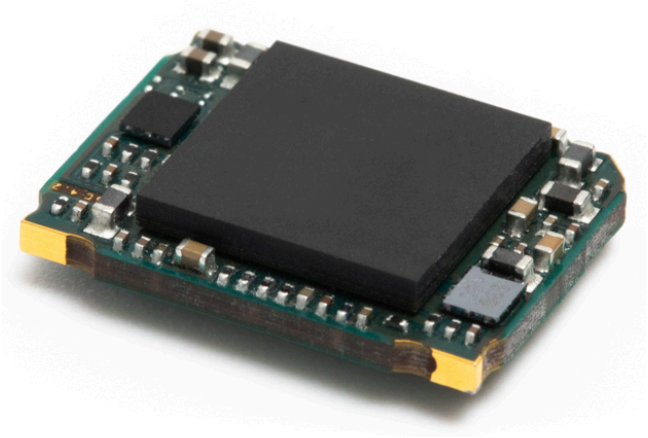
periCORE

Firmware Development



Application Note

Perinet
Seamless IoT Connectivity

Abstract

Perinets *periCORE* Single Pair Ethernet-based communication module [2] includes the capability to operate third party custom firmware images.
This Application Note shows how to develop a custom firmware image by using the periCORE Development Kit [4]. A custom application will be developed step by step and can be installed on the variant *periNODE 0-10V* of Perinets Smart Components [6].

Doc-No.: PRN.100.379 rev: 1 2024-01-30

Document Information

Title	periCORE
Subtitle	Firmware Development
Type	Application Note
Status	Release
Version	1
Date	2024-01-30
Disclosure Restriction	

Contents

1 Introduction

Perinet IIoT *Smart Components* are intended to be used for connecting analogue as well as digital sensors and actuators to the Local Area Network via Single Pair Ethernet (SPE). This document describes building up a custom application based on the *periNODE-distance Smart Component*.

periNODE-distance is a sensor adapter using the analogue ultrasonic distance sensor ULK-WP-M12-V [11]. It collects analogue values provided by the ULK-WP-M12-V distance sensor, converts them into digital representations of the values using the ADS1115 [10] analogue digital converter and provides them for further use at the network interface.

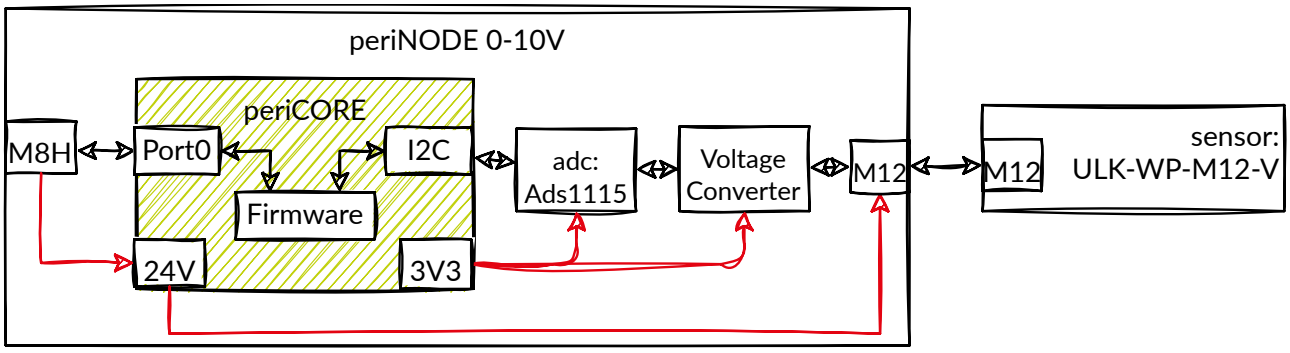


Figure 1: *periNODE distance* block diagram

Figure 1 gives an overview of the components used in *periNODE distance*:

- M8H: SPE connector
- Port0: Network port on *periCORE* board
- 24V: Power provider
- Firmware: *periNODE-distance* firmware
- I2C: I2C port on *periCORE* board
- adc: ADS1115 analogue digital converter
- VoltageConverter: reflects the platform dependent voltage divider (1/8), which is implemented on the PCBA.
- M12: M12 connector to connect the ULK-WP-M12-V distance sensor

1.1 Development Environment

For developing the custom distance application the *periCORE Development Kit* is going to be used. It expands the *periCORE* board with a surrounding development board to allow easy access to all peripheral interfaces. Also all needed parts for network connection and debugging are included. For further information please refer to *periCORE Development Kit Setup Application Note* [3].

1.2 Software Development Environment

As part of the *periCORE Development Kit* Perinet provides a *periCORE Software Development Kit* (SDK). The SDK consists of everything needed to start developing custom applications for Perinet IIoT *Smart Components*.

