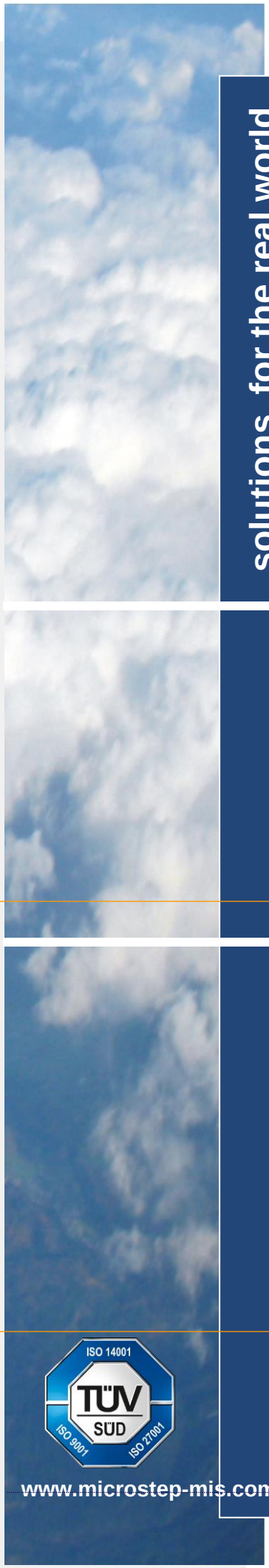


solutions for the real world

# AMS111

## AWS Setup

***Version: 3***

## User's Guide

***March 2020***

ISO 14001 TÜV SÜD ISO 9001 ISO 27001

www.microstep-mis.com

MicroStep - MIS

**Trademarks**
Windows is a registered trademark of the Microsoft Corporation.
All other trademarks or registered trademarks mentioned belong to their respective owners.

**Version of edition**

Version of edition: 1.5
Date of the edition: 2020-03-04

**Authors**

Editorial team: Mr. Csaba Ruman, MicroStep-MIS

**Address**

MicroStep-MIS
Monitoring and Information Systems
Čavojského 1
84104 Bratislava 4
Slovakia
Tel.: +421 2 602 00 100
Fax: +421 2 602 00 180
info@microstep-mis.com

MicroStep-MIS develops, manufactures and markets monitoring and information systems. The key fields of our activities are meteorology, aviation, environmental sciences, seismology, power engineering, civil defense but also information systems in tourism.

MicroStep-MIS operates worldwide. Our core customer groups are airports, meteorological and seismological institutes, environmental authorities, industry, power stations and electricity distribution companies.

# Table of contents

Table of contents ..... 2
1. Introduction ..... 4
1.1 System description and architecture ..... 4
1.2 Terminal board connections ..... 6
2. Installation ..... 6
2.1 System requirements ..... 6
2.2 Installation ..... 6
3. Using the AWS Setup ..... 7
4. Events ..... 9
5. Variables ..... 11
6. Analog channels ..... 17
6.1 10/12-bit single ended channels ..... 18
6.2 24-bit differential channels ..... 18
7. Digital channels ..... 20
7.1 Digital inputs ..... 20
7.2 Digital outputs ..... 21
8. Messages ..... 22
8.1 Date and time format ..... 23
8.2 Floating point format ..... 24
8.3 Coded floating point format (j) ..... 24
8.4 Coded floating point format (g) ..... 25
8.5 Coded floating point format (h) ..... 25
8.6 String format ..... 25
8.7 Binary format ..... 26
8.7.1 BFR file format ..... 27
8.8 Variable validity format ..... 28
8.9 Slash mode switch ..... 28
8.10 CRC checksum format ..... 29
8.11 Message checksum format ..... 29
8.12 Message XOR format ..... 30
9. Calculations ..... 31
9.1 Logic operators ..... 32
9.2 IF function ..... 33
9.3 EVENT function ..... 33
9.4 IFEVENT function ..... 33
9.5 IFEVENTS function ..... 33
9.6 SETDATE function ..... 34
9.7 FTYPE function ..... 34
9.8 CALIB function ..... 34
9.9 STATUS function ..... 35
9.10 STATBIND function ..... 36
9.11 STATEVENT function ..... 36
9.12 MESSIN function ..... 37
9.13 PACKER function ..... 37
9.14 DISC function ..... 38
10. Serial lines ..... 40
11. Statistics ..... 42
12. Macros ..... 43
12.1 Using macros in project ..... 43
12.2 Creating and editing macro definitions ..... 44
12.3 Macro evaluation algorithm ..... 47
13. Troubleshooting ..... 51
13.1 Message is too long ..... 51
13.2 Run multiple calculations asynchronously ..... 51
13.3 Run calculation on statistics event ..... 52
References ..... 53

## About this reference

The AWS Setup Users' Guide is dedicated to system integrators who use the AWS Setup software to create configurations for the MicroStep-MIS Data Loggers.

## Typographical conventions

Throughout this guide, several typographical conventions are used to help reader to follow instructions and identify the important information.

The special note for the reader, warning or example

**Note:** *If you set the **Interval** other than divisible without remainder (e.g. 25 minutes), the event will be executed when the remainder after dividing internal timestamp with the **Interval** shifted by **Time Shift** is zero.*

**Attention:** *Measured value is in **Volt [V]** units for single ended channels and **Millivolts [mV]** for the differential channels.*

**Warning:** *Turn off the device before removing the protective cover.*

**Example 1:** *To create an event, which logs the temperature to file every hour in 56 minute, set the event **Interval** to 1 hour and **Time Shift** to 56 minutes.*

Syntax of message format specifiers is described with the following convention:
Parts of the specifiers in square brackets [] are optional, parts in **bold** are constants (and have to be written as is) and parts in *italics* are varying.

| Syntax: |
| --- |
| **%**[**I**]*format***T** |

This format may be e.g. **%I***HMS***T** or **%***Dmy***T**

# 1. Introduction

MicroStep-MIS AWS Setup is designed to generate configuration files for the AMS 111, AMS 111 II and SAWS 111 Data Loggers.
AMS 111 II is an advanced micro-controller system for intelligent data measuring and collection and more other functions.
SAWS 111 is a compact version of data logger designed for standard, temporary and mobile meteorological stations, as well as for those applications where a small footprint but full functionality is required.
AWS Setup enables to configure Analog/Digital channels, serial communication lines, report messages, synchronous/asynchronous events, mathematical calculations and statistic operations over the measured values.

## 1.1 System description and architecture

**AMS 111 II** features:

Periodic measurements to variables:

- Analog input
  - Lower precision (10 bit, channels A0-A4)
  - Higher precision differential (24 bit, channels ADF0-ADF21)
- Digital input (channels DIN0-DIN11)
  - Counter
  - Timer
  - Parallel Gray code
  - Serial synchronous transmission (only channel DIN11)

Communication with digital sensors/data collection system:

- RS232
- RS485
- SDI-12
- Ethernet

**SAWS 111** features:

- Analog input
  - Higher precision differential (24 bit, channels ADF0-ADF6)
- Digital input (channels DIN0-DIN3)
  - Counter
  - Timer
  - Parallel Gray code

Communication with digital sensors/data collection system:

- RS232
- RS485
- SDI-12

**AMS 111 IV** features:

Periodic measurements to variables:

- Analog input
  - Lower precision (12 bit, channels A0-A4)
  - Higher precision differential (24 bit, channels ADF0-ADF21)
- Digital input (channels DIN0-DIN11)
  - Counter
  - Timer
  - Parallel Gray code
  - Serial synchronous transmission (only channel DIN11)

Communication with digital sensors/data collection system:

- 3xRS232
- 2xRS485
- 2xSDI-12
- Ethernet

**All** data loggers:

Statistics computations during specified interval:

- Minimum
- Maximum
- Average
- Sum

Communication messages and mathematical expressions are initiated in events.

Event sources:

- Periodic (Synchronous)
- On data receive (Asynchronous)
- On call from function (Asynchronous)
- On startup (Onetime)

Actions on event:

- Log variables
- Send messages
- Receive messages
- Run calculations

Configuration created by the AWS Setup software is a binary file, which has to be sent to the Data Logger. Some configurations may require additional files to be sent to the Data Logger. Files can be sent over the service serial line by XMODEM protocol or by SD card and then initiating command on service serial line. For easier configuration please refer to **AWS Service software [2]**, which is more user friendly to most of users.

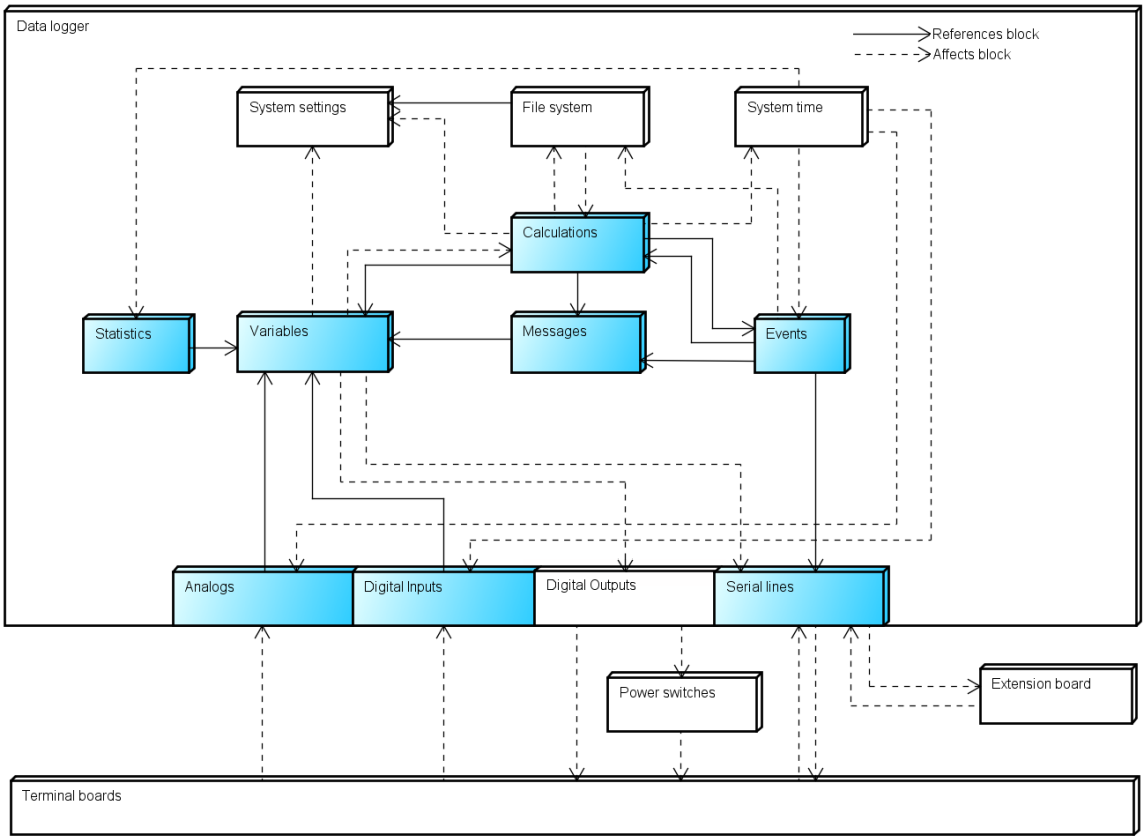



*Figure 1: Data logger configuration block diagram*

## 1.2 Terminal board connections

Data logger configuration sets the functionality of the channels, but it is required to know the exact position of the channel in the used Terminal board. Please refer to **AMS 111 Terminal boards [1]** for further help. The compact SAWS 111 Data logger has fixed terminals.

# 2. Installation

## 2.1 System requirements

Minimum system requirements:

| | |
|---|---|
| CPU: | 1 GHz |
| RAM: | 512 MB |
| OS: | Microsoft Windows 7 or higher |
| Framework: | .NET Framework 4.6.2 |

## 2.2 Installation

Run `Setup.exe` to start installation. Administrator privileges are required. You will be prompted to choose installer language.
Choose destination folder, then click next.
Choose start menu folder, then click Install.

After wizard completes installation, you can run program and create desktop shortcut by selecting checkboxes and clicking Finish button.

# 3. Using the AWS Setup

Configurations for Data loggers are made by defining the operations in few sections. These sections are Events, Variables, Messages, Analogs, Digitals, Serial lines, Calculations and Statistics.
To open a new project, click the **New Project** button and the mentioned sections will appear in tab **Configuration**. To show the contents of these sections, click on the triangle or double click the name of section.

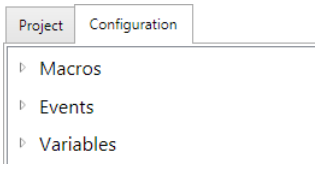



*Figure 2: The configuration tab and the sections*

Every section has a list of items. If the row is colored red, it means that there is an error in that row. If the row is yellow, the item has been changed since last **Apply**. These changes may be discarded by reverting back to previous version or accepted.

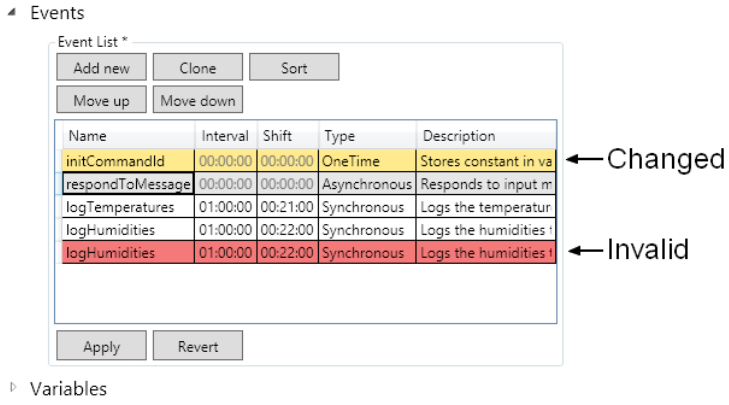



*Figure 3: Changed and invalid records*

New items are added by button **Add new**. The new item with default values will appear at the end of the list. To change a name of the item, select it, then on the right side change the name, then click **Rename** button.

Some sections must have a unique **name** to ensure that the correct item is referenced somewhere else. These names have limitations and must not contain some characters or start with numbers.

If the configuration is ready, binary file may be created by clicking the **Compile** button. The binary file is created in the same directory where the project file is saved. The name of the binary is the same as the name of the project, but with `.dat` extension. The **Compile compressed** button creates compressed (.tgz = .tar.gz) configuration file, which may be used in AWS Service Version 2.

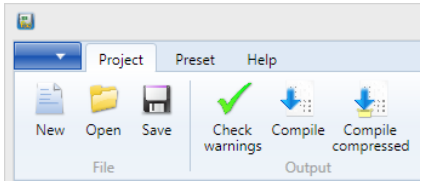



*Figure 4: Compile buttons*

You can check the project for errors by clicking **Check warnings** button.

The results will be shown at the bottom of the screen (**Error list**). The warnings are automatically checked with every compile. Checking warnings does not change any file as opposed to the compile options.

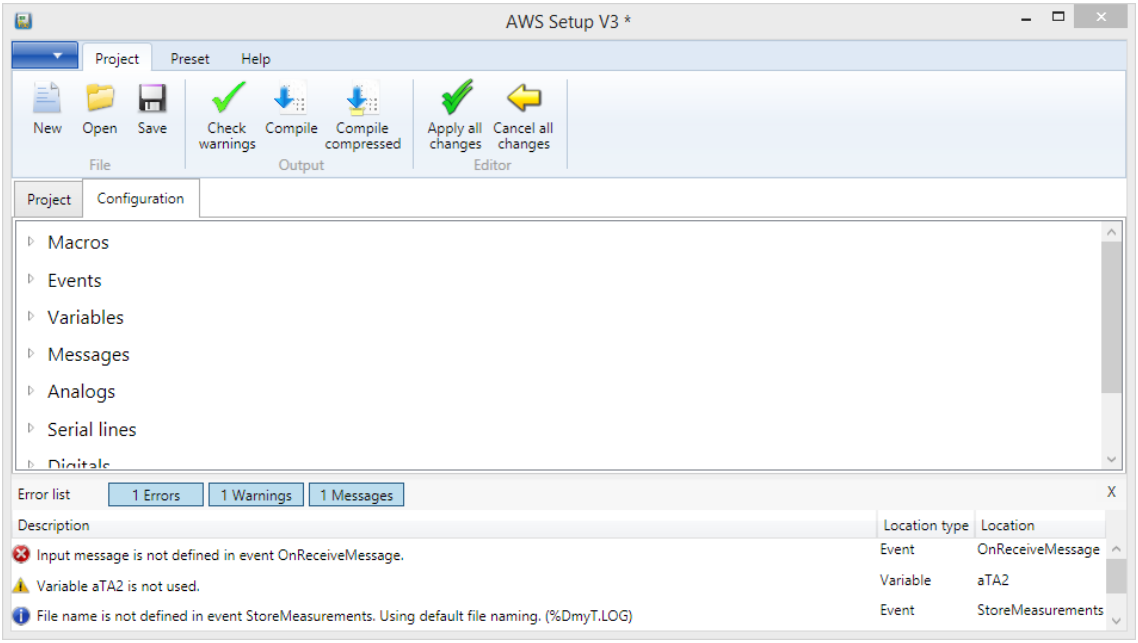



*Figure 5: Error list with an error, warning and a message*

The results in the Error list are divided into 3 categories:

- Error
- Warning
- Message

An **error** is indicating that something will definitely not work as expected.
A **warning** is indicating that something will probably not work as expected or you are wasting resources.
A **message** is indicating something that requires attention but otherwise it is completely acceptable.

