Air Mass Flow Sensor

Release 0.0.1

GianAndrea Mueller

Apr 04, 2021

CONTENTS

This project contains the software for a student experiment at ETH Zürich based on sponsorship by Sensirion AG. The student experiment aims to illustrate the sensor principle of a thermal air mass flow meter, which is investigated mainly from the control point of view. The students are required to implement control algorithms for the regulation of the heater.

The software provided here mainly does two things:

1. Communication with all attached devices:
    1. Sensirion SHT temperature sensors connected to a Sensirion Sensor Bridge
    2. A Sensirion SFM massflow meter
    3. A custom built heater being driven with a PWM signal

    This task is taken over by the *Setup* class. It handles all interactions with the hardware and for this purpose makes use of the different drivers, as seen on page *Drivers*.
2. Allowing interactions:
    1. Displaying the current system status
    2. Walking the student through different steps of the experimentation
    3. Handling interactions with the setup

    These tasks are solved with a PyQt5 based graphical user interface as described in section *GUI*.

CHAPTER

ONE

TROUBLESHOOTING

When experiencing issues with soft- or hardware consider section *FAQ*.

CHAPTER

TWO

ACKNOWLEDGEMENTS

The icons in the GUI are made by Yusuke Kamiyamane and used under CC BY 3.0.

The GUI frontend, QT 5.0 is used under LGPL 3.0.

2.1 Installation

2.1.1 Windows 10

Setup GUI

1. Install python 3.8 or newer.
2. Set your PATH variable such that it includes the *Scripts* folder of your python installation.
3. Go to *01_SETUP/WINDOWS* and run *py -m setup* in the cmd shell.
4. Install Sensirion Control Center to allow the sensor bridge to communication with the computer. Important: Select yes when asked for driver installation at the end of the process.
5. Find the finished executable at *02_SOFTWARE/disp*.

Setup Sensirion USB Sensor Viewer

1. Install the Sensirion USB Sensor Viewer.
2. Select COM HARDWARE: *RS485/USB Sensor Cable*.
3. Select Sensor Product: *DP Sensors (SDP3x/SDP8xx)*.
4. Execute *Drivers/identify_differential_pressure_sensor.py* with a local python environment. This will give you an overview of all connected sensors and print the comport ID of the pressure sensor in the final line.
5. Enter the previously found comport number in the RS485 Sensor viewer and connect.

Debugging

The installation is based on pyinstaller. It is configured via the *02_SOFTWARE/main.spec* file. Set *debug=True* and *console=True* to receive informative output on the cmd shell upon launching the program.

2.2 Setup

class setup.**Setup**(*config: Utility.ConfigurationHandler.ConfigurationHandler*)

The Setup handles all interaction with the hardware of the experiment.

Parameters

- **serials** (*dict*) – Dictionary of device names and corresponding USB serials.
- **t_sampling_s** (*float*) – Measurement sampling time in seconds.
- **interval_s** (*float*) – Total buffered time interval in seconds, which in combination with the sampling time defines the number of stored measurements.

_measure_normal_mode() → dict

Measures all devices.

Returns A dictionary with all measured signals.

_measure_simulation_mode() → dict

When no devices are connected random values are generated instead of actual measurements.

Returns A dictionary with all signals

_setup_measurement_buffer() → *Utility.MeasurementBuffer.MeasurementBuffer*

Defines the set of recorded signals and creates a corresponding MeasurementBuffer.

Returns An instance of MeasurementBuffer containing a deque instance for every signal.

See also:

Module *Utility.MeasurementBuffer.MeasurementBuffer*

close() → None

Closes all connected devices.

disable_output() → None

Disable the output for either pwm or pid mode.

enable_output(*desired_pwm_output=0*) → None

Enables the output in either pwn or pid mode.

Parameters **desired_pwm_output** (*float*) – Optionally enable pwm mode with a predefined nonzero output.

get_current_flow_value()

Getter for last set target flow value.

Returns Last set target flow value in normalized units.

measure() → None

Handles measuring and storing signals depending on the system mode and handles updating the PID controller output.

See also:

_measure_simulation_mode() *_measure_normal_mode()* *Utility.MeasurementBuffer.MeasurementBuffer*

open() → None
Finds and opens all the USB devices previously defined within *self.serials* by their serial number. If one of the devices is not responsive or cannot be found, the setup is switching to simulation mode in which all measurements are simulated. This allows to test the GUI without any attached devices.

