# Communication in MODBUS RTU network (PseDet)

**Abstract**

The Application note describes the use of MODBUS RTU protocol in PseDet control systems using a table definition. The Application note deals with both communication with other AMiT products in Slave mode as well as with communication with a superior Master unit.

Author: Zbyněk Říha
File: ap0008_en_06.pdf

**Attachments**

File contents: ap0008_en_06.zip

| | |
|---|---|
| modbs_p1_en_06.dso | Example of a control system parametrization as master. |
| modbs_p2_en_05.dso | Example of a control system parametrization as slave. |
| modbs_p3_en_05.dso | Programme operation **DMM-DI24**. |
| modbs_p4_en_05.dso | Programme operation **DMM-DO18**. |
| modbs_p5_en_05.dso | Programme operation **DMM-RDO12**. |
| modbs_p6_en_05.dso | Programme operation **DMM-AI12**. |
| modbs_p7_en_05.dso | Programme operation **DMM-AO8x**. |
| modbs_p8_en_05.dso | Programme operation **DMM-PDO6Ni6**. |
| modbs_p9_en_02.dso | Programme operation **DMM-UI8DO8**. |
| modbs_p10_en_02.dso | Programme operation **DMM-UI8RDO8**. |
| modbs_p11_en_02.dso | Programme operation **DMM-UI8AO8U**. |
| modbs_p12_en_02.dso | Programme operation **AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)**. |
| modbs_p13_en_01.dso | Programme operation **AMR-OP7xC**. |
| modbs_p14_en_01.dso | Programme operation **AMR-OP7xRHC**. |
| modbs_p15_en_01.dso | Programme operation **AMR-OP40(RH)C**. |

**Copyright (c) 2019, AMiT®, spol. s r.o.**
amitomation.com

## Contents


Contents ... 2
Revision history ... 5
Related documentation ... 5
**1 Definitions of terms ... 6**
**2 MODBUS protocol ... 7**
2.1 Supported MODBUS functions ... 7
**3 Connecting the communication network ... 8**
**4 Time conditions in the network ... 10**
4.1 Communication period ... 10
Example ... 10
4.1.1 Communication priorities ... 10
Automatic Read priority ... 10
Automatic Write priority ... 11
Manual communication priority ... 11
4.1.2 Gathering communication frames ... 12
Example ... 12
4.2 Communication in the event of a connection failure ... 12
4.3 Connection failure detection in modules DMM-xxx ... 13
**5 Control system as Master ... 14**
5.1 Communication definition ... 14
5.2 Definition of a Slave station and data points for communication ... 15
5.3 Automatic communication ... 16
5.4 Manual communication ... 17
5.5 Communication statuses ... 18
5.6 Example of a control system parametrization as Master ... 19
**6 Control system as Slave ... 21**
6.1 Communication definition ... 21
6.2 Definition of data points for communication ... 22
6.3 Communication statuses ... 22
6.4 Example of a control system parametrization as Slave ... 22
**7 Appendix A ... 25**
7.1 Compatibility with communication initialization via modules ... 25
7.1.1 MODBUS Master ... 25
7.1.2 MODBUS Slave ... 25
**8 Appendix B ... 26**
8.1 Programme operation DMM-xxx ... 26
8.1.1 DMM-DI24 ... 26
Working with classic digital inputs ... 26
Working with counter inputs ... 26
8.1.2 DMM-DO18 ... 27
Working with classic digital outputs ... 28
Working with outputs in PWM mode ... 28
Retroactive reading of outputs and PWM parameters ... 28

8.1.3 DMM-RDO12 ..... 29
Working with digital outputs ..... 29
Retroactive reading of outputs ..... 29
8.1.4 DMM-AI12 ..... 29
Working with analogue inputs ..... 30
8.1.5 DMM-AO8x ..... 30
Working with analogue outputs ..... 31
Working with LED ..... 31
Retroactive reading of outputs and LED behaviour parameters ..... 31
8.1.6 DMM-PDO6NI6 ..... 32
Working with RTD inputs ..... 32
Working with classic digital outputs ..... 33
Working with outputs in PWM mode ..... 33
Retroactive reading of outputs and PWM parameters ..... 33
8.1.7 DMM-UI8DO8 ..... 33
Working with analogue inputs ..... 34
Working with digital inputs ..... 34
Working with digital outputs ..... 34
Retroactive reading of outputs ..... 35
8.1.8 DMM-UI8RDO8 ..... 35
Working with analogue inputs ..... 36
Working with digital inputs ..... 36
Working with digital outputs ..... 36
Retroactive reading of outputs ..... 36
8.1.9 DMM-UI8AO8 ..... 36
Working with analogue inputs ..... 37
Working with digital inputs ..... 37
Working with analogue outputs ..... 37
Working with LED ..... 38
Retroactive reading of outputs and LED behaviour parameters ..... 38
**9 Appendix C ..... 39**
9.1 Programme operation AMR-OPxx ..... 39
9.1.1 AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH) ..... 39
Processing the status after a controller restart or a communication failure ..... 39
Loading new values from the controller ..... 40
Writing actual values into the controller ..... 40
9.1.2 AMR-OP7xC ..... 40
Processing the status after a controller restart or a communication failure ..... 41
Loading new values from the controller ..... 41
Writing actual values into the controller ..... 41
9.1.3 AMR-OP7xRHC ..... 42
Processing the status after a controller restart or a communication failure ..... 42
Loading new values from the controller ..... 42
Writing actual values into the controller ..... 42
9.1.4 AMR-OP40(RH)C ..... 43
Processing the status after a controller restart or a communication failure ..... 43
Loading new values from the controller ..... 43
Writing actual values into the controller ..... 43

**10 Technical support ..... 44**

**11 Warning..... 45**

## Revision history

| Version | Date | Changes by | Changes |
|---|---|---|---|
| 001 | 30. 04. 2008 | Říha Z. | New document. |
| 002 | 13. 04. 2010 | – | Descriptions in module use examples modified, sample applications modified, table with communication results revised, chapter 7 modified. |
| 003 | 20. 04. 2012 | – | Chapter 7.1.1 complemented with information on the use of matrices in the module RmtDef. Sample applications created in DetStudio version 1.7.0. |
| 004 | 05. 08. 2013 | – | Chapter 5.2 modified, new examples added, example descriptions in chapter 8 modified Examples created in DetStudio version 1.7.3.44. |
| 005 | 27. 01. 2015 | – | Related documentation modified, chapters 7.1.5, 8.11, 8.12 and images modified. |
| 006 | 05. 02. 2019 | Kupčík M. | Overall revision of the document. |
| | | | |

## Related documentation

1. Help tab in the PseDet section of the DetStudio development environment
   file: PseDet_en.chm
2. Datasheet for the module **DMM-xxx**
   file: dmm-xxx_d_en_xxx.pdf
3. Operation manual for **AMR-OPxx**
   file: amr-opxx_g_en_xxx.pdf
4. Description of the basic sample application for **AMR-OP3xA**
   file: ta-op3x_fw01m_en_xxx.pdf
5. Application note AP0016 – Principles of RS485 interface usage
   file: ap0016_en_xx.pdf

# 1 Definitions of terms

**PseDet control system**
They are control systems and terminals made by AMiT in which process algorithms are programmed in a section of the DetStudio environment called PseDet. E.g. **AMiNi4DW2**, **ADiS**, **AMAP99W3** or **APT4000W3**.

**Master station**
This station actively communicates with Slave stations. There is only a single Master station on a single communication interface.

**Slave station**
It is a station with a unique address which passively listens on the communication interface and responds only after receiving a particular frame from the Master. There may be up to 247 Slave stations.

**RS485**
It is a half-duplex serial bus enabling communication of multiple stations at a single signal pair. The maximum number of stations connected on a single segment depends on the device type. The number ranges from 32 to 256. More information is available in the document AP0016 – Principles of RS485 interface usage.

**Data point**
It is a definition of a register (input or output) or binary (input or output) which usually represents an input or output on a Slave station. Each data point is directly assigned a (matrix) variable or bit into which read values are to be written or from which values for writing into the Slave station shall be taken.

**Modules DMM-xxx**
These modules allow us to extend the number of inputs and outputs in devices programmed as MODBUS Master by using the MODBUS RTU protocol. We can connect up to 63 modules into a single MODBUS network.

**On-wall controllers AMR-OPxx**
On-wall controllers may listen on RS485 interface as MODBUS RTU Slave stations either thanks to an application loaded into them within production, or to a loaded specific sample application or even using a user-created application which uses the communication object **`ModbusSlave`** or **`SerialBusN`**.

# 2 MODBUS protocol

MODBUS is an open communication protocol developed by the Modicon company. Originally, the protocol was designed for an RS232 bus; however, it soon transitioned to RS485 because of its better reliability and options of connecting multiple devices at longer distances. The protocol is flexible but at the same time easy to implement, and therefore soon various producers started implementing it into their devices. At present, communication via MODBUS protocol is supported not only in microcontrollers or industrial PCs, but also by a variety of intelligent sensors, action elements and other simple controls.

On the other hand, the protocol precisely defines e.g. time conditions in the network and error responses. If the counterpart fails to respect these regulations, communication with such devices will not work.

Certain implementations of the MODBUS protocol even support Multimastering; however, it is not supported in AMiT systems. AMiT supports communication via MODBUS protocol in extension modules and communication converters of the series **DMM-xxx** (MODBUS RTU) and in all of its control systems, control terminals and programmable controllers as well as in controllers with interfaces RS232 or RS485 (MODBUS RTU, in case of PseDet control systems also MODBUS ASCII). The master/slave determination depends on a specific implementation.

***Note***
*Communication interface of the control system where MODBUS network is connected cannot be used to connect a device with another communication protocol.*

***Caution!***
*Various producers may have various interpretations of data point addressing, despite MODBUS protocol specification. Find out more in the Help section of DetStudio called "PseDet – Creating control processes", in chapter "Contents/Communication/Modbus" in the section "Addressing Registers/Binaries".*

## 2.1 Supported MODBUS functions

PseDet control systems made by AMiT support the following functions of the MODBUS protocol. Functions stem from the MODBUS protocol definition and define the type of the frame used.

| Function No. | Description |
|---|---|
| 1 | Reading status of binary outputs (coils). |
| 2 | Reading binary inputs. |
| 3 | Reading output (holding) registers. |
| 4 | Reading input (holding) registers. |
| 5 | Setting one binary output (coil). |
| 6 | Setting one output (holding) register. |
| 15 | Setting binary outputs (coils). |
| 16 | Setting output (holding) registers. |

The stated description is only general and for orientation. Specific descriptions of individual functions depend on specific type of device.

For analogue values, we frequently use pairs of registers, so writing analogue outputs is performed using function No. 16. Find out more in the Help section of DetStudio called "PseDet – Creating control processes", in chapter "Contents/Communication/Modbus" in the section "Communication Points Mapping".

# 3 Connecting the communication network

In order for the entire MODBUS network to work properly, it is necessary to design, connect and configure individual network modules and to programme communication in control systems.

Communication via MODBUS protocol usually uses RS485 network. We can connect another device to the control system (e.g. modules **DMM-xxx** made by AMiT) directly to the RS485 interface or to RS232 interface by means of a converter (e.g. **DM-232TO485**).

When wiring the network on a serial interface RS485, it is necessary to follow the recommendations stated in Application note AP0016 – Principles of RS485 interface usage.

An AMiT control system may behave as Master or Slave in a MODBUS network. In a Master role, usually in combination with **DMM-xxx** modules, on-wall controllers **AMR-OPxx** and third-party technological devices (e.g. action elements), or in a Slave role as a part of more complex networks.

Extension modules and communication converters with MODBUS protocol designated **DMM-xxx** are always in the Slave role in the network.

