Embedded systems programming

Project requirements

Requirements

- Control the speed of a ventilation fan
- Fan is connected to Produal MIO 12-V
  - IO-pins are controlled using Modbus protocol (Address: 1)
  - 0-10V output controls the ventilation fan speed
  - Digital input counts rotation pulses from the fan. Counter value is used so check if the fan is running
- Vaisala GMP256 CO2 probe
  - Modbus protocol (Address: 240)
- Vaisala HMP60
  - Relative humidity and temperature
  - Modbus protocol (Address: 241)
- Two operating modes:
  - Manual mode
  - Automatic mode
- LCD user interface
- Web interface from Joe’s IoT course

Automatic mode

- Set the pressure level in the ventilation duct (0 – 120 pa) in the UI
  - Pressure level is pressure difference between the room and the ventilation duct
- Controller measures pressure level and keeps it at the required level by adjusting the fan speed
  - If required level can’t be reached within a reasonable time user is notified on the UI

Manual mode

- Set the speed of the fan in UI (0 – 100%)
- Display current fan setting and pressure level in the UI

Dual interface

- Settings can be adjusted both from LCD UI and the web UI
- Changes must be reported to both Uis
  - Changes in settings
  - Changes in measured values

Documentation

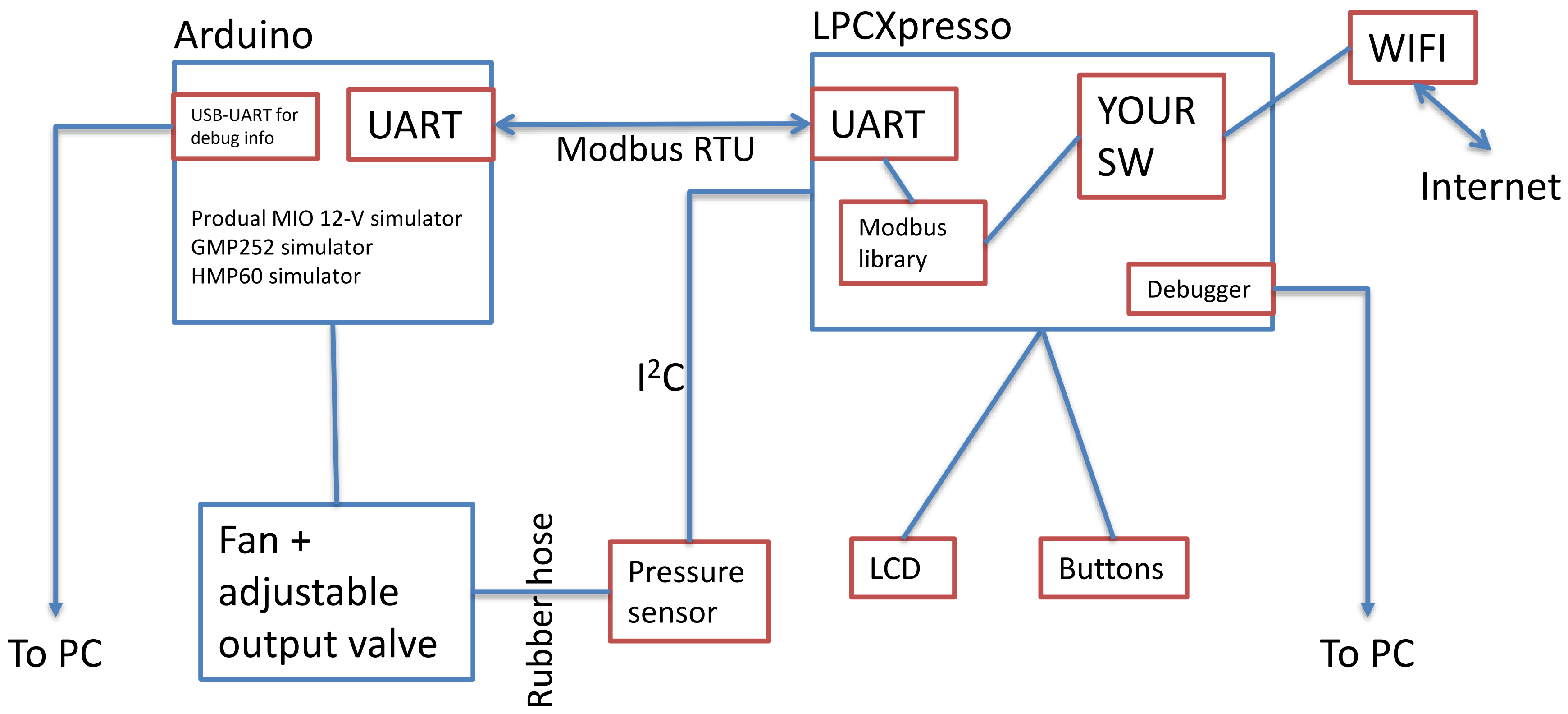
- User manual
- Wiring diagrams
- Program documentation

