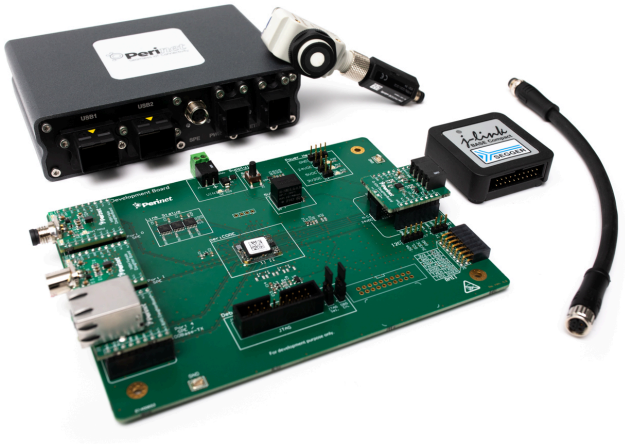
Application Note

Developing Firmware for the periCORE Module



periCORE Development Kit

Perinet
Seamless IoT Connectivity

Abstract

This application note guides developers and engineers in the application development for the *MikroE SHT click sensorboard* that is part of the *periCORE Development Kit*. It is formulated as a step-by-step guide, starting with an existing application and going through the quick and easy firmware adaptation and development to the point of production ready firmware displaying measured information on the *periCORE WebUI*.

Doc-No.: PRN.100.646 rev: 2 2023-09-06

Document Information

Title	Application Note
Subtitle	Developing Firmware for the periCORE Module
Type	periCORE Development Kit
Status	Release
Version	2
Date	2023-09-06
Disclosure Restriction	

Contents

1 Introduction

This application note provides a comprehensive guide to developing firmware with the periCORE Development Kit. The hardware used to showcase the firmware development process is the *SHT click board*, which is part of the periCORE Development Kit. The board contains a temperature and humidity sensing IC and is equipped to communicate digitized sensor values via I2C.
The starting point for the firmware development process is the *periNODE distance application* included in the periCORE Development Kit. This guide is divided into several sections:
The first section provides a brief overview of both the *periNODE distance application* and the *periCORE SHT application*.
The second section introduces concepts of developing firmware with the periCORE Development Kit by exemplified firmware implementation. This section includes terminology and recommended sequences of development steps. The third section demonstrates the finalization of the firmware development process including adaptation of the webUI.
By the end of this application note, developers will have a basic understanding of firmware development for the periCORE communication module. Code samples and comprehensive visual software representations facilitate firmware development for custom projects.

2 Overview

This section provides brief introductions into the software architecture of the periNODE-distance application and the periCORE-SHT application. Furthermore, it introduces terminology and concepts referenced in this guide.

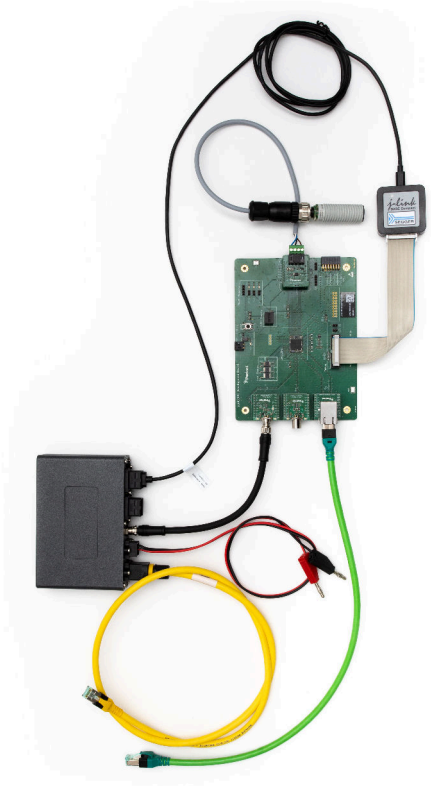


Figure 1: Assembled periCORE Development Kit

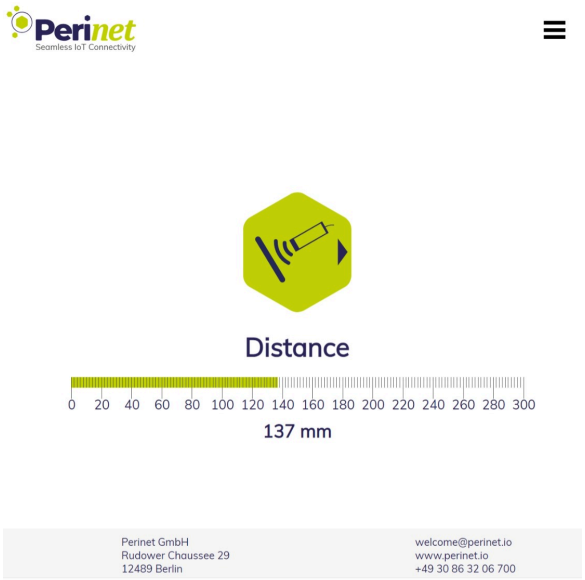


Figure 2: Dashboard of webUI hosted by periNODE-distance application

An assembled periCORE Development Kit is shown in Figure 1. A distance sensor is connected to the periCORE Development Board. Additionally, the debugger connected together with the periMICA edge computer will serve as the debugging device. On the network side, the periMICA converts 100BASE-T1 (Single Pair Ethernet) to 100BASE-TX (Fast Ethernet). The periCORE hosts a webserver, of which the homepage can be seen in Figure 2. If this setup is not working as shown, please refer to the periCORE Development Kit Setup Guide [2].

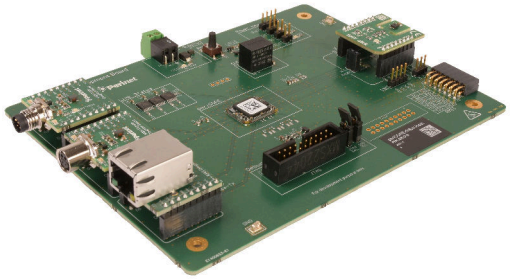


Figure 3: Development Board with SHT click board plugged into

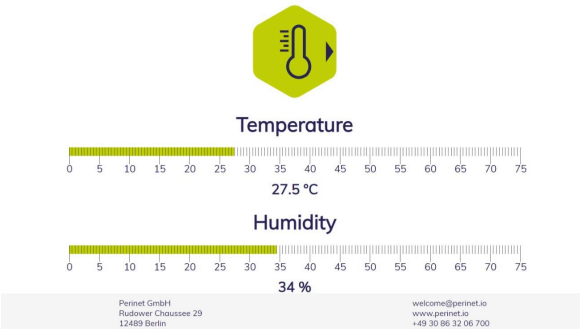


Figure 4: Homepage of webUI hosted by periCORE (SHT application)

The SHT click board plugged into the Development Board is shown in Figure 3. Both hardware modules are part of the Development Kit. The homepage of the website hosted by the periCORE is shown in Figure 4 and displays information measured by the SHT click board. It shows the targeted result of this guide.

2.1 periCORE-specific vocabulary

Some vocabulary specific to the periCORE development environment will be introduced here:

- component: software block with defined interfaces, represents one specific hardware module or software function
- port: interface of software component
- inport: interface for receiving data
- outport: interface for sending data
- event: reception of data on an inport
- event channel: connection between an inport and an outport

2.2 periNODE-distance firmware

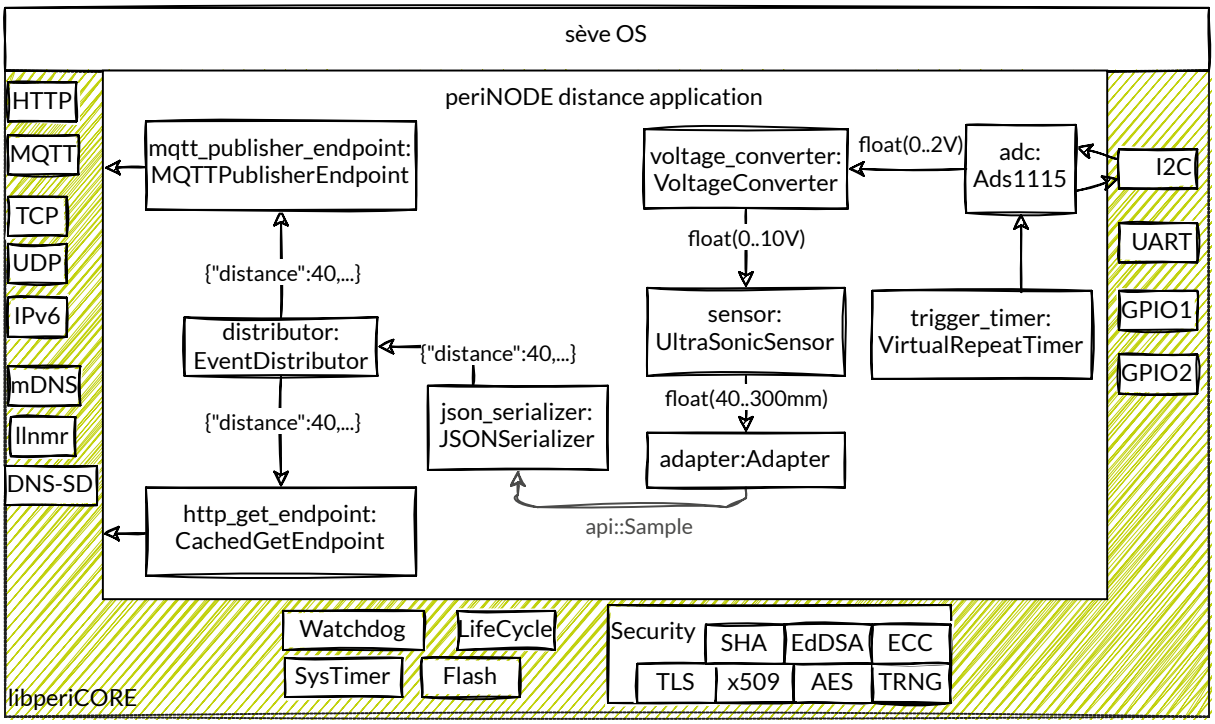


Figure 5: eventflow diagram, periNODE-distance firmware

This section references the eventflow diagram of the periNODE distance firmware shown in Figure 5. The `trigger_timer` component periodically generates an event for the `adc` component. The event causes the `adc` component to start to read the sensor value by initiating an I2C communication.
At the end of the communication sequence, measured sensor data is received by the `adc` component. The `adc` component will convert the register value, which represents the analogue 0-2V received by the signal pin of the ADC IC, into a float type and will send it to the `voltage_converter` component. Here, the 0-2V value is converted into a 0-10V value and sent to the `sensor` component. The `sensor` component computes a distance value in mm and sends it to the `adapter` component.
The `adapter` component creates a struct of value(float-type) and unit(string-type) and appends meta information. The `json_serializer` component receives information as `sample` type from the `api` component. The `json_serializer` creates a JSON string with the provided information. The `distributor` component duplicates the JSON string and sends it to the `mqtt_publisher_endpoint` and `http_get_endpoint` components. The `mqtt_publisher_endpoint` component sends an MQTT message to an MQTT broker, if available. The `http_get_endpoint` component cashes the message and answers to HTTP GET requests.