You can disable/enable each category to show in list. By default the messages category is disabled.

# 4. Events

Events may be synchronous, asynchronous or called only at system initialization after reboot. Synchronous events are periodically called. Events are synchronized with clock. If the interval is set to zero, the event is never called by system, but it is possible to run this event from a calculation. Asynchronous events are triggered by receiving a message from a defined communication line. **One-time** events are started only once, at the system start. These events cannot be asynchronous.

**Example 1:** *To create an event, which logs the temperature to file every hour in 56 minute, set the event **Interval** to 1 hour and **Time Shift** to 56 minutes.*

**Note:** *If you set the **Interval** other than divisible without remainder (e.g. 25 minutes), the event will be executed when the remainder after dividing internal timestamp with the **Interval** shifted by **Time Shift** is zero.*

*Table 1: Event action types*

| Action | Description |
|---|---|
| *Synchronous* | |
| Run | Run a calculation periodically |
| Run – Log | Run a calculation and then store a message to file periodically. |
| Log | Store a message to file periodically. |
| Send | Send a message to serial line periodically. |
| Send – Receive | Send a message periodically and then receive the response asynchronously. The received values will not time out. |
| *Asynchronous* | |
| Receive | Receive a message from serial line if the received data satisfies criteria. |
| Receive – Send | Send a message to serial line on successful message reception |
| Receive – Run | Run a calculation on successful message reception. |

The actions determine the behavior of the data logger. Few fields are available only in few action types.
To create a new event, click the **Add new** button above the event list. After the creation the new event comes without name. Fill in a new name and then click the **Rename** button. The name is not changed until the **Rename** button is not clicked. If the name is changed and the event was used somewhere else, the program offers change the name also in these locations.

*Table 2: Event fields*

| Field | Description |
|---|---|
| Time interval | The interval of execution of synchronous event. |
| Time shift | The shift of time of execution inside the time interval. |
| Calculation | The calculation name which is executed on the event. |
| File name message | The name of the message containing the name of the log file. This message may be constant or may be formatted using time or other variables.<br>If not defined, the data logger use default file naming with syntax: DDMMYY.LOG<br>(day-month-year) |
| Log message | The name of the message which contains the format of record, which is added to the end of the file. |
| Serial line | The serial port name, to which the messages are sent and/or the messages are received from. |
| Send message | The name of the message which is sent to the serial line. |
| Receive message | The name of the message from which the variables are parsed if suitable data is received on serial line. |
| Validity | The time of the validity of parsed variables after receiving the message. If the next message not arrives in the defined time, all variables which were parsed by message become invalid.<br>If defined as zero, then the validity will not expire (infinite validity). |

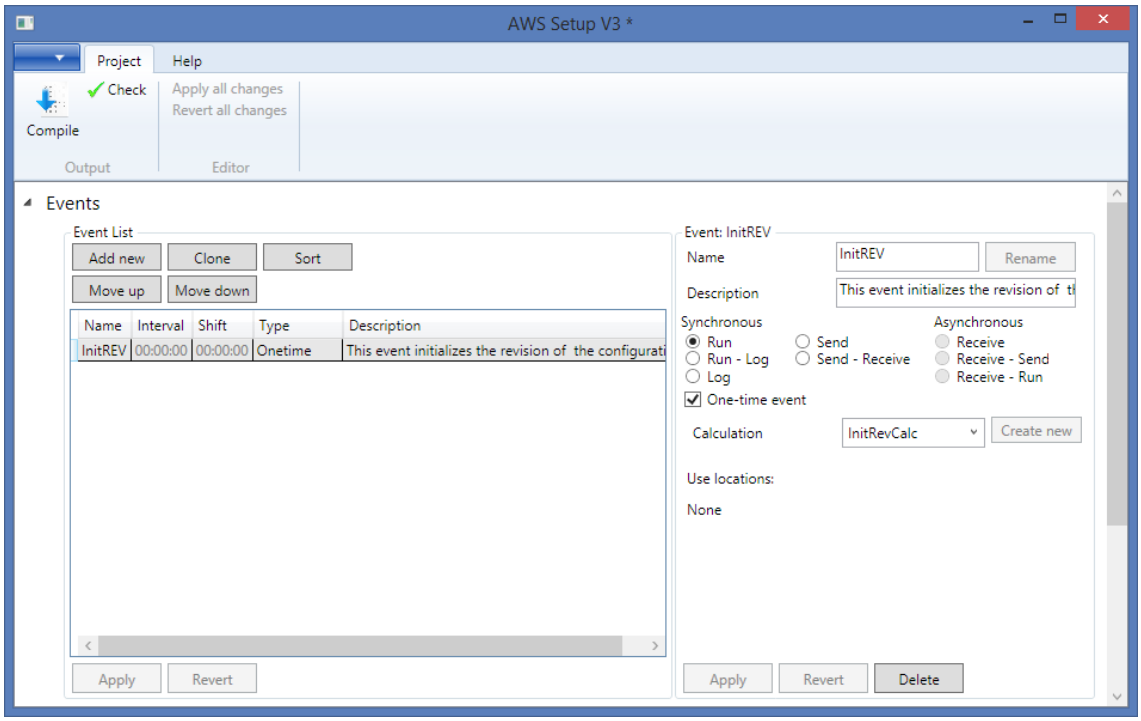



*Figure 6: New event*

# 5. Variables

Variables are used to store a value in Data Loggers' memory. This memory may be volatile (forgotten when the power is turned off) or **non-volatile** (remembered after power cycle).
Variables can have the following data **types**:

*Table 3: Variable types*

| Data type | Description |
|---|---|
| Float | 32-bit floating point number to store decimal values. |
| Int | 16-bit signed integer number <−32 768; +32 767> |
| Long | 32-bit signed integer number <−2 147 483 648; +2 147 483 647> |
| Char | 8-bit signed integer number <-128; 127> |
| String | Array of characters. **Length** must be specified for this type. Maximum length is 1024 bytes. |

Variable names have limitations in length. Maximum variable name is 8 characters. It is recommended to name variables with prefix "a" for actual measuring data. (E.g. actual temperature: aTemp)

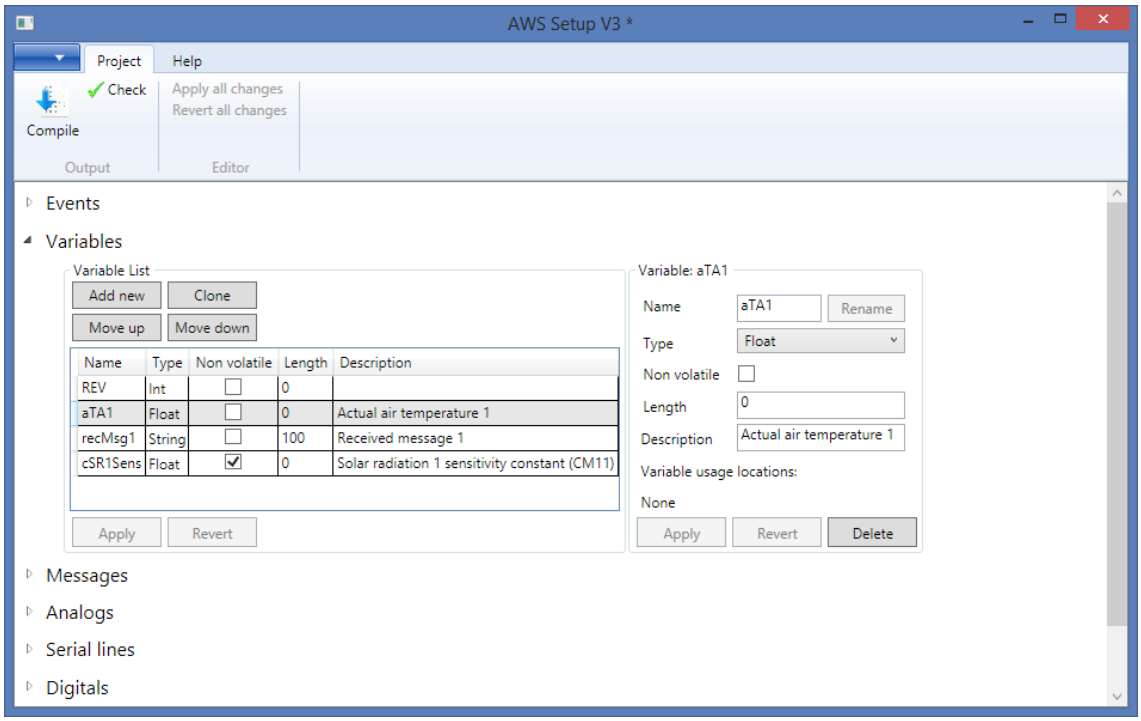



*Figure 7: Variables*

In the data logger, there are variables which are defined automatically and have special meaning.

*Table 4: Bult-in system variables of AMS 111 II*

| Built-in AMS 111 II variable name | Range | Description |
|---|---|---|
| **A*0*SLOPE** | 0-7 | Single ended analog channel conversion factor (ADC to voltage). The values are adjusted after production. Do not change these values! |
| **AnB*1*R100** | 1-2 | Current feedback resistor values in ohms. The values are adjusted after production. Do not change these values! |
| **cmdTOut** | | Command mode timeout. Timeout = value/4 seconds |
| **ETHDEFGTW** | | IP address of gateway. May be overridden by PPPISDEFGTW. |
| **ETHMAC** | | MAC address of the data logger |
| **ETHMYIP** | | IP address of the data logger for Ethernet interface. |

| Built-in AMS 111 II variable name | Range | Description |
|---|---|---|
| **ETHNETMASK** | | Subnet mask for the Ethernet interface. |
| **ETHPRIDNS** | | IP address of primary DNS server |
| **ETHSECDNS** | | IP address of secondary DNS server |
| **FtpDirUsrPsw** | | FTP client directory path, user name and password (on server).<br>Syntax: *Directory*:*User*:*Password* |
| **FtpIpPort** | | FTP IP address and port.<br>Syntax: *IP*:*PORT* |
| **GpsSecMinLimit** | | Minimal time difference in seconds to trigger the time synchronization with GPS or SNTP |
| **GpsAddMinToUTC** | | Local time zone and UTC difference in minutes when synchronizing time. |
| **HtmlLanguage** | | The language of the web interface. |
| **logDrive** | | Disk letter of the log files. |
| **mdmI2Dial** | | AT command for dialing with number<br>Example: ATD12345678 |
| **mdmIInit** | | AT command for modem initialization |
| **MdmLineEmer** | | The line number of the backup line |
| **mdmTngAPN** | | Tango modem APN |
| **mdmTOut** | | Close connection after timeout.<br>Timeout = Value/10 seconds |
| **PPPCONNLOG** | | Disk name, where the log about PPP connection are stored. |
| **PPPCONNTOUT** | | Timeout for connecting by PPP. |
| **PPPDEFGTW** | | IP address of gateway for PPP. |
| **PPPDIALNUM** | | Dial number for PPP. E.g.: **`ATD*99***1#`** |
| **PPPISDEFGTW** | | Is the PPP the default protocol for networking? If zero, the Ethernet interface is used. |
| **PPPLINEPHYS** | | PPP serial line number (without COM). |
| **PPPMYIP** | | IP address of the data logger for PPP. |
| **PPPNETMASK** | | Subnet mask for PPP. |
| **PPPPRIDNS** | | IP address of the primary DNS server for PPP. |
| **PPPSECDNS** | | IP address of the secondary DNS server for PPP |
| **PPPSOCKTOUT** | | PPP socket timeout. |
| **procSpeed** | | Processor speed. Have effect on consumption. 1 - basic speed, 2 -double speed, 3 - triple speed |
| **PWRETHER** | | Enable Ethernet or PPP interface.<br>0 - disabled, 1 - Ethernet, 2 - PPP, 3 - Ethernet+PPP |
| **UdpMaxLength** | | Maximum UDP packet length in bytes. |
| **USBDiskOpen_M_E** | | Behavior of USB port. (-1 = **COM3**, 0 = disk **M:**, 1 = disk **E:** ) |
| **VIRTUALCOM***08* | 08-23 | Virtual COM port definitions. See chapter 10 for details. |
| **CAMFILEFROM** | | Path to source of camera files (where the camera uploads the created files) |
| **CAMFILENAME** | | The destination file name. Can use %T specifiers (see chapter 8.1) and %d for LOGID |
| **CAMMAXFILES** | | The maximum number of files in the destination folder M:\CAM |
| **PPP2MDMToutH** | | Time in hours before the modem tries to switch the SIM card |

*Table 5: Bult-in system variables of SAWS 111*

| Built-in SAWS 111 variable name | Range | Description |
|---|---|---|
| **A***0***SLOPE** | 0-3 | Single ended analog channel conversion factor (ADC to voltage). The values are adjusted after production. Do not change these values! |
| **AnB1R100** | | Current feedback resistor value in ohms. The value is adjusted after production. Do not change this value! |
| **BattCheckI** | | Battery check interval in seconds |

| Built-in SAWS 111 variable name | Range | Description |
|---|---|---|
| **BattFullV** | | Full battery threshold in Volts. |
| **cmdTOut** | | Command mode timeout. Timeout = value/4 seconds |
| **FtpDirUsrPsw** | | FTP client directory path, user name and password (on server). Syntax: Directory**:**User**:**Password |
| **FtpIpPort** | | FTP IP address and port. Syntax: IP:PORT |
| **GpsSecMinLimit** | | Minimal time difference in seconds to trigger the time synchronization with GPS or SNTP |
| **GpsAddMinToUTC** | | Local time zone and UTC difference in minutes when synchronizing time. |
| **logDrive** | | Disk letter of the log files. Use M: only |
| **MdmLineEmer** | | |
| **PPPCONNLOG** | | Disk name, where the log about PPP connection are stored. |
| **PPPCONNTOUT** | | Timeout for connecting by PPP. |
| **PPPDEFGTW** | | IP address of gateway for PPP. |
| **PPPDIALNUM** | | Dial number for PPP. E.g.: **`ATD*99***1#`** |
| **PPPISDEFGTW** | | Is the PPP the default protocol for networking? If zero, the Ethernet interface is used. |
| **PPPLINEPHYS** | | PPP serial line number (without COM). |
| **PPPMYIP** | | IP address of the data logger for PPP. |
| **PPPNETMASK** | | Subnet mask for PPP. |
| **PPPPRIDNS** | | IP address of the primary DNS server for PPP. |
| **PPPSECDNS** | | IP address of the secondary DNS server for PPP |
| **PPPSOCKTOUT** | | PPP socket timeout. |
| **procSpeed** | | Processor speed. 1 - basic speed >2 – Not supported |
| **PWRETHER** | | Enable PPP interface. 0 - disabled, 1 – Not supported, 2 - PPP, 3 – Not supported |
| **USBDiskOpen_M_E** | | Behavior of USB port. (-1 = **COM3**, 0 = disk **M:**) |
| **VIRTUALCOM***04* | 04-05 | Virtual COM port definitions. See chapter 10 for details. |
| **VIRTUALCOM07** | | Limited virtual line for NTP time synchronization |

