

Set Wide Net Documentation

Nov 11, 2026

Automated Film Props

CONTENTS:

1 Principles **1**
- 1.1 Architecture . . . 1
- 1.2 State . . . 2
- 1.3 Internal events . . . 2
- 1.4 Sequences . . . 2

2 Events **3**
- 2.1 Button . . . 3
- 2.2 Switch . . . 3
- 2.3 Encoder . . . 3
- 2.4 Counter . . . 3
- 2.5 Clock . . . 4
- 2.6 SceneTime . . . 5
- 2.7 DMX . . . 5
- 2.8 HBridge . . . 5
- 2.9 LinAct . . . 5
- 2.10 LED . . . 5
- 2.11 Backlight . . . 6
- 2.12 Pulse . . . 6
- 2.13 Meter . . . 7
- 2.14 Generic Gauge . . . 7
- 2.15 Indicator . . . 10
- 2.16 Layer . . . 10
- 2.17 Blocks . . . 10
- 2.18 Stepper . . . 11
- 2.19 Nidec . . . 11
- 2.20 Screebly (Event Identifier: Gribbley) . . . 11
- 2.21 Recall . . . 12
- 2.22 Servo . . . 12
- 2.23 UI . . . 12
- 2.24 Battery . . . 12
- 2.25 Heartbeat . . . 12
- 2.26 Compass . . . 13
- 2.27 ArtHorizon . . . 13
- 2.28 NOTFS . . . 13
- 2.29 Sequence . . . 13
- 2.30 Simulation . . . 13
- 2.31 RefreshRules . . . 14
- 2.32 URL . . . 14
- 2.33 VLC . . . 14
- 2.34 Special Events . . . 14
- 2.35 Local . . . 14

3 Services **17**

3.1 SceneTime . . . 17
3.2 Timed sequences . . . 18
3.3 EDL listener . . . 20
3.4 RED camera . . . 21
3.5 Weather . . . 22

4 Networking **25**

5 ESP32 and RP2040 Hardware Props Classes **27**
5.1 Overview . . . 27
5.2 Documentation Organization . . . 27

6 LORA Classes **53**
6.1 Overview . . . 53
6.2 LoRaTransceiver (SX127x) . . . 53
6.3 LoRaTransceiver (SX1262) . . . 53
6.4 RRRS_Page . . . 53

7 OSC Service **55**
7.1 Overview . . . 55
7.2 OSC Service Module . . . 55

8 Unreal Engine Classes **57**
8.1 Overview . . . 57
8.2 ueInstance . . . 57
8.3 UE5Extractor . . . 57
8.4 Camera . . . 57

9 WebSockets Configuration **59**
9.1 Overview . . . 59
9.2 Mapping (rules.json) . . . 59
9.3 Sequences (sequences.json) . . . 60
9.4 Server-Side Utility Classes . . . 60

10 WebSocket Element Classes **63**
10.1 Overview . . . 63
10.2 Element Classes . . . 63
10.3 Energy Display Classes . . . 70
10.4 Medical Display Classes . . . 74
10.5 Utility Functions . . . 79

11 Indices and tables **81**

Index **83**

CHAPTER

ONE

PRINCIPLES

Imagine a prop where the active component is a strip of NeoPixels (Addressable RGB LEDs). This prop will need a name for each class it has, perhaps something like **Pulse1**. Now it can listen for events that are sent to it, such as:

```
{"e": "Pulse1", "d": {
    "flame": 1,
    "color" : [255, 32, 4, 128]
    }
 }
```

This event will cause the prop to select the **flame** shader and set the base color. The four numbers are red, green, blue, and white. The event:

```
{"e": "Pulse1", "d": "stop"}
```

will stop the prop from running the shader. and fade the LEDs to [0,0,0,0] (off).

We could have multiple props with the same name, so the events above could drive all of them at once.

1.1 Architecture

There needs to be a SetWideNet WebSocket server that re-publishes events to all props that are both listening and subscribed to event types. Props are typically running on microcontrollers like ESP32/Pico/Arduino, or Web Browsers or with the context of Unreal Engine or Unity. The central server allows for stored sequences to be triggered and for some limited event mapping.

The Server has a http interface for configuration and management, allowing users to control, configure and monitor the props. There are outbound handlers for HTTP such that arbitrary HTTP requests can be sent to other equipment.

OSC event mapping is also supported, allowing for integration with other systems that use OSC (Open Sound Control) for communication.

Within each prop there are several queues: - **LocalQueue**: This is a queue for events that are internal to the prop. These events are used to trigger actions within the prop itself. They could then be queued to be sent to the SetWideNet WebSocket server. - **ReceiveQueue**: This is a queue for events that are sent to the SetWideNet WebSocket server. - **SendQueue**: This is a queue for events that are sent from the prop to the SetWideNet WebSocket server.

Within the prop, queues may deliver events to classes. Classes are used to encapsulate functionality, such as a NeoPixel strip, a button, or a switch. Each class can have its own event handlers and state.

1.2 State

As much as possible state should be held at the prop. This means that some classes need to be able to hold state, such as Switches. For example, a Switch can be toggled on or off, and its state can be recalled later.

1.3 Internal events

The simple case above deals with events coming from the SetWideNet WebSocket server. Lets imagine another prop, this time with a button as well as a NeoPixel strip.

When coding the prop's firmware we have the choice to listen to the button and publish to the WebSocket server, or we can trigger an internal event. It could be that the button press is only useful to the prop itself, to trigger a test mode for example. In this case we can public the event to the LocalQueue and then direct that to the NeoPixel strip.

So although the prop itself does not have a name, each component does. The NeoPixel strip could be called **Pulse1S** and the button could be called **Button1**.

1.4 Sequences

In addition to individual events, we can also define sequences of events that should be triggered in response to a single action. For example, pressing the button could trigger a sequence that first fades the NeoPixel strip to black, then plays a sound effect, and finally starts a new animation. Sequences can be defined in the prop's firmware and triggered just like individual events.

CHAPTER

TWO

EVENTS

Here are the events that are understood by the various classes in the SetWideNet framework.

2.1 Button

A button press is an atomic event that is normally just has 'clicked' as data:

```
{"e": "ButtonName", "d" : "clicked" }
```

2.2 Switch

A switch is a button that has two states, on and off. The event data is 0 or 1 to reflect the state, or it can be a special command like "recall" to trigger a recall of the last state:

```
{"e": "SwitchName", "d" : 0 }
{"e": "SwitchName", "d": "recall"}
```

2.3 Encoder

An encoder is a device that can be turned to change a value, The event data may include the current position and the change in position since the last event:

```
{"e": "EncoderName", "d" : {
   "position": 123,
   "delta": 1
}
```

2.4 Counter

Counter events control counting behavior and display. Event data contains command keys directly (not nested in a "command" object):

```
{"e": "CounterName", "d": {
    "start": true,
    "direction": 1,
    "steps": 10
}}
```

Counters are a staple of film work, you can get ours to 'emote' with commands including:

- **stop**: Stop the counter, hold current value.
- **start**: Start the counter in the specified direction from the current value.
- **direction (1/-1)**: Increment up (1) or down (-1).
- **steps (int)**: The amount to increment or decrement the counter by per step.
- **goto (int)**: Set the current value to this number.
- **clear**: Empty the display but keep current value.
- **rate (float)**: Set the count period in seconds (time between increments).
- **jitter (float)**: Introduce a specified range of randomness to the count time.
- **glitch (0/1/2)**: Create glitchiness. 0 = no glitch, 1 = brightness glitch, 2 = random display glitch.
- **hexmode (bool)**: Display the value in Hex (for TM1637 driven counters).
- **text (string)**: Attempt to show text on a seven segment display.
- **brightness (0-7)**: Set brightness levels on TM1637 driven counters.
- **random_start (int)**, **random_end (int)**: Display a random number between these values on each update.
- **stop_value_down (int)**: When decrementing, stop at this value.
- **stop_value_up (int)**: When incrementing, stop at this value.

2.5 Clock

A clock event manages time-related actions. The event data can include commands like “datetime”, “start”, “stop”, “reset”, “rate”, “position”, “nudge”, “sleep”, and “wake”:

```
{"e": "ClockName", "d": {
    "command": "start"
}}
```

commands include:

- **datetime**: Go to this time.
- **start**: Start the clock.
- **stop**: Stop the clock.
- **reset**: Stop the clock and assume both hands are at the zero position.
- **rate**: Multiplier of seconds (e.g., 10 runs 10x faster than real time).
- **position**: Go to this position, measured in stepper motor steps.
- **nudge**: Nudge the stepper motor for the defined hand and set this position as the new zero.
- **sleep**: Remove holding current from stepper motor.
- **wake**: Enable holding current to stepper motor.

2.6 SceneTime

SceneTime is the in-universe clock for the scene. The service that owns it, and the commands it accepts, are in *Services* (page 17).

2.7 DMX

DMX events control DMX devices and can include commands for fading and sequences:

```
{"e": "DMXName", "d": {
    "command": "fade",
    "channel": 1,
    "value": 255,
    "steps": 10
}}
```

commands include:

- **fade (time_int) channel, value, steps**: Fade a channel to value in steps over time.
- **sequence (list of sequence definitions)**: Run this list until another command arrives.

2.8 HBridge

HBridge events control motors with direction and speed, and can include commands like “direction”, “speed”, “brake”, “targetspeed”, “distance”, and “report”:

```
{"e": "HBridgeName", "d": {
    "command": "direction",
    "value": 1
}}
```

2.9 LinAct

LinAct events control linear actuators with commands like “stop”, “push”, and “pull”:

```
{"e": "LinActName", "d": "push"}
```

2.10 LED

LED events control LEDs and can handle sequences, duty cycle, glitch, rate, and stop commands:

```
{"e": "LEDName", "d": {
    "command": "duty",
    "value": 32768
}}
```

commands include:

- **duty**: 0-65353, set the duty cycle of the PWM.
- **glitch (duty,[rate])**: Glitch towards duty at rate.

- **rate**: Set the rate of PWM fading.
- **sequence (list of sequence definitions)**: Execute this sequence until another event arrives.
- **stop**: Stop any sequence or fade currently running.

2.11 Backlight

Backlight events are similar to LED events.

2.12 Pulse

Pulse events control color, speed, frequency, static color, fade rate, and multiple animation patterns:

```
{"e": "PulseName", "d": {
    "color": [255, 0, 0, 128]
}}
```

Color Format: Pulse supports both RGB (3 channels) and RGBW (4 channels) color formats. The format depends on the NeoPixel strip configuration.

Pulse events support:

- **stop**: Stop any sequence.
- **start**: Start any registered pattern or static color.
- **color (array)**: Register a color to be used in a pulsing pattern. Format: [R,G,B] or [R,G,B,W] depending on pixel type.
- **static (array)**: Set all pixels to this color. Format: [R,G,B] or [R,G,B,W].
- **factor (int/float)**: The amount of color to be used (multiplier).
- **steps (int)**: Number of steps in a pulse.
- **freq (int)**: The speed in Hz to index through steps. High values can exceed the capabilities of some CPUs.
- **fade (float)**: Set the fade rate (0.0-1.0). Previously called fadeout.
- **probability (value)**: The likelihood of something happening (for probabilistic effects).
- **depth (float)**: Depth parameter for flame effect (typically 0.0-1.0).
- **bg_color (array)**: Background color for rotate effect. Format: [R,G,B] or [R,G,B,W].
- **width (int)**: Width of the rotate effect foreground color.
- **set_width (int)**: Set the width of the rotate effect.
- **computer**: Start computer pattern mode (cycling through color array).
- **stream**: Start stream pattern mode. Requires **one_set**, **two_set**, and optionally **three_set** which are arrays of color arrays.
- **reverse**: Start reverse stream pattern (reverses direction). Also requires color set arrays.
- **flame**: Start flame effect pattern.
- **rotate_start**: Start rotate pattern effect. Can include **width** parameter.
- **fade_pulse**: Start fade pulse effect pattern.
- **hypospray (object)**: Special hypospray effect with **start_color**, **end_color** (arrays), and **duration** (milliseconds).

2.13 Meter

Meter events provide value, title, units, and can include commands like “calibrate”, “position”, and “damping”:

```
{"e": "MeterName", "d": {
    "command": "position",
    "value": 50
}}
```

A Generic Gauge takes its reading on `MeterN`. The device event is `GenericGauge`. See Generic Gauge.

2.14 Generic Gauge

A Generic Gauge is a scale, a needle, and a stored look. Three names are involved.

- `MeterN` is the reading. N is 1 to 16. The gauge stores which one it follows. The default is `Meter1`.
- `GenericGauge` is the device. Get, save, and reset are sent here, with a MAC.
- `GaugeGrpN` is the group. The default is `GaugeGrp1`. A group message has no MAC. Every gauge in that group applies it.

The gauge listens for its meter event, its group, `GenericGauge`, `Backlight1`, and `SceneTime`. A gauge with several dials listens for one meter event per dial. The first dial also follows `GenericGauge` and the group. Changing a listen name reconnects the socket.

2.14.1 Readings

A reading is a number in scale units. A bare number is the same reading. `position` is 0 to 100, a percentage of the scale, whatever `value_min` and `value_max` are.

```
{"e": "Meter1", "d": {"value": 42.5}}

{"e": "Meter1", "d": 42.5}

{"e": "Meter1", "d": {"position": 50}}
```

`payload_mode` chooses how those fields are read.