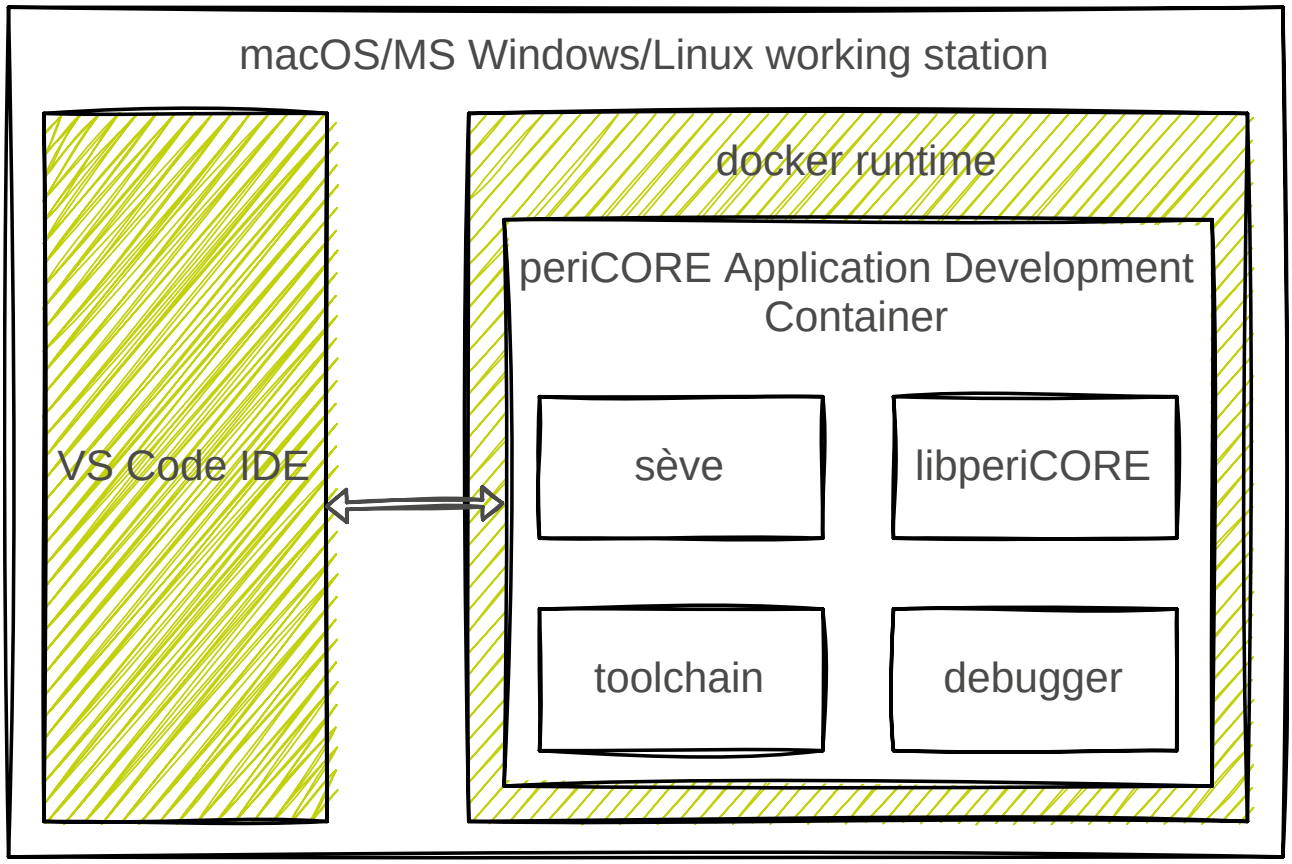


Figure 2: *periCORE SDK* block diagram

Figure 2 gives an overview of the parts installed within the SDK container:

- sève: sève Operating System [5]
- libperiCORE: API to periCORE interfaces [2]
- toolchain: tools needed for complete firmware compilation
- debugger: The GNU debugger [7]

The SDK is shipped as a Docker [**Docker**] container libperiCORE Devcontainer and can be downloaded on the URL: https://gitlab.com/perinet/periCORE/application/devcontainer. The *libperiCORE Devcontainer* is utilized to be directly used with Microsoft Visual Studio Code IDE [**Microsoft:VSCode**] and its Visual Studio Code Remote - Containers extension [**Microsoft:VSCode:Conta**
However - it could also be used to build a *Smart Components* application invoked at the terminal emulator of the used working station:

```
docker run --rm -v $(pwd):/workspace \
-e BDIR=/workspace/_build \
-e DEST=/workspace/_systroot \
registry.gitlab.com/perinet/pericore/devcontainer:latest \
make all -C /workspace
```

Listing 1: Build firmware manually

For further details and proper setup instructions please refer to *periCORE Development Kit User Guide* [4]

1.3 Build System

Within the SDK GNU *make* [8] is used to build the firmware for the *Smart Components*. GNU *make* is going to be utilized with so called *Makefiles*. A fully utilized *Makefile* is part of the *Template application repository* .

The build system provides several build targets:

- install: Build the firmware and install the binaries into the destination folder.
- update: Build the firmware and the webui, bundle them into an update container and install it into the destination folder.
- distclean: Remove all build artifacts.

2 Implementation

Sève Operating System is used as operating system. In a nutshell - sève is a component based, event driven operating system. For details refer to periCORE sève Operating System Datasheet [5].

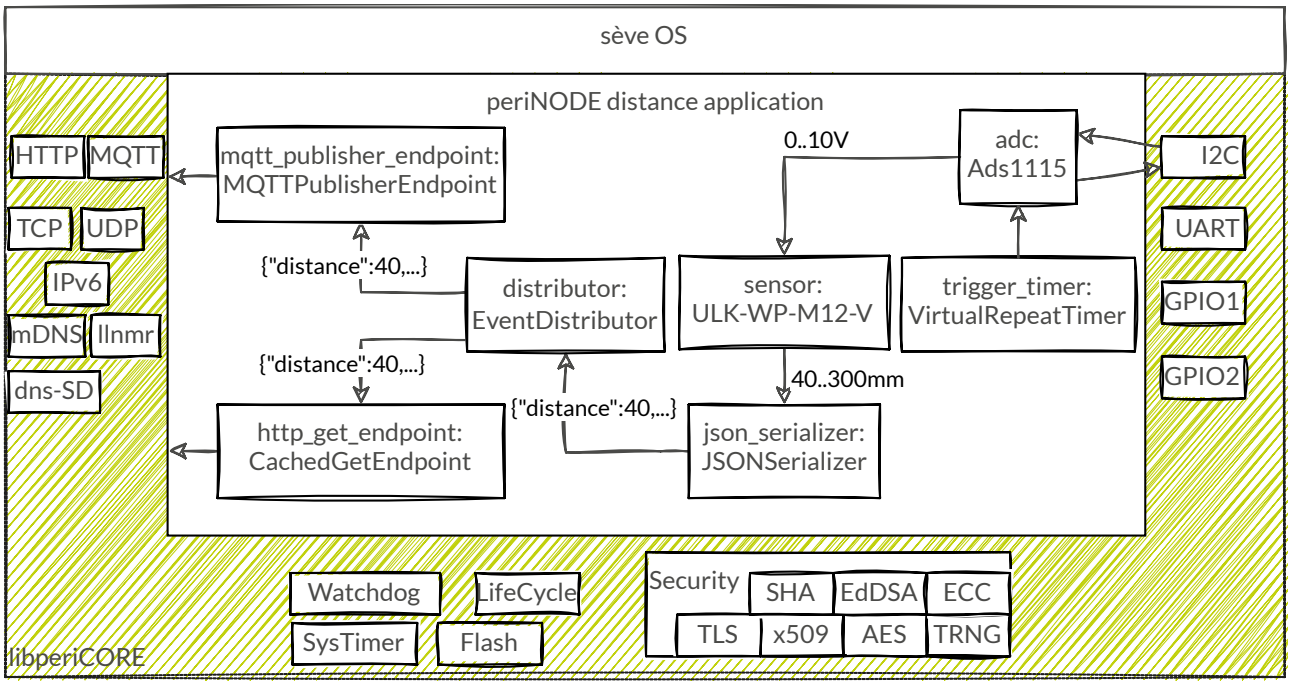


Figure 3: Block diagram of *periNODE distance* application.

Figure 3 gives an overview of the software components used within the *periNODE distance* application.

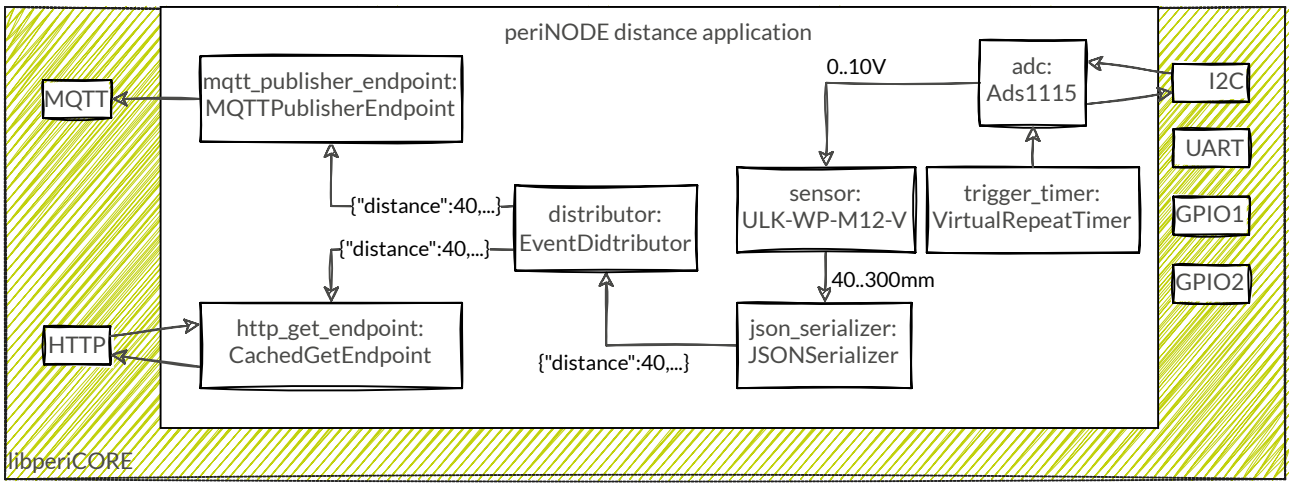


Figure 4: *libperiCORE* API for *libperiCORE distance* application.