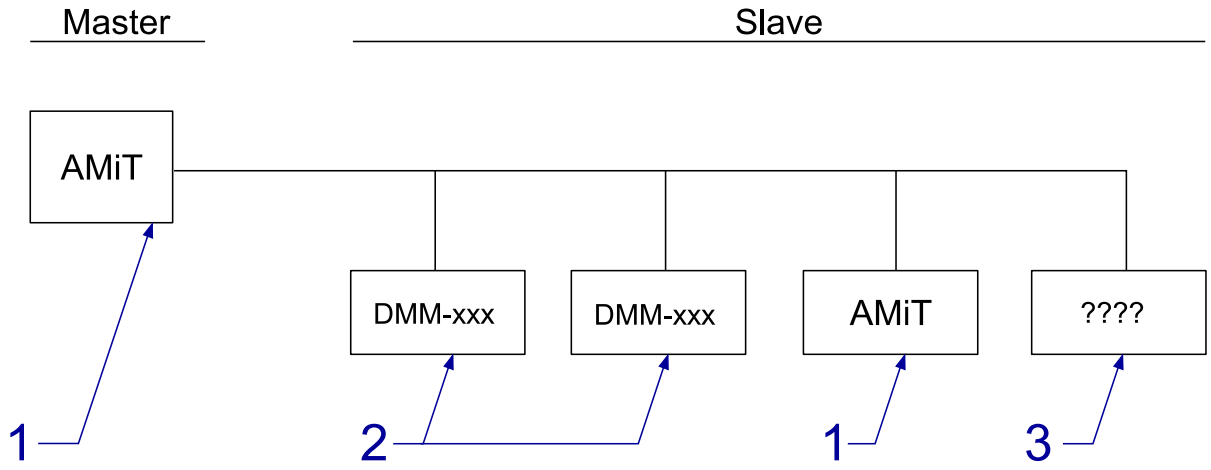



Fig. 1 – Communication via MODBUS protocol directly in the RS485 network

Legend

| Number | Description |
|---|---|
| 1 | AMiT control system |
| 2 | Extension input/output modules made by AMiT |
| 3 | Third-party devices as Slave stations |

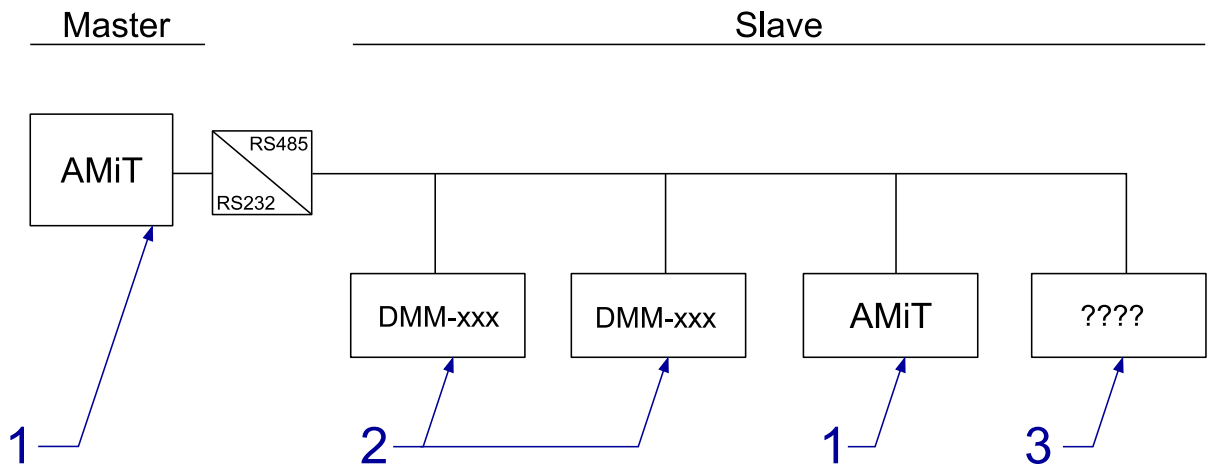



Fig. 2 – Communication via MODBUS protocol using a converter from RS232 or RS485

Legend

| Number | Description |
|---|---|
| 1 | AMiT control system |
| 2 | Extension input/output modules made by AMiT |
| 3 | Third-party devices as Slave stations |

***Note***
*Converter **DM-232TO485** connected to RS232 of an AMiT control system is set as controlled by RTS signal.*

# 4 Time conditions in the network

Communication is executed within certain periods. At the beginning of such period, master activates the interface (based on the definition table, a package of requests is created), and subsequently requests for each remote point stated in the definition table are communicated. The interface activity finishes when the last frame is communicated.

## 4.1 Communication period

Master gradually sends out requests to individual data points in the network and receives their responses. Communication periods of data points are determined by the communication function used and by the amount of data transmitted. Generally, we are able to calculate the communication period length using the following table.

| Transmission speed [bps] | Minimum comm. period [ms] | Data point communication period [ms] |
|---|---|---|
| 9,600 | 40 | 3× register, 1.5× each set of eight binaries started |
| 19,200 | 20 | 1.5× register, 0.8× each set of eight binaries started |
| 38,400 | 15 | 1× register, 0.5× each set of eight binaries started |
| 57,600 | 10 | 0.7× register, 0.4× each set of eight binaries started |

Stated values serve for setting a minimum communication period of one communication row in the table in case the basic time conditions specified by the MODBUS protocol are followed. If the counterpart takes longer to process the communication request, the entire communication gets delayed.

**Example**

We need to communicate with 10 **DMM-xxx** modules periodically. In five, we require communication of 8 registers and in five, we require communication of 18 binaries. Therefore, we define on row of remote point communication for each module (one group of register or binaries). The required communication speed is 19,200 bps. Based on the previous table, we determine the following communication periods:

$T_{reg} = 20 + 1.5 \times 8 = 32$ ms

$T_{bin} = 20 + 0.8 \times 3 = 22.4$ ms

In case of calculating $T_{bin}$, we start from the fact that 18 binaries are divided into 8 + 8 + 2, therefore the value considered is 3.

The total minimum communication period is therefore:

$T_{tot} = 32 \times 5 + 22.4 \times 5 = 272$ ms

**Amendment**

If we communicated only a single Slave station with a layout 5 × 8 registers and 5 × 18 binaries on ten rows of the definition table, the resulting period would be entirely identical.

### 4.1.1 Communication priorities

**Automatic Read priority**

DetStudio offers three reading priorities for automatic reading of values from Slave stations:

| Read priority | Communication period [ms] |
|---|---|
| Low | 5,000 |
| Normal | 1,000 |
| High | 200 |

By defining the priority, the programmer selects with what period the given row of the table is to be communicated. Obviously, it is not possible to use communication with priority **`High`** in definition rows of the table in the previous example because the interface would become completely overloaded with communication requests.

It applies that the NOS operating system goes through individual definition tables every 200 ms and if it discovers a row with automatic Read priority and is to communicate this row in the given 200 ms cycle, this row is placed in a communication request queue.

***Caution!***
*Communication requests are processed gradually until the communication request queue has been processed entirely. That means that if at one moment two rows with* ***`High`*** *priority and other 20 rows with* ***`Low`*** *priority are marked and this entire communication takes for example 600 ms, the aforementioned two rows with* ***`High`*** *priority shall be communicated repeatedly* ***only after*** *communication of all marked rows has finished. Consequently, the selected priority* ***`High`*** *is not followed.*
*If communication period 200 ms must be kept for all rows with* ***`High`*** *priority under any circumstances, all other communications must be launched manually and in such amounts so that the communication period of the given amount of communication requests and rows with* ***`High`*** *does not exceed 200 ms.*

### Automatic Write priority

In case of automatic writing, there is no defined priority with a time period; the only available option is switching to priority **`Auto`**.

If this priority is selected, the assigned variable (even with the same value) is marked upon each writing and placed at the start of the communication request queue. Writing requests are therefore always communicated before reading requests.

Due to internal mechanisms for detection of writing into the assigned variable, the given variable may be used in the definition table with priority **`Auto`** only once. For example, if we require that values from multiple cells of a matrix variable serve for writing of various registers, or that values of integer variable bits serve for writing of various binaries, we can proceed on the basis of registers layout:

- If two writing registers or binaries are in a sequence one after another, define only one definition row and have a set value of the column **`Number`** to the corresponding value. An example of such a definition is available in the Help section of DetStudio called “PseDet – Creating control processes”, in chapter “Contents/Communication/Modbus – Device table editor” in the section “Notes”.
- If writing registers or binaries are not in a sequence, it is necessary to set communication priorities for the given definition rows manually. In order to detect a binary value change, we can use the module **`BinDiff`**.

### Manual communication priority

If automatic communication priority of definition rows does not permit a correct requested mode of communication with the Slave station, it is necessary to use manual communication priority. We set it by selecting the option **`--manual--`** in the requested communication priority column.

The following modules are used to launch manual communication:

- **`MdbmMark`** – marking a rather large number of definition rows for communication,
- **`MdbmRead`** – marking a specific definition row for reading,
- **`MdbmWrite`** – marking a specific definition row for writing,
- **`MdbmWrBeg`**, **`MdbmWrFin`** – marking a specific definition row for a so called safe writing.

These modules work with labels on a specific Device definition as well as with labels for a specific definition row, except for the module **`MdbmMark`**. More information on individual modules is

available in the Help section of DetStudio called “PseDet – Creating control processes”, in descriptions of individual modules.

Unlike in case of automatic communication priority, data points in manual communication are communicated immediately after execution of the given module. This way, we are able to achieve communication even faster than 200 ms.

### 4.1.2 Gathering communication frames

When calculating a minimum communication period, it is also necessary to consider automatic gathering of communication frames going in a sequence when using communication functions 1, 2, 3, 4, 15 and 16 (see chapter 2.1 “Supported MODBUS functions”).

**Example**

Let us have a definition table with two rows for loading two registers into two variables. If addresses of the given registers are not in a sequence, e.g. addresses 0 and 2, communication is executed in two frames. At speed 19,200 bps, the period is:

T = 2 × (20 + 1.5 × 1) = 43 ms

However, if register addresses are defined in a sequence, e.g. addresses 0 and 1, communication is executed in a single frame. At speed 19,200 bps, the period is:

T = 20 + 1.5 × 2 = 23 ms

If we go back to “**Amendment**” of the original example at the beginning of this chapter (**Example**), then in case all 5 × 8 registers and 5 × 18 binaries were defined in an uninterrupted sequence, e.g. i registers with addresses 0 to 7, 8 to 15, 16 to 23, 24 to 31 and 32 to 39, then registers and binaries would be communicated each in a single frame. At speed 19,200 bps, the period is:

T = (20 + 1.5 × 40) + (20 + 0.8 × 15) = 112 ms

If we need to communicate as fast as possible, we could use automatic Read priority **`High`** in this case.

## 4.2 Communication in the event of a connection failure

If communication with the Slave station is not available, or more specifically if there has been no response to the request, the following algorithm of communication with this station is launched:

1. The frame that received no response is repeated 2× more.
2. If there is still no response, subsequent communication requests are ignored for the period of 15 seconds. This is signalled by setting bit No. 4 of the parameter “Status” of the module **`MdbmReqSt`** (for description, see chapter 5.5 “Communication statuses”) to True.
3. After this period, the table of communication requests of the given Slave station is checked. If any request is found, attempt for this communication is made. The first items to be checked are writing requests. At the same time, bit No. 4 of the parameter “Status” of the module **`MdbmReqSt`** is set to False for the period of this communication.
4. If there is still no correct response, the current time of ignoring communication requests is prolonged by 2 seconds. The bit No. 4 of the parameter “Status” of the module **`MdbmReqSt`** is set to True again.
5. If it was a writing communication request, this request maintains its flag for communication after the delay time has elapsed. If it was a reading request, the flag for communication is cancelled.
6. After a new delay time elapses, the algorithm repeats from the point 3. Maximum time of ignoring communication requests is 30 s. That gives us a row of times 15 s, 17 s, 19 s, …, 29 s, 30 s, 30 s, ….

## 4.3 Connection failure detection in modules DMM-xxx

All output modules of the series **DMM-xxx** have a simple mechanism implemented to turn off outputs in the event of a physical disruption of the network. If the module receives no valid frame in the network in 10 seconds (for any module), it sets a secure state on all outputs. This behaviour is firmly fixed in **DMM-xxx** modules and cannot be changed.

### Secure states for various output types

| **Output type** | **Secure state** |
|---|---|
| Digital outputs | 0 V |
| Relay outputs | Open |
| Analogue outputs | 0 V / 0 mA |

In case the communication is disrupted and outputs are set to secure states, the outputs will be re-set to the required values once the communication is renewed. However, this only happens in a period equal to the period of communication with modules.