*Table 6: Bult-in system variables of AMS 111 IV*

| Built-in AMS 111 II variable name | Range | Description |
|---|---|---|
| **A***0***SLOPE** | 0-7 | Single ended analog channel conversion factor (ADC to voltage). The values are adjusted after production. Do not change these values! |
| **AnB***1***R100** | 1-2 | Current feedback resistor values in ohms. The values are adjusted after production. Do not change these values! |
| **cmdTOut** | | Command mode timeout. Timeout = value/4 seconds |
| **ETHDEFGTW** | | IP address of gateway. May be overridden by PPPISDEFGTW. |
| **ETHMAC** | | MAC address of the data logger |
| **ETHMYIP** | | IP address of the data logger for Ethernet interface. |
| **ETHNETMASK** | | Subnet mask for the Ethernet interface. |
| **ETHPRIDNS** | | IP address of primary DNS server |
| **ETHSECDNS** | | IP address of secondary DNS server |
| **FtpDirUsrPsw** | | FTP client directory path, user name and password (on server). Syntax: *Directory***:***User***:***Password* |
| **FtpIpPort** | | FTP IP address and port. Syntax: *IP***:***PORT* |
| **GpsSecMinLimit** | | Minimal time difference in seconds to trigger the time synchronization with GPS or SNTP |
| **GpsAddMinToUTC** | | Local time zone and UTC difference in minutes when synchronizing time. |
| **HtmlLanguage** | | The language of the web interface. |

| Built-in AMS 111 II variable name | Range | Description |
|---|---|---|
| **logDrive** | | Disk letter of the log files. |
| **mdmI2Dial** | | AT command for dialing with number<br>Example: ATD12345678 |
| **mdmIInit** | | AT command for modem initialization |
| **MdmLineEmer** | | The line number of the backup line |
| **mdmTngAPN** | | Tango modem APN |
| **mdmTOut** | | Close connection after timeout.<br>Timeout = Value/10 seconds |
| **PPPCONNLOG** | | Disk name, where the log about PPP connection are stored. |
| **PPPCONNTOUT** | | Timeout for connecting by PPP. |
| **PPPDEFGTW** | | IP address of gateway for PPP. |
| **PPPDIALNUM** | | Dial number for PPP. E.g.: **`ATD*99***1#`** |
| **PPPISDEFGTW** | | Is the PPP the default protocol for networking? If zero, the Ethernet interface is used. |
| **PPPLINEPHYS** | | PPP serial line number (without COM). |
| **PPPMYIP** | | IP address of the data logger for PPP. |
| **PPPNETMASK** | | Subnet mask for PPP. |
| **PPPPRIDNS** | | IP address of the primary DNS server for PPP. |
| **PPPSECDNS** | | IP address of the secondary DNS server for PPP |
| **PPPSOCKTOUT** | | PPP socket timeout. |
| **procSpeed** | | Processor speed. Have effect on consumption. 1 - basic speed, 2 -double speed, 3 - triple speed |
| **PWRETHER** | | Enable Ethernet or PPP interface.<br>0 - disabled, 1 - Ethernet, 2 - PPP, 3 - Ethernet+PPP |
| **UdpMaxLength** | | Maximum UDP packet length in bytes. |
| **USBDiskOpen_M_E** | | Behavior of USB port. (-1 = **COM3**, 0 = disk **M:**, 1 = disk **E:** ) |
| **VIRTUALCOM***08* | 16-31 | Virtual COM port definitions. See chapter 10 for details. |
| **CAMFILEFROM** | | Path to source of camera files (where the camera uploads the created files) |
| **CAMFILENAME** | | The destination file name. Can use %T specifiers (see chapter 8.1) and %d for LOGID |
| **CAMMAXFILES** | | The maximum number of files in the destination folder M:\CAM |
| **PPP2MDMToutH** | | Time in hours before the modem tries to switch the SIM card |

Some of the system variables become adjustable only after creating it in the configuration. These variables are defined in the table below.

*Table 7: Optional system variables of AMS 111 II*

| Optional AMS 111 II variable name | Range | Description |
|---|---|---|
| **AdPeriod** | | Override Analog to digital converter period. If set, the period in configuration is ignored for all channels. |
| **Charge_O** | | If set, overrides default charging behavior which may be ridden in configuration. 1-turns charging on, 0-turns charging off. |
| **DispTout** | | Display timeout in seconds. |
| **FtpError** | | The error code of FTP transfer |
| **FtpFileR** | | The destination file name (fill it first) |
| **FtpFile** | | The local fine to transfer. Writing initiates transfer, |
| **LineSt***00* | 00-23 | Serial lines status. |
| **LogId** | | Data logger identification. May be used to identify data logger on buses. |
| **mdmline***0* | 0-23 | Disable modem control on serial line. |
| **pw12v** | | Digital outputs. See chapter 7.2 for details |
| **pw12v***0* | 0-7 | Digital outputs. See chapter 7.2 for details |
| **PwrLine***1* | 1-2 | Enable line converter power (for lines 1 or 2).<br>Value: 0 - disable, 1 - enable |
| **PwrSerL***0* | 0-3 | Serial extension card power for communication line. Line is identified by number in variable name.<br>Value: 0 - disable, 1 - enable |
| **PwrSerP***0* | 0-3 | Serial extension card power for external device powering. Line is identified by number in variable name.<br>Value: 0 - disable, 1 - enable |
| **rrrrAR***00* | 00-23 | Measured resistance on differential analog input identified by channel number. |
| **rrrrAU***00* | 00-31 | Measured voltage on all analog inputs. |
| **UdpZaraz** | | If set, UDP packet is not send until the contents of this variable appears in the output. |

*Table 8: Optional system variables of SAWS 111*

| Optional SAWS 111 variable name | Range | Description |
|---|---|---|
| **AdPeriod** | | Override Analog to digital converter period. If set, the period in configuration is ignored for all channels. |
| **Charge_O** | | If set, overrides default charging behavior which may be ridden in configuration. 1-turns charging on, 0-turns charging off. 2-Automatic charging according to BattCheckI and BattFullV |
| **DispTout** | | Display timeout in seconds. |
| **FtpError** | | The error code of FTP transfer |
| **FtpFileR** | | The destination file name (fill it first) |
| **FtpFile** | | The local fine to transfer. Writing initiates transfer, |
| **LogId** | | Data logger identification. May be used to identify data logger on buses. |
| **pw12v** | | Digital outputs. See chapter 7.2 for details |
| **pw12v***0* | 0-7 | Digital outputs. See chapter 7.2 for details |
| **PwrLine***1* | 1-2 | Enable line converter power (for lines 1 or 2).<br>Value: 0 - disable, 1 - enable |
| **rrrrAR***00* | 00-07 | Measured resistance on differential analog input identified by channel number. |
| **rrrrAU***00* | 00-11 | Measured voltage on all analog inputs. |

*Table 9: Optional system variables of AMS 111 IV*

| Optional AMS 111 II variable name | Range | Description |
|---|---|---|
| **DispTout** | | Display timeout in seconds. |
| **FtpError** | | The error code of FTP transfer |
| **FtpFileR** | | The destination file name (fill it first) |
| **FtpFile** | | The local fine to transfer. Writing initiates transfer, |
| **LineSt***00* | 00-31 | Serial lines status. |
| **LogId** | | Data logger identification. May be used to identify data logger on buses. |
| **mdmline***0* | 0-31 | Disable modem control on serial line. |
| **pwrline***0* | 0-7 | Power outputs. See chapter 7.2 for details |
| **dout***0* | 0-4 | Digital outputs. See chapter 7.2 for details |
| **PwrLine***1* | 1,8-9, A-B | Enable line converter power (for lines 1 or 8-11).<br>Value: 0 - disable, 1 - enable |
| **rrrrAR***00* | 00-23 | Measured resistance on differential analog input identified by channel number. |
| **rrrrAU***00* | 00-31 | Measured voltage on all analog inputs. |
| **UdpZaraz** | | If set, UDP packet is not send until the contents of this variable appears in the output. |
| **pppOut***0* | 0-7 | Power line for modem, if used other than line 1. |
| **pubPPPip** | | Public PPP IP address |
| **priPPPip** | | Private PPP IP address |

# 6. Analog channels

Analog channel voltages are periodically sampled into **floating** point variables.
Measured value can be modified before saving it to the variable. There are some predefined calculations built in into the Data Loggers' firmware.

*Table 10: Analog calculations*

| Calculation | Description |
|---|---|
| None | Voltage is unchanged |
| Temperature | Temperature in °C is calculated from the resistance of Pt100 according to ITS-90 for probes with alpha = |
| Scaling | Third order polynomial calculation with the measured voltage |
| Logic | Stores 0.0 to float variable if the voltage is below the defined threshold and 1.0 if above |

**Attention:** *Measured value is in **Volt [V]** units for **single ended** channels and **Millivolts [mV]** for the **differential** channels.*

To define an analog measurement in the AWS Setup, expand the **Analogs** group and click the **Add new** button. Click **Apply** below the list to commit all changes in the list. Select the new channel in the list and the editor appears on the right side. Output **Variable** and sampling **Period** are required. The variable name may be chosen from the drop-down list or entered as a new name. If the variable exists, you can edit it on the right side. If does not exist, you can create a new by clicking the **Create new** button. **Period** is the sample rate for this channel.

**Minimum** and **maximum** defines the valid range of the result. If the result of measurement after calculations is outside this range, the value in variable becomes invalid. This invalidity may be signalized in messages (e.g. variable value is printed as //// instead of numbers).

**Channel** has a prefix depending on the **Type** of the analog. For **Single ended**, the channels have prefix **A**. **Differential** internal channels have prefix **UR** and **Differential** external channels have prefix **ADF**.

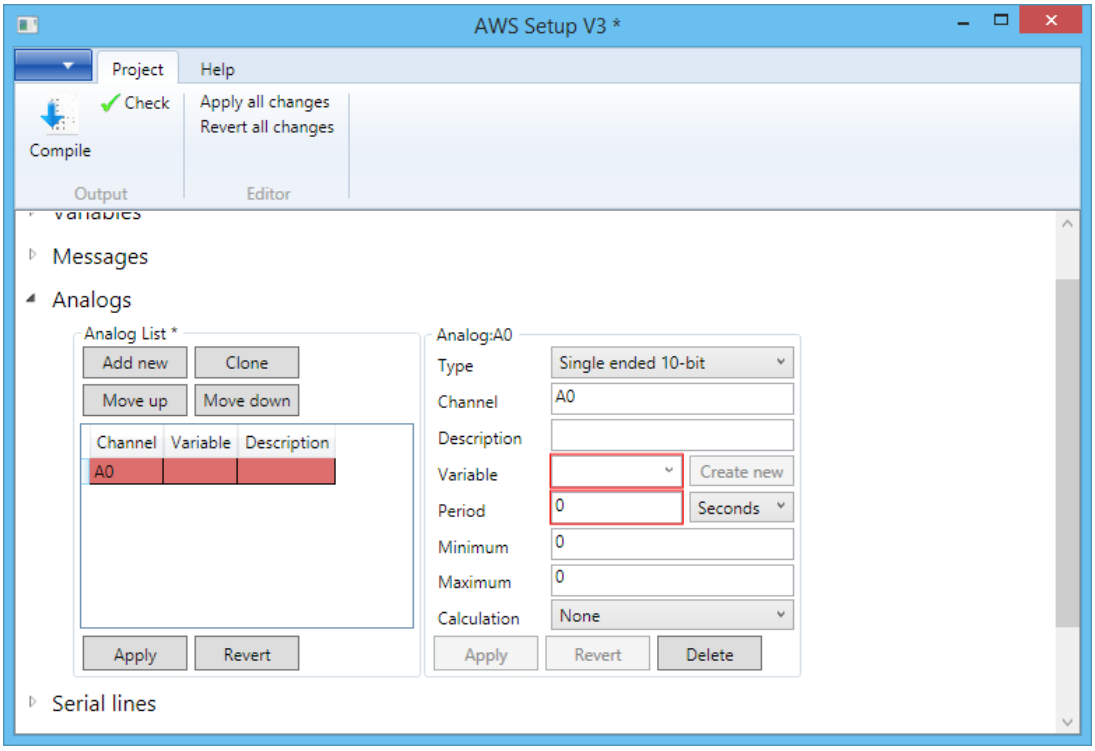



*Figure 8: New analog channel*

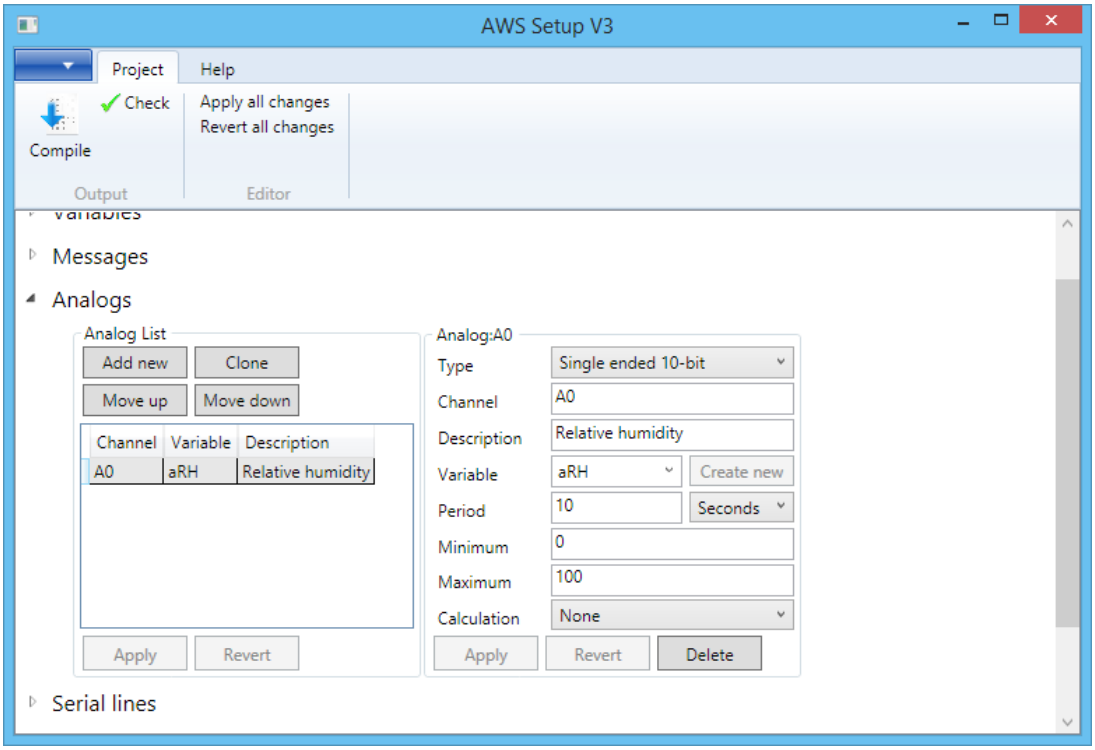



*Figure 9: Created analog channel*

## 6.1 10/12-bit single ended channels

By default these inputs have input range 0 – 5 V. This range may be changed to 0 – 2.5 V by jumper inside data logger.
Channels A5–A7 has special meaning. They are internally connected to measure power voltages.

*Table 11: Special single ended analogs*

| Channel | Description |
|---|---|
| APOWER | DC voltage for battery charging (not available on SAWS111) |
| ABATT | Input voltage for the data logger (from battery) |
| ALIBAT | Lithium battery voltage (time and RAM backup, not available on SAWS111) |
| ACURR | Current consumption of device (only SAWS111) |

## 6.2 24-bit differential channels

These channels are connected using two wires and the voltage is measured between the positive (e.g. ADFIN1P) and negative (e.g. ADFIN1N) pins. This analog converter can measure the resistance of Pt100 temperature sensors by 4-wire connection. It has 2 current sources to excite the Pt100 resistors and the differential voltage is measured on the Pt100 sensors. The value of the current is used in resistance calculation inside the **Temperature** calculation. The current is measured by measuring the voltage on two, calibrated precision resistors for each channel group. To measure Pt100, use **Gain** 16 with range 156 mV to ensure that value will remain in this range for temperatures in required range.

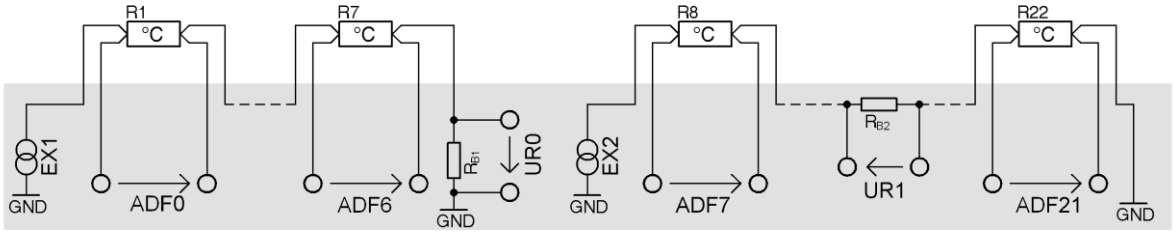



*Figure 10: Differential channels internal structure*

*Table 12: Special differential analogs*

| Channel | Description |
|---|---|
| UR0 | Current feedback for channels ADF0 – ADF6 |
| UR1 | Current feedback for channels ADF7 – ADF21 |

**Attention:** *In order to successfully measure the resistance of the Pt100 thermometers, it is **required to configure** the Data Logger to measure also the voltage on channels **UR0** and/or **UR1**.*

**Note:** *If the **Temperature** calculation is selected on the measured channels, then the Data Loggers' firmware handles these voltages internally. These values are only needed to measure and may be discarded, e.g. by setting the same dummy variable for multiple things.*

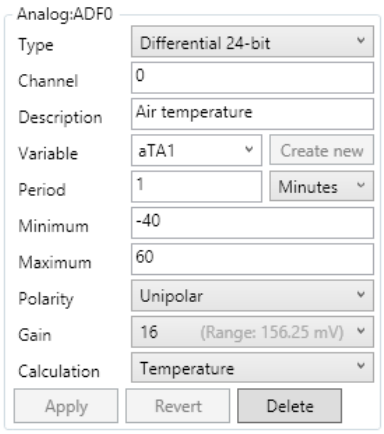



*Figure 11: Differential analog channel*

# 7. Digital channels

Values of digital inputs are definable in the configuration, but the digital outputs are only accessible using variables with special name.

