# FWS
# Flight Weather Station

Projektpraktikum

Johannes Kasberger
0616782

Markus Klein
0726101

# Contents

# List of Figures



# List of Tables

# Chapter 1

# Introduction

## 1.1 Project idea

The idea to this project came up when I was flying my model helicopter. I was wondering how nasty the wind was these days and that it would have been better not to take off into the skies.

But where should I know the conditions from? The wind speed is always different than the values provided by online weather services.

That was the point I decided to build something on our own, submitting all the weather data to our website so one can decide beforehand wether to drive to our airfield or not.

## 1.2 Available products on the market

Having a look on the internet reveals that there are plenty of weather stations available. Nearly all of them provide the basic features like temperature and pressure and a LCD for presenting the current conditions and/or a small forecast.

Some more expensive devices even provide a wind sensor and can be connected to a computer via USB cable. The software shipped with these devices also supports uploading of this data to a website.

So why spending so much time for creating your own product? Comparing the specifications of the wind speed sensors for instance reveals that these very cheap sensors only have limited quality. Also detecting the wind direction was not available in most products, which is essential for airfield applications.

Another thing is extensibility. We needed a system that can be extended by new sensors easily and that is flexible enough to generate graphs and diagrams as we like.

The result of this research process was this project called Flight Weather Station (abbr. FWS).

## 1.3 About FWS

As depicted in figure 1.1 FWS consists of the following parts:

- Sensors: These sensors capture the parameters: Wind speed, wind direction, temperature. Data are transmitted via cable.
- Controller Board: The controller board collects and processes the data from the sensors and transmits them to the analysis software via Ethernet LAN.
- Analysis (Software): This software collects the processed data of the Controller Board and performs statistical analysis. It generates diagrams and graphs which are sent to the webserver by a separate script.
- Webserver (Presentation): The generated diagrams are integrated into the website with a plugin for the CMS TYPO3 also written specifically for this project.

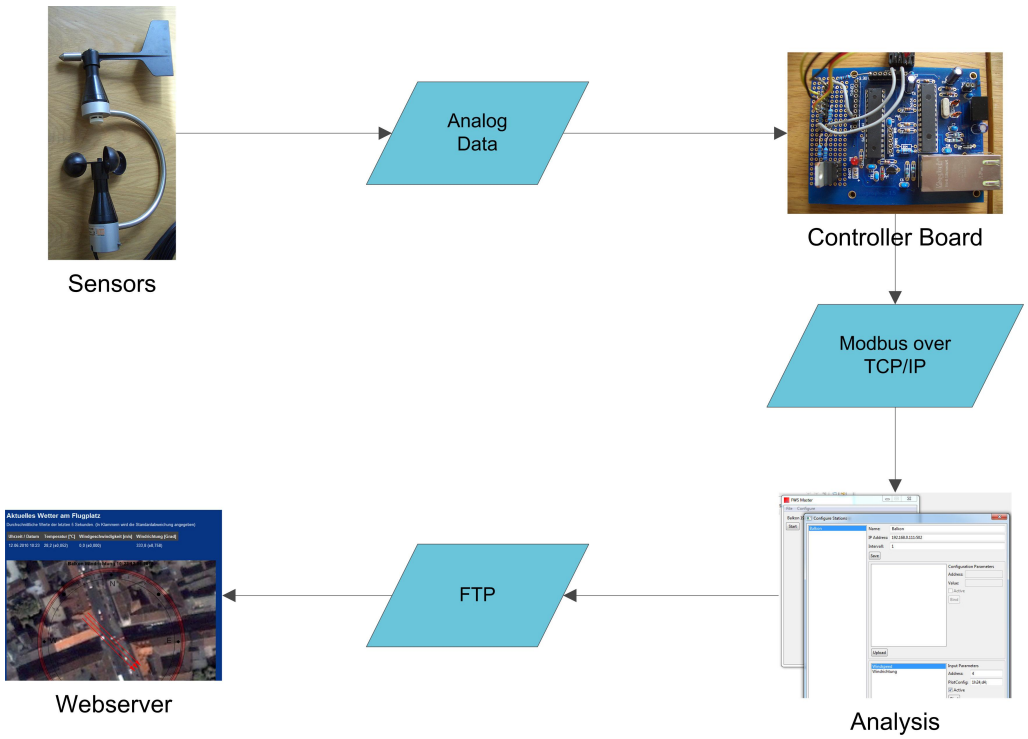



Figure 1.1: Module and protocol overview of FWS

# Chapter 2

# Slave

The term slave subsumes all parts of the the sensor capturing unit. The following explanations will discuss hardware and software individually to provide a more specific view of these parts.