# 5 Control system as Master

We define communication using MODBUS protocol in the Master role in a set of three definitions:

- creating a communication definition of the protocol in the Master role,
- creating a definition of the Slave station,
- defining data points of the given Slave station for communication.

## 5.1 Communication definition

Creating a communication definition of MODBUS protocol in the Master role represents inserting a definition of the communication item **`ModbusMaster`** into the project. We do so in DetStudio in the Project window in the node "Project/Communication/Modbus". When calling the context menu for this item, we select the item **`Add Master`**.

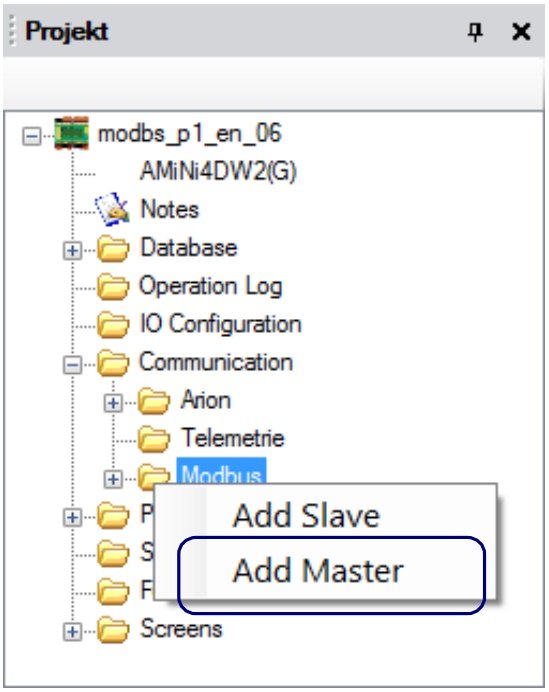



Fig. 3 – Item "Add Master" in the definition of "Modbus" communication

After we insert this definition, a communication node **`ModbusMaster0`** is created with the following values of properties:

- **`BaudRate`**: 19,200
- **`Mode`**: SerialLineRTU
- **`Parity`**: Even
- **`SerialPort`**: 0
- **`StopBit`**: One
- **`ToReceive`**: 30
- **`ToTransmit`**: 4

It applies that for communication via RS232 interface it is necessary to set the value of the property **`SerialPort`** to 0; whereas in case of RS485 interface it is necessary to set this property value to 1.

More information is available in the Help section of DetStudio called "PseDet – Creating control processes", in chapter "Contents/Communication/Modbus/Master – creating and setting general parameters".

## 5.2 Definition of a Slave station and data points for communication

In order to communicate with individual Slave stations, we must define addresses in MODBUS network and a list of communication points.

Individual Slave stations called *Device* define specific ModbusMasterX definitions in the Project window directly in the node “Project/Communication/Modbus”. After calling the context menu for this item, we select the item **`Add Device`**.

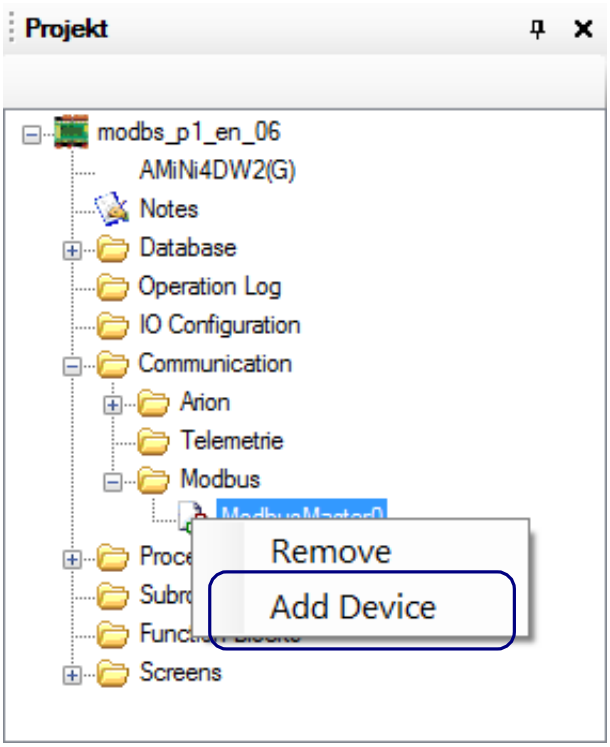



Fig. 4 – Item “Add Device” in the definition of the node “ModbusMasterx”

After we insert this definition, a communication node **`ModbusDevice0`** is created with default values of properties:

- **`Address`**: 1
- **`ByteOrder`**: 0-1-2-3 (Modbus default)
- **`ClientLabel`**: -1

Since 32-bit types (Long and Float) are not defined in the MODBUS protocol in any way, the manner of these extension register implementation (if any) is only up to the given device’s manufacturer. Due to the fact that the sequence of bytes in 16-bit words is defined as Big-Endian, in AMiT products this manner of coding has been also applied to the aforementioned 32-bit types.

If communication of 32-bit values results in values in variables significantly different than actual values in the Slave station, it is recommended we change the value of the property **`ByteOrder`**, usually to **`2-3-0-1`**.

Value of the property **`ClientLabel`** is used in case of manual communication with the Slave station and when determining communication statuses (see chapters 5.4 “Manual communication” and 5.5 “Communication statuses”).

After we define the node **`ModbusDeviceX`**, we are able to double-click it in the Project window and call up the definition table of communication data points of this Slave station.

In the following three chapters, we shall presume communication with **DMM-UI8DO8** in which there is a request for reading analogue input values and writing of digital outputs. The operation manual for this module states that analogue values are read using function 4 – reading input registers; writing into digital outputs is done through function 5 – setting one binary output (coil) or 15 – setting binary outputs (coils). Therefore, the definition table of the node **`ModbusDeviceX`** includes tabs “Input registers” and “Coils”.

We can define individual data points e.g. By dragging them from the Toolbox window. More information on definition of data points is available in the Help section of DetStudio called “PseDet – Creating control processes”, in chapter “Contents/Communication/Modbus - Device table editor”.

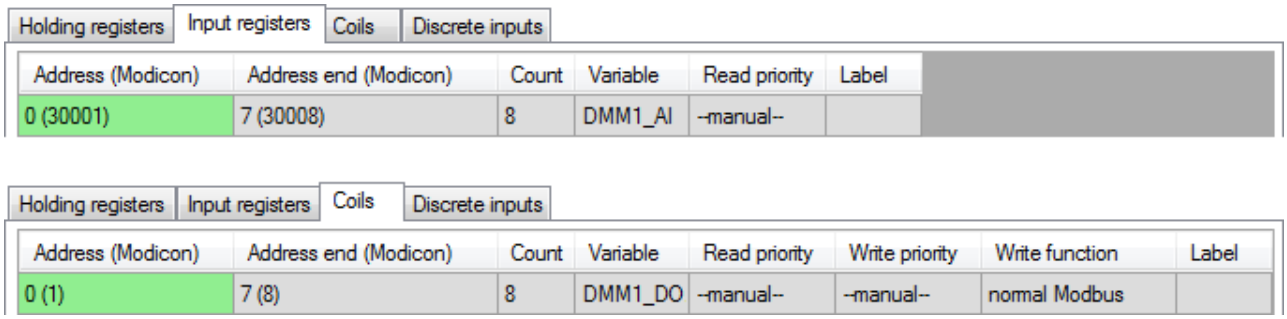



Fig. 5 – Basic definition of data points and assigned variables

## 5.3 Automatic communication

After we define data points, items **`--manual--`** are pre-set in the definition table columns “Read priority” and “Write priority”. For automatic communication, it is necessary to change their settings to one of the automatic priorities stated in chapter 4.1.1 “Communication priorities”.

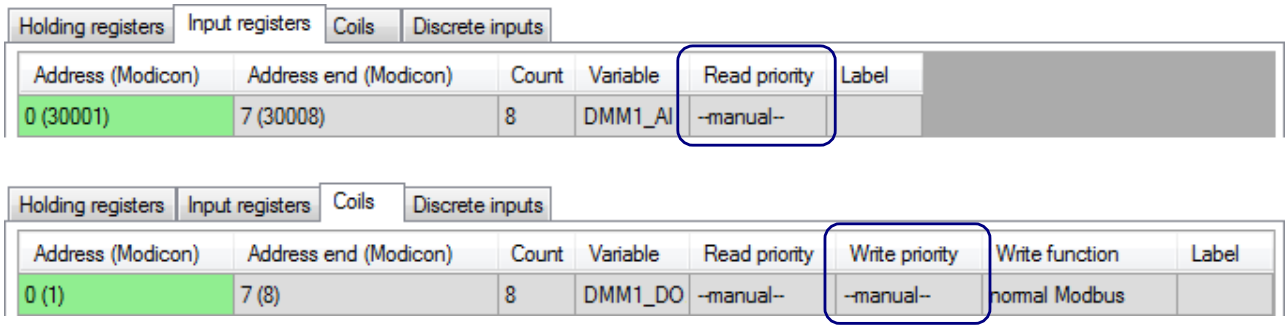



Fig. 6 – Definition of data points for automatic communication

***Recommendation***

If we wish to prevent writing event when the identical value is written into the variable, we can recommend the following code that only executes writing into a communication variable at the moment when the value of the auxiliary working variable changes:

```
If DMM1_DO != DMM1_DO_pr
  Let DMM1_DO_pr = DMM1_DO
EndIf
```

In application, this auxiliary working variable is to be used in the application code, whereas the communication variable will only be used in the definition table.

***Note***
*Due to internal algorithms for the use of automatic Write priority, it is not suitable to have both automatic Read priority and automatic Write priority on a single row. More information is available in the Help section of DetStudio called "PseDet – Creating control processes", in chapter "Contents/Communication/Modbus - Device table editor" in the section "Notes".*

## 5.4 Manual communication

For manual communication, it is necessary to define labels. There are two types of labels:

- Slave station definition label – property **`ClientLabel`**,
- definition row label – column "Label".

Value label values must not be negative. The label **`ClientLabel`** must be unique within the project, the label in the data points definition table must be unique within the given table.

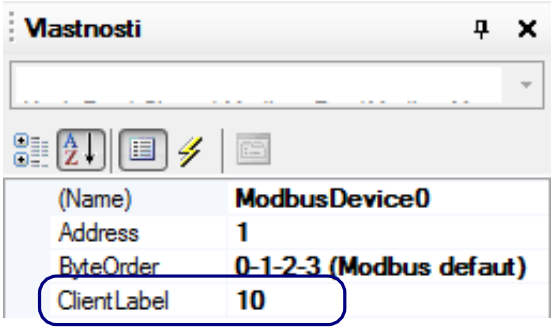



Fig. 7 – Label **`ClientLabel`** definition

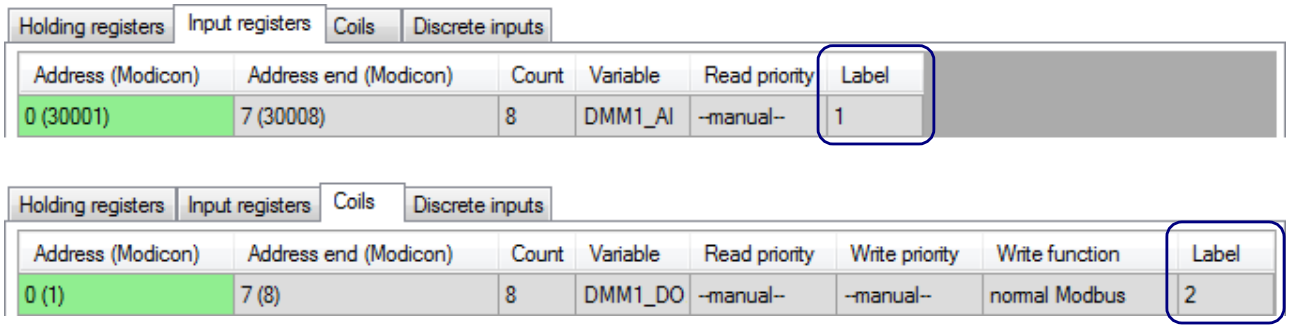



Fig. 8 – Data points label definition

As soon as the labels are defined, we are able to use modules **`Mdbm***`** mentioned in chapter 4.1.1 "Communication priorities", section "Manual communication priority".

