Set the internal gain of the sensor.

Can be useful for low-light conditions

Possible values are:

- `sensor::TSL2561_GAIN_1X` (default)
- `sensor::TSL2561_GAIN_16X`

Parameters

- `gain`: The new gain.

```
void sensor::TSL2561Sensorset_is_cs_package(bool package_cs)
```

The “CS” package of this sensor has a slightly different formula for converting the raw values.

Use this setting to indicate that this is a CS package. Defaults to false (not a CS package)

Parameters

- `package_cs`: Is this a CS package.

```
void sensor::TSL2561Sensorsetup()
```

Where the component’s initialization should happen.

Analogous to Arduino’s `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

```
void sensor::TSL2561Sensorupdate()
```

```
std::string sensor::TSL2561Sensorunit_of_measurement()
```

Override this to set the Home Assistant unit of measurement for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “°C”.

```
std::string sensor::TSL2561Sensoricon()
```

Override this to set the Home Assistant icon for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “mdi:battery”.

```
int8_t sensor::TSL2561Sensoraccuracy_decimals()
```

Return the accuracy in decimals for this sensor.

```
float sensor::TSL2561Sensorget_setup_priority() const
```

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
bool sensor::TSL2561Sensortsl2561_read_byte(uint8_t register_, uint8_t *value)
```

```
bool sensor::TSL2561Sensortsl2561_read_uint(uint8_t register_, uint16_t *value)
```

```
bool sensor::TSL2561Sensortsl2561_write_byte(uint8_t register_, uint8_t value)
```

Protected Functions

```
float sensor::TSL2561Sensorget_integration_time_ms_()
```

```
void sensor::TSL2561Sensorread_data_()
```

```
float sensor::TSL2561Sensorcalculate_lx_(uint16_t ch0, uint16_t ch1)
```

Protected Attributes

```
TSL2561IntegrationTime sensor::TSL2561Sensorintroduction_time_ = {TSL2561_INTEGRATION_402MS}
TSL2561Gain sensor::TSL2561Sensorgain_ = {TSL2561_GAIN_1X}
bool sensor::TSL2561Sensorpackage_cs_ = {false}

enum sensor::TSL2561IntegrationTime
    Enum listing all conversion/integration time settings for the TSL2561.
    Higher values mean more accurate results, but will take more energy/more time.

    Values:
        sensorTSL2561_INTEGRATION_14MS = 0b00
        sensorTSL2561_INTEGRATION_101MS = 0b01
        sensorTSL2561_INTEGRATION_402MS = 0b10

enum sensor::TSL2561Gain
    Enum listing all gain settings for the TSL2561.
    Higher values are better for low light situations, but can increase noise.

    Values:
        sensorTSL2561_GAIN_1X = 0
        sensorTSL2561_GAIN_16X = 1
```

BME280 Temperature/Pressure/Humidity Sensor

The BME280 sensor allows you to use your BME280 i2c-enabled temperature+pressure+humidity sensor with esphomelib ([datasheet](#), [adafruit](#)). It requires i2c to be setup to work.

Example Usage

```
// Basic
auto bme280 = App.make_bme280_sensor("BME280 Temperature", "BME280 Pressure", "BME280 Humidity");

// Advanced settings
// set infinite impulse response filter, default is OFF.
bme280.bme280->set_iir_filter(sensor::BME280_IIR_FILTER_4X);
// set over value sampling, default is 16x
bme280.bme280->set_temperature_oversampling(sensor::BME280_OVERSAMPLING_16X);
bme280.bme280->set_humidity_oversampling(sensor::BME280_OVERSAMPLING_4X);
bme280.bme280->set_pressure_oversampling(sensor::BME280_OVERSAMPLING_16X);
```

See [Application::make_bme280_sensor\(\)](#).

API Reference

```
class sensor::BME280Component
    This class implements support for the BME280 Temperature+Pressure+Humidity i2c sensor.

    Inherits from PollingComponent, I2CDevice
```

Public Functions

```
sensor::BME280Component BME280Component(I2CComponent *parent, const std::string
                                         &temperature_name, const std::string &pressure_name, const std::string &humidity_name,
                                         uint8_t address = 0x77, uint32_t update_interval
                                         = 15000)
```

void sensor::BME280Component::set_temperature_oversampling(BME280Oversampling temperature_over_sampling)

Set the oversampling value for the temperature sensor. Default is 16x.

void sensor::BME280Component::set_pressure_oversampling(BME280Oversampling pressure_over_sampling)

Set the oversampling value for the pressure sensor. Default is 16x.

void sensor::BME280Component::set_humidity_oversampling(BME280Oversampling humidity_over_sampling)

Set the oversampling value for the humidity sensor. Default is 16x.

void sensor::BME280Component::set_iir_filter(BME280IIRFilter iir_filter)

Set the IIR Filter used to increase accuracy, defaults to no IIR Filter.

BME280TemperatureSensor *sensor::BME280Component::get_temperature_sensor() const

BME280PressureSensor *sensor::BME280Component::get_pressure_sensor() const

BME280HumiditySensor *sensor::BME280Component::get_humidity_sensor() const

void sensor::BME280Component::setup()

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

float sensor::BME280Component::get_setup_priority() const

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void sensor::BME280Component::update()
```

Protected Functions

```
float sensor::BME280Component::read_temperature_(int32_t *t_fine)
```

Read the temperature value and store the calculated ambient temperature in t_fine.

```
float sensor::BME280Component::read_pressure_(int32_t t_fine)
```

Read the pressure value in hPa using the provided t_fine value.

```
float sensor::BME280Component::read_humidity_(int32_t t_fine)
```

Read the humidity value in % using the provided t_fine value.

```
uint8_t sensor::BME280Componentread_u8(uint8_t register_)

uint16_t sensor::BME280Componentread_u16_le(uint8_t register_)

int16_t sensor::BME280Componentread_s16_le(uint8_t register_)
```

Protected Attributes

```
BME280CalibrationData sensor::BME280Componentcalibration_
BME280Oversampling sensor::BME280Componenttemperature_oversampling_ = {BME280_OVERSAMPLING_16X}
BME280Oversampling sensor::BME280Componentpressure_oversampling_ = {BME280_OVERSAMPLING_16X}
BME280Oversampling sensor::BME280Componenthumidity_oversampling_ = {BME280_OVERSAMPLING_16X}
BME280IIRFilter sensor::BME280Componentiir_filter_ = {BME280_IIR_FILTER_OFF}
BME280TemperatureSensor *sensor::BME280Componenttemperature_sensor_
BME280PressureSensor *sensor::BME280Componentpressure_sensor_
BME280HumiditySensor *sensor::BME280Componenthumidity_sensor_

enum sensor::BME280Oversampling
Enum listing all Oversampling values for the BME280.

Oversampling basically means measuring a condition multiple times. Higher oversampling values therefore increase the time required to read sensor values but increase accuracy.

    Values:

sensorBME280_OVERSAMPLING_NONE = 0b000
sensorBME280_OVERSAMPLING_1X = 0b001
sensorBME280_OVERSAMPLING_2X = 0b010
sensorBME280_OVERSAMPLING_4X = 0b011
sensorBME280_OVERSAMPLING_8X = 0b100
sensorBME280_OVERSAMPLING_16X = 0b101

enum sensor::BME280IIRFilter
Enum listing all Infinite Impulse Filter values for the BME280.

Higher values increase accuracy, but decrease response time.

    Values:

sensorBME280_IIR_FILTER_OFF = 0b000
sensorBME280_IIR_FILTER_2X = 0b001
sensorBME280_IIR_FILTER_4X = 0b010
sensorBME280_IIR_FILTER_8X = 0b011
sensorBME280_IIR_FILTER_16X = 0b100

using sensor::BME280TemperatureSensor = typedef sensor::EmptyPollingParentSensor<1, ICON_EMPTY, UNIT_C>
using sensor::BME280PressureSensor = typedef sensor::EmptyPollingParentSensor<1, ICON_GAUGE, UNIT_HPA>
using sensor::BME280HumiditySensor = typedef sensor::EmptyPollingParentSensor<1, ICON_WATER_PERCENT, UN
```

```
struct sensor::BME280CalibrationData
```

Internal struct storing the calibration values of an BME280.

Public Members

```
uint16_t sensor::BME280CalibrationDatat1
int16_t sensor::BME280CalibrationDatat2
int16_t sensor::BME280CalibrationDatat3
uint16_t sensor::BME280CalibrationDatap1
int16_t sensor::BME280CalibrationDatap2
int16_t sensor::BME280CalibrationDatap3
int16_t sensor::BME280CalibrationDatap4
int16_t sensor::BME280CalibrationDatap5
int16_t sensor::BME280CalibrationDatap6
int16_t sensor::BME280CalibrationDatap7
int16_t sensor::BME280CalibrationDatap8
int16_t sensor::BME280CalibrationDatap9
uint8_t sensor::BME280CalibrationDatah1
int16_t sensor::BME280CalibrationDatah2
uint8_t sensor::BME280CalibrationDatah3
int16_t sensor::BME280CalibrationDatah4
int16_t sensor::BME280CalibrationDatah5
int8_t sensor::BME280CalibrationDatah6
```

BME680 Temperature/Pressure/Humidity/Gas Sensor

Warning: This sensor is experimental has not been tested yet. If you can verify it works (or if it doesn't), notify me on [discord](#).

The BME680 sensor allows you to use your BME680 i2c-enabled temperature+pressure+humidity+gas sensor with esphomelib (datasheet, adafruit). It requires i2c to be setup to work.

Example Usage

```
// Basic
auto bme680 = App.make_bme680_sensor("BME680 Temperature",
                                         "BME680 Pressure",
                                         "BME680 Humidity",
                                         "BME680 Gas Resistance");
```

(continues on next page)

(continued from previous page)

```
// default is no iir filter
bme680.bme680->set_iir_filter(sensor::BME680_IIR_FILTER_15X);
// set heater to 200°C for 100ms, default is off
bme680.bme680->set_heater(200, 100);
```

See [Application::make_bme680_sensor\(\)](#).

API Reference

class sensor::BME680Component
Inherits from [PollingComponent](#), [I2CDevice](#)

Public Functions

sensor::BME680Component(*I2CComponent *parent*, *const std::string &temperature_name*, *const std::string &pressure_name*, *const std::string &humidity_name*, *const std::string &gas_resistance_name*, *uint8_t address = 0x76*, *uint32_t update_interval = 15000*)

void sensor::set_temperature_oversampling(*BME680Oversampling temperature_oversampling*)

Set the temperature oversampling value. Defaults to 16X.

void sensor::set_pressure_oversampling(*BME680Oversampling pressure_oversampling*)

Set the pressure oversampling value. Defaults to 16X.

void sensor::set_humidity_oversampling(*BME680Oversampling humidity_oversampling*)

Set the humidity oversampling value. Defaults to 16X.

void sensor::set_iir_filter(*BME680IIRFilter iir_filter*)

Set the IIR *Filter* value. Defaults to no IIR *Filter*.

void sensor::set_heater(*uint16_t heater_temperature*, *uint16_t heater_duration*)

Set how the internal heater should operate.

By default, the heater is off. If you want to have more reliable humidity and Gas Resistance values, you can however setup the heater with this method.

Parameters

- **heater_temperature**: The temperature of the heater in °C.
- **heater_duration**: The duration in ms that the heater should turn on for when measuring.

void sensor::setup()

Where the component's initialization should happen.

Analogous to Arduino's [setup\(\)](#). This method is guaranteed to only be called once. Defaults to doing noting.

```
float sensor::BME680Componentget_setup_priority() const
    priority of setup().
```

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void sensor::BME680Componentupdate()
BME680TemperatureSensor *sensor::BME680Componentget_temperature_sensor() const
BME680PressureSensor *sensor::BME680Componentget_pressure_sensor() const
BME680HumiditySensor *sensor::BME680Componentget_humidity_sensor() const
BME680GasResistanceSensor *sensor::BME680Componentget_gas_resistance_sensor()
                                         const
```

Protected Functions

```
uint8_t sensor::BME680Componentcalc_heater_resistance_(uint16_t temperature)
    Calculate the heater resistance value to send to the BME680 register.
```

```
uint8_t sensor::BME680Componentcalc_heater_duration_(uint16_t duration)
    Calculate the heater duration value to send to the BME680 register.
```

```
void sensor::BME680Componentread_data_()
    Read data from the BME680 and publish results.
```

```
float sensor::BME680Componentcalc_temperature_(uint32_t raw_temperature)
    Calculate the temperature in °C using the provided raw ADC value.
```

```
float sensor::BME680Componentcalc_pressure_(uint32_t raw_pressure)
    Calculate the pressure in hPa using the provided raw ADC value.
```

```
float sensor::BME680Componentcalc_humidity_(uint16_t raw_humidity)
    Calculate the relative humidity in % using the provided raw ADC value.
```

```
uint32_t sensor::BME680Componentcalc_gas_resistance_(uint16_t raw_gas, uint8_t range)
    Calculate the gas resistance in Ω using the provided raw ADC value.
```

```
uint32_t sensor::BME680Componentcalc_meas_duration_()
    Calculate how long the sensor will take until we can retrieve data.
```

Protected Attributes

```
BME680CalibrationData sensor::BME680Componentcalibration_
```

```
BME680Oversampling sensor::BME680Componenttemperature_oversampling_ = {BME680_OVERSAMPLING_16}
```

```
BME680Oversampling sensor::BME680Componentpressure_oversampling_ = {BME680_OVERSAMPLING_16X}
```

```
BME680Oversampling sensor::BME680Componenthumidity_oversampling_ = {BME680_OVERSAMPLING_16X}
```

```
BME680IIRFilter sensor::BME680Componentiir_filter_ = {BME680_IIR_FILTER_OFF}
```

```
uint16_t sensor::BME680Componentheater_temperature_ = {0}
uint16_t sensor::BME680Componentheater_duration_ = {0}
BME680TemperatureSensor *sensor::BME680Componenttemperature_sensor_
BME680PressureSensor *sensor::BME680Componentpressure_sensor_
BME680HumiditySensor *sensor::BME680Componenthumidity_sensor_
BME680GasResistanceSensor *sensor::BME680Componentgas_resistance_sensor_

enum sensor::BME680Oversampling
Enum listing all oversampling options for the BME680.

    Values:
    sensorBME680_OVERSAMPLING_NONE = 0b000
    sensorBME680_OVERSAMPLING_1X = 0b001
    sensorBME680_OVERSAMPLING_2X = 0b010
    sensorBME680_OVERSAMPLING_4X = 0b011
    sensorBME680_OVERSAMPLING_8X = 0b100
    sensorBME680_OVERSAMPLING_16X = 0b101

enum sensor::BME680IIRFilter
Enum listing all IIR Filter options for the BME680.

    Values:
    sensorBME680_IIR_FILTER_OFF = 0b000
    sensorBME680_IIR_FILTER_1X = 0b001
    sensorBME680_IIR_FILTER_3X = 0b010
    sensorBME680_IIR_FILTER_7X = 0b011
    sensorBME680_IIR_FILTER_15X = 0b100
    sensorBME680_IIR_FILTER_31X = 0b101
    sensorBME680_IIR_FILTER_63X = 0b110
    sensorBME680_IIR_FILTER_127X = 0b111

using sensor::BME680TemperatureSensor = typedef sensor::EmptyPollingParentSensor<1, ICON_EMPTY, UNIT_C>
using sensor::BME680PressureSensor = typedef sensor::EmptyPollingParentSensor<1, ICON_GAUGE, UNIT_HPA>
using sensor::BME680HumiditySensor = typedef sensor::EmptyPollingParentSensor<1, ICON_WATER_PERCENT, UN
```

Warning: doxygen typedef: Cannot find typedef “sensor::BME2680GasResistanceSensor” in doxygen xml output for project “esphomelib” from directory: ./_doxyxml/

```
struct sensor::BME680CalibrationData
Struct for storing calibration data for the BME680.
```

Public Members

```
uint16_t sensor::BME680CalibrationDatat1
uint16_t sensor::BME680CalibrationDatat2
uint8_t sensor::BME680CalibrationDatat3
uint16_t sensor::BME680CalibrationDatap1
int16_t sensor::BME680CalibrationDatap2
int8_t sensor::BME680CalibrationDatap3
int16_t sensor::BME680CalibrationDatap4
int16_t sensor::BME680CalibrationDatap5
int8_t sensor::BME680CalibrationDatap6
int8_t sensor::BME680CalibrationDatap7
int16_t sensor::BME680CalibrationDatap8
int16_t sensor::BME680CalibrationDatap9
int8_t sensor::BME680CalibrationDatap10
uint16_t sensor::BME680CalibrationDatah1
uint16_t sensor::BME680CalibrationDatah2
int8_t sensor::BME680CalibrationDatah3
int8_t sensor::BME680CalibrationDatah4
int8_t sensor::BME680CalibrationDatah5
uint8_t sensor::BME680CalibrationDatah6
int8_t sensor::BME680CalibrationDatah7
int8_t sensor::BME680CalibrationDatagh1
int16_t sensor::BME680CalibrationDatagh2
int8_t sensor::BME680CalibrationDatagh3
uint8_t sensor::BME680CalibrationDatares_heat_range
uint8_t sensor::BME680CalibrationDatares_heat_val
uint8_t sensor::BME680CalibrationDatarange_sw_err
float sensor::BME680CalibrationDatatfine
uint8_t sensor::BME680CalibrationDataambient_temperature
```

SHT3XD Temperature/Humidity Sensor

Warning: This sensor is experimental has not been tested yet. If you can verify it works (or if it doesn't), notify me on [discord](#).

The SHT3XD component allows you to use your SHT3x-DIS i2c-enabled temperature+humidity+gas sensor with esphomelib ([datasheet](#), [adafruit](#)). It requires i2c to be setup to work.

Example Usage

```
// Basic
auto sht3xd = App.make_sht3xd_sensor("SHT31D Temperature", "SHT31D Humidity");

// Advanced
// default accuracy is high
sht3xd.sht3xd->set_accuracy(sensor::SHT3XD_ACCURACY_LOW);
```

See [Application::make_sht3xd_sensor\(\)](#).

API Reference

class sensor::SHT3XDComponent

This class implements support for the SHT3x-DIS family of temperature+humidity i2c sensors.

Inherits from [PollingComponent](#), [I2CDevice](#)

Public Functions

```
sensor::SHT3XDComponent::SHT3XDComponent(I2CComponent *parent, const std::string &temperature_name, const std::string &humidity_name, uint8_t address = 0x44, uint32_t update_interval = 15000)
```

SHT3XDTemperatureSensor *sensor::SHT3XDComponent::get_temperature_sensor() const

SHT3XDHumiditySensor *sensor::SHT3XDComponent::get_humidity_sensor() const

void sensor::SHT3XDComponent::setup()

Where the component's initialization should happen.

Analogous to Arduino's [setup\(\)](#). This method is guaranteed to only be called once. Defaults to doing noting.

```
float sensor::SHT3XDComponent::get_setup_priority() const
priority of setup().
```

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void sensor::SHT3XDComponent::update()
```

Protected Functions

```
bool sensor::SHT3XDComponent::write_command(uint16_t command)
```

```
bool sensor::SHT3XDComponent::read_data(uint16_t *data, uint8_t len)
```

Protected Attributes

```
SHT3XDTemperatureSensor *sensor::SHT3XDComponenttemperature_sensor_
SHT3XDHumiditySensor *sensor::SHT3XDComponenthumidity_sensor_
```

Warning: doxygenenum: Cannot find enum “sensor::SHT3XDAccuracy” in doxygen xml output for project “esphomelib” from directory: ./_doxyxml/

```
class sensor::SHT3XDTemperatureSensor
```

Helper class exposing an SHT3xD temperature sensor with a unique id.

Inherits from *sensor::EmptyPollingParentSensor< 1, ICON_EMPTY, UNIT_C, SHT3XDComponent >*

Public Functions

```
sensor::SHT3XDTemperatureSensorSHT3XDTemperatureSensor(const std::string &name,
                                                       SHT3XDComponent *parent)
```

```
std::string sensor::SHT3XDTemperatureSensorunique_id()
```

A unique ID for this sensor, empty for no unique id.

See unique ID requirements: https://developers.home-assistant.io/docs/en/entity_registry_index.html#unique-id-requirements

Return The unique id as a string.

Protected Attributes

```
friend sensor::SHT3XDTemperatureSensor::SHT3XDComponent
```

```
std::string sensor::SHT3XDTemperatureSensorunique_id_
```

```
class sensor::SHT3XDHumiditySensor
```

Helper class exposing an SHT3xD humidity sensor with a unique id.

Inherits from *sensor::EmptyPollingParentSensor< 1, ICON_WATER_PERCENT, UNIT_PERCENT, SHT3XDComponent >*

Public Functions

```
sensor::SHT3XDHumiditySensorSHT3XDHumiditySensor(const std::string &name,
                                                   SHT3XDComponent *parent)
```

```
std::string sensor::SHT3XDHumiditySensorunique_id()
```

A unique ID for this sensor, empty for no unique id.

See unique ID requirements: https://developers.home-assistant.io/docs/en/entity_registry_index.html#unique-id-requirements

Return The unique id as a string.

Protected Attributes

```
friend sensor::SHT3XDHumiditySensor::SHT3XDComponent  
std::string sensor::SHT3XDHumiditySensorunique_id_
```

BH1750 Ambient Light Sensor

Warning: This sensor is experimental has not been tested yet. If you can verify it works (or if it doesn't), notify me on [discord](#).

The BH1750 sensor allows you to use your BH1750 i2c-enabled ambient light sensor with esphomelib ([datasheet](#)). It requires i2c to be setup to work.

Example Usage

```
// Basic  
auto bh1750 = App.make_bh1750_sensor("BH1750 Illuminance");  
  
// Advanced settings  
// default resolution is 0.5 LX  
bh1750.bh1750->set_resolution(sensor::BH1750_RESOLUTION_1P0_LX);
```

See [Application::make_bh1750_sensor\(\)](#).

API Reference

class sensor::BH1750Sensor

This class implements support for the i2c-based BH1750 ambient light sensor.

Inherits from [sensor::PollingSensorComponent](#), [I2CDevice](#)

Public Functions

```
sensor::BH1750SensorBH1750Sensor(I2CComponent *parent, const std::string &name, uint8_t  
address = 0x23, uint32_t update_interval = 15000)
```

void sensor::BH1750Sensorset_resolution(BH1750Resolution resolution)

Set the resolution of this sensor.

Possible values are:

- BH1750_RESOLUTION_4P0_LX
- BH1750_RESOLUTION_1P0_LX
- BH1750_RESOLUTION_0P5_LX (default)

Parameters

- **resolution:** The new resolution of the sensor.

```
void sensor::BH1750Sensorsetup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing nothing.

```
void sensor::BH1750Sensorupdate()
```

```
float sensor::BH1750Sensorget_setup_priority() const
```

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
std::string sensor::BH1750Sensorunit_of_measurement()
```

Override this to set the Home Assistant unit of measurement for this sensor.

Return "" to disable this feature.

Return The icon of this sensor, for example "°C".

```
std::string sensor::BH1750Sensoricon()
```

Override this to set the Home Assistant icon for this sensor.

Return "" to disable this feature.

Return The icon of this sensor, for example "mdi:battery".

```
int8_t sensor::BH1750Sensoraccuracy_decimals()
```

Return the accuracy in decimals for this sensor.

Protected Functions

```
void sensor::BH1750Sensorread_data_()
```

Protected Attributes

```
BH1750Resolution sensor::BH1750Sensorresolution_ = {BH1750_RESOLUTION_0P5_LX}
```

```
enum sensor::BH1750Resolution
```

Enum listing all resolutions that can be used with the BH1750.

Values:

```
sensorBH1750_RESOLUTION_4P0_LX = 0b00100011
```

```
sensorBH1750_RESOLUTION_1P0_LX = 0b00100000
```

```
sensorBH1750_RESOLUTION_0P5_LX = 0b00100001
```

MAX6675 K-Type Thermocouple Temperature Sensor

See `Application::make_max6675_sensor()`.

API Reference

```
class sensor::MAX6675Sensor  
Inherits from sensor::PollingSensorComponent
```

Public Functions

```
sensor::MAX6675Sensor::MAX6675Sensor(const std::string &name, GPIOPin *cs, GPIOPin  
*clock, GPIOPin *miso, uint32_t update_interval =  
15000)
```

```
void sensor::MAX6675Sensor::setup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

```
float sensor::MAX6675Sensor::get_setup_priority() const  
priority of setup().
```

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void sensor::MAX6675Sensor::update()
```

```
std::string sensor::MAX6675Sensor::unit_of_measurement()
```

Override this to set the Home Assistant unit of measurement for this sensor.

Return "" to disable this feature.

Return The icon of this sensor, for example "°C".

```
std::string sensor::MAX6675Sensor::icon()
```

Override this to set the Home Assistant icon for this sensor.

Return "" to disable this feature.

Return The icon of this sensor, for example "mdi:battery".

```
int8_t sensor::MAX6675Sensor::accuracy_decimals()
```

Return the accuracy in decimals for this sensor.

Protected Functions

```
void sensor::MAX6675Sensor::read_data_()
```

```
uint8_t sensor::MAX6675Sensor::read_spi_()
```

Protected Attributes

```
GPIOPin *sensor::MAX6675Sensorcs_
GPIOPin *sensor::MAX6675Sensorclock_
GPIOPin *sensor::MAX6675Sensormiso_
```

Rotary Encoder Sensor

See `Application::make_rotary_encoder_sensor()`.

API Reference

```
class sensor::RotaryEncoderSensor
Inherits from sensor::Sensor, Component
```

Public Functions

```
sensor::RotaryEncoderSensor::RotaryEncoderSensor(const std::string &name, GPIOPin *pin_a,
                                                GPIOPin *pin_b)
```

```
void sensor::RotaryEncoderSensor::set_resolution(RotaryEncoderResolution mode)
```

Set the resolution of the rotary encoder.

By default, this component will increment the counter by 1 with every A-B input cycle. You can however change this behavior to have more coarse resolutions like 4 counter increases per A-B cycle.

Parameters

- `mode`: The new mode of the encoder.

```
void sensor::RotaryEncoderSensor::set_reset_pin(const GPIOInputPin &pin_i)
```

```
void sensor::RotaryEncoderSensor::setup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

```
void sensor::RotaryEncoderSensor::loop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

```
std::string sensor::RotaryEncoderSensor::unit_of_measurement()
```

Override this to set the Home Assistant unit of measurement for this sensor.

Return "" to disable this feature.

Return The icon of this sensor, for example "°C".

```
std::string sensor::RotaryEncoderSensoricon()
    Override this to set the Home Assistant icon for this sensor.
    Return "" to disable this feature.
```

Return The icon of this sensor, for example "mdi:battery".

```
int8_t sensor::RotaryEncoderSensoraccuracy_decimals()
    Return the accuracy in decimals for this sensor.
```

Protected Functions

```
void sensor::RotaryEncoderSensorprocess_state_machine_()
    Process the state machine state of this rotary encoder. Called from encoder_isr_.
```

Protected Attributes

```
GPIOPin *sensor::RotaryEncoderSensorpin_a_
GPIOPin *sensor::RotaryEncoderSensorpin_b_
GPIOPin *sensor::RotaryEncoderSensorpin_i_ = {nullptr}
volatile int32_t sensor::RotaryEncoderSensorcounter_ = {0}
    Index pin, if this is not nullptr, the counter will reset to 0 once this pin is HIGH.
volatile bool sensor::RotaryEncoderSensorhas_changed_ = {true}
    The internal counter for steps.
uint16_t sensor::RotaryEncoderSensorstate_ = {0}
RotaryEncoderResolution sensor::RotaryEncoderSensorresolution_ = {ROTARY_ENCODER_1_PULSE_PER_C
```

Protected Static Functions

```
void sensor::RotaryEncoderSensorencoder_isr_()
    The ISR that handles pushing all interrupts to process_state_machine_ of all rotary encoders.
```

Template Sensor

See *Application::make_template_sensor()*.

API Reference

```
class sensor::TemplateSensor
Inherits from sensor::PollingSensorComponent
```

Public Functions

```
sensor::TemplateSensorTemplateSensor(const std::string &name,
                                      std::function<optional<float>> > &&uint32_t update_interval = 15000
void sensor::TemplateSensorupdate()
```

Protected Attributes

```
std::function<optional<float>> sensor::TemplateSensor::f_
```

API Reference

Sensor

class sensor::Sensor

Base-class for all sensors.

A sensor has unit of measurement and can use push_new_value to send out a new value with the specified accuracy.

Inherits from Nameable

Subclassed by *sensor::EmptySensor< 3, ICON_FLASH, UNIT_V >*, *sensor::EmptySensor< default_accuracy_decimals, default_icon, default_unit_of_measurement >*, *sensor::PollingSensorComponent*, *sensor::RotaryEncoderSensor*

Public Functions

```
sensor::SensorSensor(const std::string &name)
```

```
void sensor::Sensorset_unit_of_measurement(const std::string &unit_of_measurement)
```

Manually set the unit of measurement of this sensor.

By default the sensor's default defined by *unit_of_measurement()* is used.

Parameters

- **unit_of_measurement:** The unit of measurement, “” to disable.

```
void sensor::Sensorset_icon(const std::string &icon)
```

Manually set the icon of this sensor.

By default the sensor's default defined by *icon()* is used.

Parameters

- **icon:** The icon, for example “mdi:flash”. “” to disable.

```
void sensor::Sensorset_accuracy_decimals(int8_t accuracy_decimals)
```

Manually set the accuracy in decimals for this sensor.

By default, the sensor's default defined by *accuracy_decimals()* is used.

Parameters

- `accuracy_decimals`: The accuracy decimal that should be used.

```
void sensor::Sensoradd_filter(Filter *filter)
```

Add a filter to the filter chain. Will be appended to the back.

```
void sensor::Sensoradd_filters(const std::list<Filter *> &filters)
```

Add a list of vectors to the back of the filter chain.

This may look like:

```
sensor->add_filters({ LambdaFilter([&](float value) -> optional<float> { return 42/value; }),  
OffsetFilter(1), SlidingWindowMovingAverageFilter(15, 15), // average over last 15 values });
```

```
void sensor::Sensorset_filters(const std::list<Filter *> &filters)
```

Clear the filters and replace them by filters.

```
void sensor::Sensoradd_lambda_filter(lambda_filter_t filter)
```

Add a lambda filter to the back of the filter chain.

For example: `sensor->add_lambda_filter([](float value) -> optional<float> { return value * 42; })`;

If you return an unset Optional, the value will be discarded and no filters after this one will get the value.

```
void sensor::Sensoradd_offset_filter(float offset)
```

Helper to add a simple offset filter to the back of the filter chain.

This can be used to easily correct for sensors that have a small offset in their value reporting.

Parameters

- `offset`: The offset that will be added to each value.

```
void sensor::Sensoradd_multiply_filter(float multiplier)
```

Helper to add a simple multiply filter to the back of the filter chain.

Each value will be multiplied by this multiplier. Can be used to convert units easily. For example converting “pulses/min” to a more reasonable unit like kW.

Parameters

- `multiplier`: The multiplier each value will be multiplied with.

```
void sensor::Sensoradd_filter_out_value_filter(float values_to_filter_out)
```

Helper to add a simple filter that aborts the filter chain every time it receives a specific value.

Parameters

- `values_to_filter_out`: The value that should be filtered out.

```
void sensor::Sensoradd_sliding_window_average_filter(size_t window_size, size_t  
send_every)
```

Helper to make adding sliding window moving average filters a bit easier.

```
void sensor::Sensoradd_exponential_moving_average_filter(float alpha, size_t send_every)
```

Helper to make adding exponential decay average filters a bit easier.

```
void sensor::Sensorclear_filters()
    Clear the entire filter chain.

float sensor::Sensorget_value() const
    Get the latest filtered value from this sensor.

float sensor::Sensorget_raw_value() const
    Get the latest raw value from this sensor.

int8_t sensor::Sensorget_accuracy_decimals()
    Get the accuracy in decimals used by this MQTT Sensor, first checks override, then sensor.

std::string sensor::Sensorget_unit_of_measurement()
    Get the unit of measurements advertised to Home Assistant. First checks override, then sensor.

std::string sensor::Sensorget_icon()
    Get the icon advertised to Home Assistant.

void sensor::Sensorpush_new_value(float value)
    Push a new value to the MQTT front-end.

    Note that you should publish the raw value here, i.e. without any rounding as the user can later
    override this accuracy.
```

Parameters

- **value:** The floating point value.

```
std::string sensor::Sensorunit_of_measurement()
    Override this to set the Home Assistant unit of measurement for this sensor.

    Return "" to disable this feature.
```

Return The icon of this sensor, for example “°C”.

```
std::string sensor::Sensoricon()
    Override this to set the Home Assistant icon for this sensor.

    Return "" to disable this feature.
```

Return The icon of this sensor, for example “mdi:battery”.

```
uint32_t sensor::Sensorupdate_interval()
    Return with which interval the sensor is polled. Return 0 for non-polling mode.
```

```
int8_t sensor::Sensoraccuracy_decimals()
    Return the accuracy in decimals for this sensor.
```

```
void sensor::Sensoradd_on_value_callback(sensor_callback_t callback)
    Add a callback that will be called every time a filtered value arrives.
```

```
void sensor::Sensoradd_on_raw_value_callback(sensor_callback_t callback)
    Add a callback that will be called every time the sensor sends a raw value.
```

```
std::string sensor::Sensorunique_id()
    A unique ID for this sensor, empty for no unique id.
```

See unique ID requirements: https://developers.home-assistant.io/docs/en/entity_registry_index.html#unique-id-requirements

Return The unique id as a string.

```
SensorValueTrigger *sensor::Sensormake_value_trigger()  
RawSensorValueTrigger *sensor::Sensormake_raw_value_trigger()  
ValueRangeTrigger *sensor::Sensormake_value_range_trigger()
```

Public Members

float sensor::Sensor**value** = {NAN}
Stores the last filtered value. Public because of lambdas.

float sensor::Sensor**raw_value** = {NAN}
Stores the last raw value. Public because of lambdas.