Figure 4 shows the API components implemented within *libperiCORE* and their interaction

with the *periNODE distance* application.

Network:

- HTTP: Transmit data with HTTP protocol
- MQTT: Transmit data with MQTT protocol

Sensor/Actor:

- I2C: Transmit Data with *periCORE*s I2C data bus
- UART: Transmit Data with *periCORE*s UART serial interface
- GPIO1: Interact with *periCORE*s GPIO1 peripheral
- GPIO2: Interact with *periCORE*s GPIO2 peripheral

In Figure 4 also all application based components and their interconnection is shown.

- trigger_timer: Periodical trigger timer to start A/D conversion
- Adc: Interface to the ADS1115 [10] Adc
- sensor: Interface to the ULK-WP-M12-V distance sensor
- json_serializer: Data formatter
- distributor: Data distributor
- http_get_endpoint: Interface to the HTTP endpoint
- mqtt_endpoint: Interface to the MQTT endpoint

2.1 Message flow

Figure 4 illustrates the message flow of the *periNODE distance* application.

The `trigger_timer` sends periodically the *trigger* message to the `Adc`. By receiving the *trigger* message the `Adc` generates the *start conversion* message and sends it out to the `I2C` component. When the conversion result is ready the `I2C` component sends the *conversion result* message to the `Adc`. Out of the *conversion result* message data and its configuration the `Adc` calculates the voltage value. This is then sent with the *voltage* message to the `sensor` component. The `sensor` component calculates out of the received voltage value the corresponding distance value. The distance value is then, combined with its unit, sent with the *sample* message to the `json_serializer` component. The `json_serializer` generates a serialized json object which is sent out to the `distributor` component. The `distributor` creates a copy of the serialized json object and sends one to the `mqtt_publisher_endpoint` and the other to the `http_get_endpoint`. Both of them finally sends them out to their corresponding components within *libperiCORE*.

2.2 Component description

2.2.1 Trigger timer

The `trigger_timer` is an object of *sèves* `VirtualRepeatTimer` class.

```
//periodic timer to trigger the measurement
seve::lib::VirtualRepeatTimer trigger_timer {0/*us*/};
```

Listing 2: Create a VirtualRepeat Timer

Its period is constructed with `0` (off). The `trigger_timer` gets the real period during runtime. It is taken from the persistent node config and can be configured with the web UI. It defaults to `t = 1s`.

```
// fetch, configure and start periodic timer
auto period_us = node_environment.get_node_config(). \
    source_interface_config.period_seconds * 1000000;
trigger_timer.set_period(period_us);
```

Listing 3: Change period of `VirtualRepeatTimer` during runtime

The Adc driver implements an `in port` to receive the *trigger* message:

```
// In port for the conversion timer
seve::eventflow::SingleValue<Ads1115> inport_conversion_timer
{
    this,
    &Ads1115::conversion_timer_fired
};
```

Listing 4: Timer `inport` of Adc

The method `void Ads1115::conversion_timer_fired()` is the handler method bound to the `inport_conversion_timer`. It is executed when the *trigger* message is received.

The `out port` of the `trigger timer` and the `in port` of the `Adc` needs to be connected to implement the message flow as shown in Figure 4. Therefore the `VirtualRepeatTimer` provides the `set_outport(...);` method and the `Ads1115` provides the

`get_inport_conversion_timer();` method. The `in port` is bound to the `outport` as follows:

```
// periodic timer to measurement trigger port
trigger_timer.set_outport(adc.get_inport_conversion_timer());
```

Listing 5: Bound Adc `inport` to `timer outport`

2.2.2 A/D converter

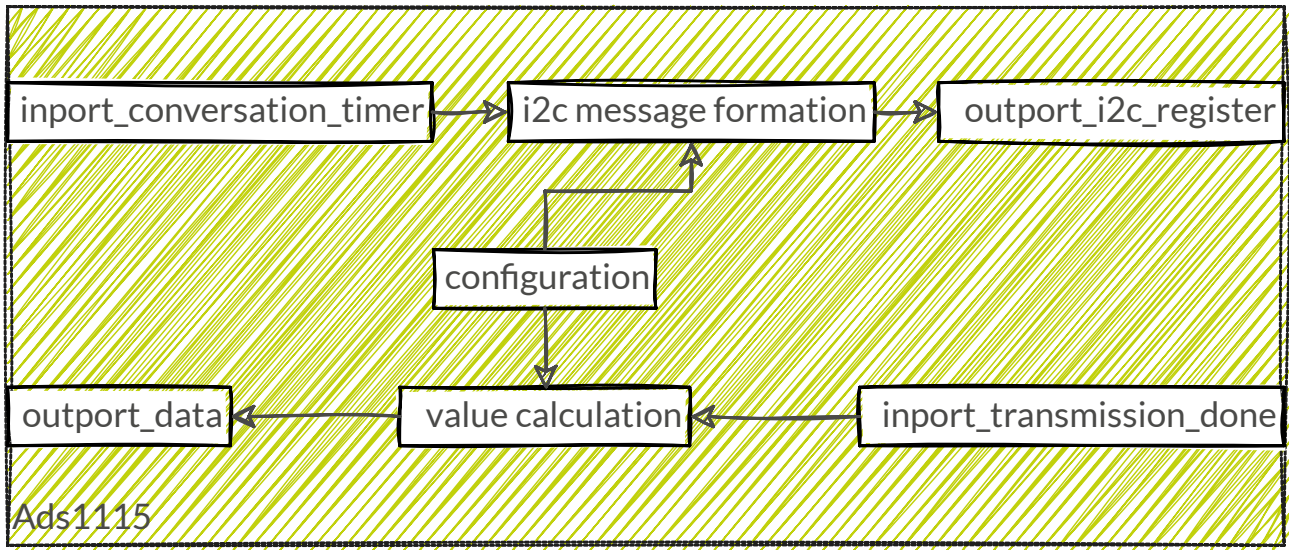


Figure 5: *Ads1115* overview

The *periNODE distance Smart Component* uses an ADS1115 [10] Adc to convert analogue voltage measure values into their corresponding digital representations. Figure 5 gives an overview of the Adc component.

The configuration of the Adc is as follows:

- Full scale range (Fsr): +-2.048V
- Input channel: AIN0 to GND
- Device operating mode: Single-shot mode
- Data rate: 128 SPS
- Comparator Queue and disable: Comparator disabled

With this configuration each conversion needs to be triggered explicitly. This is done with the *trigger* message. When the *trigger* message is received the ADS1115 driver forms the `start conversion` message and sends it out to the I2C driver.

When the *conversion done* message is received the ADS1115 driver evaluates the received data and calculates based on the data and its configuration the measured voltage value. This value is then send out with the *voltage* message.