## 2.1 Hardware

For capturing the monitored parameters wind speed, wind direction and temperature a product of ThiesKlima is used combining all needed sensors into one device. The connection between sensor device and microcontroller board is established by a 6-pin RJ plug.

The measurement data acquisition and TCP/IP handling is done by an ATmega328p chip, whereas the basic Ethernet communication is handled by an ENC28J60 chip from Microchip. Both chips are communicating via a SPI connection.[1]

The mainboard is operated at $3.3V$ via a voltage regulator. Due to the fact that the sensor device requires a supply voltage of $5V$, a separate voltage regulator has been added. This regulator is sourced by Con3 of the mainboard which can be switch on and off by software via the transistor connected upstream.

To be able to process the analog signals from the sensor device at the ATmega328p two 1:1 voltage dividers have been installed. This limits the maximum input voltage at the microcontroller's side to $2.5V$.

The signal generated by the reed contacts for capturing the wind speed is processed by using the Input Capture Unit of the microcontroller. The necessary pullup resistor is provided externally to ensure a fixed voltage ref-

[1]The mainboard is a product of http://tuxgraphics.org

erence. The internal pullup resistor may be too big due to production variations and may lead to undetectable signals.

The whole system runs at $12.5MHz$ system clock speed.

## 2.2 Software

### Main module

The main loop of the program uses polling to check for new data arriving from the Ethernet chip.

All other functionalities are implemented in submodules which are using interrupts and callback routines the return the results. In general problems of parallel processes are avoided as interrupt routines cannot be interrupted.

### Modbus module

This module consists of several submodules which encapsulate the access to the ENC28J60 via SPI as well as the implementation of the protocols TCP/IP, ICMP and ARP. These submodules where taken from tuxgraphics.org-library and adapted as needed.

The Modbus module itself is an in-house development. It currently supports the Modbus function codes for writing and reading single registers. It also has the possibility to request the device id. This function has not been implemented completely as the Java-Modbus-Library at the master does not provide this feature.

Registers (variables located in main module) that should be available for Modbus transfers are registered using the module's API.

### Analog signals - Temperature and wind direction

The Analog-Digital-Converter unit is encapsulated by the ADC sub module designed for subsequent capturing of multiple analog channels. Since the ATmega328p has only one ADC unit just one channel can be converted at once.

As this application needs to sample two channels this has to be done one after the other. To ensure safe channel switching the ADC unit is driven using single conversions with highest possible prescaler providing maximum resolution.

Naturally analog signals are subject to interference leading to non-precise results. In order to achieve a better data quality the ADC module samples the

current channel 256 times and calculates an average value before triggering the callback function and before switching to the next channel.

### Index signal - Wind speed

The wind speed is triggered by a Reed contact in the sensor device generating a frequency of $2.53Hz$ per $1m/s$ wind speed.

This signal is captured using the Input Capture Unit of 16-bit Timer 1. Counting the elapsed time between two index pulses allows to calculate the actual wind speed. This wind speed module also includes specific limits for valid data. For instance, in cases where there's hardly any wind and the wind wheel comes to stop right on top of the Reed contact, input capture interrupts are triggered permanently causing the calculation result to reach unfeasible values. This cases can be avoided by dropping all results not matching the specified values of the wind speed sensor.

## 2.3 Configuration parameters and data persistence

All described parameters are permanently saved to internal EEPROM. In case of EEPROM error they are reset to their default values. To modify the parameters write them over Modbus (see address reference table in section 2.4). All changes are stored in EEPROM automatically.

### Sensor enable

The microcontroller has a configuration parameter for enabling/disabling the the sensor device completely by switching the connected transistor.