Some help and instructions

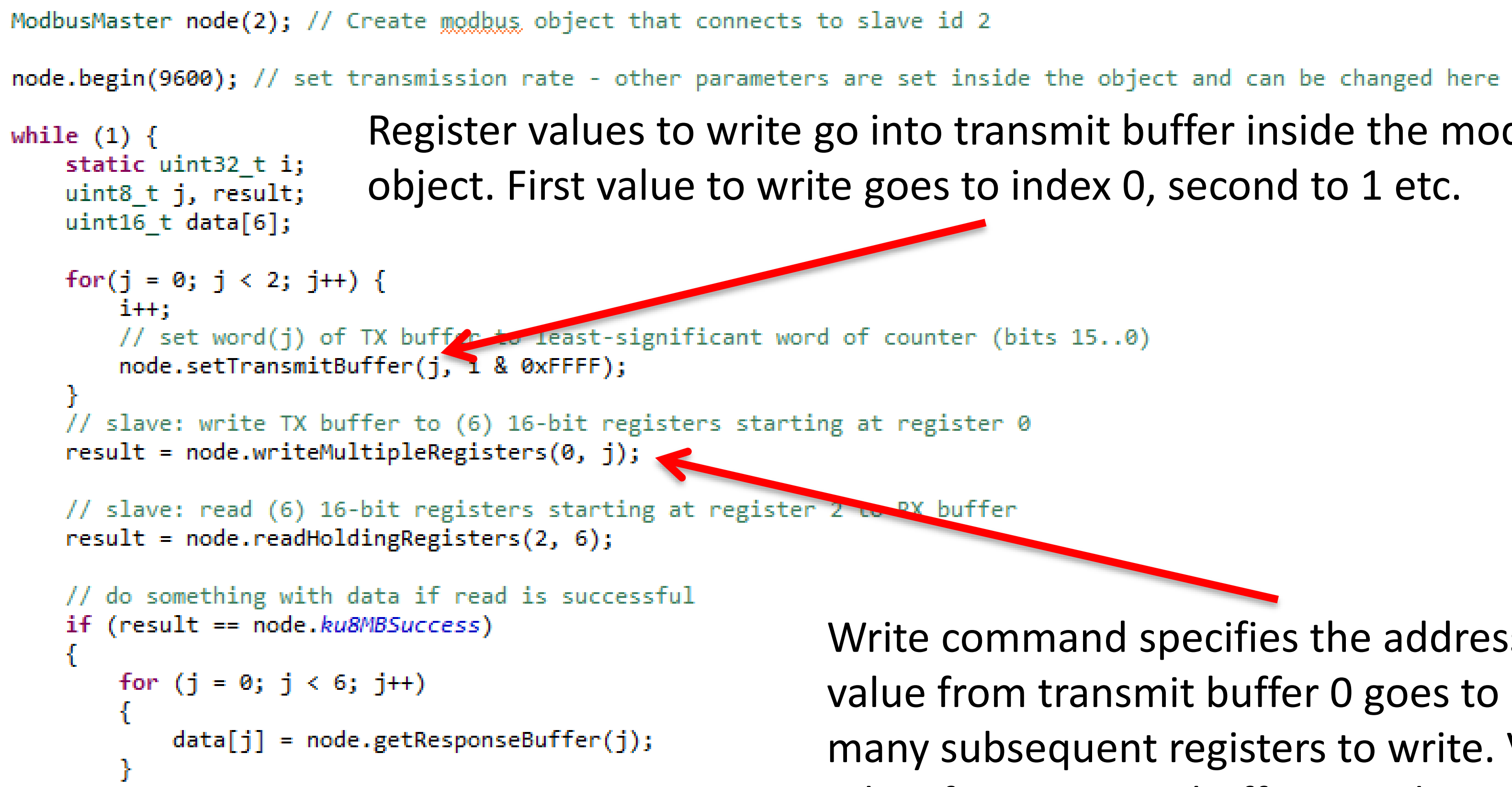
Wiring instructions, sensors, system diagram, etc.