## 7.1 Digital inputs

Data logger can measure binary state of the digital inputs. Some operations require exactly one channel selected; some may use multiple channels. **Channels** are shown as checkboxes from D1 to D11. The results from the digital inputs must be stored in **long** integer variable. The following table summarizes the available functions.

*Table 13: Digital operations*

| Operation | Description |
|---|---|
| None | 0 or 1 if one channel is selected<br>-OR-<br>Binary value of the bits with the lowest bit as LSB if multiple channels are selected. |
| Counter | Number of pulses during the period. Not available for channel D11 |
| Frequency | Number of pulses divided by the period. Result is in Hertz [Hz] |
| Gray code | Vaisala WAV151 wind direction sensor |
| Wind speed | Vaisala WAA151 wind speed |
| Thies Wind speed | Thies wind speed |
| 2030 Wind speed | All Weather model 2030 anemometer |
| Serial synchronous | Thies first class. Available only for channel D11 |

Maximum and moving average of frequency during defined period with sample rate of 250 ms may be calculated by defining two digitals on the same channel and checking "**Maximum of 250 ms samples**" on one of them. The checked channel will measure the maximum sampled frequency during the set **period**. The not checked digital (on the same channel) will measure the average frequency during the **period** in 250 ms samples.

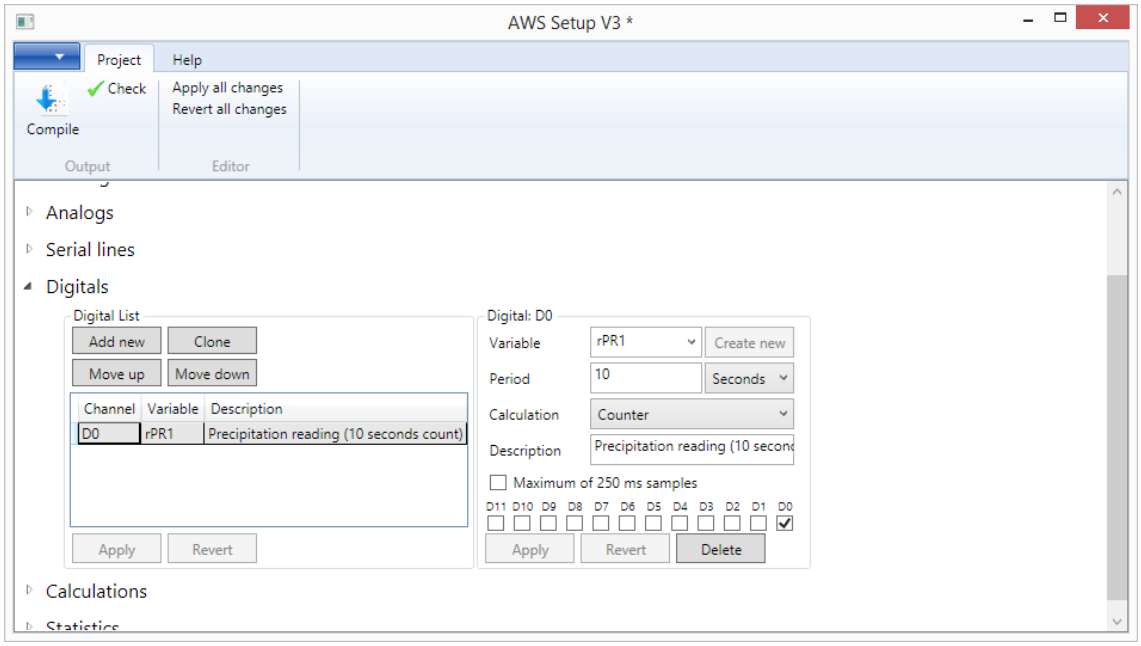



*Figure 12: Digital channels*

The sample rate is defined under the **Period** item.

## 7.2 Digital outputs

Digital outputs are accessible using variables with special name and type **long**. If these variables are not defined in the configuration, these outputs are not accessible.

*Table 14: Digital output variables on AMS111 II and SAWS 111*

| Variable name | Description |
|---|---|
| pwr12v | Whole register of digital output bits, which are described below. |
| pwr12v0 | Connected to PWR_OUTC0. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwr12v1 | Connected to PWR_OUTC1. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwr12v2 | Connected to PWR_OUTC2. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwr12v3 | Connected to PWR_OUTC3. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwr12v4 | Connected to DOUT0P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |
| pwr12v5 | Connected to DOUT1P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |
| pwr12v6 | Connected to DOUT2P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |
| pwr12v7 | Connected to DOUT3P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |

*Table 15: Digital output variables on AMS111 IV*

| Variable name | Description |
|---|---|
| pwrout0 | Connected to PWR_OUTC0. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout1 | Connected to PWR_OUTC1. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout2 | Connected to PWR_OUTC2. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout3 | Connected to PWR_OUTC3. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout4 | Connected to PWR_OUTC4. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout5 | Connected to PWR_OUTC5. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout6 | Connected to PWR_OUTC6. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| pwrout7 | Connected to PWR_OUTC7. If the value is 0, the output is connected to Ground; otherwise the output is connected to 12V. |
| dout0 | Connected to DOUT0P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |
| dout1 | Connected to DOUT1P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |
| dout2 | Connected to DOUT2P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |
| dout3 | Connected to DOUT3P. If the value is 0, the output is in high impedance; otherwise the output is connected to ground. |

For backward compatibility the pw12v*n* variables are also available in AMS 111 IV, but if pwrout*n* variable is defined, they have higher priority (decision is made per pin).

# 8. Messages

Messages may be defined for formatting and parsing purpose. For parsing purpose, the **Input message** checkbox must be checked and at least **End String** must be defined.
Message size is limited to 64 bytes (80 bytes in configuration version 2); therefore, longer messages must be concatenated in a **Calculation** (Operation **+** for string output variable). See chapter 13.1 for details.

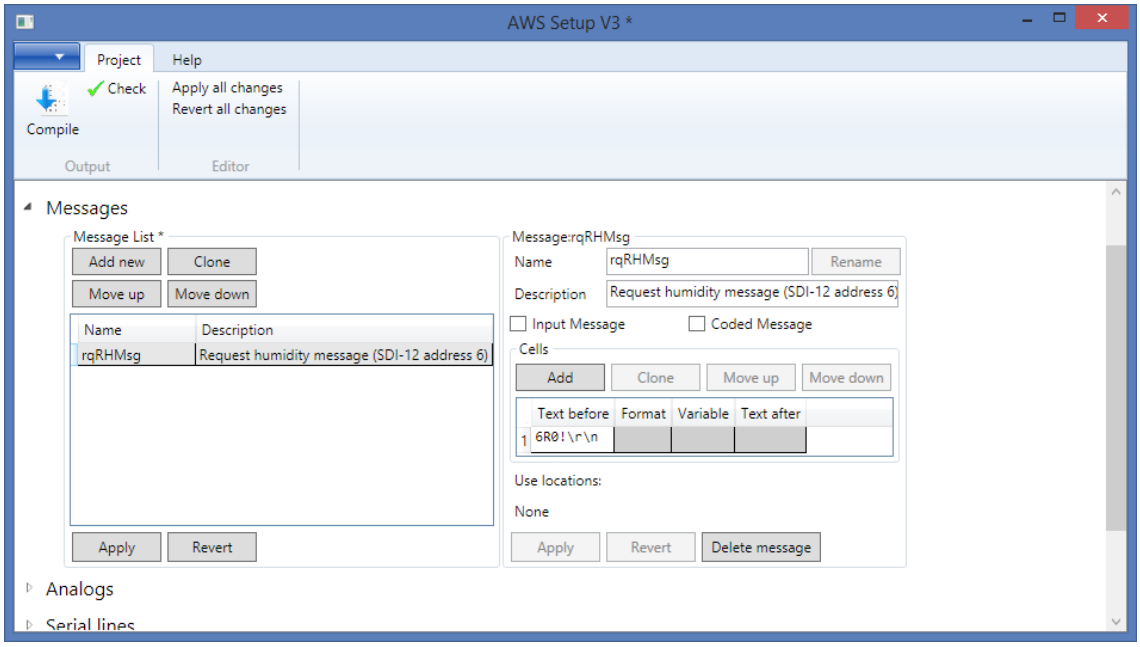



*Figure 13: Output message*

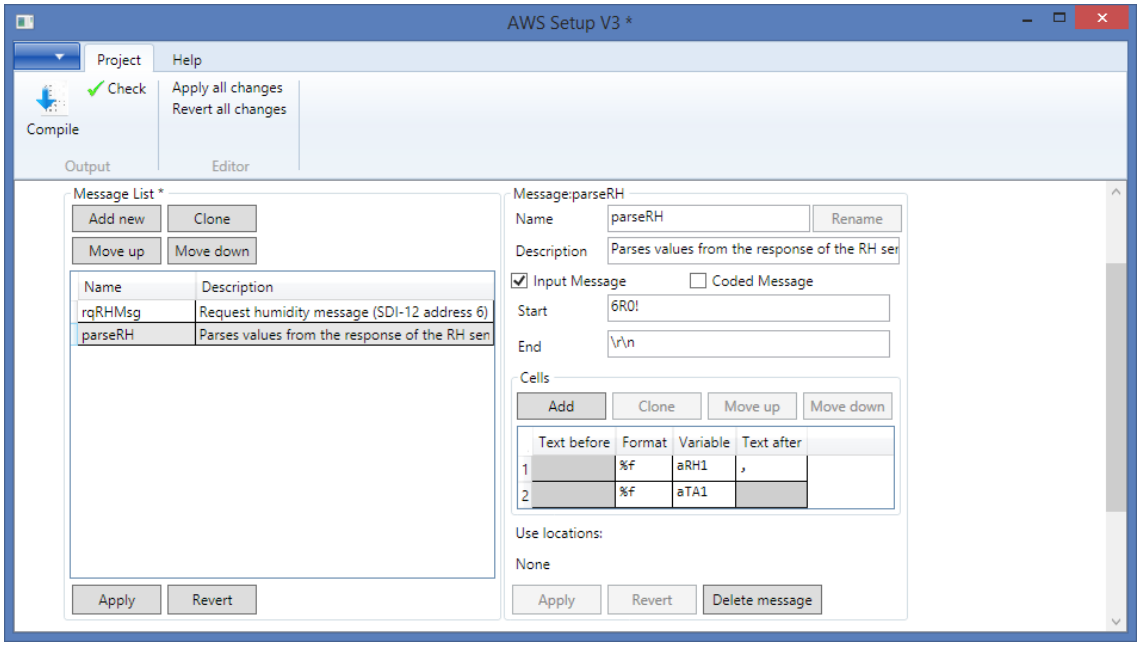



*Figure 14: Input message*

The **Coded Message** option is for advanced users, when the contents of the entered strings may contain special commands for binary concatenation. If the **Coded Message** option is not checked, the data logger will interpret the message as it is written. If the **Coded Message** is checked, then the user is responsible to handling binary options, with the ability to use extended binary features.

Message is formatted from / parsed to variables, which are defined in a cell. The cells are merged into one format string, but the table notation is clearer and allows changing the order of each fields. The contents of **Format** and **Variable** cells belong together. The "**Text before**" and "**Text after**" cells are written before or after the format specifier. If the variable entered into the Variable field does not exist, it is possible to create it after clicking the cell and then on the right by clicking the **Create new** button. Enter the format specifiers **only** to the **Format** cell, because otherwise the AWS Setup will consider these format strings as text and not by special meaning.

There are few special **Format** specifiers, which does not require input/output **variable**. These are for example checksum, CRC or time formats. Full list of specifiers is in the following table.

*Table 16: Message format styles*

| Format | Description | Details |
|---|---|---|
| *Standard C-style format specifiers* | | |
| %f | Floating point number | chapter 8.2 |
| %d | Integer number | |
| %u | Unsigned number | |
| %e, %E | Scientific notation of floating point numbers | |
| *Custom format specifiers* | | |
| %s | String | chapter 8.6 |
| %j | Coded float, sign (-) | chapter 8.3 |
| %g | Coded float, sign (0/1) | chapter 8.4 |
| %h | Coded float, sign (M) | chapter 8.5 |
| %T | Date/time | chapter 8.1 |
| %B | Binary format | chapter 8.7 |
| %N | Variable validity (0/1) | chapter 8.8 |
| %L | Slash mode | chapter 8.9 |
| %c, %C | Message CRC<br>-OR-<br>Print character | chapter 8.10 |
| %x, %X | Message checksum | chapter 8.11 |
| %Z | Message XOR | chapter 8.12 |

Detailed description of the format specifiers are in next chapters. Syntax in these has the following meaning: Parts of the specifiers in square brackets [] are optional, parts in **bold** are constants (and have to be written as is) and parts in *italics* are varying.

## 8.1 Date and time format

This format specifier does not require variable, unless %**I**…T is specified. If variable is provided, the date and time is processed from this variable and not from the system time.
This variable may be captured from system variable TIME which is of type **Long**.

Syntax:

**%[I]***format***T**

*Table 17: Date and time field description*

| Field | Meaning |
|---|---|
| I | First character is capital i. If present, indicates, that the timestamp is from the provided variable, otherwise no variable is needed and system time is used. |
| format | String representing how to display date/time. See the table below |

Format string may contain these special characters, which are replaced:

*Table 18: Date and time format characters*

| Character | Meaning |
|---|---|
| Y | Full year (e.g. 2014) |
| y | Short year (e.g. 14) |
| m | Month |
| D | Day |
| H | Hour |
| M | Minute |
| S | Second |

Other characters **except** the below listed are permitted in format string:
I, f, d, u, e, E, s, g, h, j, T, B, N, L, c, C, x, X, z, Z

## 8.2 Floating point format

Floating point format is identical to standard C-style

Syntax:

**%**[width][.precision]**f**

*Table 19: Floating point format fields*

| Field | Meaning for output | Meaning for input |
|---|---|---|
| width | Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.<br>If the value is not defined, width of the resulting string will vary depending on the variable value. | Maximum number of characters to be read in the current reading operation.<br>This may be used for parsing from fixed length messages without separator. |
| .precision | Number of decimal digits to be displayed. If not defined 6 decimal places will be shown. | Not used |

## 8.3 Coded floating point format (j)

The value to be displayed is firstly calculated using the following equation and then it is displayed as an integer number.

$$value = variable \cdot 10^{powerOfTen}$$

The sign of the value is **not displayed for positive values** and **'-' for negative values**

Syntax:

**%***[width][.powerOfTen]***j**

*Table 20: Coded floating point format j fields*

| Field | Meaning for output | Meaning for input |
|---|---|---|
| width | Minimum number of characters to be printed. | Maximum number of characters to be read in the current reading operation. |
| .powerOfTen | This number is used in an equation before printing the number. | |

## 8.4 Coded floating point format (g)

The value to be displayed is firstly calculated using the following equation and then it is displayed as an integer number.

$$value = variable \cdot 10^{powerOfTen}$$

This format type differs from the previous by the way of displaying the number sign. The sign of the value is **'0' for positive values** (displayed always) and **'1' for negative values**.

**Example:** *If the format is %6.2g and the value in the variable is -12.345, then the result is: "101234"*

| Syntax: |
|---|
| **%***[width][.powerOfTen]***g** |

## 8.5 Coded floating point format (h)

The value to be displayed is firstly calculated using the following equation and then it is displayed as an integer number.

$$value = variable \cdot 10^{powerOfTen}$$

This format type differs from the previous by the way of displaying the number sign. The sign of the value is **not displayed for positive values** and **'M' for negative values**.

**Example:** *If the format is %6.2h and the value in variable is -12.345, then the result is: "M01234"*

| Syntax: |
|---|
| **%***[width][.powerOfTen]***h** |

## 8.6 String format

If used in output message, the entire string is printed out.
In input messages for parsing it reads string to variable until first whitespace character (' ', '\n', '\r', '\t', '\0'), comma (',') or any ASCII character with value less than 33 if the width is not defined

| Syntax: |
|---|
| **%***[width]***s** |

*Table 21: String format fields*

| Field | Meaning for output | Meaning for input |
|---|---|---|
| width | Minimum number of characters to be printed. | Maximum number of characters to be read. |

## 8.7 Binary format

Depending on the type of the output variable of the message, the format specifier works differently.