2.2.3 I2C Device Driver

The I2C device driver is implemented within *libperiCORE* and represents the API to the I2C interface of *periCORE*. It uses the `RegisterAccessDescr` type to transfer data to and from the I2C driver. The `RegisterAccessDescr` works register oriented. This implies each transmission needs to address a register within the adressed I2C slave device.

I2C Register access

```
class RegisterAccessDescr : public seve::lib::Chainable
{
public:
    RegisterAccessDescr(
        uint8_t i2c_slave_address,
        seve::eventflow::Sink<RegisterAccessDescr*>& done_port)
        : i2c_slave_address{i2c_slave_address}, done_port{done_port} {}

    bool isRx{};
    uint8_t i2c_slave_address;
    uint8_t register_address{};
    uint8_t* data{};
    uint32_t len{};
    seve::eventflow::Sink<RegisterAccessDescr*>& done_port;
};
```

Listing 6: `RegisterAccessDescr` type

The `RegisterAccessDescr` contains the following items:

- `isRx`: Indicator for the I2C driver - the required transmission is/isn't a receive transmission.
- `i2c_slave_address`: The I2C slave address of the receiving device.
- `register_address`: The address of the register within the I2C slave device to be accessed.
- `data`: The data to be read/written beginning at `register_address`.
- `len`: The number of bytes of `data`.
- `done_port`: The `out port` assigned by the I2C driver when the transmission is completely done.

The `data` within `RegisterAccessDescr` is just a pointer of type `unit8_t`. The memory (usually a `uint8_t` array) behind this pointer needs to be allocated seperately. The livetime of this memory during the livetime of the `RegisterAccessDescr` object must be garanteed by the caller.

```
periCORE::peripherals::i2c::RegisterAccessDescr register_access;
uint8_t i2c_data[2];

register_access.data = i2c_data;
```

Listing 7: Bind `data` pointer to memory

The `done_port` is assigned by the I2C driver with the same object of type `RegisterAccessDescr` its `in_port` was assigned with. When the I2C transmission was an rx transmission (`isRx = true`) the received data can be found within its `data` memory.

2.2.4 Sensor Value Transformation

Figure 6: *ULK-WP-M12-V* overview

Figure 6 gives an overview of the *sensor* component.

At the `inport_voltage` it takes a `float` value representing a voltage. This value is according to ULK-WP-M12-V [11] specification calculated into a distance value.

```
// calculate measurements based on sensor
const float p1_x = 0.00449; // Volts
const float p1_y = 310;     // mm
const float p2_x = 1.09725; // Volts
const float p2_y = 30;      // mm
// calculate
const float m = (p1_y - p2_y) / (p1_x - p2_x);
const float n = p2_y - m * p2_x;
float distance = m * value + n;
```

Listing 8: Calculate distance out of voltage

By using the `Sample` type. It then is assigned to the `outport_sample`.

```
class Sample
{
public:
    uint32_t incarnation;
    uint64_t sequence_number;

    std::array<DistanceSample,1> data{};

    constexpr static auto memberMap = std::make_tuple(
        std::make_pair(
            std::to_array("incarnation"), &Sample::incarnation),
        std::make_pair(
            std::to_array("sequence_number"), &Sample::sequence_number),
        std::make_pair(
            std::to_array("data"), &Sample::data));
};
```

Listing 9: `Sample` type

The `Sample` contains:

- `incarnation`: Number of reboots since production of the node.
- `sequence_number`: Number of sample transmissions since last reboot.
- `data`: The data to be transmitted.

The data to be transmitted is of the type `DistanceSample`.

```
class DistanceSample
{
    using string_t = perinet::periNODE::api::fixed_short_string_t;

public:
    string_t unit;
    float distance;

    constexpr static auto memberMap = std::make_tuple(
        std::make_pair(std::to_array("unit"), &DistanceSample::unit),
        std::make_pair(std::to_array("distance"), &DistanceSample::
   distance));
};
```

Listing 10: `DistanceSample` type

It contains the distance value and its unit.

The `sample` initialization is done at construction time as follows:

```
sample.incarnation = node_environment.get_incarnation();
sample.sequence_number = 0;
sample.data[0].distance = 0.0f;
sample.data[0].unit = std::to_array("mm\0\0");
```

Listing 11: Initialization of `sample` at construction time

With each `sample` the `incarnation` needs to be incremented and the distance value needs to be set before sending it out:

```
sample.data[0].distance = distance;
sample.sequence_number += 1;

// send sample to out port
if (outport_sample)
{
    outport_sample->assign(sample);
}
```