See also:

Module *Drivers.DeviceIdentifier.DeviceIdentifier*

reset_temperature_calibration() → None
Reset the current temperature offset to zero.

reverse_temp_sensors(*update=True*) → None
Reverse the order of the temperature sensors if the have been set up wrongly.

save_measurement_buffer(*folder*, *name*, *type='mat'*)
Saves the current measurement buffer to a file. :param folder: Destination folder. :param name: Name of the file. A time tag will be appended for uniqueness. :param type: To allow different export filetypes.

set_flow(*value*)
Interface to the SFC5xxx drive for defining the current flow setpoint

Parameters **flow** (*float*) – The desired massflow in normalized units, in [0, 1].

set_kd(*kd: float*) → None
Allows setting the Kd gain of the controller.

Parameters **kd** (*float*) – Kd gain of the controller

set_ki(*ki: float*) → None
Allows setting the Ki gain of the controller.

Parameters **ki** (*float*) – Ki gain of the controller

set_kp(*kp: float*) → None
Allows setting the Kp gain of the controller.

Parameters **kp** (*float*) – Kp gain of the controller.

set_pid_parameters(*kp=None*, *ki=None*, *kd=None*) → None
Interface to the pid-setting functionality of simple_pid.

Parameters

- **kp** (*float*) – Kp gain of the controller
- **ki** (*float*) – Ki gain of the controller
- **kd** (*float*) – Kd gain of the controller

set_pwm(*value: float*) → None
Safely sets the desired PWM value depending on the current system mode.

Parameters **value** (*float*) – Desired PWM value as a normalized value between 0 and 1.

See also:

setup.Mode

set_setpoint(*value: float*) → None
Allows to define the temperature difference setpoint.

Parameters **value** (*float*) – Positive value smaller 20 degrees.

set_temperature_calibration() → None
Record the current temperature offset, assuming steady state.

start_buffering() → None
Start recording measurements in the MeasurementBuffer and delete previously recorded measurements.

start_direct_power_setting() → None
Start pwm mode with the output set to off.

start_measurement_thread() → None
Creates a thread.Timer that schedules future measurements at the desired sampling time.

See also:

Utility.Timer.RepeatTimer

start_pid_controller(*setpoint=None*) → None
Start pid mode with the output set to off.

Parameters **setpoint** (*float*) – Can be used to define a new temperature difference setpoint.

stop_buffering() → None
Stop recording measurements in the MeasurementBuffer.

stop_measurement_thread() → None
Cancels the current measurement thread.

2.2.1 Setup Modes

class setup.**Mode**(*value*)
Defines a set of system modes.

1. IDLE: Before any of the experiment modes has been loaded the system is idle.
2. FORCE_PWM_OFF: In this mode the pwm can be set directly, but the output is currently turned off.
3. FORCE_PWM_ON: In this mode the pwm can be set directly.
4. PID_OFF: In this mode the pid parameters can be set, but the output is currently turned off.
5. PID_ON: In this mode the pid parameters can be set and the controller is allowed to set pwm values.

2.2.2 Additional Utility

Logging Facility

Utility.Logger.**setup_custom_logger**(*name: str*, *level: int*) → logging.log
Sets the logging format, level and name of the logger.

Parameters

- **name** (*str*) – Name of the logger.
- **level** (*int*) – Initial logging level.

Returns Returns a log.

Measurement Buffer

class Utility.MeasurementBuffer.**MeasurementBuffer**(*signals: list*, *sampling_time_s: float*, *buffer_interval_s: float*)

The MeasurementBuffer holds a number of deque instances, one for each recorded signal and manages them as a ring buffer, always keeping a record of the most up to date measurements, reaching back *buffer_interval_s* seconds.

Parameters

- **signals** (*list*) – List of signal names.
- **sampling_time_s** (*float*) – Measurement sampling time in seconds.
- **buffer_interval_s** (*float*) – Total buffered time interval in seconds which together with the sampling time defines the number of measurements to be stored.

clear() → None

Clears the buffer.

update(*measurement: dict*) → None

A buffer update is done by adding an entry to each signal buffer. Before the buffer is full this leads to an increase in length, afterwards the deque instances automatically forget their oldest entry in favor of the new one.