We can use the following definition to mark the row with label 1 for reading of analogue inputs:

```
MdbmRead 10, 1, DMM1_AI_rslt
         |   |  └ Module execution result
         |   └ Definition row label
         └ ClientLabel
```

or:

```
MdbmMark 1, 4, 0, 8, DMM1_AI_rslt
         |  |  |  |  └ Module execution result
         |  |  |  └ Number of addresses for labelling
         |  |  └ Starting address for labelling
         |  └ Communication function definition
         └ ClientLabel
```

Using the module **MdbmMark** obviously does not require a definition for labels for specific rows. This module therefore allows us to batch-label a large number of definition rows of a single group of data points.

We can use the following definition to mark the row with label 2 for writing of digital outputs:

```
MdbmWrite 10, 2, DMM1_DO_rslt
          |   |  └ Module execution result
          |   └ Definition row label
          └ ClientLabel
```

or:

```
MdbmMark 1, 5, 0, 8, DMM1_DO_rslt
```

In this example, it is not necessary to consider the necessity to use a so called safe writing by means of modules **MdbmWrBeg** and **MdbmWrFin**, because the given definition row serves only for writing.

***Note***
*We can have both automatic and manual communication defined in a single definition row in the table. These two communications are not mutually exclusive.*

## 5.5 Communication statuses

We use the following modules to detect communication statuses:

- **MdbmCliSt** – detecting the communication status via specified communication interface
- **MdbmReqSt** – detecting the communication of a specific communication request

Module **MdbmReqSt** can be used for detection of failed communication with the Slave station using bit No. 4 (see the text under tables). In order to be able to detect a communication failure, we use the label of the most frequently communicated definition row of the table.

```
MdbmCliSt 10, DMM1_ClSt, DMM1_CS_rslt
          |   |          └ Module execution result
          |   └ Status of the client, or more spec. of the communication interface
          └ ClientLabel

MdbmReqSt 10, 1, DMM1_RqSt, DMM1_RS_rslt
          |   |  |          └ Module execution result
          |   |  └ Communication request status
          |   └ Definition row status
          └ ClientLabel
```

Using the module **MdbmReqSt** is definitely recommended for communication debugging, when we can acquire information about potential communication errors based on the value of the communication request status parameter.

This parameter's value gets various bit-coded values depending on the current status of the data point communication request entry and on the current communication status according to the following table.

| Bit | Description |
|---|---|
| 0 | Has value 1 if communication is currently in progress. |
| 1 | Has value 1 if the previous finished communication has finished successfully. |
| 2 | Has value 1 if the previous finished communication has finished in an error. |
| 4 | Has value 1 if the next attempt for communication is ignored. |
| 6 | Has value 1 if the request is marked for reading and awaits communication. |
| 7 | Has value 1 if the request is marked for writing and awaits communication. |
| 8 | Has value 1 if the request has been blocked by the module **MdbmWrBeg**. |
| 9 | Has value 1 if the request is repeatedly automatically marked for writing and awaits communication. |
| 10 | Has value 1 if the currently communicated request is for writing. |
| 11 | Has value 1 if the previous finished communication was for writing. |
| 12 to 15 | If the communication ended up in an error (bit 2 has value 1), these bits contain the communication error codes according to the following table. Otherwise, values of these bits are not defined. |

### Error codes in bits 12 to 15

| Error code | Description |
|---|---|
| 0 | Station responded negatively, with an unspecified error. |
| 1 | Station response: “Incorrect function”. |
| 2 | Station response: “Incorrect register/binary address”. |
| 3 | Station response: “Incorrect data value.” |
| 4 | Unknown unspecified error. |
| 5 | Station has not responded within the required period. |
| 6 | Transmission error (incorrect CRC, incorrect response length, etc.). |
| 7 | Connection error, usually in case of MODBUS TCP communication. |

With respect to the fact that bit No. 4 only signalizes ignoring of a subsequent communication, we can recommend using the module **RS** to signalize a failure in communication with the given Slave station:

```
RS DMM1_RS_rslt.4, DMM1_RS_rslt.1, DMM1_Problem.0
```

## 5.6 Example of a control system parametrization as Master

Let us have an application in which one control system AMiNi4W2 is to serve as a Slave station for the second control system AMiNi4W2. Slave station’s register layout is described in chapter 6.4 “Example of a control system parametrization as Slave”.

We also know the holding registers’ layout:

- address 0 – DI,
- addresses 1 to 8 – AI_Integer,
- addresses 10 to 25 – AI_Float,
- address 100 – DO,
- addresses 101 to 104 – AO.

First, variables are created to be assigned to the given definition rows:

- **AMiNi_DI** – type I,
- **AMiNi_AI** – type MI, dimension [1×8],
- **AMiNi_AI_F** – type MF, dimension [1×8],
- **AMiNi_DO** – type I,
- **AMiNi_AO** – type MI, dimension [1×4].

The next step is to create a definition node **ModbusMaster0** and within it create a definition **ModbusDevice0**.

In the definition of the table **ModbusDevice0**, there are 5 rows defined in the tab "Holding registers". We select priorities in definition rows for example as follows:

- address 0 – automatic reading with priority **Normal**,
- addresses 1 to 8 – automatic reading with priority **Low**,
- addresses 10 to 25 – automatic reading with priority **Low**,
- address 100 – manual writing,
- addresses 101 to 104 – automatic writing.

Because manual writing is defined and the request for detection of connection status with the Slave station, the definition of **ModbusDevice0** includes a parameter **ClientLabel,** e.g. to value 10. In order to detect the connection status to the Slave station, a label is defined in the row of the register with address 0 and for manual execution of communication another label is defined in the row of the register with address 100.

Due to the fact that AMiT product support communication frames 6 as well as 16, the column "Writing function" maintains the option **normal Modbus**.

The resulting table's appearance is illustrated in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (40001) | 0 (40001) | 1 | AMiNi_DI | Normal | --manual-- | normal Modbus | 1 |
| 1 (40002) | 8 (40009) | 8 | AMiNi_AI | Low | --manual-- | normal Modbus | |
| 10 (40011) | 25 (40026) | 16 | AMiNi_AI_F | Low | --manual-- | normal Modbus | |
| 100 (40101) | 100 (40101) | 1 | AMiNi_DO | --manual-- | --manual-- | normal Modbus | 2 |
| 101 (40102) | 104 (40105) | 4 | AMiNi_AO | --manual-- | Auto | normal Modbus | |

Fig. 9 – Basic definition of data points and assigned variables

The last step is defining modules **MdbmReqSt** and **MdbmWrite**. In order to prevent excessive communications of register 100, we use the algorithm described in chapter 5.4 "Manual communication".

The resulting code in the periodic process is written as follows:

```
MdbmReqSt 10, 1, AMiNi_RqSt, AMiNi_RS_rs

If not AMiNi_RqSt.4
  If AMiNi_DO != AMiNi_DO_pr
    Let AMiNi_DO_pr = AMiNi_DO
    MdbmWrite 10, 2, AMiNi_DO_rs
  EndIf
EndIf
```

***Note***

*The application code should subsequently deal with conversion of values of analogue quantities, which is not considered in this example.*

The stated algorithm is included in annex ap0008_en_xx.zip. It is a sample project called "modbs_p1_en_xx.dso" created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu "Tools/Change Station".

# 6 Control system as Slave

We define communication using MODBUS protocol in the Slave role in two definitions:

- creating a communication definition of the protocol in the Slave role,
- definition of data points for communication.

## 6.1 Communication definition

Creating a communication definition of MODBUS protocol in the Slave role represents inserting a definition of the communication item **ModbusSlave** into the project. We do so in DetStudio in the Project window in the node "Project/Communication/Modbus". After calling the context menu for this item, we select the item **Add Slave**.

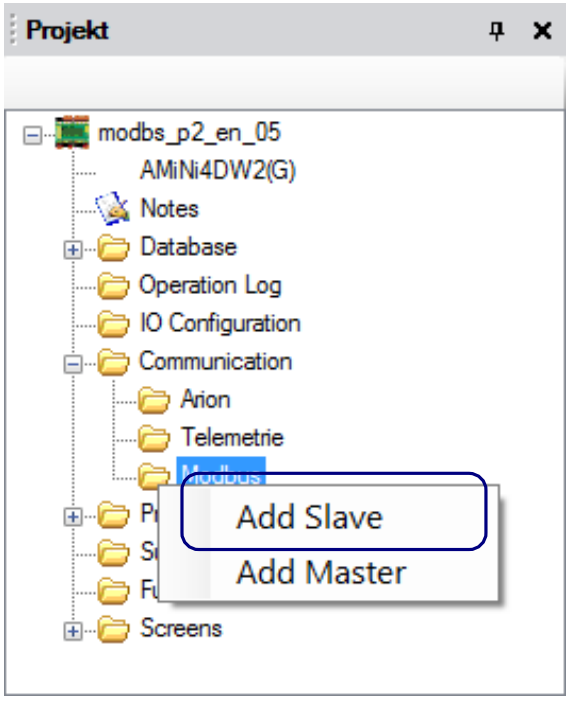



Fig. 10 – Item "Add Slave" in "Modbus" communication definition

After we insert this definition, a communication node **Modbus0** is created with default values of properties:

- **Address**: 1
- **BaudRate**: 19,200
- **DataBits**: 8
- **LastError**: NONE
- **Mode**: SerialLineRTU
- **Parity**: Even
- **SerialPort**: 0
- **StopBit**: One

It applies that for communication via RS232 interface it is necessary to set the value of the property **SerialPort** to 0; whereas in case of RS485 interface it is necessary to set this property value to 1.

More information is available in the Help section of DetStudio called "PseDet – Creating control processes", in chapter "Contents/Communication/Modbus/Slave - creating and setting general parameters".

## 6.2 Definition of data points for communication

After we define the node **ModbusX**, we are able to double-click it in the Project window and call up the definition table of communication data points.

In this table, control system variables are defined in individual tabs of data point groups (Holding registers, Input registers, Coils and Discrete inputs); these variables are to be available under selected addresses.

We can define individual data points e.g. By dragging them from the Toolbox window. More information on definition of data points is available in the Help section of DetStudio called “PseDet – Creating control processes”, in chapter “Contents/Communication/Modbus/Slave - table editor”.

## 6.3 Communication statuses

Communication status on part of the Master is available after we assign a variable to the property **LastError** in definition of the **ModbusX** communication. The expected variable type is I.

We recommend using this property especially in communication debugging when its value is able to provide us with information on potential communication error. The assign variable is to take values according to the following table:

| Error code | Description |
|---|---|
| 0 | No error. |
| 1 | NOS version too low. Requires version at least 3.25 or higher. |
| 2 | System timer allocation error. |
| 3 | Communication port allocation error. |
| 4 | Last frame received had an incorrect check sum. |
| 5 | Last frame received had an incorrect length. |
| 6 | Last frame received included a request for an unmapped address. |
| 7 | Last frame received included a request for an unsupported function. |
| 8 | Last frame received required more data than is available for a response frame. |
| 9 | Last frame received included incorrect data (function 6 ON, OFF). |
| 10 | Too wide space between incoming characters. |
| 11 | Error in ASCII reception:<br>– frame too long,<br>– unexpected character (only textual hexa digits must be inside the frame),<br>– no LF followed after CR. |
| 12 | The module has not been launched yet (no parameter evaluation and communication port allocation). |

## 6.4 Example of a control system parametrization as Slave

Let us assume we have a request for creation of an application that makes the control system AMiNi4W2 work as a module of remote inputs and outputs communicating via MODBUS RTU protocol. At the same time, the application is to be universal enough to be able to communicate analogue values in a decimal or integer form.