- **value** (the default): `value` is in scale units. If `value` is absent, `position` is read as scale units too.
- **position**: either field is 0 to 100.
- **auto**: `value` is scale units and `position` is 0 to 100.

A message that contains `value` or `position`, or that is a bare number, moves the needle. A message with neither is configuration.

2.14.2 Motion

Speed, damping, and jitter can ride on the same message. Left out, the needle keeps the motion already stored.

```
{"e": "Meter1", "d": {
    "value": 42.5,
    "speed": 30000,
    "damping": 0,
    "jitter": 0,
```

(continues on next page)

(continued from previous page)

```
    "jitter_rate": 200
}}
```

- **speed**: the same unit a stepper uses, from 40 to 30000. The needle moves at `speed / 64` percent of the scale per second, so a full sweep takes about `6400 / speed` seconds. 30000 is about a fifth of a second. 640 is about 10 seconds.
- **damping**: how the needle eases as it arrives, in those same steps. 0 is no ease. A gauge that snaps jumps to the new value. A gauge that holds a steady speed keeps that speed for the whole move. A larger damping slows the needle as it gets close.
- **jitter**: how far the needle is shaken. 0 is still.
- **jitter_rate**: the shake interval in milliseconds.

2.14.3 Scale, title, and zones

Send these on `MeterN` with the device MAC. `persist` true writes them. `GenericGauge` accepts the same fields when the MAC is present.

```
{"e": "Meter1", "d": {
    "MAC": "AABBCCDDEEFF",
    "title": "BUS VOLTS",
    "units": "V",
    "value_min": 0,
    "value_max": 100,
    "value_decimals": 1,
    "threshold1": 70,
    "threshold2": 90,
    "threshold1_mode": "over",
    "zone1_color": "NormalArc",
    "zone2_color": "Warning",
    "zone3_color": "Danger",
    "warning_text": "WARNING",
    "theme": "GenGauge1",
    "persist": true
}}
```

- **title**, **units**: the captions. Defaults are `Gauge` and empty.
- **value_min**, **value_max**: the scale. Defaults are 0 and 100.
- **value_decimals**: digits after the point. Default 1.
- **threshold1**, **threshold2**: where the colours change. A threshold above `value_max` turns the higher zones off. `threshold2` is kept at or above `threshold1`.
- **threshold1_mode**: `over` means the first warning applies above `threshold1`. `under` means it applies below.
- **zone1_color**, **zone2_color**, **zone3_color**: theme slot names. Defaults are `NormalArc`, `Warning`, and `Danger`.
- **warning_text**, **warning_text2**: the captions for the first and second warning. Defaults are `WARNING` and empty.
- **indicator_override**: a colour such as `#6EB5FF` for the normal indicator only. The warning zones still use the theme slots.
- **theme**: `Default`, `GenGauge1`, `GenGauge2`, `GenGauge3`, or `Inverse`. An unknown name is stored as `GenGauge1`.

`warning_threshold` is the earlier name for `threshold2`. `warning_color` is the earlier name for `zone3_color`.

Below `threshold1` the arc uses zone 1. Between the thresholds it uses zone 2. Above `threshold2` it uses zone 3. In `over` mode the first caption shows in zone 2, and zone 3 uses `warning_text2`, or `warning_text` when that is empty. In `under` mode the first caption shows in zone 1.

A gauge with more than one dial names the fields per dial: `meter1_title`, `meter1_units`, `meter1_theme`, and the same for dial 2 and dial 3. `panel_theme` is the page. A single dial uses the names without a number.

2.14.4 Group

A group message sets the shared look. It has no MAC.

```
{"e": "GaugeGrp1", "d": {"theme": "GenGauge1"}}

{"e": "GaugeGrp1", "d": {
    "panel_theme": "GenGauge1",
    "meter1_theme": "GenGauge1",
    "meter2_theme": "GenGauge1"
}}
```

2.14.5 Configuration

`nvs_get`, `nvs_save`, and `nvs_reset` require a MAC and are sent on `GenericGauge`.

```
{"e": "GenericGauge", "d": {
    "MAC": "AABBCCDDEEFF",
    "nvs_get": true
}}
```

`nvs_get` replies with `nvs` set to `config` and the stored fields. When the prop reports them, the reply includes how many meters and buttons it has. `nvs_save` writes the current configuration. `nvs_reset` restores the defaults.

`show_mac` is sent on `GenericGauge` with no MAC. Every gauge shows or hides its MAC in place of the title.

```
{"e": "GenericGauge", "d": {"show_mac": 1}}
```

`listen_event` is the meter name a single dial follows. `listen_group` is `GaugeGrp1` through `GaugeGrp99`. A multi-dial gauge stores one name per dial, as `meterN_listen_event` or `gaugeN_listen`. The defaults are `Meter1`, `Meter2`, and `Meter3`. `persist` true, with a MAC, writes the fields in that message.

2.14.6 Backlight

`Backlight1` sets the brightness. `duty` is 0 to 100. `brightness` is the same number. A MAC limits it to one gauge. With no MAC, every gauge that is listening applies it. `rate`, when present, is the fade.

```
{"e": "Backlight1", "d": {"duty": 80}}

{"e": "Backlight1", "d": {
    "MAC": "AABBCCDDEEFF",
    "duty": 40,
    "rate": 5000
}}
```

2.14.7 Pages and buttons

A gauge that has more than one screen listens for `Main` and for `PageN`. `SHOW` loads that screen. The data may instead name another screen. Readings still arrive while that dial is off the screen.

```
{"e": "Page1", "d": "SHOW"}

{"e": "Main", "d": "SHOW"}
```

A button on the gauge sends `CLICK` on the event name stored for it. An empty name sends nothing.

```
{"e": "CueGo", "d": "CLICK"}
```

2.15 Indicator

Indicators are used in Virtual Props, they are SVG elements with text, subtext, and color:

```
{"e": "IndicatorName", "d": {
    "indtext": "ON",
    "legendtext": "Power",
    "bezel": "green"
}}
```

2.16 Layer

Layer events control visibility of layers in SVG graphics:

```
{"e": "LayerName", "d": {
    "layerName": "background",
    "visible": 1
}}
```

2.17 Blocks

Blocks events control block-based pattern displays on NeoPixel strips. Most commands require a **block_number** parameter:

```
{"e": "BlocksName", "d": {
    "block_number": 0,
    "block_define": true,
    "start": 0,
    "end": 10,
    "color1": [255, 0, 0, 0],
    "color2": [0, 0, 255, 0]
}}
```

Color Format: Blocks supports both RGB (3 channels) and RGBW (4 channels) color formats.

Blocks events support:

- **block_number (int)**: Required for most commands. Identifies which block (0-15 typically).
- **block_define**: Define a new block. Requires **block_number**, **start** (pixel start), **end** (pixel end). Optional: **color1**, **color2**, **alarm_color**, **mode**, **delay**, **reverse**.

- **start (int)**: Starting pixel index for block definition.
- **end (int)**: Ending pixel index for block definition.
- **color1 (array)**: Primary color for block. Format: [R,G,B] or [R,G,B,W].
- **color2 (array)**: Secondary color for block (used in transitions/meters). Format: [R,G,B] or [R,G,B,W].
- **alarm_color (array)**: Color to show when alarm threshold is reached. Format: [R,G,B] or [R,G,B,W].
- **delay (int)**: Delay in milliseconds for meter animations.
- **reverse (bool)**: Reverse the direction of meter animations.
- **block_color (array)**: Set the block to a specific color. Format: [R,G,B] or [R,G,B,W].
- **block_off**: Turn off the block (set to black).
- **block_fade**: Fade the block.
- **block_meter (int)**: Display a meter (0-100) on the block. Optional: **alarm_threshold (int)**, **animate (bool)**.
- **string_clear**: Clear the entire NeoPixel string (all blocks).

2.18 Stepper

Stepper events control stepper motors. Event data contains command keys directly:

```
{"e": "StepperName", "d": {
    "target": 1000,
    "direction": 1
}}
```

2.19 Nidec

Nidec events are for stepper motor and encoder combinations.

2.20 Screebly (Event Identifier: Gribbley)

Screebly events control Screebly devices. Note that in SetWideNet event calls, the event name uses "Gribbley" (e.g., "Gribbley1", "Gribbley2") as the identifier, while "Screebly" is the actual product name.

These events instruct a Screebly device to go to a particular URL or control UI element visibility. Within the Web Browser, the Screebly device name is set by the SWNId cookie.

```
{"e": "Gribbley1", "d": {
    "url": "http://example.com"
}}
```

2.21 Recall

Recall events cause a prop to re-issue events that have state. See Switches.

2.22 Servo

Servo events set position and speed:

```
{"e": "ServoName", "d": {
    "position": 90
}}
```

2.23 UI

UI events control HTML div visibility or DOM properties. UI elements are identified by their ID, so a UI section might look like this:

```
<div id="UI3" style="display: none;">
    <!-- UI content that is initially hidden -->
</div>
```

and the event to show it would be:

```
{"e": "UI3", "d": "visible"}
```

2.24 Battery

Battery events provide state and thresholds for batteries in Prop Modules:

```
{"e": "BatteryName", "d": {
    "volts": 3.7,
    "lowvolts": 3.2,
    "maxvolts": 4.2,}
}}
```

2.25 Heartbeat

Heartbeat events provide network and battery data for a Prop Module. They are useful for both monitoring and identification. The event data includes:

```
{"e": "Heartbeat", "d": {
    "mac": "00:11:22:33:44:55",
    "ip": "192.168.1.100",
    "volts": 3.7,
    "lowvolts": 3.2,
    "maxvolts": 4.2,
    "subscribedevents": ["Button", "Switch"]
}}
```

Heartbeats provide:

- **mac**: MAC address.
- **ip**: IP address.
- **volts**: Battery volts or 'USB' if USB powered.
- **lowvolts**: What is low volts for this device.
- **maxvolts**: What would be fully charged for this device.
- **subscribedevents**: What event names this device will listen for.

2.26 Compass

Compass events provide data for SVG compass displays.

2.27 ArtHorizon

ArtHorizon events provide data for artificial horizon SVG UI elements.

2.28 NOTFS

NOTFS events provide data for notification objects. It was orginally part of a virtual prop, but we often use it as a diagnostic tool to show the events on prop control pages

```
{"e": "NOTFS", "d": {
    "message": "This is a notification",
    "type": "info"
}}
```

2.29 Sequence

A `Sequence` event fires one instant burst from `sequences.json`. The data is the name of that burst. A timed sequence is a different service, and it never sends this event. See *Services* (page 17).

```
{"e": "Sequence", "d": "Sequence1"}
```

2.30 Simulation

Simulation events pass data to a simulation script. These events will be custom to the simulation and can include any data structure needed by the script:

2.31 RefreshRules

RefreshRules events cause the server to reload event mapping rules. Its a command to the WebSocket server to refresh its rules:

```
{"e": "RefreshRules", "d": "true"}
```

2.32 URL

URL events ask the server to call a URL.

2.33 VLC

VLC events send commands to a VLC video player instance.

2.34 Special Events

Some events are special cases with unique behaviors or purposes. For example:

2.35 Local

The "local" event never leaves a Prop module and is typically used for network and calibration notifications.

```
{"e": "local", "d": {
    "type": "network",
    "status": "connected"
}}
```

2.35.1 HBridge Event Commands

HBridge events control motors:

- **direction 1/-1**: Forward/reverse.
- **speed (num)**: Frequency of PWM.
- **brake**: Attempt to hold position for motors that support this.
- **targetspeed**: Attempt to hold at this speed in pulses per second.
- **distance**: Attempt to move distance in steps (requires Hall effect device).
- **report**: Generate an event with current distance as payload.

2.35.2 LinAct Event Commands

Linear actuator commands:

- **stop**: Stop whatever motion is happening.
- **push**: Push.
- **pull**: Pull.

2.35.3 Meter Event Commands

Meter events can include:

- **calibrate**: Go past the zero position to set the zero position for steppers with endstops.
- **position (0..100)**: Go to a position that is 0-100 of full scale deflection.
- **damping (0-5000+)**: Apply damping as the needle approaches its target position.

2.35.4 Indicator Event Commands

Indicator events can set:

- **indtext**: The indicated text.
- **legendtext**: The legend text.
- **bezel**: The color of the bezel.

2.35.5 Layer Event Commands

Layer events for SVG graphics:

- **layerName 0/1**: The layer to be made visible or hidden.

2.35.6 Stepper Event Commands

Stepper motor commands:

- **target (int)**: The position to go to, in stepper steps. Can be combined with **direction**.
- **position (float)**: Go to position as percentage (0-100) of full scale deflection. See Meter above.
- **direction (1/-1)**: Direction for target movement (optional, used with target).
- **speed (int)**: Set stepper motor speed.
- **nudge (float)**: Nudge the stepper motor and set this position as the new zero.
- **calibrate**: Go past the zero position to set the zero position for steppers with endstops.
- **sleep**: Remove holding current from stepper motor.
- **wake**: Enable holding current to stepper motor.
- **damping (int)**: Apply damping as the needle approaches target position (0-5000+). Only available on X27-like motor classes.

2.35.7 Screebly Event Commands (Event Name: Gribbley)

Screebly events control Screebly devices. The event name uses "Gribbley" as the identifier (e.g., "Gribbley1", "Gribbley2"), while "Screebly" is the product name.

Screebly events can:

- **url**: The URL to navigate to.
- **UI**Num hidden/visible: Control visibility of UI elements identified by their ID.