System diagram

Modbus devices are controlled by reading/writing Modbus registers.
Modbus register is a ”variable” that can be accessed using Modbus protocol.



Writing Modbus registers

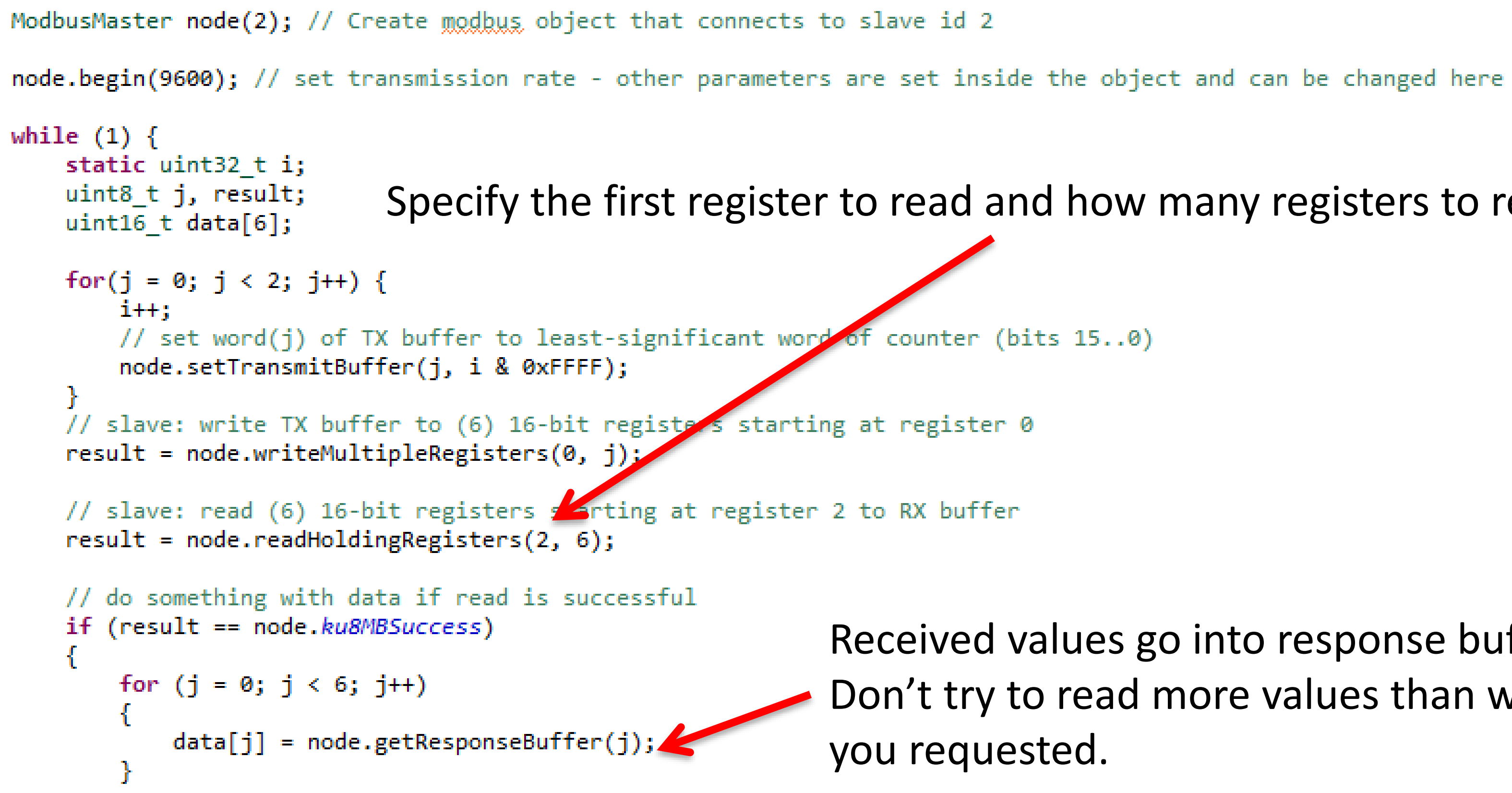


```
ModbusMaster node(2); // Create modbus object that connects to slave id 2

node.begin(9600); // set transmission rate - other parameters are set inside the object and can be changed here

while (1) {
    static uint32_t i;
    uint8_t j, result;
    uint16_t data[6];

    for(j = 0; j < 2; j++) {
        i++;
        // set word(j) of TX buffer to least-significant word of counter (bits 15..0)
        node.setTransmitBuffer(j, i & 0xFFFF);
    }
    // slave: write TX buffer to (6) 16-bit registers starting at register 0
    result = node.writeMultipleRegisters(0, j);

    // slave: read (6) 16-bit registers starting at register 2 to RX buffer
    result = node.readHoldingRegisters(2, 6);

    // do something with data if read is successful
    if (result == node.ku8MBSuccess)
    {
        for (j = 0; j < 6; j++)
        {
            data[j] = node.getResponseBuffer(j);
        }
```