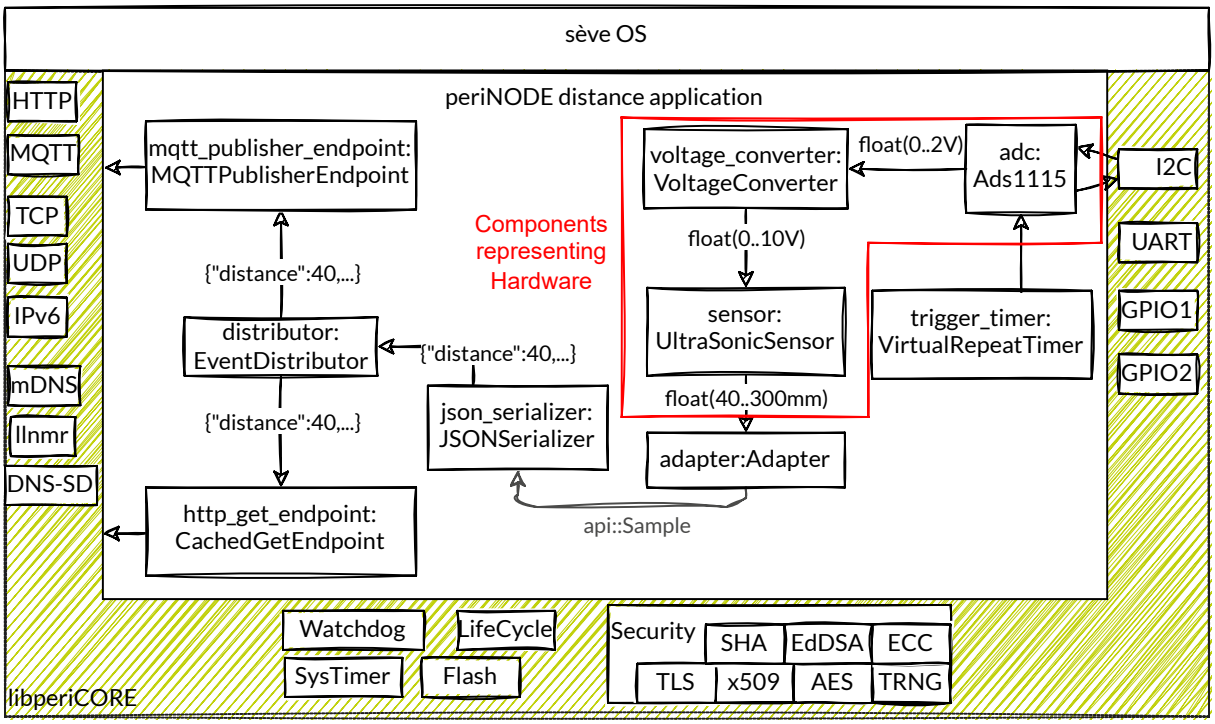


Figure 6: eventflow diagram, periNODE-distance firmware, components representing hardware are highlighted

Components highlighted in Figure 6 represent hardware modules: The `Ads1115` (ADC) and `VoltageConverter` are part of the 0-10V sensor daughterboard. The `ULK-WP-M12-V` is the sensor connected to the periCORE Development Board via 0-10V sensor daughterboard. With this approach, connecting a different 0-10V distance sensor only requires changing the `sensor` component with all other existing components remaining the same. When connecting a different type of sensor, e.g. a CO_2 sensor, the `sensor` and `adapter` component as well as the `Sample` type need editing. The changes are solely the adaptation of the `unit` string, as the measured quantity is different. No other changes are necessary to receive and display information. Additionally, we recommend a webUI adaptation to display the information sensibly.

2.3 periNODE-SHT firmware

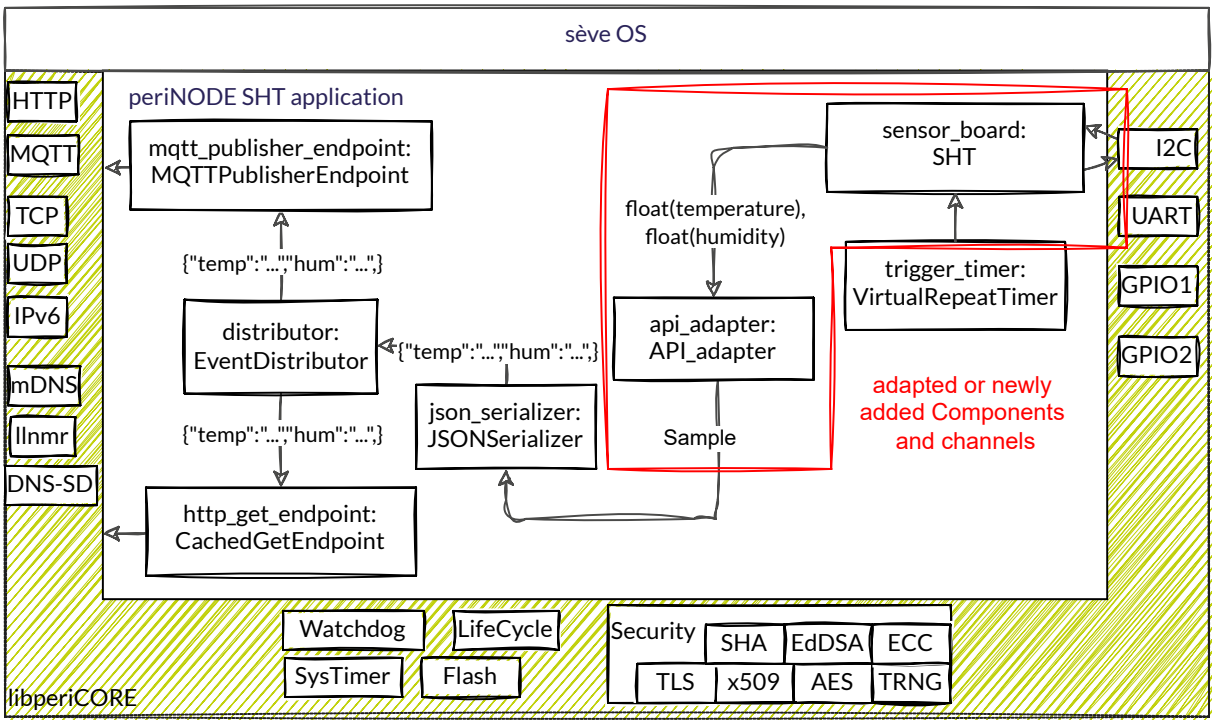


Figure 7: eventflow diagram, periCORE SHT firmware

The eventflow diagram shown in Figure 7 gives an overview of the final periNODE-SHT firmware. Components and event channels within the marked area differ from the periNODE-distance firmware. The `sensor_board` component represents the hardware component SHT click board. It communicates with the I2C component provided by libperiCORE. The `api_adapter` component differs from the `adapter` component of the periNODE-distance firmware shown in Figure 5. The `Sample` type (event channel's data type between `api_adapter` and `json_serializer`) is adjusted according to the measured information.

2.4 Before starting

Before starting the firmware implementation process, make sure all points listed here are checked:

- Visual Studio Code installed and updated
- Development Board set up and working
- Docker and Dev Container installed
- periNODE-distance application compiling and debuggable
- periCORE webserver accessible

If one or more of these points are not met, please refer to
periCORE Development Kit Setup Application Note [1].

2.5 Renaming the project

Naming the project according to the targeted application can prevent confusion, especially when working on more than one project. This is done by some simple adjustments.

Changes in multiple files will be presented in one code block here:

```
In "Makefile" (top-level project folder):
...
# TODO: update firmware application name
APP_NAME = periNODE-SHT
...

In "src/defaultNodeInfo.cc":
...
{//version_info
    {FIRMWARE_VERSION, FIRMWARE_API_VERSION, FIRMWARE_BUILD_VERSION},
    { "SHT" }, // will be shown on Web UI
}
...

In "src/defaultNodeConfig.cc":
...
{""} //application_name
...
perinet::periNODE::api::InterfaceType::TEMPERATURE_SOURCE //type
...
,{""} //element_name
...

In ".vscode/settings.json":
...
"APP_NAME" : "periNODE-SHT",
...

In "webui/fs_root/js/update.js":
...
// accepted names for firmware updates on the update page of the webUI
const firmware_variants = ['distance','0-10V','SHT']
...
```

Listing 1: Renaming project, multiple files

3 SHT component

The visual representation of the fully implemented `SHT` component is shown in Figure 8.

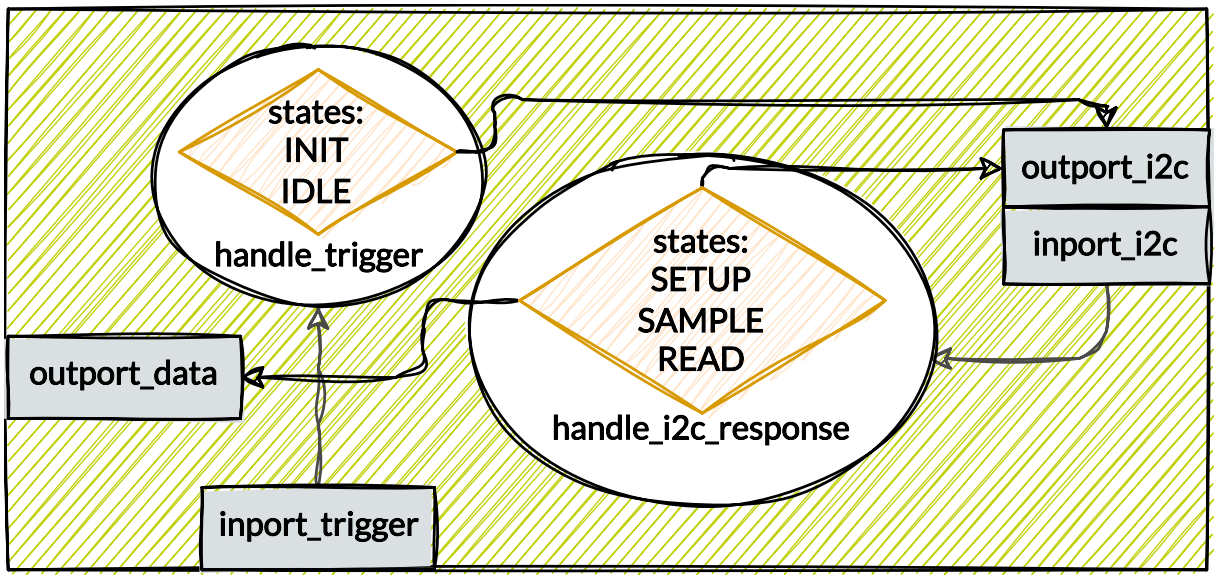


Figure 8: Visual representation of the SHT component

The first step of implementation will be adding an `SHT` component (called `sensor_board`) (Section 3.1). From there, implementation will be described sequentially.

3.1 Add files to project's file system

The folder in `src` named `platform` contains all components of this application that represent hardware. Add two files (`source, header`) into the `platform` folder. Name the files according to the component's name in the eventflow diagram (`SHT`).