Protected Functions

```
void sensor::Sensorsend_value_to_frontend(float value)
```

Protected Attributes

friend sensor::Sensor::Filter

friend sensor::Sensor::MQTTSensorComponent

CallbackManager<void(float)> sensor::Sensor**raw_callback_**
Storage for raw value callbacks.

CallbackManager<void(float)> sensor::Sensor**callback_**
Storage for filtered value callbacks.

optional<std::string> sensor::Sensor**unit_of_measurement_**
Override the unit of measurement.

optional<std::string> sensor::Sensor**icon_**

optional<int8_t> sensor::Sensor**accuracy_decimals_**
Override the icon advertised to Home Assistant, otherwise sensor's icon will be used.

Override the accuracy in decimals, otherwise the sensor's values will be used.

Filter *sensor::Sensor**filter_list_** = {nullptr}
Store all active filters.

```
using sensor::sensor_callback_t = typedef std::function<void(float)>
```

```
class sensor::PollingSensorComponent
```

Inherits from *PollingComponent*, *sensor::Sensor*

Subclassed by *sensor::ADCSensorComponent*, *sensor::BH1750Sensor*, *sensor::ESP32HallSensor*,
sensor::MAX6675Sensor, *sensor::PulseCounterSensorComponent*, *sensor::TemplateSensor*, *sen-*
sor::TSL2561Sensor, *sensor::UltrasonicSensorComponent*

Public Functions

```
sensor::PollingSensorComponentPollingSensorComponent(const std::string &name, uint32_t
update_interval)
```

```
uint32_t sensor::PollingSensorComponentupdate_interval()
```

Return with which interval the sensor is polled. Return 0 for non-polling mode.

```
template <int8_t default_accuracy_decimals, const char * default_icon, const char * default_unit_of_measurement>
class sensor::EmptySensor
```

Inherits from `sensor::Sensor`

Subclassed by `sensor::EmptyPollingParentSensor< 1, ICON_EMPTY, UNIT_C, DallasComponent >`, `sensor::EmptyPollingParentSensor< 1, ICON_EMPTY, UNIT_C, SHT3XDComponent >`, `sensor::EmptyPollingParentSensor< 1, ICON_WATER_PERCENT, UNIT_PERCENT, SHT3XDComponent >`, `sensor::EmptyPollingParentSensor< default_accuracy_decimals, default_icon, default_unit_of_measurement, ParentType >`

Public Functions

```
sensor::EmptySensorEmptySensor(const std::string &name)
```

```
std::string sensor::EmptySensorunit_of_measurement()
```

Override this to set the Home Assistant unit of measurement for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “°C”.

```
std::string sensor::EmptySensoricon()
```

Override this to set the Home Assistant icon for this sensor.

Return “” to disable this feature.

Return The icon of this sensor, for example “mdi:battery”.

```
int8_t sensor::EmptySensoraccuracy_decimals()
```

Return the accuracy in decimals for this sensor.

```
template <int8_t default_accuracy_decimals, const char * default_icon, const char * default_unit_of_measurement, class
class sensor::EmptyPollingParentSensor
```

Inherits from `sensor::EmptySensor< default_accuracy_decimals, default_icon, default_unit_of_measurement >`

Public Functions

```
sensor::EmptyPollingParentSensorEmptyPollingParentSensor(const std::string &name, ParentType *parent)
```

```
uint32_t sensor::EmptyPollingParentSensorupdate_interval()
```

Return with which interval the sensor is polled. Return 0 for non-polling mode.

Protected Attributes

```
ParentType *sensor::EmptyPollingParentSensorparent_
const char sensor::ICON_EMPTY = ""
const char sensor::ICON_WATER_PERCENT = "mdi:water-percent"
const char sensor::ICON_GAUGE = "mdi:gauge"
const char sensor::ICON_FLASH = "mdi:flash"
const char sensor::ICON_SCREEN_ROTATION = "mdi:screen-rotation"
const char sensor::ICON_BRIEFCASE_DOWNLOAD = "mdi:briefcase-download"
const char sensor::UNIT_C = "°C"
const char sensor::UNIT_PERCENT = "%"
const char sensor::UNIT_HPA = "hPa"
const char sensor::UNIT_V = "V"
const char sensor::UNIT_DEGREES_PER_SECOND = "°/s"
const char sensor::UNIT_M_PER_S_SQUARED = "m/s²"
```

Filter

```
class sensor::Filter
```

Apply a filter to sensor values such as moving average.

This class is purposefully kept quite simple, since more complicated filters should really be done with the filter sensor in Home Assistant.

Subclassed by *sensor::DebounceFilter*, *sensor::DeltaFilter*, *sensor::ExponentialMovingAverageFilter*, *sensor::FilterOutNANFilter*, *sensor::FilterOutValueFilter*, *sensor::HeartbeatFilter*, *sensor::LambdaFilter*, *sensor::MultiplyFilter*, *sensor::OffsetFilter*, *sensor::OrFilter*, *sensor::SlidingWindowMovingAverageFilter*, *sensor::ThrottleFilter*, *sensor::UniqueFilter*

Public Functions

```
virtual optional<float> sensor::Filternew_value(float value) = 0
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

```
sensor::Filter~Filter()
```

```
void sensor::Filterinitialize(std::function<void> float
> &&output
```

```
void sensor::Filter(float value)
uint32_t sensor::Filterexpected_interval(uint32_t input)
    Return the amount of time that this filter is expected to take based on the input time interval.
```

Protected Attributes

```
friend sensor::Filter::Sensor
friend sensor::Filter::MQTTSensorComponent
std::function<void(float)> sensor::Filteroutput_
Filter *sensor::Filternext_ = {nullptr}
class sensor::SlidingWindowMovingAverageFilter
Simple sliding window moving average filter.

Essentially just takes the average of the last window_size values and pushes them out every
send_every.

Inherits from sensor::Filter
```

Public Functions

```
sensor::SlidingWindowMovingAverageFilterSlidingWindowMovingAverageFilter(size_t      window_size,
size_t      send_every)
```

Construct a *SlidingWindowMovingAverageFilter*.

Parameters

- window_size**: The number of values that should be averaged.
- send_every**: After how many sensor values should a new one be pushed out.

```
optional<float> sensor::SlidingWindowMovingAverageFilternew_value(float value)
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- value**: The new value.

```
size_t sensor::SlidingWindowMovingAverageFilterget_send_every() const
void sensor::SlidingWindowMovingAverageFilterset_send_every(size_t send_every)
size_t sensor::SlidingWindowMovingAverageFilterget_window_size() const
void sensor::SlidingWindowMovingAverageFilterset_window_size(size_t window_size)
uint32_t sensor::SlidingWindowMovingAverageFilterexpected_interval(uint32_t input)
    Return the amount of time that this filter is expected to take based on the input time interval.
```

Protected Attributes

```
SlidingWindowMovingAverage<float> sensor::SlidingWindowMovingAverageFiltervalue_average_
size_t sensor::SlidingWindowMovingAverageFiltersend_every_
size_t sensor::SlidingWindowMovingAverageFiltersend_at_

class sensor::ExponentialMovingAverageFilter
```

Simple exponential moving average filter.

Essentially just takes the average of the last few values using exponentially decaying weights. Use alpha to adjust decay rate.

Inherits from [sensor::Filter](#)

Public Functions

```
sensor::ExponentialMovingAverageFilterExponentialMovingAverageFilter(float alpha, size_t
send_every)
```

```
optional<float> sensor::ExponentialMovingAverageFilternew_value(float value)
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

```
size_t sensor::ExponentialMovingAverageFilterget_send_every() const
```

```
void sensor::ExponentialMovingAverageFilterset_send_every(size_t send_every)
```

```
float sensor::ExponentialMovingAverageFilterget_alpha() const
```

```
void sensor::ExponentialMovingAverageFilterset_alpha(float alpha)
```

```
uint32_t sensor::ExponentialMovingAverageFilterexpected_interval(uint32_t input)
```

Return the amount of time that this filter is expected to take based on the input time interval.

Protected Attributes

```
ExponentialMovingAverage sensor::ExponentialMovingAverageFiltervalue_average_
ExponentialMovingAverage sensor::ExponentialMovingAverageFilteraccuracy_average_
size_t sensor::ExponentialMovingAverageFiltersend_every_
size_t sensor::ExponentialMovingAverageFiltersend_at_
```

```
using sensor::lambda_filter_t = typedef std::function<optional<float>(float)>
```

class sensor::LambdaFilter

This class allows for creation of simple template filters.

The constructor accepts a lambda of the form float -> optional<float>. It will be called with each new value in the filter chain and returns the modified value that shall be passed down the filter chain. Returning an empty Optional means that the value shall be discarded.

Inherits from [sensor::Filter](#)

Public Functions

```
sensor::LambdaFilterLambdaFilter(lambda_filter_t lambda_filter)
```

```
optional<float> sensor::LambdaFilternew_value(float value)
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

```
const lambda_filter_t &sensor::LambdaFilterget_lambda_filter() const
```

```
void sensor::LambdaFilterset_lambda_filter(const lambda_filter_t &lambda_filter)
```

Protected Attributes

```
lambda_filter_t sensor::LambdaFilterlambda_filter_
```

class sensor::OffsetFilter

A simple filter that adds **offset** to each value it receives.

Inherits from [sensor::Filter](#)

Public Functions

```
sensor::OffsetFilterOffsetFilter(float offset)
```

```
optional<float> sensor::OffsetFilternew_value(float value)
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

Protected Attributes

```
float sensor::OffsetFilteroffset_
class sensor::MultiplyFilter
A simple filter that multiplies to each value it receives by multiplier.
Inherits from sensor::Filter
```

Public Functions

```
sensor::MultiplyFilterMultiplyFilter(float multiplier)
```

optional<float> `sensor::MultiplyFilternew_value`(float `value`)
This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- `value`: The new value.

Protected Attributes

```
float sensor::MultiplyFiltermultiplier_
class sensor::FilterOutValueFilter
A simple filter that only forwards the filter chain if it doesn't receive value_to_filter_out.
Inherits from sensor::Filter
```

Public Functions

```
sensor::FilterOutValueFilterFilterOutValueFilter(float values_to_filter_out)
```

optional<float> `sensor::FilterOutValueFilternew_value`(float `value`)
This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- `value`: The new value.

Protected Attributes

```
float sensor::FilterOutValueFiltervalue_to_filter_out_
class sensor::FilterOutNANfilter
A simple filter that only forwards the filter chain if it doesn't receive nan.
Inherits from sensor::Filter
```

Public Functions

optional<float> sensor::FilterOutNANfilternew_value(float value)

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

```
class sensor::ThrottleFilter
Inherits from sensor::Filter
```

Public Functions

sensor::ThrottleFilterThrottleFilter(uint32_t min_time_between_inputs)

optional<float> sensor::ThrottleFilternew_value(float value)

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

Protected Attributes

```
uint32_t sensor::ThrottleFilterlast_input_ = {0}
uint32_t sensor::ThrottleFiltermin_time_between_inputs_
class sensor::HeartbeatFilter
Inherits from sensor::Filter, Component
```

Public Functions

```
sensor::HeartbeatFilterHeartbeatFilter(uint32_t time_period)
```

```
void sensor::HeartbeatFiltersetup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing nothing.

```
optional<float> sensor::HeartbeatFilternew_value(float value)
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- `value`: The new value.

```
uint32_t sensor::HeartbeatFilterexpected_interval(uint32_t input)
```

Return the amount of time that this filter is expected to take based on the input time interval.

Protected Attributes

```
uint32_t sensor::HeartbeatFiltertime_period_
```

```
float sensor::HeartbeatFilterlast_input_
```

```
class sensor::DebounceFilter
```

Inherits from `sensor::Filter`, `Component`

Public Functions

```
sensor::DebounceFilterDebounceFilter(uint32_t time_period)
```

```
optional<float> sensor::DebounceFilternew_value(float value)
```

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- `value`: The new value.

Protected Attributes

```
uint32_t sensor::DebounceFiltertime_period_
```

```
class sensor::DeltaFilter
```

Inherits from `sensor::Filter`

Public Functions

`sensor::DeltaFilterDeltaFilter(float min_delta)`

optional<float> `sensor::DeltaFilternew_value(float value)`

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- `value`: The new value.

Protected Attributes

`float sensor::DeltaFiltermin_delta_`

`float sensor::DeltaFilterlast_value_ = {NAN}`

class `sensor::OrFilter`

Inherits from `sensor::Filter`

Public Functions

`sensor::OrFilterOrFilter(std::list<Filter *> filters)`

`sensor::OrFilter~OrFilter()`

`void sensor::OrFilterinitialize(std::function<void> float > &&output`

`uint32_t sensor::OrFilterexpected_interval(uint32_t input)`

Return the amount of time that this filter is expected to take based on the input time interval.

optional<float> `sensor::OrFilternew_value(float value)`

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- `value`: The new value.

Protected Attributes

`std::list<Filter *> sensor::OrFilterfilters_`

class `sensor::UniqueFilter`

Inherits from `sensor::Filter`

Public Functions

optional<float> sensor::*UniqueFilter***new_value**(float *value*)

This will be called every time the filter receives a new value.

It can return an empty optional to indicate that the filter chain should stop, otherwise the value in the filter will be passed down the chain.

Return An optional float, the new value that should be pushed out.

Parameters

- **value:** The new value.

Protected Attributes

float sensor::*UniqueFilter***last_value_** = {NAN}

MQTTSensorComponent

class sensor::MQTTSensorComponent

Class that exposes sensors to the MQTT frontend.

Inherits from *mqtt::MQTTComponent*

Public Functions

sensor::MQTTSensorComponent**MQTTSensorComponent**(*Sensor* **sensor*)

Construct this *MQTTSensorComponent* instance with the provided friendly_name and sensor.

Note the sensor is never stored and is only used for initializing some values of this class. If sensor is nullptr, then automatic initialization of these fields is disabled.

Parameters

- **sensor:** The sensor, this can be null to disable automatic setup.

void sensor::*MQTTSensorComponent***set_expire_after**(uint32_t *expire_after*)

Setup an expiry, 0 disables it.

void sensor::*MQTTSensorComponent***disable_expire_after**()

Disable Home Assistant value expiry.

void sensor::*MQTTSensorComponent***send_discovery**(JsonBuffer &*buffer*, JsonObject &*root*,

mqtt::SendDiscoveryConfig &*config*)

Send discovery info to the Home Assistant, override this.

void sensor::*MQTTSensorComponent***setup**()

Override setup.

uint32_t sensor::*MQTTSensorComponent***get_expire_after**() const

Get the expire_after in milliseconds used for Home Assistant discovery, first checks override.

Protected Functions

`std::string sensor::MQTTSensorComponentcomponent_type() const`
 Override for MQTTComponent, returns “sensor”.

`std::string sensor::MQTTSensorComponentfriendly_name() const`
 Get the friendly name of this MQTT component.

Protected Attributes

`Sensor *sensor::MQTTSensorComponentsensor_`
`optional<uint32_t> sensor::MQTTSensorComponentexpire_after_`

3.3.3 Binary Sensor

In esphomelib, every component that exposes a binary state, is a *BinarySensor*.

To create your own binary sensor, simply subclass *BinarySensor* and call *BinarySensor::publish_state()* to tell the frontend that you have a new state. Inversion is automatically done for you when publishing state and can be changed by the user with *BinarySensor::set_inverted()*.

Supported Binary Sensors

GPIO Binary Sensor

Example Usage

```
// Basic
App.make_gpio_binary_sensor("Window Open", 36);
// Custom pinMode
App.make_gpio_binary_sensor("Window Open", GPIOInputPin(36, INPUT_PULLUP));
```

See *Application::make_gpio_binary_sensor()*.

API Reference

`class binary_sensor::GPIOBinarySensorComponent`
 Simple binary_sensor component for a GPIO pin.

This class allows you to observe the digital state of a certain GPIO pin.

Inherits from *binary_sensor::BinarySensor, Component*

Public Functions

`binary_sensor::GPIOBinarySensorComponentGPIOBinarySensorComponent(const std::string &name, GPIOPin *pin)`

Construct a *GPIOBinarySensorComponent*.

Parameters

- `name`: The name for this binary sensor.
- `pin`: The input pin, can either be an integer or `GPIOInputPin`.

```
void binary_sensor::GPIOBinarySensorComponentsetup()  
    Setup pin.  
  
float binary_sensor::GPIOBinarySensorComponentget_setup_priority() const  
    Hardware priority.  
  
void binary_sensor::GPIOBinarySensorComponentloop()  
    Check sensor.
```

Protected Attributes

```
GPIOPin *binary_sensor::GPIOBinarySensorComponentpin_
```

Node Status Binary Sensor

This binary sensor platform allows you to create a switch that uses the birth and last will messages by the MQTT client to show an ON/OFF state of the node.

Example Usage

```
App.make_status_binary_sensor("Livingroom Node Status");
```

See `Application::make_status_binary_sensor()`.

API Reference

```
class binary_sensor::StatusBinarySensor
```

Simple binary sensor that reports the online/offline state of the node using MQTT.

Most of the magic doesn't happen here, but in `Application.make_status_binary_sensor()`.

Inherits from `binary_sensor::BinarySensor`

Public Functions

```
binary_sensor::StatusBinarySensor::StatusBinarySensor(const std::string &name)  
    Construct the status binary sensor.
```

Protected Functions

```
std::string binary_sensor::StatusBinarySensor::device_class()  
    "connectivity" device class.
```

ESP32 Touch Binary Sensor

Example Usage

```
auto *touch = App.make_esp32_touch_component();
touch->set_setup_mode(true);
touch->set_iir_filter(1000);
App.register_binary_sensor(touch_hub->make_touch_pad("ESP32 Touch Pad 9", TOUCH_PAD_NUM9, 1000));
```

See [Application::make_esp32_touch_component\(\)](#).

API Reference

class binary_sensor::ESP32TouchComponent

This class is a hub for all touch pads on the ESP32 and only one of these can exist at once.

With this component you can use any of the touch-enabled pins of the ESP32 to detected touches using the internal touch peripheral. It works by rapidly charging the touch pads and measuring how long it takes for them to discharge again. This process is done many times on all touch pins and the result is a 16-bit unsigned value. This value is usually around 0-800 when a touch has been detected and around 1000-2000 when no touch has been detected. So the smaller the value, the more likely a touch is happening. Note that this value varies greatly between all touch pins and also between boards.

This component uses the measured touch value and applies a simple threshold. If the measured value is below the threshold, the binary sensor for a touch pad will go ON, and if it's above, the binary sensor will report an OFF state again.

If you notice the values have a lot of noise for your device and cause many false-positive touch events, you can optionally setup an IIR Filter for globally across all touch pads. If this filter value is large (100 ms+) the touch pad will become less responsive and only trigger if the user touches it for a longer time, but fewer ghost touch events will happen. The IIR Filter is off by default.

Additionally, if you want to tinker around with the internal values for esp-idf in order to improve performance and decrease power usage, you can use the `set_sleep_cycle`, `set_meas_cycle()`, `set_low_voltage_reference()`, `set_high_voltage_reference()` and `set_voltage_attenuation()` methods.

When setting up the threshold values, it can be useful to get the “real” measured touch values and not just the binary ON/OFF state. For this there’s the `set_setup_mode()` method which can be used to enable, as the name implies, a setup mode. In this mode the component will regularly send out the measured touch values in the logs using the DEBUG log level. Be sure to turn it off again afterwards!

Inherits from [Component](#)

Public Functions

```
ESP32TouchBinarySensor *binary_sensor::ESP32TouchComponent::make_touch_pad(const
std::string
&name,
touch_pad_t
touch_pad,
uint16_t
threshold)
```

Create a single touch pad binary sensor using the provided threshold value.

Values for touch_pad can be:

- TOUCH_PAD_NUM0 (GPIO 4)
- TOUCH_PAD_NUM1 (GPIO 0)
- TOUCH_PAD_NUM2 (GPIO 2)
- TOUCH_PAD_NUM3 (GPIO 15)
- TOUCH_PAD_NUM4 (GPIO 13)
- TOUCH_PAD_NUM5 (GPIO 12)
- TOUCH_PAD_NUM6 (GPIO 14)
- TOUCH_PAD_NUM7 (GPIO 27)
- TOUCH_PAD_NUM8 (GPIO 33)
- TOUCH_PAD_NUM9 (GPIO 32)

Additionally, you need to register the touch pad in the application instance like this:

```
App.register_binary_sensor(
    touch_hub->make_touch_pad("ESP32 Touch Pad 9", TOUCH_PAD_NUM9, 1000)
);
```

See [set_setup_mode](#) for finding values for the threshold parameter.

Return An *ESP32TouchBinarySensor* that needs to be registered in the *Application* instance.

Parameters

- **name**: The name of the binary sensor.
- **touch_pad**: The touch pad to use.
- **threshold**: The threshold touch value for detecting touches, smaller means higher probability that a touch is active.

```
void binary_sensor::ESP32TouchComponentset_setup_mode(bool setup_mode)
```

Put this hub into a setup mode in which all touch pad readings will be fed into the debug logs.

Be sure to turn this off again after determining the required threshold values for each touch pad!

Parameters

- **setup_mode**: The new setup mode, default is OFF (false).

```
void binary_sensor::ESP32TouchComponentset_iir_filter(uint32_t iir_filter)
```

Setup an infinite impulse response filter to improve accuracy of the touch readings.

See [Filtering Pulses](#). Good values for this can be 20ms or so, but often the default of no filter works just fine.

Parameters

- **iir_filter**: The new IIR Filter period in ms. Default is OFF (0).

```
void binary_sensor::ESP32TouchComponentset_sleep_duration(uint16_t sleep_duration)
Set how many RTC SLOW clock cycles (150kHz) the touch peripheral should sleep between
measurements.
```

Smaller values can improve power usage, but negatively affect response times.

See *set_measurement_duration*

Parameters

- **sleep_duration:** The sleep length. Default is 4096 (27.3 ms)

```
void binary_sensor::ESP32TouchComponentset_measurement_duration(uint16_t
                                                               meas_cycle)
```

Set how many APB clock cycles (8MHz) the touch pad peripheral should stay active for measuring.

Higher values can improve response times, but negatively affect power usage.

See *set_sleep_duration*

Parameters

- **meas_cycle:** The measurement cycle length. Default is the maximum of 65535 (8ms).

```
void binary_sensor::ESP32TouchComponentset_low_voltage_reference(touch_low_volt_t
                                                               low_voltage_reference)
```

Set the touch sensor low voltage reference for discharging.

Use this together with *set_high_voltage_reference* to improve touch detection. A higher difference between the high and low voltage means that more discharge/charge cycles can be made and thus increase the sensitivity. However, a smaller gap usually also means that more noise will be generated, but that can be alleviated with the IIR software filter.

Possible values are:

- **TOUCH_LVOLT_OV5** (default)
- **TOUCH_LVOLT_OV6**
- **TOUCH_LVOLT_OV7**
- **TOUCH_LVOLT_OV8**

See *set_high_voltage_reference* and *set_voltage_attenuation*

Parameters

- **low_voltage_reference:** The new low voltage reference.

```
void binary_sensor::ESP32TouchComponentset_high_voltage_reference(touch_high_volt_t
                                                               high_voltage_reference)
```

Set the touch sensor high voltage reference for charging.

Use this together with *set_low_voltage_reference* to improve touch detection. A higher difference between the high and low voltage means that more discharge/charge cycles can be made and thus increase the sensitivity. However, a smaller gap usually also means that more noise will be generated, but that can be alleviated with the IIR software filter.

Possible values are:

- **TOUCH_HVOLT_2V4**

- TOUCH_HVOLT_2V5
- TOUCH_HVOLT_2V6
- TOUCH_HVOLT_2V7 (default)

See `set_low_voltage_reference` and `set_voltage_attenuation`

Parameters

- `high_voltage_reference`: The new high voltage reference.

```
void binary_sensor::ESP32TouchComponentset_voltage_attenuation(touch_volt_atten_t  
voltage_attenuation)
```

Set the voltage attenuation of the touch peripheral.

Possible values are:

- TOUCH_HVOLT_ATTEN_1V5
- TOUCH_HVOLT_ATTEN_1V
- TOUCH_HVOLT_ATTEN_0V5
- TOUCH_HVOLT_ATTEN_0V (default)

See `set_low_voltage_reference` and `set_high_voltage_reference`

Parameters

- `voltage_attenuation`: The new voltage attenuation

```
void binary_sensor::ESP32TouchComponentsetup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

```
void binary_sensor::ESP32TouchComponentloop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

```
float binary_sensor::ESP32TouchComponentget_setup_priority() const
```

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

Protected Functions

```
bool binary_sensor::ESP32TouchComponentiir_filter_enabled() const
```

Is the IIR filter enabled?

Protected Attributes

```
uint16_t binary_sensor::ESP32TouchComponentsleep_cycle_ = {4096}
uint16_t binary_sensor::ESP32TouchComponentmeas_cycle_ = {65535}
touch_low_volt_t binary_sensor::ESP32TouchComponentlow_voltage_reference_ = {TOUCH_LVOLT_0V5}
touch_high_volt_t binary_sensor::ESP32TouchComponenthigh_voltage_reference_ = {TOUCH_HVOLT_2V7}
touch_volt_atten_t binary_sensor::ESP32TouchComponentvoltage_attenuation_ = {TOUCH_HVOLT_ATTEN_0}
std::vector<ESP32TouchBinarySensor *> binary_sensor::ESP32TouchComponentchildren_
bool binary_sensor::ESP32TouchComponentsetup_mode_ = {false}
uint32_t binary_sensor::ESP32TouchComponentiir_filter_ = {0}

class binary_sensor::ESP32TouchBinarySensor
    Simple helper class to expose a touch pad value as a binary sensor.
```

Inherits from *binary_sensor::BinarySensor*

Public Functions

```
binary_sensor::ESP32TouchBinarySensorESP32TouchBinarySensor(const std::string &name,
                                                               touch_pad_t touch_pad,
                                                               uint16_t threshold)
touch_pad_t binary_sensor::ESP32TouchBinarySensorget_touch_pad() const
uint16_t binary_sensor::ESP32TouchBinarySensorget_threshold() const
```

Protected Attributes

```
touch_pad_t binary_sensor::ESP32TouchBinarySensortouch_pad_
uint16_t binary_sensor::ESP32TouchBinarySensorthreshold_
```

Template Binary Sensor

See *Application::make_template_binary_sensor()*.

API Reference

```
class binary_sensor::TemplateBinarySensor
    Inherits from Component, binary_sensor::BinarySensor
```

Public Functions

```
binary_sensor::TemplateBinarySensorTemplateBinarySensor(const std::string &name,
                                                               std::function<optional<bool>>
                                                               > &&f)
```

```
void binary_sensor::TemplateBinarySensorloop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

Protected Attributes

```
std::function<optional<bool>>&& binary_sensor::TemplateBinarySensor::f_
```

Example Usage

```
// Basic
App.register_binary_sensor(custom_binary_sensor);
// GPIO Binary Sensor
App.make_gpio_binary_sensor("Window Open", 36);
```

See `Application::register_binary_sensor()` and `Application::make_gpio_binary_sensor()`.

API Reference

BinarySensor

```
class binary_sensor::BinarySensor
```

Base class for all binary_sensor-type classes.

This class includes a callback that components such as MQTT can subscribe to for state changes. The sub classes should notify the front-end of new states via the `publish_state()` method which handles inverted inputs for you.

Inherits from Nameable

Subclassed by `binary_sensor::ESP32TouchBinarySensor`, `binary_sensor::GPIOBinarySensorComponent`, `binary_sensor::StatusBinarySensor`, `binary_sensor::TemplateBinarySensor`, `ESP32BLEDevice`, `switch_::Switch`

Public Functions

```
binary_sensor::BinarySensorBinarySensor(const std::string &name)
```

Construct a binary sensor with the specified name.

Parameters

- `name`: Name of this binary sensor.

```
void binary_sensor::BinarySensoradd_on_state_callback(binary_callback_t &&callback)
```

Set callback for state changes.

Parameters

- `callback`: The void(bool) callback.

```
void binary_sensor::BinarySensorset_inverted(bool inverted)
    Set the inverted state of this binary sensor. If true, each published value will be inverted.

void binary_sensor::BinarySensorpublish_state(bool state)
    Publish a new state.

Inverted input is handled by this method and sub-classes don't need to worry about inverting themselves.
```

Parameters

- **state:** The new state.

```
bool binary_sensor::BinarySensorget_value() const
    Get the current boolean value of this binary sensor.

void binary_sensor::BinarySensorset_device_class(const std::string &device_class)
    Manually set the Home Assistant device class (see esphomelib::binary_sensor::device_class)

std::string binary_sensor::BinarySensorget_device_class()
    Get the device class for this binary sensor, using the manual override if specified.

PressTrigger *binary_sensor::BinarySensormake_press_trigger()

ReleaseTrigger *binary_sensor::BinarySensormake_release_trigger()

ClickTrigger *binary_sensor::BinarySensormake_click_trigger(uint32_t min_length,
                                                       uint32_t max_length)

DoubleClickTrigger *binary_sensor::BinarySensormake_double_click_trigger(uint32_t
                                                               min_length,
                                                               uint32_t
                                                               max_length)

bool binary_sensor::BinarySensoris_inverted() const
    Return whether all states of this binary sensor should be inverted.
```

Public Members

```
bool binary_sensor::BinarySensorvalue = {false}
```

Protected Functions

```
std::string binary_sensor::BinarySensordevice_class()
    Get the default device class for this sensor, or empty string for no default.
```

Protected Attributes

```
CallbackManager<void(bool)> binary_sensor::BinarySensorstate_callback_ = {}
bool binary_sensor::BinarySensorinverted_ = {false}
bool binary_sensor::BinarySensorfirst_value_ = {true}
optional<std::string> binary_sensor::BinarySensordevice_class_ = {}
    Stores the override of the device class.
```

MQTTBinarySensorComponent

```
class binary_sensor::MQTTBinarySensorComponent
    Simple MQTT front-end component for a binary_sensor.
```

After construction of this class, it should be connected to the *BinarySensor* by setting the callback returned by `create_on_new_state_callback()` in `BinarySensor::on_new_state()`.

Inherits from *mqtt::MQTTComponent*

Subclassed by *switch_::MQTTSwitchComponent*

Public Functions

```
binary_sensor::MQTTBinarySensorComponent::MQTTBinarySensorComponent(BinarySensor
    *binary_sensor)
```

Construct a *MQTTBinarySensorComponent*.

Parameters

- `binary_sensor`: The binary sensor.

```
void binary_sensor::MQTTBinarySensorComponent::setup()
    Send discovery.
```

```
void binary_sensor::MQTTBinarySensorComponent::send_discovery(JsonBuffer &buffer,
    JsonObject &obj,
    mqtt::SendDiscoveryConfig &config)
```

Send Home Assistant discovery info.

```
const std::string &binary_sensor::MQTTBinarySensorComponent::get_payload_on() const
    Get the payload this binary sensor uses for an ON value.
```

```
void binary_sensor::MQTTBinarySensorComponent::set_payload_on(std::string payload_on)
    Set the custom payload this binary sensor uses for an ON value.
```

```
const std::string &binary_sensor::MQTTBinarySensorComponent::get_payload_off() const
    Get the payload this binary sensor uses for an OFF value.
```

```
void binary_sensor::MQTTBinarySensorComponent::set_payload_off(std::string payload_off)
    Set the custom payload this binary sensor uses for an OFF value.
```

Protected Functions

```
std::string binary_sensor::MQTTBinarySensorComponent::friendly_name() const
    Return the friendly name of this binary sensor.
```

```
std::string binary_sensor::MQTTBinarySensorComponent::component_type() const
    "binary_sensor" component type.
```

```
void binary_sensor::MQTTBinarySensorComponent::publish_state(bool state)
```

Protected Attributes

```
BinarySensor *binary_sensor::MQTTBinarySensorComponentbinary_sensor_
std::string binary_sensor::MQTTBinarySensorComponentpayload_on_ = {"ON"}
std::string binary_sensor::MQTTBinarySensorComponentpayload_off_ = {"OFF"}
```

3.3.4 Output

The *output* namespace contains all peripheral output components.