2.35.8 URL

Ask the server to call a URL. Useful for working with products like ShellyIO or other HTTP based devices.

```
{"e": "URL", "d": {
    "url": "http://example.com/api/action"
}}
```

2.35.9 Other Events

- **Recall**: Any prop that implements recall will re-issue events that have state on receipt of a Recall event.
- **Nidec**: Stepper motor and encoder combined (legacy implementation).
- **VLC**: Send events to a VLC video player instance.

CHAPTER

THREE

SERVICES

These five processes sit beside the WebSocket hub. Each one connects as a client, listens for its own event, and publishes the result back onto the bus. The hub forwards. It does not keep the scene clock, the weather, the timed score, the EDL, or the camera.

Supervisor starts them from `WebSockets/Supervisor`. `scenetime` and `sequence` start with the hub. `edl_listener`, `weather`, and `red_rcp2_listener` stay off until they are started. A control page reloads from the file share. A service does not, until it is restarted:

```
sudo supervisorctl restart scenetime
sudo supervisorctl start sequence
sudo supervisorctl start edl_listener
sudo supervisorctl start weather
sudo supervisorctl start red_rcp2_listener
```

3.1 SceneTime

SceneTime is the time in the story. It is not the wall clock, and it is not a clock prop. `ClockN` is the prop. The service can drive a coupled clock from the scene time. The control page is `/swn/scenetime_control`. The stored settings are `WebSockets/scenetime_config.json`. The control page writes this file. A hand edit is read when the service restarts.

```
{
  "server": "ws://127.0.0.1:8000",
  "mode": "scene",
  "rate": 1,
  "timezone": "Europe/London",
  "start_time": "8 Sep 2030 18:59:00",
  "start_slot": 1,
  "start_slots": [
    {"label": "Scene1_A", "start_time": "8 Sep 2030 18:59:00"}
  ],
  "clock": "Clock1",
  "clocks": [
    {"name": "Clock1", "enabled": true, "offset_hours": 0}
  ]
}
```

`server` is the hub. `mode` is `scene` or `wall`. `rate` is the signed multiplier. `timezone` is used in wall mode. `start_time` is the origin of the selected slot. `start_slot` is 1–10, or 0 when none of the ten is selected. `start_slots` holds up to ten named origins. `clock` is the first enabled clock. `clocks` is Clock1 through Clock10, each with `enabled` and `offset_hours`.

Two modes:

- **scene**. The origin is an in-universe datetime, set by the operator or by a clapper. Stop and cut freeze it. Reset jumps back to the origin and stops. The running time is `origin + (wall elapsed) × rate`.
- **wall**. The clock is the live timezone time, always at rate 1. Start, stop, cut, and reset do not freeze it. A clapper still marks the EDL.

`rate` is a signed multiplier of wall seconds. `1` is realtime, `10` is ten times, `-1` runs backwards, `0` freezes. A coupled clock prop needs a positive rate and a direction of `1` or `-1`. The service splits the sign. Rate `0` sends that clock `stop`.

Ten named origins are stored as `start_slots`. `start_slot` 1–10 applies that origin and switches to scene mode. `start_slot` 0 means none of them is selected.

Commands:

```
{"e": "SceneTime", "d": "get"}

{"e": "SceneTime", "d": "start"}

{"e": "SceneTime", "d": "stop"}

{"e": "SceneTime", "d": "reset"}

{"e": "SceneTime", "d": {"start_time": "4 May 2330 16:30:00"}}

{"e": "SceneTime", "d": {"rate": 1.1}}

{"e": "SceneTime", "d": {"mode": "scene"}}

{"e": "SceneTime", "d": {"mode": "wall"}}

{"e": "SceneTime", "d": {"start_slot": 1}}

{"e": "Clapperboard", "d": "start"}

{"e": "Clapperboard", "d": "cut"}
```

`get` asks for the current snapshot. A prop sends it after the socket comes up, then ticks locally. `start` on a clapper is the take. `cut` ends it.

The snapshot carries `src` `SceneTimeService`, so the service ignores its own publish. `datetime` is the full scene time. `display` is the short clock string. `running`, `rate`, `mode`, `start_time`, `start_slot`, `start_slots`, and `timezone` travel with it. `clocks` lists the coupled `ClockN` props and an `offset_hours` for each.

The service sends a coupled clock its datetime on connect, on a clapper, and when the rate, the origin, the mode, or the coupling changes. It does not repeat that message every second.

3.2 Timed sequences

A timed sequence is a score. The service event is `SequenceService`. The file is `WebSockets/SequenceService/timed_sequences.json`. The control page is `/swn/sequence_control`. The event catalog the editor offers is `event_catalog.json` beside that file. `save` writes the document. A hand edit is read when the service restarts.

```
{
  "server": "ws://127.0.0.1:8000",
  "sequences": [
    {
      "name": "Door",
```

(continues on next page)

(continued from previous page)

```
      "clock": "scene",
      "starts": [{"type": "manual"}],
      "blocks": [
        {"timeline": [
          {"rel": 0, "do": {"e": "Clock1", "d": "start"}},
          {"rel": 1, "do": {"e": "Alert", "d": {"text": "Open"}}}
        ]}
      ]
    }
  ]
}
```

`server` is the hub. Each sequence has a `name`, a `clock` of `wall` or `scene`, at least one `starts` entry, and `blocks`. A start `type` is `manual` (the `go` command), `event` (an incoming `e` and optional `d`), or `scene` (an `at` time `HH:MM:SS`). A block is a `timeline` of rows. `rel` is seconds from the group start. `do` is the event that row sends. A block may instead be `delta`, `wait`, or `if`.

A hub `Sequence` event is not one of these. `{"e": "Sequence", "d": "Sequence1"}` fires one instant burst from `WebSockets/sequences.json`, and the hub consumes it. A timed sequence never sends that event, and that event never starts a timed sequence.

Time inside a score is relative to the start of a group, then a delta after the previous row. `wall` delays are wall seconds. `scene` delays follow the SceneTime rate and hold while that clock is stopped.

`WAIT FOR` releases when a matching event arrives. It does not look at the last stored value. A meter already past the threshold stays waiting until the next matching event. `IF` tests the last payload already seen.

A row can start another timed sequence. That sequence gets its own clock and its own run. The caller keeps going. A row cannot name the sequence it sits in, and a running sequence cannot start the one that started it. Starting a sequence that is already running restarts it. One run per name.

Commands:

```
{"e": "SequenceService", "d": "get"}

{"e": "SequenceService", "d": {"go": "Door"}}

{"e": "SequenceService", "d": {"stop": "Door"}}

{"e": "SequenceService", "d": {"arm": {"name": "Door", "index": 1, "armed": true}}}

{"e": "SequenceService", "d": {"save": {"sequences": []}}}
```

`go` starts the named score. `stop` ends that run. `arm` arms or disarms one row. `save` replaces the stored document. `get` returns it. The snapshot carries `src` `SequenceService`, the sequences, the run phase, row issues, and the catalog.

The hub delivers each fired event to every listener. A prop acts when `d.MAC` is its own address. Leave the MAC out and every listener acts.

3.3 EDL listener

The EDL listener watches the bus and writes an Edit Decision List for the take. Resolve imports that file as markers. The program is `edl_listener`. It does not start with the hub.

`WebSockets/edl_active.json` names the rule file to load. Changing it takes effect when the listener restarts.

```
{"config": "marble_edl_config.json"}
```

Two rule files ship beside it: `edl_list.json` and `marble_edl_config.json`. Each one names the hub, the output directory, and one or more tracks. A track has a frame rate, a `start` rule, a list of `log` rules, and an `end` rule.

```
{
  "server": "ws://127.0.0.1:8000",
  "output_dir": "setwidenetweb/media/edls",
  "edls": [
    {
      "edl_file": "example_take",
      "frame_rate": 24,
      "start": {"e": "Clapperboard", "d": "start"},
      "log": [
        {
          "match": {"e": "Marble1", "d": {"steps": "*"}},
          "event_name": "Marble steps {steps}",
          "marker_name": "Steps {steps}",
          "note": "Marble steps {steps}",
          "colour": "green",
          "duration": 1,
          "replay": true
        }
      ],
      "end": {"e": "Clapperboard", "d": "cut"}
    }
  ]
}
```

On each message the listener tests end, then log, then start. One message can close a take and open the next. The usual pair is a clapper:

```
{"e": "Clapperboard", "d": "start"}

{"e": "Clapperboard", "d": "cut"}
```

A rule matches on the event name. If it also has `d`, that value must equal the message, or each listed key must match. A value of `"*"` accepts any present value for that key.

A log rule names the marker. `event_name` is the default label. `marker_name` is the Resolve marker name. `note` is the Resolve note. `duration` is the marker length in frames. The default length is 1. `colour` is a Resolve colour. `replay` true also appends `{"t": seconds, "d": {...}}` to a `.jsonl` file beside the EDL. `t` is seconds after the clapper start.

`marker_name`, `note`, and `event_name` may use `{e}` and any key from `d` when `d` is an object. `Weapon1 trigger rounds={rounds}` fills `rounds` from that message.

Files are written under `setwidenetweb/media/edls` unless the track sets `output_dir`. The download page is `/swn/edl-downloads`. When the RED camera has reported a next take, a new session is named from that take. Otherwise the name is the track's `edl_file` stem plus the time.

3.4 RED camera

The RED listener speaks RCP2 to a DSMC3 camera (V-RAPTOR, V-RAPTOR XL, KOMODO, KOMODO-X) and bridges record start and stop onto the hub. The program is `red_rcp2_listener`. It does not start with the hub. The camera and the hub are both named in `WebSockets/red_rcp2_config.json`. A hand edit is read when the listener restarts.

```
{
  "id": "vraptor",
  "host": "192.168.1.50",
  "port": 9998,
  "client_name": "SetWideNet RED Listener",
  "client_version": "1.0.0",
  "reconnect_seconds": 5,
  "swn_server": "ws://127.0.0.1:8000",
  "swn_event": "RedCamera",
  "hub_record_only": true
}
```

`id` is the `camera_id` a command can name. `host` and `port` are the camera. RCP2 is `ws://<host>:<port>`, and `9998` is the camera's usual port. `client_name` and `client_version` are what the listener tells the camera. `reconnect_seconds` is the pause before another try, for the camera and for the hub. `swn_server` is the hub. `swn_event` is the event name, `RedCamera` unless this file changes it. `hub_record_only` true publishes only record start and record stop. False also publishes the other record-state changes.

The event name is `RedCamera`. The camera rolling publishes `record start`. The camera stopping publishes `record stop`. Hub rules turn those two into `Clapperboard start` and `cut`, which is what the EDL listener uses for the take.

Commands to the camera use a string, or an object with `do`. They do not use `cmd` or `record`. Those two keys are the ones the hub rules remap, so a command sent that way becomes a clapper instead of a camera command.

```
{"e": "RedCamera", "d": "start"}

{"e": "RedCamera", "d": "stop"}

{"e": "RedCamera", "d": "toggle"}

{"e": "RedCamera", "d": "status"}

{"e": "RedCamera", "d": "set_time"}

{"e": "RedCamera", "d": {"do": "start", "camera_id": "vraptor"}}
```

`start`, `stop`, and `toggle` set the camera record state. `status` asks for a snapshot. `set_time` sets the camera clock from the clock of the machine running the listener. A command with `camera_id` or `host` applies to that listener only. A command with neither applies to every listener.

The status snapshot has `status` true and a `cameras` list. Each row is one listener. The listener subscribes to the camera's record state and to the clip name and the media remaining.

3.5 Weather

Weather is the weather for the set, either the real weather at a place or a narrative the operator sets. It is not a prop. The HTML HUD and the e-paper display both read the same snapshot. The control page is `/swn/weather_control`. The program is `weather`. It does not start with the hub. The stored settings are `WebSockets/weather_config.json`. The control page writes this file, including the last snapshot. A hand edit is read when the service restarts.

```
{
  "server": "ws://127.0.0.1:8000",
  "mode": "real",
  "location": "Cardiff",
  "lat": 51.48,
  "lon": -3.18,
  "poll_s": 900,
  "provider": "open-meteo",
  "units": "C",
  "preset": 0,
  "presets": [
    {"label": "Clear", "sky": "CLEAR", "temp_c": 18, "warning": "none"}
  ]
}
```

`server` is the hub. `mode` is `real` or `scene`. `location`, `lat`, and `lon` are the place. In real mode the service polls `provider` every `poll_s` seconds. `units` is how a display draws the temperature. The numbers in the file stay metric. `preset` is the selected row of `presets`. The rest of the file is the last snapshot: temperature, wind, sky, warning, sunrise, and sunset.

Two modes:

- **real**. The service polls Open-Meteo for the stored latitude and longitude. There is no API key. The default period is 900 seconds. Switching to real fetches at once. A failed fetch keeps the last snapshot and sets `stale` true.
- **scene**. Presets and sliders. No network. Editing a preset, a sky token, or a slider switches to this mode.

The service is the only process that calls Open-Meteo. Props and browsers do not.

Commands:

```
{"e": "Weather", "d": "get"}

{"e": "Weather", "d": "refresh"}

{"e": "Weather", "d": {"mode": "real"}}

{"e": "Weather", "d": {"mode": "scene"}}

{"e": "Weather", "d": {"location": "Cardiff"}}

{"e": "Weather", "d": {"location": "Gotham", "geocode": false}}

{"e": "Weather", "d": {"preset": 11}}

{"e": "Weather", "d": {"sky": "HURRICANE", "warning": "red"}}

{"e": "Weather", "d": {"poll_s": 900}}

{"e": "Weather", "d": {"units": "C"}}
```

In real mode, `location` is looked up and the first match is kept, then fetched. In scene mode, `location` is a label and is not looked up. `geocode` true forces a lookup. `geocode` false stores the typed name even in real mode. `refresh` fetches again in real mode. In scene mode it republishes the snapshot.