Register values to write go into transmit buffer inside the modbus object. First value to write goes to index 0, second to 1 etc.

Write command specifies the address where value from transmit buffer 0 goes to and how many subsequent registers to write. Values are taken from tranmit buffer in order starting with index 0.

Reading Modbus registers



```
ModbusMaster node(2); // Create modbus object that connects to slave id 2

node.begin(9600); // set transmission rate - other parameters are set inside the object and can be changed here

while (1) {
    static uint32_t i;
    uint8_t j, result;
    uint16_t data[6];

    for(j = 0; j < 2; j++) {
        i++;
        // set word(j) of TX buffer to least-significant word of counter (bits 15..0)
        node.setTransmitBuffer(j, i & 0xFFFF);
    }
    // slave: write TX buffer to (6) 16-bit registers starting at register 0
    result = node.writeMultipleRegisters(0, j);

    // slave: read (6) 16-bit registers starting at register 2 to RX buffer
    result = node.readHoldingRegisters(2, 6);

    // do something with data if read is successful
    if (result == node.ku8MBSuccess)
    {
        for (j = 0; j < 6; j++)
        {
            data[j] = node.getResponseBuffer(j);
        }
```

Specify the first register to read and how many registers to read.

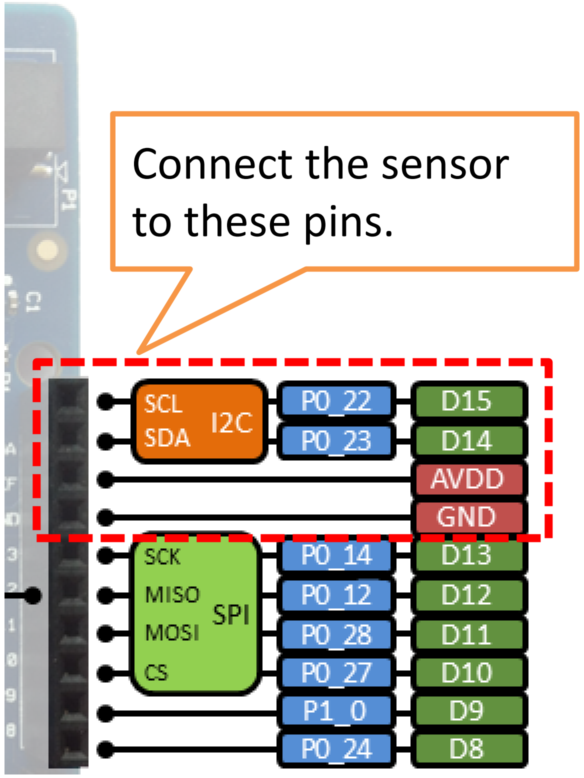
Received values go into response buffer. Don’t try to read more values than what you requested.