#### Read and Write address: 2

#### Possible values:

- 0 = disable
- 1 = enable (default)

## Network

The mainboard supports $10Mbit/s$ T-Base Ethernet LAN. The user can change the IP address according to his needs. The port 502 is fixed and is reserved for Modbus communication by standards.

- Possible values: any IPv4 IP address
- Default value: 192.168.0.111

**Write address: 0**

- Value high byte: First segment of IP address (e.g. 192)
- Value low byte: Second segment (e.g. 168)

**Write address: 1**

- Value high byte: Third segment of IP address (e.g. 0)
- Value low byte: Fourth segment (e.g. 111)

IP address change sequence: First write to address 0 then address 1! Changes will take effect when complete address has been received and Modbus response has been sent with old IP address.

## Temperature calibration

The temperature calculation uses the following formula:

$$Temperature = (k * Value_sensor + d)/div$$

Where the factors k, d and div are responsible for calibration. Sensor values are between 0 and 720. All factors are 32-bit values. This means 2 transmission per factor are necessary.

**Possible values:** any 32 bit signed value

**Default values:**

- $k = -1134$
- $d = 690000$
- $div = 1000$

**Change sequence:** Always write all 3 factors beginning with the high word in the order k, d and finally div. Changes will take effect when lower word of div has been received.

## 2.4 Modbus address and data structure reference

### Modbus Read Addresses:

| Address | Description | Valid values | Data format |
|---|---|---|---|
| 2 | Sensor enable | 0 (disable), 1 (enable) | 65535 |
| 3 | Wind direction | 0..3375 (steps of 225); 65535 = error | 6553.5 |
| 4 | Wind speed | 0..1000 (steps of 1); 65535 = error | 6553.5 |
| 5 | Temperature | 16-bit signed; 32767 = error | 6553.5 |
| 6 | errcnt (debugging) | 16-bit unsigned | 65535 |
| 7 | cnt (debugging) | 16-bit unsigned | 65535 |

Table 2.1: Modbus Read Addresses

### Modbus Write Addresses:

| Address | Description | Valid values | Data format |
|---|---|---|---|
| 0 | IP address high segments | 16-bit unsigned | 65535 |
| 1 | IP address low segments | 16-bit unsigned | 65535 |
| 2 | Sensor enable | 0 (disable), 1 (enable) | 65535 |
| 3 | k high word | 16-bit unsigned | 65535 |
| 4 | k low word | 16-bit unsigned | 65535 |
| 5 | d high word | 16-bit unsigned | 65535 |
| 6 | d low word | 16-bit unsigned | 65535 |
| 7 | div high word | 16-bit unsigned | 65535 |
| 8 | div low word | 16-bit unsigned | 65535 |

Table 2.2: Modbus Write Addresses

# Chapter 3

# Master

## 3.1 About the Master

The master is written in Java and uses the jamod Library[1] for the communication with the sensors using the modbus protocol. The view is created with the SWT Library[2]. The description of the master is separated in several parts:

- View (see 3.2)
- Configuration (see 3.3)
- Stations (see 3.4)
- History (see 3.5)

Before the collection process of the measurement values can be started it's necessary to create a configuration. In this configuration all sensors are defined. A sensor must be added to the master before it can be used. Each sensor has an individual name and ip address. In this context it's referred as a station. In the configuration phase it's necessary to specify the kind of data the stations collect. The data that is transferred from the station to the master is called Input Parameter. To configure the sensor it's possible to define Configuration Parameters. These values are transferred from the master to the sensor. To map the parameters to the memory in the station they must be bound to addresses. This happens individually for each station. So it's possible to collect different data from different stations. It's also possible to configure the kind of plots that are generated (see 3.3).

[1] http://jamod.sourceforge.net/
[2] http://www.eclipse.org/swt/

After the configuration of the stations the collection process can be started. Each station is polled in its own thread. One collector thread collects the data from the station threads, generates a text file with the information about the current values from the stations and draws the plots. The time between the polling of the stations and between generation of the output files can also be configured. As soon as the current day changes the values from the last day are aggregated to a history value. The values from the station are kept for two days so it's possible to build a plot based on hourly data from the last day.

## 3.2 View

The view is created with the help of SWT. SWT uses the os drawing apis to draw the widgets. So the look and feel of the application is very good integrated in the os it runs in.