GPIO Binary Output

Example Usage

```
// Create a binary output, *not a switch*
App.make_gpio_output(33);
// Custom pinMode
App.make_gpio_output(GPIOOutputPin(33, OUTPUT_OPEN_DRAIN));
```

See *Application::make_gpio_output()* (and *Application::make_gpio_switch()*).

API Reference

GPIOBinaryOutputComponent

class `output::GPIOBinaryOutputComponent`

Simple binary output component for a GPIO pin.

This component allows you to control a GPIO pin as a switch.

Note that with this output component you actually have two ways of inverting the output: either through the *GPIOOutputPin*, or the *BinaryOutput* API. You can use either one of these.

This is only an *output component*, not a *switch*, if what you want is a switch, take a look at `App.make_gpio_switch();`

Inherits from *output::BinaryOutput, Component*

Public Functions

`output::GPIOBinaryOutputComponentGPIOBinaryOutputComponent(GPIOPin *pin)`

Construct the GPIO binary output.

Parameters

- `pin`: The output pin to use for this output, can be integer or *GPIOOutputPin*.

`void output::GPIOBinaryOutputComponentsetup()`

Set pin mode.

```
float output::GPIOBinaryOutputComponentget_setup_priority() const
    Hardware setup priority.
```

```
void output::GPIOBinaryOutputComponentwrite_enabled(bool value)
    Override the BinaryOutput method for writing values to HW.
```

Protected Attributes

```
GPIOPin *output::GPIOBinaryOutputComponentpin_
```

ESP32 LEDC Output

Example Usage

```
// Basic
App.make_ledc_output(33);
// Custom Frequency
App.make_ledc_output(33, 2000.0);
```

See *Application::make_ledc_output()*.

API Reference

LEDCOutputComponent

```
class output::LEDCOutputComponent
    ESP32 LEDC output component.
```

Inherits from *output::FloatOutput, Component*

Public Functions

```
output::LEDCOutputComponentLEDCOutputComponent(uint8_t pin, float frequency = 1000.0f,
                                                uint8_t bit_depth = 12)
    Construct a LEDCOutputComponent. The channel will be set using the next_ledc_channel global variable.
```

```
void output::LEDCOutputComponentset_pin(uint8_t pin)
    Manually set the pin used for this output.
```

```
void output::LEDCOutputComponentset_channel(uint8_t channel)
    Manually set the ledc_channel used for this component.
```

```
void output::LEDCOutputComponentset_bit_depth(uint8_t bit_depth)
    Manually set the bit depth. Defaults to 12.
```

```
void output::LEDCOutputComponentset_frequency(float frequency)
    Manually set frequency the LEDC timer should operate on.
```

Two adjacent LEDC channels will usually receive the same timer and so can only have the same frequency.

Parameters

- **frequency**: The frequency in Hz.

```
void output::LEDCOutputComponentsetup()
    Setup LEDC.

float output::LEDCOutputComponentget_setup_priority() const
    HARDWARE setup priority.

void output::LEDCOutputComponentwrite_state(float adjusted_value)
    Override FloatOutput's write_state.

float output::LEDCOutputComponentget_frequency() const
uint8_t output::LEDCOutputComponentget_bit_depth() const
uint8_t output::LEDCOutputComponentget_channel() const
uint8_t output::LEDCOutputComponentget_pin() const
```

Protected Attributes

```
uint8_t output::LEDCOutputComponentpin_
uint8_t output::LEDCOutputComponentchannel_
uint8_t output::LEDCOutputComponentbit_depth_
float output::LEDCOutputComponentfrequency_
uint8_t output::next_ledc_channel = 0
```

PCA9685 PWM

FloatOutput support for an PCA9695 16-Channel PWM Driver.

Example Usage

```
// Create the PCA9685 Output hub with frequency 500Hz.
auto *pca9685 = App.make_pca9685_component(500.0f);
```

See *Application::make_pca9685_component()*.

API Reference

PCA9685OutputComponent

```
class output::PCA9685OutputComponent
    PCA9685 float output component.

Inherits from Component, I2CDevice
```

Public Functions

```
output::PCA9685OutputComponentPCA9685OutputComponent(I2CComponent *parent, float
frequency, uint8_t mode =
PCA9685_MODE_OUTPUT_ONACK
| PCA9685_MODE_OUTPUT_TOTEM_POLE)
```

Construct the component.

Parameters

- **frequency**: The frequency of the PCA9685.
- **phase_balancer**: How to balance phases.
- **mode**: The output mode. For example, PCA9685_MODE_OUTPUT_ONACK or PCA9685_MODE_OUTPUT_TOTEM_POLE.

```
PCA9685OutputComponent::Channel *output::PCA9685OutputComponentcreate_channel(uint8_t
chan-
nel,
Power-
Supply-
Component
*power_supply
=
nullptr,
float
max_power
=
1.0f)
```

Get a PCA9685 output channel.

Return The new channel output component.

Parameters

- **channel**: The channel number.
- **power_supply**: The power supply that should be set for this channel. Default: nullptr.
- **max_power**: The maximum power output of this channel. Each value will be multiplied by this.

```
void output::PCA9685OutputComponentset_frequency(float frequency)
```

Manually set the frequency of this PCA9685.

```
void output::PCA9685OutputComponentset_mode(uint8_t mode)
```

Manually set the PCA9685 output mode, see constructor for more details.

```
float output::PCA9685OutputComponentget_frequency() const
```

```
uint8_t output::PCA9685OutputComponentget_mode() const
```

```
void output::PCA9685OutputComponentsetup()
```

Setup the PCA9685.

```
float output::PCA9685OutputComponentget_setup_priority() const
    HARDWARE setup_priority.

void output::PCA9685OutputComponentloop()
    Send new values if they were updated.
```

Protected Functions

```
void output::PCA9685OutputComponentset_channel_value(uint8_t channel, uint16_t value)
```

Protected Attributes

```
float output::PCA9685OutputComponentfrequency_
uint8_t output::PCA9685OutputComponentmode_
uint8_t output::PCA9685OutputComponentmin_channel_
uint8_t output::PCA9685OutputComponentmax_channel_
uint16_t output::PCA9685OutputComponentpwm_amounts_[16]
bool output::PCA9685OutputComponentupdate_

class output::PCA9685OutputComponentChannel
    Inherits from output::FloatOutput
```

Public Functions

```
output::PCA9685OutputComponent::ChannelChannel(PCA9685OutputComponent *parent,
                                                uint8_t channel)

void output::PCA9685OutputComponent::Channelwrite_state(float state)
    Write a floating point state to hardware, inversion and max_power is already applied.
```

Protected Attributes

```
PCA9685OutputComponent *output::PCA9685OutputComponent::Channelparent_
uint8_t output::PCA9685OutputComponent::Channelchannel_
```

ESP8266 Software PWM

Software PWM for the ESP8266. Warning: This is a *software* PWM and therefore can have noticeable flickering. Additionally, this software PWM can't output values higher than 80%. That's a known limitation.

Example Usage

```
// Basic
auto *output = App.make_esp8266_pwm_output(D2);
// Create a brightness-only light with it:
App.make_monochromatic_light("Desk Lamp", output);

// Advanced: Setting a custom frequency globally
analogWriteFreq(500);
```

See [Application::make_esp8266_pwm_output\(\)](#).

API Reference

ESP8266PWMOutput

class output::ESP8266PWMOutput

Software PWM output component for ESP8266.

Supported pins are 0-16. By default, this uses a frequency of 1000Hz, and this can only be changed globally for all software PWM pins. To change the PWM frequency, do the following:

```
analogWriteFreq(500); // 500Hz frequency for all pins.
```

Note that this is a software PWM and can have noticeable flickering because of other interrupts on the ESP8266 (like WiFi). Additionally, this PWM output can only be 80% on at max. That is a known limitation, if it's a deal breaker, consider using the ESP32 instead - it has up to 16 integrated hardware PWM channels.

Inherits from [output::FloatOutput](#), [Component](#)

Public Functions

output::ESP8266PWMOutput::ESP8266PWMOutput(const GPIOOutputPin &pin)

Construct the Software PWM output.

Parameters

- **pin**: The pin to be used (inversion of the pin will be ignored, use the direct [set_inverted\(\)](#) method instead).

GPIOOutputPin &output::ESP8266PWMOutput::get_pin()

Get the output pin used by this component.

void output::ESP8266PWMOutput::set_pin(const GPIOOutputPin &pin)

void output::ESP8266PWMOutput::setup()

Initialize pin.

float output::ESP8266PWMOutput::get_setup_priority() const
HARDWARE *setup_priority*.

void output::ESP8266PWMOutput::write_state(float state)

Override [FloatOutput](#)'s write_state for analogWrite.

Protected Attributes

`GPIOOutputPin output::ESP8266PWMOutputpin_`

API Reference

BinaryOutput

`class output::BinaryOutput`

The base class for all binary outputs i.e.

outputs that can only be switched on/off.

This interface class provides one method you need to override in order to create a binary output component yourself: `write_value()`. This method will be called for you by the MQTT `Component` through `enable() / disable()` to indicate that a new value should be written to hardware.

Note that this class also allows the user to invert any state, but you don't need to worry about that because the value will already be inverted (if specified by the user) within `set_state()`. So `write_state` will always receive the correctly inverted state.

Additionally, this class provides high power mode capabilities using `PowerSupplyComponent`. Every time the output is enabled (independent of inversion!), the power supply will automatically be turned on.

Subclassed by `output::FloatOutput`, `output::GPIOBinaryOutputComponent`

Public Functions

`virtual void output::BinaryOutputwrite_enabled(bool enabled) = 0`

Write a binary state to hardware, inversion is already applied.

`void output::BinaryOutputset_inverted(bool inverted)`

Set the inversion state of this binary output.

`void output::BinaryOutputset_power_supply(PowerSupplyComponent *power_supply)`

Use this to connect up a power supply to this output.

Whenever this output is enabled, the power supply will automatically be turned on.

Parameters

- `power_supply`: The `PowerSupplyComponent`, set this to `nullptr` to disable the power supply.

`void output::BinaryOutputenable()`

Enable this binary output.

`void output::BinaryOutputdisable()`

Disable this binary output.

`bool output::BinaryOutputis_inverted() const`

Return whether this binary output is inverted.

`PowerSupplyComponent *output::BinaryOutputget_power_supply() const`

Return the power supply assigned to this binary output.

Protected Attributes

```
bool output::BinaryOutputinverted_ = {false}  
PowerSupplyComponent *output::BinaryOutputpower_supply_ = {nullptr}  
bool output::BinaryOutputhas_requested_high_power_ = {false}
```

FloatOutput

```
class output::FloatOutput
```

Base class for all output components that can output a variable level, like PWM.

Floating Point Outputs always use output values in the range from 0.0 to 1.0 (inclusive), where 0.0 means off and 1.0 means fully on. While using floating point numbers might make computation slower, it makes using maths much easier and (in theory) supports all possible bit depths.

If you want to create a *FloatOutput* yourself, you essentially just have to override *write_state(float)*. That method will be called for you with inversion and max power already applied. It is

This interface is compatible with *BinaryOutput* (and will automatically convert the binary states to floating point states for you). Additionally, this class provides a way for users to set a maximum power output

Inherits from *output::BinaryOutput*

Subclassed by *output::ESP8266PWMOutput*, *output::LEDCOutputComponent*, *output::PCA9685OutputComponent::Channel*

Public Functions

```
virtual void output::FloatOutputwrite_state(float state) = 0
```

Write a floating point state to hardware, inversion and max_power is already applied.

```
void output::FloatOutputset_max_power(float max_power)
```

Set the maximum power output of this component.

All values are multiplied by this float to get the adjusted value.

Parameters

- **max_power:** Automatically clamped from 0 to 1.

```
void output::FloatOutputset_state(float state)
```

This is the method that is called internally by the MQTT component.

```
void output::FloatOutputenable()
```

Override *BinaryOutput*'s *enable()* and *disable()* so that we can convert it to float.

```
void output::FloatOutputdisable()
```

Override *BinaryOutput*'s *enable()* and *disable()* so that we can convert it to float.

```
float output::FloatOutputget_max_power() const
```

Get the maximum power output.

```
void output::FloatOutputwrite_enabled(bool value)
```

Implement BinarySensor's write_enabled; this should never be called.

Protected Attributes

```
float output::FloatOutputmax_power_ = {1.0f}
```

3.3.5 Fan

Fans in esphomelib are implemented like lights. Both the hardware and the MQTT frontend access a combined *FanState* object and use only that to set state and receive state updates.

Example Usage

```
// Basic
auto fan = App.make_fan("Fan");
fan.output->set_binary(App.make_gpio_output(34));
// Speed
auto speed_fan = App.make_fan("Speed Fan");
fan.output->set_speed(App.make_ledc_output(34));
// Oscillation
auto oscillating_fan = App.make_fan("Oscillating Fan");
oscillating_fan.output->set_binary(App.make_gpio_output(34));
oscillating_fan.output->set_oscillation(App.make_gpio_output(35));
```

See *Application::make_fan()*.

API Reference

FanState

```
class fan::FanState
```

This class is shared between the hardware backend and the MQTT frontend to share state.

A fan state has several variables that determine the current state: state (ON/OFF), speed (OFF, LOW, MEDIUM, HIGH), oscillating (ON/OFF) and traits (what features are supported). Both the frontend and the backend can register callbacks whenever a state is changed from the frontend and whenever a state is actually changed and should be pushed to the frontend

Inherits from Nameable

Public Functions

```
fan::FanStateFanState(const std::string &name)
```

Construct the fan state with name.

```
void fan::FanStateadd_on_state_change_callback(std::function<void>
```

> &&update_callbackRegister a callback that will be called each time the state changes.

```
bool fan::FanStateget_state() const
```

Get the current ON/OFF state of this fan.

```
void fan::FanStateset_state(bool state)
```

Set the current ON/OFF state of this fan.

```
bool fan::FanStateis_oscillating() const
    Get the current oscillating state of this fan.

void fan::FanStateset_oscillating(bool oscillating)
    Set the current oscillating state of this fan.

FanSpeed fan::FanStateget_speed() const
    Get the current speed of this fan.

void fan::FanStateset_speed(FanSpeed speed)
    Set the current speed of this fan.

bool fan::FanStateset_speed(const char *speed)
    Get the traits of this fan (i.e. what features it supports).

const FanTraits &fan::FanStateget_traits() const
    Set the traits of this fan (i.e. what features it supports).

void fan::FanStateset_traits(const FanTraits &traits)
    Load a fan state from the preferences into this object.

void fan::FanStatesave_to_preferences()
    Save the fan state from this object into the preferences.

template <typename T>
TurnOnAction<T> *fan::FanStatemake_turn_on_action()

template <typename T>
TurnOffAction<T> *fan::FanStatemake_turn_off_action()

template <typename T>
ToggleAction<T> *fan::FanStatemake_toggle_action()
```

Protected Attributes

```
bool fan::FanStatestate_ = {false}
bool fan::FanStateoscillating_ = {false}
FanSpeed fan::FanStatespeed_ = {FAN_SPEED_HIGH}
FanTraits fan::FanStatetraits_ = {}
CallbackManager<void()> fan::FanStatestate_callback_ = {}
```

FanTraits

```
class fan::FanTraits
This class represents the capabilities/feature set of a fan.
```

Public Functions

```
fan::FanTraitsfan::FanTraits()
Construct an empty FanTraits object. All features will be marked unsupported.
```

fan::*FanTraits***FanTraits**(bool *oscillation*, bool *speed*)
Construct a *FanTraits* object with the provided oscillation and speed support.

bool fan::*FanTraits***supports_oscillation()** const
Return if this fan supports oscillation.

void fan::*FanTraits***set_oscillation(bool oscillation)**
Set whether this fan supports oscillation.

bool fan::*FanTraits***supports_speed()** const
Return if this fan supports speed modes.

void fan::*FanTraits***set_speed(bool speed)**
Set whether this fan supports speed modes.

Protected Attributes

bool fan::*FanTraits***oscillation_**
bool fan::*FanTraits***speed_**

BasicFanComponent

class fan::BasicFanComponent
Simple fan helper that pushes the states to BinaryOutput/FloatOutput devices.

Inherits from *Component*

Public Functions

void fan::*BasicFanComponent***set_binary(output::BinaryOutput *output)**
Create a fan that supports binary state operation (ON/OFF).
Can't be mixed with set_speed.

Parameters

- **output:** The binary output where all binary commands should land.

void fan::*BasicFanComponent***set_speed(output::FloatOutput *output, float off_speed = 0.0, float low_speed = 0.33, float medium_speed = 0.66, float high_speed = 1.0)**
Create a fan that supports speed operation (OFF/LOW/MEDIUM/HIGH SPEED).

Can't be mixed with set_binary.

Parameters

- **output:** The FloatOutput where all speed/state commands should land.
- **off_speed:** The speed that should be sent to the output if the fan is OFF.
- **low_speed:** The speed that should be sent to the output if the fan is in LOW speed mode.

- `medium_speed`: The speed that should be sent to the output if the fan is in MEDIUM speed mode.
- `high_speed`: The speed that should be sent to the output if the fan is in HIGH speed mode.

```
void fan::BasicFanComponentset_oscillation(output::BinaryOutput *oscillating_output)  
    Set an oscillation output for this fan.
```

Parameters

- `oscillating_output`: The binary output where all oscillation commands should land.

```
FanState *fan::BasicFanComponentget_state() const
```

```
void fan::BasicFanComponentset_state(FanState *state)
```

```
void fan::BasicFanComponentsetup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

```
void fan::BasicFanComponentloop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

Protected Attributes

```
FanState *fan::BasicFanComponentstate_ = {nullptr}  
output::BinaryOutput *fan::BasicFanComponentbinary_output_ = {nullptr}  
output::FloatOutput *fan::BasicFanComponentspeed_output_ = {nullptr}  
float fan::BasicFanComponentoff_speed_ = {}  
float fan::BasicFanComponentlow_speed_ = {}  
float fan::BasicFanComponentmedium_speed_ = {}  
float fan::BasicFanComponenthigh_speed_ = {}  
output::BinaryOutput *fan::BasicFanComponentoscillating_output_ = {nullptr}  
bool fan::BasicFanComponentnext_update_ = {true}
```

MQTTFanComponent

```
class fan::MQTTFanComponent  
Inherits from mqtt::MQTTComponent
```

Public Functions

```
fan::MQTTFanComponentMQTTFanComponent(FanState *state)

void fan::MQTTFanComponentset_custom_oscillation_command_topic(const std::string
&topic)
    Set a custom oscillation command topic. Defaults to "<base>/oscillation/command".

void fan::MQTTFanComponentset_custom_oscillation_state_topic(const std::string
&topic)
    Set a custom oscillation state topic. Defaults to "<base>/oscillation/state".

void fan::MQTTFanComponentset_custom_speed_command_topic(const std::string &topic)
    Set a custom speed command topic. Defaults to "<base>/speed/command".

void fan::MQTTFanComponentset_custom_speed_state_topic(const std::string &topic)
    Set a custom speed state topic. Defaults to "<base>/speed/state".

void fan::MQTTFanComponentsend_discovery(JsonBuffer &buffer, JsonObject &root,
                                         mqtt::SendDiscoveryConfig &config)
    Send discovery info to the Home Assistant, override this.

void fan::MQTTFanComponentsetup()
    Setup the fan subscriptions and discovery.

void fan::MQTTFanComponentsend_state()
    Send the full current state to MQTT.

std::string fan::MQTTFanComponentcomponent_type() const
    'fan' component type for discovery.

const std::string fan::MQTTFanComponentget_oscillation_command_topic() const
const std::string fan::MQTTFanComponentget_oscillation_state_topic() const
const std::string fan::MQTTFanComponentget_speed_command_topic() const
const std::string fan::MQTTFanComponentget_speed_state_topic() const

FanState *fan::MQTTFanComponentget_state() const
```

Protected Functions

```
std::string fan::MQTTFanComponentfriendly_name() const
    Get the friendly name of this MQTT component.
```

Protected Attributes

```
FanState *fan::MQTTFanComponentstate_
```

3.3.6 Light

Lights in esphomelib are implemented like fans. Both the hardware and the MQTT frontend access a combined `LightState` object and use only that to set state and receive state updates.

FastLED Light Output

Since version 1.5.0 esphomelib supports many types of addressable LEDs using the FastLED library.

Example Usage

```
// Binary
auto fast_led = App.make_fast_led_light("Fast LED Light");
// 60 NEOPIXEL LEDs on pin GPIO23
fast_led.fast_led->add leds<NEOPIXEL, 23>(60);
```

See [Application::make_fast_led_light\(\)](#).

API Reference

FastLEDLightOutputComponent

```
class light::FastLEDLightOutputComponent
```

This component implements support for many types of addressable LED lights.

To do this, it uses the FastLED library. The API for setting up the different types of lights FastLED supports is intentionally kept as close to FastLEDs defaults as possible. To use FastLED lights with esphomelib, first set up the component using the helper in [Application](#), then add the LEDs using the `add_leds` helper functions.

These `add_leds` helpers can, however, only be called once on a `FastLEDLightOutput`. Also, with this component you cannot pass in the CRGB array and offset values as you would be able to do with FastLED as the component manage the lights itself.

Inherits from `light::LightOutput, Component`

Public Functions

```
void light::FastLEDLightOutputComponentschedule_show()
```

Only for custom effects: Tell this component to write the new color values on the next `loop()` iteration.

```
CRGB *light::FastLEDLightOutputComponentget_leds() const
```

Only for custom effects: Get a pointer to the internal array of CRGB color values.

```
int light::FastLEDLightOutputComponentget_num_leds() const
```

Only for custom effects: Get the number of LEDs managed by this component.

```
CLEDCController *light::FastLEDLightOutputComponentget_controller() const
```

Only for custom effects: Get the internal controller.

```
void light::FastLEDLightOutputComponentset_max_refresh_rate(uint32_t interval_us)
```

Set a maximum refresh rate in μs as some lights do not like being updated too often.

```
void light::FastLEDLightOutputComponentprevent_writing_leds()
```

Only for custom effects: Prevent the `LightState` from writing over all color values in CRGB.

```
void light::FastLEDLightOutputComponentunprevent_writing_leds()
```

Only for custom effects: Stop prevent_writing_leds. Call this when your effect terminates.

```

void light::FastLEDLightOutputComponentset_power_supply(PowerSupplyComponent
                                                       *power_supply)

CLEDController &light::FastLEDLightOutputComponentadd_leds(CLEDController *controller,
                                                          int num_leds)
    Add some LEDS, can only be called once.

template <ESPIChipsets CHIPSET, uint8_t DATA_PIN, uint8_t CLOCK_PIN, EOrder RGB_ORDER, uint8_t t>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <ESPIChipsets CHIPSET, uint8_t DATA_PIN, uint8_t CLOCK_PIN>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <ESPIChipsets CHIPSET, uint8_t DATA_PIN, uint8_t CLOCK_PIN, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <ESPIChipsets CHIPSET>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <ESPIChipsets CHIPSET, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <ESPIChipsets CHIPSET, EOrder RGB_ORDER, uint8_t SPI_DATA_RATE>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <template< uint8_t DATA_PIN, EOrder RGB_ORDER > class CHIPSET, uint8_t DATA_PIN, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <template< uint8_t DATA_PIN, EOrder RGB_ORDER > class CHIPSET, uint8_t DATA_PIN>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <template< EOrder RGB_ORDER > class CHIPSET, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <template< EOrder RGB_ORDER > class CHIPSET>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <OWS2811 CHIPSET, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <OWS2811 CHIPSET>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <SWS2812 CHIPSET, int DATA_PIN, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <ESM CHIPSET>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <EBlockChipsets CHIPSET, int NUM_LANES, EOrder RGB_ORDER>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

template <EBlockChipsets CHIPSET, int NUM_LANES>
CLEDController &light::FastLEDLightOutputComponentadd_leds(int num_leds)

LightTraits light::FastLEDLightOutputComponentget_traits()
    Return the LightTraits of this LightOutput.

void light::FastLEDLightOutputComponentwrite_state(LightState *state)

void light::FastLEDLightOutputComponentsetup()
    Where the component's initialization should happen.

    Analogous to Arduino's setup(). This method is guaranteed to only be called once. Defaults to doing noting.

```

```
void light::FastLEDDLightOutputComponentloop()
```

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

```
float light::FastLEDDLightOutputComponentget_setup_priority() const
```

priority of `setup()`.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

Protected Attributes

```
CLEDController *light::FastLEDDLightOutputComponentcontroller_ = {nullptr}
```

```
CRGB *light::FastLEDDLightOutputComponentleds_ = {nullptr}
```

```
int light::FastLEDDLightOutputComponentnum_leds_ = {0}
```

```
uint32_t light::FastLEDDLightOutputComponentlast_refresh_ = {0}
```

```
optional<uint32_t> light::FastLEDDLightOutputComponentmax_refresh_rate_ = {}
```

```
bool light::FastLEDDLightOutputComponentprevent_writing_leds_ = {false}
```

```
bool light::FastLEDDLightOutputComponentnext_show_ = {true}
```

```
PowerSupplyComponent *light::FastLEDDLightOutputComponentpower_supply_ = {nullptr}
```

```
bool light::FastLEDDLightOutputComponenthas_requested_high_power_ = {false}
```

Example Usage

```
// Binary
App.make_binary_light("Desk Lamp", App.make_gpio_output(15));
// Brightness-only
App.make_monochromatic_light("Kitchen Lights", App.make_ledc_output(16));
// RGB, see output for information how to setup individual channels.
App.make_rgb_light("RGB Lights", red, green, blue);
App.make_rgwb_light("RGWB Lights", red, green, blue, white);
```

See `Application::make_binary_light()`, `Application::make_monochromatic_light()`,
`Application::make_rgb_light()`, `Application::make_rgwb_light()`.

API Reference

LightColorValues

```
class light::LightColorValues
```

This class represents the color state for a light object.

All values in this class are represented using floats in the range from 0.0 (off) to 1.0 (on). Not all values have to be populated though, for example a simple monochromatic light only needs to access the state and brightness attributes.

Please note all float values are automatically clamped.

`state` - Whether the light should be on/off. Represented as a float for transitions. `brightness` - The brightness of the light. `red`, `green`, `blue` - RGB values. `white` - The white value for RGBW lights.

Public Functions

`light::LightColorValues::LightColorValues()`

Construct the `LightColorValues` with all attributes enabled, but state set to 0.0.

`light::LightColorValues::LightColorValues(float state, float brightness, float red, float green, float blue, float white)`

`void light::LightColorValues::load_from_preferences(const std::string &friendly_name)`

Load the color values from the non-volatile storage container preferences into the this object.

See `save_to_preferences()`

Parameters

- `friendly_name`: The friendly name of the component that's calling this.

`void light::LightColorValues::save_to_preferences(const std::string &friendly_name) const`

Store these `LightColorValues` in the preferences object.

See `load_from_preferences()`

Parameters

- `friendly_name`: The friendly name of the component that's calling this.

`void light::LightColorValues::parse_json(const JsonObject &root)`

Parse a color from the provided JsonObject.

See Home Assistant MQTT JSON light.

Parameters

- `root`: The json root object.

`void light::LightColorValues::dump_json(JsonObject &root, const LightTraits &traits) const`

Dump this color into a JsonObject.

Only dumps values if the corresponding traits are marked supported by traits.

Parameters

- `root`: The json root object.
- `traits`: The traits object used for determining whether to include certain attributes.

`void light::LightColorValues::normalize_color(const LightTraits &traits)`

Normalize the color (RGB/W) component.

Divides all color attributes by the maximum attribute, so effectively set at least one attribute to 1. For example: r=0.3, g=0.5, b=0.4 => r=0.6, g=1.0, b=0.8

Parameters

- **traits**: Used for determining which attributes to consider.

```
void light::LightColorValuesas_binary(bool *binary) const
void light::LightColorValuesas_brightness(float *brightness) const
void light::LightColorValuesas_rgb(float *red, float *green, float *blue) const
void light::LightColorValuesas_rgby(float *red, float *green, float *blue, float *white) const
bool light::LightColorValuesoperator==(const LightColorValues &rhs) const
    Compare this LightColorValues to rhs, return true iff all attributes match.

bool light::LightColorValuesoperator!=(const LightColorValues &rhs) const

float light::LightColorValuesget_state() const
void light::LightColorValuesset_state(float state)
float light::LightColorValuesget_brightness() const
void light::LightColorValuesset_brightness(float brightness)
float light::LightColorValuesget_red() const
void light::LightColorValuesset_red(float red)
float light::LightColorValuesget_green() const
void light::LightColorValuesset_green(float green)
float light::LightColorValuesget_blue() const
void light::LightColorValuesset_blue(float blue)
float light::LightColorValuesget_white() const
void light::LightColorValuesset_white(float white)
```

Public Static Functions

LightColorValues light::*LightColorValues***lerp**(**const** *LightColorValues* &*start*, **const** *LightColorValues* &*end*, float *completion*)

Linearly interpolate between the values in start to the values in end.

This function linearly interpolates the color value by just interpolating every attribute independently.

Return The linearly interpolated *LightColorValues*.

Parameters

- **start**: The interpolation start values.
- **end**: The interpolation end values.
- **completion**: The completion value. 0 -> start, 1 -> end.

Protected Attributes

```
float light::LightColorValuesstate_  
    ON / OFF, float for transition.  
  
float light::LightColorValuesbrightness_  
float light::LightColorValuesred_  
float light::LightColorValuesgreen_  
float light::LightColorValuesblue_  
float light::LightColorValueswhite_
```

LightEffect

```
class light::LightEffect  
Base-class for all light effects.
```

In order to create a custom effect, first create a subclass implementing `apply_effect` and optionally initialize which both can use the provided `LightState` to set light colors and/or start transitions. Next, override `get_name()` and return the name of your effect. Lastly, you need to register your effect in `light_effect_entries` using the `LightEffect::Entry` struct.

Subclassed by `light::FastLEDRainbowLightEffect`, `light::NoneLightEffect`, `light::RandomLightEffect`

Public Functions

```
virtual std::string light::LightEffectget_name() const = 0  
Return the name of this effect.
```

```
void light::LightEffectinitialize(LightState *state)  
Initialize this LightEffect. Will be called once after creation.
```

```
void light::LightEffectstop(LightState *state)  
Called when this effect is about to be removed.
```

```
virtual void light::LightEffectapply_effect(LightState *state) = 0  
Apply this effect. Use the provided state for starting transitions, ...
```

```
struct light::LightEffectEntry  
Internal struct for light_effect_entries.
```

Public Members

```
std::string light::LightEffect::Entryname  
The name of your effect.
```

```
LightTraits light::LightEffect::Entryrequirements  
Which traits the light needs to have for this effect.
```

```
std::function<std::unique_ptr<LightEffect>> light::LightEffect::Entry::constructor  
A function creating a your light effect.
```

```
class light::NoneLightEffect
Default effect for all lights. Does nothing.

Inherits from light::LightEffect
```

Public Functions

```
std::string light::NoneLightEffect::get_name() const
    Return the name of this effect.

void light::NoneLightEffect::apply_effect(LightState *state)
    Apply this effect. Use the provided state for starting transitions, ...
```

Public Static Functions

```
std::unique_ptr<LightEffect> light::NoneLightEffect::create()

class light::RandomLightEffect
Random effect. Sets random colors every 10 seconds and slowly transitions between them.

Inherits from light::LightEffect
```

Public Functions

```
light::RandomLightEffect::RandomLightEffect()

std::string light::RandomLightEffect::get_name() const
    Return the name of this effect.

void light::RandomLightEffect::apply_effect(LightState *state)
    Apply this effect. Use the provided state for starting transitions, ...
```

Public Static Functions

```
std::unique_ptr<LightEffect> light::RandomLightEffect::create()
```

Protected Attributes

```
uint32_t light::RandomLightEffect::last_color_change_ = 0;

class light::FastLEDRainbowLightEffect
Rainbow effect for FastLED.

Inherits from light::LightEffect
```

Public Functions

```
light::FastLEDRainbowLightEffect::FastLEDRainbowLightEffect()

std::string light::FastLEDRainbowLightEffect::get_name() const
    Return the name of this effect.
```

```
void light::FastLEDRainbowLightEffectapply_effect(light::LightState *state)
    Apply this effect. Use the provided state for starting transitions, ...
```

```
void light::FastLEDRainbowLightEffectinitialize(light::LightState *state)
    Initialize this LightEffect. Will be called once after creation.
```

```
void light::FastLEDRainbowLightEffectstop(light::LightState *state)
    Called when this effect is about to be removed.
```

Public Static Functions

```
std::unique_ptr<light::LightEffect> light::FastLEDRainbowLightEffectcreate()
    .requirements = LightTraits(true, true, false, true), .constructor = FastLEDRainbowLightEffect::create
}, #endif }
```

LightOutput

```
class light::LightOutput
    Interface to write LightStates to hardware.
```

```
Subclassed      by      light::BinaryLightOutput,      light::FastLEDLightOutputComponent,
light::MonochromaticLightOutput, light::RGBLightOutput, light::RGBWLightOutput
```

Public Functions

```
virtual LightTraits light::LightOutputget_traits() = 0
    Return the LightTraits of this LightOutput.
```

```
virtual void light::LightOutputwrite_state(LightState *state) = 0
```

```
class light::BinaryLightOutput
    Inherits from light::LightOutput
```

Public Functions

```
light::BinaryLightOutputBinaryLightOutput(output::BinaryOutput *output)
```

```
LightTraits light::BinaryLightOutputget_traits()
    Return the LightTraits of this LightOutput.
```

```
void light::BinaryLightOutputwrite_state(LightState *state)
```

Protected Attributes

```
output::BinaryOutput *light::BinaryLightOutputoutput_
```

```
class light::MonochromaticLightOutput
    Inherits from light::LightOutput
```

Public Functions

```
light::MonochromaticLightOutput MonochromaticLightOutput(output::FloatOutput *output)
```

```
LightTraits light::MonochromaticLightOutput get_traits()
```

Return the *LightTraits* of this *LightOutput*.

```
void light::MonochromaticLightOutput write_state(LightState *state)
```

Protected Attributes

```
output::FloatOutput *light::MonochromaticLightOutput output_
```

```
class light::RGBLightOutput
```

Inherits from *light::LightOutput*

Public Functions

```
light::RGBLightOutput RGBLightOutput(output::FloatOutput *red, output::FloatOutput *green,  
                                     output::FloatOutput *blue)
```

```
LightTraits light::RGBLightOutput get_traits()
```

Return the *LightTraits* of this *LightOutput*.

```
void light::RGBLightOutput write_state(LightState *state)
```

Protected Attributes

```
output::FloatOutput *light::RGBLightOutput red_
```

```
output::FloatOutput *light::RGBLightOutput green_
```

```
output::FloatOutput *light::RGBLightOutput blue_
```

```
class light::RGBWLightOutput
```

Inherits from *light::LightOutput*

Public Functions

```
light::RGBWLightOutput RGBWLightOutput(output::FloatOutput *red, output::FloatOutput  
                                         *green, output::FloatOutput *blue, output::  
                                         FloatOutput *white)
```

```
LightTraits light::RGBWLightOutput get_traits()
```

Return the *LightTraits* of this *LightOutput*.

```
void light::RGBWLightOutput write_state(LightState *state)
```

Protected Attributes

```
output::FloatOutput *light::RGBWLightOutputred_
output::FloatOutput *light::RGBWLightOutputgreen_
output::FloatOutput *light::RGBWLightOutputblue_
output::FloatOutput *light::RGBWLightOutputwhite_
```

LightState

class light::**LightState**

This class represents the communication layer between the front-end MQTT layer and the hardware output layer.