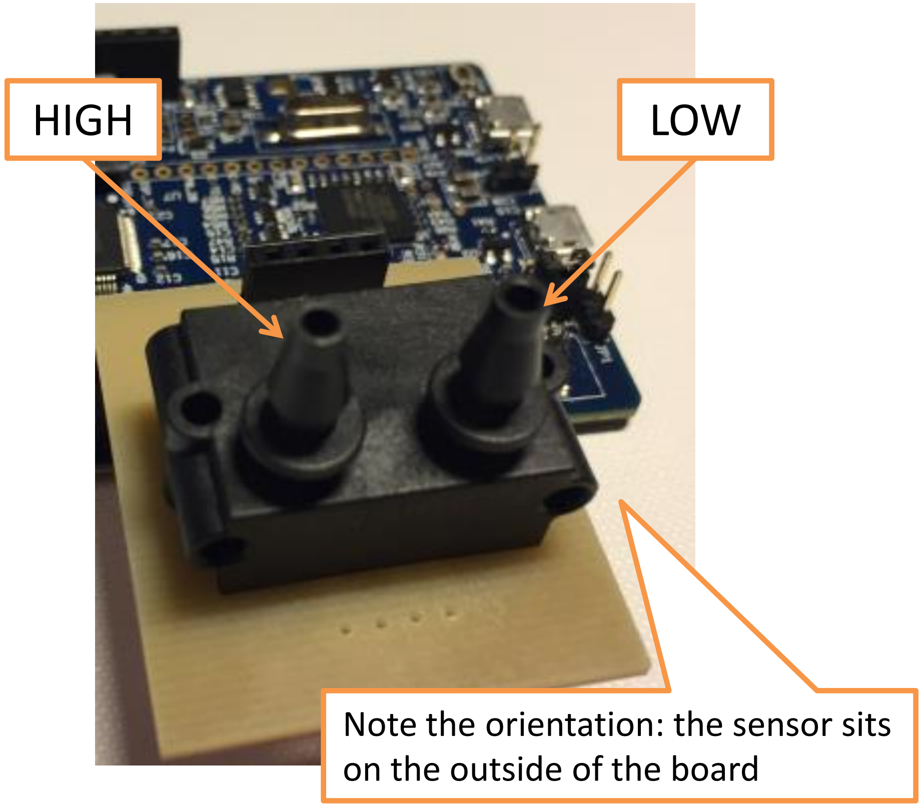
Interface

- Pressure sensor has an I^{2}C interface and operates on 3.3 V
- I^{2}C interface
  - Two signals: SCL (clock) and SDA (data)
  - SDA is bidirectional (both input and output)
  - Requires pull-up resistors of specific size → can't use built in pull-ups
    - Our board has suitable external pull-ups installed on P0-22 (SCL/D15) and P0-23 (SDA/D14)

Pressure sensor board

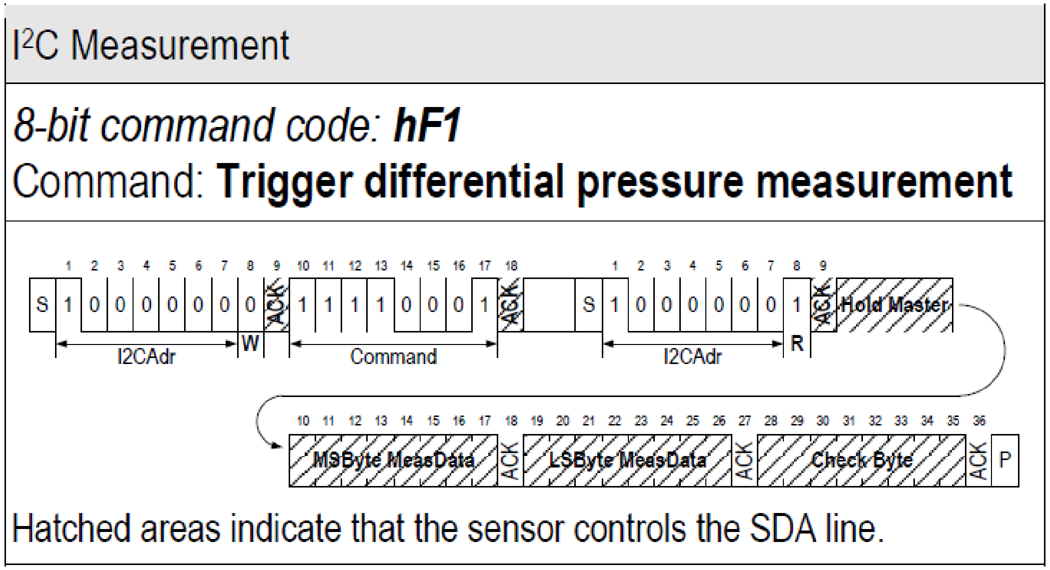
- To measure the pressure in the ducts connect a measuring hose to HIGH and the other end of the measuring hose to a connector in the ventilation duct




HIGH
LOW
Note the orientation: the sensor sits on the outside of the board

Measurement

- The address of the sensor is 0x40
- The command for reading value is 0xF1
- Sensor returns 3 bytes: 2 bytes of data and 1 byte CRC