For output messages:

Syntax:

For standard use:

Non-string, numeric variables:

**%***BitLength***.***[PowerOfTen]***.***[Offset]***.***[PositionBegin]***[.***Equation***][***DateFormat***]B**

String variables:

**%***BitLength***B**

For complex format, which is defined in a file:

**%--B(***[FileName]***)**

*Table 22: Binary format fields for output messages*

| Field | Meaning | Example |
|---|---|---|
| BitLength | Number of bits of the coded variable.<br>If zero for string values, then the whole string is added to output. | **%8B**<br>65 => 0100 0001 |
| PowerOfTen | The value multiplied by $10^{powerOfTen}$. Negative numbers are permitted. For details see explanation below table. | |
| Offset | Value added to (subtracted from) the previously adjusted variable value. | |
| PositionBegin | Bit index from the end of the message, where the result writing begins. By defining this parameter, it is allowed to overwrite previously added results.<br>If not defined, the result is added to the end. | |
| Equation | Equations to define operations before coding. For details see explanation below this table. | |
| DateFormat | If the variable supplied contains time stamp, then a character from Table 18 may be entered for choosing which date component has to be processed. | **%8….yB**<br>converts time to 8-bit short year |
| FileName | If the File name is specified, then the format is read from the specified file in the string. If it is not specified, the file name is taken from the string variable which belongs to this format specifier.<br>The variable has to be defined even if the file name is specified.<br>The file must contain format specifiers as specified in chapter 0 | **%--B(FILE.BFR)**<br>Reads formats from file G:\FILE.BFR<br><br>**%--B()**<br>Variable: fileName<br>Reads formats from file G:*fileName* |

Numeric values may be altered before writing to output to be able to specify various binary protocol formats. For simpler use, it is possible to use just the following computation:

$$BinaryValue = (\text{int})(Value \cdot 10^{PowerOfTen} + Offset)$$

If an equation is defined, the equation is evaluated first and then the above computation. The equation has the following syntax:

$$\left[\left[\left[\begin{matrix}+\\-\\*\\/\end{matrix} number1\right]\begin{matrix}+\\-\\*\\/\end{matrix} number2 \ldots\right]\begin{matrix}+\\-\\*\\/\end{matrix} numberN\right]$$

**Example:** *If the specified format is %16.2.3..*10.4-50.9/2.4+4.7B, then the value before coding is calculated this way: ((((variable * 10.4) - 50.9) / 2.4) + 4.7) * 10^2 + 3*

For input messages, parsing purpose:

| Syntax:<br>**%***ByteLength***B** |
|---|

*Table 23: Binary format fields for input messages*

| Field | Meaning | Example |
|---|---|---|
| ByteLength | Number of **bytes** to be parsed. Byte endianness is determined by the sign of this number. For negative values LSB is first. | **%2B**<br>0000 0010 0100 0001 => 577<br><br>**%-2B**<br>0000 0010 0100 0001 => 16642 |

### 8.7.1 BFR file format

This is the format which is used in the file with binary option %--B(file). The whole content of the file is used to create a resulting string using variable formatting and decision tree. Everything between quotation marks is interpreted in the same way as in the messages including format specifiers in Table 16. Outside the quotation marks hexadecimal numbers are written to output after converting to binary value.

**Example:** *The expression **00"abc"1aff** is equal to **"\x00abc\x1a\xff"***

Whitespaces outside quotation marks are ignored. If the format string in quotes requires a variable, it has to be provided immediately after the closing quotation mark in brackets.

**Example:** *The expression **"abc%f"(aTA1)** writes to the output the constant **abc,** then formats the variable **aTA1** with format **%f**. Expression **"abc%fdef"(aTA1)** is not allowed, it has to be defined this way: **"abc%f"(aTA1)"def"**.*

**Note:** *It is possible to define variables before the format specifier in the string, but it is not recommended because of clarity. E.g. **(aTA1, aTA2)"abc%fdef%fgjh"** is valid expression.*

Multiple variables may be defined in brackets, variables are separated by comma. If the character outside quotes is not a hexadecimal number, then it is a special command.

*Table 24: BFR file commands*

| Character | Meaning |
|---|---|
| ? | Conditional message |
| X | Inserts into message a bit trimmed to 1-7 bits length |

Conditional message building syntax:

**?***variable***{**
*caseValue1* **:** *message 1***,**
*caseValue2* **:** *message 2***,**
[**D:** *default message***,**]
**}**

Depending on the variable value, different parts of message are written to output. The *case value* should be a decimal number or a string constant (without format) in quotation marks. If no value is found, the optional default value is written to output. Default value is defined by **D** character.

**Example:** *The following example writes to output different messages for different values in user defined variable SECTION. The messages for value 9 are concatenated. The message for value 10 contains another conditional message.*

```
?SECTION{
  9: 00"%5...4.HB"(syTime)"%6....MB"(syTime),
  9: X500X5"%11.1.100B"(aTA1),
  10: ?rCMD{
     "WIND": "wind=%.3f"(aWD),
     D: "Unknown command",
       },
  11: "Section 11",
  D: "Unknown section"
}
```

Bit length syntax:

**X1***hexadecimalNumber***X1**
…
**X7***hexadecimalNumber***X7**

This syntax converts the length of the defined *hexadecimal number* between **X*n*** tags, where *n* denotes the new length of the field. It may be used for filling protocol with non-whole byte constants.

## 8.8 Variable validity format

This format writes to output **'0'** if the variable is **invalid** and **'1'** if it is **valid**. Validity of variable may be defined in its source. E.g.: If the value is read from the analog channel and the measured value is not in the defined range (between minimum and maximum), then the variable is invalid. If the variable is calculated from an invalid variable, it becomes also invalid.

Syntax:

**%N**

## 8.9 Slash mode switch

This is fact not a format specifier, but defines the behavior for the variables that are in the message after this switch. This switch doesn't have to be defined in the variable section of the message, because it does not require any variable. This mode is used only for output messages.
If the slash mode is enabled, then the variable is replaced with slashes '/' when the variable is invalid.

Syntax:

**%***Enable***L**

If the Enable is '0', then the slash mode is disabled after this switch.
If the Enable is '1', then the slash mode is enabled after this switch.

## 8.10 CRC checksum format

For output message the result is written as a 16-bit hexadecimal or binary number.
For input message the string must contain the same value on the same place, as would be calculated for the output message to be considered as valid message. If the value does not match, the message is not parsed.

| Syntax: | |
|---|---|
| **%***n[.ignoredCount]*[**b**]**c** | for lowercase hexadecimal values |
| **%***n[.ignoredCount]*[**b**]**C** | for uppercase hexadecimal values |

*Table 25: CRC checksum format fields*

| Field | Meaning |
|---|---|
| n | If not defined, works like standard %c for character formatting.<br>If defined as 16, then the result is CRC-16-ANSI/IBM (polynomial 0xA001) of previous message. If the defined as 8, then CRC-8-CCITT (polynomial 0xE0) is calculated.<br>Otherwise the polynomial and custom options may be defined.<br>If the polynomial is less than 0xFF, then the result is 8-bit CRC. |
| .ignoredCount | Number of bytes which are ignored at the end of the previous message. If not defined, the checksum is calculated from the whole message before this format specifier.<br>Useful if multiple checksums are used in a message. |
| b | Binary output option. If the lower **b** is not defined, the result is generated as hexadecimal number. If defined, the result is binary. The case of the format character (**C**/**c**) is used to determine the endianness in binary mode. Upper case **C** is big endian, lower case **c** is little endian. |

## 8.11 Message checksum format

This format calculates the checksum of the string. (Only the checksum of the part before this format specifier) If this specifier is used in an input message, the message is verified. (I.e. invalid string does not satisfies the checksum criteria, and therefore it is not read into the variables)
This format specifier **does not require a variable**.

| Syntax: | |
|---|---|
| **%***n[.ignoredCount]*[**b**]**x** | for lowercase hexadecimal values |
| **%***n[.ignoredCount]*[**b**]**X** | for uppercase hexadecimal values |

For output message the result is written as a hexadecimal or binary number.
For input message the string must contain the same value on the same place, as would be calculated for the output message to be considered as valid message. If the value does not match, the message is not parsed.
Checksum is calculated using this formula:

$$\sum(previous\ characters) \bmod 2^n$$

*Table 26: Checksum format fields*

| Field | Meaning |
|---|---|
| n | Checksum length in bits. |

| Field | Meaning |
|---|---|
| .ignoredCount | Number of bytes which are ignored at the end of the previous message. If not defined, the checksum is calculated from the whole message before this format specifier.<br>Useful if multiple checksums are used in a message. |
| b | Binary output option. If the lower **b** is not defined, the result is generated as hexadecimal number. If defined, the result is binary. The case of the format character (**X**/**x**) is used to determine the endianness in binary mode. Upper case **X** is big endian, lower case **x** is little endian. |

## 8.12 Message XOR format

This format calculates the exclusive OR checksum from the text before the format specifier using the following algorithm:

```
Checksum = Initial value
For each character:
   Checksum = Checksum XOR character
```

For output message the result is written as a hexadecimal or binary number.
For input message the string must contain the same value on the same place, as would be calculated for the output message to be considered as valid message. If the value does not match, the message is not parsed. This format **does not require a variable**.

| Syntax: | |
|---|---|
| **%***n[.ignoredCount]*[**b**]**z** | for lowercase hexadecimal values |
| **%***n[.ignoredCount]*[**b**]**Z** | for uppercase hexadecimal values |

*Table 27: XOR format fields*

| Field | Meaning |
|---|---|
| n | Initial value of the checksum. |
| .ignoredCount | Number of bytes which are ignored at the end of the previous message. If not defined, the checksum is calculated from the whole message before this format specifier.<br>Useful if multiple checksums are used in a message. |
| b | Binary output option. If the lower **b** is not defined, the result is generated as hexadecimal number. If defined, the result is binary. The case of the format character (**Z**/**z**) is used to determine the endianness in binary mode. Upper case **Z** is big endian, lower case **z** is little endian.. |

# 9. Calculations

Calculations are mathematical expressions used to change the values in variables, filling the variables with numeric constants, running events conditionally or concatenate strings. String constants are not supported. To add a string constant, create a message with it and include the message in the expression.
The calculation stores the result of the **Expression** into a fixed variable defined by **Output variable**.
It is possible to add an operation, variable, message or event to the end of the expression by selecting from list to ensure that the name is not misspelled.

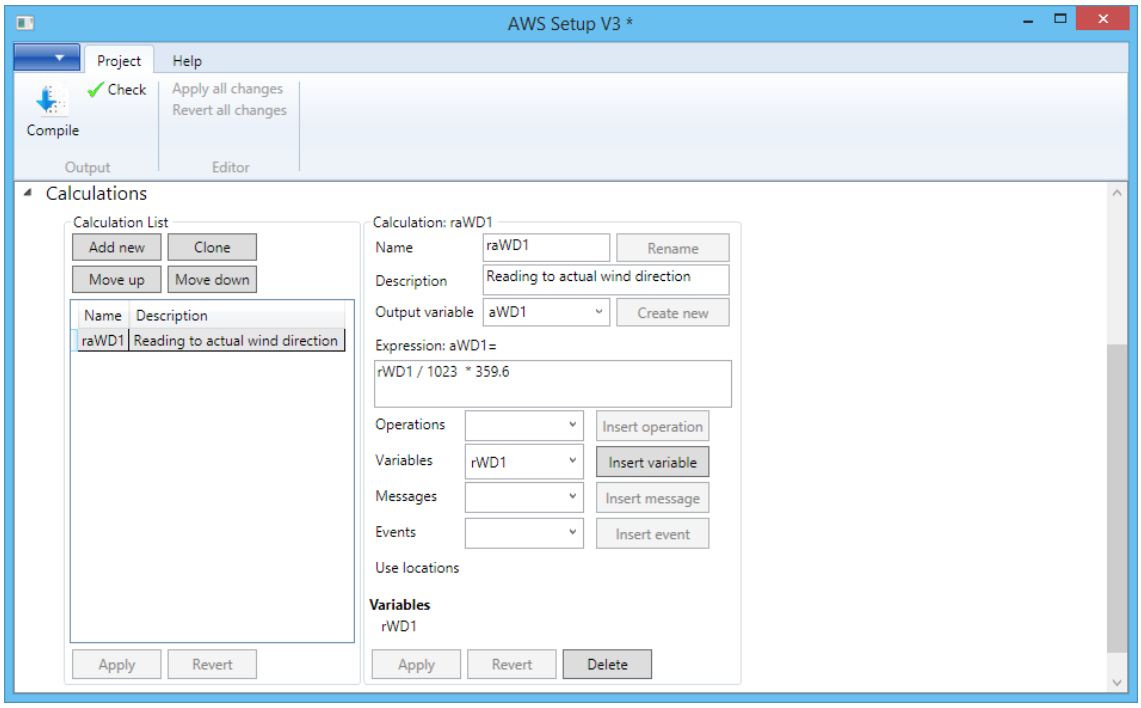



*Figure 15: Calculations*

Depending on the type of the output variable the operations can do different things. See table for complete reference of the supported commands.

*Table 28: Functions list in calculations*

| Function | Description | Details |
|---|---|---|
| *Standard math functions* | | |
| CABS | Absolute value of character | |
| ABS | Absolute value of integers ONLY | |
| LABS | Absolute value of long | |
| FABS | Absolute value of float | |
| SQRT | Square root | |
| EXP | Exponential | |
| LOG | Natural logarithm | |
| LOG10 | 10 base logarithm | |
| SIN | Sine | |
| COS | Cosine | |
| TAN | Tangent | |
| ASIN | Arcsine | |
| ACOS | Arcos | |
| ATAN | Arctangent | |
| SINH | Hyperbolic sine | |
| COSH | Hyperbolic cosine | |
| TANH | Hyperbolic tangent | |

| Function | Description | Details |
|---|---|---|
| ATAN2 | Arctangent from 2 values | |
| FMOD | Float modulo | |
| TRUNC | Truncate | |
| *System functions* | | |
| IF | Conditional value | 9.2 |
| EVENT | Run event | 9.3 |
| IFEVENT | Conditionally run event | 9.4 |
| IFEVENTS | Choose event to run conditionally | 9.5 |
| SETDATE | Set the system date | 9.6 |
| FTYPE | Read from log file | 9.7 |
| CALIB | Calibrate from file | 9.8 |
| STATUS | Get the data logger status | 9.9 |
| STATBIND | Bind vector statistics | 9.10 |
| STATEVENT | | 9.11 |
| MESSIN | Parse value from string | 9.12 |
| PACKER | Data compression | 9.13 |
| DISC | Discontinuities | 9.14 |
| *Physical/Meteorological functions* | | |
| DEWP | Dew point | |
| VPRESS | Virtual pressure | |
| SPRESS | Station pressure | |
| VIRTTEMP | Virtual temperature | |
| QNH | Pressure adjusted to sea level | |
| QFE | Pressure adjusted to reference level of runway | |
| QFF | Pressure adjusted to sea level using actual temperature | |
| GEOPOT | Geopotential | |
| ACCG | Gravitational acceleration from position | |
| FAO | Food and agriculture organization calculations | |
| CROSS | Wind cross function | |
| TAIL | Wind tail function | |

Calculations may contain these operators:

*Table 29: Operators used in calculation*

| Operator | Description | Details |
|---|---|---|
| *Math operators* | | |
| + | Plus, Concatenate | |
| - | Minus | |
| * | Multiplication | |
| / | Division | |
| % | Modulo | |
| *Logic operators* | | 9.1 |
| = | Equals, logic operator | |
| > | More than | |
| < | Less than | |
| <= | Less than or equal | |
| >= | More than or equal | |

## 9.1 Logic operators

Result of the logic expression is 0 or 1 for numeric values and "0" or "1" for string values. This result may be stored in a variable of passed as an argument for conditional functions.

## 9.2 IF function

Value of a calculation is stored into variables depending on the expression in the first argument. This expression may be a logic operator or any other variable.
For numeric output variables, the value must be zero to evaluate as false statement. Otherwise the value is true.
For string output variable the value has to be "0" (string zero) to evaluate as false.

If the expression is evaluated as true, the second argument is returned, otherwise the third argument is returned.