Inherits from Nameable, *Component*

Public Functions

light::**LightState**(**LightState**(**const std::string &name**, LightOutput ***output**)

Construct this *LightState* using the provided traits and name.

void light::**LightState****start_effect**(**const std::string &name**)

Start an effect by name.

Uses light_effect_entries in light_effect.h for finding the correct effect.

Parameters

- **name**: The name of the effect, case insensitive.

void light::**LightState****stop_effect**()

Stop the current effect (if one is active).

void light::**LightState****start_transition**(**const LightColorValues &target**, uint32_t **length**)

Start a linear transition to the provided target values for a specific amount of time.

If this light doesn't support transition (see set_traits()), sets the target values immediately.

Parameters

- **target**: The target color.
- **length**: The transition length in ms.

void light::**LightState****start_flash**(**const LightColorValues &target**, uint32_t **length**)

Start a flash for the specified amount of time.

Resets to the values that were active when the flash was started after length ms.

Parameters

- **target**: The target flash color.
- **length**: The flash length in ms.

```
void light::LightStateset_immediately(const LightColorValues &target)
    Set the color values immediately.

void light::LightStateset_immediately_without_sending(const LightColorValues &target)
    Set the color values immediately without sending the new state.

void light::LightStatestart_default_transition(const LightColorValues &target)

void light::LightStatecurrent_values_as_binary(bool *binary)

void light::LightStatecurrent_values_as_brightness(float *brightness)

void light::LightStatecurrent_values_as_rgb(float *red, float *green, float *blue)

void light::LightStatecurrent_values_as_rgbw(float *red, float *green, float *blue, float *white)

LightTraits light::LightStateget_traits()

template <typename T>
ToggleAction<T> *light::LightStatemake_toggle_action()

template <typename T>
TurnOffAction<T> *light::LightStatemake_turn_off_action()

template <typename T>
TurnOnAction<T> *light::LightStatemake_turn_on_action()

void light::LightStatesetup()
    Load state from preferences.

void light::LightStateloop()
    This method will be called repeatedly.

    Analogous to Arduino's loop(). setup() is guaranteed to be called before this. Defaults to doing nothing.

float light::LightStateget_setup_priority() const
    Shortly after HARDWARE.

LightColorValues light::LightStateget_current_valuesLightColorValues light::LightStateget_remote_valuesLightStatesend_values()
    Causes the callback defined by add_send_callback() to trigger.

LightOutput *light::LightStateget_output() const

void light::LightStateset_transformer(std::unique_ptr<LightTransformer> transformer)
    Manually set a transformer, it's recommended to use start_flash and start_transition instead.

const LightColorValues &light::LightStateget_current_values_lazy()
    Lazily get the last current values. Returns the values last returned by get_current_values().

std::string light::LightStateget_effect_name()
    Return the name of the current effect, or if no effect is active "None".
```

```
void light::LightStateadd_new_remote_values_callback(light_send_callback_t
                                                &&send_callback)
```

This lets front-end components subscribe to light change events.

This is different from add_new_current_values_callback in that it only sends events for start and end values. For example, with transitions it will only send a single callback whereas the callback passed in add_new_current_values_callback will be called every *loop()* cycle when a transition is active

Note the callback should get the output values through *get_remote_values()*.

Parameters

- `send_callback`: The callback.

```
bool light::LightStatesupports_effects()
```

Return whether the light has any effects that meet the trait requirements.

```
void light::LightStateparse_json(const JsonObject &root)
```

Parse and apply the provided JSON payload.

```
void light::LightStatedump_json(JsonBuffer &buffer, JsonObject &root)
```

Dump the state of this light as JSON.

```
uint32_t light::LightStateget_default_transition_length() const
```

Defaults to 1 second (1000 ms).

```
void light::LightStateset_default_transition_length(uint32_t default_transition_length)
```

Set the default transition length, i.e. the transition length when no transition is provided.

```
float light::LightStateget_gamma_correct() const
```

```
void light::LightStateset_gamma_correct(float gamma_correct)
```

Set the gamma correction factor.

Protected Attributes

```
uint32_t light::LightStatedefault_transition_length_ = {1000}
```

```
std::unique_ptr<LightEffect> light::LightStateeffect_ = {nullptr}
```

```
std::unique_ptr<LightTransformer> light::LightStatetransformer_ = {nullptr}
```

```
LightColorValues light::LightStatevalues_ = {}
```

```
CallbackManager<void()> light::LightStateremote_values_callback_ = {}
```

```
LightOutput *light::LightStateoutput
```

Store the output to allow effects to have more access.

```
bool light::LightStatenext_write_ = {true}
```

```
float light::LightStategamma_correct_ = {2.8f}
```

Light Traits

```
class light::LightTraits
```

This class is used to represent the capabilities of a light.

Public Functions

```
light::LightTraits::LightTraits()  
light::LightTraits::LightTraits(bool brightness, bool rgb, bool rgb_white_value, bool fast_led =  
    false)  
bool light::LightTraits::supports_traits(const LightTraits &rhs) const  
bool light::LightTraits::has_brightness() const  
bool light::LightTraits::has_rgb() const  
bool light::LightTraits::has_rgb_white_value() const  
bool light::LightTraits::has_fast_led() const  
    Hack to allow FastLED light effects without dynamic_cast.
```

Protected Attributes

```
bool light::LightTraits::brightness_ = {false}  
bool light::LightTraits::rgb_ = {false}  
bool light::LightTraits::rgb_white_value_ = {false}  
bool light::LightTraits::fast_led_ = {false}
```

LightTransformer

```
class light::LightTransformer  
Base-class for all light color transformers, such as transitions or flashes.  
Subclassed by light::LightFlashTransformer, light::LightTransitionTransformer
```

Public Functions

```
light::LightTransformer::LightTransformer(uint32_t start_time, uint32_t length, const LightColorValues &start_values, const LightColorValues &target_values)  
light::LightTransformer::LightTransformer()  
bool light::LightTransformer::is_finished()  
    Whether this transformation is finished.  
virtual bool light::LightTransformer::is_continuous() = 0  
    Whether the output needs to be written in every loop cycle.  
virtual LightColorValues light::LightTransformer::get_values() = 0  
    This will be called to get the current values for output.  
LightColorValues light::LightTransformer::get_remote_values()  
    The values that should be reported to the front-end.  
LightColorValues light::LightTransformer::get_end_values()  
    The values that should be set after this transformation is complete.
```

Protected Functions

```
float light::LightTransformerget_progress()
    Get the completion of this transformer, 0 to 1.

const LightColorValues &light::LightTransformerget_start_values() const
const LightColorValues &light::LightTransformerget_target_values() const
```

Protected Attributes

```
uint32_t light::LightTransformerstart_time_
uint32_t light::LightTransformerlength_
LightColorValues light::LightTransformerstart_values_
LightColorValues light::LightTransformertarget_values_

class light::LightTransitionTransformer
Inherits from light::LightTransformer
```

Public Functions

```
light::LightTransitionTransformerLightTransitionTransformer(uint32_t start_time, uint32_t
length, const LightColorValues &start_values, const
LightColorValues &target_values)
```

LightColorValues light::*LightTransitionTransformer***get_values()**
This will be called to get the current values for output.

bool light::*LightTransitionTransformer***is_continuous()**
Whether the output needs to be written in every loop cycle.

```
class light::LightFlashTransformer
Inherits from light::LightTransformer
```

Public Functions

```
light::LightFlashTransformerLightFlashTransformer(uint32_t start_time, uint32_t length,
const LightColorValues &start_values,
const LightColorValues &target_values)
```

LightColorValues light::*LightFlashTransformer***get_values()**
This will be called to get the current values for output.

LightColorValues light::*LightFlashTransformer***get_end_values()**
The values that should be set after this transformation is complete.

bool light::*LightFlashTransformer***is_continuous()**
Whether the output needs to be written in every loop cycle.

MQTTJSONLightComponent

```
class light::MQTTJSONLightComponent
Inherits from mqtt::MQTTComponent
```

Public Functions

```
light::MQTTJSONLightComponent::MQTTJSONLightComponent(LightState *state)
```

```
LightState *light::MQTTJSONLightComponent::get_state() const
```

```
void light::MQTTJSONLightComponent::setup()
```

Where the component's initialization should happen.

Analogous to Arduino's `setup()`. This method is guaranteed to only be called once. Defaults to doing noting.

```
void light::MQTTJSONLightComponent::send_discovery(JsonBuffer &buffer, JsonObject &root,
                                                mqtt::SendDiscoveryConfig &config)
```

Send discovery info to the Home Assistant, override this.

Protected Functions

```
std::string light::MQTTJSONLightComponent::friendly_name() const
```

Get the friendly name of this MQTT component.

```
std::string light::MQTTJSONLightComponent::component_type() const
```

Override this method to return the component type (e.g. "light", "sensor", ...)

```
void light::MQTTJSONLightComponent::send_light_values()
```

Protected Attributes

```
LightState *light::MQTTJSONLightComponent::state_
```

3.3.7 Switch

The `switch_` namespace contains all switch helpers.

IR Transmitter

Example Usage

```
// at the top of your file:
using namespace switch_::ir_;

// Create the hub
auto *ir = App.make_ir_transmitter(32);
// Create switches
auto *panasonic_on = ir->create_transmitter("Panasonic TV On", SendData::from_panasonic(0x4004, 0x100BCBD).repeat(25));
```

(continues on next page)

(continued from previous page)

```
App.register_switch.panasonic_on);
App.register_switch(ir->create_transmitter("Panasonic TV Volume Up", SendData::from_
˓→panasonic(0x4004, 0x1000405));
```

See `Application::make_ir_transmitter()` and `Application::register_switch()`.

API Reference

IRTransmitterComponent

```
class switch_::IRTransmitterComponent
```

This is an IR Transmitter hub using the ESP32 RMT peripheral.

With it you can send out IR commands from different vendors and have them appear as switches in Home Assistant.

Inherits from `Component`

Public Functions

```
switch_::IRTransmitterComponent IRTransmitterComponent(GPIOPin *pin, uint8_t carrier_duty_percent = 50)
```

Construct the IR Transmitter with the specified pin (can be inverted).

Parameters

- `pin`: The pin for this IR Transmitter.
- `carrier_duty_percent`: The duty percentage (from 0-100) for this transmitter.

```
IRTransmitterComponent::DataTransmitter *switch_::IRTransmitterComponent create_transmitter(const std::string &name, const ir::SendData &send_data)
```

Create a `DataTransmitter` from the specified `SendData` and register it in the transmitter.

Return A `DataTransmitter` which can be used as a `Switch`.

Parameters

- `send_data`: The `SendData`.

```
void switch_::IRTransmitterComponent set_carrier_duty_percent(uint8_t carrier_duty_percent)
```

Set the carrier duty percentage (from 0-100).

```
void switch_::IRTransmitterComponent setup()
```

Setup the RMT peripheral.

```
float switch_::IRTransmitterComponent get_setup_priority() const
```

HARDWARE_LATE setup priority.

```
void switch_::IRTransmitterComponentsend(ir::SendData &send_data)
    Send an IR SendData object on this pin.

void switch_::IRTransmitterComponentset_clock_divider(uint8_t clock_divider)
    Set the clock divisor for RMT.

void switch_::IRTransmitterComponentset_channel(rmt_channel_t channel)
    Set the RMT channel used.
```

Protected Functions

```
void switch_::IRTransmitterComponentrequire_carrier_frequency(uint32_t carrier_frequency)
    Setup the RMT peripheral for the specified carrier frequency, if it's already the used frequency,
    does nothing.

uint16_t switch_::IRTransmitterComponentget_ticks_for_10_us()
    Get the number of ticks (from the clock divider) for 10 µs.

void switch_::IRTransmitterComponentconfigure_rmt()
    Configure the RMT peripheral using the internal information.

void switch_::IRTransmitterComponentcalculate_on_off_time(uint32_t carrier_frequency,
    uint32_t *on_time_period,
    uint32_t *off_time_period)

void switch_::IRTransmitterComponentdelay_microseconds_accurate(uint32_t usec)

void switch_::IRTransmitterComponentmark_(uint32_t on_time, uint32_t off_time, uint32_t usec)

void switch_::IRTransmitterComponentspace_(uint32_t usec)
```

Protected Attributes

```
rmt_channel_t switch_::IRTransmitterComponentchannel_
uint8_t switch_::IRTransmitterComponentclock_divider_ = {DEFAULT_CLOCK_DIVIDER}
uint32_t switch_::IRTransmitterComponentlast_carrier_frequency_ = {DEFAULT_CARRIER_FREQUENCY_}
GPIOPin *switch_::IRTransmitterComponentpin_
uint8_t switch_::IRTransmitterComponentcarrier_duty_percent_

class switch_::IRTransmitterComponentDataTransmitter
    Internal helper class that exposes a SendData as a Switch.
    Inherits from switch_::Switch
```

Public Functions

```
switch_::IRTransmitterComponent::DataTransmitterDataTransmitter(const std::string
&name, const
ir::SendData
&send_data, IR-
TransmitterCompo-
nent *parent)
```

void switch_::IRTransmitterComponent::DataTransmitterturn_on()

Turn this switch on. When creating a switch, you should implement this (inversion will already be applied).

void switch_::IRTransmitterComponent::DataTransmitterturn_off()

Turn this switch off. When creating a switch, you should implement this (inversion will already be applied).

std::string switch_::IRTransmitterComponent::DataTransmittericon()

Override this to set the Home Assistant icon for this switch.

Return “” to disable this feature.

Return The icon of this switch, for example “mdi:fan”.

Protected Attributes

```
ir::SendData switch_::IRTransmitterComponent::DataTransmittersend_data_
IRTransmitterComponent *switch_::IRTransmitterComponent::DataTransmitterparent_
rmt_channel_t switch_::next_rmt_channel = RMT_CHANNEL_0
```

SendData

namespace switch_::ir

Namespace storing constants for vendor IR formats.

struct switch_::irSendData

#include <ir_transmitter_component.h> Struct that stores the raw data for sending out IR codes.

Internally it stores some fields like carrier frequency and a data field, that stores the raw IR codes as an array.

Public Functions

```
std::vector<rmt_item32_t> switch_::ir::SendDataget_rmt_data(uint16_t
ticks_for_10_us)
```

Return the internal data as an RMT interface-compatible vector.

void switch_::ir::SendDatamark(uint16_t duration_us)

Push back a high value for the specified duration to the data.

void switch_::ir::SendDataspace(uint16_t duration_us)

Push back a low value for the specified duration to the data.

```
void switch_::ir::SendDataadd_item(uint16_t high_us, uint16_t low_us)
    Add a high+low value for the specified durations to the data.
```

```
SendData switch_::ir::SendDatarepeat(uint16_t times, uint16_t wait_us = 30000)
    Repeat this SendData with the specified wait time between sends.
```

```
uint32_t switch_::ir::SendDatatotal_length_ms() const
    Calculate the total length required to run this IR code.
```

Public Members

```
uint32_t switch_::ir::SendDatacarrier_frequency
```

The carrier frequency of the IR protocol.

```
std::vector<int16_t> switch_::ir::SendDatadata
    Raw IR data, negative means off, positive means on.
```

```
uint16_t switch_::ir::SendDatarepeat_times = {1}
    How often to perform this data set.
```

```
uint16_t switch_::ir::SendDatarepeat_wait
    How long to wait between repeats.
```

Public Static Functions

```
SendData switch_::ir::SendDatafrom_nec(uint16_t address, uint16_t command)
    Construct a SendData from an NEC address+command.
```

```
SendData switch_::ir::SendDatafrom_lg(uint32_t data, uint8_t nbits = 28)
    Construct a SendData from LG data (with an optional number of bits).
```

```
SendData switch_::ir::SendDatafrom_sony(uint32_t data, uint8_t nbits = 12)
    Construct a SendData from Sony data (with an optional number of bits).
```

```
SendData switch_::ir::SendDatafrom_panasonic(uint16_t address, uint32_t data)
    Construct a SendData from Panasonic address+data.
```

```
SendData switch_::ir::SendDatafrom_raw(std::vector<int> &raw_data, uint32_t carrier_frequency)
    Construct a SendData from a raw data array with specified carrier frequency..
```

```
namespace switch_::irlg
```

Variables

```
const uint32_t switch_::ir::lgCARRIER_FREQUENCY_HZ = 38000
```

```
const uint32_t switch_::ir::lgHEADER_HIGH_US = 8000
```

```
const uint32_t switch_::ir::lgHEADER_LOW_US = 4000
```

```
const uint32_t switch_::ir::lgBIT_HIGH_US = 600
```

```
const uint32_t switch_::ir::lgBIT_ONE_LOW_US = 1600
```

```
const uint32_t switch_::ir::lgBIT_ZERO_LOW_US = 550
```

```
namespace switch_::irnec
```

Variables

```
const uint32_t switch_::ir::necCARRIER_FREQUENCY_HZ = 38000
const uint32_t switch_::ir::necHEADER_HIGH_US = 9000
const uint32_t switch_::ir::necHEADER_LOW_US = 4500
const uint32_t switch_::ir::necBIT_HIGH_US = 560
const uint32_t switch_::ir::necBIT_ONE_LOW_US = 1690
const uint32_t switch_::ir::necBIT_ZERO_LOW_US = 560

namespace switch_::irpanasonic
```

Variables

```
const uint32_t switch_::ir::panasonicCARRIER_FREQUENCY_HZ = 35000
const uint32_t switch_::ir::panasonicHEADER_HIGH_US = 3502
const uint32_t switch_::ir::panasonicHEADER_LOW_US = 1750
const uint32_t switch_::ir::panasonicBIT_HIGH_US = 502
const uint32_t switch_::ir::panasonicBIT_ZERO_LOW_US = 400
const uint32_t switch_::ir::panasonicBIT_ONE_LOW_US = 1244

namespace switch_::irsony
```

Variables

```
const uint32_t switch_::ir::sonyCARRIER_FREQUENCY_HZ = 40000
const uint32_t switch_::ir::sonyHEADER_HIGH_US = 2400
const uint32_t switch_::ir::sonyHEADER_LOW_US = 600
const uint32_t switch_::ir::sonyBIT_ONE_HIGH_US = 1200
const uint32_t switch_::ir::sonyBIT_ZERO_HIGH_US = 600
const uint32_t switch_::ir::sonyBIT_LOW_US = 600
```

Restart Switch

This platform allows you to restart your ESP8266/ESP32 with a simple MQTT message.

Example Usage

```
App.make_restart_switch("Livingroom Restart");
```

See *Application::make_restart_switch()*.

API Reference

```
class switch_::RestartSwitch
```

A simple switch that restarts the device when triggered.

Inherits from *switch_::Switch*

Public Functions

```
switch_::RestartSwitch::RestartSwitch(const std::string &name)
```

```
void switch_::RestartSwitch::turn_on()
```

Turn this switch on. When creating a switch, you should implement this (inversion will already be applied).

```
void switch_::RestartSwitch::turn_off()
```

Turn this switch off. When creating a switch, you should implement this (inversion will already be applied).

```
std::string switch_::RestartSwitch::icon()
```

Override this to set the Home Assistant icon for this switch.

Return “” to disable this feature.

Return The icon of this switch, for example “mdi:fan”.

Shutdown Switch

This platform allows you to put your ESP8266/ESP32 to sleep until it is manually restarted by either pressing the reset button or toggling the power supply.

It is especially useful if you’re using esphomelib with battery cells that must not be discharged too much.

Example Usage

```
App.make_shutdown_switch("Livingroom Shutdown");
```

See *Application::make_shutdown_switch()*.

API Reference

```
class switch_::ShutdownSwitch
```

A simple switch that will put the node into deep sleep indefinitely.

Inherits from *switch_::Switch*

Public Functions

```
switch_::ShutdownSwitch::ShutdownSwitch(const std::string &name)
```

```
void switch_::ShutdownSwitchturn_on()
```

Turn this switch on. When creating a switch, you should implement this (inversion will already be applied).

```
void switch_::ShutdownSwitchturn_off()
```

Turn this switch off. When creating a switch, you should implement this (inversion will already be applied).

```
std::string switch_::ShutdownSwitchicon()
```

Override this to set the Home Assistant icon for this switch.

Return “” to disable this feature.

Return The icon of this switch, for example “mdi:fan”.

Template Switch

See [*Application*::make_template_switch\(\)](#).

API Reference

```
class switch_::TemplateSwitch
Inherits from switch_::Switch
```

Public Functions

```
switch_::TemplateSwitchTemplateSwitch(const std::string &name)
```

```
void switch_::TemplateSwitchset_state_lambda(std::function<optional<bool>>
> &&f
```

```
void switch_::TemplateSwitchadd_turn_on_actions(const std::vector<Action<NoArg> * >
&actions)
```

```
void switch_::TemplateSwitchadd_turn_off_actions(const std::vector<Action<NoArg> * >
&actions)
```

```
void switch_::TemplateSwitchset_optimistic(bool optimistic)
```

```
void switch_::TemplateSwitchloop()
```

This method will be called repeatedly.

Analogous to Arduino’s *loop()*. *setup()* is guaranteed to be called before this. Defaults to doing nothing.

Protected Functions

```
bool switch_::TemplateSwitchoptimistic()
```

Return whether this switch is optimistic - i.e.

if both the ON/OFF actions should be displayed in Home Assistant because the real state is unknown.

Defaults to false.

```
void switch_::TemplateSwitchturn_on()
```

Turn this switch on. When creating a switch, you should implement this (inversion will already be applied).

```
void switch_::TemplateSwitchturn_off()
```

Turn this switch off. When creating a switch, you should implement this (inversion will already be applied).

Protected Attributes

```
optional<std::function<optional<bool>>> > switch_::TemplateSwitch::f_
bool switch_::TemplateSwitchoptimistic_ = {false}
ActionList<NoArg> switch_::TemplateSwitchturn_on_action_
ActionList<NoArg> switch_::TemplateSwitchturn_off_action_
```

API Reference

Switch

```
class switch_::Switch
```

Base class for all switches.

A switch is basically just a combination of a binary sensor (for reporting switch values) and a write_state method that writes a state to the hardware.

Inherits from *binary_sensor::BinarySensor, Component*

Subclassed by *switch_::IRTransmitterComponent::DataTransmitter, switch_::RestartSwitch, switch_::ShutdownSwitch, switch_::SimpleSwitch, switch_::TemplateSwitch*

Public Functions

```
switch_::SwitchSwitch(const std::string &name)
```

```
float switch_::Switchget_setup_priority() const
priority of setup().
```

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

```
void switch_::Switchsetup_()
```

```
void switch_::Switchpublish_state(bool state)
Publish a new state.
```

Inverted input is handled by this method and sub-classes don't need to worry about inverting themselves.

Parameters

- **state:** The new state.

```
void switch_::Switchwrite_state(bool state)
    This method is called by the front-end components.
```

```
std::string switch_::Switchicon()
    Override this to set the Home Assistant icon for this switch.
    Return "" to disable this feature.
```

Return The icon of this switch, for example “mdi:fan”.

```
void switch_::Switchset_icon(const std::string &icon)
    Set the icon for this switch. “” for no icon.

std::string switch_::Switchget_icon()
    Get the icon for this switch. Using icon() if not manually set.
```

```
template <typename T>
ToggleAction<T> *switch_::Switchmake_toggle_action()

template <typename T>
TurnOffAction<T> *switch_::Switchmake_turn_off_action()

template <typename T>
TurnOnAction<T> *switch_::Switchmake_turn_on_action()
```

```
bool switch_::Switchoptimistic()
    Return whether this switch is optimistic - i.e.
    if both the ON/OFF actions should be displayed in Home Assistant because the real state is
    unknown.

    Defaults to false.
```

Protected Functions

```
virtual void switch_::Switchturn_on() = 0
    Turn this switch on. When creating a switch, you should implement this (inversion will already
    be applied).
```

```
virtual void switch_::Switchturn_off() = 0
    Turn this switch off. When creating a switch, you should implement this (inversion will already
    be applied).
```

Protected Attributes

```
optional<std::string> switch_::Switchicon_ = {}
    The icon shown here. Not set means use default from switch. Empty means no icon.
```

SimpleSwitch

```
class switch_::SimpleSwitch
    A simple switch that exposes a binary output as a switch.
```

Inherits from *switch_::Switch*

Public Functions

```
switch_::SimpleSwitchSimpleSwitch(const std::string &name, output::BinaryOutput *output)  
Construct this SimpleSwitch with the provided BinaryOutput.
```

Protected Functions

```
void switch_::SimpleSwitchturn_on()  
Turn this SimpleSwitch on.
```

```
void switch_::SimpleSwitchturn_off()  
Turn this SimpleSwitch off.
```

Protected Attributes

```
output::BinaryOutput *switch_::SimpleSwitchoutput_
```

MQTTSwitchComponent

```
class switch_::MQTTSwitchComponent  
MQTT representation of switches.
```

Overrides MQTTBinarySensorComponent with a callback that can write values to hardware.

Inherits from *binary_sensor*::MQTTBinarySensorComponent

Public Functions

```
switch_::MQTTSwitchComponentMQTTSwitchComponent(switch_::Switch *switch_)
```

```
void switch_::MQTTSwitchComponentsetup()  
Send discovery.
```

```
void switch_::MQTTSwitchComponentsend_discovery(JsonBuffer &buffer, JsonObject &obj,  
mqtt::SendDiscoveryConfig &config)  
Send Home Assistant discovery info.
```

Protected Functions

```
std::string switch_::MQTTSwitchComponentcomponent_type() const  
“switch” component type.
```

```
void switch_::MQTTSwitchComponentturn_on()  
Helper method to turn the switch on.
```

```
void switch_::MQTTSwitchComponentturn_off()  
Helper method to turn the switch off.
```

Protected Attributes

`Switch *switch_`::MQTTSwitchComponents`switch_`

3.3.8 Cover

Supported Covers

Template Cover

See `Application::make_template_cover()`.

API Reference

`class cover::TemplateCover`

Inherits from `cover::Cover`, `Component`

Public Functions

`cover::TemplateCover::TemplateCover(const std::string &name)`

`void cover::TemplateCover::set_state_lambda(std::function<optional<CoverState>> > &&f`

`void cover::TemplateCover::add_open_actions(const std::vector<Action<NoArg> *> &actions)`

`void cover::TemplateCover::add_close_actions(const std::vector<Action<NoArg> *> &actions)`

`void cover::TemplateCover::add_stop_actions(const std::vector<Action<NoArg> *> &actions)`

`void cover::TemplateCover::set_optimistic(bool optimistic)`

`void cover::TemplateCover::loop()`

This method will be called repeatedly.

Analogous to Arduino's `loop()`. `setup()` is guaranteed to be called before this. Defaults to doing nothing.

`void cover::TemplateCover::open()`

`void cover::TemplateCover::close()`

`void cover::TemplateCover::stop()`

Protected Functions

`bool cover::TemplateCover::optimistic()`

Return whether this cover is optimistic - i.e.

if both the OPEN/CLOSE actions should be displayed in Home Assistant because the real state is unknown.

Defaults to false.

Protected Attributes

```
optional<std::function<optional<CoverState>>> > cover::TemplateCover::f_
bool cover::TemplateCoveroptimistic_ = {false}
ActionList<NoArg> cover::TemplateCoveropen_action_
ActionList<NoArg> cover::TemplateCoverclose_action_
ActionList<NoArg> cover::TemplateCoverstop_action_
```

Example Usage

See *Application::register_cover()*.

API Reference

Cover

```
class cover::Cover
Inherits from Nameable
Subclassed by cover::TemplateCover
```

Public Functions

```
cover::CoverCover(const std::string &name)
virtual void cover::Coveropen() = 0
virtual void cover::Coverclose() = 0
virtual void cover::Coverstop() = 0
void cover::Coveradd_on_publish_state_callback(std::function<void> CoverState
> &&f
void cover::Coverpublish_state(CoverState state)
template <typename T>
OpenAction<T> *cover::Covermake_open_action()
template <typename T>
CloseAction<T> *cover::Covermake_close_action()
template <typename T>
StopAction<T> *cover::Covermake_stop_action()

bool cover::Coveroptimistic()
Return whether this cover is optimistic - i.e.
if both the OPEN/CLOSE actions should be displayed in Home Assistant because the real state
is unknown.
Defaults to false.
```

Public Members

```
CoverState cover::Coverstate = {COVER_MAX}
```

Protected Attributes

```
CallbackManager<void(CoverState)> cover::Coverstate_callback_ = {}
```

```
enum cover::CoverState  
Values:  
coverCOVER_OPEN = 0  
coverCOVER_CLOSED  
coverCOVER_MAX  
template <typename T>  
class cover::OpenAction  
Inherits from Action< T >
```

Public Functions

```
cover::OpenActionOpenAction(Cover *cover)
```

```
void cover::OpenActionplay(T x)
```

Protected Attributes

```
Cover *cover::OpenActioncover_  
template <typename T>  
class cover::CloseAction  
Inherits from Action< T >
```

Public Functions

```
cover::CloseActionCloseAction(Cover *cover)
```

```
void cover::CloseActionplay(T x)
```

Protected Attributes

```
Cover *cover::CloseActioncover_  
template <typename T>  
class cover::StopAction  
Inherits from Action< T >
```

Public Functions

```
cover::StopActionStopAction(Cover *cover)  
void cover::StopActionplay(T x)
```

Protected Attributes

Cover *cover::*StopAction***cover_**

MQTTCoverComponent

```
class cover::MQTTCoverComponent  
Inherits from mqtt::MQTTComponent
```

Public Functions

```
cover::MQTTCoverComponentMQTTCoverComponent(Cover *cover)  
void cover::MQTTCoverComponentsetup()  
Where the component's initialization should happen.  
Analogous to Arduino's setup(). This method is guaranteed to only be called once. Defaults to doing noting.  
void cover::MQTTCoverComponentsend_discovery(JsonBuffer &buffer, JsonObject &root,  
                                              mqtt::SendDiscoveryConfig &config)  
Send discovery info to the Home Assistant, override this.
```

Protected Functions

```
std::string cover::MQTTCoverComponentcomponent_type() const  
Override this method to return the component type (e.g. "light", "sensor", ...)  
std::string cover::MQTTCoverComponentfriendly_name() const  
Get the friendly name of this MQTT component.
```

Protected Attributes

Cover *cover::*MQTTCoverComponent***cover_**

3.3.9 Miscellaneous Components

PCF8574 I/O Expander

The PCF8574 component allows you to use PCF8574 and PCF8575 I/O port expanders with many of esphomelib's components. With most components, you are able to specify GPIOOutputPin or GPIOInputPin for internal pins. The PCF8574 component subclasses those types.