Parameters **measurement** (*object*) – Dictionary containing a value for each signal name

Timer

class Utility.Timer.**RepeatTimer**(*interval*, *function*, *args=None*, *kwargs=None*)

The RepeatTimer is a special type of timer thread that can be run indefinitely and executes a given function each time a specified interval has passed.

Note: Example of usage:

```
def dummyfn(msg="foo"):
    print(msg)

timer = RepeatTimer(interval=1, function=dummyfn)
timer.start()
time.sleep(5)  # During which 5 calls of dummyfn will happen.
timer.cancel()
```

---

run() → None

Method representing the thread's activity.

Overrides *Timer.run* such that we have a repeated timer.

2.3 Drivers

2.3.1 Device Identifier

class Drivers.DeviceIdentifier.**DeviceIdentifier**(*serials: dict*)

The DeviceIdentifier lists all connected USB devices and tries to identify all devices listed in *self.serials* with their respective serial port, which are subsequently available as *self.serial_ports*

Parameters **serials** (*dict*) – Dictionary with USB names as keys and USB serials as values.

Note: If the USB serials are unknown when launching the program first simply supply a dictionary with placeholders. DeviceIdentifier supplies information on all available devices upon failing to find one of the devices in the serials dictionary.

Warning: Windows detects USB serials differently than Linux. As experienced in the creating of this software, a serial read on a Linux system must be appended with the letter 'A' to be detected on a Windows system. To offer platform independence the serials must be given in 'Linux-Form' and are automatically appended with the letter 'A' when the program is run on Windows.

open()

Detects the current os. For Windows the letter 'A' is appended to the Linux-specific serial of the device. For Linux an additional tty-setup script is executed to allow detection of all USB devices. After that the serials of the available devices are compared and linked to *self.serials*.

Returns Returns True if all devices listed in self.serials could be found, False otherwise.

2.3.2 Platforms

Platform Base

class Drivers.PlatformBase.**PlatformBase**(*name: str*)

Abstract base class for all platforms used in this project.

Parameters **name** (*str*) – Each platform must have a unique name.

close() → None

Disconnects the platform if it is currently connected.

open() → bool

Attempts to connect the platform and reports success.

Returns True if connected succesifully, False otherwise.

Shdlc IO Module - The Heater

class Drivers.Shdlc_IO.**ShdlcIoModule**(*serial_port: str*, *baudrate=115200*, *slave_address=0*, *input_pins=None*)

Bases: *Drivers.PlatformBase.PlatformBase*

The ShdlcIoModule represents the custom Sensirion HDLC IO Box that allows driving the heater with a PWM output.

Parameters

- **serial_port** (*str*) – Comport the IO box is connected to
- **baudrate** (*int*) – Baudrate of the connection
- **slave_address** (*int*) – Slave address
- **input_pins** (*list*) – list of integers of the input pins

connect() → bool
Attempts to connect the ShdlcIoModule

Returns True if connected successifully, False otherwise.

disconnect() → None
Sets all outputs off.

get_analog_input() → float
Measures actual voltage on ADC input

Returns A voltage between 0-10V

get_analog_output() → float
Gets actual voltage for DAC output

Returns A voltage between 0-10V

get_digital_io(*io_bit: int*) → bool
Reads a digital io pin.

Parameters **io_bit** (*int*) – Output bit to read

Returns True if digital bit is set.

get_pwm(*pwm_bit: int*) → int
Reads the current pwm setting.

Parameters **pwm_bit** (*int*) – The index of the PWM channel to be used (0, 1)

Returns A duty cycle value between 0 - 65535

is_connected() → bool
Attempts to read the serial number of the device to check if it is connected.

Returns True if connected, False otherwise.

set_all_digital_io_off() → None
Turns of all digital pins.

set_analog_output(*value: int*) → None
Sets the analog output

Parameters **value** (*int*) – A voltage between 0-10V

set_digital_io(*io_bit: int*, *value: bool*) → None
Sets a digital output pin.