1. Variables are created, according to the following table.

| Variable | Type | Comment |
|---|---|---|
| DI | I | Digital inputs. |
| AI_I | MI[1,8] | Integer values of analogue inputs. |
| AI_F | MF[1,8] | Float values of analogue inputs. |
| DO | I | Digital outputs. |
| AO_I | MI[1,4] | Integer values of analogue outputs. |
| AO_F | MF[1,4] | Float values of analogue outputs. |

| Variable | Type | Comment |
|---|---|---|
| tempF | F | Auxiliary Float variable. |
| Mdbs_Err | I | Code of the previous communication error. |

2. Addresses and data point types are selected that are to represent the given variables according to the following table.

| Variable | Address | Data point type |
|---|---|---|
| DI | 0 | Holding register |
| AI_I | 1 to 8 | Holding register |
| AI_F | 10 to 25 | Holding register |
| DO | 100 | Holding register |
| AO_I | 101 to 104 | Holding register |
| AO_F | 110 to 117 | Holding register |

3. Based on the previous table, definition rows are defined in the node **Modbus0**.

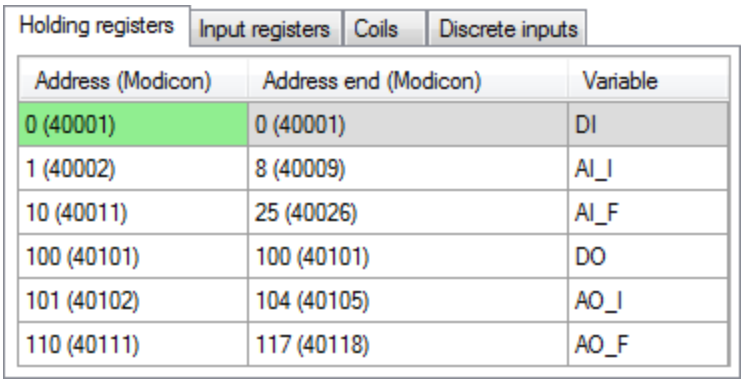



Fig. 11 – Definition of MODBUS Slave data points and assigned variables

4. Creating an algorithm in a periodic process that is to load inputs and write outputs based on variable values. The code may look as follows:

```
// ------------------------------------    DI    ------------------------------------

DigIn #0, DI, 0x0000

// ------------------------------------    AI    ------------------------------------

// AI0 and AI1 as Ni1000 / 6,180 ppm
// Temperature 12.45 °C corresponds to value 1245 in Int register
Ni1000 #Ni10001_0, AI_F[0,0], 6180
Let AI_I[0,0] = Int(AI_F[0,0] * 100)

Ni1000 #Ni10001_1, AI_F[0,1], 6180
Let AI_I[0,1] = Int(AI_F[0,1] * 100)

// AI2 and AI3 as Pt1000 / 3,850 ppm
// Temperature 12.45 °C corresponds to value 1245 in Int register
Pt1000 #Ni10001_2, AI_F[0,2], 3850
Let AI_I[0,2] = Int(AI_F[0,2] * 100)

Pt1000 #Ni10001_3, AI_F[0,3], 3850
Let AI_I[0,3] = Int(AI_F[0,3] * 100)

// AI4 and AI5 for measurement of voltage 0 to 10 V
// Value 3.678 V corresponds to value 3678 in Int register
AnIn #AI00_4, AI_F[0,4], 10.000, 0.000, 10.000, 0.000, 10.000
```

```
Let AI_I[0,4] = Int(AI_F[0,4] * 1000)

AnIn #AI00_5, AI_F[0,5], 10.000, 0.000, 10.000, 0.000, 10.000
Let AI_I[0,5] = Int(AI_F[0,5] * 1000)

// AI6 and AI7 for measurement of current 0(4) to 20 mA
// Value 15.345 mA corresponds to value 15345 in Int register
AnIn #AI00_6, AI_F[0,6], 20.000, 0.000, 20.000, 0.000, 20.000
Let AI_I[0,6] = Int(AI_F[0,6] * 1000)

AnIn #AI00_7, AI_F[0,7], 20.000, 0.000, 20.000, 0.000, 20.000
Let AI_I[0,7] = Int(AI_F[0,7] * 1000)

// -----------------------------------    DO    -----------------------------------

DigOut DO, #0, 0x0000

// -----------------------------------    AO    -----------------------------------

// AO0 to AO3 with output 0 to 10 V
// Setpoint value 2.456 V must be written in Int register as value 2456
// We either use Float registers 110-111 to 116-117 or Int registers 101 to 104
Let tempF = If(AO_F[0,0] == 0, AO_I[0,0] / 1000, AO_F[0,0])
AnOut #AO00_0, tempF, 10.000, 0.000, 10.000, 0.000, 10.000

Let tempF = If(AO_F[0,1] == 0, AO_I[0,1] / 1000, AO_F[0,1])
AnOut #AO00_1, tempF, 10.000, 0.000, 10.000, 0.000, 10.000

Let tempF = If(AO_F[0,2] == 0, AO_I[0,2] / 1000, AO_F[0,2])
AnOut #AO00_2, tempF, 10.000, 0.000, 10.000, 0.000, 10.000

Let tempF = If(AO_F[0,3] == 0, AO_I[0,3] / 1000, AO_F[0,3])
AnOut #AO00_3, tempF, 10.000, 0.000, 10.000, 0.000, 10.000
```

The stated algorithm is included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p2_en_xx.dso” created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

# 7 Appendix A

## 7.1 Compatibility with communication initialization via modules

### 7.1.1 MODBUS Master

We can also initialize MODBUS Master RTU communication by means of modules **`RmtDef`**, **`RmtAct`** and **`MODBS_R`** that are typically placed into the initialization and periodic process.

However, in terms of internal functionality, it is a different communication than a table definition. For this reason, it **is not possible** to combine both communication definitions on the same COM interface.

If each communication initialization is defined on a different COM interface, both communications are fully functional.

### 7.1.2 MODBUS Slave

We can also initialize MODBUS Master RTU communication by means of modules **`MODBS_Var`** and **`MODBS_RSl`** that are typically placed into the initialization process.

However, in terms of internal functionality, it is an identical communication to a table definition. For this reason, it **is possible** to combine both communication definitions on the same COM interface.

Find out more in the Help section of DetStudio called “PseDet – Creating control processes”, in chapter “Contents/Communication/Modbus” in the section “Backward compatibility”.

# 8 Appendix B

## 8.1 Programme operation DMM-xxx

Data point addresses in individual modules are always determined by the number of the given input/output of the module **DMM-xxx**.

### 8.1.1 DMM-DI24

The module **DMM-DI24** provides two modes of operation (mutually independent).

- Working with classic digital inputs (values True / False) – option to read one or multiple inputs.
- Working with counter inputs (counting impulses up to frequency 25 Hz) – option to read and write one or multiple counters.

The definition table of full communication with the module may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (40001) | 23 (40024) | 24 | M_DI24_cnt[0,0] | Low | --manual-- | normal Modbus | 2 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (10001) | 23 (10024) | 24 | M_DI24_DI.0 | Normal | 1 |

Fig. 12 – Example of communication definition with DMM-DI24

According to the given definition, input statuses are saved into bits 0 to 23 of the variable **`M_DI24_DI`** type Long every 1,000 ms. Counter values are saved into cells of the variable **`M_DI24_cnt`** type Matrix Int (e.g. dimension [1×24]) every 5,000 ms.

In order to detect the status of communication with the module, we use the module **`MdbmReqSt`** linked to the label of the row for reading input statuses. In our example, we set the property **`ClientLabel`** to value 10.

```
MdbmReqSt 10, 1, M_DI24_Stat, NONE
```

#### Working with classic digital inputs

We can use the communication variable **`M_DI24_DI`** directly in the application algorithm; the variable works with Discrete inputs at addresses 0 to 23.

#### Working with counter inputs

The module **DMM-DI24** allows us to use function of counting previous impulses on any of its inputs. This partially solves problems with short impulse detection. Counter values are available in Holding Registers at addresses 0 to 23.

However, we need to take into account certain limitations when using this function:

- Maximum counted value possible is 32767 (number 15 bit). After another pulse is added, the counting starts again from zero. It is necessary to handle the internal counter overflow in terms of programming.
- Maximum frequency of incoming impulses is 25 Hz. With a higher frequency, there is no guarantee that all incoming impulses are recorded.
- The module's internal counter is reset when the power-supply voltage is disconnected; we can also reset it in the programme.

Based on the table definition, periodic loading of counter values is handled. If it is desirable to reset counter values manually during the counting, we can use the following algorithm using modules for safe writing.

```
If @DI24_cntRst and not M_DI24_Stat.4
     Let @DI24_cntRst = false
     MdbmWrBeg 10, 2, NONE
     For i, 0.000, 23.000, 1.000
          Let M_DI24_cnt[0,i] = 0
     EndFor
     MdbmWrFin 10, 2, NONE
EndIf
```

In this case, modules for safe writing are entirely necessary, since it is not possible to simply determine the exact moment of when values are loaded from the Slave station. If this moment occurs between the execution of the row with the module **EndFor** and the execution of the subsequent module **MdbmWrite**, only those identical values that have just been loaded would be written into the module.

***Caution***
*It is necessary to pay attention to counter overflow! Counter range is 0 to 32767. Therefore, a counter generates a row 0, 1, 2, …, 32766, 32767, 0, 1, 2, …. Therefore, it will be necessary to count the number of overflows "**p**" of the counter, and the resulting sum of pulses will therefore be equal to: **sum = p*32768 + counter**.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called "modbs_p3_en_xx.dso" created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu "Tools/Change Station".

### 8.1.2 DMM-DO18

The module **DMM-DO18** has two operation modes.

- Working with classic digital inputs (values True / False) – option to read and write one or multiple outputs.
- Working with outputs in PWM mode – option to read and write one or multiple PWM output parameters.

The definition table of full communication with the module may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (40001) | 18 (40019) | 19 | M_DO18_PWM[0,0] | --manual-- | Auto | normal Modbus | 2 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (1) | 17 (18) | 18 | M_DO18_DO.0 | --manual-- | Auto | normal Modbus | 1 |

Fig. 13 – Example of communication definition with DMM-DO18

According to the given definition, output statuses change according to bits 0 to 17 of the variable **M_DO18_DO** type Long after every entry written into this variable. PWM output parameters will change according to cells of the variable **M_DO18_PWM** type Matrix Int (e.g. dimension [1×19]) after every entry written into this variable.

In order to detect the status of communication with the module, we use the module **MdbmReqSt** linked to the label of the row for writing outputs. In our example, we set the property **ClientLabel** to value 20.

```
MdbmReqSt 20, 1, M_DO18_St_D, NONE
```

or

```
MdbmReqSt 20, 2, M_DO18_St_P, NONE
```

depending on which communication occurs more frequently.

### Working with classic digital outputs

We can use the communication variable **M_DO18_DO** directly in the application algorithm; the variable works with Coils at addresses 0 to 17.

In order to prevent unnecessary excess communications of identical values, it is recommended to use the code stated in chapter 5.3 “Automatic communication”, section “Recommendation”.

### Working with outputs in PWM mode

In the PWM mode, we are able to change duty cycle in individual outputs of the module **DMM-DO18**. Duty cycle is available in Holding Registers at addresses 0 to 17. PWM period is entered at address 18 and it is common for all digital outputs.

The values are interpreted in a way that **0 to 32767** corresponds to **0 % to 100 %** range of the duty cycle. PWM period value is interpreted in a way that **0 to 32767** corresponds to **0 s to 100 s**.

In order to convert values, we can use the following algorithm and modules **VarWStat** and **Interpol**.

```
VarWStat DO18_P_per, @DO18_P_p_w, 0
If @DO18_P_p_w
     Interpol DO18_P_per, tmpF2, Params100_32
     Let M_DO18_PWM[0,18] = Int(tmpF2)
EndIf
```

***Note***
*The real output on the DO module **DMM-DO18** is a logical sum of the value of DO and PWM.*

### Retroactive reading of outputs and PWM parameters

While writing parameters is automatic after a value is written into a defined variable thanks to the priority **Auto**, reading has a defined priority **--manual--**. That means that we need to use modules **MdbmRead** or **MdbmMark** for non-periodic value loading.

***Caution***
*If the module **DMM-DO18** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **Low**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p4_en_xx.dso” created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

### 8.1.3 DMM-RDO12

The module **DMM-RDO12** can be operated in the mode of working with classic digital inputs (values True / False) – option to read and write one or multiple outputs.