Example Usage

```
auto *pcf8574 = App.make_pcf8574_component(0x21);
App.make_gpio_binary_sensor("PCF pin 0 sensor", pcf8574->make_input_pin(0, PCF8574_INPUT));
App.make_gpio_binary_sensor("PCF pin 0 sensor", 0);
App.make_gpio_switch("PCF pin 1 switch", pcf8574->make_output_pin(1));
auto *out = App.make_gpio_output(pcf8574->make_output_pin(2));
App.make_binary_light("PCF pin 2 light", out);
```

See [Application::make_pcf8574_component\(\)](#).

API Reference

enum PCF8574GPIOMode
Modes for PCF8574 pins.

Values:

`PCF8574_INPUT` = INPUT

`PCF8574_INPUT_PULLUP` = INPUT_PULLUP

`PCF8574_OUTPUT` = OUTPUT

class io::PCF8574Component
Inherits from [Component](#), [I2CDevice](#)

Public Functions

`io::PCF8574Component::PCF8574Component(I2CComponent *parent, uint8_t address, bool pcf8575 = false)`

`PCF8574GPIOInputPin io::PCF8574Component::make_input_pin(uint8_t pin, uint8_t mode = PCF8574_INPUT, bool inverted = false)`

Make a [GPIOPin](#) that can be used in other components.

Note that in some cases this component might not work with incompatible other integrations because for performance reasons the values are only sent once every loop cycle in a batch. For example, OneWire sensors are not supported.

Return An PCF8574GPIOPin instance.

Parameters

- `pin`: The pin number to use. 0-7 for PCF8574, 0-15 for PCF8575.
- `mode`: The pin mode to use. Only supported ones are PCF8574_INPUT, PCF8574_INPUT_PULLUP.
- `inverted`: If the pin should invert all incoming and outgoing values.

`PCF8574GPIOOutputPin io::PCF8574Component::make_output_pin(uint8_t pin, bool inverted = false)`

Make a [GPIOPin](#) that can be used in other components.

Note that in some cases this component might not work with incompatible other integrations because for performance reasons the values are only sent once every loop cycle in a batch. For example, OneWire sensors are not supported.

Return An PCF8574GPIOPin instance.

Parameters

- **pin:** The pin number to use. 0-7 for PCF8574, 0-15 for PCF8575.
- **inverted:** If the pin should invert all incoming and outgoing values.

```
void io::PCF8574Componentsetup()
```

Check i2c availability and setup masks.

```
bool io::PCF8574Componentdigital_read_(uint8_t pin)
```

Helper function to read the value of a pin. Doesn't do any I/O.

```
void io::PCF8574Componentdigital_write_(uint8_t pin, bool value)
```

Helper function to write the value of a pin. Doesn't do any I/O.

```
void io::PCF8574Componentpin_mode_(uint8_t pin, uint8_t mode)
```

Helper function to set the pin mode of a pin. Doesn't do any I/O.

Protected Functions

```
bool io::PCF8574Componentread_gpio_()
```

```
bool io::PCF8574Componentwrite_gpio_()
```

Protected Attributes

```
uint16_t io::PCF8574Componentddr_mask_ = {0x00}
```

```
uint16_t io::PCF8574Componentinput_mask_ = {0x00}
```

```
uint16_t io::PCF8574Componentport_mask_ = {0x00}
```

```
bool io::PCF8574Componentpcf8575_
```

TRUE->16-channel PCF8575, FALSE->8-channel PCF8574.

```
class io::PCF8574GPIOInputPin
```

Helper class to expose a PCF8574 pin as an internal input GPIO pin.

Inherits from *GPIOInputPin*

Public Functions

```
io::PCF8574GPIOInputPinPCF8574GPIOInputPin(PCF8574Component *parent, uint8_t pin,
                                              uint8_t mode, bool inverted = false)
```

```
GPIOPin *io::PCF8574GPIOInputPincopy() const
```

```
void io::PCF8574GPIOInputPinsetup()
```

Setup the pin mode.

```
void io::PCF8574GPIOInputPinpin_mode(uint8_t mode)
    Set the pin mode.

bool io::PCF8574GPIOInputPindigital_read()
    Read the binary value from this pin using digitalRead (and inverts automatically).

void io::PCF8574GPIOInputPindigital_write(bool value)
    Write the binary value to this pin using digitalWrite (and inverts automatically).
```

Protected Attributes

```
PCF8574Component *io::PCF8574GPIOInputPinparent_
class io::PCF8574GPIOOutputPin
    Helper class to expose a PCF8574 pin as an internal output GPIO pin.

Inherits from GPIOOutputPin
```

Public Functions

```
io::PCF8574GPIOOutputPinPCF8574GPIOOutputPin(PCF8574Component *parent, uint8_t pin,
                                                uint8_t mode, bool inverted = false)

GPIOPin *io::PCF8574GPIOOutputPincopy() const
void io::PCF8574GPIOOutputPinsetup()
    Setup the pin mode.

void io::PCF8574GPIOOutputPinpin_mode(uint8_t mode)
    Set the pin mode.

bool io::PCF8574GPIOOutputPindigital_read()
    Read the binary value from this pin using digitalRead (and inverts automatically).

void io::PCF8574GPIOOutputPindigital_write(bool value)
    Write the binary value to this pin using digitalWrite (and inverts automatically).
```

Protected Attributes

```
PCF8574Component *io::PCF8574GPIOOutputPinparent_
```

ESP32 Bluetooth Low Energy Tracker

Example Usage

```
auto *tracker = App.make_esp32_ble_tracker();
// MAC address AC:37:43:77:5F:4C
App.register_binary_sensor(tracker->make_device("ESP32 Bluetooth Beacon", {
    0xAC, 0x37, 0x43, 0x77, 0x5F, 0x4C
}));
```

See *Application::make_esp32_ble_tracker()*.

API Reference

`enum PCF8574GPIOMode`

Modes for PCF8574 pins.

Values:

`PCF8574_INPUT = INPUT`

`PCF8574_INPUT_PULLUP = INPUT_PULLUP`

`PCF8574_OUTPUT = OUTPUT`

`class ESP32BLETracker`

The `ESP32BLETracker` class is a hub for all ESP32 Bluetooth Low Energy devices.

This implementation is currently only quite basic and only supports using scan to see if BLE beacons can be found. In the future, this class will support exposing the RSSI service data and some GATTG standardized information like temperature.

The implementation uses a lightweight version of the amazing ESP32 BLE Arduino library by Neil Kolban. This was done because the ble library was quite huge and esphomelib would only ever use a small part of it anyway. Still though, when this component is used, the required flash size increases by a lot (about 500kB). As the BLE stack uses humongous amounts of stack size, this component creates a separate FreeRTOS task to handle all BLE stuff with its own stack.

Also, currently only BLE beacons seem to be picked up by this component. In the future I will try to also make this work with standard BLE devices (like smart-watches, phones, ...), but that will require a bit of reading up on the (very complicated) Bluetooth standard and might take a while.

Inherits from `Component`

Public Functions

`ESP32BLEDevice *ESP32BLETracker::make_device(const std::string &name, std::array<uint8_t, 6> address)`

Create a simple binary sensor for a MAC address.

Each time a BLE scan yields the MAC address in the address parameter, the binary sensor will immediately go to true. When a whole scan interval does not discover a device with this MAC address, the binary sensor will be set to false.

The address parameter accepts a MAC address as an array of length 6 with each MAC address part in an unsigned int. To set it up, do something like this:

```
// MAC address AC:37:43:77:5F:4C
App.register_binary_sensor(tracker->make_device("ESP32 BLE Device", {
    0xAC, 0x37, 0x43, 0x77, 0x5F, 0x4C
}));
```

You can get this MAC address by navigating to the bluetooth screen of your device and looking for an advanced settings screen, you can usually find the MAC address there. Alternatively, you can set the global debug level to DEBUG and observe the logs. Then you will find messages like this:

```
Found device AC:37:43:77:5F:4C RSSI=-80
Address Type: PUBLIC
Name: 'Google Home Mini'
```

If the Address Type shown in the logs is RANDOM, you unfortunately can't track that device at the moment as the device constantly changes its MAC address as a "security" feature.

See [set_scan_interval](#)

Return

Parameters

- **name:** The name of the binary sensor to create.
- **address:** The MAC address to match.

`void ESP32BLETracker.set_scan_interval(uint32_t scan_interval)`

Set the number of seconds (!) that a single BLE scan should take.

This parameter is useful when adjusting how long it should take for a bluetooth device to be marked as disconnected.

Parameters

- **scan_interval:** The interval in seconds to reset the scan.

`void ESP32BLETracker.setup()`

Setup the FreeRTOS task and the Bluetooth stack.

Protected Functions

`void ESP32BLETracker.start_scan(bool first)`

Start a single scan by setting up the parameters and doing some esp-idf calls.

`void ESP32BLETracker.gap_scan_result(const esp_ble_gap_cb_param_t::ble_scan_result_evt_param ¶m)`

Called when a `ESP_GAP_BLE_SCAN_RESULT_EVT` event is received.

`void ESP32BLETracker.gap_scan_set_param_complete(const esp_ble_gap_cb_param_t::ble_scan_param_cmpl ¶m)`

Called when a `ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT` event is received.

`void ESP32BLETracker.gap_scan_start_complete(const esp_ble_gap_cb_param_t::ble_scan_start_cmpl_evt_p ¶m)`

Called when a `ESP_GAP_BLE_SCAN_START_COMPLETE_EVT` event is received.

Protected Attributes

`std::vector<ESP32BLEDevice*> ESP32BLETracker.devices_`

An array of registered devices to track.

`std::vector<uint64_t> ESP32BLETracker.discovered_`

An array of MAC addresses discovered during this scan. Used to mark registered devices as undiscovered.

`esp_ble_scan_params_t ESP32BLETracker.scan_params_`

A structure holding the ESP BLE scan parameters.

`uint32_t ESP32BLETracker.scan_interval_ = {300}`

The interval in seconds to perform scans.

Protected Static Functions

```
void ESP32BLETrackerble_core_task(void *params)
    The FreeRTOS task managing the bluetooth interface.

void ESP32BLETrackergap_event_handler(esp_gap_ble_cb_event_t           event,
                                         esp_ble_gap_cb_param_t *param)
    Callback that will handle all GAP events and redistribute them to other callbacks.

class ESP32BLEDevice
    Simple helper class to expose an BLE device as a binary sensor.

    Inherits from binary_sensor::BinarySensor
```

Public Functions

```
ESP32BLEDeviceESP32BLEDevice(const std::string &name, uint64_t address)
```

Protected Functions

```
std::string ESP32BLEDevicedevice_class()
    Get the default device class for this sensor, or empty string for no default.
```

Protected Attributes

```
friend ESP32BLEDevice::ESP32BLETracker
uint64_t ESP32BLEDeviceaddress_
ESP32BLETracker *global_esp32_ble_tracker
SemaphoreHandle_t semaphore_scan_end
```

Debug Component

Example Usage

```
App.make_debug_component();
```

See *Application::make_debug_component()*.

API Reference

```
class DebugComponent
    The debug component prints out debug information like free heap size on startup.

    Inherits from Component
```

Public Functions

void *DebugComponent***setup()**

Where the component's initialization should happen.

Analogous to Arduino's *setup()*. This method is guaranteed to only be called once. Defaults to doing nothing.

void *DebugComponent***loop()**

This method will be called repeatedly.

Analogous to Arduino's *loop()*. *setup()* is guaranteed to be called before this. Defaults to doing nothing.

float *DebugComponent***get_setup_priority() const**

priority of *setup()*.

higher -> executed earlier

Defaults to 0.

Return The setup priority of this component

Protected Attributes

uint32_t *DebugComponent***free_heap_ = {}**

- genindex
- search

INDEX

Symbols

__anonymous0 (C++ type), 222

A

Action (C++ class), 245

Action::next_ (C++ member), 246

Action::play (C++ function), 245

Action::play_next (C++ function), 245

ActionList (C++ class), 245, 246

ActionList::actions_begin_ (C++ member), 245, 247

ActionList::actions_end_ (C++ member), 245, 247

ActionList::add_action (C++ function), 245, 246

ActionList::add_actions (C++ function), 245, 246

ActionList::play (C++ function), 245, 246

add_safe_shutdown_hook (C++ function), 218

add_shutdown_hook (C++ function), 218

AndCondition (C++ class), 242

AndCondition::AndCondition (C++ function), 243

AndCondition::check (C++ function), 243

AndCondition::conditions_ (C++ member), 243

App (C++ member), 198

Application (C++ class), 180

Application::application_state_ (C++ member), 193

Application::components_ (C++ member), 193

Application::controllers_ (C++ member), 193

Application::get_mqtt_client (C++ function), 193

Application::get_name (C++ function), 193

Application::get_wifi (C++ function), 193

Application::i2c_ (C++ member), 193

Application::init_i2c (C++ function), 181

Application::init_log (C++ function), 180

Application::init_mqtt (C++ function), 181

Application::init_ota (C++ function), 181

Application::init_web_server (C++ function), 182

Application::init_wifi (C++ function), 180, 181

Application::loop (C++ function), 193

Application::make_adc_sensor (C++ function), 184

Application::make_ads1115_component (C++ function), 184

Application::make_automation (C++ function), 182

Application::make_bh1750_sensor (C++ function), 187

Application::make_binary_light (C++ function), 190

Application::make_bme280_sensor (C++ function), 187

Application::make_bme680_sensor (C++ function), 188

Application::make_bmp085_sensor (C++ function), 184

Application::make_dallas_component (C++ function), 183

Application::make_debug_component (C++ function), 192

Application::make_deep_sleep_component (C++ function), 192

Application::make_dht12_sensor (C++ function), 188

Application::make_dht_sensor (C++ function), 183

Application::make_esp32_ble_tracker (C++ function), 182

Application::make_esp32_hall_sensor (C++ function), 189

Application::make_esp32_touch_component (C++ function), 183

Application::make_esp8266_pwm_output (C++ function), 190

Application::make_fan (C++ function), 192

Application::make_fast_led_light (C++ function), 191

Application::make_gpio_binary_sensor (C++ function), 182

Application::make_gpio_output (C++ function), 190

Application::make_gpio_switch (C++ function), 191

Application::make(hdc1080_sensor (C++ function), 185

Application::make_htu21d_sensor (C++ function), 185

Application::make_ir_transmitter (C++ function), 191

Application::make_ledc_output (C++ function), 189
Application::make_light_for_light_output (C++ function), 190
Application::make_max6675_sensor (C++ function), 189
Application::make_monochromatic_light (C++ function), 190
Application::make_mqtt_message_trigger (C++ function), 182
Application::make_pca9685_component (C++ function), 189
Application::make_pcf8574_component (C++ function), 192
Application::make_power_supply (C++ function), 189
Application::make_pulse_counter_sensor (C++ function), 184
Application::make_restart_switch (C++ function), 192
Application::make_rgb_light (C++ function), 190
Application::make_rgbbw_light (C++ function), 191
Application::make_rotary_encoder_sensor (C++ function), 188
Application::make_sht3xd_sensor (C++ function), 188
Application::make_shutdown_switch (C++ function), 192
Application::make_shutdown_trigger (C++ function), 182
Application::make_simple_switch (C++ function), 192
Application::make_startup_trigger (C++ function), 182
Application::make_status_binary_sensor (C++ function), 182
Application::make_template_binary_sensor (C++ function), 183
Application::make_template_cover (C++ function), 192
Application::make_template_sensor (C++ function), 189
Application::make_template_switch (C++ function), 192
Application::make tsl2561_sensor (C++ function), 186
Application::make_ultrasonic_sensor (C++ function), 186
Application::MakeADCsensor (C++ class), 193
Application::MakeADCsensor::adc (C++ member), 193
Application::MakeADCsensor::mqtt (C++ member), 193
Application::MakeBH1750Sensor (C++ class), 193
Application::MakeBH1750Sensor::bh1750 (C++ member), 193
Application::MakeBH1750Sensor::mqtt (C++ member), 193
Application::MakeBME280Sensor (C++ class), 193
Application::MakeBME280Sensor::bme280 (C++ member), 194
Application::MakeBME280Sensor::mqtt_humidity (C++ member), 194
Application::MakeBME280Sensor::mqtt_pressure (C++ member), 194
Application::MakeBME280Sensor::mqtt_temperature (C++ member), 194
Application::MakeBME680Sensor (C++ class), 194
Application::MakeBME680Sensor::bme680 (C++ member), 194
Application::MakeBME680Sensor::mqtt_gas_resistance (C++ member), 194
Application::MakeBME680Sensor::mqtt_humidity (C++ member), 194
Application::MakeBME680Sensor::mqtt_pressure (C++ member), 194
Application::MakeBME680Sensor::mqtt_temperature (C++ member), 194
Application::MakeBMP085Sensor (C++ class), 194
Application::MakeBMP085Sensor::bmp (C++ member), 194
Application::MakeBMP085Sensor::mqtt_pressure (C++ member), 194
Application::MakeBMP085Sensor::mqtt_temperature (C++ member), 194
Application::MakeDHT12Sensor (C++ class), 194
Application::MakeDHT12Sensor::dht12 (C++ member), 194
Application::MakeDHT12Sensor::mqtt_humidity (C++ member), 194
Application::MakeDHT12Sensor::mqtt_temperature (C++ member), 194
Application::MakeDHTSensor (C++ class), 194
Application::MakeDHTSensor::dht (C++ member), 194
Application::MakeDHTSensor::mqtt_humidity (C++ member), 194
Application::MakeDHTSensor::mqtt_temperature (C++ member), 194
Application::MakeESP32HallSensor (C++ class), 194
Application::MakeESP32HallSensor::hall (C++ member), 195
Application::MakeESP32HallSensor::mqtt (C++ member), 195
Application::MakeFan (C++ class), 195
Application::MakeFan::mqtt (C++ member), 195
Application::MakeFan::output (C++ member), 195

Application::MakeFan::state (C++ member), 195
 Application::MakeFastLEDDLight (C++ class), 195
 Application::MakeFastLEDDLight::fast_led (C++ member), 195
 Application::MakeFastLEDDLight::mqtt (C++ member), 195
 Application::MakeFastLEDDLight::state (C++ member), 195
 Application::MakeGPIOBinarySensor (C++ class), 195
 Application::MakeGPIOBinarySensor::gpio (C++ member), 195
 Application::MakeGPIOBinarySensor::mqtt (C++ member), 195
 Application::MakeGPIOSwitch (C++ class), 195
 Application::MakeGPIOSwitch::gpio (C++ member), 195
 Application::MakeGPIOSwitch::mqtt (C++ member), 195
 Application::MakeGPIOSwitch::switch_ (C++ member), 195
 Application::MakeHDC1080Sensor (C++ class), 195
 Application::MakeHDC1080Sensor::hdc1080 (C++ member), 195
 Application::MakeHDC1080Sensor::mqtt_humidity (C++ member), 195
 Application::MakeHDC1080Sensor::mqtt_temperature (C++ member), 195
 Application::MakeHTU21DSensor (C++ class), 195
 Application::MakeHTU21DSensor::htu21d (C++ member), 196
 Application::MakeHTU21DSensor::mqtt_humidity (C++ member), 196
 Application::MakeHTU21DSensor::mqtt_temperature (C++ member), 196
 Application::MakeLight (C++ class), 196
 Application::MakeLight::mqtt (C++ member), 196
 Application::MakeLight::output (C++ member), 196
 Application::MakeLight::state (C++ member), 196
 Application::MakeMAX6675Sensor (C++ class), 196
 Application::MakeMAX6675Sensor::max6675 (C++ member), 196
 Application::MakeMAX6675Sensor::mqtt (C++ member), 196
 Application::MakePulseCounterSensor (C++ class), 196
 Application::MakePulseCounterSensor::mqtt (C++ member), 196
 Application::MakePulseCounterSensor::pcnt (C++ member), 196
 Application::MakeRestartSwitch (C++ class), 196
 Application::MakeRestartSwitch::mqtt (C++ member), 196
 Application::MakeRestartSwitch::restart (C++ member), 196
 Application::MakeRotaryEncoderSensor (C++ class), 196
 Application::MakeRotaryEncoderSensor::mqtt (C++ member), 196
 Application::MakeRotaryEncoderSensor::rotary_encoder (C++ member), 196
 Application::MakeSHT3XDSensor (C++ class), 196
 Application::MakeSHT3XDSensor::mqtt_humidity (C++ member), 197
 Application::MakeSHT3XDSensor::mqtt_temperature (C++ member), 197
 Application::MakeSHT3XDSensor::sht3xd (C++ member), 197
 Application::MakeShutdownSwitch (C++ class), 197
 Application::MakeShutdownSwitch::mqtt (C++ member), 197
 Application::MakeShutdownSwitch::shutdown (C++ member), 197
 Application::MakeSimpleSwitch (C++ class), 197
 Application::MakeSimpleSwitch::mqtt (C++ member), 197
 Application::MakeSimpleSwitch::switch_ (C++ member), 197
 Application::MakeStatusBinarySensor (C++ class), 197
 Application::MakeStatusBinarySensor::mqtt (C++ member), 197
 Application::MakeStatusBinarySensor::status (C++ member), 197
 Application::MakeTemplateBinarySensor (C++ class), 197
 Application::MakeTemplateBinarySensor::mqtt (C++ member), 197
 Application::MakeTemplateCover (C++ class), 197
 Application::MakeTemplateCover::mqtt (C++ member), 197
 Application::MakeTemplateCover::template_ (C++ member), 197
 Application::MakeTemplateSensor (C++ class), 197
 Application::MakeTemplateSensor::mqtt (C++ member), 198
 Application::MakeTemplateSensor::template_ (C++ member), 198
 Application::MakeTemplateSwitch (C++ class), 198
 Application::MakeTemplateSwitch::mqtt (C++ member), 198
 Application::MakeTemplateSwitch::template_ (C++ member), 198
 Application::MakeTSL2561Sensor (C++ class), 198
 Application::MakeTSL2561Sensor::mqtt (C++ member), 198

Application::MakeTSL2561Sensor::tsl2561 (C++ member), 198
Application::MakeUltrasonicSensor (C++ class), 198
Application::MakeUltrasonicSensor::mqtt (C++ member), 198
Application::MakeUltrasonicSensor::ultrasonic (C++ member), 198
Application::mqtt_client_ (C++ member), 193
Application::name_ (C++ member), 193
Application::register_binary_sensor (C++ function), 182
Application::register_component (C++ function), 193
Application::register_controller (C++ function), 193
Application::register_cover (C++ function), 192
Application::register_fan (C++ function), 192
Application::register_light (C++ function), 190
Application::register_sensor (C++ function), 183
Application::register_switch (C++ function), 191
Application::set_name (C++ function), 180
Application::setup (C++ function), 193
Application::wifi_ (C++ member), 193
Automation (C++ class), 247
Automation::actions_ (C++ member), 247
Automation::add_action (C++ function), 247
Automation::add_actions (C++ function), 247
Automation::add_condition (C++ function), 247
Automation::add_conditions (C++ function), 247
Automation::Automation (C++ function), 247
Automation::conditions_ (C++ member), 247
Automation::process_trigger_ (C++ function), 247
Automation::trigger_ (C++ member), 247
AUTOMATION_TAG (C++ member), 242

B

binary_sensor::BinarySensor (C++ class), 305
binary_sensor::BinarySensor::add_on_state_callback (C++ function), 305
binary_sensor::BinarySensor::BinarySensor (C++ function), 305
binary_sensor::BinarySensor::device_class (C++ function), 306
binary_sensor::BinarySensor::device_class_ (C++ member), 306
binary_sensor::BinarySensor::first_value_ (C++ member), 306
binary_sensor::BinarySensor::get_device_class (C++ function), 306
binary_sensor::BinarySensor::get_value (C++ function), 306
binary_sensor::BinarySensor::inverted_ (C++ member), 306
binary_sensor::BinarySensor::is_inverted (C++ function), 306
binary_sensor::BinarySensor::make_click_trigger (C++ function), 306
binary_sensor::BinarySensor::make_double_click_trigger (C++ function), 306
binary_sensor::BinarySensor::make_press_trigger (C++ function), 306
binary_sensor::BinarySensor::make_release_trigger (C++ function), 306
binary_sensor::BinarySensor::publish_state (C++ function), 306
binary_sensor::BinarySensor::set_device_class (C++ function), 306
binary_sensor::BinarySensor::set_inverted (C++ function), 305
binary_sensor::BinarySensor::state_callback (C++ member), 306
binary_sensor::BinarySensor::value (C++ member), 306
binary_sensor::ESP32TouchBinarySensor (C++ class), 304
binary_sensor::ESP32TouchBinarySensor::ESP32TouchBinarySensor (C++ function), 304
binary_sensor::ESP32TouchBinarySensor::get_threshold (C++ function), 304
binary_sensor::ESP32TouchBinarySensor::get_touch_pad (C++ function), 304
binary_sensor::ESP32TouchBinarySensor::threshold_ (C++ member), 304
binary_sensor::ESP32TouchBinarySensor::touch_pad_ (C++ member), 304
binary_sensor::ESP32TouchComponent (C++ class), 300
binary_sensor::ESP32TouchComponent::children_ (C++ member), 304
binary_sensor::ESP32TouchComponent::get_setup_priority (C++ function), 303
binary_sensor::ESP32TouchComponent::high_voltage_reference_ (C++ member), 304
binary_sensor::ESP32TouchComponent::iir_filter_ (C++ member), 304
binary_sensor::ESP32TouchComponent::iir_filter_enabled_ (C++ function), 303
binary_sensor::ESP32TouchComponent::loop (C++ function), 303
binary_sensor::ESP32TouchComponent::low_voltage_reference_ (C++ member), 304
binary_sensor::ESP32TouchComponent::make_touch_pad (C++ function), 300
binary_sensor::ESP32TouchComponent::meas_cycle_ (C++ member), 304
binary_sensor::ESP32TouchComponent::set_high_voltage_reference_ (C++ function), 302
binary_sensor::ESP32TouchComponent::set_iir_filter (C++ function), 301

binary_sensor::ESP32TouchComponent::set_low_voltage
 (C++ function), 302
 binary_sensor::ESP32TouchComponent::set_measurement
 (C++ function), 302
 binary_sensor::ESP32TouchComponent::set_setup_mode
 (C++ function), 301
 binary_sensor::ESP32TouchComponent::set_sleep_duration
 (C++ function), 301
 binary_sensor::ESP32TouchComponent::set_voltage_attenuation
 (C++ function), 303
 binary_sensor::ESP32TouchComponent::setup
 (C++ function), 303
 binary_sensor::ESP32TouchComponent::setup_mode
 (C++ member), 304
 binary_sensor::ESP32TouchComponent::sleep_cycle
 (C++ member), 304
 binary_sensor::ESP32TouchComponent::voltage_attenuation_

C

binary_sensor::GPIOBinarySensorComponent
 (C++ class), 298
 binary_sensor::GPIOBinarySensorComponent::get_setup_priority
 (C++ function), 299
 binary_sensor::GPIOBinarySensorComponent::GPIOBinarySensorComponent
 (C++ function), 298
 binary_sensor::GPIOBinarySensorComponent::loop
 (C++ function), 299
 binary_sensor::GPIOBinarySensorComponent::pin_
 (C++ member), 299
 binary_sensor::GPIOBinarySensorComponent::setup
 (C++ function), 299
 binary_sensor::MQTTBinarySensorComponent
 (C++ class), 307
 binary_sensor::MQTTBinarySensorComponent::binary_sensor::CONSTRUCTION
 (C++ member), 308
 binary_sensor::MQTTBinarySensorComponent::component_type
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::friendly_name
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::get_payload_off_
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::get_payload_on_
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::MQTTBinarySensorComponent
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::payload_off_
 (C++ member), 308
 binary_sensor::MQTTBinarySensorComponent::payload_on_
 (C++ member), 308
 binary_sensor::MQTTBinarySensorComponent::publish_state_
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::send_discovery_
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::set_payload_off_
 (C++ function), 307

binary_sensor::MQTTBinarySensorComponent::set_payload_on_
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::setup_
 (C++ function), 307
 binary_sensor::MQTTBinarySensorComponent::StatusBinarySensor
 (C++ class), 299
 binary_sensor::StatusBinarySensor::device_class

binary_sensor::TemplateBinarySensor
 (C++ class), 304
 binary_sensor::TemplateBinarySensor::loop

C

binary_sensor::TemplateBinarySensor::TemplateBinarySensor
 (C++ class), 304
 binary_sensor::TemplateBinarySensor::callbacks_
 (C++ member), 221
 CallbackManager<void(Ts...)>
 (C++ class), 221

Component
 (C++ class), 198
 Component::cancel_defer

Component::cancel_interval

Component::cancel_time_function

Component::cancel_timeout

Component::component_state_
 (C++ member), 202

Component::ComponentState
 (C++ type), 199

Component::ComponentType
 (C++ enumerator), 199

Component::DEFER
 (C++ enumerator), 202

Component::defer

Component::FAILED
 (C++ enumerator), 199

Component::get_component_state

Component::get_loop_priority

Component::get_setup_priority
 (C++ function), 199

Component::IN_PROGRESS
 (C++ enumerator), 202

Component::is_failed

Component::LOOP
 (C++ enumerator), 199

Component::loop

Component::loop_
 (C++ function), 200

Component::loop_internal

Component::mark_failed

Component::set_interval

Component::SETUP
 (C++ enumerator), 199

Component::setup

Component::setup_
 (C++ function), 199

Component::setup_internal (C++ function), 200
Component::time_func_t (C++ type), 200
Component::time_functions_ (C++ member), 202
Component::TimeFunction (C++ class), 202
Component::TimeFunction::f (C++ member), 202
Component::TimeFunction::interval (C++ member), 202
Component::TimeFunction::last_execution (C++ member), 202
Component::TimeFunction::name (C++ member), 202
Component::TimeFunction::remove (C++ member), 202
Component::TimeFunction::should_run (C++ function), 202
Component::TimeFunction::type (C++ member), 202
Component::TIMEOUT (C++ enumerator), 202
Component::Type (C++ type), 202
Condition (C++ class), 242
Condition::check (C++ function), 242
Controller (C++ class), 232
Controller::register_cover (C++ function), 233
Controller::register_fan (C++ function), 232
Controller::register_light (C++ function), 233
Controller::register_sensor (C++ function), 233
Controller::register_switch (C++ function), 233
cover::CloseAction (C++ class), 348
cover::CloseAction::CloseAction (C++ function), 348
cover::CloseAction::cover_ (C++ member), 348
cover::CloseAction::play (C++ function), 348
cover::Cover (C++ class), 347
cover::Cover::add_on_publish_state_callback (C++ function), 347
cover::Cover::close (C++ function), 347
cover::Cover::Cover (C++ function), 347
cover::Cover::last_state_ (C++ member), 348
cover::Cover::make_close_action (C++ function), 347
cover::Cover::make_open_action (C++ function), 347
cover::Cover::make_stop_action (C++ function), 347
cover::Cover::open (C++ function), 347
cover::Cover::optimistic (C++ function), 347
cover::Cover::publish_state (C++ function), 347
cover::Cover::state_callback_ (C++ member), 348
cover::Cover::stop (C++ function), 347
cover::COVER_CLOSED (C++ enumerator), 348
cover::COVER_MAX (C++ enumerator), 348
cover::COVER_OPEN (C++ enumerator), 348
cover::CoverState (C++ type), 348
cover::MQTTCoverComponent (C++ class), 349
cover::MQTTCoverComponent::component_type (C++ function), 349
cover::MQTTCoverComponent::cover_ (C++ member), 349
cover::MQTTCoverComponent::friendly_name (C++ function), 349
cover::MQTTCoverComponent::MQTTCoverComponent (C++ function), 349
cover::MQTTCoverComponent::send_discovery (C++ function), 349
cover::MQTTCoverComponent::setup (C++ function), 349
cover::OpenAction (C++ class), 348
cover::OpenAction::cover_ (C++ member), 348
cover::OpenAction::OpenAction (C++ function), 348
cover::OpenAction::play (C++ function), 348
cover::StopAction (C++ class), 348
cover::StopAction::cover_ (C++ member), 349
cover::StopAction::play (C++ function), 348
cover::StopAction::StopAction (C++ function), 348
cover::TemplateCover (C++ class), 346
cover::TemplateCover::add_close_actions (C++ function), 346
cover::TemplateCover::add_open_actions (C++ function), 346
cover::TemplateCover::add_stop_actions (C++ function), 346
cover::TemplateCover::close (C++ function), 346
cover::TemplateCover::close_action_ (C++ member), 347
cover::TemplateCover::loop (C++ function), 346
cover::TemplateCover::open (C++ function), 346
cover::TemplateCover::open_action_ (C++ member), 347
cover::TemplateCover::optimistic (C++ function), 346
cover::TemplateCover::optimistic_ (C++ member), 347
cover::TemplateCover::set_optimistic (C++ function), 346
cover::TemplateCover::set_state_lambda (C++ function), 346
cover::TemplateCover::stop (C++ function), 346
cover::TemplateCover::stop_action_ (C++ member), 347
cover::TemplateCover::TemplateCover (C++ function), 346
crc8 (C++ function), 220

D

DebugComponent (C++ class), 355
DebugComponent::free_heap_ (C++ member), 356
DebugComponent::get_setup_priority (C++ function), 356