Listing 12: Manipulation of `sample` at runtime

2.2.5 Data Serialization

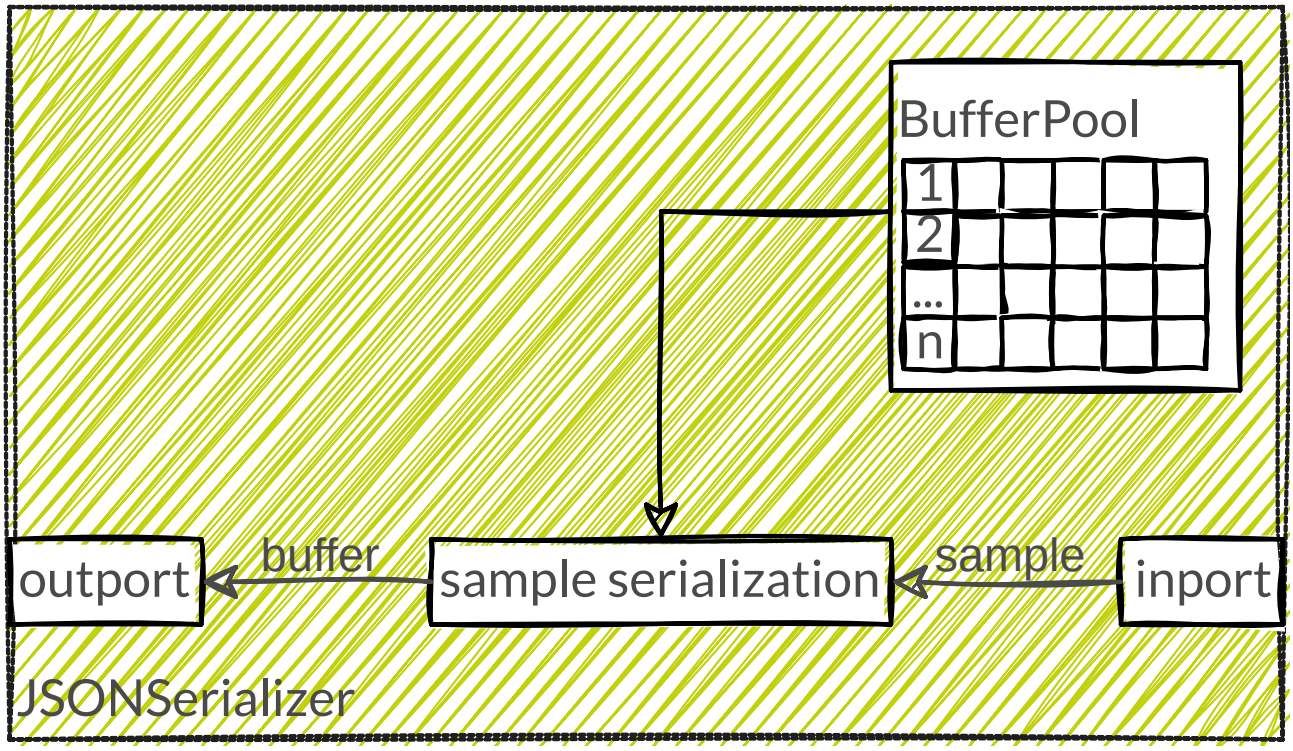


Figure 7: Data Serialization

The `sample` object created by the `sensor` component needs to be serialized in order to be transmitted with either HTTP or MQTT protocol. This is done with the `JSONSerializer` component. It takes a `sample` object as input and needs a `BufferPool` as resource. This pool needs to be utilized to be able to provide buffers in the size of the `sample` objects. Finally it sends out the serialized sample within a buffer allocated by the pool with its `out_port`.

```
// maximum length of serialized sample
const unsigned MAX_SAMPLE_SIZE = 200;

// number of buffers provided by the pool
const unsigned MAX_BUFFER_NUMBER = 5;

// memory to store serialized samples
seve::lib::memory::BufferPool pool{MAX_SAMPLE_SIZE, MAX_BUFFER_NUMBER};

// serializes a sample into a json object
periNODE_distance::lib::JSONSerializer<periNODE_distance::api::Sample>
    json_serializer{pool};
```

Listing 13: Construction of `JSONSerializer`

2.2.6 Data Distribution

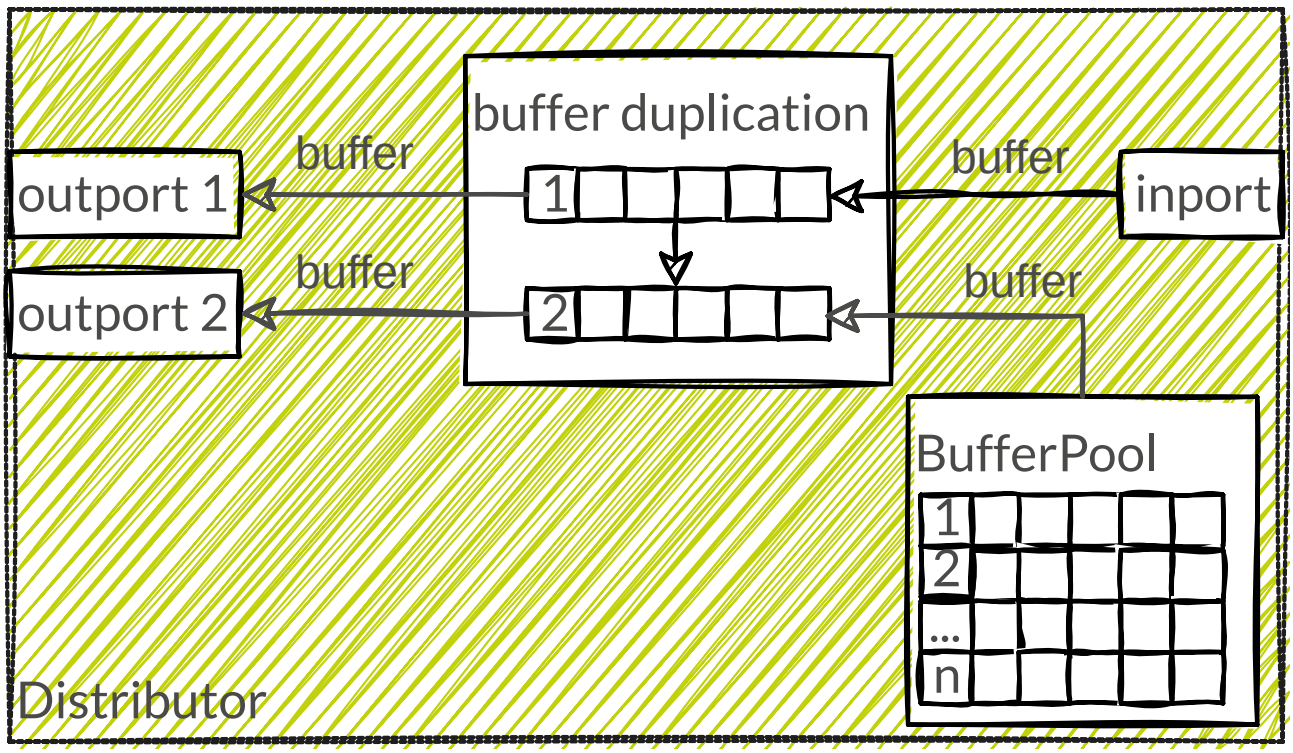


Figure 8: Data Distribution

The same data are going to be transmitted with HTTP and MQTT protocol. To let both, MQTT Publisher Endpoint (Section 2.2.7) and Http Server Endpoint (Section 2.2.8) consume a buffer created by the `JSONSerializer` a copy of this buffer needs to be created. Even for this a `BufferPool` is needed. Ideally the same `BufferPool` as for the Data Distribution is used, because both, source and destination buffer will have the same size.

```
// number of out ports
const unsigned OUT_PORTS = 2;

// component to multiply the serialized sample value,
// one inport to two outports
seve::eventflow::EventDistributor<OUT_PORTS, seve::lib::memory::Buffer*>
    distributor{pool};
```

Listing 14: Construction of `EventDistributor`

2.2.7 MQTT Publisher Endpoint

The MQTT Publisher Endpoint is responsible for publishing the MQTT messages. A MQTT message consists of a Topic and the message body.

The MQTT Topic is a combination of the `Application name` and `Element name` of the node. The combination is as follows: `/application_name/element_name`. Both, `Application name` and `Element name` can be configured with the web UI .

The MQTT Publisher Endpoint can handle one MQTT topic. It needs to be known at construction time of the MQTT Publisher Endpoint.

```
auto node_config = node_environment.get_node_config();

auto application_name = node_config.application_name.data();
auto element_name = node_config.source_interface_config.element_name.data
    ();

periCORE::network::MQTTTopic topic_publish(application_name, element_name
    );

periCORE::network::MQTTPublisherEndpoint
    mqtt_publisher_endpoint{topic_publish};
```

Listing 15: Construction of `MQTTPublisherEndpoint`

Each incoming `sample buffer` message triggers the MQTT Publisher Endpoint to try to send it to the configured MQTT Broker. When the MQTT Publisher Endpoint is connected to the broker the message is sent out and the buffer is deleted. If the connection to the broker is lost of course the message is not sent. In this case an incoming message is cached. The cache

memory has a size of one message. So each newly incoming message is overwriting the cache memory.

2.2.8 Http Server Endpoint

The Http Server Endpoint handles `GET` requests to the HTTP server. Therefore it is implemented as `CachedGetEndpoint`.

```
// endpoint to be read via HTTP GET method. it stores an serialized
   sample.
periCORE::implementation::network::httpserver::CachedGetEndpoint
    http_get_endpoint{};
```

Listing 16: Construction of `CachedGetEndpoint`

The Http server endpoint needs to be connected to the URL it is serving on:

```
// set rest url
http_get_endpoint.set_route("/sample");
```

Listing 17: `sample` route creation

Each `GET` request is going to be answered with a Sample datatype. In order to achieve this the periodically incoming messages are cached. Like at the MQTT Publisher Endpoint (Section 2.2.7) the cache memory has a size of one message, means each incoming message is overwriting the cache memory. So it is guaranteed that each `GET` request deliveres valid and actual data.

