Communication in M-Bus network (Gen2)

Abstract

Transfer of technological data between Generation 2 control systems made by AMiT and energy meters via the M-Bus protocol.

Author: Michal Kupčík
Document: ap0062_ap_en_002.pdf

Attachments

File contents: ap0062_ap_en_002.zip

mbus_p1_en_101.dsox	Loading data from meters into the control system using the UserComBinary objects including decoding
mbus_p2_en_101.dsox	Loading data from meters into the control system via MODBUS RTU and MODBUS TCP protocols including decoding

Contents

Revision history

Version	Date	Changes by	Changes
001	13. 04. 2021	Kupčík M.	New document.
002	06. 04. 2022	Kupčík M.	Local text corrections, modified memory usage values in chapter 2.2.2.

Related documentation

1. Help tab in the EsiDet section of the **DetStudio** development environment
   file: EsiDet_en.chm
2. **DM-MBUS64** – communication converter with M-Bus protocol – Operation manual
   file: dm-mbus64_g_en_xxx.pdf
3. Application note AP0056 – Communication in MODBUS RTU network
   file: ap0056_ap_cz_xxx.pdf
4. Application note AP0057 – Communication in MODBUS TCP network
   file: ap0057_ap_cz_xxx.pdf
5. EN 13757 – Set of standards defining the M-Bus protocol

Purpose of the application note

The application note is intended for Generation 2 control systems – control systems with the FreeRTOS operating system programmed in DetStudio in the EsiDet editor. More information about the system generations can be found on the **amitomation.cz** website.

1 M-Bus

M-Bus, or Meter-Bus, standard is designed to collect data from consumption meters of various media (e.g. potable water, DHW, gas, heat, electricity). It was developed by University Padeborn and the Texas Instruments company.

1.1 Basic characteristics

Considering its relatively narrow and specialised field of application, there are very specific requirements for M-Bus. It has to be able to interconnect a large number of devices at a distance of up to several kilometres. It requires high-quality protection of data transfer against errors. On the other hand, a typical feature of the application is a fairly infrequent reading of measured values with low demand for real-time response.

Standard configuration characteristics can be summed up in the following points:

- special implementation of the physical layer,
- galvanically isolated interface,
- connected devices can be optionally powered via the bus,
- two-wire line up to several kilometres long,
- communication control based on Master-Slave principle,
- up to 250 participants without the implementation of a network layer,
- asynchronous character transfer, 8 bits of data, even parity,
- baud rate of 300 bps to 38,400 bps,
- data block secured by means of a check sum.

Standard M-Bus configuration includes a single control station (Master) and up to 250 participant stations. Typically, M-Bus networks use linear topology. The length of a cable segment in this configuration must not exceed 1000 m (350 m for 9,600 bps). For more extensive systems, it is necessary to use more complex configurations when the entire system is divided into zones. Individual zones consist of segments and these are controlled by zone controllers.

A complete and detailed description of the M-Bus protocol is available in EN 13757.

1.2 Physical layer

The M-Bus standard uses special implementation of the physical layer described in standards EN 13757-2 and EN 13757-4. Typically, this layer is a two-wire bus based on a common telephone cable with half-duplex data transfer and a Master-Slave access control. To power participant stations using that cabling, M-Bus uses changes in voltage levels for transfer from the control station to participant station; it uses changes in current consumption in the opposite direction. In the direction from the control station to the subscriber stations, the logical one corresponds to the voltage +36 V at the output of the control station exciter, and the logical zero corresponds to the voltage 12 V lower, i.e. +24 V. In the direction from the subscriber station to the control station, the logic 1 is represented by a current draw of 1.5 mA, the logic 0 by a current draw of 11 mA to 20 mA higher. Current consumed while in the state of log. 1 can be used to power galvanically isolated interface and possibly even the meter itself.

While idle, the bus is on the level of log. 1 (+36 V) and the consumption from the control station driver corresponds to the number of participant stations multiplied by the consumption of a participant station in log. 1, i.e. 1.5 mA. As is evident from the standard requirements, the higher the number of participant station, the higher are the demands on the driver. Moreover, given that the number of participant stations can vary with time – as they can be added or removed at any time – the control station cannot detect the absolute current consumption, only the aforementioned difference. Similarly, the voltage at different points of the bus – as monitored by the participant stations – varies; it depends on the resistance of the line and on the number of participant stations connected to it. That means that the participant stations have to react to the changes in the voltage of the bus, not to the absolute values. Given the rather significant changes in both the voltage

(12 V) and the current (11 mA to 20 mA), the physical layer of the standard shows a high resistance to external noise.

The second option is to use wireless transmission of data. Given that the AMiT control systems do not support communication over said frequencies, this option is not discussed in this application note.