The snapshot carries `src` `WeatherService`. The numbers stay metric: `temp_c`, `feels_like_c`, `humidity`, `pressure_hpa`, `wind_ms`, `wind_dir`, `precip_mm`, `cloud_pct`. `units` only changes how a display draws the temperature. `sky` is one of `CLEAR`, `CLOUD`, `OVERCAST`, `RAIN`, `DRIZZLE`, `WIND`, `FOG`, `STORM`, `HAIL`, `SNOW`, `FLOOD`, `HURRICANE`, `TORNADO`. `warning` is `none`, `amber`, or `red`. `display` is the one-line string, an integer temperature and the sky token. `sunrise` and `sunset` are clock times. `preset` and `presets` are the operator list. `updated` is when the snapshot was made.

The service publishes on connect, on `get`, when the operator changes something, and on each poll. It does not publish every second.

CHAPTER

FOUR

NETWORKING

Currently the network options are:

1. **Wifi**: This is the primary method of communication between props. Each prop can connect to a Wifi network, allowing them to communicate with each other and with a central controller.
2. **Ethernet**: When combined with POE this makes a neat option for 'control panel' style props, easy to mount on panels or walls.
3. **LoraWan**: We're using this as a software defined radio network for long range communication. This is useful for props that need to operate over large distances or in environments where Wifi is unreliable, such as water.

Each prop has a config.json file defines the Wifi network setting, along with the WebSockets server address and port. This allows props to connect to the network and communicate with each other.

Depending on your environment you may have other connections and/or an extended Wireless Access Point (AP) to cover larger areas.

1. **Virtual Production: In these productions there are often three networks in use:**
    - A network for media and assets
    - A network for motion capture and tracking
    - A network to drive the nodes that render the virtual production content

The Set Wide Net server may need extra interfaces to connect to these networks.

CHAPTER

FIVE

ESP32 AND RP2040 HARDWARE PROPS CLASSES

Hardware Interface Classes for SetWideNet Physical Props

5.1 Overview

The ESP32Panel classes are MicroPython modules designed to run on ESP32 and RP2040 microcontrollers. They provide abstractions for hardware components commonly used in film prop automation, including:

- Sensor interfaces for environmental and positional sensing
- Display drivers for various display technologies
- Motor and actuator control classes
- LED and lighting control
- Communication protocols and IoT device integration

All classes follow a consistent interface pattern: - Initialization via `__init__()` with hardware pin assignments - Command interface accepting JSON-formatted events via `command()` method - Event output via message queues in JSON format - Async execution via `run()` coroutine methods where applicable

5.2 Documentation Organization

The classes are organized into the following categories:

5.2.1 Sensor Classes

Classes for reading environmental and positional data

Battery

Battery voltage monitoring via ADC

Class Instantiation

```
from battery_classes import Battery
from machine import ADC

battery = Battery(
    name="Battery1",
    pin_number=35,
    message_queue=None,
    max_voltage=8.4,
    low_voltage=6.8,
    calibrate=87770.0,
    attenuation=ADC.ATTN_11DB
)
```

Parameters:

- **name** (str): Unique identifier for this battery monitor instance. Used in event names.
- **pin_number** (int): GPIO pin number connected to the battery voltage divider circuit (via ADC).
- **message_queue** (list, optional): Queue for sending battery status events. If None, events are not queued. Default: None
- **max_voltage** (float, optional): Maximum expected battery voltage (fully charged). Default: 8.4
- **low_voltage** (float, optional): Low voltage threshold for alarm condition. Default: 6.8
- **calibrate** (float, optional): Calibration factor to convert ADC raw value (in microvolts) to voltage. Formula: voltage = raw_uV / calibrate. Default: 87770.0
- **attenuation** (ADC constant, optional): ADC attenuation setting. Use ADC.ATTN_11DB for 0-3.3V, ADC.ATTN_6DB for 0-2V, etc. Default: ADC.ATTN_11DB

BMP280_I2C

BMP280 pressure and temperature sensor interface via I2C

Class Instantiation

```
from bmp280_classes import BMP280_I2C
from machine import I2C, Pin

i2c = I2C(0, scl=Pin(22), sda=Pin(21))
bmp280 = BMP280_I2C(
    i2c=i2c,
    name="BMP280_1",
    address=0x76,
    queue=None
)
```

Parameters:

- **i2c** (I2C): Initialized I2C bus instance for communication with the BMP280 sensor.
- **name** (str, optional): Unique identifier for this sensor instance. Used in event names. Default: “REMTech”
- **address** (int, optional): I2C device address. BMP280 typically uses 0x76 or 0x77. Default: 0x76
- **queue** (list, optional): Queue for sending sensor data events. If None, events are not queued. Default: None

The sensor is automatically configured for weather monitoring with high oversampling and appropriate filtering.

Distance

HC-SR04 ultrasonic distance sensor interface

Class Instantiation

```
from distance_classes import Distance
from hcsr04 import HCSR04

sensor = HCSR04(trigger_pin=12, echo_pin=14)
distance = Distance(
    name="Distance1",
    sensor=sensor,
    ws_queue=None,
    thresholds=[10, 30, 60, 150]
)
```

Parameters:

- **name** (str): Unique identifier for this distance sensor instance. Used in event names.
- **sensor** (HCSR04): Initialized HCSR04 ultrasonic sensor instance.
- **ws_queue** (list): WebSocket queue for sending distance events when thresholds are crossed. Can be None if events not needed.
- **thresholds** (list, optional): List of distance thresholds in cm. Events are sent when distance crosses these thresholds. Default: [10, 30, 60, 150]. Maximum range is approximately 2.5m.

HallEffectPlain

Hall effect sensor for magnetic field detection

Class Instantiation

```
from hall_effect_classes import HallEffectPlain

hall = HallEffectPlain(
    name="Hall1",
    queue=None,
    pin=25
)
```

Parameters:

- **name** (str): Unique identifier for this hall effect sensor instance. Used in event names.
- **queue** (list, optional): Queue for sending state change events. If None, events are not queued. Default: None
- **pin** (int, optional): GPIO pin number connected to the hall effect sensor. Pin is configured as input with pull-down. Default: None

State changes trigger IRQ handlers that send events when magnetic field is detected (state changes).

HX711

HX711 24-bit ADC for load cell/scale applications

Class Instantiation

```
from hx711_classes import HX711
from machine import Pin, SPI

clock = Pin(26, Pin.OUT)
data = Pin(25, Pin.IN)
spi = SPI(1, baudrate=1000000, polarity=0, phase=0)

hx711 = HX711(
    clock=clock,
    data=data,
    spi=spi,
    gain=128
)
```

Parameters:

- **clock** (Pin): GPIO pin configured as output for HX711 clock signal.
- **data** (Pin): GPIO pin configured as input for HX711 data signal.
- **spi** (SPI): Initialized SPI bus instance for communication with HX711.
- **gain** (int, optional): Gain setting. Valid values: 128, 64, or 32. Default: 128

Note: HX711 is typically used with the Scales wrapper class rather than directly.

Scales

Scales class for weight measurement using HX711

Class Instantiation

```
from hx711_classes import Scales, HX711
from machine import Pin, SPI

clock = Pin(26, Pin.OUT)
data = Pin(25, Pin.IN)
spi = SPI(1, baudrate=1000000, polarity=0, phase=0)
hx711 = HX711(clock, data, spi)

scales = Scales(
    name="Scales1",
    queue=None,
    hx711=hx711,
    scale=1.0,
    offset=0.0,
    factor=418,
    debug=False
)
```

Parameters:

- **name** (str): Unique identifier for this scales instance. Used in event names.
- **queue** (list): Queue for sending weight measurement events. Can be None if events not needed.
- **hx711** (HX711): Initialized HX711 instance for reading load cell data.
- **scale** (float, optional): Scale factor for converting raw HX711 values to weight units. Default: 1.0
- **offset** (float, optional): Offset value to subtract from readings (for tare/zero calibration). Default: 0.0
- **factor** (int, optional): Additional scaling factor used in weight calculation. Typical value for real scales. Default: 418
- **debug** (bool, optional): Enable debug output. Default: False

Weight calculation: weight = floor(((raw_value - offset) * scale) / factor)

TF_Luna

TF Luna LiDAR distance sensor interface via UART

Class Instantiation

```
from tf_luna_classes import TF_Luna
from machine import UART, Pin

uart = UART(1, tx=Pin(17), rx=Pin(16))
lidar = TF_Luna(
    name="Lidar1",
    uart=uart,
    queue=None,
    baudrate=115200
)
```

Parameters:

- **name** (str): Unique identifier for this LiDAR sensor instance. Used in event names.
- **uart** (UART): Initialized UART instance for communication with TF Luna sensor.
- **queue** (list, optional): Queue for sending distance measurement events. If None, events are not queued. Default: None
- **baudrate** (int, optional): UART baud rate. Default: 115200

The sensor automatically defines 5 distance zones (0-30, 31-50, 51-100, 101-250, 251-800 cm) for zone-based event reporting.

BLEBeacon

BLE beacon proximity detection and scanning

Class Instantiation

```
from ble_classes import BLEBeacon

beacon = BLEBeacon(
    name="BLE1",
    queue=None,
    threshold=-70,
    near=-60,
    enable=True,
    debug=False
)
```

Parameters:

- **name** (str): Unique identifier for this BLE scanner instance. Used in event names.
- **queue** (list, optional): Queue for sending beacon detection events. If None, events are not queued. Default: None
- **threshold** (int, optional): RSSI threshold in dBm. Only beacons with RSSI above this threshold are considered. Default: -70
- **near** (int, optional): RSSI threshold in dBm for "near" detection. Beacons above this value trigger near events. Default: -60
- **enable** (bool, optional): Enable scanning on initialization. Default: True
- **debug** (bool, optional): Enable debug output. Default: False

Beacons must be registered using the `register` command before proximity detection works.

5.2.2 Display Classes

Classes for driving various display technologies

Clock

Digital clock display using TM1637 4-digit 7-segment display

Class Instantiation

```
from tm_classes import Clock
import tm1637

display = tm1637.TM1637(clk=Pin(18), dio=Pin(19))
clock = Clock(
    display=display,
    name="Clock1",
    message_queue=None,
    message="Alarm",
    brightness=1,
    timer_id=2
)
```

Parameters:

- **display** (TM1637): Initialized TM1637 display instance.

- **name** (str): Unique identifier for this clock instance. Used in event names.
- **message_queue** (list, optional): Queue for sending clock events (alarms, ticks). If None, events are not queued. Default: None
- **message** (str, optional): Alarm message text. Default: "Alarm"
- **brightness** (int, optional): Display brightness level (0-7). Default: 1
- **timer_id** (int, optional): Timer ID for the periodic tick callback. Default: 2

Counter6

Six-digit counter display using TM1637 6-digit 7-segment display

Class Instantiation

```
from tm_classes import Counter6
import tm1637

display = tm1637.TM1637(clk=Pin(18), dio=Pin(19))
counter = Counter6(
    display=display,
    name="Counter1",
    message_queue=None,
    alarm_message="Alarm",
    brightness=1
)
```

Parameters:

- **display** (TM1637): Initialized TM1637 display instance (6-digit).
- **name** (str): Unique identifier for this counter instance. Used in event names.
- **message_queue** (list, optional): Queue for sending counter events. If None, events are not queued. Default: None
- **alarm_message** (str, optional): Alarm message text. Default: "Alarm"
- **brightness** (int, optional): Display brightness level (0-7). Default: 1

Note: The 6-digit display uses a non-standard grid mapping. Digits left-to-right are controlled by grids 3, 2, 1, 6, 5, 4 (not 1-6 in order).

Counter4

Four-digit counter display (inherits from Counter6)

Class Instantiation

```
from tm_classes import Counter4
import tm1637

display = tm1637.TM1637(clk=Pin(18), dio=Pin(19))
counter = Counter4(
    display=display,
    name="Counter1",
```

(continues on next page)

(continued from previous page)

```
    message_queue=None,
    alarm_message="Alarm",
    brightness=1
)
```

Parameters:

- **display** (TM1637): Initialized TM1637 display instance (4-digit).
- **name** (str): Unique identifier for this counter instance. Used in event names.
- **message_queue** (list, optional): Queue for sending counter events. If None, events are not queued. Default: None
- **alarm_message** (str, optional): Alarm message text. Default: "Alarm"
- **brightness** (int, optional): Display brightness level (0-7). Default: 1

Note: Counter4 inherits from Counter6 but is configured for 4-digit displays.

ILIDISPLAY (ILI9341)

ILI9341 TFT display controller interface

Class Instantiation

```
from ili_classes import ILIDISPLAY

display = ILIDISPLAY(
    message_queue=None,
    timer_id=2
)
```

Parameters:

- **message_queue** (list, optional): Queue for sending display events. If None, events are not queued. Default: None
- **timer_id** (int, optional): Timer ID for clock functionality. Default: 2

Note: The ILI9341 display uses fixed pin assignments: SPI on pins 14 (SCK), 13 (MOSI), DC on pin 2, CS on pin 15, RST on pin 15, backlight on pin 21.