### Screenshots

See Figure 3.1

## 3.3 Configuration

The views that support the configuration are 3.1a, 3.1b,

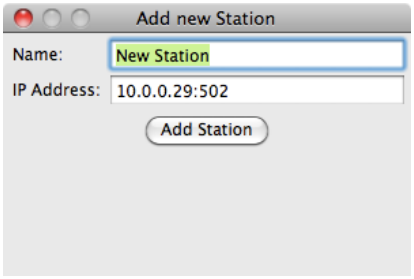
The result of the configuration phase is an xml file with all settings in it. The location of this file is os dependent.

- **OS X**: /Users/{username}/Library/Application Support/FWSMaster/settings.xml
- **Linux**: ~/.fws_master/settings.xml
- **Windows**: C:\{User Dir} \.fws_master \settings.xml

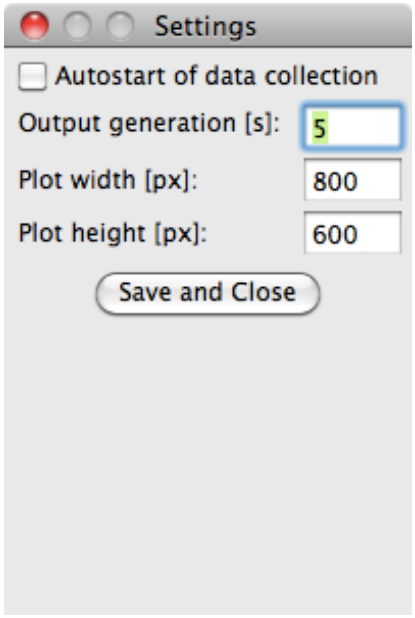
See example in Listing 3.1

Listing 3.1: Sample settings file

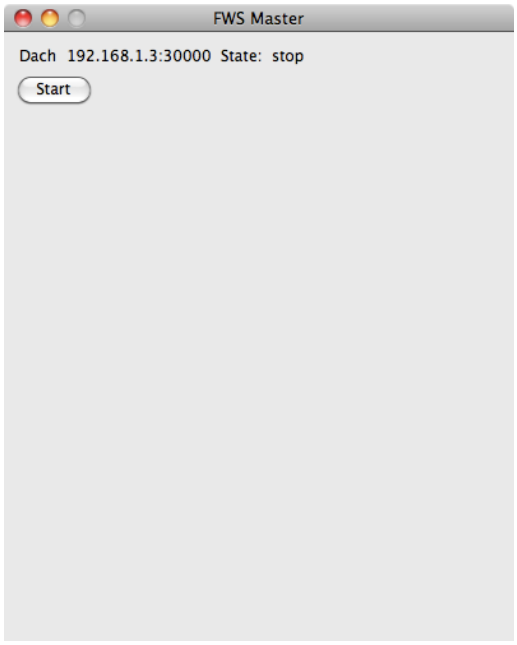
```
<?xml version="1.0" encoding="UTF—8" standalone="no"?>
<fws_config>
<path>/home/user/FWSMaster/output</path>
<generatortime>5</generatortime>
<autostart>false</autostart>
```




(a) Add new Station




(b) Configuration of the basic settings




(c) Quick overview of the stations and their status

Figure 3.1: Screenshots of FWS Master

```
<plotwidth>800</plotwidth>
<plotheight>600</plotheight>
<parameter typ="config">
<name>Messintervall</name>
</parameter>
<parameter typ="input">
<name>Temperatur</name>
<unit>°C</unit>
<format>6553.6</format>
<history>MAX</history>
</parameter>
<parameter typ="input">
<name>Windrichtung</name>
<unit>Richtung</unit>
<format>65536</format>
<history>AVG</history>
</parameter>
<station intervall="2" ip="192.168.1.3:30000" name="Dach">
<binding active="true" address="3" parameter="Messintervall"
    transfered="false" type="config" value="1000"/>
<binding active="true" address="0" parameter="Temperatur" plotconfig="
    d4;1h24;" type="input"/>
<binding active="true" address="1" parameter="Windrichtung" plotconfig
    ="d4;c1;h24;" type="input"/>
</station>
</fws_config>
```

## Parameters

There are two types of parameters. Input Parameters and Configuration Parameters. Each parameter has a unique name and a boolean value if it's enabled. The name is the identifier of the parameter so it has to be unique. Beside of these common attributes the Configuration and Input Parameters have further attributes. The parameters must be assigned to station addresses. This process is referred as binding a parameter to a station. To bind a parameter to a station, the user must define the address on the station where the parameter is saved.