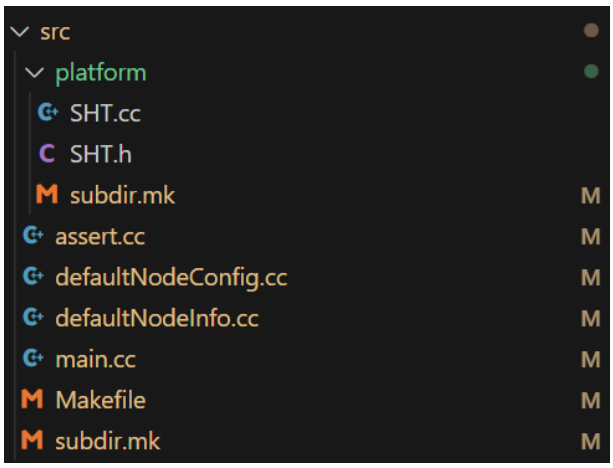


Figure 9: Screenshot of partial file system with added files

Add the class syntax to `SHT.h` (Listing 2).

```
#pragma once
namespace platform
{
class SHT
{
public:
private:
};
}
```

Listing 2: Initial SHT header file, SHT.h

Include the `SHT.h` file into `main.cc`, remove `periNODE_distance firmware` related includes and instantiations as well as channels. Instantiate an object of the SHT class.

```
...
// removed includes
// #include "api/Adapter.h"
// #include "platform/UltraSonicSensor.h"
// #include "platform/ADS1115.h"
// #include "platform/VoltageConverter.h"

// newly added includes
#include "platform/SHT.h"

...
// newly added instantiations
platform::SHT sensor_board;

// removed instantiations
// platform::ADS1115 adc{0x48};
// platform::oltageConverter voltage_converter{};
// platform::ULK_WP_M12_V sensor;
// api::Adapter adapter;
// periCORE::lib::JSONSerializer<api::Sample> json_serializer{pool};

...
    // removed channels within main():
    // adc.set_outport_i2c(node_environment.peripherals.i2c.get_inport_bus());
    // adc.set_outport_measurement(voltage_converter.get_inport_adc());
    // voltage_converter.set_outport_voltage(sensor.get_inport_voltage());
    // trigger_timer.set_outport(adc.get_inport_trigger());
    // sensor.set_outport_distance(adapter.get_inport_distance());
    // adapter.set_outport_sample(json_serializer.get_inport());
    // json_serializer.set_outport(&distributor);
    ...
```

Listing 3: Changed code, commented lines show what is removed, main.cc

Delete all files from the `platform` folder except the newly added ones. Additionally, delete all `.cc, .h` files that are no longer included in `main.cc`. Note that in every folder, a `subdir.mk` file must exist. Any `subdir.mk` file within the project's file system can be used as is. Compile the project.

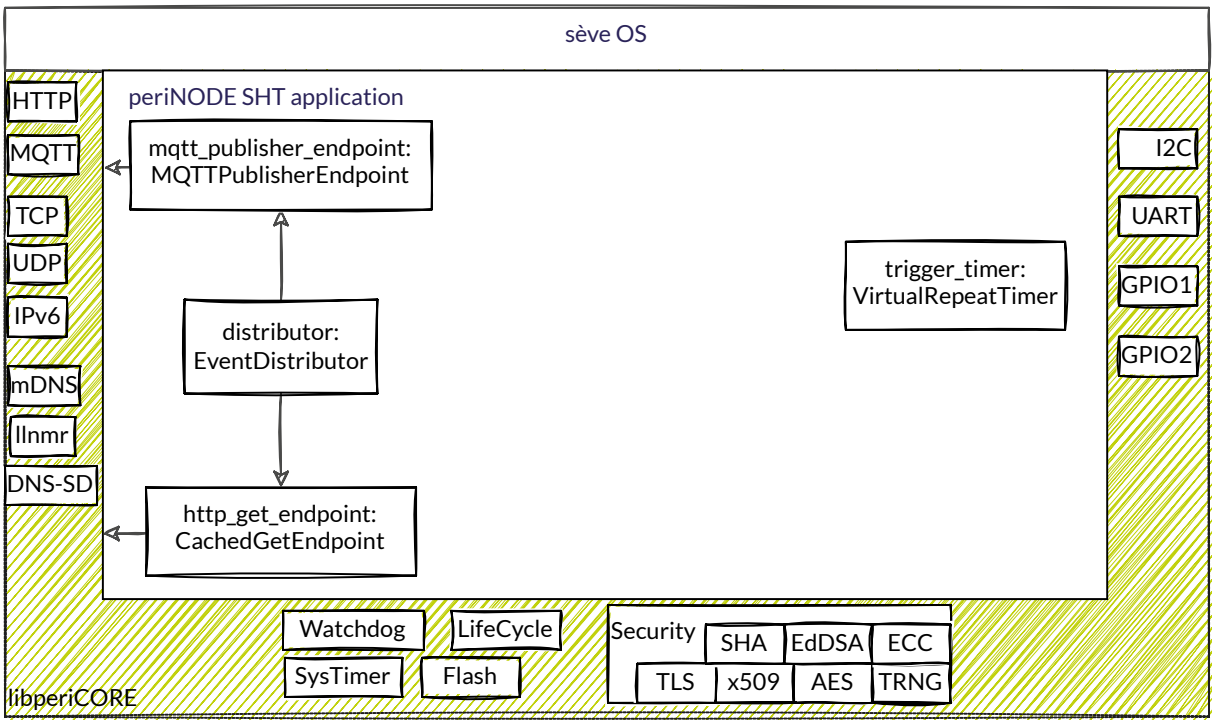


Figure 10: Eventflow diagram, removed components, added sensor_board (Listing 3)

3.2 Add trigger inport to the SHT component

Declare ports in the `private` section of the `SHT.h` file. Declare the getter-methods in the `public` section of the `SHT.h` file. Every inport must have an associated event handler method. Declare this method in the `private` section of the `SHT.h` file.

```
#pragma once
#include "seve/eventflow/SingleValue.h"
namespace platform
{
class SHT
{
public:
    seve::eventflow::Sink<>* get_inport_trigger() { return &inport_trigger; };
private:
    seve::eventflow::SingleValue<SHT> inport_trigger {this, &SHT::
    ↪ handle_trigger};
    // event handler methods
    void handle_trigger();
};
}
```

Listing 4: Declaring an event handler method associated to an inport, SHT.h

We used a `SingleValue` inport here. A `SingleValue` inport will store exactly one value of its defined data type. In case of the trigger inport, the data type is empty. Upon reception of a trigger event, the associated event handler method is scheduled.
Define the `handle_trigger` method in the `SHT.cc` file.

```
#include "platform/SHT.h"

using namespace platform;
void SHT::handle_trigger() {};
```

Listing 5: Defining handle_trigger associated to inport_trigger, SHT.cc

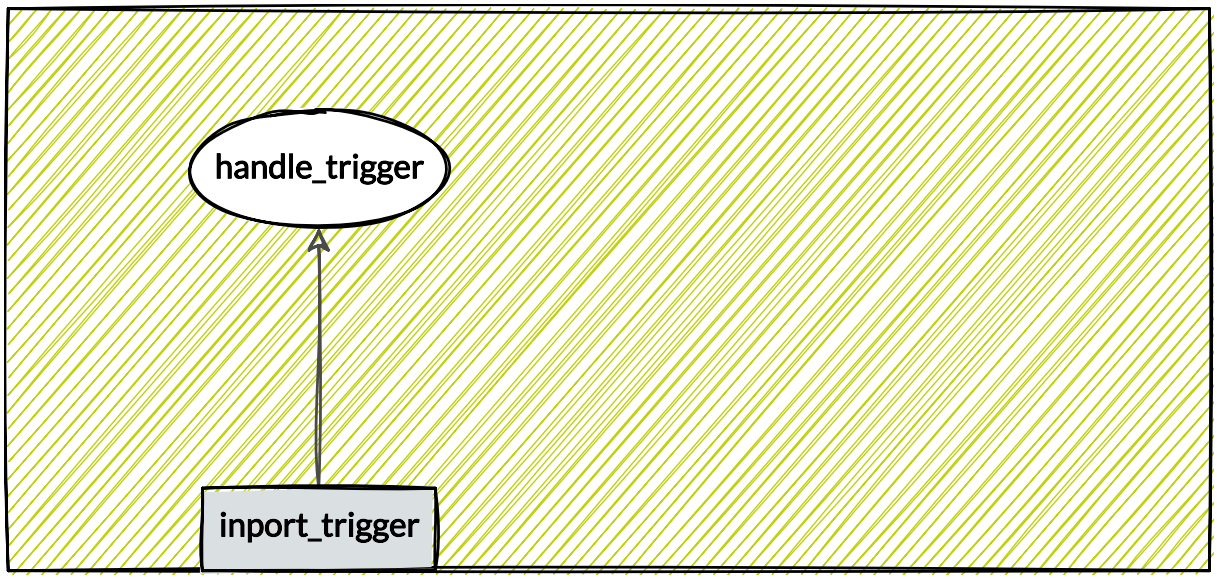


Figure 11: sensor_board's visual representation with added trigger inport and corresponding event handler method (Listing 4)

Create an event channel between the `trigger_timer`'s outport and the `sensor_board`'s inport in `main.cc`. This is done by calling the `trigger_timer.set_outport` method with the parameter being the `sensor_board.get_inport_trigger` method.
Note that ports must be connected before the `sevecore.start();` line.

```
...
int main() {
    ...
    // channel between trigger_timer's outport and sensor_board's trigger
   ↪ inport
    trigger_timer.set_outport(sensor_board.get_inport_trigger());
...
```

Listing 6: Implementing an event channel from trigger_timer's outport to sensor_board's inport_trigger, main.cc

The `trigger_timer` will send a message to the `sensor_board`'s trigger inport every second. The `handle_trigger` event handler method will be executed when all previously scheduled methods have run to completion *(run-to-completion-semantic, FIFO)*.

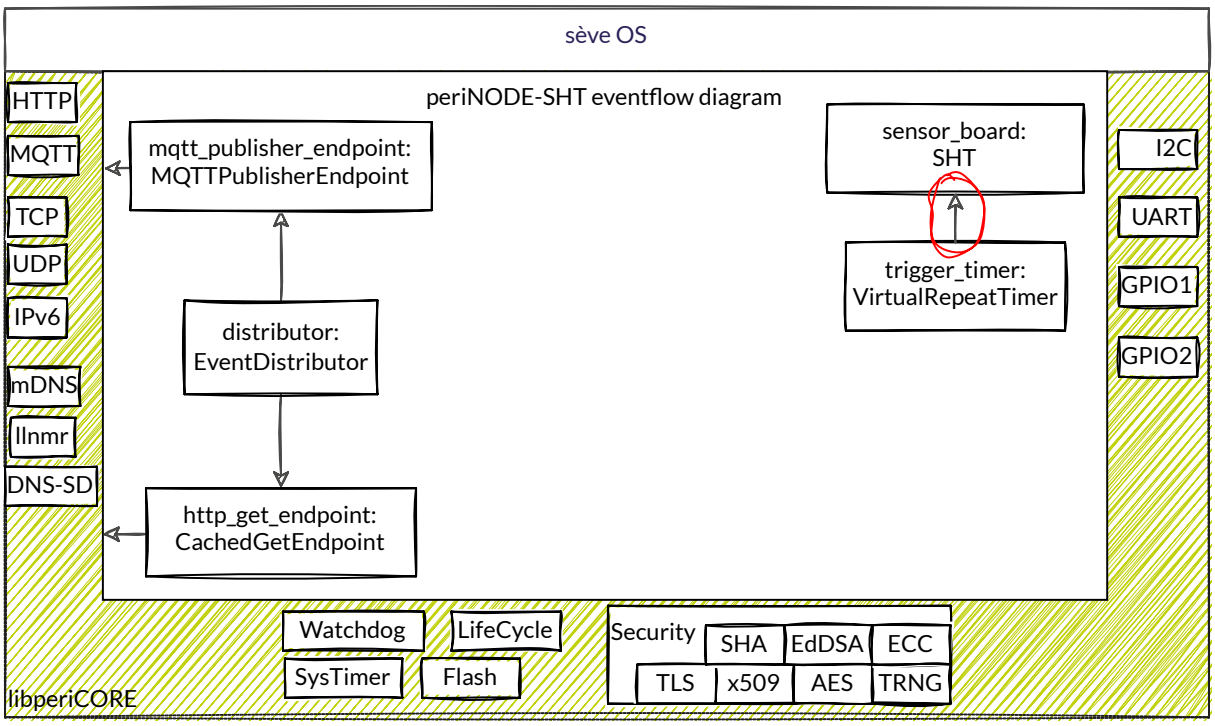


Figure 12: Eventflow diagram with added trigger_timer event channel (Listing 6)

Stop here to compile and debug. Set a breakpoint at the `handle_trigger` method in `SHT.cc`. You should witness the debugger stop here.