The definition table of full communication with the module may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (1) | 11 (12) | 12 | M_RDO12_DO.0 | --manual-- | Auto | normal Modbus | 1 |

Fig. 14 – Example of communication definition with DMM-RDO12

According to the given definition, output statuses change according to bits 0 to 11 of the variable **`M_RDO12_DO`** type Int after every entry written into this variable.

In order to detect the status of communication with the module, we use the module **`MdbmReqSt`** linked to the label of the row for output status. In our example, we set the property **`ClientLabel`** to value 30.

```
MdbmReqSt 30, 1, M_RDO12_Stat, NONE
```

#### Working with digital outputs

We can use the communication variable **`M_RDO12_DO`** directly in the application algorithm; the variable works with Coils at addresses 0 to 11.

In order to prevent unnecessary excess communications of identical values, it is recommended to use the code stated in chapter 5.3 “Automatic communication”, section “Recommendation”.

***Caution***
*If the module **DMM-RDO12** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **`Low`**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

#### Retroactive reading of outputs

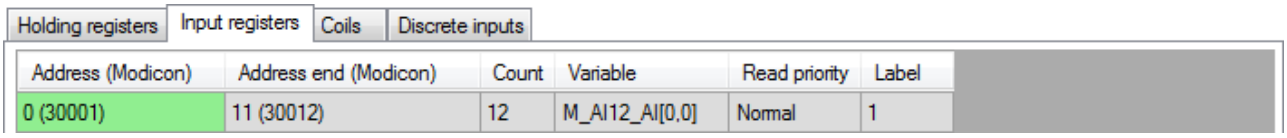
While writing parameters is automatic after a value is written into a defined variable thanks to the priority **`Auto`**, reading has a defined priority **`--manual--`**. That means that we need to use modules **`MdbmRead`** or **`MdbmMark`** for non-periodic value loading.

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p5_en_xx.dso” created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

### 8.1.4 DMM-AI12

The module **DMM-AI12** allows us to read one or multiple inputs.

The definition table of full communication with the module may look like in the following image.



Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (30001) | 11 (30012) | 12 | M_AI12_AI[0,0] | Normal | 1 |

Fig. 15 – Example of communication definition with DMM-AI12

Loaded input values are saved into cells 0 to 11 of the variable **M_AI12_AI** type Matrix Int (e.g. dimension [1×12]) every 1,000 ms.

In order to detect the status of communication with the module, we use the module **MdbmReqSt** linked to the label of the row for reading input values. In our example, we set the property **ClientLabel** to value 40.

```
MdbmReqSt 40, 1, M_AI12_Stat, NONE
```

### Working with analogue inputs

The values of inputs in Input registers at addresses 0 to 11 are interpreted in a way that **0 to 32767** corresponds to **0 % to 100 %** range of the input range.

In order to convert the values to the range 0 V to 5 V, we use e.g. the following algorithm using the module **Interpol**.

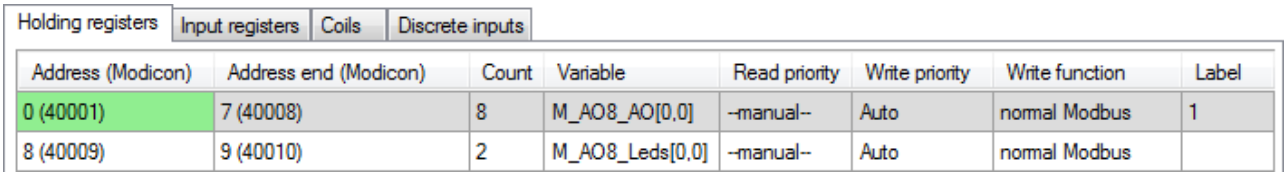
```
Let AI12_f[0,0] = M_AI12_AI[0,0]
Interpol AI12_f[0,0], AI12_AI[0,0], Range0_5V
```

Stated algorithms (including the sample of conversion of a measured value to 0 V to 10 V, 0 mA to 20 mA and conversion to temperature measured using sensors Ni1000 and Pt1000) are included in the annex ap0008_en_xx.zip. It is a project called mpbus_p6_en_xx.dso created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

## 8.1.5 DMM-AO8x

The module **DMM-AO8x** allows us to read and write one or multiple outputs and parameters of LED behaviour.

The definition table of full communication with the module may look like in the following image.



Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (40001) | 7 (40008) | 8 | M_AO8_AO[0,0] | --manual-- | Auto | normal Modbus | 1 |
| 8 (40009) | 9 (40010) | 2 | M_AO8_Leds[0,0] | --manual-- | Auto | normal Modbus | |

Fig. 16 – Example of communication definition with DMM-AO8x

According to the given definition, output values change according to cells 0 to 7 of the variable **M_AO8_AO** type Matrix Int (e.g. dimension [1×8]) after every entry written into this variable. LED behaviour parameters will change according to cells of the variable **M_AO8_Leds** type Matrix Int (e.g. dimension [1×2]) after every entry written into this variable.

In order to detect the status of communication with the module, we use the module **MdbmReqSt** linked to the label of the row for output values. In our example, we set the property **ClientLabel** to value 50.

```
MdbmReqSt 50, 1, M_AO8_Stat, NONE
```

### Working with analogue outputs

Values for analogue outputs are interpreted in a way that the range **0 to 32767** corresponds to **0 % to 100 %** of the analogue output range. The analogue output range is available in Holding Registers at addresses 0 to 7.

In order to convert values, we can use the following algorithm and modules **VarWStat** and **Interpol**.

```
VarWStat AO8x_AO, @AO8x_AO_w, 0
If @AO8x_AO_w
     For i, 0.000, 7.000, 1.000
          Let tmpF1 = AO8x_AO[0,i]
          Interpol tmpF1, tmpF2, Range_AO
          Let M_AO8_AO[0,i] = Int(tmpF2)
     EndFor
EndIf
```

### Working with LED

In the module **DMM-AO8x**, we are able to programme behaviour of LEDs on the module by means of MODBUS that correspond to individual analogue outputs. This behaviour can be set using two Holding Registers at addresses 8 and 9. LED behaviour is then as follows:

1. Value on the outputs is higher than the value of the register at address 9 – LED is blinking.
2. Value on the outputs is higher than the value of the register at address 8 and lower than the value of the register at the address 9 – LED is on.
3. Value on outputs is lower than the value of the register at address 8 – LED is off.

The second condition is processed only if the first condition is not met. If neither the first nor the second condition are met, the relevant LEDx is off.
Both limits can be identical or even zero. It is therefore not necessary to enter them. In zero output, LEDx is off; for non-zero output, LEDx is on. LEDx does not get into the blinking mode when the limits are identical.
If the value of register 9 is selected higher than the value of register 8, and if value on the outputs is lower than the value of the register on position 8, LEDx is off.
If the value on the outputs is higher than the value of register 8 and lower than the value of register 9, LEDx is on.
If the value on the outputs is higher than the value of register on position 9, LEDx is blinking.

It also applies in this case that the range **0 to 32767** corresponds to **0 % to 100 %** of the analogue output range.

### Retroactive reading of outputs and LED behaviour parameters

While writing values and parameters is automatic after a value is written into a defined variable thanks to the priority **Auto**, reading has a defined priority **--manual--**. That means that we need to use modules **MdbmRead** or **MdbmMark** for non-periodic value loading.

***Caution***
*If the module **DMM-AO8U** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **Low**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a project called mpbus_p7_en_xx.dso created in DetStudio development environment. This project has been

created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

### 8.1.6 DMM-PDO6NI6

The module **DMM-PDO6NI6** allows us to read one or multiple RTD inputs. The module provides two modes for work with digital outputs.

- Working with classic digital inputs (values True / False) – option to read and write one or multiple outputs.
- Working with outputs in PWM mode – option to read and write one or multiple PWM output parameters.

The definition table of full communication with the module may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (40001) | 6 (40007) | 7 | M_PDONi_PWM[0,0] | --manual-- | Auto | normal Modbus | 3 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (30001) | 5 (30006) | 6 | M_PDONi_Ni[0,0] | Normal | 1 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (1) | 5 (6) | 6 | M_PDONi_DO.0 | --manual-- | Auto | normal Modbus | 2 |

Fig. 17 – Example of communication definition with DMM-PDO6NI6

According to the given definition, PWM output parameters change according to cells of the variable **M_PDONi_PWM** type Matrix Int (e.g. dimension [1×7]) after every entry written into this variable. Output statuses change according to bits 0 to 5 of the variable **M_PDONi_DO** type Int after every entry written into this variable. Loaded RTD input values are saved into cells 0 to 5 of the variable **M_PDONi_Ni** type Matrix Int (e.g. dimension [1×6]) every 1,000 ms.

In order to detect the status of communication with the module, we use the module **MdbmReqSt** linked to the label of the row for reading RTD input values. In our example, we set the property **ClientLabel** to value 60.

```
MdbmReqSt 60, 1, M_PDONi_Stat, NONE
```

#### Working with RTD inputs

The values of inputs in Input registers at addresses 0 to 5 are interpreted in a way that the range **0 to 32767** corresponds to **0 % to 100 %** of the input range.

In order to convert the values to the range 0 V to 5 V and subsequently to the measured temperature value, we use e.g. the following algorithm using the module **Interpol**.

```
For i, 0.000, 5.000, 1.000
     Let tmpF1 = M_PDONi_Ni[0,i]
     Interpol tmpF1, tmpF2, Range0_5V
     Let PDONi_AI[0,i] = tmpF2
EndFor

Ni1000U2T PDONi_AI[0,0], PDONi_T[0,0], 6180, 15.000, 3920.000
Ni1000U2T PDONi_AI[0,1], PDONi_T[0,1], 6180, 15.000, 3920.000
…
```

### Working with classic digital outputs

We can use the communication variable **M_PDONi_DO** directly in the application algorithm; the variable works with Coils at addresses 0 to 5.

In order to prevent unnecessary excess communications of identical values, it is recommended to use the code stated in chapter 5.3 “Automatic communication”, section “Recommendation”.

### Working with outputs in PWM mode

In the PWM mode, we are able to change duty cycle in individual outputs. Duty cycle is available in Holding Registers at addresses 0 to 5. PWM period is entered at address 6 and it is common for all digital outputs.

The values are interpreted in a way that **0 to 32767** corresponds to **0 % to 100 %** range of the duty cycle. PWM period value is interpreted in a way that **0 to 32767** corresponds to **0 s to 100 s**.

In order to convert values, we can use the following algorithm and modules **VarWStat** and **Interpol**.

```
VarWStat PDONi_P_per, @PDONi_P_p_w, 0
If @PDONi_P_p_w
     Interpol PDONi_P_per, tmpF2, Params100_32
     Let M_PDONi_PWM[0,6] = Int(tmpF2)
EndIf
```

***Note***
*The real output on the DO module **DMM-PDO6NI6** is a logical sum of the value of DO and PWM.*

### Retroactive reading of outputs and PWM parameters

While writing parameters is automatic after a value is written into a defined variable thanks to the priority **Auto**, reading has a defined priority **--manual--**. That means that we need to use modules **MdbmRead** or **MdbmMark** for non-periodic value loading.

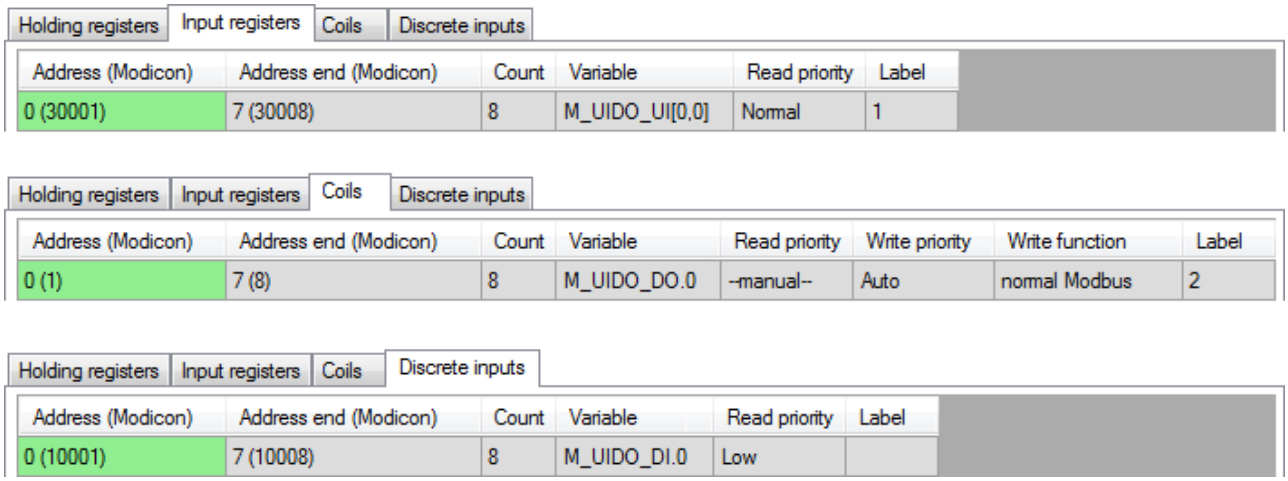
***Caution***
*If the module **DMM-PDO6NI6** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **Low**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a project called mpbus_p8_en_xx.dso created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

## 8.1.7 DMM-UI8DO8

The module **DMM-UI8DO8** allows us to read one or multiple analogue inputs in the form of analogue values and binary states. We can read and write digital outputs one by one or by multiple outputs.