### Input Parameter

Additional attributes of an Input Parameter:

- Unit
- Format
- History Function

**Unit** The Unit is just used for the description in the generated files. Available units are:

- speed $\frac{m}{s}$
- speed $\frac{km}{h}$
- frequenzy $Hz$
- direction
- temperature $°C$

**Format** The transferred value from the station is a 16 bit integer value. To be able to display floating point numbers it's possible to set the output format to the desired form. Example: The temperature equals $23.3°C$. The station measures the temperature and converts it to $233$. This integer value is transmitted to the master. The master converts the number back to the desired format. So this example would result in $23.3$.

**History Function** There are three different history functions available. For more details about these functions and how they are used please read 3.5.

- average
- minimum
- maximum

### Configuration Parameter

Additional attributes of a Configuration Parameter:

- value

**Value** A configuration parameter is a value that is transferred from the master to the slave. The value that is transferred is saved in the attribute value. This value must be an 16 Bit integer value.

## Plots

The plots are generated with the help of the jFreeChart Library [3]. The plots can be configured for every binding of input parameters. Each binding can have several plots assigned. It's also possible to plot more than one data into one plot. The plots are configured with a string. The simplest configuration looks like `h24;` With that configuration one plot is generated and the data for this plot are the values from the last 24 hours. The plots are generated in the output directory that can be set by the user.

It's possible to use different data ranges for the plots. To allow the user to choose one range there are three different time bases available:

- c - current
- h - hours
- d - days

**Current** Use the newest values for the plot. Currently this is only implemented for the wind direction. See example in figure 3.2.

**Hours** Use the values of the last hours for the plot. The amount of the hours is specified in the plot configuration. It's the number after the timebase. Example of an 24 hour plot see figure 3.3.

**Days** Same plot as with the hours timebase but the values from the last days are used. For more details about the history see section 3.5.

### Configuration Syntax

Each plot is specified by three different parts: [Number]CharacterNumber;

**First Number: ID** This number is optional and controls if more than one input parameter is drawn in one plot. All configured plots with the same number are drawn in the same plot.

[3] http://www.jfree.org/jfreechart/

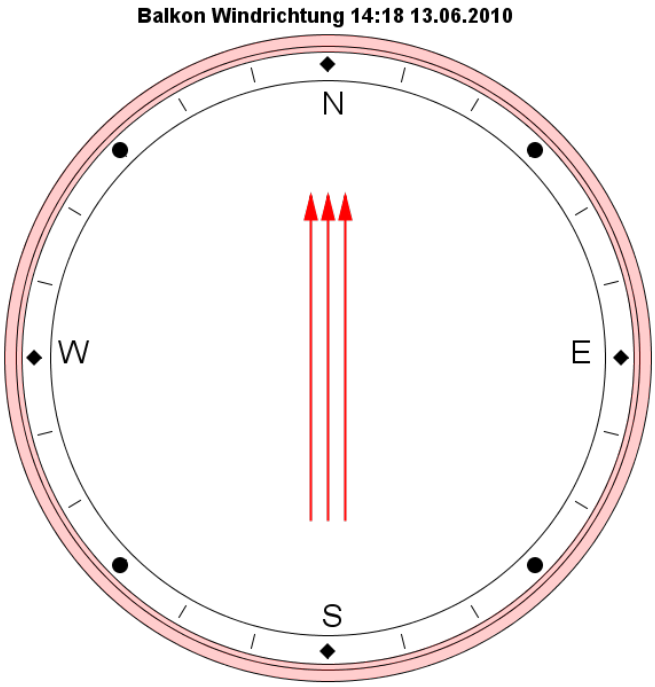



Figure 3.2: Current Wind direction plot

**Character: Timebase** Defines the timebase. Explained in paragraphs above.

**Second Number: Amount of data** The second number defines how much data will be in the plot. `d4;` will plot the last four days.