ILIDISPLAY (CYD)

ILI9341 display driver for CYD hardware

Class Instantiation

```
from cyd_classes import ILIDISPLAY

display = ILIDISPLAY()
```

Parameters:

No parameters required. The CYD version uses fixed pin assignments: SPI on pins 14 (SCK), 13 (MOSI), DC on pin 2, CS on pin 15, RST on pin 4, backlight on pin 21.

RGBLED

RGB LED control for CYD hardware

Class Instantiation

```
from cyd_classes import RGBLED

rgb = RGBLED(
    red_pin=4,
    green_pin=16,
    blue_pin=17
)
```

Parameters:

- **red_pin** (int, optional): GPIO pin for red LED channel. Default: 4
- **green_pin** (int, optional): GPIO pin for green LED channel. Default: 16
- **blue_pin** (int, optional): GPIO pin for blue LED channel. Default: 17

Note: Uses inverse logic (0 = on, 1 = off).

TOUCH

XPT2046 touch controller for CYD hardware

Class Instantiation

```
from cyd_classes import TOUCH

touch = TOUCH(
    int_handler=None
)
```

Parameters:

- **int_handler** (function, optional): Interrupt handler function for touch events. If None, uses default handler that prints touch coordinates. Default: None

Note: Uses fixed pin assignments: SPI2 on pins 25 (SCK), 32 (MOSI), 39 (MISO), CS on pin 33, INT on pin 36.

Status_Page

Generic status display page for OLED displays

Lift_Status_Page

Lift control status page for OLED displays

Class Instantiation

```
from page_classes import Lift_Status_Page
from machine import I2C, Pin
import ssd1306

i2c = I2C(0, scl=Pin(22), sda=Pin(21))
display = ssd1306.SSD1306_I2C(128, 64, i2c)
obj_list = []  # List of objects to show status for

lift_page = Lift_Status_Page(
    display=display,
    obj_list=obj_list
)
```

Parameters:

- **display** (SSD1306_I2C): Initialized SSD1306 OLED display instance (128x64).
- **obj_list** (list): List of objects (typically Shelly devices) to display status for.

Displays lift floor, target floor, and floor speed information.

Power_Page

Power monitoring display page for OLED displays

Class Instantiation

```
from page_classes import Power_Page
from shellyio_classes import ShellyPlusPM1

power_socket = ShellyPlusPM1(ip_address="192.168.1.100", name="Power1")
power_page = Power_Page(
    display=display,
    power_socket=power_socket
)
```

Parameters:

- **display** (SSD1306_I2C): Initialized SSD1306 OLED display instance (128x64).
- **power_socket** (ShellyPlusPM1): Initialized ShellyPlusPM1 power monitoring device instance.

Displays power socket state, power consumption, voltage, current, and temperature.

Dim_Page

Dimmer control display page for OLED displays

Class Instantiation

```
from page_classes import Dim_Page
from shellyio_classes import ShellyDimmer2

dimmer = ShellyDimmer2(ip_address="192.168.1.101", name="Dimmer1")
dim_page = Dim_Page(
    display=display,
    dimmer=dimmer
)
```

Parameters:

- **display** (SSD1306_I2C): Initialized SSD1306 OLED display instance (128x64).
- **dimmer** (ShellyDimmer2): Initialized ShellyDimmer2 dimmer device instance.

Displays dimmer state and brightness level.

Event_Page

Event log display page for OLED displays

Class Instantiation

```
from page_classes import Event_Page

event_page = Event_Page(
    display=display,
    name="EventPage1",
    event={"e": "Event1", "d": {}},
    queue=message_queue,
    debug=False
)
```

Parameters:

- **display** (SSD1306_I2C): Initialized SSD1306 OLED display instance (128x64).
- **name** (str): Name for this event page.
- **event** (dict): JSON event dictionary to send when center button is pressed.
- **queue** (list): Message queue for sending events.
- **debug** (bool, optional): Enable debug output. Default: False

VLC_page

VLC media player control page for OLED displays

Class Instantiation

```
from page_classes import VLC_page
from encoder_classes import Encoder_Send

encoder = Encoder_Send(name="VLCEncoder", queue=message_queue)
vlc_page = VLC_page(
    display=display,
    enc=encoder,
    url="http://192.168.1.50:8080/",
    password="password"
)
```

Parameters:

- **display** (SSD1306_I2C): Initialized SSD1306 OLED display instance (128x64).
- **enc** (Encoder_Send): Initialized Encoder_Send instance for controlling VLC playback rate.
- **url** (str, optional): VLC web interface URL. Default: None
- **password** (str, optional): VLC web interface password. Default: None

page_handler

Manages page transitions and updates for OLED displays

Class Instantiation

```
from page_classes import page_handler

pages = [status_page, power_page, dim_page]
handler = page_handler(
    display=display,
    pages=pages
)
```

Parameters:

- **display** (SSD1306_I2C): Initialized SSD1306 OLED display instance (128x64).
- **pages** (list): List of page objects (Status_Page, Power_Page, Dim_Page, etc.) to manage.

Manages cycling through pages and handling focus for updates.

5.2.3 Motor and Actuator Classes

Classes for controlling motion systems

Servo

PWM servo motor control

Class Instantiation

```
from servo_classes import Servo
from machine import Pin

servo = Servo(
    name="Servo1",
    PWM_pin=Pin(18),
    message_queue=None,
    low=3300,
    high=6600,
    debug=False
)
```

Parameters:

- **name** (str): Unique identifier for this servo instance. Used in event names.
- **PWM_pin** (Pin): GPIO pin configured for PWM output (50Hz for servos).
- **message_queue** (list, optional): Queue for sending servo position events. If None, events are not queued. Default: None
- **low** (int, optional): Minimum PWM duty value (16-bit) for servo position 0%. Default: 3300
- **high** (int, optional): Maximum PWM duty value (16-bit) for servo position 100%. Default: 6600
- **debug** (bool, optional): Enable debug output. Default: False

PWM frequency is fixed at 50Hz for servo compatibility. Position is specified as percentage (0-100).

PWM_LED

PWM-controlled LED brightness control

Class Instantiation

```
from servo_classes import PWM_LED
from machine import Pin

led = PWM_LED(
    name="LED1",
    PWM_pin=Pin(18),
    message_queue=None,
    low=0,
    high=65535
)
```

Parameters:

- **name** (str): Unique identifier for this LED instance. Used in event names.
- **PWM_pin** (Pin): GPIO pin configured for PWM output (5kHz).
- **message_queue** (list, optional): Queue for sending LED state events. If None, events are not queued. Default: None
- **low** (int, optional): Minimum PWM duty value (16-bit) for 0% brightness. Default: 0
- **high** (int, optional): Maximum PWM duty value (16-bit) for 100% brightness. Default: 65535

PWM frequency is 5kHz. Supports sequences and glitch effects.

CentreMeter

Center-zero meter using two PWM outputs

Class Instantiation

```
from servo_classes import CentreMeter
from machine import Pin

meter = CentreMeter(
    name="Meter1",
    PWM_pin=Pin(18),
    PWM_pin2=Pin(19),
    message_queue=None,
    low=-65535,
    high=65535,
    glitch_rest=50,
    glitch_depth=20000
)
```

Parameters:

- **name** (str): Unique identifier for this meter instance. Used in event names.
- **PWM_pin** (Pin): GPIO pin configured for PWM output (first channel, 5kHz).
- **PWM_pin2** (Pin): GPIO pin configured for PWM output (second channel, 5kHz).
- **message_queue** (list, optional): Queue for sending meter position events. If None, events are not queued. Default: None
- **low** (int, optional): Minimum PWM duty value (negative for center-zero meters). Default: -65535
- **high** (int, optional): Maximum PWM duty value. Default: 65535
- **glitch_rest** (int, optional): Time between glitches in milliseconds. Default: 50
- **glitch_depth** (int, optional): Amplitude of glitch effect. Default: 20000

Center-zero meters use two PWM channels for bidirectional deflection from center.

HBridge

H-Bridge DC motor control using L298N

Class Instantiation

```
from HBridge_classes import HBridge
from machine import Pin

motor = HBridge(
    name="Motor1",
    queue=None,
    forward_pin=18,
    reverse_pin=19,
    hall_effect=25,
    freq=1000,
    timer=1
)
```

Parameters:

- **name** (str): Unique identifier for this motor instance. Used in event names.
- **queue** (list, optional): Queue for sending motor speed/position events. If None, events are not queued. Default: None
- **forward_pin** (int, optional): GPIO pin number for forward direction PWM control. Default: None
- **reverse_pin** (int, optional): GPIO pin number for reverse direction PWM control. Default: None
- **hall_effect** (int, optional): GPIO pin number for hall effect sensor input (for position feedback). If None, no feedback. Default: None
- **freq** (int, optional): PWM frequency in Hz. Default: 1000
- **timer** (int, optional): Timer ID for periodic updates. Default: 1

Supports bidirectional control with optional hall effect sensor feedback for position tracking.

Fader

Linear fader control with position feedback

Class Instantiation

```
from HBridge_classes import Fader
from machine import Pin, ADC

fader = Fader(
    name="Fader1",
    forward_pin=18,
    reverse_pin=19,
    analogue_pin=34,
    queue=None,
    freq=10000,
    max_duty=800
)
```

Parameters:

- **name** (str): Unique identifier for this fader instance. Used in event names.
- **forward_pin** (int): GPIO pin number for forward direction PWM control.
- **reverse_pin** (int): GPIO pin number for reverse direction PWM control.
- **analogue_pin** (int, optional): ADC pin number for position feedback. If None, no position feedback. Default: None
- **queue** (list, optional): Queue for sending fader position events. If None, events are not queued. Default: None
- **freq** (int, optional): PWM frequency in Hz. Default: 10000
- **max_duty** (int, optional): Maximum PWM duty value for motor control. Default: 800

Requires calibration to establish position limits using the analogue feedback.

StepperX27

Stepper motor control for X27.168 gauge motors with EasyDriver

Class Instantiation

```
from stepper_classes import StepperX27
from machine import Pin

stepper = StepperX27(
    name="Gauge1",
    pin_step=Pin(18),
    pin_dir=Pin(19),
    message_queue=None,
    steps_per_rev=1480,
    speed=1200,
    rotation=270,
    mode="HALF",
    ms1_pin=Pin(20),
    ms2_pin=Pin(21),
    pin_sleep=Pin(22),
    timer_id=3,
    X27Mode=True,
    damping=100
)
```

Parameters:

- **name** (str): Unique identifier for this stepper instance. Used in event names.
- **pin_step** (Pin): GPIO pin for step signal output.
- **pin_dir** (Pin): GPIO pin for direction signal output.
- **message_queue** (list, optional): Queue for sending stepper position events. If None, events are not queued. Default: None
- **steps_per_rev** (int, optional): Number of steps per full revolution. For X27.168 with 1/2 step: 1480. Default: 1480
- **speed** (int, optional): Step speed in steps per second. Default: 1200
- **rotation** (int, optional): Maximum rotation angle in degrees. For gauges: 270. Default: 270
- **mode** (str, optional): Step mode: “FULL”, “HALF”, “QUARTER”, or “EIGHTH”. Default: “FULL”

- **ms1_pin** (Pin, optional): GPIO pin for MS1 microstep control. Required if using microstepping. Default: None
- **ms2_pin** (Pin, optional): GPIO pin for MS2 microstep control. Required if using microstepping. Default: None
- **pin_sleep** (Pin, optional): GPIO pin for sleep/enable control. Default: None
- **timer_id** (int, optional): Timer ID for step generation. Default: 3
- **X27Mode** (bool, optional): Enable X27-specific damping mode. Default: True
- **damping** (int, optional): Damping factor for smooth movement. Default: 100

For X27.168 gauge motors with EasyDriver, use 1/2 step mode, 1480 steps/rev, speed 1200, and 270° rotation for a 12V supply.

StepperX27Hand

Clock hand stepper motor (inherits from StepperX27)

Class Instantiation

```
from stepper_classes import StepperX27Hand
from machine import Pin

hand = StepperX27Hand(
    name="ClockHand1",
    pin_step=Pin(18),
    pin_dir=Pin(19),
    message_queue=None,
    steps_per_rev=1480,
    speed=1200,
    rotation=270,
    mode="HALF",
    ms1_pin=Pin(20),
    ms2_pin=Pin(21),
    pin_sleep=Pin(22),
    timer_id=3
)
```

Parameters:

- **name** (str): Unique identifier for this stepper instance. Used in event names.
- **pin_step** (Pin): GPIO pin for step signal output.
- **pin_dir** (Pin): GPIO pin for direction signal output.
- **message_queue** (list, optional): Queue for sending stepper position events. If None, events are not queued. Default: None
- **steps_per_rev** (int, optional): Number of steps per full revolution. Default: 1480
- **speed** (int, optional): Step speed in steps per second. Default: 1200
- **rotation** (int, optional): Maximum rotation angle in degrees. Default: 270
- **mode** (str, optional): Step mode: “FULL”, “HALF”, “QUARTER”, or “EIGHTH”. Default: “FULL”
- **ms1_pin** (Pin, optional): GPIO pin for MS1 microstep control. Default: None
- **ms2_pin** (Pin, optional): GPIO pin for MS2 microstep control. Default: None
- **pin_sleep** (Pin, optional): GPIO pin for sleep/enable control. Default: None

- **timer_id** (int, optional): Timer ID for step generation. Default: 3

Inherits from StepperX27 but configured for clock hand applications without X27Mode damping.