The definition table of full communication with the module may look like in the following image.



Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (30001) | 7 (30008) | 8 | M_UIDO_UI[0,0] | Normal | 1 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (1) | 7 (8) | 8 | M_UIDO_DO.0 | --manual-- | Auto | normal Modbus | 2 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (10001) | 7 (10008) | 8 | M_UIDO_DI.0 | Low | |

Fig. 18 – Example of communication definition with DMM-UI8DO8

According to the given definition, loaded universal-input values are saved into cells 0 to 7 of the variable **M_UIDO_UI** type Matrix Int (e.g. dimension [1×8]) in the form of analogue values every 1,000 ms. Loaded input values are saved into bits 0 to 7 of the variable **M_UIDO_DI** type Int in the form of binary values every 5,000 ms. Output statuses change according to bits 0 to 7 of the variable **M_UIDO_DO** type Int after every entry written into this variable.

In order to detect the status of communication with the module, we use the module **MdbmReqSt** linked to the label of the row for reading analogue input values. In our example, we set the property **ClientLabel** to value 70.

```
MdbmReqSt 70, 1, M_UIDO_Stat, NONE
```

### Working with analogue inputs

The values of inputs in Input registers at addresses 0 to 7 are interpreted in a way that the range **0 to 32767** corresponds to **0 % to 100 %** of the input range.

In order to convert the values to the range 0 V to 5 V, we use the following algorithm using the module **Interpol**.

```
Let UIDO_f[0,0] = M_UIDO_AI[0,0]
Interpol UIDO_f[0,0], UIDO_AI[0,0], Range0_5V
```

### Working with digital inputs

We can use the communication variable **M_UIDO_DI** directly in the application algorithm; the variable works with Discrete inputs at addresses 0 to 7.

### Working with digital outputs

We can use the communication variable **M_UIDO_DO** directly in the application algorithm; the variable works with Coils at addresses 0 to 7.

In order to prevent unnecessary excess communications of identical values, it is recommended to use the code stated in chapter 5.3 “Automatic communication”, section “Recommendation”.

### Retroactive reading of outputs

While writing states is automatic after a value is written into a defined variable thanks to the priority **`Auto`**, reading has a defined priority **`--manual--`**. That means that we need to use modules **`MdbmRead`** or **`MdbmMark`** for non-periodic value loading.

***Caution***

*If the module **DMM-UI8DO8** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **`Low`**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a project called mpbus_p9_en_xx.dso created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

## 8.1.8 DMM-UI8RDO8

The module **DMM-UI8RDO8** allows us to read one or multiple analogue inputs in the form of analogue values and binary states. We can read and write digital outputs one by one or by multiple outputs.

The definition table of full communication with the module may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (30001) | 7 (30008) | 8 | M_UIRDO_UI[0,0] | Normal | 1 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (1) | 7 (8) | 8 | M_UIRDO_DO.0 | --manual-- | Auto | normal Modbus | 2 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (10001) | 7 (10008) | 8 | M_UIRDO_DI.0 | Low | |

Fig. 19 – Example of communication definition with DMM-UI8RDO8

According to the given definition, loaded universal-input values are saved into cells 0 to 7 of the variable **`M_UIRDO_UI`** type Matrix Int (e.g. dimension [1×8]) in the form of analogue values every 1,000 ms. Loaded input values are saved into bits 0 to 7 of the variable **`M_UIRDO_DI`** type Int in the form of binary values every 5,000 ms. Output statuses change according to bits 0 to 7 of the variable **`M_UIRDO_DO`** type Int after every entry written into this variable.

In order to detect the status of communication with the module, we use the module **`MdbmReqSt`** linked to the label of the row for reading analogue input values. In our example, we set the property **`ClientLabel`** to value 80.

```
MdbmReqSt 80, 1, M_UIRDO_Stat, NONE
```

### Working with analogue inputs

The values of inputs in Input registers at addresses 0 to 7 are interpreted in a way that the range **0 to 32767** corresponds to **0 % to 100 %** of the input range.

In order to convert the values to the range 0 V to 5 V, we use the following algorithm using the module **Interpol**.

```
Let UIRDO_f[0,0] = M_UIRDO_AI[0,0]
Interpol UIRDO_f[0,0], UIRDO_AI[0,0], Range0_5V
```

### Working with digital inputs

We can use the communication variable **M_UIRDO_DI** directly in the application algorithm; the variable works with Discrete inputs at addresses 0 to 7.

### Working with digital outputs

We can use the communication variable **M_UIRDO_DO** directly in the application algorithm; the variable works with Coils at addresses 0 to 7.

In order to prevent unnecessary excess communications of identical values, it is recommended to use the code stated in chapter 5.3 “Automatic communication”, section “Recommendation”.

### Retroactive reading of outputs

While writing states is automatic after a value is written into a defined variable thanks to the priority **Auto**, reading has a defined priority **--manual--**. That means that we need to use modules **MdbmRead** or **MdbmMark** for non-periodic value loading.

***Caution***
*If the module **DMM-UI8RDO8** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **Low**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called mpbus_p10_en_xx.dso created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

## 8.1.9 DMM-UI8AO8

The module **DMM-UI8AO8** allows us to read one or multiple analogue inputs in the form of analogue values and binary states. We can read and write analogue outputs one by one or by multiple outputs.

The definition table of full communication with the module may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 0 (40001) | 7 (40008) | 8 | M_UIAO_AO[0,0] | --manual-- | Auto | normal Modbus | 2 |
| 8 (40009) | 9 (40010) | 2 | M_UIAO_Leds[0,0] | --manual-- | Auto | normal Modbus | |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (30001) | 7 (30008) | 8 | M_UIAO_UI[0,0] | Normal | 1 |

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Label |
|---|---|---|---|---|---|
| 0 (10001) | 7 (10008) | 8 | M_UIAO_DI.0 | Low | |

Fig. 20 – Example of communication definition with DMM-UI8AO8

According to the given definition, loaded universal-input values are saved into cells 0 to 7 of the variable **M_UIAO_UI** type Matrix Int (e.g. dimension [1×8]) in the form of analogue values every 1,000 ms. Loaded input values are saved into bits 0 to 7 of the variable **M_UIAO_DI** type Int in the form of binary values every 5,000 ms. Output values are saved into cells 0 to 7 of the variable **M_UIAO_AO** type Matrix Int (e.g. dimension [1×8]) after every entry written into this variable. LED behaviour parameters are saved into cells of the variable **M_UIAO_Leds** type Matrix Int (e.g. dimension [1×2]) after every entry written into this variable.

In order to detect the status of communication with the module, we use the module **MdbmReqSt** linked to the label of the row for reading analogue input values. In our example, we set the property **ClientLabel** to value 90.

```
MdbmReqSt 90, 1, M_UIAO_Stat, NONE
```

### Working with analogue inputs

The values of inputs in Input registers at addresses 0 to 7 are interpreted in a way that the range **0 to 32767** corresponds to **0 % to 100 %** of the input range.

In order to convert the values to the range 0 V to 5 V, we use e.g. the following algorithm using the module **Interpol**.

```
Let UIAO_f[0,0] = M_UIAO_AI[0,0]
Interpol UIAO_f[0,0], UIAO_AI[0,0], Range0_5V
```

### Working with digital inputs

We can use the communication variable **M_UIAO_DI** directly in the application algorithm; the variable works with Discrete inputs at addresses 0 to 7.

### Working with analogue outputs

Values for analogue outputs are interpreted in a way that the range **0 to 32767** corresponds to **0 % to 100 %** of the analogue output range. The analogue output range is available in Holding Registers at addresses 0 to 7.

In order to convert values, we can use e.g. the following algorithm and modules **VarWStat** and **Interpol**.

```
VarWStat UIAO_AO, @UIAO_AO_w, 0
If @UIAO_AO_w
     For i, 0.000, 7.000, 1.000
          Let tmpF1 = UIAO_AO[0,i]
          Interpol tmpF1, tmpF2, Range_AO
          Let M_UIAO_AO[0,i] = Int(tmpF2)
     EndFor
EndIf
```

### Working with LED

In the module **DMM-UI8AO8U**, we are able to programme behaviour of LEDs on the module by means of MODBUS that correspond to individual analogue outputs. This behaviour can be set using two Holding Registers at addresses 8 and 9. LED behaviour is then as follows:

4. Value on the outputs is higher than the value of the register at address 9 – LED is blinking.
5. Value on the outputs is higher than the value of register at address 8 and lower than the value of the register at the address 9 – LED is on.
6. Value on the outputs is lower than the value of the register at the address 8 – LED is off.

The second condition is processed only if the first condition is not met. If neither the first nor the second condition are met, the relevant LEDx is off.
Both limits can be identical or even zero. It is therefore not necessary to enter them. In zero output, LEDx is off; for non-zero output, LEDx is on. LEDx does not get in the blinking mode when the limits are identical.
If the value of register 9 is selected higher than the value of register 8, and if value on the outputs is lower than the value of the register on position 8, LEDx is off.
If the value on the outputs is higher than the value of register 8 and lower than the value of register 9, LEDx is on.
If the value on the outputs is higher than the value of register on position 9, LEDx is blinking.

It also applies in this case that the range **0 to 32767** corresponds to **0 % to 100 %** of the analogue output range.

### Retroactive reading of outputs and LED behaviour parameters

While writing values and parameters is automatic after a value is written into a defined variable thanks to the priority **Auto**, reading has a defined priority **--manual--**. That means that we need to use modules **MdbmRead** or **MdbmMark** for non-periodic value loading.

***Caution***
*If the module **DMM-UI8AO8U** does not detect any frame (with any address) on the bus for a period of 10 s, it interprets it as a communication failure and switches to a secure state. If it is in a network with input modules, we recommend you choose a communication period with input modules max. 5 s (automatic priority **Low**). If it is in a network with only output modules, it is necessary to make sure the variable written on outputs is marked for communication at least once in 5 s. We usually do so by writing any value (even an identical value) into the given variable.*

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called "modbs_p11_en_xx.dso" created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu "Tools/Change Station".

# 9 Appendix C

## 9.1 Programme operation AMR-OPxx

### 9.1.1 AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)

On-wall controllers **AMR-OP7x(RH)**, **AMR-OP6x**, **AMR-OP4x** and **AMR-OP3xA(RH)** provide (either directly from production or after loading a relevant sample application) reading or writing one or multiple values of Holding Registers in the form of analogue values and binary states.

The definition table of full communication with the on-wall controller may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 2 (40003) | 3 (40004) | 2 | OP7x3x_Time | --manual-- | Auto | normal Modbus | |
| 100 (40101) | 100 (40101) | 1 | OP7x3x_Set | --manual-- | Auto | normal Modbus | |
| 101 (40102) | 101 (40102) | 1 | OP7x3x_Reset | --manual-- | Auto | normal Modbus | |
| 102 (40103) | 103 (40104) | 2 | OP7x3x_Stats | Normal | --manual-- | normal Modbus | 1 |
| 104 (40105) | 105 (40106) | 2 | OP7x3x_Corr | --manual-- | --manual-- | normal Modbus | 2 |
| 106 (40107) | 107 (40108) | 2 | OP7x3x_SetPt | --manual-- | Auto | normal Modbus | |
| 108 (40109) | 111 (40112) | 4 | OP7x3x_Vals[0,0] | --manual-- | --manual-- | normal Modbus | 3 |
| 112 (40113) | 113 (40114) | 2 | OP7x3x_LED | --manual-- | Auto | normal Modbus | |

Fig. 21 – Example of communication definition with AMR-OP7x(RH)/AMR-OP6x/ AMR-OP4x/AMR-OP3xA(RH)

More detailed descriptions of individual registers are available in the documentation for the given on-wall controllers or in the description of the sample application.