**End of Configuration** Each configuration must end with an `';'`.

**Example** For the configuration string `h24;1h24;d30;c1;d365;` following plots are generated:

- Last 24 hours
- Last 24 hours with other data with ID 1
- Last 30 days
- Current value
- Last 365 days

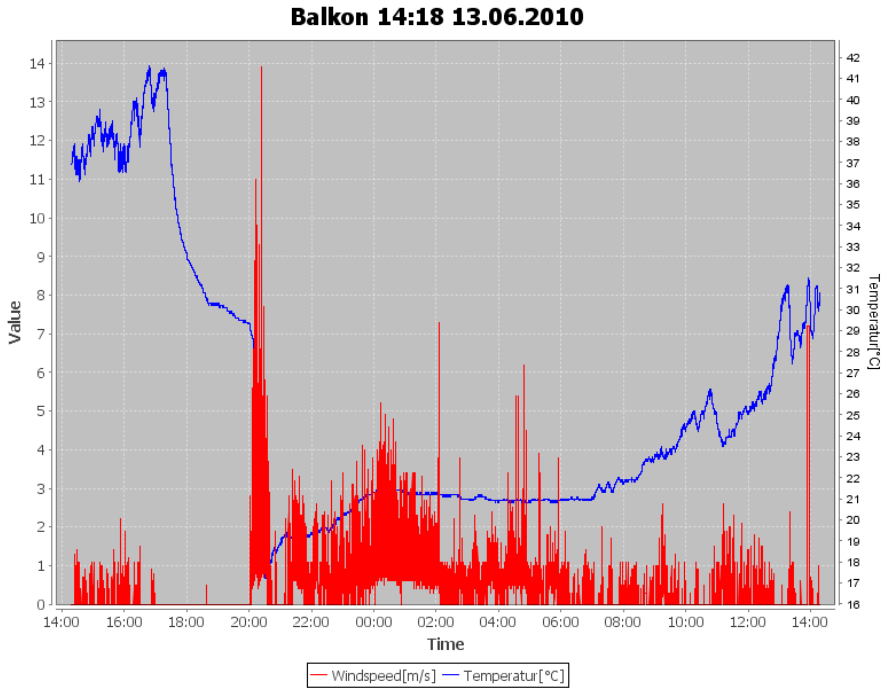



Figure 3.3: Last 24 hours of two separate parameters in one plot

### Detailed Information about the plots

If two different data sets should be plotted in one diagram there are two axis with an own scale generated. When more that two data sets should be drawn in one diagram they share one axis. So it's advisable to generate more plots with two datasets each in it to keep the diagrams tidy.

The direction parameters values are not connected with a line. Otherwise the diagrams would be complicated if the wind direction changes a lot.

The direction parameter values are mapped to a description of the direction. So 0° result in N(orth) see figure 3.4

## 3.4 Stations

After the stations are configured they are added to the station controller. This controller takes care for starting and pausing the station threads.

### Transfering the Data

Each station runs in it's own thread. This thread polls in its own interval the data values from the station. For each transmission a new TCP con-