DebugComponent::loop (C++ function), 356
 DebugComponent::setup (C++ function), 356
 DeepSleepComponent (C++ class), 236
 DeepSleepComponent::at_loop_cycle_ (C++ member), 237
 DeepSleepComponent::begin_sleep (C++ function), 237
 DeepSleepComponent::get_loop_priority (C++ function), 236
 DeepSleepComponent::get_setup_priority (C++ function), 237
 DeepSleepComponent::loop (C++ function), 236
 DeepSleepComponent::loop_cycles_ (C++ member), 237
 DeepSleepComponent::run_duration_ (C++ member), 237
 DeepSleepComponent::set_run_cycles (C++ function), 236
 DeepSleepComponent::set_run_duration (C++ function), 236
 DeepSleepComponent::set_sleep_duration (C++ function), 236
 DeepSleepComponent::set_wakeup_pin (C++ function), 236
 DeepSleepComponent::setup (C++ function), 236
 DeepSleepComponent::sleep_duration_ (C++ member), 237
 DeepSleepComponent::wakeup_pin_ (C++ member), 237
E
 EMPTY (C++ enumerator), 222
 enable_interrupts (C++ function), 220
 ESP32BLEDevice (C++ class), 355
 ESP32BLEDevice::address_ (C++ member), 355
 ESP32BLEDevice::device_class (C++ function), 355
 ESP32BLEDevice::ESP32BLEDevice (C++ function), 355
 ESP32BLETracker (C++ class), 353
 ESP32BLETracker::ble_core_task (C++ function), 355
 ESP32BLETracker::devices_ (C++ member), 354
 ESP32BLETracker::discovered_ (C++ member), 354
 ESP32BLETracker::gap_event_handler (C++ function), 355
 ESP32BLETracker::gap_scan_result (C++ function), 354
 ESP32BLETracker::gap_scan_set_param_complete (C++ function), 354
 ESP32BLETracker::gap_scan_start_complete (C++ function), 354
 ESP32BLETracker::make_device (C++ function), 353
 ESP32BLETracker::scan_interval_ (C++ member), 354
 ESP32BLETracker::scan_params_ (C++ member), 354
 ESP32BLETracker::set_scan_interval (C++ function), 354
 ESP32BLETracker::setup (C++ function), 354
 ESP32BLETracker::start_scan (C++ function), 354
 ESPOneWire (C++ class), 228
 ESPOneWire::ESPOneWire (C++ function), 228
 ESPOneWire::last_device_flag_ (C++ member), 229
 ESPOneWire::last_discrepancy_ (C++ member), 229
 ESPOneWire::last_family_discrepancy_ (C++ member), 229
 ESPOneWire::pin_ (C++ member), 229
 ESPOneWire::read64 (C++ function), 229
 ESPOneWire::read8 (C++ function), 229
 ESPOneWire::read_bit (C++ function), 228
 ESPOneWire::reset (C++ function), 228
 ESPOneWire::reset_search (C++ function), 229
 ESPOneWire::rom_number (C++ member), 229
 ESPOneWire::rom_number8_ (C++ function), 229
 ESPOneWire::search (C++ function), 229
 ESPOneWire::search_vec (C++ function), 229
 ESPOneWire::select (C++ function), 229
 ESPOneWire::skip (C++ function), 229
 ESPOneWire::write64 (C++ function), 229
 ESPOneWire::write8 (C++ function), 228
 ESPOneWire::write_bit (C++ function), 228
 ESPPreferences (C++ class), 225
 ESPPreferences::begin (C++ function), 225
 ESPPreferences::get_bool (C++ function), 225
 ESPPreferences::get_double (C++ function), 226
 ESPPreferences::get_float (C++ function), 226
 ESPPreferences::get_int16 (C++ function), 225
 ESPPreferences::get_int32 (C++ function), 226
 ESPPreferences::get_int64 (C++ function), 226
 ESPPreferences::get_int8 (C++ function), 225
 ESPPreferences::get_preference_key (C++ function), 226
 ESPPreferences::get_uint16 (C++ function), 226
 ESPPreferences::get_uint32 (C++ function), 226
 ESPPreferences::get_uint64 (C++ function), 226
 ESPPreferences::get_uint8 (C++ function), 225
 ESPPreferences::preferences_ (C++ member), 226
 ESPPreferences::put_bool (C++ function), 225

ESPPreferences::put_double (C++ function), 225
ESPPreferences::put_float (C++ function), 225
ESPPreferences::put_int16 (C++ function), 225
ESPPreferences::put_int32 (C++ function), 225
ESPPreferences::put_int64 (C++ function), 225
ESPPreferences::put_int8 (C++ function), 225
ESPPreferences::put_uint16 (C++ function), 225
ESPPreferences::put_uint32 (C++ function), 225
ESPPreferences::put_uint64 (C++ function), 225
ESPPreferences::put_uint8 (C++ function), 225
ExponentialMovingAverage (C++ class), 221
ExponentialMovingAverage::accumulator_ (C++ member), 221
ExponentialMovingAverage::alpha_ (C++ member), 221
ExponentialMovingAverage::calculate_average (C++ function), 221
ExponentialMovingAverage::ExponentialMovingAverage (C++ function), 221
ExponentialMovingAverage::get_alpha (C++ function), 221
ExponentialMovingAverage::next_value (C++ function), 221
ExponentialMovingAverage::set_alpha (C++ function), 221

F

fan::BasicFanComponent (C++ class), 318
fan::BasicFanComponent::binary_output_ (C++ member), 319
fan::BasicFanComponent::get_state (C++ function), 319
fan::BasicFanComponent::high_speed_ (C++ member), 319
fan::BasicFanComponent::loop (C++ function), 319
fan::BasicFanComponent::low_speed_ (C++ member), 319
fan::BasicFanComponent::medium_speed_ (C++ member), 319
fan::BasicFanComponent::next_update_ (C++ member), 319
fan::BasicFanComponent::off_speed_ (C++ member), 319
fan::BasicFanComponent::oscillating_output_ (C++ member), 319
fan::BasicFanComponent::set_binary (C++ function), 318
fan::BasicFanComponent::set_oscillation (C++ function), 319
fan::BasicFanComponent::set_speed (C++ function), 318
fan::BasicFanComponent::set_state (C++ function), 319
fan::BasicFanComponent::setup (C++ function), 319

fan::BasicFanComponent::speed_output_ (C++ member), 319
fan::BasicFanComponent::state_ (C++ member), 319
fan::FanState (C++ class), 316
fan::FanState::add_on_state_change_callback (C++ function), 316
fan::FanState::FanState (C++ function), 316
fan::FanState::get_speed (C++ function), 317
fan::FanState::get_state (C++ function), 316
fan::FanState::get_traits (C++ function), 317
fan::FanState::is_oscillating (C++ function), 316
fan::FanState::load_from_preferences (C++ function), 317
fan::FanState::make_toggle_action (C++ function), 317
fan::FanState::make_turn_off_action (C++ function), 317
fan::FanState::make_turn_on_action (C++ function), 317
fan::FanState::oscillating_ (C++ member), 317
fan::FanState::save_to_preferences (C++ function), 317
fan::FanState::set_oscillating (C++ function), 317
fan::FanState::set_speed (C++ function), 317
fan::FanState::set_state (C++ function), 316
fan::FanState::set_traits (C++ function), 317
fan::FanState::speed_ (C++ member), 317
fan::FanState::state_ (C++ member), 317
fan::FanState::state_callback_ (C++ member), 317
fan::FanState::traits_ (C++ member), 317
fan::FanTraits (C++ class), 317
fan::FanTraits::FanTraits (C++ function), 317
fan::FanTraits::oscillation_ (C++ member), 318
fan::FanTraits::set_oscillation (C++ function), 318
fan::FanTraits::set_speed (C++ function), 318
fan::FanTraits::speed_ (C++ member), 318
fan::FanTraits::supports_oscillation (C++ function), 318
fan::FanTraits::supports_speed (C++ function), 318
fan::MQTTFanComponent (C++ class), 319
fan::MQTTFanComponent::component_type (C++ function), 320
fan::MQTTFanComponent::friendly_name (C++ function), 320
fan::MQTTFanComponent::get_oscillation_command_topic (C++ function), 320
fan::MQTTFanComponent::get_oscillation_state_topic (C++ function), 320
fan::MQTTFanComponent::get_speed_command_topic (C++ function), 320
fan::MQTTFanComponent::get_speed_state_topic (C++ function), 320

fan::MQTTFanComponent::get_state (C++ function), 320

fan::MQTTFanComponent::MQTTFanComponent (C++ function), 320

fan::MQTTFanComponent::send_discovery (C++ function), 320

fan::MQTTFanComponent::send_state (C++ function), 320

fan::MQTTFanComponent::set_custom_oscillation_ (C++ function), 320

fan::MQTTFanComponent::set_custom_oscillation_ (C++ function), 320

fan::MQTTFanComponent::set_custom_speed_command_ (C++ function), 320

fan::MQTTFanComponent::set_custom_speed_state_ (C++ function), 320

fan::MQTTFanComponent::setup (C++ function), 320

fan::MQTTFanComponent::state_ (C++ member), 320

G

gamma_correct (C++ function), 219

generate_hostname (C++ function), 218

get_mac_address (C++ function), 218

global_esp32_ble_tracker (C++ member), 355

global_preferences (C++ member), 226

GPIOInputPin (C++ class), 228

GPIOInputPin::GPIOInputPin (C++ function), 228

GPIOOutputPin (C++ class), 227

GPIOOutputPin::GPIOOutputPin (C++ function), 228

GPIOPin (C++ class), 226

GPIOPin::copy (C++ function), 227

GPIOPin::digital_read (C++ function), 227

GPIOPin::digital_write (C++ function), 227

GPIOPin::get_mode (C++ function), 227

GPIOPin::get_pin (C++ function), 227

GPIOPin::GPIOPin (C++ function), 226, 227

GPIOPin::inverted_ (C++ member), 227

GPIOPin::is_inverted (C++ function), 227

GPIOPin::mode_ (C++ member), 227

GPIOPin::pin_ (C++ member), 227

GPIOPin::pin_mode (C++ function), 227

GPIOPin::set_inverted (C++ function), 227

GPIOPin::set_mode (C++ function), 227

GPIOPin::set_pin (C++ function), 227

GPIOPin::setup (C++ function), 227

H

HOSTNAME_CHARACTER_WHITELIST (C++ member), 220

I

I2CComponent (C++ class), 237

I2CComponent::begin_transmission_ (C++ function), 239

I2CComponent::end_transmission_ (C++ function), 239

I2CComponent::frequency_ (C++ member), 240

I2CComponent::get_setup_priority (C++ function), 240

I2CComponent::I2CComponent (C++ function), 238

I2CComponent::loop (C++ function), 240

I2CComponent::read_ (C++ function), 239

I2CComponent::read_topic

I2CComponent::read_byte (C++ function), 239

I2CComponent::read_byte_16 (C++ function), 239

I2CComponent::read_bytes (C++ function), 238

I2CComponent::read_bytes_16 (C++ function), 238

I2CComponent::receive_ (C++ function), 240

I2CComponent::receive_16_ (C++ function), 240

I2CComponent::receive_timeout_ (C++ member), 240

I2CComponent::request_from_ (C++ function), 239

I2CComponent::scan_ (C++ member), 240

I2CComponent::scl_pin_ (C++ member), 240

I2CComponent::sda_pin_ (C++ member), 240

I2CComponent::set_frequency (C++ function), 238

I2CComponent::set_receive_timeout_ (C++ function), 238

I2CComponent::set_scan (C++ function), 238

I2CComponent::set_scl_pin (C++ function), 238

I2CComponent::set_sda_pin (C++ function), 238

I2CComponent::setup (C++ function), 240

I2CComponent::wire_ (C++ member), 240

I2CComponent::write_ (C++ function), 240

I2CComponent::write_16_ (C++ function), 240

I2CComponent::write_byte (C++ function), 239

I2CComponent::write_byte_16 (C++ function), 239

I2CComponent::write_bytes (C++ function), 239

I2CComponent::write_bytes_16 (C++ function), 239

I2DDevice (C++ class), 240

I2DDevice::address_ (C++ member), 242

I2DDevice::I2DDevice (C++ function), 241

I2DDevice::parent_ (C++ member), 242

I2DDevice::read_byte (C++ function), 241

I2DDevice::read_byte_16 (C++ function), 241

I2DDevice::read_bytes (C++ function), 241

I2DDevice::read_bytes_16 (C++ function), 241

I2DDevice::set_address (C++ function), 241

I2DDevice::set_parent (C++ function), 241

I2DDevice::write_byte (C++ function), 242

I2DDevice::write_byte_16 (C++ function), 242

I2DDevice::write_bytes (C++ function), 241

I2DDevice::write_bytes_16 (C++ function), 242

io::PCF8574Component (C++ class), 350

io::PCF8574Component::ddr_mask_ (C++ member), 351
io::PCF8574Component::digital_read_ (C++ function), 351
io::PCF8574Component::digital_write_ (C++ function), 351
io::PCF8574Component::input_mask_ (C++ member), 351
io::PCF8574Component::make_input_pin (C++ function), 350
io::PCF8574Component::make_output_pin (C++ function), 350
io::PCF8574Component::PCF8574Component (C++ function), 350
io::PCF8574Component::pcf8575_ (C++ member), 351
io::PCF8574Component::pin_mode_ (C++ function), 351
io::PCF8574Component::port_mask_ (C++ member), 351
io::PCF8574Component::read_gpio_ (C++ function), 351
io::PCF8574Component::setup (C++ function), 351
io::PCF8574Component::write_gpio_ (C++ function), 351
io::PCF8574GPIOInputPin (C++ class), 351
io::PCF8574GPIOInputPin::copy (C++ function), 351
io::PCF8574GPIOInputPin::digital_read (C++ function), 351
io::PCF8574GPIOInputPin::digital_write (C++ function), 352
io::PCF8574GPIOInputPin::parent_ (C++ member), 352
io::PCF8574GPIOInputPin::PCF8574GPIOInputPin (C++ function), 351
io::PCF8574GPIOInputPin::pin_mode (C++ function), 351
io::PCF8574GPIOInputPin::setup (C++ function), 351
io::PCF8574GPIOOutputPin (C++ class), 352
io::PCF8574GPIOOutputPin::copy (C++ function), 352
io::PCF8574GPIOOutputPin::digital_read (C++ function), 352
io::PCF8574GPIOOutputPin::digital_write (C++ function), 352
io::PCF8574GPIOOutputPin::parent_ (C++ member), 352
io::PCF8574GPIOOutputPin::PCF8574GPIOOutputPin (C++ function), 352
io::PCF8574GPIOOutputPin::pin_mode (C++ function), 352
io::PCF8574GPIOOutputPin::setup (C++ function), 352
is_empty (C++ function), 218
J
json_build_t (C++ type), 218
json_parse_t (C++ type), 218
L
LAMBDA (C++ enumerator), 222
LambdaAction (C++ class), 246
LambdaAction::f_ (C++ member), 246
LambdaAction::LambdaAction (C++ function), 246
LambdaAction::play (C++ function), 246
LambdaCondition (C++ class), 243
LambdaCondition::check (C++ function), 243
LambdaCondition::f_ (C++ member), 243
LambdaCondition::LambdaCondition (C++ function), 243
lerp (C++ function), 219
light::BinaryLightOutput (C++ class), 328
light::BinaryLightOutput::BinaryLightOutput (C++ function), 328
light::BinaryLightOutput::get_traits (C++ function), 328
light::BinaryLightOutput::output_ (C++ member), 328
light::BinaryLightOutput::write_state (C++ function), 328
light::FastLEDLightOutputComponent (C++ class), 321
light::FastLEDLightOutputComponent::add_leds (C++ function), 322
light::FastLEDLightOutputComponent::controller_ (C++ member), 323
light::FastLEDLightOutputComponent::get_controller (C++ function), 321
light::FastLEDLightOutputComponent::get_leds (C++ function), 321
light::FastLEDLightOutputComponent::get_num_leds (C++ function), 321
light::FastLEDLightOutputComponent::get_setup_priority (C++ function), 323
light::FastLEDLightOutputComponent::get_traits (C++ function), 322
light::FastLEDLightOutputComponent::has_requested_high_power_ (C++ member), 323
light::FastLEDLightOutputComponent::last_refresh_ (C++ member), 323
light::FastLEDLightOutputComponent::leds_ (C++ member), 323
light::FastLEDLightOutputComponent::loop (C++ function), 322
light::FastLEDLightOutputComponent::max_refresh_rate_ (C++ member), 323

light::FastLEDDOutputComponent::next_show_
 (C++ member), 323
 light::FastLEDDOutputComponent::num_leds_
 (C++ member), 323
 light::FastLEDDOutputComponent::power_supply_
 (C++ member), 323
 light::FastLEDDOutputComponent::prevent_writing_leds_
 (C++ function), 321
 light::FastLEDDOutputComponent::prevent_writing_lighted_
 (C++ member), 323
 light::FastLEDDOutputComponent::schedule_show_
 (C++ function), 321
 light::FastLEDDOutputComponent::set_max_refresh_rate_
 (C++ function), 321
 light::FastLEDDOutputComponent::set_power_supply_
 (C++ function), 322
 light::FastLEDDOutputComponent::setup (C++
 function), 322
 light::FastLEDDOutputComponent::unprevent_writing_led\$ion), 325
 (C++ function), 321
 light::FastLEDDOutputComponent::write_state_
 (C++ function), 322
 light::FastLEDRainbowLightEffect (C++ class), 327
 light::FastLEDRainbowLightEffect::apply_effect
 (C++ function), 328
 light::FastLEDRainbowLightEffect::create_ (C++
 function), 328
 light::FastLEDRainbowLightEffect::FastLEDRainbowLightEffect
 (C++ function), 327
 light::FastLEDRainbowLightEffect::get_name
 (C++ function), 327
 light::FastLEDRainbowLightEffect::initialize_ (C++
 function), 328
 light::FastLEDRainbowLightEffect::stop (C++ func-
 tion), 328
 light::light_effect_entries (C++ member), 328
 light::LightColorValues (C++ class), 323
 light::LightColorValues::as_binary (C++ function),
 325
 light::LightColorValues::as_brightness (C++ func-
 tion), 325
 light::LightColorValues::as_rgb (C++ function), 325
 light::LightColorValues::as_rbgw (C++ function),
 325
 light::LightColorValues::blue_ (C++ member), 326
 light::LightColorValues::brightness_ (C++ mem-
 ber), 326
 light::LightColorValues::dump_json_ (C++ func-
 tion), 324
 light::LightColorValues::get_blue_ (C++ function),
 325
 light::LightColorValues::get_brightness (C++ func-
 tion), 325
 light::LightColorValues::get_green_ (C++ function),
 325
 light::LightColorValues::get_red_ (C++ function),
 325
 light::LightColorValues::get_state (C++ function),
 325
 light::LightColorValues::get_white (C++ function),
 325
 light::LightColorValues::green_ (C++ member), 326
 light::LightColorValues::lerp (C++ function), 325
 light::LightColorValues::LightColorValues (C++
 function), 324
 light::LightColorValues::load_from_preferences
 (C++ function), 324
 light::LightColorValues::normalize_color
 (C++ function), 324
 light::LightColorValues::operator
 = (C++ function), 325
 light::LightColorValues::operator== (C++ func-
 tion), 325
 light::LightColorValues::parse_json (C++ function),
 324
 light::LightColorValues::red_ (C++ member), 326
 light::LightColorValues::save_to_preferences (C++
 function), 324
 light::LightColorValues::set_blue (C++ function),
 325
 light::LightColorValues::set_brightness (C++ func-
 tion), 325
 light::LightColorValues::set_red_ (C++ function),
 325
 light::LightColorValues::set_state (C++ function),
 325
 light::LightColorValues::set_white (C++ function),
 325
 light::LightColorValues::state_ (C++ member), 326
 light::LightColorValues::white_ (C++ member), 326
 light::LightEffect (C++ class), 326
 light::LightEffect::apply_effect (C++ function), 326
 light::LightEffect::Entry (C++ class), 326
 light::LightEffect::Entry::name (C++ member), 326
 light::LightEffect::Entry::requirements (C++ mem-
 ber), 326
 light::LightEffect::get_name (C++ function), 326
 light::LightEffect::initialize (C++ function), 326
 light::LightEffect::stop (C++ function), 326
 light::LightFlashTransformer (C++ class), 334
 light::LightFlashTransformer::get_end_values
 (C++ function), 334
 light::LightFlashTransformer::get_values_ (C++
 function), 334
 light::LightFlashTransformer::is_continuous_ (C++
 function), 334

light::LightFlashTransformer::LightFlashTransformer (C++ function), 334
light::LightOutput (C++ class), 328
light::LightOutput::get_traits (C++ function), 328
light::LightOutput::write_state (C++ function), 328
light::LightState (C++ class), 330
light::LightState::add_new_remote_values_callback (C++ function), 331
light::LightState::current_values_as_binary (C++ function), 331
light::LightState::current_values_as_brightness (C++ function), 331
light::LightState::current_values_as_rgb (C++ function), 331
light::LightState::current_values_as_rgbw (C++ function), 331
light::LightState::default_transition_length_ (C++ member), 332
light::LightState::dump_json (C++ function), 332
light::LightState::effect_ (C++ member), 332
light::LightState::gamma_correct_ (C++ member), 332
light::LightState::get_current_values (C++ function), 331
light::LightState::get_current_values_lazy (C++ function), 331
light::LightState::get_default_transition_length (C++ function), 332
light::LightState::get_effect_name (C++ function), 331
light::LightState::get_gamma_correct (C++ function), 332
light::LightState::get_output (C++ function), 331
light::LightState::get_remote_values (C++ function), 331
light::LightState::get_setup_priority (C++ function), 331
light::LightState::get_traits (C++ function), 331
light::LightState::LightState (C++ function), 330
light::LightState::loop (C++ function), 331
light::LightState::make_toggle_action (C++ function), 331
light::LightState::make_turn_off_action (C++ function), 331
light::LightState::make_turn_on_action (C++ function), 331
light::LightState::next_write_ (C++ member), 332
light::LightState::output_ (C++ member), 332
light::LightState::parse_json (C++ function), 332
light::LightState::remote_values_callback_ (C++ member), 332
light::LightState::send_values (C++ function), 331
light::LightState::set_default_transition_length (C++ function), 332
light::LightState::set_gamma_correct (C++ function), 332
light::LightState::set_immediately (C++ function), 330
light::LightState::set_immediately_without_sending (C++ function), 331
light::LightState::set_transformer (C++ function), 331
light::LightState::setup (C++ function), 331
light::LightState::start_default_transition (C++ function), 331
light::LightState::start_effect (C++ function), 330
light::LightState::start_flash (C++ function), 330
light::LightState::start_transition (C++ function), 330
light::LightState::stop_effect (C++ function), 330
light::LightState::supports_effects (C++ function), 332
light::LightState::transformer_ (C++ member), 332
light::LightState::values_ (C++ member), 332
light::LightTraits (C++ class), 332
light::LightTraits::brightness_ (C++ member), 333
light::LightTraits::fast_led_ (C++ member), 333
light::LightTraits::has_brightness (C++ function), 333
light::LightTraits::has_fast_led (C++ function), 333
light::LightTraits::has_rgb (C++ function), 333
light::LightTraits::has_rgb_white_value (C++ function), 333
light::LightTraits::LightTraits (C++ function), 333
light::LightTraits::rgb_ (C++ member), 333
light::LightTraits::rgb_white_value_ (C++ member), 333
light::LightTraits::supports_traits (C++ function), 333
light::LightTransformer (C++ class), 333
light::LightTransformer::get_end_values (C++ function), 333
light::LightTransformer::get_progress (C++ function), 334
light::LightTransformer::get_remote_values (C++ function), 333
light::LightTransformer::get_start_values (C++ function), 334
light::LightTransformer::get_target_values (C++ function), 334
light::LightTransformer::get_values (C++ function), 333
light::LightTransformer::is_continuous (C++ function), 333
light::LightTransformer::is_finished (C++ function), 333
light::LightTransformer::length_ (C++ member), 334

light::LightTransformer::LightTransformer (C++ function), 333
 light::LightTransformer::start_time_ (C++ member), 334
 light::LightTransformer::start_values_ (C++ member), 334
 light::LightTransformer::target_values_ (C++ member), 334
 light::LightTransitionTransformer (C++ class), 334
 light::LightTransitionTransformer::get_values (C++ function), 334
 light::LightTransitionTransformer::is_continuous (C++ function), 334
 light::LightTransitionTransformer::LightTransitionTransformer (C++ function), 334
 light::MonochromaticLightOutput (C++ class), 328
 light::MonochromaticLightOutput::get_traits (C++ function), 329
 light::MonochromaticLightOutput::MonochromaticLightOutput (C++ function), 329
 light::MonochromaticLightOutput::output_ (C++ member), 329
 light::MonochromaticLightOutput::write_state (C++ function), 329
 light::MQTTJSONLightComponent (C++ class), 335
 light::MQTTJSONLightComponent::component_type (C++ function), 335
 light::MQTTJSONLightComponent::friendly_name (C++ function), 335
 light::MQTTJSONLightComponent::get_state (C++ function), 335
 light::MQTTJSONLightComponent::MQTTJSONLightComponent (C++ function), 335
 light::MQTTJSONLightComponent::send_discovery (C++ function), 335
 light::MQTTJSONLightComponent::send_light_value (C++ function), 335
 light::MQTTJSONLightComponent::setup (C++ function), 335
 light::MQTTJSONLightComponent::state_ (C++ member), 335
 light::NoneLightEffect (C++ class), 326
 light::NoneLightEffect::apply_effect (C++ function), 327
 light::NoneLightEffect::create (C++ function), 327
 light::NoneLightEffect::get_name (C++ function), 327
 light::RandomLightEffect (C++ class), 327
 light::RandomLightEffect::apply_effect (C++ function), 327
 light::RandomLightEffect::create (C++ function), 327
 light::RandomLightEffect::get_name (C++ function), 327
 tion), 327
 light::RandomLightEffect::last_color_change_ (C++ member), 327
 light::RandomLightEffect::RandomLightEffect (C++ function), 327
 light::RGBLightOutput (C++ class), 329
 light::RGBLightOutput::blue_ (C++ member), 329
 light::RGBLightOutput::get_traits (C++ function), 329
 light::RGBLightOutput::green_ (C++ member), 329
 light::RGBLightOutput::red_ (C++ member), 329
 light::RGBLightOutput::RGBLightOutput (C++ function), 329
 light::RGBLightOutput::write_state (C++ function), 329
 light::RGBWLightOutput (C++ class), 329
 light::RGBWLightOutput::blue_ (C++ member), 330
 light::RGBWLightOutput::get_traits (C++ function), 329
 light::RGBWLightOutput::green_ (C++ member), 330
 light::RGBWLightOutput::red_ (C++ member), 330
 light::RGBWLightOutput::RGBWLightOutput (C++ function), 329
 light::RGBWLightOutput::white_ (C++ member), 330
 light::RGBWLightOutput::write_state (C++ function), 329
 LogComponent (C++ class), 230
 LogComponent::add_on_log_callback (C++ function), 230
 LogComponent::baud_rate_ (C++ member), 230
 LogComponent::get_baud_rate (C++ function), 230
 LogComponent::get_tx_buffer_size (C++ function), 230
 LogComponent::global_log_level_ (C++ member), 230
 LogComponent::log_callback_ (C++ member), 231
 LogComponent::log_levels_ (C++ member), 230
 LogComponent::log_vprintf_ (C++ function), 230
 LogComponent::LogComponent (C++ function), 230
 LogComponent::pre_setup (C++ function), 230
 LogComponent::set_baud_rate (C++ function), 230
 LogComponent::set_global_log_level (C++ function), 230
 LogComponent::set_log_level (C++ function), 230
 LogComponent::set_tx_buffer_size (C++ function), 230
 LogComponent::tx_buffer_ (C++ member), 230

M

make_optional (C++ function), 223
make_unique (C++ function), 219
ManualIP (C++ class), 206
ManualIP::dns1 (C++ member), 206
ManualIP::dns2 (C++ member), 206
ManualIP::gateway (C++ member), 206
ManualIP::static_ip (C++ member), 206
ManualIP::subnet (C++ member), 206
mqtt::Availability (C++ class), 211
mqtt::Availability::payload_available (C++ member), 211
mqtt::Availability::payload_not_available (C++ member), 211
mqtt::Availability::topic (C++ member), 211
mqtt::global_mqtt_client (C++ member), 211
mqtt::MQTTClientComponent (C++ class), 206
mqtt::MQTTClientComponent::add_on_connect_callback (C++ function), 209
mqtt::MQTTClientComponent::add_ssl_fingerprint (C++ function), 207
mqtt::MQTTClientComponent::availability_ (C++ member), 210
mqtt::MQTTClientComponent::birth_message_ (C++ member), 209
mqtt::MQTTClientComponent::credentials_ (C++ member), 209
mqtt::MQTTClientComponent::disable_birth_message (C++ function), 207
mqtt::MQTTClientComponent::disable_discovery (C++ function), 207
mqtt::MQTTClientComponent::disable_last_will (C++ function), 206
mqtt::MQTTClientComponent::disable_log_message (C++ function), 208
mqtt::MQTTClientComponent::discovery_info_ (C++ member), 210
mqtt::MQTTClientComponent::get_availability (C++ function), 207
mqtt::MQTTClientComponent::get_discovery_info (C++ function), 207
mqtt::MQTTClientComponent::get_setup_priority (C++ function), 209
mqtt::MQTTClientComponent::get_topic_prefix (C++ function), 208
mqtt::MQTTClientComponent::is_connected (C++ function), 209
mqtt::MQTTClientComponent::is_discovery_enabled (C++ function), 207
mqtt::MQTTClientComponent::is_log_message_enabled (C++ function), 208
mqtt::MQTTClientComponent::last_will_ (C++ member), 209
mqtt::MQTTClientComponent::log_message_ (C++ member), 210
mqtt::MQTTClientComponent::loop (C++ function), 209
mqtt::MQTTClientComponent::make_message_trigger (C++ function), 209
mqtt::MQTTClientComponent::make_publish_action (C++ function), 209
mqtt::MQTTClientComponent::mqtt_client_ (C++ member), 210
mqtt::MQTTClientComponent::MQTTClientComponent (C++ function), 206
mqtt::MQTTClientComponent::on_connect_ (C++ member), 210
mqtt::MQTTClientComponent::on_message_ (C++ function), 209
mqtt::MQTTClientComponent::publish (C++ function), 208
mqtt::MQTTClientComponent::publish_json (C++ function), 208
mqtt::MQTTClientComponent::recalculate_availability (C++ function), 209
mqtt::MQTTClientComponent::reconnect (C++ function), 209
mqtt::MQTTClientComponent::set_birth_message (C++ function), 206
mqtt::MQTTClientComponent::set_client_id (C++ function), 207
mqtt::MQTTClientComponent::set_discovery_info (C++ function), 207
mqtt::MQTTClientComponent::set_keep_alive (C++ function), 207
mqtt::MQTTClientComponent::set_last_will (C++ function), 206
mqtt::MQTTClientComponent::set_log_message_template (C++ function), 208
mqtt::MQTTClientComponent::set_topic_prefix (C++ function), 207
mqtt::MQTTClientComponent::setup (C++ function), 209
mqtt::MQTTClientComponent::subscribe (C++ function), 208
mqtt::MQTTClientComponent::subscribe_json (C++ function), 208
mqtt::MQTTClientComponent::subscriptions_ (C++ member), 210
mqtt::MQTTClientComponent::topic_prefix_ (C++ member), 210
mqtt::MQTTComponent (C++ class), 211
mqtt::MQTTComponent::availability_ (C++ member), 214
mqtt::MQTTComponent::component_type (C++ function), 212
mqtt::MQTTComponent::custom_topics_ (C++

member), 214
`mqtt::MQTTComponent::disable_availability` (C++ function), 213
`mqtt::MQTTComponent::disable_discovery` (C++ function), 212
`mqtt::MQTTComponent::discovery_enabled` (C++ member), 214
`mqtt::MQTTComponent::friendly_name` (C++ function), 213
`mqtt::MQTTComponent::get_command_topic` (C++ function), 213
`mqtt::MQTTComponent::get_default_object_id` (C++ function), 214
`mqtt::MQTTComponent::get_default_topic_for` (C++ function), 213
`mqtt::MQTTComponent::get_discovery_topic` (C++ function), 213
`mqtt::MQTTComponent::get_retain` (C++ function), 212
`mqtt::MQTTComponent::get_setup_priority` (C++ function), 212
`mqtt::MQTTComponent::get_state_topic` (C++ function), 213
`mqtt::MQTTComponent::get_topic_for` (C++ function), 213
`mqtt::MQTTComponent::is_discovery_enabled` (C++ function), 212
`mqtt::MQTTComponent::loop` (C++ function), 212
`mqtt::MQTTComponent::MQTTComponent` (C++ function), 212
`mqtt::MQTTComponent::next_send_discovery` (C++ member), 214
`mqtt::MQTTComponent::retain` (C++ member), 214
`mqtt::MQTTComponent::send_discovery` (C++ function), 212
`mqtt::MQTTComponent::send_discovery` (C++ function), 213
`mqtt::MQTTComponent::send_json_message` (C++ function), 213
`mqtt::MQTTComponent::send_message` (C++ function), 213
`mqtt::MQTTComponent::set_availability` (C++ function), 212
`mqtt::MQTTComponent::set_custom_command_topic` (C++ function), 212
`mqtt::MQTTComponent::set_custom_state_topic` (C++ function), 212
`mqtt::MQTTComponent::set_custom_topic` (C++ function), 212
`mqtt::MQTTComponent::set_retain` (C++ function), 212
`mqtt::MQTTComponent::setup` (C++ function), 212
`mqtt::MQTTComponent::subscribe` (C++ function), 214
`mqtt::MQTTComponent::subscribe_json` (C++ function), 214
`mqtt::MQTTCredentials` (C++ class), 210
`mqtt::MQTTCredentials::address` (C++ member), 211
`mqtt::MQTTCredentials::client_id` (C++ member), 211
`mqtt::MQTTCredentials::password` (C++ member), 211
`mqtt::MQTTCredentials::port` (C++ member), 211
`mqtt::MQTTCredentials::username` (C++ member), 211
`mqtt::MQTTDiscoveryInfo` (C++ class), 211
`mqtt::MQTTDiscoveryInfo::prefix` (C++ member), 211
`mqtt::MQTTDiscoveryInfo::retain` (C++ member), 211
`mqtt::MQTTMessage` (C++ class), 210
`mqtt::MQTTMessage::payload` (C++ member), 210
`mqtt::MQTTMessage::qos` (C++ member), 210
`mqtt::MQTTMessage::retain` (C++ member), 210
`mqtt::MQTTMessage::topic` (C++ member), 210
`mqtt::MQTTSubscription` (C++ class), 210
`mqtt::MQTTSubscription::callback` (C++ member), 210
`mqtt::MQTTSubscription::qos` (C++ member), 210
`mqtt::MQTTSubscription::topic` (C++ member), 210
`mqtt::SendDiscoveryConfig` (C++ class), 214
`mqtt::SendDiscoveryConfig::command_topic` (C++ member), 215
`mqtt::SendDiscoveryConfig::platform` (C++ member), 215
`mqtt::SendDiscoveryConfig::state_topic` (C++ member), 215