3 RESTFul API Extension

In addition to the RESTFul [1] API common for all nodes (for details please refer to the peri-CORE Datasheet [2]) the distance application provides the route `/sample`. Accessing it is defined as follows:

`GET` expects empty body.

`200` Return `Sample` object.

`204` Return empty body. No data is available.

`401` Unauthorized access, returns empty body

`500` Internal server error on unexpected error, returns empty body.

```
http get https://perinode-jj2br.local/sample
HTTP/1.1 200 OK
Connection: close
Content-Type: application/json

{
    "data": [
        {
            "distance": 19.7,
            "unit": "mm"
        }
    ],
    "incarnation": 3,
    "sequence_number": 8
}
```

Listing 18: Example `GET` request/response to `/sample` route

4 Web-based User Interface

After developing the distance application in the previous sections, the sensor values should be available in the HTTP endpoint that is connected to the HTTP server over the URI "/sample".

The goal in this section is to present the distance values in a user friendly interface customizing the webUI template delivered with the periCORE-SDK.

We will start with an overview of the structure of the template directory the webUI provides, followed by the explanation how to build and create the bundled container file to update the periCORE, and finally we will follow the steps to customize the interface that the developer can enhance afterwards.

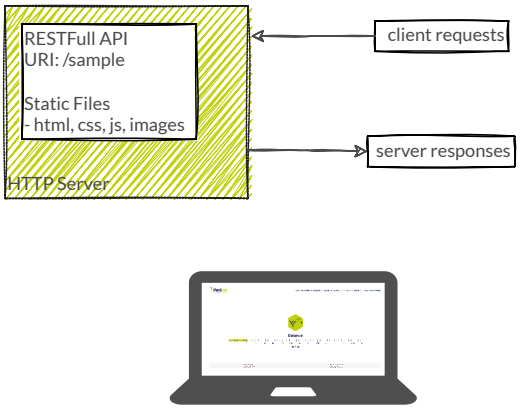


Figure 9: Overview webUI

4.1 Structure of webUI template

A HTTP server is a simple process receiving client requests and responding with the content of static files or errors messages according to the HTTP protocol. The clients, such as browsers or any other user agent, are receiving and processing the files to render the web pages and/or run scripts in the client side.

The hierarchy of the directory webUI will be the same stored in the periCORE flash, so refer to the `PATH` files accordinly.

The structure of the webUI folder presented in periCORE-SDK:

/: Directly in the root folder are the pre-configured html files working with the RESTFul API.

css: Style files defining the interface style like colors, fonts, sizes, etc.

css-node: Dedicated periNODE css files.

fonts: Font files to be used.

images: Image files png, jpg, etc.

js: Javascript files.

js-node: Dedicated periNODE Javascript files.

256KB are reserved to webUI Storage Data. When the files size exceeds this amount an error will inform the issue and the final update file won't be create.

During the compilation process, the resource files are minified as well as the images are optimized which reduces the size of the bundle.

4.2 Create the home page to show the distance value

In this section we will use the webUI template files to customize the home page to show the distance sample read by the sensor. The goal is to present the sensor value in a ruler that would be updated. The frequency is configurable within the configuration page of the webUI like explained in Section 4.5. The `html` files presented in template folder are mainly calling the `peri_base.js` script that will generate the page dynamically. In Listing 19 you can see the content of `home.html` file.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <link rel="shortcut icon" type="image/png" href="images/perinode.
png" />
    <script src="js/peri_base.js"></script>
</head>
<body onload="init()">
</body>
</html>
```

Listing 19: home.html content file

The `ruler.css` defines the style for our customized ruler that we will use to create our customized home page that will show updated measurements of the distance sensor.

Lets create a file called `home.js` that will be the script to create the home.html. For this let's define the space between the ruler steps and the number of steps to show we like to show in the ruler. We define here 20 and 15 respectively.

```
const label_step = 20;
const label_count = 15;
```

Listing 20: Define constants for creating the ruler

Define a function to provide dinamically the label for the DOM. It should receive a number that will be concatenated in the label and return the container. We will use the `class sunit` previously defined in the `ruler.css`.

```
function get_label_dom(num) {
    return `<div class='unit'>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <div class='sunit'></div>
    <label>` + num + `</label>
</div>`
}
```

Listing 21: Prepare the ruler

Now define a function that will create the step labels of the ruler. This is the main element in home page.

```
function dom_home() {
var labels = "";

for(var i = 0; i < label_count; i++)
{
    const val = 0 + label_step * i;
    const val_str = val.toString();
    labels += get_label_dom(val_str);
}
```

Listing 22: Create the Ruler

Now to create the ruler, use the function created before to place the labels between the steps. The initial value is defined as *50mm* as you can see in Listing 23:

```
    return `<section>` + get_perinode_image_figure() + `<h1>Distance</h1>
    <div class='ruler'>
        <div id='bar'></div>
        <div>` + labels + `
        <div class='unit'>
            <label>` + (label_step*label_count).toString() + `</label>
        </div>
        </div>
    </div>
    <br>
    <h2 id="value">50mm</h2>
    </section>`;
    }
```

Listing 23: Define the ruler section

To create a bar that moves and shows the distance over the ruler, create the function called `draw_distance` shown in Listing 24. Some calculation is needed here and a message "out of range" is configured to be shown in case of no valid value received.

```
    function draw_distance(sample) {
        var value = sample.distance;
        var min = 0;
        var max = label_step * label_count;
        if (min > value) {
            value = "out of range"
            $("bar").style.width = ((0) * 100) / (max - min) + "%";
            $("value").innerHTML = value;
        }else if (value > max){
            value = "out of range"
            $("bar").style.width = ((max - min) * 100) / (max - min) + "%
";
            $("value").innerHTML = value;
        } else {
            $("bar").style.width = ((value - min) * 100) / (max - min) +
"%";
            $("value").innerHTML = value + " " + sample.unit;
        }
     }
```

Listing 24: Draw Distance

Prepare a function that parses a JSON, that will be the response coming from the API and draws the sample read in the previously created function (`draw_distance(sample)`).