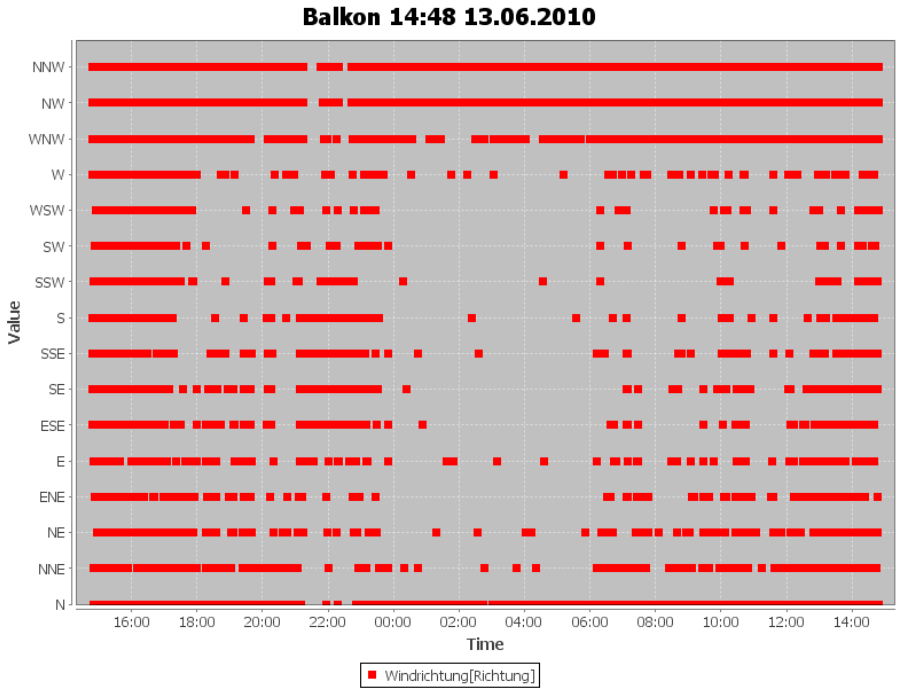



Figure 3.4: Axis description of direction parameter

nection is opened. After the value has been transferred the connection gets closed. Each station has a list of measurements. A measurement consists of a timestamp and a value. These lists are collected from the Output Generation Thread. If a station is paused its thread gets suspended and waits for the wake up signal from the controller.

The Modbus function that is used to transfer the data is read register.

### Transferring the configuration

When the user wants to upload a new configuration the station will call Write Register for each Configuration Parameter and reads back the answer of the station. If the answer equals the write register value the value is marked as transferred. A transferred value isn't uploaded again if it is unchanged.

### Change the IP Address

The IP Address represents a special configuration parameter. It is always mapped to the address 0 and 1 on the station. These two 16 bit registers are used to save the ip address. When the ip address has changed it's transferred

to the station. After the successful transmission of the two values the new ip is saved. Otherwise the old ip is kept.

### Collecting the Data

A data collector thread runs in background an collects the data from all the stations. For each station/parameter combination an own list is kept with all the recent values in it. But it's not the normal measurement that is kept. The measurements get converted in a simpler data type that can be serialized. This datatype has no references to other classes.

In the collector thread the text file is generated that contains the information about the current status of the stations. Each station is represented in this file. The information written in this file are the current values of the Input Parameters bound to that station. This value is the average of the measurements since the last run of the collector. To be able to see whether the value changes the standard deviation is calculated and written to the output. For example result file see listing 3.2. For each parameter a new line is started in the result file. Syntax is `name[unit]:value;standard deviation;`

Listing 3.2: Sample result file

```
16:53:36 13.06.2010
Balkon
Windspeed[m/s]:0.0;0.0;
Temperatur[°C]:25.474999999999998;0.0452267016866652;
Windrichtung[Richtung]:232.5;138.14518054963375;
====
Dach
Windspeed[m/s]:0.0;0.0;
Temperatur[°C]:26.672999999999998;0.03345864;
Windrichtung[Richtung]:232.5;138.14518054963375;
====
```

## 3.5 History

The collector adds the measurements to the history controller. This controller converts the values to the entries in the history. Each input parameter has two histories. One short term history and one long term history.

### Short term history

In the short term history there are the history entries from the last two days. This history is queried when plotting a diagram with the 'h' timebase. To this list all new measurements are added as soon as new data arrive from the collector. During this phase it's checked whether the day has changed since the last time new measurements have been added. If that is the case the day is transferred to the long term history. There are at least the last 24 hours in the short term history.

### Long term history

As soon as a new day has started the past day gets transferred to the long term history. To avoid to much values in this list one representing value is calculated for the day. This happens with one of the history functions listed in 3.3. During the calculation of the history value all measurements older than the last day are removed from the short day history. The representing value is added to the long term history.

### Storage of the history

The short and long term history are serialized to the hard disk. After new data have been added to the history it's saved to the hard disk. To prevent the user closing the master during the I/O operations a semaphore is used. [4]

An old history is kept to prevent loosing the history if the master crashes during writing the history. Before saving the history the thread locks the semaphore. After that it renames the current history to another filename. After that the new history is saved. The history is saved now so the semaphore is released.