Parameters

- **io_bit** (*int*) – The digital pin index to set
- **value** (*bool*) – True if set to on

set_pwm(*pwm_bit: int*, *dc: int*) → None
Sets the pwm output

Parameters

- **pwm_bit** (*int*) – The index of the PWM channel to be used (0, 1)
- **dc** (*int*) –

Returns A duty cycle value between 0 - 65535

Sensirion Sensor Bridge (EKS)

class Drivers.SHT.**EKS**(*serial_port: str*)
Bases: *Drivers.PlatformBase.PlatformBase*

EKS represents a Sensirion Sensor Bridge which is used to communicate to a range of sensor via I2C.

Parameters **serial_port** (*str*) – Name of the port to which the EKS is connected.

connect() → bool
Attempts to connect to the EKS.

Returns True if connected sucessifully, otherwise the encountered exception will be returned.

connect_sensors() → None
Attempts to connect sensors at both EKS ports.

disconnect() → None
Closes all connected sensors.

is_connected() → bool
Tests if the EKS is responsive.

Returns True if the EKs serial number can be read, False otherwise.

measure() → list
Measures both channels if a sensor is attached

Returns A list of measured values.

2.3.3 Sensors

Sensor Base

class Drivers.SensorBase.**SensorBase**(*name*)
Abstract base class for all sensors used in this project.

Parameters **name** (*str*) – Each sensor must have a unique name.

close() → None
Disconnects the sensor if it is currently connected.

open() → bool
Attempts to connect the sensor and reports success. :return: True fi connected successifully, False otherwise

Sensirion Humidity Temperature (STH)

class Drivers.SHT.**SHT**(*device_port: int*, *shdlc_device: sensirion_shdlc_sensorbridge.device.SensorBridgeShdlcDevice*, *name='SHT'*)

Bases: *Drivers.SensorBase.SensorBase*

SHT represents either an SHT85 or an STH31 of the Sensirion Humidity Temperature (SHT) sensor range, connected via the Sensirion Sensor Bridge (EKS).

Parameters

- **device_port** (*SensorBridgePort*) – EKS port, either ONE or TWO.
- **shdlc_device** (*SensorBridgeShdlcDevice*) – Instance of the controlling EKS.
- **name** (*str*) – Name of the sensor.

_convert_humidity(*data: bytearray*) → float

Converts the raw sensor data to the actual measured humidity according to the data sheet Sensirion_Humidity_Sensors_SHT3x

Parameters **data** (*bytearray*) – 2 bytes, namely number 4 (humidity MSB) and 5 (humidity LSB) of the answer delivered by the sensor.

Returns The relative humidity measured by the sensor in percent.

_convert_temperature(*data: bytearray*) → float

Converts the raw sensor data to the actual measured temperature according to the data sheet Sensirion_Humidity_Sensors_SHT3x

Parameters **data** (*bytearray*) – 2 bytes, namely number 1 (temperature MSB) and 2 (humidity LSB) of the answer delivered by the sensor.

Returns The temperature measured by the sensor in degrees Celsius.

connect() → bool

Attempts to connect the sensor and signals success by blinking the corresponding port's LEDs.

Returns Returns True if connected successifully, False otherwise.

connect_sensor(*supply_voltage: float*, *frequency: int*) → None

Connection of a sensor attached to the sensirion sensor bridge according to the quick start guide to sensirion-shdlc-sensorbridge.

Parameters

- **supply_voltage** (*float*) – Desired supply voltage in Volts.
- **frequency** (*int*) – I2C frequency in Hz

disconnect() → None

Called by SensorBase.close upon deletion of this class. Switches supply off.

is_connected() → bool

Check if the sensor operates correctly

Returns True if the status register can be read, False otherwise

measure() → dict

Implementats a single shot measurement according to the SHT3x datasheet. A high repeatability measurement with clock stretching enabled is performed.

Returns Dictionary containing temperature in degrees Celsius and relative humiditiy in percent.

read_status_reg() → bytearray

Reads the status register

Returns Status register value as bytearray.

Sensirion Mass Flow Meter / Controller (SFM / SFC)

class Drivers.SFX5400.**SFX5400**(*serial_port: str*, *name='Sfc5400'*)
Bases: *Drivers.SensorBase.SensorBase*

SFX5400 represents either a Sensirion Flow Controller (SFC) or a Sensirion Flow Meter (SFM) of type 5400.