**Note:** *The expressions are always evaluated, but may be discarded if the does not satisfy the condition. (E.g. in expression "IF (0, EVENT (e1), 0)" the event e1 is always started.) Use IFEVENT for running events conditionally.*

| Syntax: |
|---|
| **IF (***condition***,** *true value***,** *false value***)** |
| Returns: |
| First argument if the condition is not 0 otherwise the third argument. |

## 9.3 EVENT function

When the calculation is evaluated, the event is queued to execute.

| Syntax: |
|---|
| **EVENT (***event name***)** |
| Returns: |
| In non string calculation: |
| 1 if the event is successfully queued for execution, otherwise 0. |
| In string calculation: |
| Returns always "\0" |

## 9.4 IFEVENT function

Run event conditionally. If the condition is false, then the whole calculation is discarded and the destination variable is not updated.

| Syntax: |
|---|
| **IFEVENT (***condition***,** *event name***)** |
| Returns: |
| In non string calculation: |
| 1 if the event is successfully queued for execution, otherwise 0. |
| In string calculation: |
| Returns always "\0" |

## 9.5 IFEVENTS function

Run event conditionally. If the condition is non-zero, the *true event* is queued for execution; otherwise the *false event* is queued for execution.

| Syntax:<br>**IFEVENTS (***condition***,** *true event***,** *false event***)** |
|---|
| Returns:<br>In non string calculation:<br>1 if the event is successfully queued for execution, otherwise 0.<br>In string calculation:<br>Returns always "\0" |

## 9.6 SETDATE function

This function sets the system date and time from variables.

| Syntax:<br>**SETDATE (***day***,** *month***,** *year***,** *hour***,** *minute***,** *second***)** |
|---|
| Returns:<br>Number of seconds since 1/1/1970 of the set value. |

## 9.7 FTYPE function

This function reads string from files. The start denotes the number of line. Use this function only in calculations with output variable of string type.

| Syntax:<br>**FTYPE (***filename***,** *start***,** *count***)** |
|---|
| Returns:<br>The string read from the file. |

## 9.8 CALIB function

This function evaluates the correction for a variable which is defined in calibration configuration file. The variable must exist in the *expression*.
To use this function, the CONFIG.CAL file must exist on disk G:\.

| Syntax:<br>**CALIB (***expression***)** |
|---|
| Returns:<br>Result of expression adjusted by the correction from the file.<br>$value + correction\,(value)$ |

CONFIG.CAL file has the following syntax:

*VariableName1 SymbolName1*
*VariableName2 SymbolName2*
*..*
*VariableNameN SymbolNameN*

Symbols denote where to find the calibration constants. System searches the calibration constants in two places: **ALLINONE.CAL** and *SymbolName***.CAL** files on disk G.

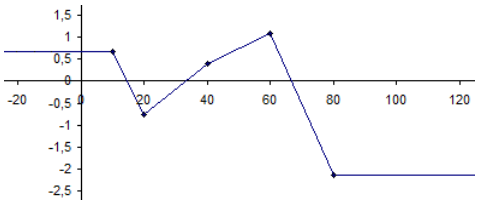
The syntax of calibration file is the following:

*Value1 Correction1*
*Value2 Correction2*
*…*
*ValueN CorrectionN*

Values must be in increasing order. If the value for which the correction is evaluated is between two values, linear interpolation is used to examine the value of correction. If the value is out of the defined values, the closest value of correction is used, not an extrapolated value.

**Example:** *The definition below creates a function of correction shown below.*

```
10 0.659
20 -0.75
40 0.401
60 1.1
80 -2.145
```




The ALLINONE.CAL file structure is below:

**[***SymbolName1***]**
*Table1*
**[***SymbolName2***]**
*Table2*
…

## 9.9 STATUS function

Every variable has its status, which means validity. Is the variable status is zero, the variable is valid. In case that the variable is not valid, it is possible to detect the reason of invalidity. Status is 1 byte number with flags on different bits.

Syntax:

**STATUS (***expression***)**

| Returns: |
| --- |
| The result of bitwise OR for status codes of all variables in *expression*. |

The bits of status code have the following meaning. If the bit is set to 1, the flag is set; otherwise the flag is not set.

*Table 30: Status code binary flags*

| Bit | Flag | Description |
| --- | --- | --- |
| 0 | Timeout | Value was not updated in specified time. |
| 1 | Event | Invalidated by an event (Validity for receive messages) |
| 2 | String | Unable to convert string to number |
| 3 | Min | Minimum value exceeded |
| 4 | Max | Maximum value exceeded |
| 5 | NaN | Not a number (e.g. division by zero) |
| 6 | Out | Sensor is not connected |
| 7 | Disable | Variable is disabled |

The status flags of variables are propagating to resulting variables in the calculations. If some flag is set in any of the calculations' variables, this flag will be set in the resulting variable.
Propagating of flags, which are not connected directly by calculations (e.g. by events, which are not started and fail is not propagated), is possible by defining this behavior in file G:\STATUS.TXT. In this file it is also possible to define which variable disables another variable. If the disabling variable is non-zero, the disabled variables' flag **Disable** is set.

## 9.10 STATBIND function

This function alters the behavior of the statistic, which is identified by its output variable *statisticVariable*.
If the third variable is 128, then it changes the number of ignored extremes. If the statistic calculates the maximum or minimum, by default it stores to output variable the highest/lowest measured extreme during the interval. This function may be used to filter out spikes in measurement and the *ignoredExtremes* extremes are discarded.
This function should be called once, after the system initialization.

| Syntax: |
| --- |
| **STATBIND (***ignoredExtremes***,** *statisticVariable***,** 128**)** |
| Returns: |
| 1 if the command was successful, otherwise 0. |

## 9.11 STATEVENT function

This function alters the behavior of the statistic. Statistics may have internal event during its operation (e.g. minimum/maximum is found). In these events the result is updated. By this function is possible to run a user defined event on these internal events.
This function should be called once, after the system initialization.

| Syntax: |
| --- |
| **STATEVENT (** *statisticVariable***,** *event***)** |
| Returns: |
| 1 if the command was successful, otherwise 0. |

## 9.12 MESSIN function

This function initiates the parsing process from string *variable*. In this case the value is not parsed from serial line. The string is passed to all messages in the interval to try parsing the values, until one of them successfully parses the message.
The interval means the messages in the **message list** between the *first message* and *last message*. If only one message is needed, use the same message as second and third parameter.

| Syntax: |
|---|
| **MESSIN (***variable***,** *first message***,** *last message***)** |
| Returns: |
| Always empty string ("\0"). |

Every message in the sequence must have at least one event, in which the message is used as input message, even if this event is not used on existing serial line. If the parsing from existing serial line is not wanted, define a serial line which does not really exist (COM24 and higher) to ensure that the message will not be parsed from an existing line.
In this event the variable validity interval is defined and this event may run another calculations if wanted.

## 9.13 PACKER function

The PACKER function compresses and decompresses the data using various formats, which is defined in *mode* parameter.
This function works only for string variables.

| Syntax: |
|---|
| For modes 0-3 and 5: |
| **PACKER (***mode***,** *stringToPack***,** unused**)** |
| For mode 4: |
| **PACKER (***mode***,** *stringToPack***,** *bufferLength***)** |
| For modes 6-21: |
| **PACKER (***mode***,** *sourceFile***,** *destinationFile***)** |
| Returns: |
| The resulting string for modes 0-5. |
| For modes 6-13: |
| "0" on success, otherwise error code |
| For modes 14-21: |
| Always "0" |

*Table 31: PACKER function arguments*

| Argument | Meaning |
|---|---|
| mode | Operation mode |
| stringToPack | String which has to be compressed |
| unused | Fill this parameter with any value (e.g. 0) |
| sourceFile | Source file to be compressed. This file will remain unchanged. |
| destinationFile | Destination file where the result of compression is stored. |

*Table 32: PACKER function modes*

| Mode | Description |
|---|---|
| 0 | Unpack string until first error |
| 1 | Unpack or return error message |
| 2 | Unpack only string with good checksum |
| 3 | Unpack only string with good checksum except last with error |
| 4 | Unpack only *bufferLength* bytes. |
| 5 | Pack string. |
| *Blocking functions* | |
| 6 | Pack file to **gzip**. |
| 7 | Unpack file from **gz**. |
| 8 | Unpack file from **rec**. |
| 9 | Unpack file from **rec** and then pack to **gzip**. |
| 10 | Unpack file from **agz** and then pack to **gzip**. |
| 11 | Unpack file from **agz**. |
| 12 | Unpack file from **rgz**. |
| 13 | Unpack file from **rgz** and then pack to **gzip**. |
| *Non-blocking functions* | |
| 14 | Pack file to **gzip**. |
| 15 | Unpack file from **gz**. |
| 16 | Unpack file from **rec**. |
| 17 | Unpack file from **rec** and then pack to **gzip**. |
| 18 | Unpack file from **agz** and then pack to **gzip**. |
| 19 | Unpack file from **agz**. |
| 20 | Unpack file from **rgz**. |
| 21 | Unpack file from **rgz** and then pack to **gzip**. |

## 9.14 DISC function

This function calculates whether discontinuity in measurement occurred. When the discontinuity occurs, the statistics are reset.

Syntax:

For positive first argument:

**DISC (** *diff1*, *abs1*, *diff2*, *abs2*, *statVar1*, *statVar2* **)**

For negative first arguments:

**DISC ( -1,** *length*, unused, unused, *destStatVar*, *sourceStatVar* **)**

**DISC ( -2,** *length*, *diff*, *abs*, *destStatVar*, *sourceStatVar***)**

**DISC ( -3,** *MinMax*, *before*, *rise*, *after*, *statVar* **)**

**DISC ( -4,** *cross1*, *cross2*, *cross3*, *cross4*, *statVar* **)**

Returns:

For first argument -1:

Always 0

For first argument other than -1:

The detected discontinuity level.

The discontinuity is set with first argument -1. The length describes how many samples are preserved. The discontinuity is realized on the statistic with output variable *destStatVar*. If the *sourceStatVar* differs from the *destStatVar* the values of statistic are get from the statistic defined by *sourceStatVar*. Fill the unused arguments with any constant (e.g. zero).

Other function calls return the discontinuity level. The discontinuity level is described in the table.

*Table 33: Discontinuity levels*

| Discontinuity level | Description |
|---|---|
| 0 | Normal operation |
| 1 | Absolute difference between samples exceeds *diff*. |
| 2 | Absolute value of the statistic exceeds *abs*. |
| 3 | Discontinuity level 1 and 2 at the same time |

The evaluation of these levels depends on the sign of the *diff* and *abs* values.

*Table 34: Discontinuity evaluation*

| diff | abs | Description |
|---|---|---|
| >0 | >0 | Normal evaluation, difference and absolute criteria are evaluated. |
| >0 | =0 | Only the difference criteria is evaluated |
| >0 | <0 | Only the difference criteria is evaluated in the circle defined in *abs* (e.g. -360 for degree units) |

# 10. Serial lines

Serial lines of the data logger are communication channels where the data is transferred both ways.

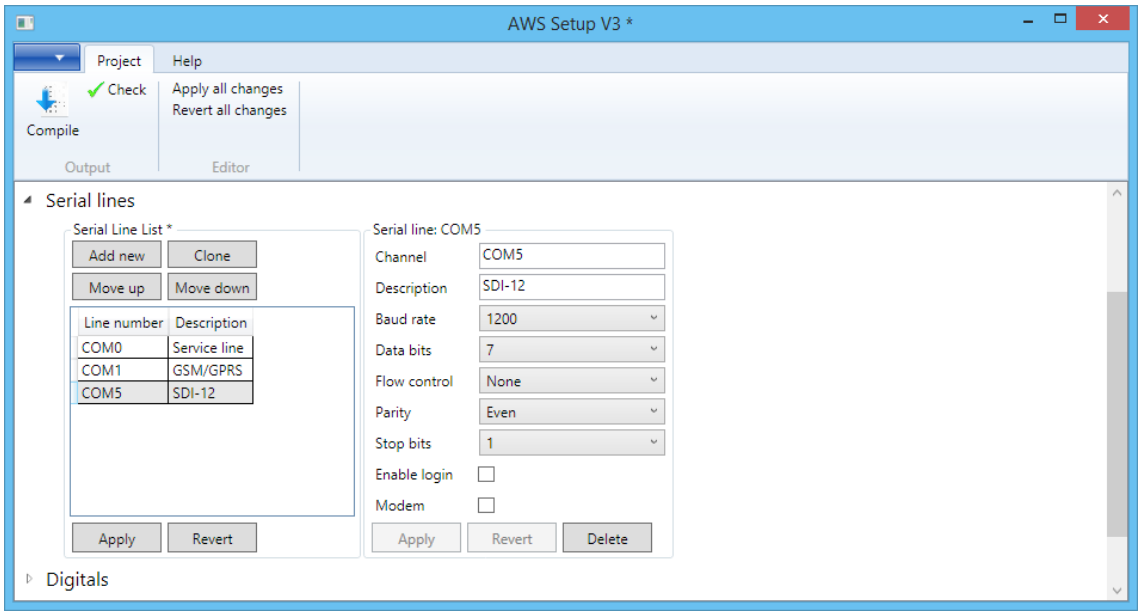



*Figure 16: Serial line*

If the **login** is enabled, the user/software on this line can log in to the command line interface of the data logger.
The **modem** option enables the automatic communication with modem on this communication line.
The serial lines may be physical interfaces, which are connected to a specific COM line, or virtual.

*Table 35: List of serial lines of AMS 111 II*

| Port name | Function |
|---|---|
| COM0 | RS-232 port on AWS 111 Main Board |
| COM1 | Internal GSM or PSTN modem module |
| COM2 | RS-485 port on AWS 111 Main Board |
| COM3 | USB virtual serial line |
| COM4 | Reserved |
| COM5 | SDI-12 |
| COM6 | Reserved |
| COM7 | Reserved |
| COM8 – COM23 | Virtual serial line defined trough TCP/UDP or using serial extension module. |

*Table 36: List of serial lines of AMS 111 IV*

| Port name | Function |
|---|---|
| COM0 | RS-232-V port on AWS 111 Main Board |
| COM1 | Internal GSM or PSTN modem module |
| COM2 | TTL serial on AWS 111 Main Board |
| COM3 | USB virtual serial line |
| COM4 | Reserved |
| COM5 | Reserved |
| COM6 | Internal (line to sub processor) |
| COM7 | Reserved |
| COM8 | RS232-0 |
| COM9 | RS232-1 |
| COM10 | RS485-0 |
| COM11 | RS485-1 |

| Port name | Function |
|---|---|
| COM12 | Reserved |
| COM13 | SDI-12-0 |
| COM14 | SDI-12-1 |
| COM15 | Reserved |
| COM16 – COM31 | Virtual serial line defined trough TCP/UDP. |

*Table 37: List of serial lines of SAWS 111*

| Port name | Function |
|---|---|
| COM0 | RS-232 port on AWS 111 Main Board |
| COM1 | Internal modem module |
| COM2 | RS-485 port on AWS 111 Main Board |
| COM3 | USB virtual serial line |
| COM4 | Virtual serial line defined trough TCP/UDP |
| COM5 | Virtual serial line defined trough TCP/UDP |
| COM6 | SDI-12 |
| COM7 | Limited virtual serial line for NTP |

Virtual serial lines may be configured by setting variable `VIRTUALCOM`*`nn`* with a string with following syntax:

*Type*:*Parameter1[[*:*Parameter2]*:*Parameter3]*

*Table 38: Virtualcom variable syntax*

| Type | Parameter1 | Parameter2 | Parameter3 | Description |
|---|---|---|---|---|
| TCP | *IP address* | *Port[,timeout[**D**]]* | <not used> | TCP client |
| TCP | 0.0.0.0 | *Port[,timeout]* | <not used> | TCP server |
| UDP | *IP address* | *Port[,timeout[**D**]]* | <not used> | UDP client |
| UDP | 0.0.0.0 | *Port[,timeout]* | <not used> | UDP server |
| SER | *Extension channel* | <not used> | <not used> | Bind virtual serial line with extension board channel |
| NTP | *NTP server IP address* | Port (usually 123) | | SNTP client for time synchronization |

Argument *timeout* for TCP/UDP modes is in seconds. If nothing is queued to write to this serial line, the connection is terminated. If the timeout is not defined, the line remains connected until manual disconnect.

**Example 1:** *Command "*SETS VIRTUALCOM08=TCP:0.0.0.0:4001*" starts TCP server on virtual serial line COM8 listening on port 4001.*

**Example 2:** *Command "*SETS VIRTUALCOM09 = TCP:192.168.1.50:4001*" starts TCP client on virtual serial line COM8, which connects to the IP address 192.168.1.50 on port 4001.*

**Example 3:** *Command "*SETS VIRTUALCOM10 = SER:0*" binds virtual serial port COM10 with the channel 0 on serial extension board.*

SNTP (Simple Network Time Protocol) may be used to synchronize the data logger time over the network. To trigger the synchronization, send any character to the chosen virtual line. (Event with action Send which sends message). Conditions:
The VIRTUALCOM*nn* must be set correctly to NTP, the Serial Line must exist in configuration and the network communication must be correctly set (over modem or Ethernet).