Nidec

Nidec brushless DC motor control

Class Instantiation

```
from nidec_classes import Nidec
from machine import Pin

motor = Nidec(
    name="Nidec1",
    queue=None,
    speed_pin=18,
    direction_pin=19,
    brake_pin=20,
    encoderA=21,
    encoderB=22
)
```

Parameters:

- **name** (str): Unique identifier for this motor instance. Used in event names.
- **queue** (list, optional): Queue for sending motor speed/position events. If None, events are not queued. Default: None
- **speed_pin** (int, optional): GPIO pin number for speed control PWM (20kHz). Default: None
- **direction_pin** (int, optional): GPIO pin number for direction control. Default: None
- **brake_pin** (int, optional): GPIO pin number for brake control. Default: None
- **encoderA** (int, optional): GPIO pin number for encoder channel A. Default: None
- **encoderB** (int, optional): GPIO pin number for encoder channel B. Default: None

Note: Nidec motors require 8 connections. PWM operates at 20kHz with duty cycle range 15-100%. Speed control is inverted (1023 = off, 0 = full speed).

LinAct

Linear actuator control (push/pull)

Class Instantiation

```
from linact_classes import LinAct

actuator = LinAct(
    name="LinAct1",
    pull_pin=18,
    push_pin=19,
    message_queue=None
)
```

Parameters:

- **name** (str): Unique identifier for this actuator instance. Used in event names.
- **pull_pin** (int): GPIO pin number for pull (retract) control.
- **push_pin** (int): GPIO pin number for push (extend) control.
- **message_queue** (list, optional): Queue for sending actuator state events. If None, events are not queued. Default: None

Note: Travel time defaults to 3.0 seconds. Both pins are set low on initialization for safety.

5.2.4 Lighting Classes

Classes for LED and lighting control

Alarm

Simple alarm/indicator pattern for NeoPixel strips

Class Instantiation

```
from neo_classes import Alarm
from machine import Pin
from neopixel import NeoPixel

pin = Pin(18, Pin.OUT)
neo_strip = NeoPixel(pin, 60)  # 60 pixels

alarm = Alarm(
    name="Alarm1",
    neo_strip=neo_strip,
    count=60,
    message_queue=None,
    message="Alarm",
    max=255,
    type=3
)
```

Parameters:

- **name** (str): Unique identifier for this alarm instance. Used in event names.
- **neo_strip** (NeoPixel): Initialized NeoPixel strip instance.
- **count** (int): Number of pixels in the strip.
- **message_queue** (list, optional): Queue for sending alarm events. If None, events are not queued. Default: None
- **message** (str, optional): Alarm message text. Default: “Alarm”
- **max** (int, optional): Maximum brightness value (0-255). Default: 255
- **type** (int, optional): Alarm type identifier. Default: 3

Supports chase patterns and alarm timing.

Pulse

Advanced pulsing and animation patterns for NeoPixel strips

Class Instantiation

```
from neo_classes import Pulse
from machine import Pin
from neopixel import NeoPixel

pin = Pin(18, Pin.OUT)
neo_strip = NeoPixel(pin, 60)  # 60 pixels

pulse = Pulse(
    name="Pulse1",
    neo_strip=neo_strip,
    count=60,
    message_queue=None,
    message="Pulse",
    max=255,
    fade_rate=0.9,
    timernumber=1
)
```

Parameters:

- **name** (str): Unique identifier for this pulse instance. Used in event names.
- **neo_strip** (NeoPixel): Initialized NeoPixel strip instance (supports 3 or 4 bytes per pixel).
- **count** (int): Number of pixels in the strip.
- **message_queue** (list, optional): Queue for sending pulse events. If None, events are not queued. Default: None
- **message** (str, optional): Event message text. Default: "Pulse"
- **max** (int, optional): Maximum brightness value (0-255). Default: 255
- **fade_rate** (float, optional): Fade rate multiplier (0.0-1.0). Higher values = slower fade. Default: 0.9
- **timernumber** (int, optional): Timer ID for animation updates. Default: 1

Supports multiple animation patterns: pulse, stream, flame, hypospray, fade_pulse, rotate, and more.

Blocks

Block-based pattern system for NeoPixel strips (meter displays, segmented patterns)

Class Instantiation

```
from neo_classes import Blocks
from machine import Pin
from neopixel import NeoPixel

pin = Pin(18, Pin.OUT)
neo_strip = NeoPixel(pin, 60)  # 60 pixels
```

(continues on next page)

(continued from previous page)

```
blocks = Blocks(
    name="Meter1",
    neo_strip=neo_strip,
    count=60,
    message_queue=None,
    max=255,
    fade_rate=0.9,
    timernumber=1
)
```

Parameters:

- **name** (str): Unique identifier for this blocks instance. Used in event names.
- **neo_strip** (NeoPixel): Initialized NeoPixel strip instance (supports 3 or 4 bytes per pixel).
- **count** (int): Number of pixels in the strip.
- **message_queue** (list, optional): Queue for sending block pattern events. If None, events are not queued. Default: None
- **max** (int, optional): Maximum brightness value (0-255). Default: 255
- **fade_rate** (float, optional): Fade rate multiplier (0.0-1.0). Higher values = slower fade. Default: 0.9
- **timernumber** (int, optional): Timer ID for pattern updates. Default: 1

Supports up to 16 blocks (expandable to 24) for creating segmented patterns, meter displays, and multi-zone effects. Each block can have independent colors, modes, and behavior.

DMX

DMX512 lighting protocol controller via MAX485 module

Class Instantiation

```
from dmx_classes import DMX

dmx = DMX(
    port=2
)
```

Parameters:

- **port** (int, optional): UART port number for DMX communication. Default: 2

Note: DMX uses UART for communication. The DMX message buffer is 513 bytes (1 start byte + 512 channels). Supports fading and sequence playback.

5.2.5 Communication Classes

Classes for data communication and IoT integration

Encoder_Send

Rotary encoder event generator with debouncing and position tracking

Class Instantiation

```
from encoder_classes import Encoder_Send
from rotary import RotaryIRQ  # or other encoder library

# Optional: physical encoder instance
physical_encoder = RotaryIRQ(pin_num_clk=18, pin_num_dt=19, reverse=True)

encoder = Encoder_Send(
    name="Encoder1",
    queue=message_queue,
    debounce=250,
    distance_per_step=10,
    physical_encoder=physical_encoder,
    circumference=None,
    localQueue=None,
    debug=False
)
```

Parameters:

- **name** (str): Unique identifier for this encoder instance. Used in event names.
- **queue** (list): Queue for sending encoder position/distance events.
- **debounce** (int, optional): Debounce time in milliseconds. Waits this long after encoder stops moving before sending event. Default: 250
- **distance_per_step** (int, optional): Distance in arbitrary units per encoder step (for linear encoders). Default: 10
- **physical_encoder** (RotaryIRQ, optional): Physical encoder instance (e.g., RotaryIRQ). If None, position must be set manually. Default: None
- **circumference** (float, optional): Circumference for turntable applications (for angle calculation). Default: None
- **localQueue** (list, optional): Additional local queue for events. Default: None
- **debug** (bool, optional): Enable debug output. Default: False

Supports both rotary and linear encoders. Can calculate angles for turntables when circumference is provided. Supports OSC integration for sending messages to external systems.

CC1101

Attention: This module is **Experimental** and may change in future versions.

CC1101 low-power sub-1GHz radio transceiver interface

Note: Experimental - API may change in future versions

Class Instantiation

```
from C1101_classes import CC1101
from machine import SPI, Pin

spi = SPI(1, baudrate=50000, polarity=0, phase=0, sck=Pin(14), mosi=Pin(13),␣
↪miso=Pin(12))
cs = Pin(5, Pin.OUT)
gdo0 = Pin(4, Pin.IN)

radio = CC1101(
    spi=spi,
    cs=cs,
    gdo0=gdo0,
    baudrate=50000,
    frequency=433000000,  # 433 MHz
    syncword="ABCD",  # 4-byte hex string
    offset=0,
    debug=False
)
```

Parameters:

- **spi** (SPI): Initialized SPI bus instance for communication with CC1101.
- **cs** (Pin): GPIO pin configured as output for chip select.
- **gdo0** (Pin): GPIO pin configured as input for GDO0 (general purpose digital output 0).
- **baudrate** (int): SPI baud rate in Hz.
- **frequency** (int): RF frequency in Hz (e.g., 433000000 for 433 MHz).
- **syncword** (str): 4-character hex string for packet synchronization (e.g., “ABCD”).
- **offset** (int, optional): Frequency offset in Hz. Default: 0
- **debug** (bool, optional): Enable debug output. Default: False

Note: Experimental module. The CC1101 is automatically reset and configured with sample rate 4000 Hz.

ShellyPlusPM1

Interface to Shelly Plus PM1 power monitoring switch

Class Instantiation

```
from shellyio_classes import ShellyPlusPM1

switch = ShellyPlusPM1(
    ip_address="192.168.1.100",
    name="PM1",
    queue=None
)
```

Parameters:

- **ip_address** (str): IP address of the Shelly Plus PM1 device on the network.
- **name** (str, optional): Unique identifier for this switch instance. Default: "PM1"
- **queue** (list, optional): Queue for sending status events. If None, events are not queued. Default: None

Communicates via HTTP REST API. Provides power monitoring (voltage, current, power consumption) and switch control (on/off/toggle). Status includes temperature monitoring.

ShellyDimmer2

Interface to Shelly Dimmer 2 dimming switch

Class Instantiation

```
from shellyio_classes import ShellyDimmer2

dimmer = ShellyDimmer2(
    ip_address="192.168.1.101",
    name="Dimmer2",
    queue=None
)
```

Parameters:

- **ip_address** (str): IP address of the Shelly Dimmer 2 device on the network.
- **name** (str, optional): Unique identifier for this dimmer instance. Default: "Dimmer2"
- **queue** (list, optional): Queue for sending status events. If None, events are not queued. Default: None

Communicates via HTTP REST API. Provides dimming control (0-100% brightness) and power monitoring. Supports calibration mode for proper operation.

WebSocket Client (wsclasses)

WebSocket client implementation and utility functions

Note: This is a module containing utility functions and async coroutines for WebSocket communication. It's not a class-based module but provides network connection, button handling, and WebSocket message processing functions.

The module handles:

- WiFi connection management via `wifi_connect()`
- Button debouncing and action queuing
- WebSocket client connection and message handling
- Async event loops for button and WebSocket communication

Typical Usage:

The module expects a `config.json` file with WebSocket configuration and sets up button handlers on GPIO pins 12, 14, and 27. It creates a global WebSocket client instance and manages message queues for communication with the SetWideNet server.

Key Functions:

- `wifi_connect()`: Async function to connect to WiFi network
- `button_loop()`: Async loop for handling button presses and sending via WebSocket
- `ws_loop()`: Async loop for receiving and processing WebSocket messages

CHAPTER

SIX

LORA CLASSES

Long Range Radio Communication Classes for Remote Controls

6.1 Overview

The LORA classes are MicroPython modules designed for long-range radio communication. These classes provide abstractions for LoRa transceivers used in remote control systems, orthogonal to physical props. They enable wireless communication over long distances using LoRa radio modules such as SX127x and SX1262.

All classes follow a consistent interface pattern: - Initialization via `__init__()` with hardware pin assignments - Async TX/RX loops for sending and receiving messages - Queue-based message handling - Integration with SetWideNet event system

6.2 LoRaTransceiver (SX127x)

LoRa transceiver for SX127x radio modules with async TX/RX loops

Module: lora_classes.py

6.3 LoRaTransceiver (SX1262)

LoRa transceiver for SX1262 radio modules with async TX/RX loops

Module: lora_sx1262_classes.py

6.4 RRRS_Page

Display page class for RRRS (Run/Record/Remote/Stop) status display on OLED

Module: page_classes.py

CHAPTER

SEVEN

OSC SERVICE

OSC Service for communicating with Unreal Engine and showrunner software

7.1 Overview

The OSC service bridges SetWideNet JSON events with the OSC (Open Sound Control) protocol. This allows integration with systems that use OSC for communication, such as Unreal Engine and showrunner software. The service can map incoming JSON events to OSC messages and vice versa, enabling bidirectional communication.

7.2 OSC Service Module

OSC service for mapping JSON events to OSC messages and vice versa

Note: Bridges SetWideNet JSON events with OSC protocol for integration with Unreal Engine and showrunner software

CHAPTER

EIGHT

UNREAL ENGINE CLASSES

Unreal Engine Integration Classes

8.1 Overview

The Unreal Engine classes provide integration with Unreal Engine 5.5's Remote Control API. These classes enable Python scripts to connect to Unreal Engine, control cameras, trigger functions, set properties, and interact with exposed actors and presets.

All classes follow a consistent interface pattern: - HTTP-based API communication with Unreal Engine Remote Control server - WebSocket connections for real-time updates - Preset and actor management - Function and property manipulation

8.2 ueInstance

Main class for connecting to and controlling Unreal Engine Remote Control API

8.3 UE5Extractor

Extracts and presents exposed functions, properties, and actors from UE5 Remote Control presets

8.4 Camera

Camera control class for Unreal Engine

CHAPTER

NINE

WEBSOCKETS CONFIGURATION

Mapping and Sequences Configuration for WebSocket Server

9.1 Overview

The WebSocket server uses two key configuration files to manage event mapping and sequences:

- **rules.json**: Maps incoming events to other events or sequences
- **sequences.json**: Defines named sequences of events that can be triggered

These configuration files enable complex event routing and automated sequences without requiring code changes.