Parameters

- **serial_port** (*str*) – Name of the comport the SFX is connected to.
- **name** (*str*) – Name of the device.

connect() → bool
Attempts to connect to the SFX and reports success.

Returns True if connected successifully, False otherwise.

disconnect() → None
Disconnects the device.

get_device_information(*index: int*) → str
Retrieves device information depending on the index given.

Parameters **index** (*int*) – Integer between 1 and 3 to request on of the data below:

1. Product Name
2. Article Code
3. Serial number

Returns String containing the requested information.

is_connected()
Checks if the device is connected by reading its serial number.

Returns True if connected, False if not.

measure() → dict
Measures the current mass flow.

Returns Dictionary containing the measurement.

set_flow(*setpoint_normalized: float*) → bool
Sets the current desired mass flow if a flow controller is connected.

Parameters **setpoint_normalized** (*float*) – Flow setpoint as normalized input between 0 and 1.

Returns True if set successifully, False if exception occured.

2.4 GUI

The structure of the graphical user interface can be described as follows: The outermost layer is within the main file, which deploys the Qt application and loads the main window.

The main window then controls the different experiment pages `GUI.ExperimentPages.ExperimentPage`, one for each experimentation step, with a stacked layout and manages the switching between those pages. The pages are built up from a series of widgets as defined in sections *Widgets* and *Live Plots*.

2.4.1 Main Window

class GUI.MainWindow.**MainWindow**(*setup:* setup.Setup, **args*, ***kwargs*)

Defines the main window of the application.

Parameters **setup** (Setup) – Instance of Setup to allow access to sensors and actuators.

_calibrate_temperature() → None
Toolbar action; Allows to set the current delta T to zero by saving the current temperature difference and subtracting it from the second temperature measurement.

_change_competition_mode() → None
Toolbar action; Allows to set the current view to competition mode.

_go_to_next_view() → None
Toolbar action; Switches to the next view in the main layout stack.

_go_to_previous_view() → None
Toolbar action; Switches to the previous view in the main layout stack.

_reset_plots() → None
Toolbar action; Allows to reset all visible plots to their original view.

_reset_temperature_calibration() → None
Toolbar action; Allows to reset the calibration temperature difference to zero.

_reverse_temperature_sensors() → None
Menu action; Allows to switch the order of the temperature sensors if the hardware setup is the wrong way around.

_save_measurement_buffer()
Toolbar aciton; Allows to save the measurement buffer as a Matlab .mat file.

_start_recording() → None
Toolbar action; Allows to restart recording measurements. Clears the buffer.

_stop_recording() → None
Toolbar action; Allows to stop recording measurements and thus freeze the plots. :return:

_toggle_massflow(*state=None*) → None
Toolbar action; Allows to turn the massflow output on or off

Parameters **state** (*bool*) – Set True to turn the output state to on, or False vice versa.

_toggle_output(*state=None*) → None
Toolbar action; Allows to turn the pwm output on or off.

Parameters **state** (*bool*) – Set True to turn the output state to on, or False vice versa.

_toggle_setpoint()
Toolbar action; Allows to change the temperature difference setpoint

setup_status_bar() → None
Sets up a status bar displaying sponsor layouts and tips for hovered over widgets.

setup_tool_bar() → None
Adds a toolbar to the main window and defines a set of actions for it.

2.4.2 Widgets

class GUI.CustomWidgets.Widgets.**FancyPointCounter**(*setup*, **args*, ***kwargs*)
Bases: PyQt5.QtWidgets.QLCDNumber

Custom version of the QLCDNumber.

property value

class GUI.CustomWidgets.Widgets.**CompetitionWidget**(*setup: setup.Setup*, *start_recording_action: Callable*, *stop_recording_action: Callable*, *enable_output_action: Callable*, **args*, ***kwargs*)
Bases: GUI.CustomWidgets.BaseWidgets.FramedWidget

The CompetitionWidget allows to start a recording of the current performance and displays the number of points reached.

_update_process_values(*running_time_s*) → None
Container function for updates that are specific to the inheriting widgets

_update_progress() → None
Update the progressbar to show the current remaining time. If the recording interval has passed the process is stopped.

reset() → None
Reset the competition widget upon reloading it.

class GUI.CustomWidgets.Widgets.**StatusWidget**(*setup*, **args*, ***kwargs*)
Bases: GUI.CustomWidgets.BaseWidgets.FramedWidget

The StatusWidget displays the current temperatures, flow and temperature difference measured.