3.3 I2C communication implementation

Declare an object of type `periCORE::peripherals::i2c::Descriptor` as an attribute of the SHT class. It expects an array of data type `uint_8` as it's initialization parameter. Include `NodeEnvironment.h`.

```
#include "periCORE/NodeEnvironment.h"
...
private:
    ...
    // i2c buffer
    uint8_t i2c_data[7];
    periCORE::peripherals::i2c::Descriptor i2c_descriptor {i2c_data};
...
```

Listing 7: Allocating memory as I2C message buffer, SHT.h

The I2C component uses an I2C descriptor for operation. This descriptor points to memory in which the transferred data is stored. Because the operating system has no dynamic memory allocation (due to lack of memory), the `sensor_board` must allocate memory required for I2C communication statically.
In Listing 7, the allocation of data memory (`i2c_data`) as well as the descriptor (`i2c_buffer`) is shown. The maximum size of transmitted data is 7 bytes for the SHT IC.

Set up the `sensor_board`'s I2C communication infrastructure by adding ports and an event channel. Declare the `get_inport_i2c` and `set_outport_i2c` methods when declaring in- and outport.
As always, an event handler method is required when declaring an inport.

```
...
public:
...
    void set_outport_i2c( seve::eventflow::Sink<periCORE::peripherals::i2c::
    ↪ Descriptor*>* outport) { outport_i2c = outport; };
    seve::eventflow::Sink<periCORE::peripherals::i2c::Descriptor*>*
    ↪ get_inport_i2c() { return &inport_i2c; };
private:
...
    seve::eventflow::Sink<periCORE::peripherals::i2c::Descriptor*>*
    ↪ outport_i2c{nullptr};
    seve::eventflow::Queue<SHT, periCORE::peripherals::i2c::Descriptor>
    ↪ inport_i2c {this, &SHT::handle_i2c_response};
...
void handle_i2c_response(periCORE::peripherals::i2c::Descriptor* rx);
};
}
```

Listing 8: Adding I2C ports, SHT.h

The `Queue` inport template can buffer multiple messages. We used it here to guarantee no message loss in the I2C communication.
The `sensor_board`'s `inport_i2c` will receive a response from the I2C bus driver Define the associated event handler method.

```
...
void SHT::handle_i2c_response(periCORE::peripherals::i2c::Descriptor* rx) {};
```

Listing 9: Defining handle_i2c_response, SHT.cc

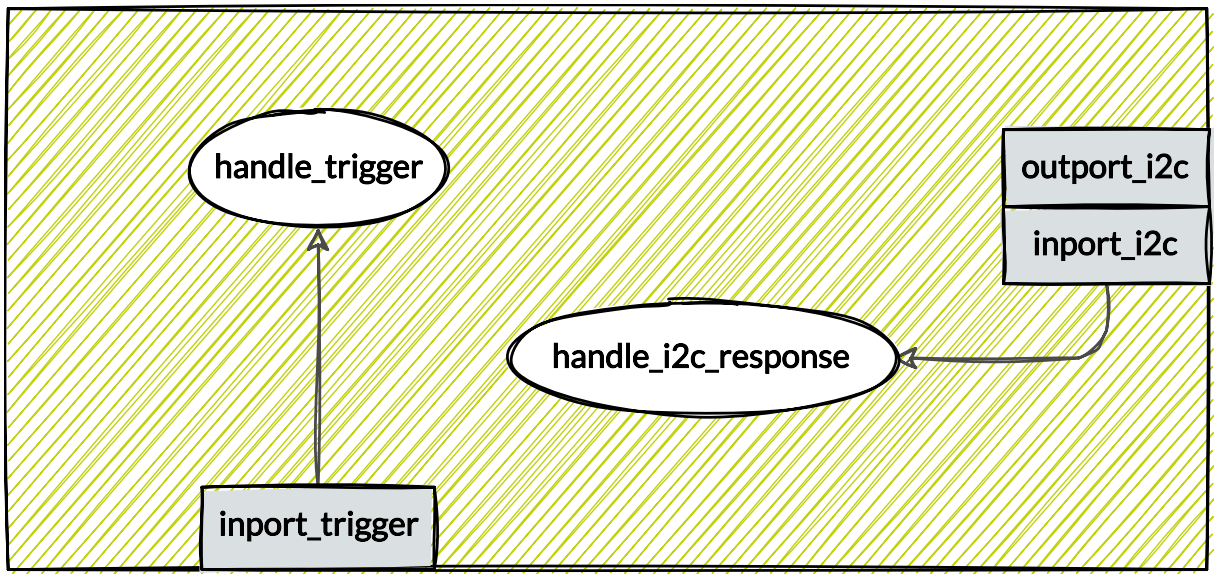


Figure 13: Visual representation of sensor_board's structure with added I2C ports (Listing 8)

3.4 State machine

The I2C communication sequence consists of three messages. The first message initializes the SHT IC to function in *Periodic Measurement Mode*. The SHT IC will then continuously measure temperature and humidity. The second message initiates the transmission of measurement data. The third message is a *dummy write*. It clocks the I2C bus to receive measurement data. The three I2C responses to these messages have to be distinguished. As every inport can only have one associated event handler method, it is necessary to implement a state machine.

Additional to the three I2C states, a state for the very first `inport_trigger` event is required to initialize the SHT IC's I2C interface. In sum, declare five states.

```
...
private:
    ...
    // state variables
    enum STATES { INIT, SETUP, IDLE, SAMPLE, READ };
    STATES state = INIT;
...
```

Listing 10: Declaring state variables, SHT.h

```
...
void SHT::handle_trigger() {
    switch(state) {
    // initialize the SHT IC's I2C interface
    case INIT:
        // code
        state = SETUP;
        return;
    // initiate measurement data transmission
    case IDLE:
        // code
        state = SAMPLE;
        return;
    default: return;
    }
};

void SHT::handle_i2c_response(periCORE::peripherals::i2c::Descriptor* rx) {
    switch(state) {
    // check if initialization was successful
    case SETUP:
        // code
        state = IDLE;
        return;
    // send dummy write to receive measurement data
    case SAMPLE:
        // code
        state = READ;
        return;
    // compute information from measurement data
    case READ:
        // code
        state = IDLE;
        return;
    default: return;
    }
};
```

Listing 11: SHT class' state machine, SHT.cc

In Figure 14, the `sensor_board`'s state machine is visualized. Within the I2C communication sequence, two different causes for state changes can occur: `inport_i2c` events and `inport_trigger` events. `inport_trigger` events occur in configurable intervals, while `inport_i2c` events occur in succession to each sent I2C message.

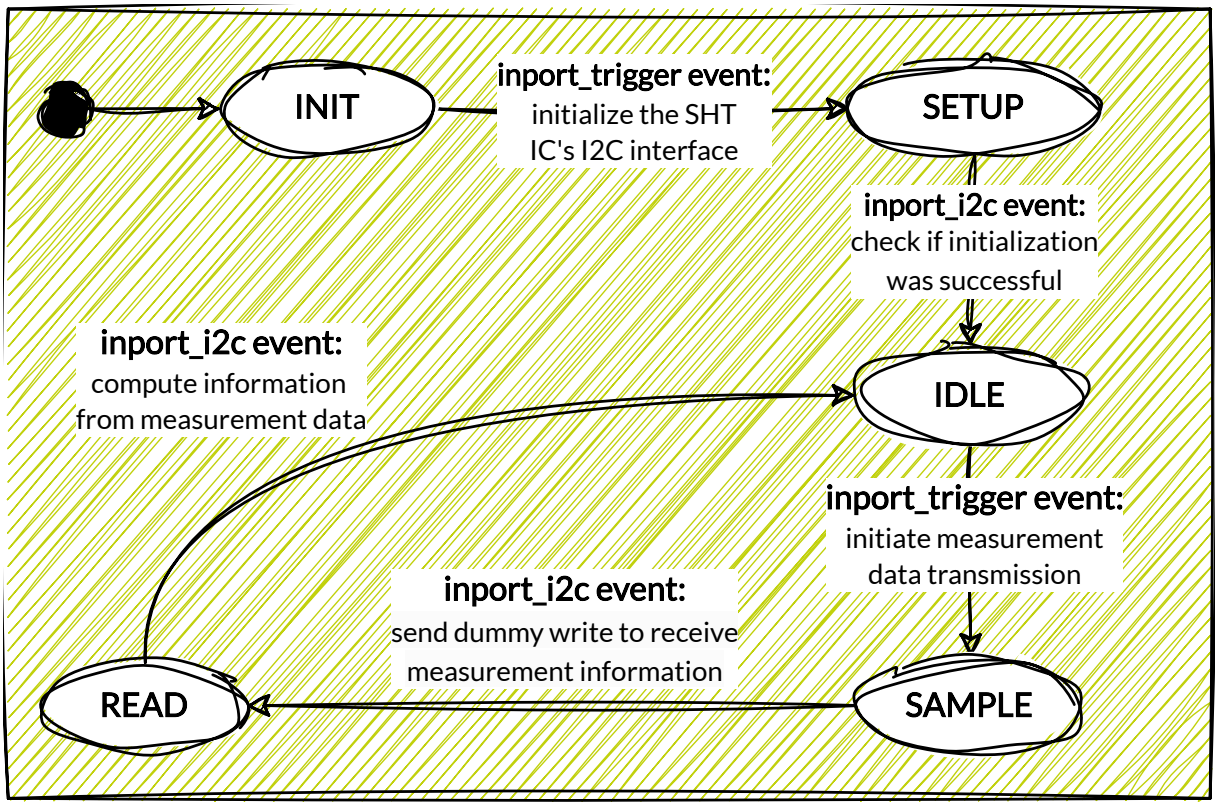


Figure 14: SHT class' state machine (Listing 11)

3.5 I2C bus driver's response implementation

The I2C driver component of `libperiCORE` requires a reference to the inport to which the I2C driver component should respond back to. This allows for multiple participants on the I2C bus while using only one instance of the I2C bus driver class. Call the `set_callback` method with a reference to the `inport_i2c` as parameter within the initialization message setup.

```
...
case INIT:
    // i2c driver specific initialization
    // contains information on which address to respond to
    i2c_descriptor.set_callback(&inport_i2c);
...
```