If an exception encounters during loading the history the master tries to load the old history file. If that also results in an exception a new history is created.

## 3.6 Implementation details

The source code can be downloaded at github[5]. In the doc folder there is the javadoc generated source code documentation.

[4] Without that semaphore we often had corrupted history files because the master closed at a critical instant.

[5] http://www.github.com/schugabe/fws

In the following list you see which classes implement which functions. For a class diagram see figure 3.5.

- View: All classes that start with view
- Configuration: Parameter, InputParameter, ConfigParameter, StationInputBinding, StationConfigBinding, Station
- Save the configuration: PersitencePreferences, all classes with contentenhandler in its name
- Data collection: Station, MeasurementCollector, Measurement, StationInputBinding, MeasurementHistoryController, MeasurementHistory, MeasurementHistoryEntry
- Plotting and Output generation: MeasurementCollector, PlotBase, TimePlot, CurrentPlot
- ModBus: ModbusWrapper, Station

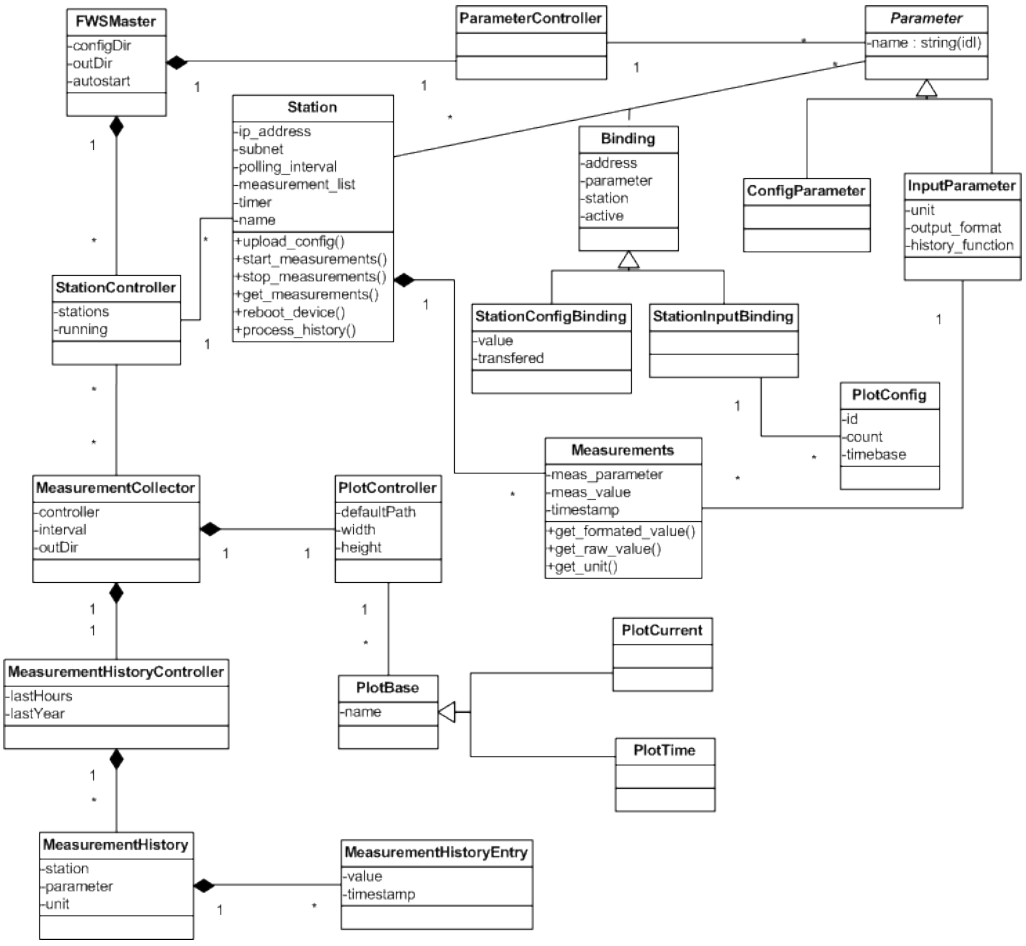



Figure 3.5: Class diagram of the master