_update_lcds() → None
Updates the displayed values.

2.4.3 Live Plots

class GUI.CustomWidgets.LivePlots.**LivePlotSignal**(*name: str*, *identifier: str*, *color: str*, *width=1*)
A LivePlotSignal stores all the information needed to identify and plot a single signal.

Parameters

- **name** (*str*) – Name of the signal, to be displayed on the legend of the plot the signal is shown on
- **identifier** (*str*) – Identifier of the signal, used to retrieve the signal from the measurement buffer of the setup
- **color** (*str*) – Color of the plotted line used to instantiate the corresponding pen
- **width** (*float*) – Width of the plotted line used to instantiate the corresponding pen

Note: Selecting integer values for the width parameter results in smoother plots.

class GUI.CustomWidgets.LivePlots.**LivePlotWidget**(*setup: setup.Setup*, *title: str*, *ylabel: str*, *ylims: Tuple*, **args*, ***kwargs*)

Bases: pyqtgraph.widgets.PlotWidget.PlotWidget

The LivePlotWidget makes use of pyqtgraph to allow plotting a number of signals. It automatically updates.

Parameters

- **setup** (*Setup*) – Instance of the current setup to allow access to the measurement buffer
- **title** (*str*) – Title of the plot
- **ylabel** (*str*) – Label of the y-axis
- **ylims** (*Tuple*) – Limits of the y-axis

add_signals(*signals: list*) → None
Add a list of signals to the plot.

Parameters **signals** (*list*) – List of LivePlotSignals

reset_plot_layout() → None
Allows to reset the plot layout to the original view

update_plot_data()
Handles the updating of a LivePLotWidget.

class GUI.CustomWidgets.LivePlots.**LivePlotWidgetCompetition**(*setup: setup.Setup*, *title*, *ylabel*, *ylims*, **args*, ***kwargs*)

Bases: *GUI.CustomWidgets.LivePlots.LivePlotWidget*

Specialized LivePLotWidget allowing only two signals and adding color between the two corresponding lines. Used to visualize the integral of the control error.

add_signals(*reference_signal*, *actual_signal*)
Add a list of signals to the plot.

Parameters **signals** (*list*) – List of LivePlotSignals

update_plot_data()
Handles the updating of a LivePLotWidget.

class GUI.CustomWidgets.LivePlots.**PlotWidgetFactory**(*setup*)

Bases: object

The PlotWidgetFactory defines a simple interface for creating instances of previously defined LivePlotWidgets.

2.5 FAQ

Below known issues, their possible causes and subsequent fixes are listed. The fixes are ordered by decreasing likelihood so go from top to bottom retrying if the problem has been solved after every step.

2.5.1 Why don't I see real data?

Whenever the GUI is launched the setup tries to connect to all sensors. If that fails only simulated measurement values are shown. This can happen for the following reasons:

- The hardware does not have power.
    1. Check whether the experiment is plugged in.
    2. Check whether the power switch on the back is turned on.
    3. Check whether the fuses are intact.
- One of the USB devices is not connected.
    1. Connect the setup to your computer and validate that new devices are registered. Check the serials entry in `Utility\config.json` to see how many devices are expected to connect.
    2. When working on Windows, it can happen that two devices are registered under the same comport ID. To check, open the device manager and see if a comport ID appears twice. To fix either reassign one of the comport IDs manually or simply reboot your computer.

2.5.2 Why does the temperature difference decrease when heating?

- The temperature sensors are registered in the wrong order.
    1. Navigate to the dropdown menu on the top left `Configuration -> Reverse Temperature Sensors`.

2.5.3 Why does the flow controller not deliver sufficient flow?

- The pressurized air supply has only limited pressure.
    1. Open the pressure reduction valve a bit more until 100slm of flow can be delivered.

2.5.4 Why . . . ?

- The program runs, the sensors are connected but nothing works as expected.
    1. Turn on debug mode, which allows you to view real time logs. See section *Debugging*.

PYTHON MODULE INDEX

U

INDEX

Symbols

A

C

D

E

F

G

I

L

M

O

P

R

S

U

V