I²C Measurement

8-bit command code: ***hF1***

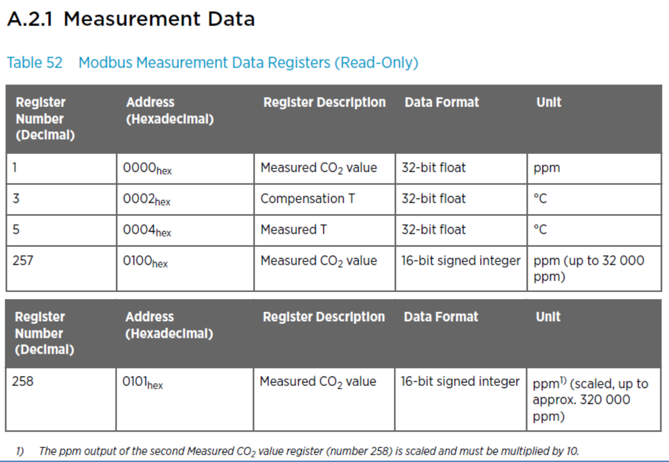
Command: **Trigger differential pressure measurement**

Hatched areas indicate that the sensor controls the SDA line.

Measurement

- Sensor returns a signed 16 bit value
- The value must be converted to physical value (pascals)
  - Scale factor (see data sheet chapter 2)
  - Altitude correction (see data sheet chapter 5)
  - Hose length compensation (see data sheet chapter 8) – not needed for hose length up to 1 m

Vaisala sensor simulator supported registers



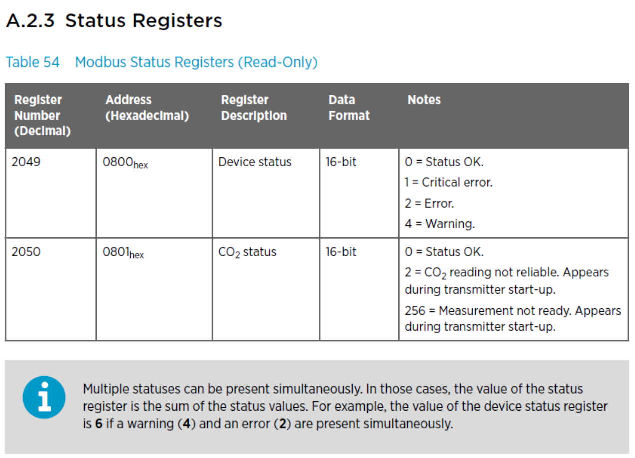
A.2.1 Measurement Data

Table 52 Modbus Measurement Data Registers (Read-Only)

Register Number (Decimal)	Address (Hexadecimal)	Register Description	Data Format	Unit
1	0000_{hex}	Measured CO_2 value	32-bit float	ppm
3	0002_{hex}	Compensation T	32-bit float	°C
5	0004_{hex}	Measured T	32-bit float	°C
257	0100_{hex}	Measured CO_2 value	16-bit signed integer	ppm (up to 32 000 ppm)

Register Number (Decimal)	Address (Hexadecimal)	Register Description	Data Format	Unit
258	0101_{hex}	Measured CO_2 value	16-bit signed integer	ppm1) (scaled, up to approx. 320 000 ppm)

1) *The ppm output of the second Measured CO_2 value register (number 258) is scaled and must be multiplied by 10.*



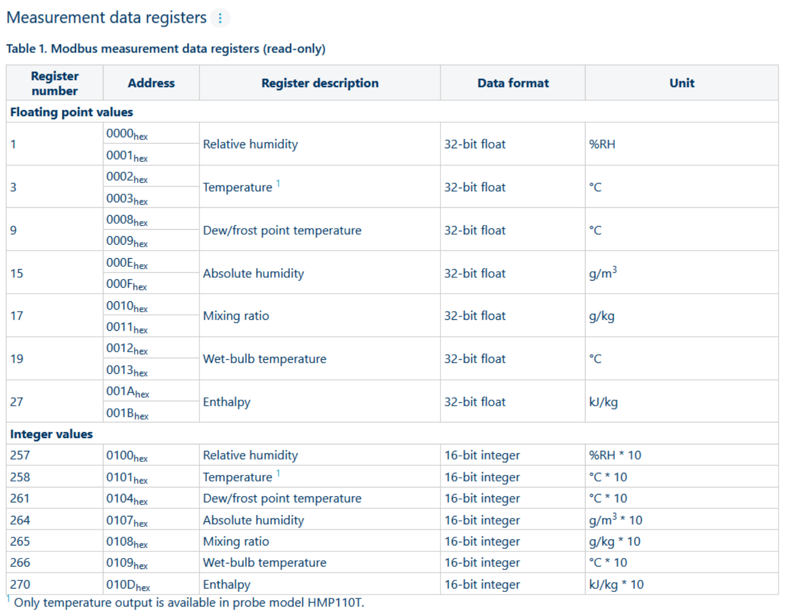
A.2.3 Status Registers

Table 54 Modbus Status Registers (Read-Only)