```
    function set_sample(_http) {
        var sample = JSON.parse(_http.responseText);
        var sample = sample.data[0];
        draw_distance(sample);
    };
```

Listing 25: Function set_sample

Use the provided class `reconnection_update` and use the method `overload_update` to update the same. Take a look on its definition:

```
overload_update(uri, interval, callback)
```

Listing 26: overload_update definition

Note: The `reconnection_update` is provided by `peri_base.js`

This method receives the URI that it will send a `GET` request every `interval` of miliseconds and when it receives the response the `callback` function will be called.

In order to have the sample up-to-date each second, define a function called `reload\_home` which uses the provided `reconnection_update` as explained before. The function to update the ruler will look like this:

```
    function reload_home() {
        $("content_id").innerHTML = dom_home();
        document.title = "periNODE Web Server | Distance";
        reconnection_update.overload_update("/sample", 1000, set_sample);
    }
    load_css_script("css_node/ruler.css");
    update_content("home", reload_home);
```

Listing 27: Reload the home.js to update the sample

As a result of the page you can see the page as shown in Figure 10:

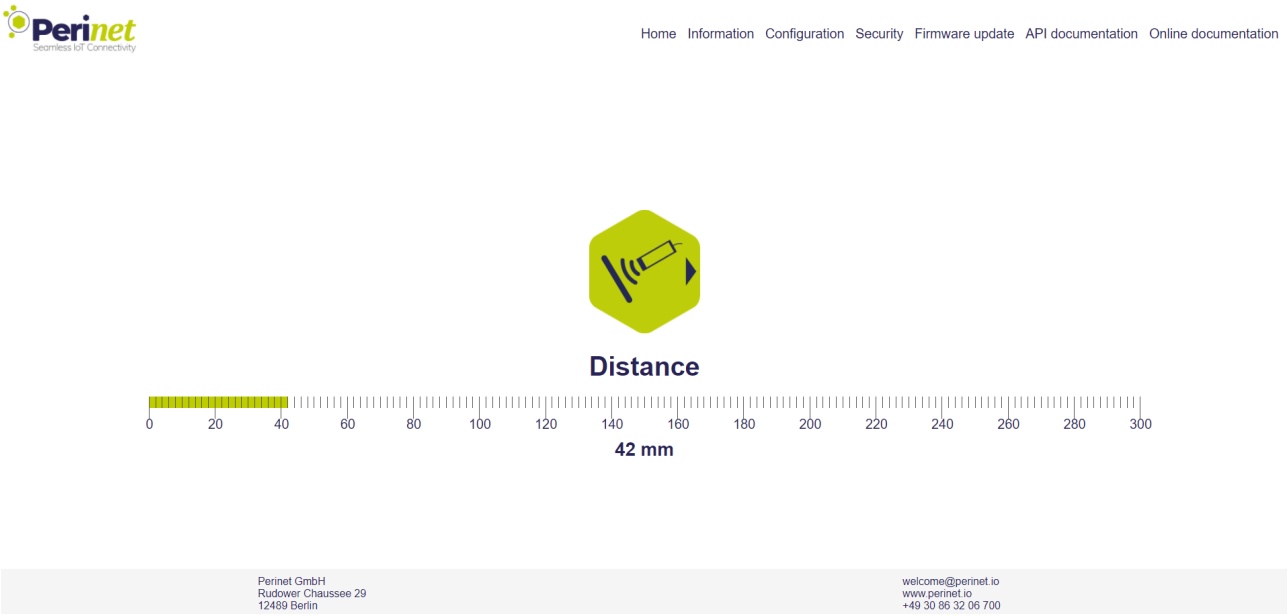


Figure 10: Example of distance sensor home page

The intension here is to exemplify the usage of the template. The developers can customize the style, change the images, create new elements and functions as they prefer.

4.3 Customize the webUI

As said before the template is up to be changed and here we will focus on customizing the home page just as an example that can be followed to customize other files as well.

4.3.1 Changing the image in the header

In order to have your own identity you can change the file `peri_base.js`. To change the Perinet logo for instance, for this refer to function `dom_header()`, change HTML image element's `src`-tag that is by default set to the file `images/perinet-logo.png`.

```
    function dom_header(){
        return `<header class="header">
        <a href="https://www.perinet.io" target="_blank" class="logo"
> <img src="images/perinet-logo.png" alt="Perinet Logo" width="180"></
a>
        <input class="menu-btn" type="checkbox" id="menu-btn" />
        <label class="menu-icon" for="menu-btn"><span class="nav-icon
"></span></label>
        <ul class="menu">
            <li><a href="home.html" onclick="return reload('home')">
Home</a></li>
            <li><a href="info.html" onclick="return reload('info')">
Information</a></li>
            <li><a href="config.html" onclick="return reload('config'
)">Configuration</a></li>
            <li><a href="security.html" onclick="return reload('
security')">Security</a></li>
            <li><a href="update.html" onclick="return reload('update'
)">Firmware update</a></li>
            <li><a href="doc/api.proto" target="_blank">API
documentation</a></li>
            <li><a href="https://docs.perinet.io" target="_blank">
Online documentation</a></li>
        </ul>
        </header>`;
    }
```

Listing 28: Change the Perinet logo in peri_base.js

Following the peri_base.js file you can also customize the company information in the `dom_footer()` function.

```
function dom_footer(){
return `<footer>
<address>
    <table>
        <tr>
            <td>
                <p translate="no">
                    Perinet GmbH<br>
                    Rudower Chaussee 29<br>
                    12489 Berlin
                </p>
            </td>
            <td>
                <p translate="no">
                    <a href="mailto:welcome@perinet.io">
welcome@perinet.io</a><br>
                    <a href="https://perinet.io">www.perinet.io</a><
br>
                    +49 30 86 32 06 700
                </p>
            </td>
        </tr>
    </table>
</address>
</footer>`;
}
```

Listing 29: Changing the footer

4.4 Compile and update

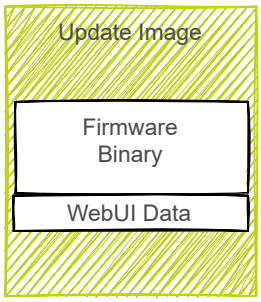


Figure 11: Update image bundle

The webUI folder stores the resource files provided by HTTP when it receives requests from the clients. To make it possible this files should be stored in periCORE flash memory in advance and for this it needs to be pre-processed.

The periCORE-SDK build process is prepared to pre-process as well as it is able to create the image that is supposed to be used to update the periCORE which contains the firmware binary

and the webUI resources you can see in Figure 11.

The task to build an `update` image is available on the vscode workspace and it will generate the file called: `update_periNODE_distance-periCORE4-armv7_r-14.0.update.img.signed`. This file contains the application firmware binary as well as the webUI data.

Note: The space dedicated to the webUI storage is limited to 256KB. A build error should be throw when the space would be overloaded.

After generating the update file it is necessary to store it in the periCORE before any changes in the webUI will be applied. Remember that the files are static and the periCORE takes the resoures from the flash, and it is not possible to debug the webUI.