**Attention:** *After changing any of VIRTUALCOMnn variables, restart of data logger is needed to take effect.*

# 11. Statistics

Statistics are intended to periodically sample values in the variables and store calculated values in another variable. These operations may be:

*Table 39: Statistic operations*

| Operation | Description |
|---|---|
| Average | Average value of the variable during the period |
| Average angle | Average for degree units (0– 360°) |
| Minimum | Minimum value of variable during the period |
| Minimum angle | Minimum for degree units |
| Maximum | Maximum value of variable during the period |
| Maximum angle | Maximum for degree units |
| Sum | Sum of the samples during the period |
| Difference | Difference between actual interval and previous interval. |
| Standard deviation | The standard deviation of the variable during the period |

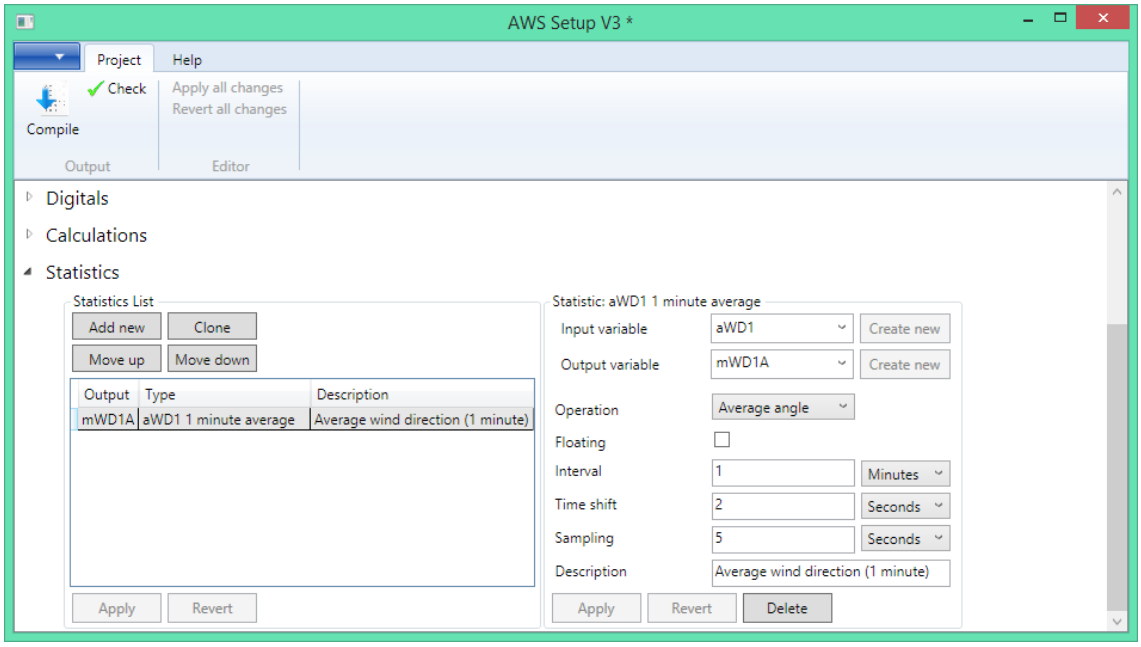



*Figure 17: Statistics*

**Floating** means that the output variable is updated after every sample. If floating is not checked, the value is updated once in the **interval**. In the figure is an example time graph of average statistics. The same applies to other types of statistics too.

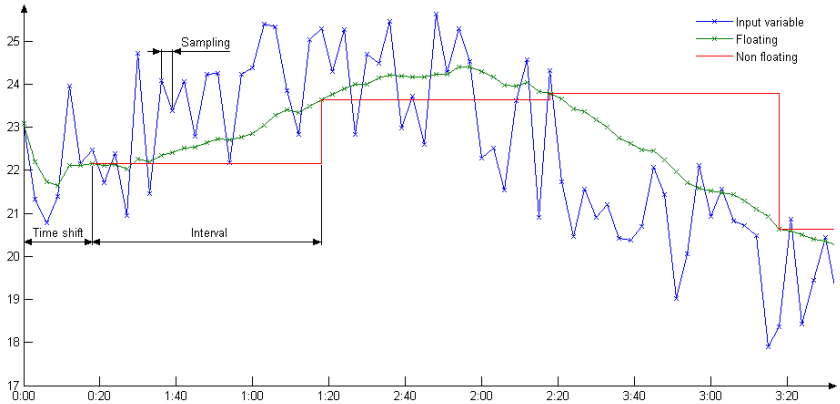



*Figure 18: Statistics average graph*

# 12. Macros

Macros are intended to automatically insert items into the configuration. These may be not only fixed copies, but they may be parameterized. That means that you can insert the same macro twice, but with a little difference. These parameters can be e.g. measurement interval, sensor range, etc.

AWS Setup has some preinstalled macros for sensor support and for commonly used configuration steps.

The **macro definition** is the file, which describes what to do when the macro is used and which parameters may be changed when the macro is used.
The **macro instance** is the actual usage of a macro definition. This contains the filled parameters.

In the **Configuration tab** the list of **macro instances** can be managed. The workstation contains global **macro definitions** (which can be used in different projects) and the actually opened project contains **macro instances**.

When a macro is used, its definition is also copied to the project file. This way you can open it on any computer, even where the global macro definition is not present. Once the file is saved, it will use the same macro definition, regardless of how the global macro definition is changed.

The macro definitions can be managed on the **Preset** -> **Manage macros** menu. Here you can import/export macro definitions and update the outdated macro definitions which are stored in a project file. Also if you want to use an updated macro definition in your project, you can replace it by a newer global version.

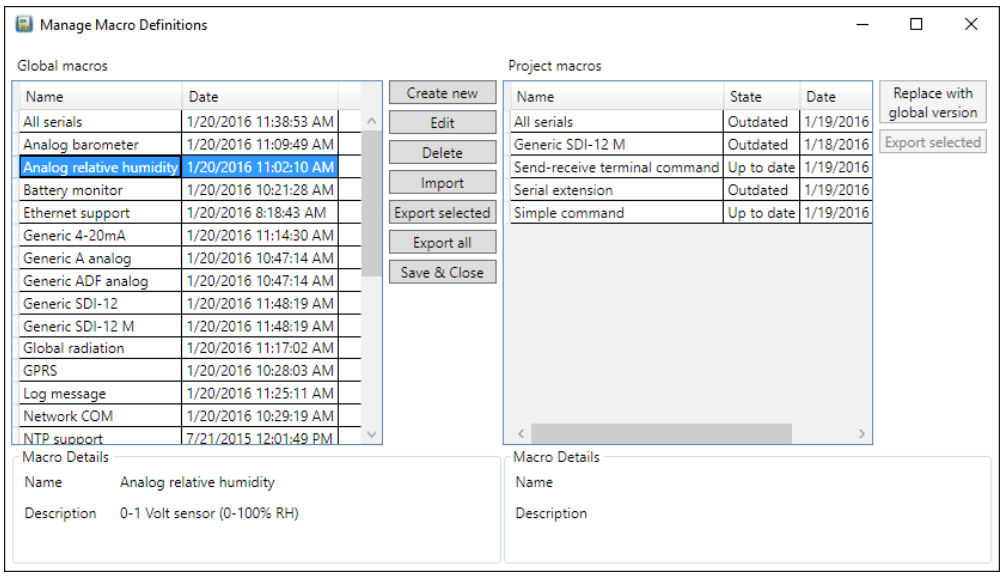



*Figure 19: Manage macros window*

To execute the macros click on **Check Warnings** button or **Preset** -> **Execute macros**. The macros are executed also if you click **Compile** or **Compile compressed** buttons. The created macros are not editable, because after each execution they are updated, so changes would be overwritten.

## 12.1 Using macros in project

To use a macro go to **Configuration** tab and expand the **Macros** category.

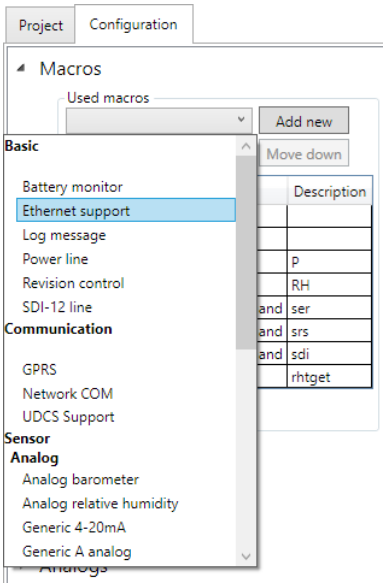



*Figure 20: Macro instances*

Select the desired macro and click Add new. If you can't find any suitable macro, you can create a new one (Chapter 12.2). After you added the macro it is shown in the list. Click on the macro and on the right side an editor appears with the possible parameters. The parameters are set to default values after creating the macro. If you want to change a parameter, edit it, and then click on **Preset** -> **Execute macros**. This will cause re-creation of items with updated parameters.

## 12.2 Creating and editing macro definitions

To create/edit the macro definitions click **Preset** -> **Manage macros** (Figure 19). If you want to create a new macro definition, click on **Create new**. You have to choose a unique name of macro, which will be used to identify the macro definition in the project. After you chose the name, it **cannot be changed**.

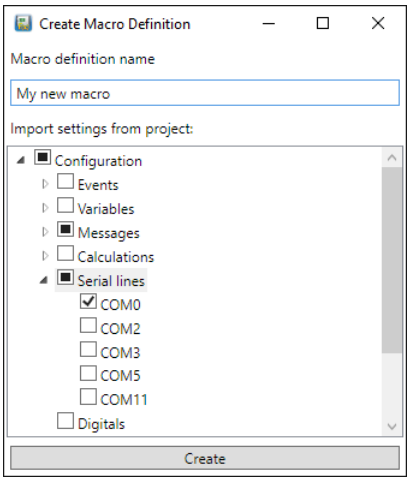



*Figure 21: Create macro definition window*

You can import settings from the actually open project. Check the checkboxes near the desired items. This is useful, if you **create** the configuration manually, **test** it, and then if everything works, you **create a macro** to be able to reuse your settings later in another project. When the chosen name is unique, you create a macro by **Create** button, and then it is followed by editing the newly created macro. The editing is the same like for the existing items (choosing **Edit** command in manage macros window). You cannot change the ID of macro definition.

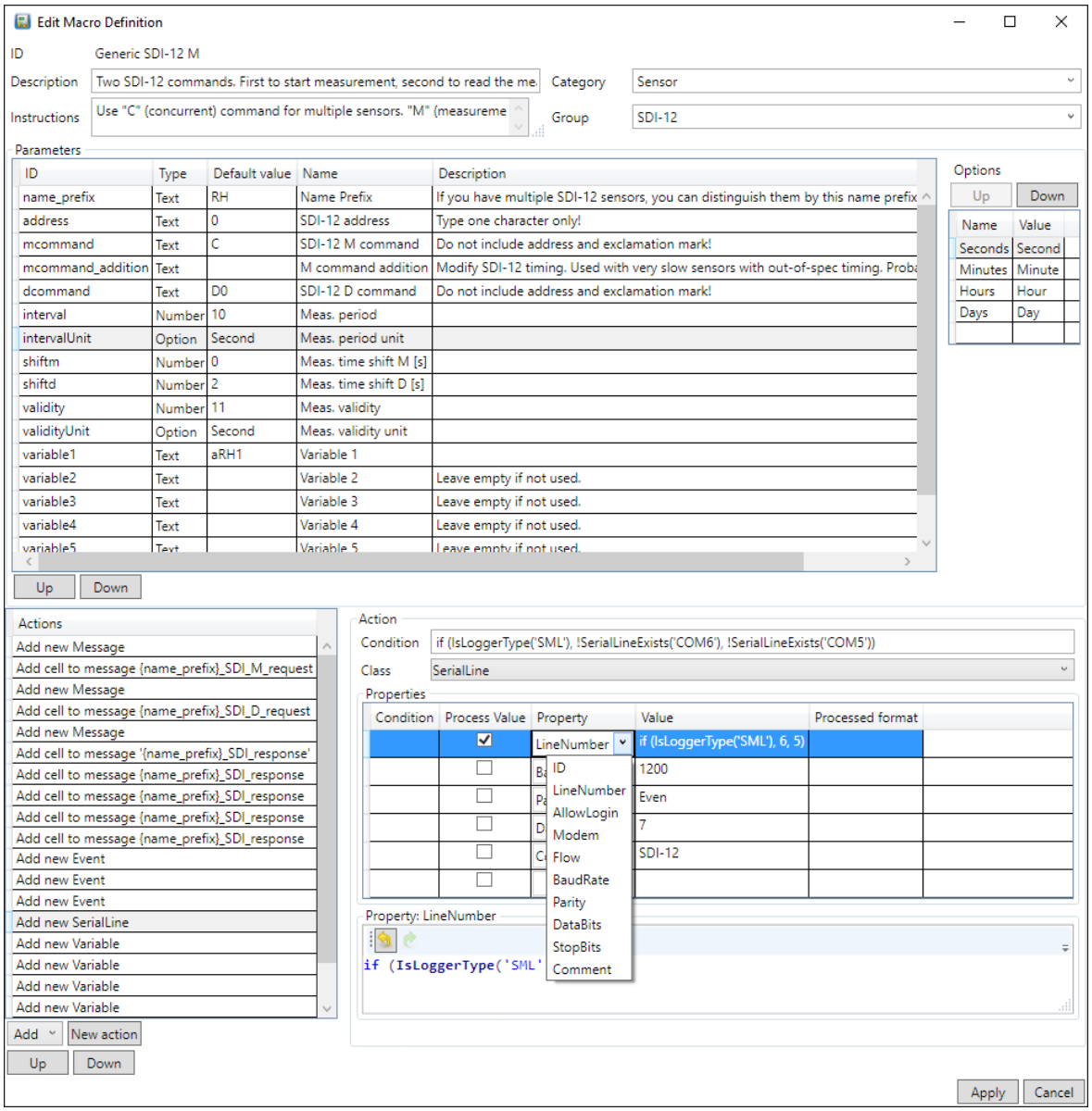



*Figure 22: Edit macro definition window*

The upper part of the window contains some information about the macro. The **Description** is shown in the Manage macros window. The **Instructions** are shown in the project, above the parameters. The **Category** and the **Group** are used to group the macros logically in the project (Figure 20).

## Parameters

In the middle part you can define a parameter by writing to the last (empty) row of the table and hitting Enter. You can reorder the parameter list by Up/Down buttons below the table.
A parameter must have an ID. This variable works like a variable name.

**Note:** *If you change the ID of parameter in a macro definition which was already used in an existing project, the customized value in project will be changed to default value after updating the project macro. It is due to the impossibility to identify the value, when the ID changes. If the parameter ID remains the same, then the entered value in the project remains the same, even is you change other properties of the parameter.*

The **Type** of a parameter defines which values can be entered by user (Table 40). The **Default value** is the value which the parameter has after the creation. The **Name** is shown to user and the **Description** is shown behind the question mark next to the editor.

*Table 40: Macro parameter types*

| Type | Description |
|---|---|
| **Number** | Integer number. Editable by entering a number. |
| **Real** | Decimal number (floating point). Editable by entering a number. |
| **Text** | Editable by entering a text. |
| **Boolean** | True/False. Editable by checkbox |
| **Option** | One of the options. Editable by entering a choosing from a list. The options offered to user are defined on the right side, near to the table of parameters. The **Name** is displayed to the user and the **Value** is used in macro as a value. The **Default Value** must be one of the **Values** in the table |

## Actions

In the bottom part of the window you can define actions, which will change the configuration file when the macro is executed. There are 4 types of actions: **Add**, **Cell**, **Warning** and **Variable**. The **Add** action will add a new item to the configuration, the **Cell** will add a row to the message and the **Warning** will add warnings or errors to the warning list of the project, when the condition is met. To add a new action, select the action type below the action table and click on **New Action** button.

Every action has a **Condition**, which decides whether the whole action will be performed. If this action is not filled, the action is by default evaluated. You can use expressions (chapter 12.3) which can decide about the running. The equations are always performed with logical processor. The result of the equation must be Boolean. You can also use constants `true` or `false`.

The **Add** action is used most of a time. This action creates new item in the configuration. The user can add new Variable, Analog, Digital, Message, Calculation, Event, Serial Line, Statistic, Default value or File. These types are listed under the **Class** option. Every type of item has different properties. The list of the allowed properties for the selected **Class** is shown after opening the list under the **Property** column. Each row may have a **Condition**, which are always evaluated using expressions. If the **Process value** is not selected for the row, the **Value** is just filled with parameters (see note in chapter 12.3).
The result of the **Value** is always converted to the right type, which is needed by the selected **Property**.