Register Number (Decimal)	Address (Hexadecimal)	Register Description	Data Format	Notes
2049	0800_{hex}	Device status	16-bit	0 = Status OK. 1 = Critical error. 2 = Error. 4 = Warning.
2050	0801_{hex}	CO_2 status	16-bit	0 = Status OK. 2 = CO_2 reading not reliable. Appears during transmitter start-up. 256 = Measurement not ready. Appears during transmitter start-up.

Multiple statuses can be present simultaneously. In those cases, the value of the status register is the sum of the status values. For example, the value of the device status register is **6** if a warning (**4**) and an error (**2**) are present simultaneously.

GMP252

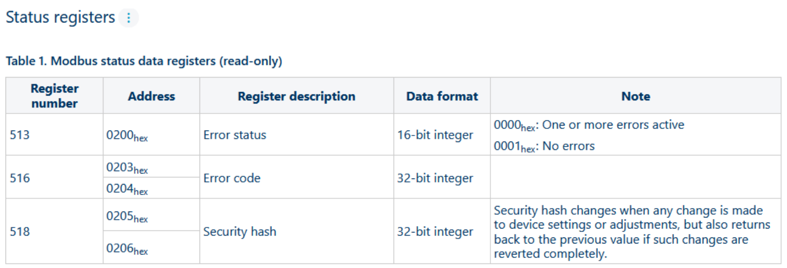


Measurement data registers

Table 1. Modbus measurement data registers (read-only)

Register number	Address	Register description	Data format	Unit
Floating point values				
1	0000_{hex} 0001_{hex}	Relative humidity	32-bit float	%RH
3	0002_{hex} 0003_{hex}	Temperature [1]	32-bit float	°C
9	0008_{hex} 0009_{hex}	Dew/frost point temperature	32-bit float	°C
15	$000E_{hex}$ $000F_{hex}$	Absolute humidity	32-bit float	g/m^3
17	0010_{hex} 0011_{hex}	Mixing ratio	32-bit float	g/kg
19	0012_{hex} 0013_{hex}	Wet-bulb temperature	32-bit float	°C
27	$001A_{hex}$ $001B_{hex}$	Enthalpy	32-bit float	kJ/kg
Integer values				
257	0100_{hex}	Relative humidity	16-bit integer	%RH * 10
258	0101_{hex}	Temperature [1]	16-bit integer	°C * 10
261	0104_{hex}	Dew/frost point temperature	16-bit integer	°C * 10
264	0107_{hex}	Absolute humidity	16-bit integer	g/m^3 * 10
265	0108_{hex}	Mixing ratio	16-bit integer	g/kg * 10
266	0109_{hex}	Wet-bulb temperature	16-bit integer	°C * 10
270	$010D_{hex}$	Enthalpy	16-bit integer	kJ/kg * 10

[1] Only temperature output is available in probe model HMP110T.



Status registers

Table 1. Modbus status data registers (read-only)

Register number	Address	Register description	Data format	Note
513	0200_{hex}	Error status	16-bit integer	0000_{hex}: One or more errors active 0001_{hex}: No errors
516	0203_{hex} 0204_{hex}	Error code	32-bit integer	
518	0205_{hex} 0206_{hex}	Security hash	32-bit integer	Security hash changes when any change is made to device settings or adjustments, but also returns back to the previous value if such changes are reverted completely.