In order to detect the status of communication with the on-wall controller, we use the module **`MdbmReqSt`** linked to the label of the row for reading double-registers 102-103. In our example, we set the property **`ClientLabel`** to value 10.

```
MdbmReqSt 10, 1, OP7x3x_ReqSt, NONE
```

#### Processing the status after a controller restart or a communication failure

In case of a controller restart or a communication failure, the value of the double-register 102-103 is set to value **`0xFF`**. We expected such a combination of bits to be written into registers 100 and 101 in order for the controller to have a value room mode and fan mode.

```
Let OP7x3x_Reset = 0xFF
Let OP7x3x_Set = (FanMode << 4) | (RoomMode << 1)
```

Furthermore, it is recommended to write previous correction values, setpoint room temperature and LED brightness and a new time for the screensaver into the controller as well.

```
MdbmWrite 10, 2, NONE
Let OP7x3x_SetPt = OP7x3x_SetPt
Let OP7x3x_LED = OP7x3x_LED
GetTime OP7x3x_Time, NONE, NONE
```

### Loading new values from the controller

On-wall controllers use bit 0 of the double-register 102-103 to signalize a value change on part of the controller. This bit can be used in a condition, and when this condition is met, the bit resets, and when the process is running, new values are read from the controller.

```
If OP7x3x_Stats.0
     Let OP7x3x_Reset = 0b1
     Let OP7x3x_Set = 0
     Let @OP3x7x_read = true
Else
     If @OP3x7x_read
          Let @OP3x7x_read = false
          MdbmRead 10, 2, NONE
          MdbmRead 10, 3, NONE
          Let FanMode = Int((OP7x3x_Stats & 0b1110000) >> 4)
          Let RoomMode = Int((OP7x3x_Stats & 0b110) >> 1)
     EndIf
EndIf
```

### Writing actual values into the controller

If the controller is not after a restart or recovering from a communication failure, and no new values are being read from the controller, we can write our own values into the controller.

In order to write a room mode and fan mode, we use the comparison of the previous loaded value of the double-register 102-103 with current values in variables.

```
If (FanMode != Int((OP7x3x_Stats & 0b1110000) >> 4)) or (RoomMode != Int((OP7x3x_Stats
& 0b110) >> 1))
     Let OP7x3x_Set = (FanMode << 4 ) | (RoomMode << 1)
     Let OP7x3x_Reset = 0b1110110 - (FanMode << 4) - (RoomMode << 1)
EndIf
```

It is necessary to write the actual correction value by means of modules of so called safe writing.

```
If @OP3x7x_w_cr
     Let @OP3x7x_w_cr = false
     MdbmWrBeg 10, 2, NONE
     Let OP7x3x_Corr = NewCorr
     MdbmWrFin 10, 2, NONE
EndIf
```

We are able to write the setpoint room temperature value at any time thanks to Write priority **Auto**.

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p12_en_xx.dso” created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

## 9.1.2 AMR-OP7xC

Unlike in on-wall controllers **AMR-OP7x(RH)**, **AMR-OP6x**, **AMR-OP4x** and **AMR-OP3xA(RH)**, the layout of communication Holding Registers is different in terms of presence of registers for working with $CO_2$.

The definition table of full communication with the on-wall controller may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 2 (40003) | 3 (40004) | 2 | OP7xC_Time | --manual-- | Auto | normal Modbus | |
| 100 (40101) | 100 (40101) | 1 | OP7xC_Set | --manual-- | Auto | normal Modbus | |
| 101 (40102) | 101 (40102) | 1 | OP7xC_Reset | --manual-- | Auto | normal Modbus | |
| 102 (40103) | 103 (40104) | 2 | OP7xC_Status | Normal | --manual-- | normal Modbus | 1 |
| 104 (40105) | 105 (40106) | 2 | OP7xC_Corr | --manual-- | --manual-- | normal Modbus | 2 |
| 106 (40107) | 107 (40108) | 2 | OP7xC_SetPnt | --manual-- | Auto | normal Modbus | |
| 108 (40109) | 111 (40112) | 4 | OP7xC_Values[0,0] | --manual-- | --manual-- | normal Modbus | 3 |
| 114 (40115) | 114 (40115) | 1 | OP7xC_LmtCO2 | --manual-- | Auto | normal Modbus | |
| 115 (40116) | 115 (40116) | 1 | OP7xC_MerCO2 | --manual-- | --manual-- | normal Modbus | 4 |

Fig. 22 – Example of communication definition with AMR-OP7xC

More detailed descriptions of individual registers are available in the documentation for the given on-wall controller.

In order to detect the status of communication with the on-wall controller, we use the module **`MdbmReqSt`** linked to the label of the row for reading double-registers 102-103. In our example, we set the property **`ClientLabel`** to value 20.

```
MdbmReqSt 20, 1, OP7xC_ReqSt, NONE
```

### Processing the status after a controller restart or a communication failure

The processing procedure corresponds to the previous case from chapter 9.1.1 “AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)”, section “Processing the status after a controller restart or a communication failure”.

The only difference is in writing the limit value of $CO_2$ concentration instead of the LED brightness value.

```
Let OP7xC_LmtCO2 = OP7xC_LmtCO2
```

### Loading new values from the controller

The processing procedure corresponds to the previous case from chapter 9.1.1 “AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)”, section “Loading new values from the controller”.

The only difference is added reading of the measured $CO_2$ value.

```
MdbmRead 20, 4, NONE
```

### Writing actual values into the controller

The processing procedure corresponds to the previous case from chapter 9.1.1 “AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)”, section “Writing actual values into the controller”.

The only difference is the option to write the limit value of $CO_2$ concentration at any moment thanks to Write priority **`Auto`** in this register.

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p13_en_xx.dso” created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

### 9.1.3 AMR-OP7xRHC

Unlike in the on-wall controller **AMR-OP7xC**, the layout of communication Holding Registers is different in terms of register order for working with $CO_2$.

The definition table of full communication with the on-wall controller may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 2 (40003) | 3 (40004) | 2 | OP7RHC_Time | --manual-- | Auto | normal Modbus | |
| 100 (40101) | 100 (40101) | 1 | OP7RHC_Set | --manual-- | Auto | normal Modbus | |
| 101 (40102) | 101 (40102) | 1 | OP7RHC_Reset | --manual-- | Auto | normal Modbus | |
| 102 (40103) | 103 (40104) | 2 | OP7RHC_Stat | Normal | --manual-- | normal Modbus | 1 |
| 104 (40105) | 105 (40106) | 2 | OP7RHC_Corr | --manual-- | --manual-- | normal Modbus | 2 |
| 106 (40107) | 107 (40108) | 2 | OP7RHC_SetPt | --manual-- | Auto | normal Modbus | |
| 108 (40109) | 113 (40114) | 6 | OP7RHC_Vals[0,0] | --manual-- | --manual-- | normal Modbus | 3 |
| 116 (40117) | 116 (40117) | 1 | OP7RHC_LmtH2 | --manual-- | Auto | normal Modbus | |

Fig. 23 – Example of communication definition with AMR-OP7xRHC

More detailed descriptions of individual registers are available in the documentation for the given on-wall controller.

In order to detect the status of communication with the on-wall controller, we use the module **`MdbmReqSt`** linked to the label of the row for reading double-registers 102-103. In our example, we set the property **`ClientLabel`** to value 30.

```
MdbmReqSt 30, 1, OP7RHC_ReqSt, NONE
```

#### Processing the status after a controller restart or a communication failure

The processing procedure corresponds to the previous case from chapter 9.1.1 “AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)”, section “Processing the status after a controller restart or a communication failure”.

The only difference is in writing the limit value of $CO_2$ concentration instead of the LED brightness value.

```
Let OP7RHC_LmtH2 = OP7RHC_LmtH2
```

#### Loading new values from the controller

The processing procedure corresponds to the previous case from chapter 9.1.1 “AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)”, section “Loading new values from the controller”.

The only difference is that the matrix variable **`OP7RHC_Vals`** also includes the loaded value of the measured $CO_2$ concentration.

#### Writing actual values into the controller

The processing procedure corresponds to the previous case from chapter 9.1.1 “AMR-OP7x(RH) / AMR-OP6x / AMR-OP4x / AMR-OP3xA(RH)”, section “Writing actual values into the controller”.

The only difference is the option to write the limit value of $CO_2$ concentration at any moment thanks to Write priority **`Auto`** in this register.

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p14_en_xx.dso” created in DetStudio development environment. This project has been

created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

### 9.1.4 AMR-OP40(RH)C

Unlike the aforementioned on-wall controllers, layout of communication Holding Registers is different since it does not use status registers.

The definition table of full communication with the on-wall controller may look like in the following image.

Holding registers | Input registers | Coils | Discrete inputs

| Address (Modicon) | Address end (Modicon) | Count | Variable | Read priority | Write priority | Write function | Label |
|---|---|---|---|---|---|---|---|
| 108 (40109) | 113 (40114) | 6 | OP40C_Values[0,0] | Low | --manual-- | normal Modbus | 1 |
| 115 (40116) | 115 (40116) | 1 | OP40C_LED | --manual-- | Auto | normal Modbus | |
| 116 (40117) | 116 (40117) | 1 | OP40C_LmtH2 | --manual-- | Auto | normal Modbus | |
| 117 (40118) | 117 (40118) | 1 | OP40C_LmtH1 | --manual-- | Auto | normal Modbus | |

Fig. 24 – Example of communication definition with AMR-OP40(RH)C

More detailed descriptions of individual registers are available in the documentation for the given on-wall controller.

In order to detect the status of communication with the on-wall controller, we use the module **`MdbmReqSt`** linked to the label of the row for reading registers 108 to 113. In our example, we set the property **`ClientLabel`** to value 40.

```
MdbmReqSt 40, 1, OP40C_ReqSt, NONE
```

#### Processing the status after a controller restart or a communication failure

The on-wall controller does not signalize a restart or a communication failure. For this purpose, we can use a variable with a communication status. LED brightness and limit values of $CO_2$ concentration are therefore written when a communication failure has been detected and the communication has been re-established.

```
If OP40C_ReqSt.4
     Let @OP40C_write = true
EndIf

If @OP40C_write and OP40C_ReqSt.1
     Let @OP40C_write = false
     Let OP40C_LED = OP40C_LED
     Let OP40C_LmtH1 = OP40C_LmtH1
     Let OP40C_LmtH2 = OP40C_LmtH2
EndIf
```

#### Loading new values from the controller

Values from the controller are loaded periodically with **`Low`** priority, i.e. every 5,000 ms.

#### Writing actual values into the controller

Values can be written at any time thanks to Write priority **`Auto`** in given registers.

Stated algorithms are included in annex ap0008_en_xx.zip. It is a sample project called “modbs_p15_en_xx.dso” created in DetStudio development environment. This project has been created for the control system **AMiNi4DW2**. However, it can be modified to suit any control system fitted with a serial communication interface using the DetStudio menu “Tools/Change Station”.

# 10 Technical support

The AMiT Technical Support Department provides all information regarding communication in MODBUS RTU network. The Technical support is best contacted via e-mail at **support@amit.cz**.

# 11 Warning

In this document, AMiT, spol. s r.o. provides information as it is, and the company does not provide any warranty concerning the contents of this publication and reserves the right to change the documentation content without any obligation to inform anyone or any authority about it.

This document can be copied and redistributed under the following conditions:

1. The whole text (all pages) must be copied without making any modifications.
2. All redistributed copies must retain the AMiT, spol. s r.o. copyright notice and any other notices contained in the documentation.
3. This document must not be distributed for profit.

The names of products and companies used herein may be trademarks or registered trademarks of their respective owners.