N

`NoArg` (C++ type), 242
`nullopt_t` (C++ class), 223
`nullopt_t::nullopt_t` (C++ function), 223

O

`operator=` (C++ function), 222, 223
`operator==` (C++ function), 222, 223
`operator>` (C++ function), 222, 223
`operator>=` (C++ function), 222, 223
`operator<` (C++ function), 222, 223
`operator<=` (C++ function), 222, 223
`optional` (C++ class), 223
`optional::has_value` (C++ function), 224

optional::has_value_ (C++ member), 225
optional::initialize (C++ function), 224
optional::operator safe_bool (C++ function), 224
optional::operator* (C++ function), 224
optional::operator-> (C++ function), 224
optional::operator= (C++ function), 224
optional::optional (C++ function), 224
optional::reset (C++ function), 224
optional::swap (C++ function), 224
optional::this_type_does_not_support_comparisons
 (C++ function), 224
optional::value (C++ function), 224
optional::value_ (C++ member), 225
optional::value_or (C++ function), 224
optional::value_type (C++ type), 224
optional<T>::safe_bool (C++ type), 224
OrCondition (C++ class), 243
OrCondition::check (C++ function), 243
OrCondition::conditions_ (C++ member), 243
OrCondition::OrCondition (C++ function), 243
OTACComponent (C++ class), 215
OTACComponent::__anonymous1 (C++ type), 217
OTACComponent::at_ota_progress_message_
 (C++ member), 217
OTACComponent::clean_rtc (C++ function), 217
OTACComponent::get_hostname (C++ function),
 217
OTACComponent::get_port (C++ function), 217
OTACComponent::get_setup_priority (C++ func-
 tion), 216
OTACComponent::has_safe_mode_ (C++ member),
 217
OTACComponent::HASH (C++ enumerator), 217
OTACComponent::hostname_ (C++ member), 217
OTACComponent::loop (C++ function), 216
OTACComponent::OPEN (C++ enumerator), 217
OTACComponent::ota_triggered_ (C++ member),
 217
OTACComponent::OTACComponent (C++ function),
 215
OTACComponent::password_ (C++ member), 217
OTACComponent::PLAINTEXT (C++ enumerator),
 217
OTACComponent::port_ (C++ member), 217
OTACComponent::read_rtc_ (C++ function), 217
OTACComponent::safe_mode_enable_time_ (C++
 member), 217
OTACComponent::safe_mode_num_attempts_
 (C++ member), 217
OTACComponent::safe_mode_rtc_value_ (C++
 member), 217
OTACComponent::safe_mode_start_time_ (C++
 member), 217
OTACComponent::server_ (C++ member), 217
OTACComponent::set_auth_open (C++ function),
 215
OTACComponent::set_auth_password_hash (C++
 function), 216
OTACComponent::set_auth_plaintext_password
 (C++ function), 216
OTACComponent::set_hostname (C++ function),
 216
OTACComponent::set_port (C++ function), 216
OTACComponent::setup (C++ function), 216
OTACComponent::start_safe_mode (C++ function),
 216
OTACComponent::write_rtc_ (C++ function), 217
output::BinaryOutput (C++ class), 314
output::BinaryOutput::disable (C++ function), 314
output::BinaryOutput::enable (C++ function), 314
output::BinaryOutput::get_power_supply
 (C++ function), 314
output::BinaryOutput::has_requested_high_power_
 (C++ member), 315
output::BinaryOutput::inverted_ (C++ member),
 315
output::BinaryOutput::is_inverted (C++ function),
 314
output::BinaryOutput::power_supply_ (C++ mem-
 ber), 315
output::BinaryOutput::set_inverted (C++ func-
 tion), 314
output::BinaryOutput::set_power_supply (C++
 function), 314
output::BinaryOutput::write_enabled (C++ func-
 tion), 314
output::ESP8266PWMOutput (C++ class), 313
output::ESP8266PWMOutput::ESP8266PWMOutput
 (C++ function), 313
output::ESP8266PWMOutput::get_pin (C++ func-
 tion), 313
output::ESP8266PWMOutput::get_setup_priority
 (C++ function), 313
output::ESP8266PWMOutput::pin_ (C++ mem-
 ber), 314
output::ESP8266PWMOutput::set_pin (C++ func-
 tion), 313
output::ESP8266PWMOutput::setup (C++ func-
 tion), 313
output::ESP8266PWMOutput::write_state (C++
 function), 313
output::FloatOutput (C++ class), 315
output::FloatOutput::disable (C++ function), 315
output::FloatOutput::enable (C++ function), 315
output::FloatOutput::get_max_power (C++ func-
 tion), 315
output::FloatOutput::max_power_ (C++ member),
 316

output::FloatOutput::set_max_power (C++ function), 315
 output::FloatOutput::set_state_ (C++ function), 315
 output::FloatOutput::write_enabled (C++ function), 315
 output::FloatOutput::write_state (C++ function), 315
 output::GPIOBinaryOutputComponent (C++ class), 308
 output::GPIOBinaryOutputComponent::get_setup_priority (C++ function), 308
 output::GPIOBinaryOutputComponent::GPIOBinaryOutputComponent (C++ class), 308
 output::GPIOBinaryOutputComponent::pin_ (C++ member), 309
 output::GPIOBinaryOutputComponent::setup (C++ function), 308
 output::GPIOBinaryOutputComponent::write_enabled (C++ function), 309
 output::LEDCOutputComponent (C++ class), 309
 output::LEDCOutputComponent::bit_depth_ (C++ member), 310
 output::LEDCOutputComponent::channel_ (C++ member), 310
 output::LEDCOutputComponent::frequency_ (C++ member), 310
 output::LEDCOutputComponent::get_bit_depth (C++ function), 310
 output::LEDCOutputComponent::get_channel (C++ function), 310
 output::LEDCOutputComponent::get_frequency (C++ function), 310
 output::LEDCOutputComponent::get_pin (C++ function), 310
 output::LEDCOutputComponent::get_setup_priority (C++ function), 310
 output::LEDCOutputComponent::LEDCOutputComponent (C++ function), 309
 output::LEDCOutputComponent::pin_ (C++ member), 310
 output::LEDCOutputComponent::set_bit_depth (C++ function), 309
 output::LEDCOutputComponent::set_channel (C++ function), 309
 output::LEDCOutputComponent::set_frequency (C++ function), 309
 output::LEDCOutputComponent::set_pin (C++ function), 309
 output::LEDCOutputComponent::setup (C++ function), 310
 output::LEDCOutputComponent::write_state (C++ function), 310
 output::next_ledc_channel (C++ member), 310
 output::PCA9685OutputComponent (C++ class), 310
 output::PCA9685OutputComponent::Channel (C++ class), 312
 output::PCA9685OutputComponent::Channel::Channel (C++ function), 312
 output::PCA9685OutputComponent::Channel::channel_ (C++ member), 312
 output::PCA9685OutputComponent::Channel::parent_ (C++ member), 312
 output::PCA9685OutputComponent::Channel::write_state (C++ function), 312
 output::PCA9685OutputComponent::create_channel (C++ function), 311
 output::PCA9685OutputComponent::frequency_ (C++ member), 312
 output::PCA9685OutputComponent::get_frequency (C++ function), 311
 output::PCA9685OutputComponent::get_mode (C++ function), 311
 output::PCA9685OutputComponent::get_setup_priority (C++ function), 312
 output::PCA9685OutputComponent::loop (C++ function), 312
 output::PCA9685OutputComponent::max_channel_ (C++ member), 312
 output::PCA9685OutputComponent::min_channel_ (C++ member), 312
 output::PCA9685OutputComponent::mode_ (C++ member), 312
 output::PCA9685OutputComponent::PCA9685OutputComponent (C++ function), 311
 output::PCA9685OutputComponent::pwm_amounts_ (C++ member), 312
 output::PCA9685OutputComponent::set_channel_value (C++ function), 312
 output::PCA9685OutputComponent::set_frequency (C++ function), 311
 output::PCA9685OutputComponent::set_mode (C++ function), 311
 output::PCA9685OutputComponent::setup (C++ function), 311
 output::PCA9685OutputComponent::update_ (C++ member), 312

P

parse_json (C++ function), 218
 parse_on_off (C++ function), 220
 PCF8574_INPUT (C++ enumerator), 350, 353
 PCF8574_INPUT_PULLUP (C++ enumerator), 350, 353
 PCF8574_OUTPUT (C++ enumerator), 350, 353
 PCF8574GPIOMode (C++ type), 350, 353
 PollingComponent (C++ class), 203

PollingComponent::get_update_interval (C++ function), 203
PollingComponent::PollingComponent (C++ function), 203
PollingComponent::set_update_interval (C++ function), 203
PollingComponent::setup_ (C++ function), 203
PollingComponent::update (C++ function), 203
PollingComponent::update_interval_ (C++ member), 203
PowerSupplyComponent (C++ class), 231
PowerSupplyComponent::active_requests_ (C++ member), 232
PowerSupplyComponent::enable_time_ (C++ member), 232
PowerSupplyComponent::enabled_ (C++ member), 232
PowerSupplyComponent::get_enable_time (C++ function), 232
PowerSupplyComponent::get_keep_on_time (C++ function), 232
PowerSupplyComponent::get_setup_priority (C++ function), 232
PowerSupplyComponent::is_enabled (C++ function), 232
PowerSupplyComponent::keep_on_time_ (C++ member), 232
PowerSupplyComponent::pin_ (C++ member), 232
PowerSupplyComponent::PowerSupplyComponent (C++ function), 231
PowerSupplyComponent::request_high_power (C++ function), 232
PowerSupplyComponent::set_enable_time (C++ function), 232
PowerSupplyComponent::set_keep_on_time (C++ function), 231
PowerSupplyComponent::setup (C++ function), 232
PowerSupplyComponent::unrequest_high_power (C++ function), 232

R

random_double (C++ function), 219
random_float (C++ function), 219
random_uint32 (C++ function), 219
RangeCondition (C++ class), 243
RangeCondition::check (C++ function), 243
RangeCondition::max_ (C++ member), 244
RangeCondition::min_ (C++ member), 244
RangeCondition::RangeCondition (C++ function), 243
RangeCondition::set_max (C++ function), 244
RangeCondition::set_min (C++ function), 243
reboot (C++ function), 218
run_safe_shutdown_hooks (C++ function), 218
run_shutdown_hooks (C++ function), 218

S

safe_reboot (C++ function), 218
safe_shutdown_hooks (C++ member), 220
sanitize_hostname (C++ function), 218
sanitize_string_whitelist (C++ function), 219
semaphore_scan_end (C++ member), 355
sensor::ADCSensorComponent (C++ class), 248
sensor::ADCSensorComponent::accuracy_decimals (C++ function), 248
sensor::ADCSensorComponent::ADCSensorComponent (C++ function), 248
sensor::ADCSensorComponent::attenuation_ (C++ member), 249
sensor::ADCSensorComponent::get_attenuation (C++ function), 248
sensor::ADCSensorComponent::get_pin (C++ function), 248
sensor::ADCSensorComponent::get_setup_priority (C++ function), 248
sensor::ADCSensorComponent::icon (C++ function), 248
sensor::ADCSensorComponent::pin_ (C++ member), 249
sensor::ADCSensorComponent::set_attenuation (C++ function), 248
sensor::ADCSensorComponent::set_pin (C++ function), 248
sensor::ADCSensorComponent::setup (C++ function), 248
sensor::ADCSensorComponent::unique_id (C++ function), 248
sensor::ADCSensorComponent::unit_of_measurement (C++ function), 248
sensor::ADCSensorComponent::update (C++ function), 248
sensor::ADS1115Component (C++ class), 259
sensor::ADS1115Component::ADS1115Component (C++ function), 260
sensor::ADS1115Component::get_sensor (C++ function), 260
sensor::ADS1115Component::get_setup_priority (C++ function), 260
sensor::ADS1115Component::request_measurement_ (C++ function), 260
sensor::ADS1115Component::sensors_ (C++ member), 260
sensor::ADS1115Component::setup (C++ function), 260
sensor::ADS1115Sensor (C++ class), 260
sensor::ADS1115Sensor::ADS1115Sensor (C++ function), 260
sensor::ADS1115Sensor::gain_ (C++ member), 261

sensor::ADS1115Sensor::get_gain (C++ function), 261
 sensor::ADS1115Sensor::get_multiplexer (C++ function), 261
 sensor::ADS1115Sensor::multiplexer_ (C++ member), 261
 sensor::ADS1115Sensor::set_gain (C++ function), 261
 sensor::ADS1115Sensor::set_multiplexer (C++ function), 260
 sensor::ADS1115Sensor::update_interval (C++ function), 261
 sensor::ADS1115Sensor::update_interval_ (C++ member), 261
 sensor::BH1750_RESOLUTION_0P5_LX (C++ enumerator), 280
 sensor::BH1750_RESOLUTION_1P0_LX (C++ enumerator), 280
 sensor::BH1750_RESOLUTION_4P0_LX (C++ enumerator), 280
 sensor::BH1750Resolution (C++ type), 280
 sensor::BH1750Sensor (C++ class), 279
 sensor::BH1750Sensor::accuracy_decimals (C++ function), 280
 sensor::BH1750Sensor::BH1750Sensor (C++ function), 279
 sensor::BH1750Sensor::get_setup_priority (C++ function), 280
 sensor::BH1750Sensor::icon (C++ function), 280
 sensor::BH1750Sensor::read_data_ (C++ function), 280
 sensor::BH1750Sensor::resolution_ (C++ member), 280
 sensor::BH1750Sensor::set_resolution (C++ function), 279
 sensor::BH1750Sensor::setup (C++ function), 279
 sensor::BH1750Sensor::unit_of_measurement (C++ function), 280
 sensor::BH1750Sensor::update (C++ function), 279
 sensor::BME280_IIR_FILTER_16X (C++ enumerator), 271
 sensor::BME280_IIR_FILTER_2X (C++ enumerator), 271
 sensor::BME280_IIR_FILTER_4X (C++ enumerator), 271
 sensor::BME280_IIR_FILTER_8X (C++ enumerator), 271
 sensor::BME280_IIR_FILTER_OFF (C++ enumerator), 271
 sensor::BME280_OVERSAMPLING_16X (C++ enumerator), 271
 sensor::BME280_OVERSAMPLING_1X (C++ enumerator), 271
 sensor::BME280_OVERSAMPLING_2X (C++ enumerator), 271
 sensor::BME280_OVERSAMPLING_4X (C++ enumerator), 271
 sensor::BME280_OVERSAMPLING_8X (C++ enumerator), 271
 sensor::BME280_OVERSAMPLING_NONE (C++ enumerator), 271
 sensor::BME280CalibrationData (C++ class), 271
 sensor::BME280CalibrationData::h1 (C++ member), 272
 sensor::BME280CalibrationData::h2 (C++ member), 272
 sensor::BME280CalibrationData::h3 (C++ member), 272
 sensor::BME280CalibrationData::h4 (C++ member), 272
 sensor::BME280CalibrationData::h5 (C++ member), 272
 sensor::BME280CalibrationData::h6 (C++ member), 272
 sensor::BME280CalibrationData::p1 (C++ member), 271
 sensor::BME280CalibrationData::p2 (C++ member), 271
 sensor::BME280CalibrationData::p3 (C++ member), 271
 sensor::BME280CalibrationData::p4 (C++ member), 272
 sensor::BME280CalibrationData::p5 (C++ member), 272
 sensor::BME280CalibrationData::p6 (C++ member), 272
 sensor::BME280CalibrationData::p7 (C++ member), 272
 sensor::BME280CalibrationData::p8 (C++ member), 272
 sensor::BME280CalibrationData::p9 (C++ member), 272
 sensor::BME280CalibrationData::t1 (C++ member), 271
 sensor::BME280CalibrationData::t2 (C++ member), 271
 sensor::BME280CalibrationData::t3 (C++ member), 271
 sensor::BME280Component (C++ class), 269
 sensor::BME280Component::BME280Component (C++ function), 269
 sensor::BME280Component::calibration_ (C++ member), 270
 sensor::BME280Component::get_humidity_sensor (C++ function), 270
 sensor::BME280Component::get_pressure_sensor (C++ function), 270
 sensor::BME280Component::get_setup_priority

(C++ function), 270
sensor::BME280Component::get_temperature_sensor sensor::BME680_IIR_FILTER_7X (C++ enumerator)
(C++ function), 270
sensor::BME280Component::humidity_oversampling_sensor::BME680_IIR_FILTER_OFF (C++ enumerator)
(C++ member), 270
sensor::BME280Component::humidity_sensor_ sensor::BME680_OVERSAMPLING_16X (C++
(C++ member), 271 enumerator), 275
sensor::BME280Component::iir_filter_ sensor::BME680_OVERSAMPLING_1X (C++
(C++ member), 271 enumerator), 274
sensor::BME280Component::pressure_oversampling_ sensor::BME680_OVERSAMPLING_2X (C++
(C++ member), 270 enumerator), 274
sensor::BME280Component::pressure_sensor_ sensor::BME680_OVERSAMPLING_4X (C++
(C++ member), 271 enumerator), 275
sensor::BME280Component::read_humidity_ sensor::BME680_OVERSAMPLING_8X (C++
(C++ function), 270 enumerator), 275
sensor::BME280Component::read_pressure_ sensor::BME680_OVERSAMPLING_NONE (C++
(C++ function), 270 enumerator), 274
sensor::BME280Component::read_s16_le sensor::BME680CalibrationData (C++ class), 275
(C++ function), 270
sensor::BME280Component::ambient_temperature
sensor::BME280Component::read_temperature_ (C++ member), 276
(C++ function), 270
sensor::BME280Component::read_u16_le (C++ member), 276
sensor::BME280Component::read_u8 (C++ function), 276
sensor::BME280Component::set_humidity_oversampling
sensor::BME280Component::set_iir_filter (C++ function), 276
sensor::BME280Component::set_pressure_oversampling
sensor::BME280Component::set_temperature_oversampling
sensor::BME280Component::setup (C++ function), 276
sensor::BME280Component::temperature_oversampling_ (C++ member), 276
sensor::BME280Component::temperature_sensor_ (C++ member), 276
sensor::BME280Component::update (C++ function), 276
sensor::BME280IIRFilter (C++ type), 276
sensor::BME280Oversampling (C++ type), 276
sensor::BME680_IIR_FILTER_127X (C++ enumerator), 275
sensor::BME680_IIR_FILTER_15X (C++ enumerator), 275
sensor::BME680_IIR_FILTER_1X (C++ enumerator), 275
sensor::BME680_IIR_FILTER_31X (C++ enumerator), 275
sensor::BME680_IIR_FILTER_3X (C++ enumerator), 275
sensor::BME680_IIR_FILTER_63X (C++ enumerator), 275

sensor::BME680CalibrationData::p7 (C++ member), 275
 sensor::BME680CalibrationData::p8 (C++ member), 275
 sensor::BME680CalibrationData::p9 (C++ member), 275
 sensor::BME680CalibrationData::range_sw_err (C++ member), 276
 sensor::BME680CalibrationData::res_heat_range (C++ member), 276
 sensor::BME680CalibrationData::res_heat_val (C++ member), 276
 sensor::BME680CalibrationData::t1 (C++ member), 275
 sensor::BME680CalibrationData::t2 (C++ member), 275
 sensor::BME680CalibrationData::t3 (C++ member), 275
 sensor::BME680CalibrationData::tfine (C++ member), 276
 sensor::BME680Component (C++ class), 272
 sensor::BME680Component::BME680Component (C++ function), 273
 sensor::BME680Component::calc_gas_resistance_ (C++ function), 274
 sensor::BME680Component::calc_heater_duration_ (C++ function), 274
 sensor::BME680Component::calc_heater_resistance_ (C++ function), 274
 sensor::BME680Component::calc_humidity_ (C++ function), 274
 sensor::BME680Component::calc_meas_duration_ (C++ function), 274
 sensor::BME680Component::calc_pressure_ (C++ function), 274
 sensor::BME680Component::calc_temperature_ (C++ function), 274
 sensor::BME680Component::calibration_ (C++ member), 274
 sensor::BME680Component::gas_resistance_sensor_ (C++ member), 274
 sensor::BME680Component::get_gas_resistance_sensor_ (C++ function), 273
 sensor::BME680Component::get_humidity_sensor_ (C++ function), 273
 sensor::BME680Component::get_pressure_sensor_ (C++ function), 273
 sensor::BME680Component::get_setup_priority_ (C++ function), 273
 sensor::BME680Component::get_temperature_sensor_ (C++ function), 273
 sensor::BME680Component::heater_duration_ (C++ member), 274
 sensor::BME680Component::heater_temperature_ (C++ member), 274
 sensor::BME680Component::humidity_oversampling_ (C++ member), 274
 sensor::BME680Component::humidity_sensor_ (C++ member), 274
 sensor::BME680Component::iir_filter_ (C++ member), 274
 sensor::BME680Component::pressure_oversampling_ (C++ member), 274
 sensor::BME680Component::pressure_sensor_ (C++ member), 274
 sensor::BME680Component::read_data_ (C++ function), 274
 sensor::BME680Component::set_heater (C++ function), 273
 sensor::BME680Component::set_humidity_oversampling_ (C++ function), 273
 sensor::BME680Component::set_iir_filter_ (C++ function), 273
 sensor::BME680Component::set_pressure_oversampling_ (C++ function), 273
 sensor::BME680Component::set_temperature_oversampling_ (C++ function), 273
 sensor::BME680Component::setup (C++ function), 273
 sensor::BME680Component::temperature_oversampling_ (C++ member), 274
 sensor::BME680Component::temperature_sensor_ (C++ member), 274
 sensor::BME680Component::update (C++ function), 273
 sensor::BME680IIRFilter (C++ type), 275
 sensor::BME680Oversampling (C++ type), 274
 sensor::BMP085Component (C++ class), 261
 sensor::BMP085Component::BMP085Component (C++ function), 261
 sensor::BMP085Component::calibration_ (C++ member), 262
 sensor::BMP085Component::CalibrationData (C++ class), 262
 sensor::BMP085Component::CalibrationData::ac1 (C++ member), 262
 sensor::BMP085Component::CalibrationData::ac2 (C++ member), 262
 sensor::BMP085Component::CalibrationData::ac3 (C++ member), 262
 sensor::BMP085Component::CalibrationData::ac4 (C++ member), 262
 sensor::BMP085Component::CalibrationData::ac5 (C++ member), 262
 sensor::BMP085Component::CalibrationData::ac6 (C++ member), 262
 sensor::BMP085Component::CalibrationData::b1 (C++ member), 262

sensor::BMP085Component::CalibrationData::b2
 (C++ member), 262

sensor::BMP085Component::CalibrationData::b5
 (C++ member), 262

sensor::BMP085Component::CalibrationData::mb
 (C++ member), 262

sensor::BMP085Component::CalibrationData::mc
 (C++ member), 262

sensor::BMP085Component::CalibrationData::md
 (C++ member), 262

sensor::BMP085Component::get_pressure_sensor
 (C++ function), 262

sensor::BMP085Component::get_temperature_sensor
 (C++ function), 261

sensor::BMP085Component::pressure_ (C++ member), 262

sensor::BMP085Component::read_pressure_ (C++ function), 262

sensor::BMP085Component::read_temperature_ (C++ function), 262

sensor::BMP085Component::set_mode_ (C++ function), 262

sensor::BMP085Component::setup (C++ function), 262

sensor::BMP085Component::temperature_ (C++ member), 262

sensor::BMP085Component::update (C++ function), 262

sensor::DallasComponent (C++ class), 249

sensor::DallasComponent::DallasComponent (C++ function), 249

sensor::DallasComponent::get_one_wire (C++ function), 250

sensor::DallasComponent::get_sensor_by_address
 (C++ function), 249

sensor::DallasComponent::get_sensor_by_index
 (C++ function), 250

sensor::DallasComponent::get_setup_priority (C++ function), 250

sensor::DallasComponent::one_wire_ (C++ member), 250

sensor::DallasComponent::sensors_ (C++ member), 250

sensor::DallasComponent::set_one_wire (C++ function), 250

sensor::DallasComponent::setup (C++ function), 250

sensor::DallasComponent::update (C++ function), 250

sensor::DallasTemperatureSensor (C++ class), 250

sensor::DallasTemperatureSensor::address_ (C++ member), 251

sensor::DallasTemperatureSensor::address_name_ (C++ member), 251

sensor::DallasTemperatureSensor::check_scratch_pad_
 (C++ function), 251

sensor::DallasTemperatureSensor::DallasTemperatureSensor
 (C++ function), 250

sensor::DallasTemperatureSensor::get_address
 (C++ function), 251

sensor::DallasTemperatureSensor::get_address8
 (C++ function), 251

sensor::DallasTemperatureSensor::get_address_name
 (C++ function), 251

sensor::DallasTemperatureSensor::get_index (C++ function), 251

sensor::DallasTemperatureSensor::get_resolution
 (C++ function), 251

sensor::DallasTemperatureSensor::get_temp_c
 (C++ function), 251

sensor::DallasTemperatureSensor::index_ (C++ member), 251

sensor::DallasTemperatureSensor::millis_to_wait_for_conversion_ (C++ function), 251

sensor::DallasTemperatureSensor::read_scratch_pad_ (C++ function), 251

sensor::DallasTemperatureSensor::resolution_ (C++ member), 251

sensor::DallasTemperatureSensor::scratch_pad_ (C++ member), 251

sensor::DallasTemperatureSensor::set_address
 (C++ function), 251

sensor::DallasTemperatureSensor::set_index (C++ function), 251

sensor::DallasTemperatureSensor::set_resolution
 (C++ function), 251

sensor::DallasTemperatureSensor::setup_sensor_ (C++ function), 251

sensor::DallasTemperatureSensor::unique_id (C++ function), 251

sensor::DebounceFilter (C++ class), 295

sensor::DebounceFilter::DebounceFilter (C++ function), 295

sensor::DebounceFilter::new_value (C++ function), 295

sensor::DebounceFilter::time_period_ (C++ member), 295

sensor::DeltaFilter (C++ class), 295

sensor::DeltaFilter::DeltaFilter (C++ function), 295

sensor::DeltaFilter::last_value_ (C++ member), 296

sensor::DeltaFilter::min_delta_ (C++ member), 296

sensor::DeltaFilter::new_value (C++ function), 295

sensor::DHT12Component (C++ class), 253

sensor::DHT12Component::DHT12Component
 (C++ function), 253

sensor::DHT12Component::get_humidity_sensor
 (C++ function), 254

sensor::DHT12Component::get_setup_priority

(C++ function), 253
sensor::DHT12Component::get_temperature_sensor (C++ function), 254
sensor::DHT12Component::humidity_sensor (C++ member), 254
sensor::DHT12Component::read_data (C++ function), 254
sensor::DHT12Component::setup (C++ function), 253
sensor::DHT12Component::temperature_sensor (C++ member), 254
sensor::DHT12Component::update (C++ function), 254
sensor::DHTComponent (C++ class), 252
sensor::DHTComponent::DHTComponent (C++ function), 252
sensor::DHTComponent::get_humidity_sensor (C++ function), 252
sensor::DHTComponent::get_setup_priority (C++ function), 253
sensor::DHTComponent::get_temperature_sensor (C++ function), 252
sensor::DHTComponent::humidity_sensor (C++ member), 253
sensor::DHTComponent::model (C++ member), 253
sensor::DHTComponent::pin (C++ member), 253
sensor::DHTComponent::read_sensor (C++ function), 253
sensor::DHTComponent::read_sensor_safe (C++ function), 253
sensor::DHTComponent::set_dht_model (C++ function), 252
sensor::DHTComponent::setup (C++ function), 252
sensor::DHTComponent::temperature_sensor (C++ member), 253
sensor::DHTComponent::update (C++ function), 252
sensor::EmptyPollingParentSensor (C++ class), 288
sensor::EmptyPollingParentSensor::EmptyPollingParentSensor (C++ function), 288
sensor::EmptyPollingParentSensor::parent (C++ member), 288
sensor::EmptyPollingParentSensor::update_interval (C++ function), 288
sensor::EmptySensor (C++ class), 287
sensor::EmptySensor::accuracy_decimals (C++ function), 288
sensor::EmptySensor::EmptySensor (C++ function), 288
sensor::EmptySensor::icon (C++ function), 288
sensor::EmptySensor::unit_of_measurement (C++ function), 288
sensor::ExponentialMovingAverageFilter (C++ class), 290
sensor::ExponentialMovingAverageFilter::accuracy_average (C++ member), 291
sensor::ExponentialMovingAverageFilter::expected_interval (C++ function), 291
sensor::ExponentialMovingAverageFilter::ExponentialMovingAverage (C++ function), 291
sensor::ExponentialMovingAverageFilter::get_alpha (C++ function), 291
sensor::ExponentialMovingAverageFilter::get_send_every (C++ function), 291
sensor::ExponentialMovingAverageFilter::new_value (C++ function), 291
sensor::ExponentialMovingAverageFilter::send_at (C++ member), 291
sensor::ExponentialMovingAverageFilter::send_every (C++ member), 291
sensor::ExponentialMovingAverageFilter::set_alpha (C++ function), 291
sensor::ExponentialMovingAverageFilter::set_send_every (C++ function), 291
sensor::ExponentialMovingAverageFilter::value_average (C++ member), 291
sensor::Filter (C++ class), 289
sensor::Filter::~Filter (C++ function), 289
sensor::Filter::expected_interval (C++ function), 289
sensor::Filter::initialize (C++ function), 289
sensor::Filter::input (C++ function), 289
sensor::Filter::new_value (C++ function), 289
sensor::Filter::next (C++ member), 289
sensor::Filter::output (C++ member), 289
sensor::FilterOutNANFilter (C++ class), 293
sensor::FilterOutNANFilter::new_value (C++ function), 294
sensor::FilterOutValueFilter (C++ class), 293
sensor::FilterOutValueFilter::FilterOutValueFilter (C++ function), 293
sensor::FilterOutValueFilter::new_value (C++ function), 293
sensor::FilterOutValueFilter::value_to_filter_out (C++ member), 293
sensor::HDC1080Component (C++ class), 264
sensor::HDC1080Component::get_humidity_sensor (C++ function), 264
sensor::HDC1080Component::get_temperature_sensor (C++ function), 264
sensor::HDC1080Component::HDC1080Component (C++ function), 264
sensor::HDC1080Component::humidity (C++ member), 264
sensor::HDC1080Component::setup (C++ function), 264
sensor::HDC1080Component::temperature (C++

member), 264
sensor::HDC1080Component::update (C++ function), 264
sensor::HeartbeatFilter (C++ class), 294
sensor::HeartbeatFilter::expected_interval (C++ function), 295
sensor::HeartbeatFilter::HeartbeatFilter (C++ function), 294
sensor::HeartbeatFilter::last_input_ (C++ member), 295
sensor::HeartbeatFilter::new_value (C++ function), 294
sensor::HeartbeatFilter::setup (C++ function), 294
sensor::HeartbeatFilter::time_period_ (C++ member), 295
sensor::HTU21DComponent (C++ class), 263
sensor::HTU21DComponent::get_humidity_sensor (C++ function), 263
sensor::HTU21DComponent::get_temperature_sensor (C++ function), 263
sensor::HTU21DComponent::HTU21DComponent (C++ function), 263
sensor::HTU21DComponent::humidity_ (C++ member), 263
sensor::HTU21DComponent::setup (C++ function), 263
sensor::HTU21DComponent::temperature_ (C++ member), 263
sensor::HTU21DComponent::update (C++ function), 263
sensor::ICON_BRIEFCASE_DOWNLOAD (C++ member), 288
sensor::ICON_EMPTY (C++ member), 288
sensor::ICON_FLASH (C++ member), 288
sensor::ICON_GAUGE (C++ member), 288
sensor::ICON_SCREEN_ROTATION (C++ member), 288
sensor::ICON_WATER_PERCENT (C++ member), 288
sensor::LambdaFilter (C++ class), 291
sensor::LambdaFilter::get_lambda_filter (C++ function), 292
sensor::LambdaFilter::lambda_filter_ (C++ member), 292
sensor::LambdaFilter::LambdaFilter (C++ function), 292
sensor::LambdaFilter::new_value (C++ function), 292
sensor::LambdaFilter::set_lambda_filter (C++ function), 292
sensor::MAX6675Sensor (C++ class), 280
sensor::MAX6675Sensor::accuracy_decimals (C++ function), 281
sensor::MAX6675Sensor::clock_ (C++ member), 281
sensor::MAX6675Sensor::cs_ (C++ member), 281
sensor::MAX6675Sensor::get_setup_priority (C++ function), 281
sensor::MAX6675Sensor::icon (C++ function), 281
sensor::MAX6675Sensor::MAX6675Sensor (C++ function), 281
sensor::MAX6675Sensor::miso_ (C++ member), 281
sensor::MAX6675Sensor::read_data_ (C++ function), 281
sensor::MAX6675Sensor::read_spi_ (C++ function), 281
sensor::MAX6675Sensor::setup (C++ function), 281
sensor::MAX6675Sensor::unit_of_measurement (C++ function), 281
sensor::MAX6675Sensor::update (C++ function), 281
sensor::MPU6050Component (C++ class), 265
sensor::MPU6050Component::accel_x_sensor_ (C++ member), 266
sensor::MPU6050Component::accel_y_sensor_ (C++ member), 266
sensor::MPU6050Component::accel_z_sensor_ (C++ member), 266
sensor::MPU6050Component::get_setup_priority (C++ function), 265
sensor::MPU6050Component::gyro_x_sensor_ (C++ member), 266
sensor::MPU6050Component::gyro_y_sensor_ (C++ member), 266
sensor::MPU6050Component::gyro_z_sensor_ (C++ member), 266
sensor::MPU6050Component::make_accel_x_sensor (C++ function), 265
sensor::MPU6050Component::make_accel_y_sensor (C++ function), 265
sensor::MPU6050Component::make_accel_z_sensor (C++ function), 266
sensor::MPU6050Component::make_gyro_x_sensor (C++ function), 266
sensor::MPU6050Component::make_gyro_y_sensor (C++ function), 266
sensor::MPU6050Component::make_gyro_z_sensor (C++ function), 266
sensor::MPU6050Component::make_temperature_sensor (C++ function), 266
sensor::MPU6050Component::setup (C++ function), 265
sensor::MPU6050Component::temperature_sensor_ (C++ member), 266
sensor::MPU6050Component::update (C++ function), 265
sensor::MQTTSensorComponent (C++ class), 297
sensor::MQTTSensorComponent::component_type