9.2 Mapping (rules.json)

The `rules.json` file defines event mappings. When an event is received that matches a key in the mapping, it is transformed or redirected according to the mapping definition.

Rules can map:

- Button clicks to sequences
- Switch states (ON/OFF) to different events
- Alarm events to sequences
- Encoder events to sequences

Example mapping structure:

```
{
  "Button1": {"e": "Sequence", "d": "Sequence1"},
  "Button2": {"e": "Sequence", "d": "Sequence2"},
  "Switch4ON": {"e": "LinAct1", "d": "push"},
  "Switch4OFF": {"e": "LinAct1", "d": "pull"},
  "Alarm50": {"e": "Sequence", "d": "Emote-Launch"}
}
```

When a button named “Button1” is clicked, it triggers the “Sequence1” sequence. Switch states are automatically appended (e.g., “Switch4ON” or “Switch4OFF”) to create unique mapping keys.

9.3 Sequences (sequences.json)

The `sequences.json` file defines named sequences of events. Each sequence is an array of JSON events that are executed in order when the sequence is triggered.

Example sequence structure:

```
{
  "Sequence1": [
    {"e": "Clock1", "d": "blank"},
    {"e": "Counter1", "d": "stop"},
    {"e": "Counter1", "d": "clear"},
    {"e": "Meter1", "d": 50},
    {"e": "Indicator1", "d": {"text": "Normal", "colour": "green"}}
  ],
  "Sequence2": [
    {"e": "Clock1", "d": "start"},
    {"e": "Counter1", "d": {"goto": 4760, "direction": 1}},
    {"e": "Counter1", "d": "start"}
  ]
}
```

To trigger a sequence, send a Sequence event:

```
{"e": "Sequence", "d": "Sequence1"}
```

9.3.1 Usage Pattern

The typical usage pattern is:

1. An event arrives at the WebSocket server (e.g., a button click)
2. The server checks `rules.json` for a mapping
3. If mapped to a sequence, the server looks up the sequence in `sequences.json`
4. All events in the sequence are sent to subscribed clients

This allows complex event chains to be configured without code changes. Rules can be reloaded dynamically using the `RefreshRules` event.

9.4 Server-Side Utility Classes

9.4.1 EDL (Edit Decision List) Class

The EDL class provides functionality for generating Edit Decision List files for video editing software. EDL files are used to mark events during filming, typically starting at a clapperboard event and ending at a CUT event.**Module:** edl_classes.py**class EDL**(edl_file, frame_rate=24)

Initialize EDL generator.

param edl_file
Base filename for EDL file (timestamp will be appended)

param frame_rate
Frame rate for timecode calculation (default: 24 fps)

type edl_file
string

type frame_rate
integer

Creates an EDL generator that will produce timestamped event markers compatible with video editing software like DaVinci Resolve.

start_edl_session()

Start a new EDL recording session. Opens a new EDL file with timestamped filename and writes the standard header. Logs an initial "Clapperboard started EDL session" event.

log_event(event_name, colour="blue")

Log an event to the EDL file. :param event_name: Description of the event to log :param colour: Marker colour - "red", "green", "blue", "yellow", "purple", or "orange" (default: "blue") :type event_name: string :type colour: string

Creates an EDL entry with timecode relative to the session start. Each event is assigned a sequential marker number and colour coding.

end()

Close the EDL file and finalize the session.

Closes the file handle and resets the session state. The EDL file is saved with a timestamped filename.

Example Usage:

```
from edl_classes import EDL

edl = EDL("scene_01")
edl.start_edl_session()
edl.log_event("Button A Clicked", colour="red")
edl.log_event("Switch C Activated", colour="green")
edl.log_event("Distance Sensor Triggered", colour="blue")
edl.end()
```

The generated EDL file can be imported into video editing software to automatically create markers at the logged event timestamps.

CHAPTER

TEN

WEBSOCKET ELEMENT CLASSES

Web Interface Element Classes for SetWideNet Virtual Props

10.1 Overview

The Element Classes are JavaScript ES6 classes that provide interactive UI elements for SetWideNet web interfaces. These classes handle browser-based UI components that communicate via WebSocket connections to display and control prop states. They are used for virtual props and their related control pages.

All classes follow a consistent pattern: - Constructor initialization with DOM element ID and configuration - `update()` method for receiving state changes via WebSocket events - Event-driven communication with SetWideNet server

Terminology Note: "Screebly" is the product name for the hardware device. In SetWideNet event calls, "Gribbley" (or variations like "Gribbley1", "Gribbley2") is used as the event identifier to target a specific Screebly device. When sending events to control a Screebly, use "Gribbley" (with an ID) as the event name, even though the product itself is called Screebly.

10.2 Element Classes

10.2.1 Button

Interactive button element with visual feedback and WebSocket communication.

constructor(obj, socket, height, width, name="Facility", legend=null, idleColour="grey", onColour="green", offColour="black")

Initialize button element.

param obj
DOM element ID where button will be rendered

param socket
WebSocket connection instance

param height
Button height in pixels

param width
Button width in pixels

param name
Event name sent when button is clicked (default: "Facility")

param legend
Button label text (defaults to name if null)

param idleColour
Background color when idle (default: “grey”)

param onColour
Background color when state is “1” (default: “green”)

param offColour
Background color when state is “0” (default: “black”)

Creates a styled button element with mouse interaction handlers and sends JSON events on click: `{"e": name, "d": "CLICK"}`

update(state)

Update button visual state.

param state
State string (“idle”, “0”, or “1”)

type state
string

Changes button background color based on state and definitions configured in constructor.

click()

Handle button click event. Sends WebSocket message with event name and “CLICK” data.

10.2.2 SwitchMonitor

Read-only button variant for displaying switch states without click interaction.

Extends `Button`.

constructor(obj, socket, height, width, name=”Facility”, legend=null, idleColour=”grey”, onColour=”green”, offColour=”#000018”)

Initialize switch monitor display. Parameters same as Button, with darker default off color.

click()

Override of Button click() - does nothing since this is a monitor-only display.

10.2.3 Meter

Meter/gauge display with animated pointer and value display.

constructor(obj, height, width, bezel=”Bezel.png”, pointer=”Pointer.png”, scale=240, title=”Meter”, units=”Watts”, min=0, max=100, novalue=false)

Initialize meter element.

param obj
DOM element ID where meter will be rendered

param height
Meter height in pixels

param width
Meter width in pixels

param bezel
Image file path for meter bezel/background

param pointer
Image file path for meter pointer

param scale
Pointer rotation scale in degrees (default: 240)

param title
Meter title text

param units
Units string to display with value

param min
Minimum value for scaling

param max
Maximum value for scaling

param novalue
If true, don't display numeric value (default: false)

update(event_data)

Update meter value and pointer position.

param event_data
Numeric value or object with delta/dampening properties

Accepts numeric value for direct update, or object with 'delta' or 'dampening' properties for animated updates.

10.2.4 Battery

Battery gauge display extending Meter class.

Extends `Meter`.

constructor(obj, height, width, bezel="Bezel.png", pointer="Pointer.png", scale=240, title="Battery", units="V", min=0, max=100)

Initialize battery meter. Parameters same as Meter, with battery-specific defaults.

10.2.5 Stepper

Stepper/gauge display with position control.

constructor(obj, height, width, bezel="Bezel.png", pointer="Pointer.png", scale=240, title="Stepper", min=0, max=100)

Initialize stepper display element.

param obj
DOM element ID where stepper will be rendered

param height
Stepper height in pixels

param width
Stepper width in pixels

param bezel
Image file path for stepper bezel/background

param pointer
Image file path for stepper pointer

param scale
Pointer rotation scale in degrees (default: 240)

param title
Stepper title text

param min
Minimum position value

param max
Maximum position value

update(event_data)

Update stepper position.

param event_data
Object with 'position' property, or numeric value

10.2.6 Compass

Compass display with heading indication.

constructor(obj, height, width, bezel="Bezel.png", pointer="Pointer.png", title="Compass")

Initialize compass display element.

param obj
DOM element ID where compass will be rendered

param height
Compass height in pixels

param width
Compass width in pixels

param bezel
Image file path for compass bezel/background

param pointer
Image file path for compass pointer/needle

param title
Compass title text

update(event_data)

Update compass heading.

param event_data
Object with 'heading' property (0-360 degrees), or numeric heading value

10.2.7 Clock

Analog clock display with hour and minute hands.

constructor(obj, height, width, bezel="Bezel.png", hourHand="HourHand.png", minuteHand="MinuteHand.png", title="Clock")

Initialize analog clock display element.

param obj
DOM element ID where clock will be rendered

param height
Clock height in pixels

param width
Clock width in pixels

param bezel
Image file path for clock bezel/background

param hourHand
Image file path for hour hand

param minuteHand
Image file path for minute hand

param title
Clock title text

update(event_data)

Update clock time.

param event_data
Object with 'hours' and 'minutes' properties, or time string in HH:MM format

10.2.8 DigitalClock

Digital clock display with numeric time representation.

constructor(elementId)

Initialize digital clock display element.

param elementId
DOM element ID where clock will be rendered

update()

Update digital clock time. Called automatically via interval timer.

10.2.9 Indicator

SVG-based indicator display with text and color states.

constructor(obj, height, width, SVGTemplate="IndicatorTest.svg", legend="Facility")

Initialize indicator element.

param obj
DOM element ID where indicator will be rendered

param height
Indicator height in pixels

param width
Indicator width in pixels

param SVGTemplate
SVG template file path

param legend
Legend text to display

update(condition)

Update indicator state.

param condition
Object with 'text' and 'colour' properties, plus optional Layer properties for multi-layer indicators

10.2.10 ArtificialHorizon

Artificial horizon display showing pitch and bank angles.

constructor(obj, height, width, SVGTemplate="ArtificialHorizon.svg", legend="Facility")

Initialize artificial horizon element.

param obj
DOM element ID where artificial horizon will be rendered

param height
Display height in pixels (should be square, e.g., 2160)

param width
Display width in pixels (should be square, e.g., 2160)

param SVGTemplate
SVG template file path

param legend
Legend text

update(bank, pitch)

Update artificial horizon display.

param bank
Bank angle in degrees

param pitch
Pitch angle in degrees

10.2.11 GenericSVG

Generic SVG display class for custom SVG-based UI elements.

constructor(obj, height, width, SVGTemplate="GenericSVG.svg", legend="Facility")

Initialize generic SVG element.

param obj
DOM element ID where SVG will be rendered

param height
SVG height in pixels

param width
SVG width in pixels

param SVGTemplate
SVG template file path

param legend
Legend text

rotate_item(item, angle, ua=none, rate=0.5)

Rotate an item within the SVG around its center.

param item
ID of the SVG element to rotate

param angle
Rotation angle in degrees

param ua
User agent string (for browser-specific handling)

param rate
Animation rate in seconds (default: 0.5)

layer_visible(layer, state)

Set the visibility of a layer in the SVG.

param layer
ID of the SVG layer element

param state
Visibility state ("visible" or true for visible, "hidden" or false for hidden)

10.2.12 Grid

Generic grid class for managing a grid of cells in JavaScript. Used for complex layout displays.

Module: grid_class.js

constructor(id, cols, rows, innerHTML="", isRound=false, circleCenterX=null, circleCenterY=null, circleRadius=null)

Initialize grid layout.

param id
Base ID for grid cells (cells will be named id_r{row}_c{col})

param cols
Number of columns in grid

param rows
Number of rows in grid

param innerHTML
Default HTML content for each cell

param isRound
If true, grid cells outside circle radius are hidden (default: false)

param circleCenterX
X coordinate of circle center for round grids (defaults to cols/2)

param circleCenterY
Y coordinate of circle center for round grids (defaults to rows/2)

param circleRadius
Radius in cells for round grids (defaults to min(cols, rows)/2)

getCell(row, col)

Get cell object at specified row and column.

param row
Row index (0-based)

param col
Column index (0-based)

returns
Cell object

raises Error
If row or col is out of bounds

setCell(row, col, innerHTML, overrideLock=false)

Set HTML content of a cell.

param row
Row index

param col
Column index

param innerHTML
HTML content to set

param overrideLock
If true, update even if cell is locked (default: false)

setCellBG(row, col, color, overrideLock=false)

Set background color of a cell.

param row
Row index

param col
Column index

param color
CSS color value

param overrideLock
If true, update even if cell is locked (default: false)

lockCells(cell_list)

Lock cells to prevent updates. Locked cells can only be updated with overrideLock=true.

param cell_list
Array of cell positions in "row_col" format

unlockAllCells()

Remove all cell locks.

10.3 Energy Display Classes

The energy display classes provide battery and fuel gauge visualization for energy monitoring applications.