```
curl --data-binary '@<filename>' \
-H "Content-Type: application/octet-stream" \
-X PUT https://periCORE-serno.local/update
```

Listing 30: Curl command to update the periCORE

Please refer to the periCORE Datasheet[2] to get more information about how to update regarding the periCORE life cycle.

4.5 Configuration Values and OTP data

The templates to work with default configuration values and OTP data are also available and works like the home page presented in the previous sections. There is a `config.html` which calls the corresponding `config_base.js` which will construct the elements of the page and can be extended as needed.

Please refer to periCORE Datasheet[2] to get more information about the complete RESTFul API.

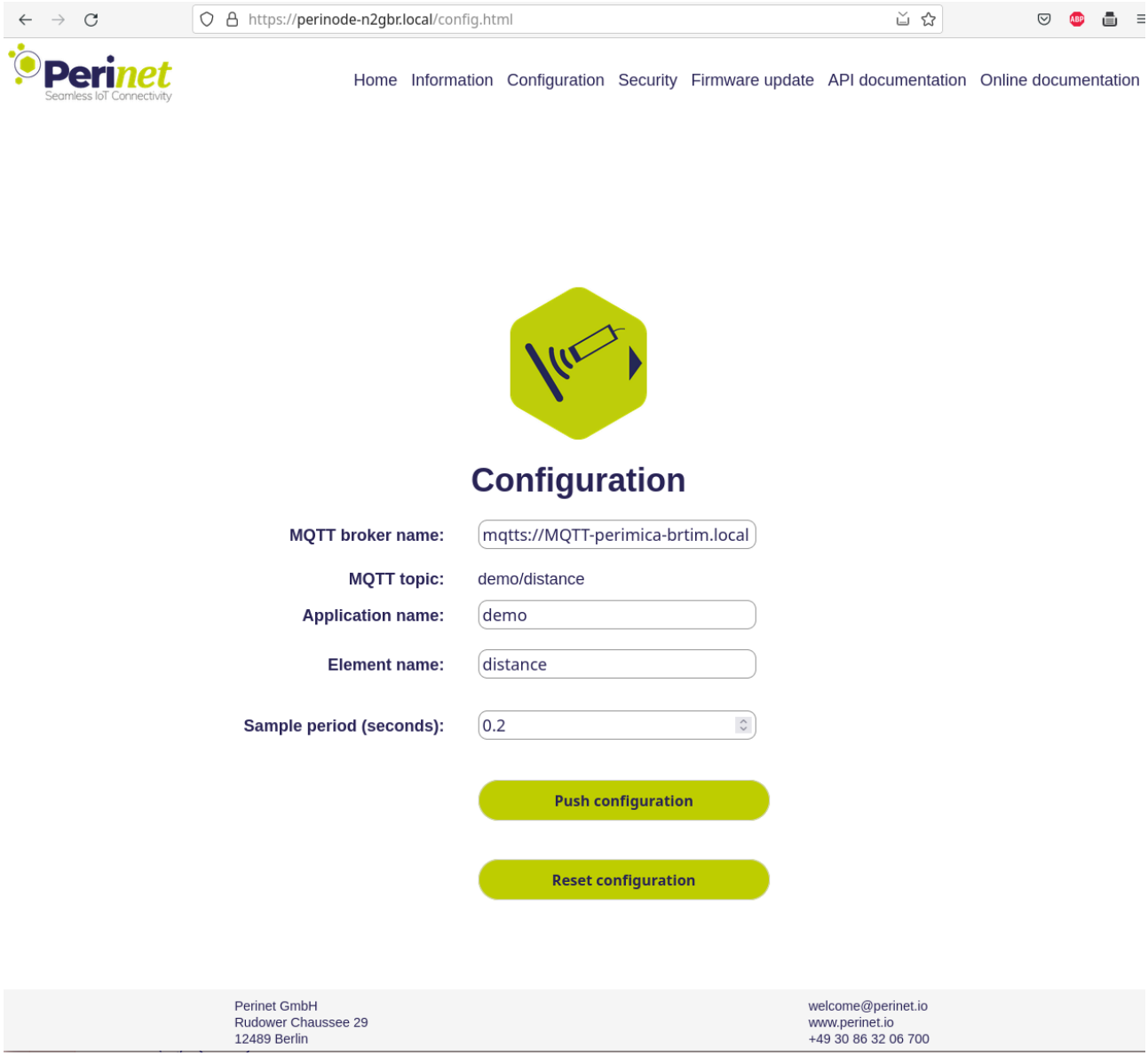


Figure 12: Example of distance sensor configuration page

4.6 Recommended Browsers

- Windows Edge
- Safari

Note: Chrome based browsers cannot resolves IPv6 link local names and that is why it is not considered compatible with periCORE based products.

Note: Firefox has a known issue compatibility with the update process for periCORE and until this issue is not solved we also don't recommend to use it.

5 Contact & Support

For customer support, please call us at **+49 30 863 206 701** or send an e-mail to *support@perinet.io*.

For complete contact information visit us at www.perinet.io

A List of Figures

B List of Listings

C Glossary

A/D conversion Analog Digital conversion. 7

adc Analog Digital converter. 7–9, 30

API Application Programming Interface. 6, 10, 17, 18, 21, 26

css Cascading Style Sheets. 18

DOM Document Object Model. 19

Fsr Full Scale Range. 9

GNU GNU's Not Unix. 4, 5

GPIO General-Purpose Input/Output. 7

HTTP Hypertext Transfer Protocol is an application-layer protocol for transmitting hypermedia documents, such as HTML. 7, 14–16, 18, 25

I2C Inter-Integrated Circuit is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial bus. 7, 9–11

IDE Integrated Development Environment. 4

IIoT Industrial Internet of Things. 3, 4

IPv6 Internet Protocol Version 6 [9], a communication protocol. 27

js JavaScript. 18

JSON JavaScript Object Notation is standard text-based format for representing structured data based on JavaScript object syntax. 21

Makefile Definition of set of tasks to be executed by GNU make tool. 5

MQTT Message Queuing Telemetry Transport is a lightweight, publish-subscribe based network protocol that transports messages between devices. 7, 14, 15

OTP One-Time Programmable memory. 2, 26

PCBA Printed Circuit Board Assembly is a PCB populated with electronic components. 3

REST REpresentational State Transfer, a web API style. 17

SDK Software Development Kit is a colletion of development tools. 4, 18, 25

UART A Universal Asynchronous Receiver-Transmitter is a computer hardware device for asynchronous serial communication. 7

UI User Interface. 2, 8, 15, 18, 19, 25, 26, 29

URI Uniform Resource Identifier. 18, 22

URL Uniform Resource Locator. 4, 16

D References

[1] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. http://www.rfc-editor.org/rfc/rfc7231.txt. RFC Editor, June 2014. URL: http://www.rfc-editor.org/rfc/rfc7231.txt.

[2] Perinet GmbH. periCORE Datasheet. PRN.100.375. https://docs.perinet.io/PRN100375-periCOREDatasheet.pdf.

[3] Perinet GmbH. periCORE Development Kit Setup Application Note. PRN.100.376. https://docs.perinet.io/PRN100376-periCOREDevelopmentKitSetupApplicationNote.pdf.

[4] Perinet GmbH. periCORE Development Kit User Guide. PRN.100.378. https://docs.perinet.io/PRN100378-periCOREDevelopmentKitUserGuide.pdf.

[5] Perinet GmbH. sève Operating System Datasheet. PRN.100.377. https://docs.perinet.io/.

[6] Perinet GmbH. Smart Components Datasheet. PRN.100.387. https://docs.perinet.io/PRN100387-SmartComponentsDatasheet.pdf.

[7] GNU. *GDB: The GNU Project Debugger*. URL: https://www.sourceware.org/gdb.

[8] GNU. *GNU Make*. URL: https://www.gnu.org/software/make/.

[9] R. Hinden and S. Deering. *IP Version 6 Addressing Architecture*. RFC 4291. http://www.rfc-editor.org/rfc/rfc4291.txt. RFC Editor, Feb. 2006. URL: http://www.rfc-editor.org/rfc/rfc4291.txt.

[10] Texas Instruments. *ADS1115*. URL: https://www.ti.com/product/ADS1115.

[11] Sensorshop24. *ULK-WP-M12-V*. URL: https://www.sensorshop24.de/productpdf/download/file/id/2088/name/Ultraschallsensor_mit_M12_Steckverbindung_und_Regulierung_durch_Teach-In-Knopf_-_0-10V_-_Schaltabstand_300mm.pdf/.

E Revision History

Revision	Date	Author(s)	Description
1	2024-01-30	tkla, show, kwal	Initial Release