Listing 12: Implementation of the I2C bus driver response mechanism, SHT.cc

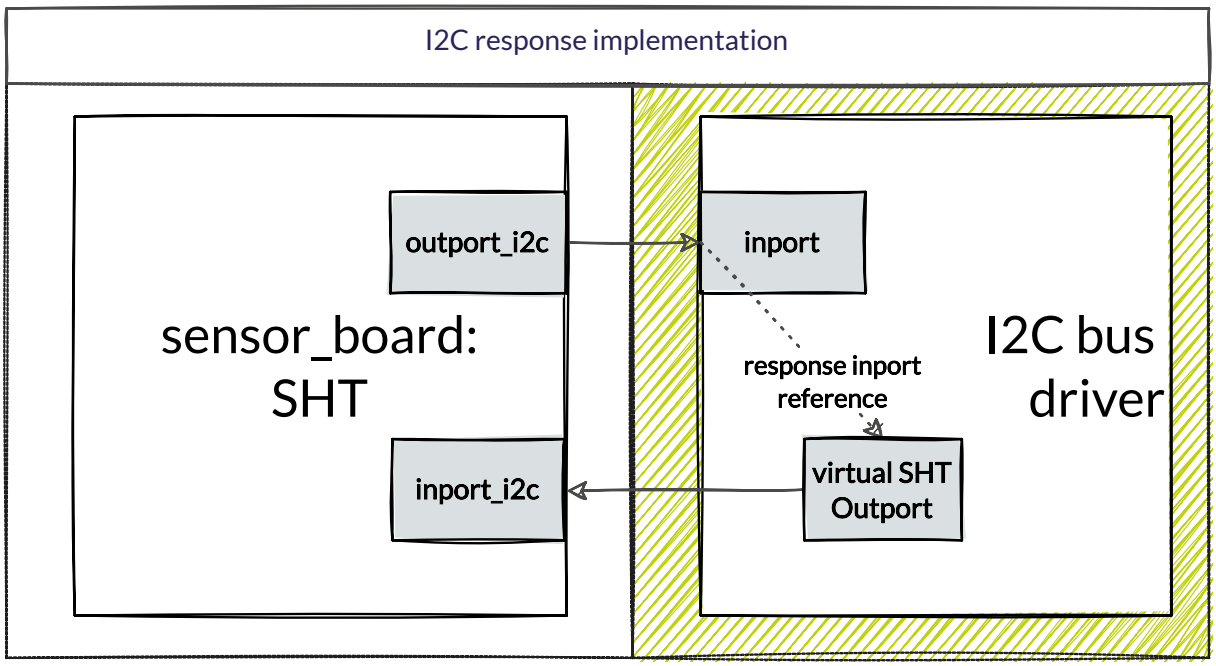


Figure 15: I2C bus driver and sensor_board: response implementation (Listing 12)

3.6 I2C message content implementation

Implement the I2C communication.
The I2C messages' contents as well as addresses are shown in the SHT IC's datasheet (https://download.mikroe.com/documents/datasheets/SHT31-DIS_datasheet.pdf) on pages 9 to 11.

```
...
enum I2C_DEFINES : uint8_t {
    SHT_WRITE_ADDRESS = 0x44 << 1,
    SHT_READ_ADDRESS  = 0x44 << 1 | 0x1
};
...
```

Listing 13: SHT IC's I2C slave addresses, SHT.h

The SHT click board's I2C addresses can be found on page 11 of the SHT datasheet.

```
...
case INIT:
    // i2c driver specific initialization
    // contains information on which address to respond to
    i2c_descriptor.set_callback(&inport_i2c);
    // 1 address byte, 2 command bytes
    i2c_descriptor.set_frame_length(3);
    // SHT datasheet page 11: sht i2c address->byte0, i2c command->byte1&2
    // ( 0x2737: periodic measurement mode )
    i2c_data[0] = SHT_WRITE_ADDRESS;
    i2c_data[1] = 0x27;
    i2c_data[2] = 0x37;
    // assign puts data into the pipeline of an outport
    outport_i2c->assign(&i2c_descriptor);
    state = SETUP;
    return;
// initiate measurement data transmission
case IDLE:
    // 1 address byte, 2 command bytes
    i2c_descriptor.set_frame_length(3);
    // datasheet page 9 ( 0xe000: fetch data )
    i2c_data[0] = SHT_WRITE_ADDRESS;
    i2c_data[1] = 0xe0;
    i2c_data[2] = 0x00;
    outport_i2c->assign(&i2c_descriptor);
    state = SAMPLE;
    return;
default: return;
...
```

Listing 14: Implementation of handle_trigger method, SHT.cc

The `INIT` and `IDLE` state are the only two states in which an `inport_trigger` event is sensible. If the state machine is in a different state at the time of an `inport_trigger` event, the `handle_trigger` method does nothing and therefore waits for the I2C communication sequence to be done.

3.7 Implementing handle_i2c_response with error handling

```
...
case SETUP:
    if (i2c_data[1] == 0 && i2c_data[2] == 0) {
        // sht ic initialized successfully, set to idle state
        state = IDLE;
    } else {
        // not successful, set back to init state
        state = INIT;
    }
    return;
case SAMPLE:
    if (i2c_data[1] == 0 && i2c_data[2] == 0) {
        // initial message successful, read measurement data
        i2c_descriptor.set_frame_length(7);
        i2c_data[0] = SHT_READ_ADDRESS;
        outport_i2c->assign(&i2c_descriptor);
        state = READ;
    } else {
        // start over
        i2c_data[1] = 0xe0;
        i2c_data[2] = 0x00;
        outport_i2c->assign(&i2c_descriptor);
    }
    return;
case READ:
    // faulty measurement data from the SHT IC is 0xFF
    if (i2c_data[1] == 0xFF || i2c_data[4] == 0xFF) {
        // start over
        i2c_descriptor.set_frame_length(3);
        i2c_data[0] = SHT_WRITE_ADDRESS;
        i2c_data[1] = 0xe0;
        i2c_data[2] = 0x00;
        outport_i2c->assign(&i2c_descriptor);
        state = SAMPLE;
    } else {
        // compute information from measurement data
        state = IDLE;
    }
    return;
default: assert(false);
...
```

Listing 15: Implementing handle_i2c_response method, SHT.cc

Contrary to an `inport_trigger` event, an incorrect `inport_i2c` event must be caught on occurrence. As `inport_i2c` events only occur as a response to an I2C message, it indicates an error in the I2C communication sequence. The `default: assert(false);` line of code triggers a breakpoint while debugging. During runtime, it will cause the periCORE to execute a soft reset.

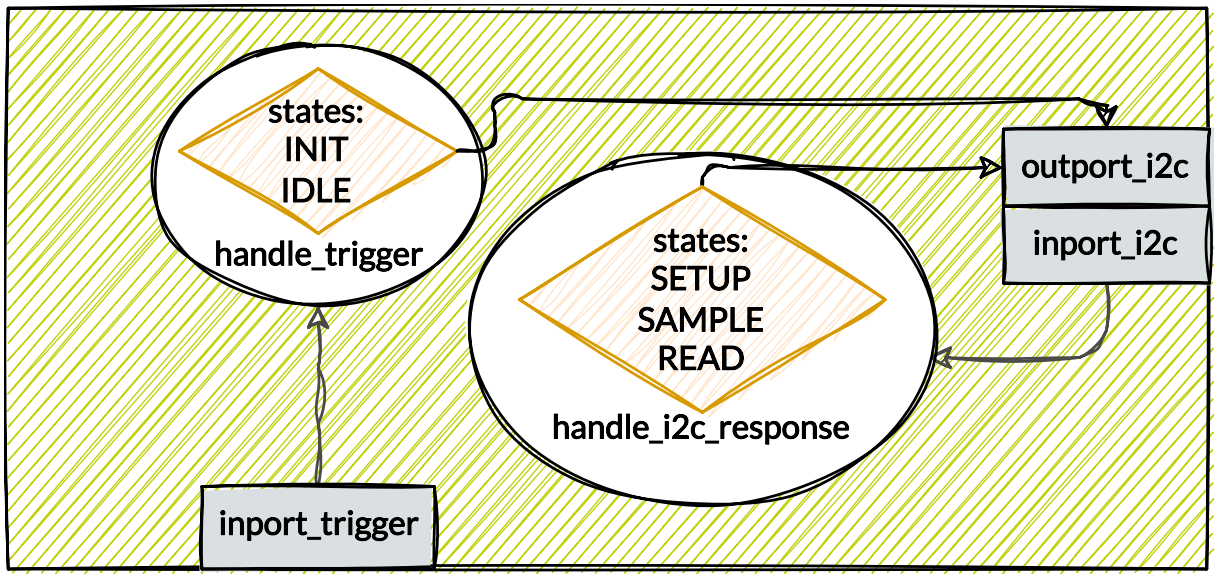


Figure 16: Visual representation of sensor_board's structure with added state machine (Listing 15)

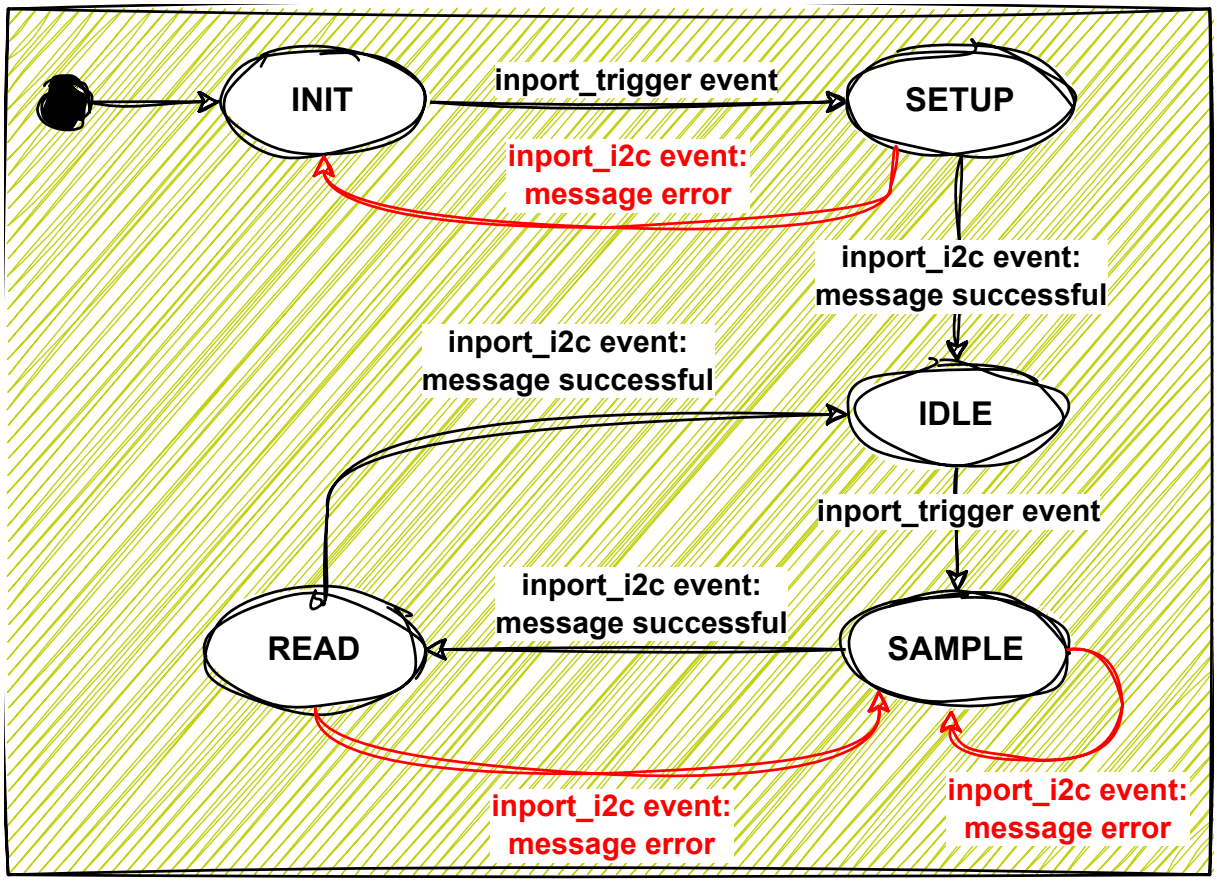


Figure 17: SHT class' state machine (error handling) (Listing 15)