**Example 1:**

The **Property** is `Comment`, The **Process value** is checked and the **Value** contains `1 + 2`
The **Value** is evaluated as numeric 3, which is converted to text, because the `Comment` is a string.

**Example 2:**

The **Property** is `IntervalUnit`, the **Process value** is checked and the **Value** contains `'Second'`
The **Value** is evaluated as string, which is converted to PeriodUnit type enum, because the type of `IntervalUnit` is an **enum** of the following possible states: **Second**, **Minute**, **Hour**, **Day**. No other values are accepted.

The **Cell** action adds a row to a **Message**. It is useful if the macro is used to build a message from multiple macros. If the Message does not exist, then it is created. If the message exists, it just adds a row. The **Condition** field is evaluated always with logical processor, but the other fields are evaluated depending on the **Process Value** checkbox.

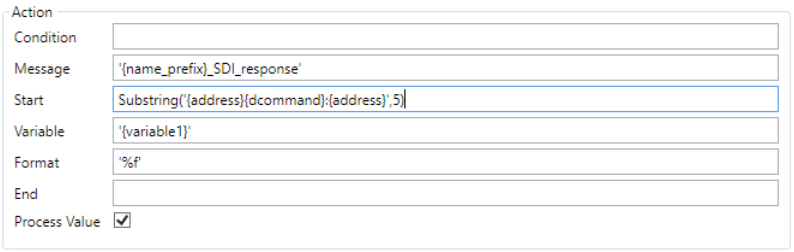



*Figure 23: Add cell action*

The **Warning** action will add warnings or errors to the warning list of the project, when the condition is met. You can choose different levels of warning: **Information**, **Warning** and **Error**. The **Message** is the text which will be shown to user. You can use equations to modify the text with more accurate values.

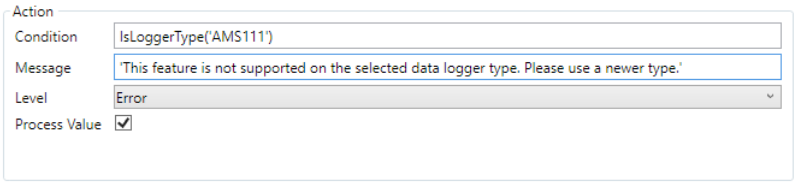



*Figure 24: Warning action*

The **Variable** action is used to create a local internal variable only for executing the macro. The result of this variable is available in the other actions which are defined below this action in the same way as the parameters defined by the user.

## 12.3 Macro evaluation algorithm

When a field is evaluated it processes the text and parameters. Firstly the macro inserts parameters into the equation. The parameters are between `{` and `}` brackets. If the user needs to use `{` or `}` bracket, enter `{{` or `}}` respectively (doubled). Then the logical processor evaluates the result. If there is a **Processed Format** defined, then this value can be formatted (e.g. round the numeric values). Logical processor can use the available functions (see Table 41). The functions have typed parameters and return values. That means, that if the function requires string parameter, it must be between single quotes (') or the result of a function must be string. Similarly, if the function needs a numeric parameter, it must not be a string (e.g. `Cos('12.5')` is not acceptable, just `Cos(12.5)`). The numbers use decimal **point**, not decimal comma (`12,5` is not a number, just `12.5`). You can use scientific notation (e.g. +1.25e+1)

**Example 1:**

The equation is: `'ADF{channel}'`
The channel is a name of a parameter. Assume that the user entered into that field value 5
Then the parameter is inserted into that equation and the result is: `'ADF5'`
Then it goes to logical processor and detects that this is a text (string), because it is between single quotes (')

**Example 2:**

The equation is: `({max}-{min})/{range}`
Assume that the user entered max = 5, min = 1.5, range = 0.5
Then the parameter is inserted into that equation and the result is: `(5-1.5)/0.5`
Then it goes to logical processor and it evaluates it as a numeric equation. `3.5/0.5 = 7`
The result is numeric `7`. If the Processed format is: `{0:0.0}`, then the result is a text: `"7.0"`

**Note:** *The Logical processor can be disabled by unchecking the **Process Value** option (e.g. in Add action, in every property row separately, or in message cell action). Please note that the default state is disabled and you have to turn it on. If the Logical processor is disabled, the whole equation is treated as string. In this case you don't need to enter the*

*strings into single quotes ('). The replacement logic (replacing parameters between { and } brackets) remains the same even when the logical processor is disabled.*

*Table 41: Functions in macro definitions*

| Function | Description |
|---|---|
| string **ActualDate**(*[string **format**="u"]*) | Returns the actual date. The optional argument specifies the datetime format. |
| bool **AnalogExists**(string **analogName**) | Checks whether Analog exists. |
| byte[] **Base64Decode**(string **base64**) | Converts base64 text to byte array |
| string **Base64Encode** (byte[] **binary**) | Converts binary array to base64 text |
| bool **CalculationExists**(string **calculationName**) | Checks whether Calculation exists. |
| int **ConfigurationVersion**() | Returns the actual configuration version number. |
| string **ConstantSeparateSign** (float **number**) | Adds space between sign of number. |
| bool **DefaultValueExists**(string **defaultValueName**) | Checks whether Default value exists. |
| string **DefaultVariableValue**(string **id**) | Gets the value of already defined Default value, or empty if not defined. |
| bool **DigitalExists**(string **digitalName**) | Checks whether Digital exists. |
| bool **Equals**(string **t1**, string **t2**) | Checks whether the two strings are equal. |
| string **Escape**(string **text**) | Escapes backslash characters from the argument. (\r, \t, etc.) |
| bool **EventExists**(string **eventName**) | Checks whether Event exists. |
| bool **FileExists**(string **filename**) | Checks whether Data logger file exists. |
| string **FileName**() | Returns the actual project file name. |
| string **FilePath**() | Returns the actual project file path. |
| string **Format**(any **anything**, string **format**) | Converts anything to text using the provided format. (.Net syntax) |
| byte[] **GetBytes**(string **text**) | Gets byte array of text using UTF-8 encoding. |
| string **GetComName**(string [int] **line**, string **parameter**) | Get serial line signal name. parameter is e.g. TxD for RS232 or A+ for RS485. Line can be either 'COM1' or simply 1. |
| string **GetString**(byte[] **binary**) | Get string from binary array using UTF-8 encoding. |
| bool **GreaterThan**(double **num1**, double **num2**) | Compares two numbers |
| string **HexaByte**(ulong **data**, int **byteNumber**) | Converts a number to hexadecimal byte (2 characters). Byte number is a number from 0 to 3. |
| string **IdentificationVariable**() | Returns the actually selected identification variable. (default LOGID) |
| string **IdentificationVariableType**() | Returns the data type of the actually selected identification variable. |
| any **Index**(any **index**, any **val1** *[, any **val2** [, any **val3**…]]*) | Selects val1 if index is 1, val2 if index2, etc. |
| bool **IsEmpty**(string **text**) | Checks whether text is empty. |
| bool **IsLoggerType**(string **requestedType**) | Checks whether the data logger type is the entered type. |
| bool **IsNotEmpty**(string **text**) | Checks whether text is not empty. |
| string **LoggerType**() | Returns the actual data logger type. |
| int **LoggerTypeNum**() | Returns the actual data logger type number. |
| bool **MessageExists**(string **messageName**) | Checks whether Message exists. |
| string **PolynomimalEquationText**(string **variable**, params double[] **coefficients**) | Creates an equation text executable by data logger from the array of coefficients. Higher ranks are collected. |

| Function | Description |
|---|---|
| bool **PowerChannelAvailable**(int **powerChannel**) | Returns false if the selected pwrout is not available |
| double **SecondsInTimeUnit**(string **timeUnit**) | Calculates the number of seconds in the specified timeUnit text (e.g. 'Hour' == 3600) |
| bool **SerialLineExists**(string **serialName**) | Checks whether Serial Line exists. |
| bool **StatisticExists**(string **statisticName**) | Checks whether Statistic exists. |
| int **Strlen**(string **text**) | Gets the length of the text. |
| string **Substring**(string **text**, int **startIndex** *[, int* ***length****]*) | Gets the part of the text, beginning at start index and optionally with length |
| int **TerminalCurrentChannel**(int **terminalNum**) | Returns the right internal ADC channel for current loop associated with the provided ADF number. |
| int **TerminalCurrentIndex**(int **terminalNum**) | Returns the right current loop number associated with the provided ADF number. (0->UR0, 1-> UR1) |
| int **TerminalToAnalogChannel**(int **terminalNum**) | Converts ADF channel numbers to internal ADC channel numbers |
| double **TimeUnitMultiplier**(string **fromUnit**, string **toUnit**) | Calculated a conversion factor between two time units (e.g. Hour->Minute = 60) |
| string **ToString**(any **value**) | Converts anything to text using default format |
| bool **VariableExists**(string **variableName**) | Checks whether Variable exists. |
| bool **VariableIsOfType**(string **variable**, string **type**) | Checks whether the Variable exists and the variable type is **type**. |
| **Math Functions** | |
| decimal **Abs**(decimal **number**) | Returns the absolute value of a specified number. |
| double **Acos**(double **number**) | Returns the angle in radians whose cosine is the specified number. |
| double **Asin**(double **number**) | Returns the angle in radians whose sine is the specified number. |
| double **Atan**(double **number**) | Returns the angle in radians whose tangent is the specified number. |
| int **Ceiling**(double **number**) | Returns the smallest integer greater than or equal to the specified number. |
| double **Cos**(double **angleRadians**) | Returns the cosine of the specified angle. |
| double **Exp**(double **number**) | Returns e raised to the specified power. |
| int **Floor**(double **number**) | Returns the largest integer less than or equal to the specified number. |
| double **IEEERemainder**(double **dividend**, double **divisor**) | Returns the remainder resulting from the division of a specified number by another specified number. |
| double **Log**(double **number**, double **base**) | Returns the logarithm of a specified number and base. |
| double **Log10**(double **number**) | Returns the base 10 logarithm of a specified number. |
| double **Max**(double **num1**, double **num2**) | Returns the larger of two numbers. |
| double **Min**(double **num1**, double **num2**) | Returns the smaller of two numbers. |
| double **Pow**(double **number**, double **power**) | Returns a specified number raised to the specified power. |
| double **Round**(double **number**, int **decimalPlaces**) | Rounds a value to the nearest integer or specified number of decimal places. |
| int **Sign**(double **number**) | Returns a value indicating the sign of a number. |
| double **Sin**(double **angleRadians**) | Returns the sine of the specified angle. |

| Function | Description |
|---|---|
| double **Sqrt**(double **number**) | Returns the square root of a specified number. |
| double **Tan**(double **angleRadians**) | Returns the tangent of the specified angle. |
| int **Truncate**(double **number**) | Calculates the integral part of a number. |
| **Conditional Functions** | |
| any **if**(bool **condition**, any **valueTrue**, any **valueFalse**) | If the condition is true, the first parameter is the result, otherwise the second. The type of the both values can be any (text, number, etc.), but must be the same for both. The type of the result is the same as the type the values. |
| bool **in**(any **value**, any **val1** *[, any **val2** [, any **val3**…]]*) | Looks whether the first argument is between the next arguments. The number of the arguments is not limited. |

You can use the following operators to connect the functions:

*Table 42: Operators in macro definitions*

| Operator | Description | Example |
|---|---|---|
| + | Plus or string concatenation. String + number is a string (e.g. 'a' + 5 = 'a5') | 1+2, 'a'+'b' |
| - | Minus | 1-2 |
| * / % | Multiply, Divide, Modulo | 2*3, 1/2, 5%3 |
| \|\| or | Logical OR | true \|\| false, true or false |
| && and | Logical AND | true && false, true and false |
| & \| ^ << >> | Bitwise AND, Bitwise OR, Bitwise XOR, Left shift, Right shift | 1 & 2, 1 \| 2,<br>3 ^ 2, 1 << 2, 2 >> 1 |
| ~ | Bitwise NOT | ~2 |
| ! not | Logical NOT | !true, not true |
| == | Comparison (Equals) | 1 == 1 |
| = | Comparison (Equals) | 1 = 1 |
| != <> | Comparison (Not equals) | 1 != 2, 1 <> 2 |
| < <= > >= | Less than, less than or equal, more than, more than or equal | 1 < 2, 2<=2, 2 > 1, 2 >= 2 |

# 13. Troubleshooting

Sometimes the configuration structure does not allow defining some operations easily. In these cases try to do changes, which are described below.

## 13.1 Message is too long

If a message is too long, and does not fit into system limit 64 bytes total (with variables, formats, start/end sections, etc.), then the message has to be stored in a string variable. The resulting message is concatenated from multiple messages in a calculation.

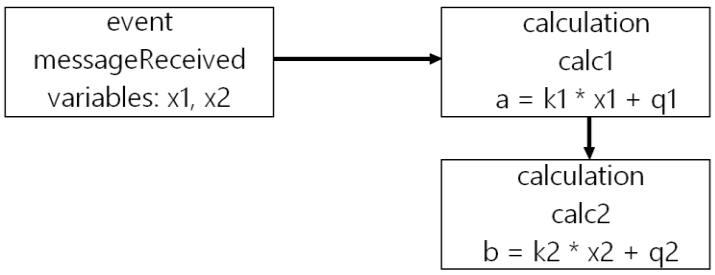
If the message is used in a synchronous event, it is possible to create another event which runs with the same frequency and in the same second. To create an event which stores the concatenated messages into a string variable, move this event before the synchronous event in the list. This ensures that the event (which stores the result to variable) will be executed before the event that send it to serial line or logs it to the file. Then define a message, which has one variable (the saved one) with format %s.

If the message is used in asynchronous event as an output message, then store the messages in a calculation and then start a new event from this calculation using EVENT (9.3) function. This event should print/log the resulting message from variable.

If the message is used in asynchronous event as an input message, replace some fields to be parsed with format %s to string variable. Change the type of event action to Receive-Run. Then add a calculation which starts the parsing of the rest, which was not parsed in this message. This can be achieved by MESSIN (9.12) function. Specify a message which parses the rest, and create an event for it. If this message is still not enough, repeat the same steps. (Variable > MESSIN > message > event)

## 13.2 Run multiple calculations asynchronously

Sometimes it is required to run multiple calculations on single asynchronous event (with action: Receive–Run).




Run a calculation calc1 from event messageReceived.
In calculation calc1 define the expression:

```
(k1 * x1 + q1) * EVENT(calc2)
```

This will calculate the expression "k1 * x1 + q1" and multiply the result by 1, which is the result of the EVENT function (9.3). The result will not change, because the return value should be always 1 for numeric values if no error is occurred.

Create calculation calc2 with expression:

```
k2 * x2 + q2
```

If the message is used for string output variable (e.g. concatenate strings), the result of EVENT function is "\0" which may be concatenated to end of the string, without changing the resulting string. ("\0" is string end character)

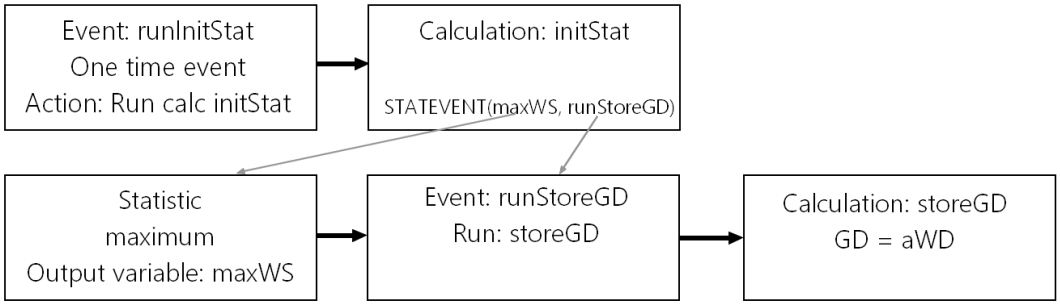
## 13.3 Run calculation on statistics event

Typical application of this is to capture the wind direction when the maximum speed is detected (Gust direction). The value is monitored in a statistics. When the statistics detects the defined (internal) event (e.g. minimum/maximum is found) we want to store the value of another variable. On this internal event it is possible to run a defined event. To set up the data logger with this behavior, run a calculation on system initialization with the expression below, where *statVar* is the output variable for the statistics and *event* is the event which has to be started on internal event.

```
STATEVENT (statVar, event)
```

It is possible to set up multiple statistic events in one calculation:

```
STATEVENT (statVar1, event1) * STATEVENT (statVar2, event2)
```



# References

[1] MicroStep-MIS (2014): AMS111 Terminal boards, Reference manual.
[2] MicroStep-MIS (2015): AWS Service, User guide