Module: energy_classes.js

10.3.1 BatteryDisplay

Battery level display with segmented energy bars and voltage/percentage readouts.

constructor(elementIdOrOptions, orientation='vertical', numBars=5, width=null, height=null, showVoltage=true, showPercentage=true, numCells=1, voltagePerCell=4.2, cellLowVoltage=3.3)

Initialize battery display element.

param elementIdOrOptions
DOM element ID string, or options object with properties:

param orientation
Display orientation - 'vertical' (bars fill bottom-to-top) or 'horizontal' (bars fill left-to-right) (default: 'vertical')

param numBars
Number of segmented energy bars to display (default: 5)

param width
Optional width in pixels (default: null, uses container size)

param height
Optional height in pixels (default: null, uses container size)

param showVoltage
Show voltage display (default: true)

param showPercentage
Show percentage display (default: true)

param numCells
Number of battery cells in series (default: 1)

param voltagePerCell
Voltage per cell when fully charged, in volts (default: 4.2V for Li-ion)

param cellLowVoltage
Low voltage per cell (0% level), in volts (default: 3.3V for Li-ion)

Supports both positional parameters (legacy) and named parameters via options object. Creates a segmented bar display with color-coded states (normal/warning/alarm) based on battery level.

setLevel(percentage)

Set battery level as percentage (0-100%).

param percentage
Battery level percentage (0-100)

type percentage
number

Automatically calculates voltage from level based on cell configuration.

setVoltage(voltage)

Set battery voltage directly.

param voltage
Total battery voltage in volts

type voltage
number

Automatically calculates level percentage from voltage based on cell configuration.

setNumCells(numCells)

Set number of battery cells in series.

param numCells
Number of cells (1-10)

type numCells
number

setVoltagePerCell(voltagePerCell)

Set voltage per cell when fully charged.

param voltagePerCell
Voltage per cell in volts (2.0-5.0V)

type voltagePerCell
number

setCellLowVoltage(cellLowVoltage)

Set low voltage threshold per cell (0% level).

param cellLowVoltage
Low voltage per cell in volts (2.0-4.0V)

type cellLowVoltage
number

setOrientation(orientation)

Change display orientation.

param orientation
'vertical' or 'horizontal'

type orientation
string

setNumBars(numBars)

Change number of segmented bars (requires reinitialization).

param numBars
Number of bars (1-20)

type numBars
number

update(evt)

Update battery display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'level' (percentage), 'voltage' (volts), 'numCells', 'voltagePerCell', 'cellLowVoltage', 'orientation', 'width', 'height', 'showVoltage', 'showPercentage'.

10.3.2 FuelGauge

Classic fuel gauge display with 120-degree arc and rotating needle.

constructor(elementIdOrOptions, pivotPosition='bottom', size=200, showValue=true, minValue=0, maxValue=100, showLabels=true)

Initialize fuel gauge element.

param elementIdOrOptions
DOM element ID string, or options object with properties:

param pivotPosition
Needle pivot position - 'top' or 'bottom' (default: 'bottom')

param size
Gauge size in pixels (width and height, square) (default: 200)

param showValue
Show numeric value display (default: true)

param minValue
Minimum value for gauge range (default: 0)

param maxValue
Maximum value for gauge range (default: 100)

param showLabels
Show 'E' (Empty) and 'F' (Full) labels at arc ends (default: true)

Creates a circular gauge with 120-degree arc, rotating needle, and color-coded states based on value.

setValue(value)

Set gauge value.

param value
Gauge value (within minValue-maxValue range)

type value
number

setMinValue(minValue)

Set minimum value for gauge range.

param minValue
Minimum value

type minValue
number

setMaxValue(maxValue)

Set maximum value for gauge range.

param maxValue
Maximum value

type maxValue
number

setSize(size)

Change gauge size (requires reinitialization).

param size
Size in pixels (square)

type size
number

setShowValue(show)

Show or hide numeric value display.

param show
true to show, false to hide

type show
boolean

update(evt)

Update fuel gauge from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'value' or 'level' (0-100), 'minValue', 'maxValue', 'size', 'showValue'.

10.4 Medical Display Classes

The medical display classes provide specialized medical monitoring displays for patient monitoring applications.

Module: medical_classes.js

10.4.1 ECGDisplay

ECG (electrocardiogram) monitor with scrolling trace and heart rate display.

constructor(elementId, width=1200, height=400, heartRate=72)

Initialize ECG display element.

param elementId
DOM element ID where ECG will be rendered

param width
Display width in pixels (default: 1200)

param height
Display height in pixels (default: 400)

param heartRate
Initial heart rate in beats per minute (default: 72)

Creates an animated ECG trace display with scrolling waveform, grid overlay, and moving dot indicator. Supports multiple cardiac conditions: normal, tachycardia, bradycardia, atrial fibrillation (afib), ventricular fibrillation (vfib), and asystole.

setHeartRate(bpm)

Set heart rate in beats per minute.

param bpm
Heart rate (30-200 BPM)

type bpm
number

setCondition(condition)

Set cardiac condition/rhythm type.

param condition
Condition string - 'normal', 'tachycardia', 'bradycardia', 'afib', 'vfib', or 'asystole'

type condition
string

Changes the waveform pattern to match the specified cardiac condition.

setTraceColor(color)

Set ECG trace line color.

param color
CSS color value

type color
string

setBackgroundColor(color)

Set display background color.

param color
CSS color value

type color
string

start()

Start ECG animation and trace scrolling.

stop()

Stop ECG animation.

reset()

Clear trace and reset to initial state.

update(evt)

Update ECG display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'heartRate' (BPM), 'condition' (rhythm type), 'traceColor', 'backgroundColor', or string commands: 'start', 'stop', 'reset'.

10.4.2 BloodPressureDisplay

Blood pressure monitor display with systolic/diastolic values and heart rate.

constructor(elementId, ecgDisplayInstance, width=null, height=null)

Initialize blood pressure display element.

param elementId
DOM element ID where display will be rendered

param ecgDisplayInstance
Reference to ECGDisplay instance for heart rate display (optional)

param width
Optional width in pixels (default: null)

param height
Optional height in pixels (default: null)

Displays blood pressure in mmHg format (systolic/diastolic) with color-coded states based on AHA guidelines. Optionally displays heart rate from linked ECGDisplay instance.

setSystolic(value)

Set systolic blood pressure.

param value
Systolic pressure in mmHg (40-250)

type value
number

setDiastolic(value)

Set diastolic blood pressure.

param value
Diastolic pressure in mmHg (20-150)

type value
number

setBloodPressure(systolic, diastolic)

Set both systolic and diastolic values.

param systolic
Systolic pressure in mmHg

param diastolic
Diastolic pressure in mmHg

type systolic
number

type diastolic
number

update(evt)

Update blood pressure display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'systolic', 'diastolic', 'width', 'height'.

10.4.3 O2SaturationDisplay

Oxygen saturation (SpO_2) monitor display.

constructor(elementId, width=null, height=null)

Initialize O2 saturation display element.

param elementId
DOM element ID where display will be rendered

param width
Optional width in pixels (default: null)

param height
Optional height in pixels (default: null)

Displays oxygen saturation percentage with color-coded states: normal (95-100%), mild hypoxemia (90-95%), moderate/severe hypoxemia (<90%).

setSaturation(value)

Set oxygen saturation percentage.

param value
Saturation percentage (0-100%)

type value
number

update(evt)

Update O2 saturation display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'saturation' (percentage), 'width', 'height'.

10.4.4 RespirationRateDisplay

Respiration rate monitor display.

constructor(elementId, width=null, height=null)

Initialize respiration rate display element.

param elementId
DOM element ID where display will be rendered

param width
Optional width in pixels (default: null)

param height
Optional height in pixels (default: null)

Displays respiration rate in breaths per minute with color-coded states: normal (12-24), moderate warning (<12 or >24), severe alarm (<8 or >30).

setRespirationRate(value)

Set respiration rate.

param value
Breaths per minute (0-60)

type value
number

update(evt)

Update respiration rate display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'respirationRate' (breaths/min), 'width', 'height'.

10.4.5 BodyTemperatureDisplay

Body temperature monitor display with Celsius/Fahrenheit support.

constructor(elementId, width=null, height=null)

Initialize body temperature display element.

param elementId
DOM element ID where display will be rendered

param width
Optional width in pixels (default: null)

param height
Optional height in pixels (default: null)

Displays body temperature with color-coded states: normal (36.0-38.5°C), moderate warning (<36.0 or >38.5°C), severe alarm (<35.0 or >40.0°C). Supports both Celsius and Fahrenheit display.

setTemperature(value)

Set body temperature.

param value
Temperature in Celsius (30.0-45.0°C)

type value
number

setUseFahrenheit(useFahrenheit)

Switch between Celsius and Fahrenheit display.

param useFahrenheit
true for Fahrenheit, false for Celsius

type useFahrenheit
boolean

update(evt)

Update body temperature display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'temperature' (Celsius), 'useFahrenheit' (boolean), 'width', 'height'.

10.4.6 SyringePumpDisplay

Syringe pump rate and state monitor display.

constructor(elementId, width=null, height=null)

Initialize syringe pump display element.

param elementId
DOM element ID where display will be rendered

param width
Optional width in pixels (default: null)

param height
Optional height in pixels (default: null)

Displays pump rate in ml/h and state (STOPPED, RUNNING, PAUSED, ALARM, ERROR, WARNING) with color-coded states.

setRate(value)

Set pump rate.

param value
Pump rate in ml/h (0-1000), or null for stopped

type value
number or null

setState(state)

Set pump state.

param state
State string - 'STOPPED', 'RUNNING', 'PAUSED', 'ALARM', 'ERROR', or 'WARNING'

type state
string

update(evt)

Update syringe pump display from WebSocket event.

param evt
Event object with 'd' property containing update data

type evt
object

Accepts event data with properties: 'rate' (ml/h), 'state' (state string), 'width', 'height'.

10.5 Utility Functions

10.5.1 process_gribbley(evt, swnID=false)

Process events for Screebly devices (see terminology note above about Gribbley/Screebly naming).

Note: This function name uses "gribbley" because event identifiers use "Gribbley" (e.g., "Gribbley1") to target Screebly devices in the event system, while "Screebly" is the actual product name.

10.5.2 recall_states(ws)

Recall and restore states from server for all elements.

10.5.3 open_ws()

Open WebSocket connection to SetWideNet server.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

7-segment display, 32

A

ADC, 27
Alarm
    NeoPixel, 45
animation, 46

B

Battery
    voltage monitoring, 27
BLE
    beacon, 31
Blocks
    NeoPixel, 46
Bluetooth
    beacon, 31
BMP280
    pressure sensor, 28
    temperature sensor, 28
bridge, 55

C

Camera
    Unreal Engine, 57
CC1101
    radio, 49
CentreMeter
    PWM, 40
Clock
    TM1637, 32
communication
    encoder, 48
    IoT, 50
    LoRa, 53
    OSC, 55
    radio, 49
    websocket, 51
Counter
    4-digit, 33
    6-digit, 33
    TM1637, 33

D

Dim_Page
    OLED, 37
dimmer, 50
displays
    clock, 32
    counter, 33
    CYD, 34
    LED, 35
    meter, 40
    OLED, 35–38, 53
    TFT, 34
    touchscreen, 35
Distance
    ultrasonic sensor, 29
DMX
    DMX512, 47

E

EasyDriver, 42
Edit Decision List, 19
EDL listener, 19
Encoder_Send
    rotary encoder, 48
Event_Page
    OLED, 37
experimental, 49

F

Fader
    linear actuator, 41

H

Hall effect
    magnetic sensor, 29
HBridge
    DC motor, 41
HC-SR04, 29
HX711
    ADC, 30
    load cell, 30

I

I2C, 28
ILI9341
    CYD, 34
    TFT, 34
ILIDISPLAY, 34

integration
    Unreal Engine, 57
IoT
    Shelly, 50

L

L298N
    H-Bridge, 41
LED indicator, 45
Lift_Status_Page
    OLED, 36
lighting
    alarm, 45
    blocks, 46
    DMX, 47
    pulse, 46
    PWM, 39
lighting protocol, 47
LinAct
    linear actuator, 44
LoRa
    SX1262, 53
    SX127x, 53
LoRaTransceiver
    SX1262, 53
    SX127x, 53

M

magnetic field, 29
MAX485, 47
meter display, 46
motors
    brushless, 44
    DC motor, 41
    fader, 41
    linear actuator, 44
    servo, 39
    stepper, 42, 43

N

Nidec
    brushless DC, 44

O

OSC
    Open Sound Control, 55

P

page_handler
    OLED, 38
power monitoring, 50
Power_Page
    OLED, 36
proximity detection, 31
Pulse
    NeoPixel, 46
PWM_LED
    PWM, 39

R

RED camera, 20
RedCamera, 20
RGB LED
    CYD, 35
RRRS_Page
    OLED, 53

S

Scales
    weight measurement, 30
SceneTime, 17
sensors
    battery, 27
    bluetooth, 31
    distance, 29
    hall effect, 29
    lidar, 31
    pressure, 28
    temperature, 28
    weight, 30
SequenceService, 18
services
    EDL, 19
    RED camera, 20
    SceneTime, 17
    timed sequences, 18
    Weather, 21
Servo
    PWM, 39
ShellyDimmer2
    Shelly, 50
ShellyPlusPM1
    Shelly, 50
showrunner
    OSC, 55
Status_Page
    OLED, 35
StepperX27
    stepper motor, 42
StepperX27Hand
    clock hand, 43
sub-1GHz
    radio, 49

T

TF Luna
    LiDAR, 31
timed sequences, 18

U

UART, 31
UE5
    camera, 57
    extractor, 57
    Remote Control, 57

UE5Extractor
    Unreal Engine, 57
ueInstance
    Unreal Engine, 57
Unreal Engine
    OSC, 55

V

VLC
    media player, 38
VLC_page
    OLED, 38

W

Weather, 21
WebSocket
    client, 51

X

X27
    stepper motor, 42
XPT2046
    touch, 35