3.8 Event channel

Create an event channel between the `sensor_board`'s outport and the I2C bus driver component's inport in main.cc. Note that the event channel from I2C bus driver component back to the `sensor_board` component has been implemented in Listing 12.

```
...
int main() {
    ...
    // connect sensor_board to i2c bus driver component
    sensor_board.set_outport_i2c (node_environment.peripherals.i2c.
   ↪ get_inport_bus());
    ...
```

Listing 16: I2C bus driver to sensor_board event channel implementation, main.cc

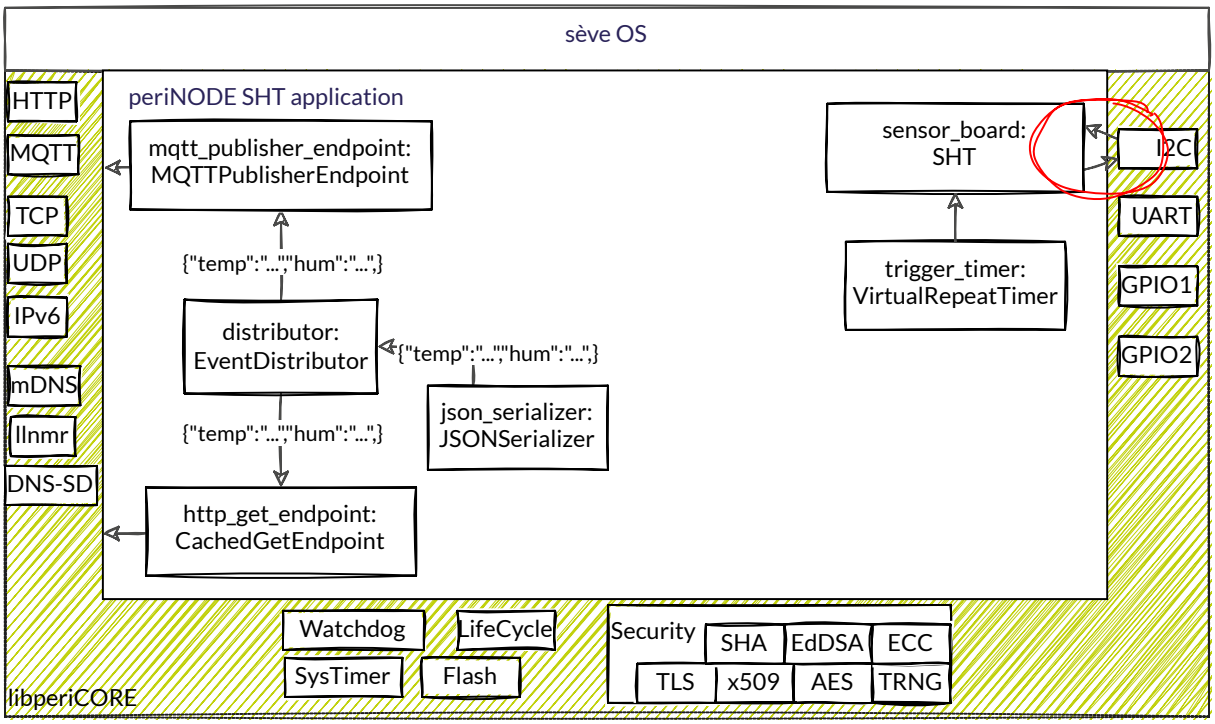


Figure 18: Eventflow diagram with added I2C event channels (Listing 16)

3.9 Converting data into information

As the last step, compute readable information from measurement data (datasheet page 14).

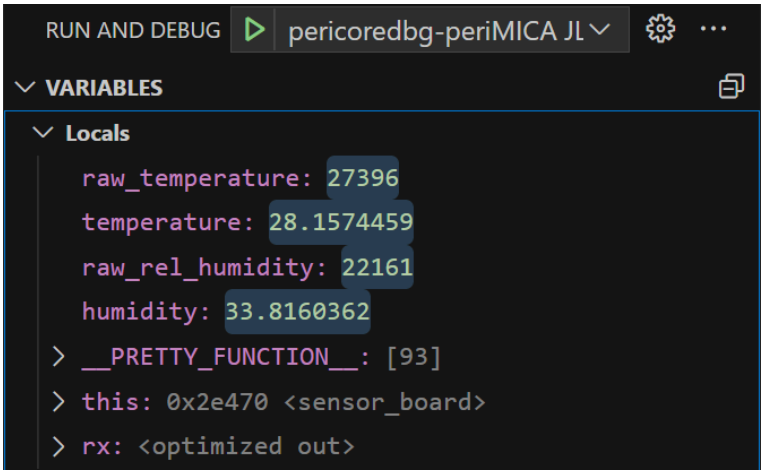


Figure 19: Screenshot of correct measurement information in Visual Studio Code

```
case READ:
...
} else {
    // byte 1 (High byte) and 2 (Low byte) of I2C-frame represent temperature
    ↪ data as 16-bit integer
    uint16_t raw_temperature = i2c_data[1] << 8 | i2c_data[2];
    // conversion of signal output, page 14, section 4.13 of datasheet (
    ↪ temperature)
    float temperature = ((175.0*raw_temperature)/(0xFFFF-1))-45;
    // byte 4 (High byte) and 5 (Low byte) of I2C-frame represent temperature
    ↪ data as 16-bit integer
    uint16_t raw_rel_humidity = i2c_data[4] << 8 | i2c_data[5];
    // conversion of signal output, page 14, section 4.13 of datasheet (
    ↪ humidity)
    float humidity = (100.0*raw_rel_humidity)/(0xFFFF-1);
    // event channel to API component in the following section
    // outport_data->assign(temperature, humidity);
    state = IDLE;
}
...
```

Listing 17: Computing readable information within handle_i2c_response method, SHT.cc

Stop here to compile and debug. Set a breakpoint at the `state = IDLE;` line within the `handle_i2c_response` method in `SHT.cc` after adding `volatile` to the `temperature, humidity` variables. Those variables are otherwise optimized out by the compiler. Check for the temperature and humidity values to be correct (Figure 19). Once checked, remove the `volatile` prefix.

3.10 Add outport_data (data outport to API_adapter component)

Add the `outport_data` to `SHT.h` and **remove the comments of the assign in `handle_i2c_response(SHT.cc)`** to let the `sensor_board` send it's computed information. The data type of the `outport_data` is `float, float`.

```
public:
    ...
    void set_outport_data(seve::eventflow::Sink<float, float>* outport) {
    ↪ outport_data = outport; };
private:
    ...
    seve::eventflow::Sink<float, float>* outport_data{nullptr};
...
```

Listing 18: Adding outport_data, SHT.h

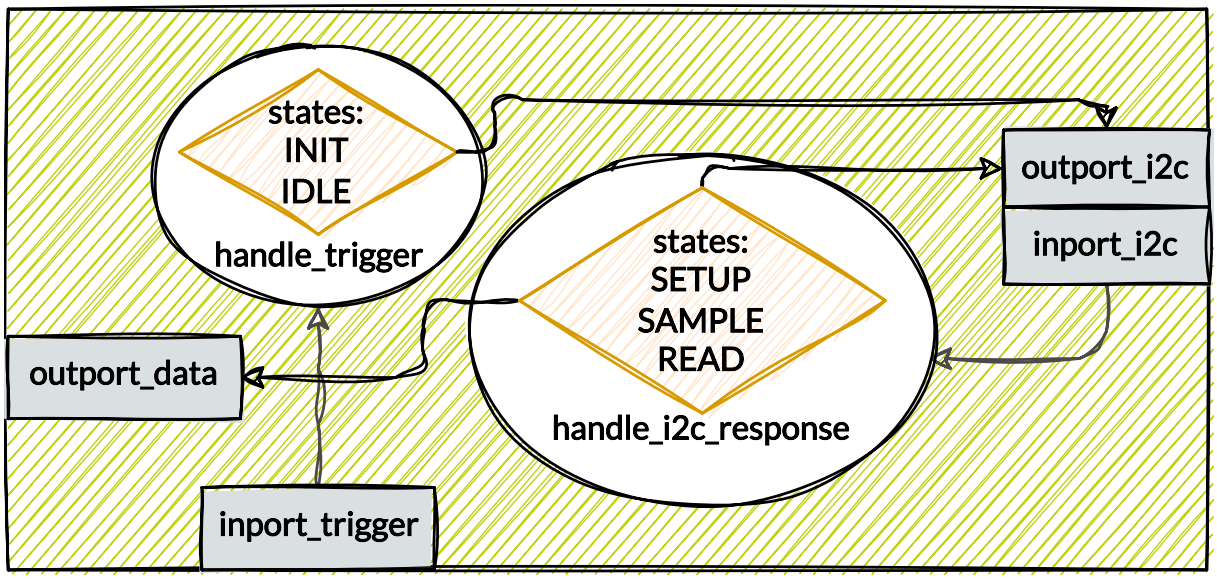


Figure 20: Visual representation of the fully implemented sensor_board (Listing 18)

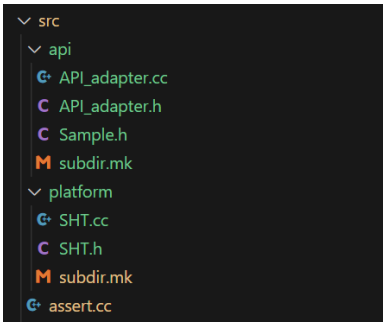


Figure 22: Screenshot of final `src` folder structure

4 API adapter component

A visual representation of the `api_adapter` is shown in Figure 21 including its ports and event handler method.