(C++ function), 298
 sensor::MQTTSensorComponent::disable_expire_aftersensor::PulseCounterSensorComponent::get_pcnt_unit
 (C++ function), 297
 sensor::MQTTSensorComponent::expire_after
 (C++ member), 298
 sensor::MQTTSensorComponent::friendly_name
 (C++ function), 298
 sensor::MQTTSensorComponent::get_expire_after
 (C++ function), 297
 sensor::MQTTSensorComponent::MQTTSensorComponent
 (C++ function), 297
 sensor::MQTTSensorComponent::send_discovery
 (C++ function), 297
 sensor::MQTTSensorComponent::sensor_ (C++
 member), 298
 sensor::MQTTSensorComponent::set_expire_after
 (C++ function), 297
 sensor::MQTTSensorComponent::setup (C++ func-
 tion), 297
 sensor::MultiplyFilter (C++ class), 292
 sensor::MultiplyFilter::multiplier_ (C++ member),
 293
 sensor::MultiplyFilter::MultiplyFilter (C++ func-
 tion), 293
 sensor::MultiplyFilter::new_value (C++ function),
 293
 sensor::OffsetFilter (C++ class), 292
 sensor::OffsetFilter::new_value (C++ function), 292
 sensor::OffsetFilter::offset_ (C++ member), 292
 sensor::OffsetFilter::OffsetFilter (C++ function), 292
 sensor::OrFilter (C++ class), 296
 sensor::OrFilter::~OrFilter (C++ function), 296
 sensor::OrFilter::expected_interval (C++ function),
 296
 sensor::OrFilter::filters_ (C++ member), 296
 sensor::OrFilter::initialize (C++ function), 296
 sensor::OrFilter::new_value (C++ function), 296
 sensor::OrFilter::OrFilter (C++ function), 296
 sensor::PollingSensorComponent (C++ class), 287
 sensor::PollingSensorComponent::PollingSensorCompon-
 (C++ function), 287
 sensor::PollingSensorComponent::update_interval
 (C++ function), 287
 sensor::PulseCounterSensorComponent (C++ class),
 254
 sensor::PulseCounterSensorComponent::accuracy_decimals
 (C++ function), 256
 sensor::PulseCounterSensorComponent::falling_edge_mode_ (C++ mem-
 ber), 256
 sensor::PulseCounterSensorComponent::filter_
 (C++ member), 256
 sensor::PulseCounterSensorComponent::get_falling_edge_mode
 (C++ function), 256
 sensor::PulseCounterSensorComponent::get_filter
 (C++ function), 255
 sensor::PulseCounterSensorComponent::get_pcnt_unit
 (C++ function), 256
 sensor::PulseCounterSensorComponent::get_pin
 (C++ function), 256
 sensor::PulseCounterSensorComponent::get_pull_mode
 (C++ function), 255
 sensor::PulseCounterSensorComponent::get_rising_edge_mode
 (C++ function), 256
 sensor::PulseCounterSensorComponent::get_setup_priority
 (C++ function), 256
 sensor::PulseCounterSensorComponent::icon (C++
 function), 256
 sensor::PulseCounterSensorComponent::last_value_
 (C++ member), 256
 sensor::PulseCounterSensorComponent::pcnt_unit_
 (C++ member), 256
 sensor::PulseCounterSensorComponent::pin_ (C++
 member), 256
 sensor::PulseCounterSensorComponent::pull_mode_
 (C++ member), 256
 sensor::PulseCounterSensorComponent::PulseCounterSensorCompon-
 (C++ function), 255
 sensor::PulseCounterSensorComponent::rising_edge_mode_
 (C++ member), 256
 sensor::PulseCounterSensorComponent::set_edge_mode
 (C++ function), 255
 sensor::PulseCounterSensorComponent::set_filter
 (C++ function), 255
 sensor::PulseCounterSensorComponent::set_pcnt_unit
 (C++ function), 255
 sensor::PulseCounterSensorComponent::set_pin
 (C++ function), 255
 sensor::PulseCounterSensorComponent::set_pull_mode
 (C++ function), 255
 sensor::PulseCounterSensorComponent::setup (C++
 function), 256
 sensor::PulseCounterSensorComponent::unit_of_measurement
 (C++ function), 256
 sensor::PulseCounterSensorComponent::update
 (C++ function), 256
 sensor::RotaryEncoderSensor (C++ class), 282
 sensor::RotaryEncoderSensor::accuracy_decimals
 (C++ function), 282
 sensor::RotaryEncoderSensor::counter_ (C++ mem-
 ber), 283
 sensor::RotaryEncoderSensor::encoder_isr_ (C++
 function), 283
 sensor::RotaryEncoderSensor::has_changed_ (C++
 member), 283
 sensor::RotaryEncoderSensor::icon (C++ function),
 282
 sensor::RotaryEncoderSensor::loop (C++ function),
 282

sensor::RotaryEncoderSensor::pin_a_ (C++ member), 283
sensor::RotaryEncoderSensor::pin_b_ (C++ member), 283
sensor::RotaryEncoderSensor::pin_i_ (C++ member), 283
sensor::RotaryEncoderSensor::process_state_machine_ (C++ function), 283
sensor::RotaryEncoderSensor::resolution_ (C++ member), 283
sensor::RotaryEncoderSensor::RotaryEncoderSensor (C++ function), 282
sensor::RotaryEncoderSensor::set_reset_pin (C++ function), 282
sensor::RotaryEncoderSensor::set_resolution (C++ function), 282
sensor::RotaryEncoderSensor::setup (C++ function), 282
sensor::RotaryEncoderSensor::state_ (C++ member), 283
sensor::RotaryEncoderSensor::unit_of_measurement (C++ function), 282
sensor::Sensor (C++ class), 284
sensor::Sensor::accuracy_decimals (C++ function), 286
sensor::Sensor::accuracy_decimals_ (C++ member), 287
sensor::Sensor::add_exponential_moving_average_filter (C++ function), 285
sensor::Sensor::add_filter (C++ function), 284
sensor::Sensor::add_filter_out_value_filter (C++ function), 285
sensor::Sensor::add_filters (C++ function), 284
sensor::Sensor::add_lambda_filter (C++ function), 285
sensor::Sensor::add_multiply_filter (C++ function), 285
sensor::Sensor::add_offset_filter (C++ function), 285
sensor::Sensor::add_on_raw_value_callback (C++ function), 286
sensor::Sensor::add_on_value_callback (C++ function), 286
sensor::Sensor::add_sliding_window_average_filter (C++ function), 285
sensor::Sensor::callback_ (C++ member), 287
sensor::Sensor::clear_filters (C++ function), 285
sensor::Sensor::filter_list_ (C++ member), 287
sensor::Sensor::get_accuracy_decimals (C++ function), 285
sensor::Sensor::get_icon (C++ function), 285
sensor::Sensor::get_raw_value (C++ function), 285
sensor::Sensor::get_unit_of_measurement (C++ function), 285
sensor::Sensor::get_value (C++ function), 285
sensor::Sensor::icon (C++ function), 286
sensor::Sensor::icon_ (C++ member), 287
sensor::Sensor::make_raw_value_trigger (C++ function), 286
sensor::Sensor::make_value_range_trigger (C++ function), 286
sensor::Sensor::make_value_trigger (C++ function), 286
sensor::Sensor::push_new_value (C++ function), 286
sensor::Sensor::raw_callback_ (C++ member), 287
sensor::Sensor::raw_value (C++ member), 287
sensor::Sensor::send_value_to_frontend (C++ function), 287
sensor::Sensor::Sensor (C++ function), 284
sensor::Sensor::set_accuracy_decimals (C++ function), 284
sensor::Sensor::set_filters (C++ function), 284
sensor::Sensor::set_icon (C++ function), 284
sensor::Sensor::set_unit_of_measurement (C++ function), 284
sensor::Sensor::unique_id (C++ function), 286
sensor::Sensor::unit_of_measurement (C++ function), 286
sensor::Sensor::unit_of_measurement_ (C++ member), 287
sensor::Sensor::update_interval (C++ function), 286
sensor::Sensor::value (C++ member), 287
sensor::SHT3XD_ACCURACY_HIGH (C++ enumerator), 277
sensor::SHT3XD_ACCURACY_LOW (C++ enumerator), 277
sensor::SHT3XD_ACCURACY_MEDIUM (C++ enumerator), 277
sensor::SHT3XDAccuracy (C++ type), 277
sensor::SHT3XDComponent (C++ class), 276
sensor::SHT3XDComponent::accuracy_ (C++ member), 277
sensor::SHT3XDComponent::get_humidity_sensor (C++ function), 277
sensor::SHT3XDComponent::get_setup_priority (C++ function), 277
sensor::SHT3XDComponent::get_temperature_sensor (C++ function), 277
sensor::SHT3XDComponent::humidity_sensor_ (C++ member), 277
sensor::SHT3XDComponent::read_data (C++ function), 277
sensor::SHT3XDComponent::set_accuracy (C++ function), 277
sensor::SHT3XDComponent::setup (C++ function), 277
sensor::SHT3XDComponent::SHT3XDComponent

sensor::SHT3XDComponent::temperature_sensor_ (C++ member),	277	sensor::ThrottleFilter::ThrottleFilter (C++ function),	294
sensor::SHT3XDComponent::update (C++ function),	277	sensor::TSL2561_GAIN_16X (C++ enumerator),	269
sensor::SHT3XDComponent::write_command (C++ function),	277	sensor::TSL2561_GAIN_1X (C++ enumerator),	269
sensor::SHT3XDHumiditySensor (C++ class),	278	sensor::TSL2561_INTEGRATION_101MS (C++ enumerator),	268
sensor::SHT3XDHumiditySensor::SHT3XDHumiditySensor (C++ function),	278	sensor::TSL2561_INTEGRATION_14MS (C++ enumerator),	268
sensor::SHT3XDHumiditySensor::unique_id (C++ function),	278	sensor::TSL2561_INTEGRATION_402MS (C++ enumerator),	269
sensor::SHT3XDHumiditySensor::unique_id_ (C++ member),	278	sensor::TSL2561Gain (C++ type),	269
sensor::SHT3XDTemperatureSensor (C++ class),	277	sensor::TSL2561IntegrationTime (C++ type),	268
sensor::SHT3XDTemperatureSensor::SHT3XDTemperatureSensor (C++ function),	278	sensor::TSL2561Sensor (C++ class),	267
sensor::SHT3XDTemperatureSensor::unique_id (C++ function),	278	sensor::TSL2561Sensor::accuracy_decimals (C++ function),	268
sensor::SHT3XDTemperatureSensor::unique_id_ (C++ member),	278	sensor::TSL2561Sensor::calculate_lx_ (C++ function),	268
sensor::SlidingWindowMovingAverageFilter (C++ class),	290	sensor::TSL2561Sensor::gain_ (C++ member),	268
sensor::SlidingWindowMovingAverageFilter::expected_value (C++ function),	290	sensor::TSL2561Sensor::get_integration_time_ms_ (C++ function),	268
sensor::SlidingWindowMovingAverageFilter::get_send_size		sensor::TSL2561Sensor::get_setup_priority (C++ interval function),	268
sensor::SlidingWindowMovingAverageFilter::get_window_size		sensor::TSL2561Sensor::icon (C++ function),	268
sensor::SlidingWindowMovingAverageFilter::new_value (C++ function),	290	sensor::TSL2561Sensor::integration_time_ (C++ member),	268
sensor::SlidingWindowMovingAverageFilter::send_at_sensor		sensor::TSL2561Sensor::package_cs_ (C++ member),	268
sensor::SlidingWindowMovingAverageFilter::send_every		sensor::TSL2561Sensor::read_data_ (C++ function),	268
sensor::SlidingWindowMovingAverageFilter::set_send_size		sensor::TSL2561Sensor::set_gain_ (C++ function),	267
sensor::SlidingWindowMovingAverageFilter::set_window_size		sensor::TSL2561Sensor::set_integration_time (C++ function),	267
sensor::SlidingWindowMovingAverageFilter::SlidingWindowMovingAverageFilter		sensor::TSL2561Sensor::set_is_cs_package (C++ function),	267
sensor::SlidingWindowMovingAverageFilter::value_average_ (C++ member),	290	sensor::TSL2561Sensor::setup (C++ function),	267
sensor::TemplateSensor (C++ class),	283	sensor::TSL2561Sensor::tsl2561_read_byte (C++ function),	267
sensor::TemplateSensor::TemplateSensor (C++ function),	283	sensor::TSL2561Sensor::tsl2561_read_uint (C++ function),	268
sensor::TemplateSensor::update (C++ function),	283	sensor::TSL2561Sensor::tsl2561_write_byte (C++ function),	268
sensor::ThrottleFilter (C++ class),	294	sensor::TSL2561Sensor::unit_of_measurement (C++ function),	268
sensor::ThrottleFilter::last_input_ (C++ member),	294	sensor::TSL2561Sensor::update (C++ function),	267
sensor::ThrottleFilter::min_time_between_inputs_ (C++ member),	294	sensor::UltrasonicSensorComponent (C++ class),	257
sensor::ThrottleFilter::new_value (C++ function),		sensor::UltrasonicSensorComponent::accuracy_decimals (C++ function),	258
		sensor::UltrasonicSensorComponent::echo_pin_ (C++ member),	258

sensor::UltrasonicSensorComponent::get_pulse_time [setup_priority::MQTT_COMPONENT](#) (C++ member), 204
(C++ function), 258

sensor::UltrasonicSensorComponent::get_setup_priority [setup_priority::WIFI](#) (C++ member), 204
(C++ function), 258

sensor::UltrasonicSensorComponent::get_timeout_m (C++ function), 258

sensor::UltrasonicSensorComponent::get_timeout_us (C++ function), 258

sensor::UltrasonicSensorComponent::icon (C++ function), 258

sensor::UltrasonicSensorComponent::m_to_us (C++ function), 259

sensor::UltrasonicSensorComponent::pulse_time_us_ (C++ member), 258

sensor::UltrasonicSensorComponent::set_pulse_time_ (C++ function), 258

sensor::UltrasonicSensorComponent::set_timeout_m (C++ function), 258

sensor::UltrasonicSensorComponent::set_timeout_us (C++ function), 257

sensor::UltrasonicSensorComponent::setup (C++ function), 258

sensor::UltrasonicSensorComponent::timeout_us_ (C++ member), 258

sensor::UltrasonicSensorComponent::trigger_pin_ (C++ member), 258

sensor::UltrasonicSensorComponent::UltrasonicSensorComponent (C++ function), 257

sensor::UltrasonicSensorComponent::unit_of_measurement (C++ function), 258

sensor::UltrasonicSensorComponent::update (C++ function), 258

sensor::UltrasonicSensorComponent::us_to_m (C++ function), 259

sensor::UniqueFilter (C++ class), 296

sensor::UniqueFilter::last_value_ (C++ member), 297

sensor::UniqueFilter::new_value_ (C++ function), 297

sensor::UNIT_C (C++ member), 289

sensor::UNIT_DEGREES_PER_SECOND (C++ member), 289

sensor::UNIT_HPA (C++ member), 289

sensor::UNIT_M_PER_S_SQUARED (C++ member), 289

sensor::UNIT_PERCENT (C++ member), 289

sensor::UNIT_V (C++ member), 289

setup_priority (C++ type), 203

setup_priority::HARDWARE (C++ member), 204

setup_priority::HARDWARE_LATE (C++ member), 204

setup_priority::LATE (C++ member), 204

setup_priority::MQTT_CLIENT (C++ member), 204

sensor::UltrasonicSensorComponent::get_pulse_time [setup_priority::MQTT_COMPONENT](#) (C++ member), 204
shutdown_hooks (C++ member), 220

ShutdownTrigger (C++ class), 245

ShutdownTrigger::ShutdownTrigger (C++ function), 245

SlidingWindowMovingAverage (C++ class), 220

SlidingWindowMovingAverage::calculate_average (C++ function), 220

SlidingWindowMovingAverage::get_max_size (C++ function), 220

SlidingWindowMovingAverage::max_size_ (C++ member), 221

SlidingWindowMovingAverage::next_value_ (C++ function), 220

SlidingWindowMovingAverage::queue_ (C++ member), 221

SlidingWindowMovingAverage::set_max_size (C++ function), 220

SlidingWindowMovingAverage::SlidingWindowMovingAverage (C++ function), 220

SlidingWindowMovingAverage::sum_ (C++ member), 221

StartupTrigger (C++ class), 244

StartupTrigger::get_setup_priority (C++ function), 245

StoringController (C++ class), 233

StoringController::binary_sensors_ (C++ member), 233

StoringController::covers_ (C++ member), 233

StoringController::fans_ (C++ member), 233

StoringController::lights_ (C++ member), 233

StoringController::register_binary_sensor (C++ function), 233

StoringController::register_cover (C++ function), 233

StoringController::register_fan (C++ function), 233

StoringController::register_light (C++ function), 233

StoringController::register_sensor (C++ function), 233

StoringController::register_switch (C++ function), 233

StoringController::sensors_ (C++ member), 233

StoringController::switches_ (C++ member), 233

swap (C++ function), 223

switch_::ir (C++ type), 338

switch_::ir::lg (C++ type), 339

switch_::ir::lg::BIT_HIGH_US (C++ member), 339

switch_::ir::lg::BIT_ONE_LOW_US (C++ member), 339

switch_::ir::lg::BIT_ZERO_LOW_US (C++ mem-

```

ber), 339
switch_::ir::lg::CARRIER_FREQUENCY_HZ
    (C++ member), 339
switch_::ir::lg::HEADER_HIGH_US (C++ member), 339
switch_::ir::lg::HEADER_LOW_US (C++ member), 339
switch_::ir::nec (C++ type), 339
switch_::ir::nec::BIT_HIGH_US (C++ member), 340
switch_::ir::nec::BIT_ONE_LOW_US (C++ member), 340
switch_::ir::nec::BIT_ZERO_LOW_US (C++ member), 340
switch_::ir::nec::CARRIER_FREQUENCY_HZ
    (C++ member), 340
switch_::ir::nec::HEADER_HIGH_US (C++ member), 340
switch_::ir::nec::HEADER_LOW_US (C++ member), 340
switch_::ir::panasonic (C++ type), 340
switch_::ir::panasonic::BIT_HIGH_US (C++ member), 340
switch_::ir::panasonic::BIT_ONE_LOW_US (C++ member), 340
switch_::ir::panasonic::BIT_ZERO_LOW_US
    (C++ member), 340
switch_::ir::panasonic::CARRIER_FREQUENCY_HZ
    (C++ member), 340
switch_::ir::panasonic::HEADER_HIGH_US (C++ member), 340
switch_::ir::panasonic::HEADER_LOW_US (C++ member), 340
switch_::ir::SendData (C++ class), 338
switch_::ir::SendData::add_item (C++ function),
    339
switch_::ir::SendData::carrier_frequency
    (C++ member), 339
switch_::ir::SendData::data (C++ member), 339
switch_::ir::SendData::from_lg (C++ function), 339
switch_::ir::SendData::from_nec (C++ function),
    339
switch_::ir::SendData::from_panasonic (C++ function), 339
switch_::ir::SendData::from_raw (C++ function),
    339
switch_::ir::SendData::from_sony (C++ function),
    339
switch_::ir::SendData::get_rmt_data (C++ function), 338
switch_::ir::SendData::mark (C++ function), 338
switch_::ir::SendData::repeat (C++ function), 339
switch_::ir::SendData::repeat_times (C++ member), 339
switch_::ir::SendData::repeat_wait (C++ member),
    339
switch_::ir::SendData::space (C++ function), 338
switch_::ir::SendData::total_length_ms (C++ function), 339
switch_::ir::sony (C++ type), 340
switch_::ir::sony::BIT_LOW_US (C++ member),
    340
switch_::ir::sony::BIT_ONE_HIGH_US
    (C++ member), 340
switch_::ir::sony::BIT_ZERO_HIGH_US
    (C++ member), 340
switch_::ir::sony::CARRIER_FREQUENCY_HZ
    (C++ member), 340
switch_::ir::sony::HEADER_HIGH_US
    (C++ member), 340
switch_::ir::sony::HEADER_LOW_US (C++ member), 340
switch_::IRTransmitterComponent (C++ class), 336
switch_::IRTransmitterComponent::calculate_on_off_time_
    (C++ function), 337
switch_::IRTransmitterComponent::carrier_duty_percent_
    (C++ member), 337
switch_::IRTransmitterComponent::channel_
    (C++ member), 337
switch_::IRTransmitterComponent::clock_divider_
    (C++ member), 337
switch_::IRTransmitterComponent::configure_rmt
    (C++ function), 337
switch_::IRTransmitterComponent::create_transmitter
    (C++ function), 336
switch_::IRTransmitterComponent::DataTransmitter
    (C++ class), 337
switch_::IRTransmitterComponent::DataTransmitter::DataTransmitt
    (C++ function), 338
switch_::IRTransmitterComponent::DataTransmitter::icon
    (C++ function), 338
switch_::IRTransmitterComponent::DataTransmitter::parent_
    (C++ member), 338
switch_::IRTransmitterComponent::DataTransmitter::send_data
    (C++ member), 338
switch_::IRTransmitterComponent::DataTransmitter::turn_off
    (C++ function), 338
switch_::IRTransmitterComponent::DataTransmitter::turn_on
    (C++ function), 338
switch_::IRTransmitterComponent::delay_microseconds_accurate_
    (C++ function), 337
switch_::IRTransmitterComponent::get_setup_priority
    (C++ function), 336
switch_::IRTransmitterComponent::get_ticks_for_10_us
    (C++ function), 337
switch_::IRTransmitterComponent::IRTransmitterComponent
    (C++ function), 336
switch_::IRTransmitterComponent::last_carrier_frequency_

```

switch_::IRTransmitterComponent::mark_ (C++ member), 337
switch_::IRTransmitterComponent::pin_ (C++ member), 337
switch_::IRTransmitterComponent::require_carrier_frequency (C++ function), 337
switch_::IRTransmitterComponent::send (C++ function), 336
switch_::IRTransmitterComponent::set_carrier_duty (C++ function), 336
switch_::IRTransmitterComponent::set_channel (C++ function), 337
switch_::IRTransmitterComponent::set_clock_divider (C++ function), 337
switch_::IRTransmitterComponent::setup (C++ function), 336
switch_::IRTransmitterComponent::space_ (C++ function), 337
switch_::MQTTSwitchComponent (C++ class), 345
switch_::MQTTSwitchComponent::component_type (C++ function), 345
switch_::MQTTSwitchComponent::MQTTSwitchComponent (C++ function), 345
switch_::MQTTSwitchComponent::send_discovery (C++ function), 345
switch_::MQTTSwitchComponent::setup (C++ function), 345
switch_::MQTTSwitchComponent::switch_ (C++ member), 346
switch_::MQTTSwitchComponent::turn_off (C++ function), 345
switch_::MQTTSwitchComponent::turn_on (C++ function), 345
switch_::next_rmt_channel (C++ member), 338
switch_::RestartSwitch (C++ class), 341
switch_::RestartSwitch::icon (C++ function), 341
switch_::RestartSwitch::RestartSwitch (C++ function), 341
switch_::RestartSwitch::turn_off (C++ function), 341
switch_::RestartSwitch::turn_on (C++ function), 341
switch_::ShutdownSwitch (C++ class), 341
switch_::ShutdownSwitch::icon (C++ function), 342
switch_::ShutdownSwitch::ShutdownSwitch (C++ function), 341
switch_::ShutdownSwitch::turn_off (C++ function), 342
switch_::ShutdownSwitch::turn_on (C++ function), 341
switch_::SimpleSwitch (C++ class), 344
switch_::SimpleSwitch::output_ (C++ member), 345
switch_::SimpleSwitch::SimpleSwitch (C++ function), 345
switch_::SimpleSwitch::turn_off (C++ function), 345
switch_::SimpleSwitch::turn_on (C++ function), 345
switch_::Switch (C++ class), 343
switch_::Switch::get_icon (C++ function), 344
switch_::Switch::get_setup_priority (C++ function), 343
switch_::Switch::icon (C++ function), 344
switch_::Switch::icon_ (C++ member), 344
switch_::Switch::make_toggle_action (C++ function), 344
switch_::Switch::make_turn_off_action (C++ function), 344
switch_::Switch::make_turn_on_action (C++ function), 344
switch_::Switch::optimistic (C++ function), 344
switch_::Switch::publish_state (C++ function), 343
switch_::Switch::set_icon (C++ function), 344
switch_::Switch::setup_ (C++ function), 343
switch_::Switch::Switch (C++ function), 343
switch_::Switch::turn_off (C++ function), 344
switch_::Switch::turn_on (C++ function), 344
switch_::Switch::write_state (C++ function), 344
switch_::TemplateSwitch (C++ class), 342
switch_::TemplateSwitch::add_turn_off_actions (C++ function), 342
switch_::TemplateSwitch::add_turn_on_actions (C++ function), 342
switch_::TemplateSwitch::loop (C++ function), 342
switch_::TemplateSwitch::optimistic (C++ function), 342
switch_::TemplateSwitch::optimistic_ (C++ member), 343
switch_::TemplateSwitch::set_optimistic (C++ function), 342
switch_::TemplateSwitch::set_state_lambda (C++ function), 342
switch_::TemplateSwitch::TemplateSwitch (C++ function), 342
switch_::TemplateSwitch::turn_off (C++ function), 343
switch_::TemplateSwitch::turn_off_action_ (C++ member), 343
switch_::TemplateSwitch::turn_on (C++ function), 342
switch_::TemplateSwitch::turn_on_action_ (C++ member), 343

T

TemplatableValue (C++ class), 221
TemplatableValue::f_ (C++ member), 222

TemplatableValue::has_value (C++ function), 222
 TemplatableValue::TemplatableValue (C++ function), 222
 TemplatableValue::value (C++ function), 222
 TemplatableValue::value_ (C++ member), 222
 to_lowercase_underscore (C++ function), 218
 Trigger (C++ class), 244
 Trigger::add_on_trigger_callback (C++ function), 244
 Trigger::on_trigger_ (C++ member), 244
 Trigger::trigger (C++ function), 244
 Trigger<NoArg> (C++ class), 244
 truncate_string (C++ function), 218

U

uint32_to_string (C++ function), 219
 uint64_to_string (C++ function), 219
 UrlMatch (C++ class), 235
 UrlMatch::domain (C++ member), 236
 UrlMatch::id (C++ member), 236
 UrlMatch::method (C++ member), 236
 UrlMatch::valid (C++ member), 236

V

VALUE (C++ enumerator), 222
 value_accuracy_to_string (C++ function), 219

W

WebServer (C++ class), 233
 WebServer::binary_sensor_json (C++ function), 235
 WebServer::canHandle (C++ function), 235
 WebServer::css_url_ (C++ member), 235
 WebServer::events_ (C++ member), 235
 WebServer::fan_json (C++ function), 235
 WebServer::get_setup_priority (C++ function), 234
 WebServer::handle_binary_sensor_request (C++ function), 235
 WebServer::handle_fan_request (C++ function), 235
 WebServer::handle_index_request (C++ function), 234
 WebServer::handle_light_request (C++ function), 235
 WebServer::handle_sensor_request (C++ function), 234
 WebServer::handle_switch_request (C++ function), 234
 WebServer::handleRequest (C++ function), 235
 WebServer::isRequestHandlerTrivial (C++ function), 235
 WebServer::js_url_ (C++ member), 235
 WebServer::light_json (C++ function), 235
 WebServer::port_ (C++ member), 235

WebServer::register_binary_sensor (C++ function), 235
 WebServer::register_fan (C++ function), 235
 WebServer::register_light (C++ function), 235
 WebServer::register_sensor (C++ function), 234
 WebServer::register_switch (C++ function), 234
 WebServer::sensor_json (C++ function), 234
 WebServer::server_ (C++ member), 235
 WebServer::set_css_url (C++ function), 234
 WebServer::set_js_url (C++ function), 234
 WebServer::set_port (C++ function), 234
 WebServer::setup (C++ function), 234
 WebServer::switch_json (C++ function), 234
 WebServer::WebServer (C++ function), 234
 WiFiComponent (C++ class), 204
 WiFiComponent::ap_channel_ (C++ member), 206
 WiFiComponent::ap_manual_ip_ (C++ member), 206
 WiFiComponent::ap_on_ (C++ member), 206
 WiFiComponent::ap_password_ (C++ member), 206
 WiFiComponent::ap_ssid_ (C++ member), 206
 WiFiComponent::get_hostname (C++ function), 205
 WiFiComponent::get_loop_priority (C++ function), 205
 WiFiComponent::get_setup_priority (C++ function), 205
 WiFiComponent::has_ap (C++ function), 205
 WiFiComponent::has_sta (C++ function), 205
 WiFiComponent::hostname_ (C++ member), 205
 WiFiComponent::loop (C++ function), 205
 WiFiComponent::on_wifi_event (C++ function), 206
 WiFiComponent::set_ap (C++ function), 204
 WiFiComponent::set_ap_manual_ip (C++ function), 205
 WiFiComponent::set_hostname (C++ function), 205
 WiFiComponent::set_sta (C++ function), 204
 WiFiComponent::set_sta_manual_ip (C++ function), 204
 WiFiComponent::setup (C++ function), 205
 WiFiComponent::setup_ap_config (C++ function), 205
 WiFiComponent::setup_sto_config (C++ function), 205
 WiFiComponent::sta_connected (C++ function), 205
 WiFiComponent::sta_manual_ip_ (C++ member), 206
 WiFiComponent::sta_on_ (C++ member), 205
 WiFiComponent::sta_password_ (C++ member), 205

WiFiComponent::sta_ssid_ (C++ member), [205](#)
WiFiComponent::wait_for_sta (C++ function), [205](#)
WiFiComponent::WiFiComponent (C++ function),
[204](#)