HMP60

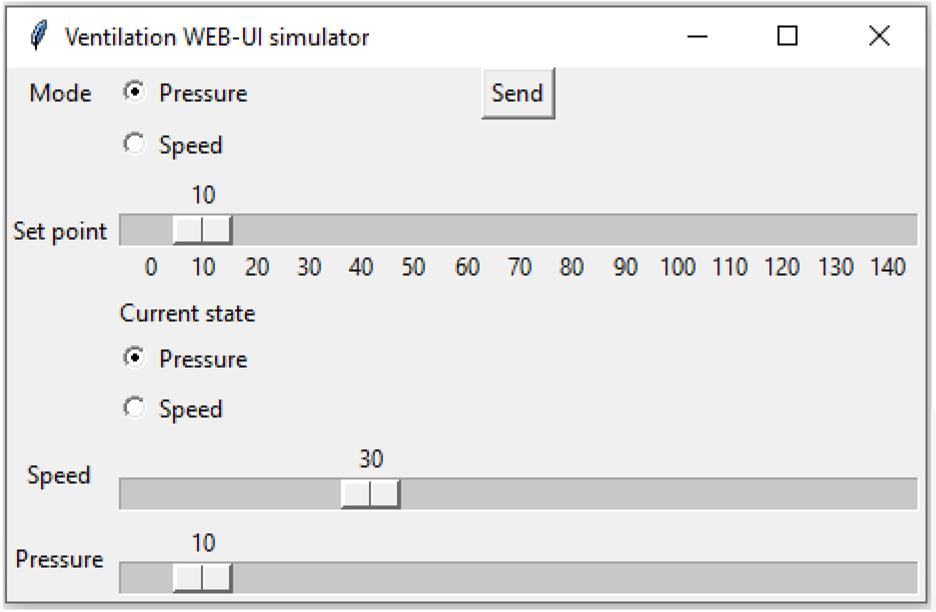
Python programs

- Packages to install:
  - Numpy
    ```
    pip install numpy
    ```
  - MQTT
    ```
    pip install paho-mqtt
    ```
  - tk

    should be installed by default – if not
    ```
    pip install tk
    ```

WEB UI-simulator

- LPC1549 can be controlled either using local UI – buttons and LCD or by sending JSON messages over MQTT
- A python program can be used to simulate the web interface. The program can be used to set parameters and it subscribes to status messages that the ventilation controller sends
- Note that implementing the WEB UI is not part of this course



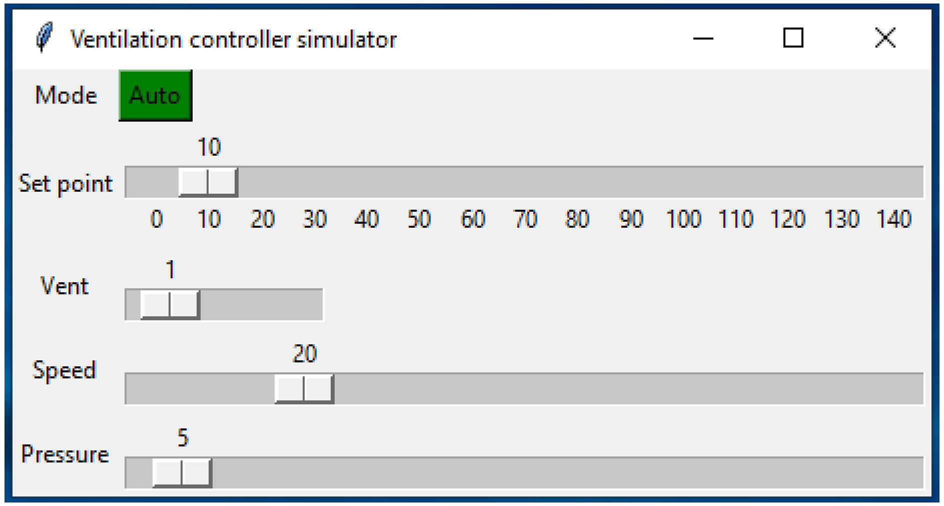
```
# the following can be given as command line parameters
default_broker = '192.168.1.10'  # 1st parameter
default_port = 1883   # 2nd parameter
default_pub_topic = "controller/settings"  # 3rd parameter
default_sub_topic = "controller/status"  # 4th parameter
```

How to run controller?

- python controller.py [broker ip] [port] [publish topic] [subscribe topic]
- If no parameters are given then defaults that are set in the source are used

Ventilation controller simulator

- This python programs simulates ventilation controller and generates status messages for the UI
- This program can be used to test the WEB UI if ventilation controller is not available



How to run ventilation simulator?

- python vss.py [broker ip] [port] [publish topic] [subscribe topic]
- If no parameters are given then defaults that are set in the source are used

Example project

- Checkout from gitlab.metropolia.fi:
  ```
  git clone https://gitlab.metropolia.fi/lansk/modbus_mqtt.git
  ```
- Open MCUXpresso and switch workspace to modbus_mqtt-directory
  - Import existing projects into the workspace