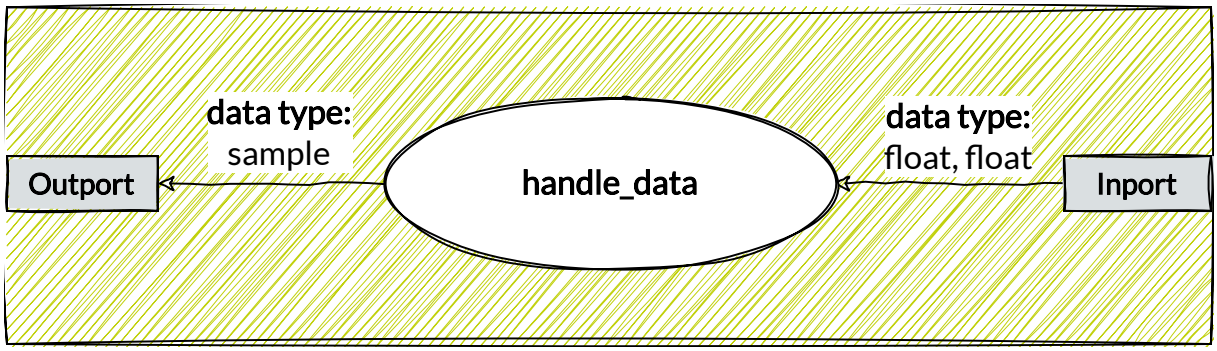


Figure 21: Visual representation of api_adapter's structure

Add a folder named `api` to the project's folder structure under `src`. Into this folder, copy a `subdir.mk` file. Create three files in this folder: `API_adapter.cc`, `API_adapter.h`, `Sample.h` (Figure 22).

4.1 API_adapter component's ports

Add the inport with event handler method to `API_adapter.h`. As the inport will have an event channel to the `outport_data` of the `sensor_board`, it must have the same data type, `float`, `float`.

```
#pragma once
#include "seve/eventflow/Sink.h"
#include "seve/eventflow/SingleValue.h"

namespace api
{
class API_adapter
{
public:
    seve::eventflow::Sink<float,float>* get_inport() { return &inport; };
private:
    seve::eventflow::SingleValue<API_adapter, float, float> inport
        {this, &API_adapter::handle_data};

    void handle_data(float, float);
};
}
```

Listing 19: Adding an inport to api_adapter, API_adapter.h

Add the event channel from `sensor_board's outport_data` to `api_adapter's inport` in `main.cc`. Before that, include and instantiate an object of the `api_adapter` class.

```
...
// newly added includes
#include "api/API_adapter.h"

...
// newly added instantiations
api::API_adapter api_adapter;

...
int main() {

    ...
    // connect SHT component to API adapter
    sensor_board.set_outport_data(api_adapter.get_inport());
    ...
```

Listing 20: Adding include and instantiation of api_adapter and it's inport event channel, main.cc

Define the `handle_data` method of the `api_adapter`.

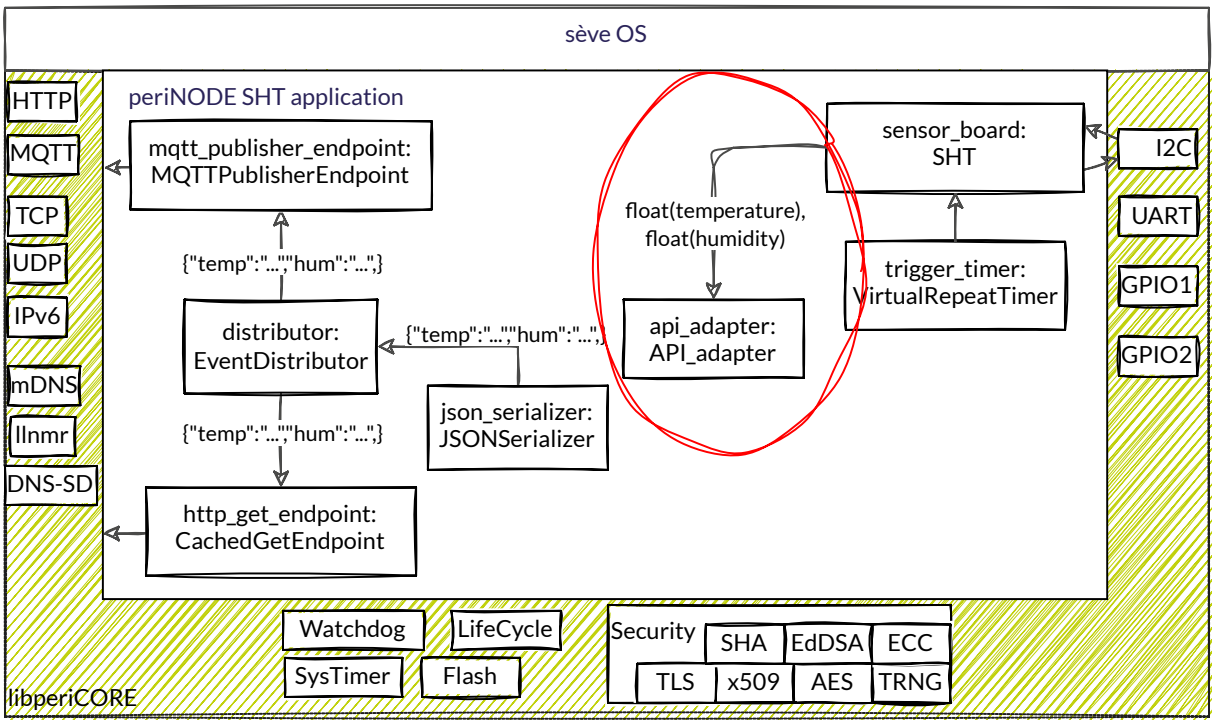


Figure 23: eventflow diagram with added api_adapter component and event channel (Listing 20)

```
#include "api/API_adapter.h"
#include <array> //to_array

using namespace api;
void API_adapter::handle_data(float temperature, float humidity) {};
```

Listing 21: Defining the handle_data method of api_adapter, API_adapter.cc

Stop here to compile and debug. Set a breakpoint at the `handle_data` method in `API_adapter.cc`. You should witness the debugger stop here.
Add an outport to `API_adapter.h`. For that, the `Sample` class is required.

4.2 Sample.h

```
...
#include "api/Sample.h"
...
public:
    ...
    void set_outport(seve::eventflow::Sink<api::Sample>* port) {outport = port
    ↪ ; };
private:
    ...
    seve::eventflow::Sink<api::Sample>* outport{nullptr};
...
```

Listing 22: Adding outport, API_adapter.h

The reason for the `api_adapter`'s implementation is to unify the information into a structure that can be accepted by the `json_serializer`. For that, we use the `Sample` class.

The `api_adapter` converts information into an object of the `Sample` class. The `Sample` class contains following member variables:

- incarnation: number of times the device was rebooted
- sequence_number: number of samples sent since the last reboot
- unit_temperature: measured unit, in this case °C
- unit_humidity: measured unit, in this case % humidity
- temperature: measured temperature value
- humidity: measured humidity value

Declare `Sample` class with its member variables.

```
#pragma once
#include "perinet/periNODE/api/types.h"
#include <array>
#include <cstdint>
#include <tuple>
#include <utility>

namespace api
{
class Sample
{
using string_t = perinet::periNODE::api::fixed_short_string_t;
public:
    uint32_t incarnation;
    uint64_t sequence_number;
    string_t unit_temperature;
    string_t unit_humidity;
    float temperature;
    float humidity;
    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("unit_temperature"), &Sample::
    ↪ unit_temperature),
        std::make_pair(std::to_array("unit_humidity"), &Sample::unit_humidity)
    ↪ ,
        std::make_pair(std::to_array("temperature"), &Sample::temperature),
        std::make_pair(std::to_array("humidity"), &Sample::humidity));
};
}
```

Listing 23: Implementation of Sample.h

Assign the measured information to the previously declared `Sample`. As multiple member variables need to be set only once in a power cycle, it is sensible to use the constructor method for this step.

```
...
public:
    API_adapter();
    ...
private:
    ...
    api::Sample template_sample;
...
```

Listing 24: Adding constructor and required class attribute, API_adapter.h

```
#include "periCORE/NodeEnvironment.h"
...

void API_adapter::handle_data(float temperature, float humidity)
{
    // sequence_number assures unique sample
    template_sample.sequence_number += 1;

    Sample sample = template_sample;
    sample.temperature = temperature;
    sample.humidity = humidity;

    assert(outport != nullptr);
    outport->assign(sample);
};

API_adapter::API_adapter()
{
    // initialize lifecycle information
    template_sample.incarnation = node_environment.get_incarnation();
    template_sample.sequence_number = 0;
    // units are set to be 4 characters long, (° is represented by two
    ↪ characters)
    template_sample.unit_temperature = std::to_array("°C\0");
    template_sample.unit_humidity = std::to_array("%\0\0\0");
}
```

Listing 25: Implementation of API.cc

Create the event channel from `api_adapter's outport` to `json_serializer's inport`. Note that the `periCORE::lib::JSONSerializer<api::Sample> json_serializerpool;` line of code must be placed after the `seve::lib::memory::BufferPool poolMAX_SAMPLE_SIZE, 5; // memory to store serialized samples` line of code.

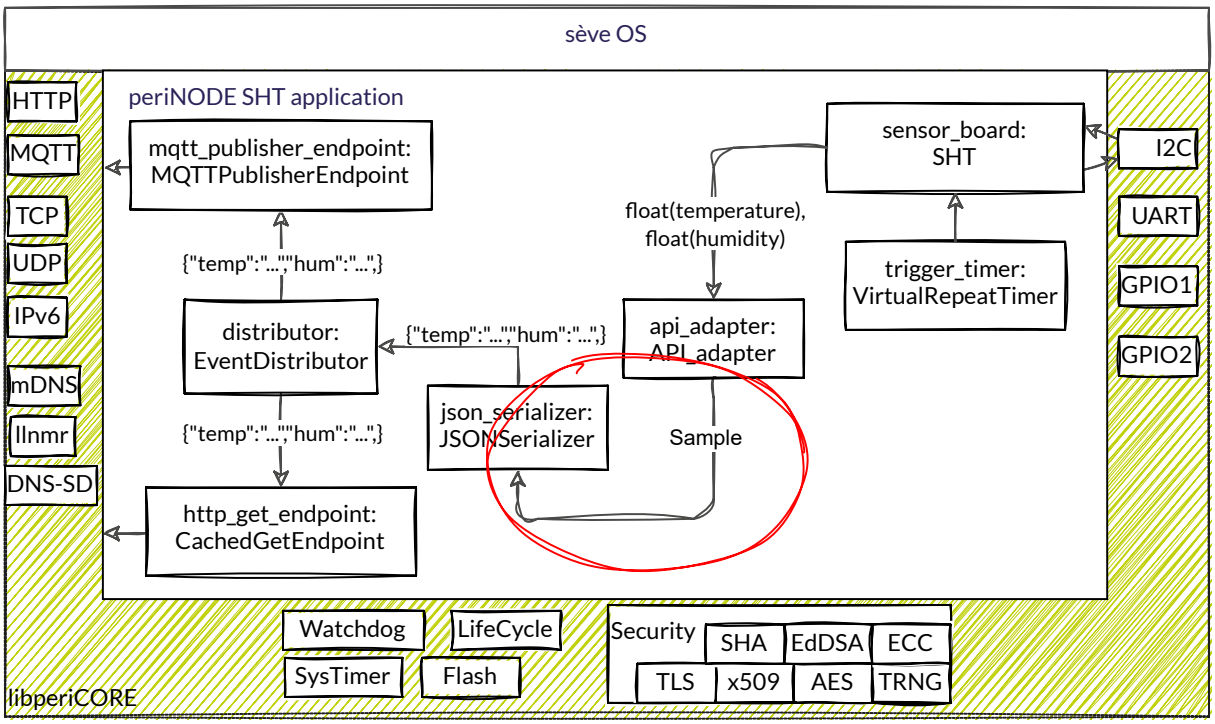


Figure 24: eventflow diagram with added event channel (Listing 26)

```
...
periCORE::lib::JSONSerializer<api::Sample> json_serializer{pool};
...
int main()
{
    ...
    // encapsulate measurements into an API conform sample
    api_adapter.set_outport(json_serializer.get_inport());
...
```

Listing 26: Creating event channel from api_adapter outport to json_serializer inport, main.cc

Check, if the following channels have been created:

1. `trigger_timer`'s outport to `sensor_board`'s `inport_trigger`
2. `sensor_board's outport_i2c` to I2C bus driver's inport
3. `sensor_board's outport_data` to `api_adapter's inport`
4. `api_adapter's outport` to `json_serializer's inport`

Stop here to compile and debug. Set a breakpoint at the `outport->assign(sample);` line in `API_adapter.cc`. You should witness the debugger stop here. The content of `sample` should be optimized out, `temperature` and `humidity` should still be visible.

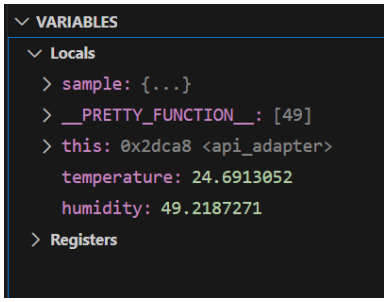


Figure 25: Screenshot of correct measurement information in Visual Studio Code, API_-adapter

5 Web UI adaptation

Open the file `webui -> fs_root -> js_node` all code specific to the display of information will be duplicated and renamed by adding `_temp` to the copied names of variables and `_hum` to all others.

`label_step` and `label_count` define the range of the `ruler` (the grafic display of information). In this case, both temperature and humidity are displayed in the range from 0 to 75. If the range shall not start from 0, change the 0 to the desired start of the range within the designated `for-loop` in home.js:

```
const label_step_temp = 5;
const label_count_temp = 15;
const label_step_hum = 5;
const label_count_hum = 15;
...

function dom_home() {
var labels_temp = "";
var labels_hum = "";

for(var i = 0; i < label_count_temp; i++)
{
    const val_temp = 0 + label_step_temp * i;
    const val_str_temp = val_temp.toString();

    console.log(val_str_temp);
    labels_temp += get_label_dom(val_str_temp);
}

for(var i = 0; i < label_count_hum; i++)
{
    const val_hum = 0 + label_step_hum * i;
    const val_str_hum = val_hum.toString();

    console.log(val_str_hum);
    labels_hum += get_label_dom(val_str_hum);
}
// continued on next page
```

Listing 27: Editing webUI's dashboard part 1, home.js

In home.js, continued:

```
...
return `<section>` + get_perinode_image_figure() + `<h1>Temperature</h1>
<div class='ruler'>
    <div id='bar_temp'></div>
    <div>` + labels_temp + `
    <div class='unit'>
        <label>` + (label_step_temp*label_count_temp).toString() + `</label>
    </div>
    </div>
</div>
<br>
<h2 id="value_temp">10°C</h2>

<h1>Humidity</h1>
<div class='ruler'>
    <div id='bar_hum'></div>
    <div>` + labels_hum + `
    <div class='unit'>
        <label>` + (label_step_hum*label_count_hum).toString() + `</label>
    </div>
    </div>
</div>
<br>
<h2 id="value_hum">20%</h2>
</section>`;
}
// continued on next page
```

Listing 28: Editing webUI's dashboard part 2, home.js

In home.js, continued:

```
...
function draw_temperature_and_humidity(sample) {
    var value_temp = sample.temperature;
    var value_hum = sample.humidity;
    var min_temp = 0;
    var max_temp = label_step_temp * label_count_temp;
    var min_hum = 0;
    var max_hum = label_step_hum * label_count_hum;
    if (min_temp > value_temp) {
        value_temp = "out of range"
        $("bar_temp").style.width = ((0) * 100) / (max_temp - min_temp) + "%";
        $("value_temp").innerHTML = value_temp;
    }else if (value_temp > max_temp){
        value_temp = "out of range"
        $("bar_temp").style.width =
            ((max_temp - min_temp) * 100) / (max_temp - min_temp) + "%";
        $("value_temp").innerHTML = value_temp;
    } else {
        $("bar_temp").style.width =
            ((value_temp - min_temp) * 100) / (max_temp - min_temp) + "%";
        $("value_temp").innerHTML =
            value_temp.toFixed(1) + " " + sample.unit_temperature;
    }
    if (min_hum > value_hum) {
        value_hum = "out of range"
        $("bar_hum").style.width = ((0) * 100) / (max_hum - min_hum) + "%";
        $("value_hum").innerHTML = value_hum;
    }else if (value_hum > max_hum){
        value_hum = "out of range"
        $("bar_hum").style.width =
            ((max_hum - min_hum) * 100) / (max_hum - min_hum) + "%";
        $("value_hum").innerHTML = value_hum;
    } else {
        $("bar_hum").style.width =
            ((value_hum - min_hum) * 100) / (max_hum - min_hum) + "%";
        $("value_hum").innerHTML =
            value_hum.toFixed(0) + " " + sample.unit_humidity;
    }
}
// continued on next page
```

Listing 29: Editing webUI's dashboard part 3, home.js

In home.js, continued:

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

function reload_home() {
$("content_id").innerHTML = dom_home();
document.title = "periNODE Web Server | Temperature and Humidity";
draw_temperature_and_humidity(
    {"temperature":0,"unit_temperature":"°C","humidity":0,"unit_humidity":"%"}
); //out of range
reconnection_update.overload_update("/sample", 1000, set_sample);
}
load_css_script("css_node/ruler.css");
update_content("home", reload_home);
```

Listing 30: Editing webUI's dashboard part 4, home.js

6 Conclusion

After completing this guide, the homepage of the periNODE website should look like this:

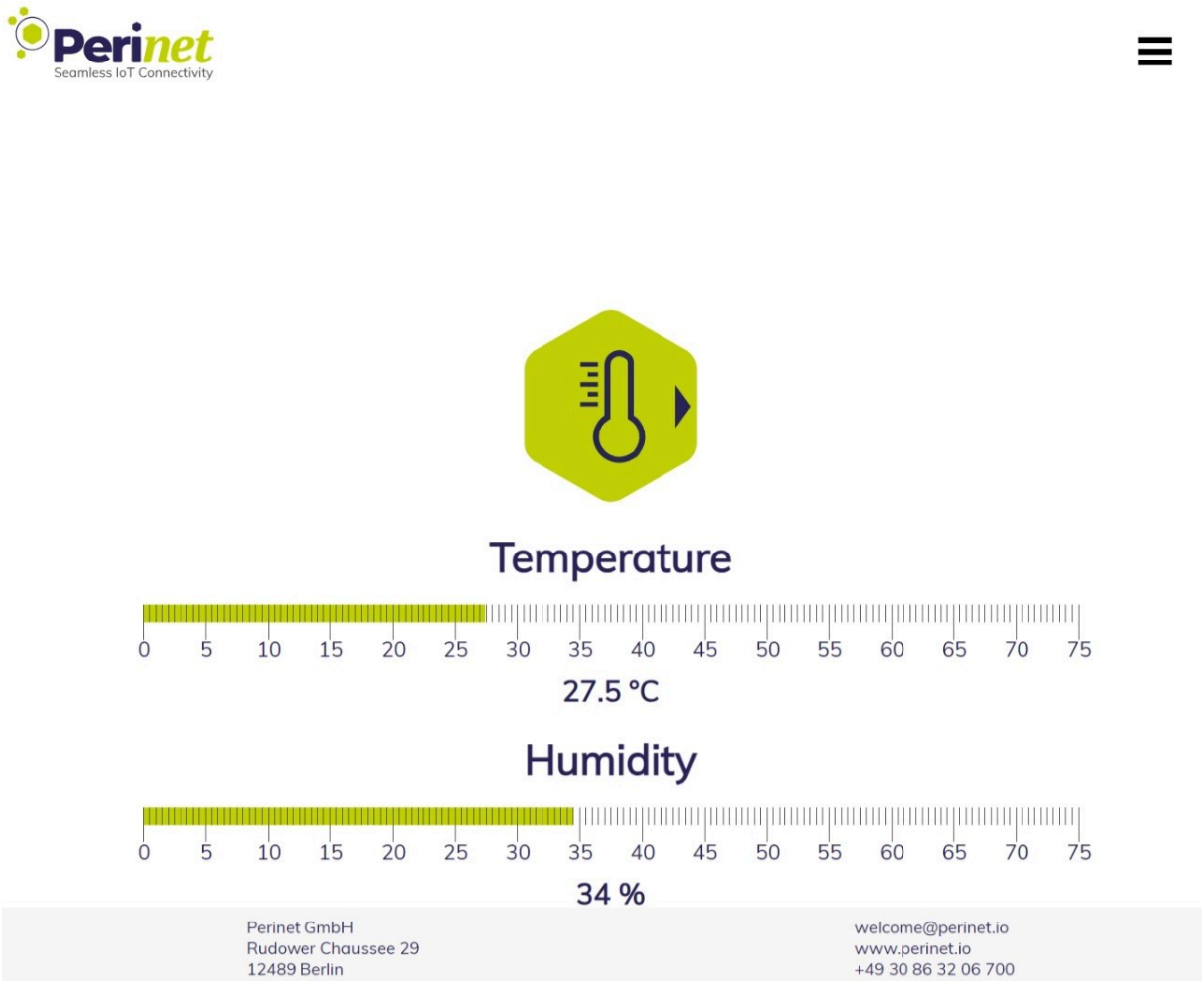


Figure 26: periNODE website's dashboard displaying SHT's measurements

If data is not being displayed, the website is not loading at all, the project is not compiling or any other problems occur, consider starting from scratch and compiling the project often to track when errors occur. If problems cannot be solved through this resource, we will happily assist in problem-solving, just contact us!

7 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

100BASE-T1 A Ethernet Standard where two endpoints are connected by a single twisted pair cable. It is one of the so-called Single Pair Ethernet (SPE) standards. It operates in full-duplex with a data rate of 100 MBit per second. Furthermore, it uses PAM-3 modulation with a voltage level from -1 to +1V, differentially on the two wires. 5

100BASE-TX A Ethernet Standard where two twisted pairs with differential signals are used, one for each direction. The data rate is 100 MBit per second. It is also called "Fast Ethernet". 5

ADC Analog Digital Converter. 7, 8

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

I2C Inter-Integrated Circuit is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial bus. 4, 7, 9, 18, 19, 21–24, 26, 27, 37

IC Integrated Circuit. 4, 7, 18, 19, 23

JSON JavaScript Object Notation is standard text-based format for representing structured data based on JavaScript object syntax. 7

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

UI User Interface. 4

D References

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

[2] Perinet GmbH. Starter Kit Plus User Guide. PRN.100.548. https://docs.perinet.io/PRN100548-StarterKitPlusUserGuide.pdf.

E Revision History

Revision	Date	Author(s)	Description
1	2023-1-26	clip	Initial Draft
2	2023-09-06	clip	First Release