1.3 Frame format

M-Bus uses four frame formats:
- Single Character,
- Short Frame,
- Control Frame,
- Long Frame.

Single character
0xE5

Short frame
Start 0x10
C field
A field
Checksum
Stop 0x16

Control frame
Start 0x68
L field = 3
L field = 3
Start 0x68
C field
A field
CI field
Checksum
Stop 0x16

Long frame
Start 0x68
L field
L field
Start 0x68
C field
A field
CI field
User data (0 to 252) byte
Checksum
Stop 0x16

Single character
This frame consists of a single character, namely 0xE5. It is used to confirm that another transmitted frame has been received.

Short frame
This frame has a fixed length. It starts with introductory character 0x10, followed by field C, field A, check sum and the closing character 0x16. Check sum is calculated only from fields C and A.

Control frame
The content of the control frame corresponds to the long frame, but it does not include “user data”. Check sum is calculated from fields C, A and CI.

Long frame
This frame starts with character 0x68, followed by a twice repeated L field containing the data length, and then again by the introductory character 0x68. This is followed by fields C, A and CI, user data with length ranging from (0 to 252) byte, check sum and the terminating character 0x16. L field contains the number of bytes of user data increased by three (i.e. by the length of fields C, A and CI). Check sum is calculated from fields C, A and CI and from user data.

1.3.1 Frame fields description

All fields have a length of one character which corresponds to eight bits.

C field (Control Field, Function Field)
This field contains frame control parameters. Among other things, it determines the data flow direction. The description below calls it the main byte of the frame.

A field (Address Field)
The A field is used for addressing or determining a participant station or to transmit/receive a frame.

Individual participant stations (slaves) can take addresses from 0 to 250. Entering address 255 sends a broadcast that requires no reply. Entering address 254 sends a broadcast that requires a reply. Clearly, in the second case, replies will collide when there is more than one participant station connected. It is used solely for test purposes. Address 253 indicates so-called secondary addresses. Addresses 251 and 252 are reserved for future use.

The secondary addresses are not discussed in this application note.

CI field (Control Information Field)
CI field carries information that is already included in the application protocol. Among other things, it is used to distinguish the format of a long and control frame.

User data
When it is a reply from a meter with usual data (CI field has a value of 0x72 or 0x76 – RSP_UD), these bytes have the following significance:

- Header bytes 8 to 19:
  - 4 bytes of ID number of the meter; these usually correspond to a part of the serial number of the meter or its secondary address,
  - 2 bytes code three identification letters of the meter manufacturer,
  - 1 byte with the meter version,
  - 1 byte codes the metered medium,
  - 1 byte with the frame counter,
  - 1 byte codes the meter state,
  - 2 bytes “signature” – reserved for future specification, values 0x00 0x00.
- following data bytes:
  - 1 byte DIF – definition of the value number type; when the high bit is set, the following part is DIFE,
  - 0 to 10 bytes DIFE – extension of the value identifier; when the high bit is set, the following part is another DIFE,
  - 1 byte VIF – value unit and multiplier; when the high bit is set, the following bit is VIFE,
  - 0 to 10 bytes VIFE – extended identifier of the value unit and multiplier; when the high bit is set, the following bit is another VIFE,
  - 0 to xx bytes for value; in the case of ASCII unit, the unit is sent first, the value bytes come after,
  - what follows is that many combinations of DIF (+ n*DIFE) + VIF (+ m*VIFE) + value, how many quantities there are.

1.3.2 Usage of frames

This application note uses the following frames:

- single character – meter response to initialisation,
- short frame – meter initialisation (SND_NKE) – using the main byte of the frame 0x40,
- short frame – request for usual data (SND_UD2) – using the main byte of the frame 0x5B or 0x7B,
- long frame – response of the meter with usual data (RSP_UD) – received main byte 0x08/0x18/0x28/0x38.

2 Communication of control systems via M-Bus

Control systems of the **AMiT** company always take the role of the **master** in the **M-Bus** network, therefore, they cannot be used as slaves in M-Bus networks.

2.1 HW parametrisation

It is necessary to use a converter for communication of control systems in a M-Bus network. The AMiT company offers **DM-MBUS64** – an M-Bus converter – that can be connected to control systems both via serial line and via the Ethernet interface. The converter **does not perform** any direct coding of the meter data.

Parametrisation of the converter is usually done via a web site while it is in the so-called service mode. More information on the parametrisation of the converter can be found in the corresponding operation manual.

The converter supports communication via transparent data transfer (Direct mode), encapsulation of M-Bus data into the APE protocol and through MODBUS protocols. The following chapters operate with the transparent data transfer (Direct mode).

Further description of the APE protocol is in the operation manual for the converter.

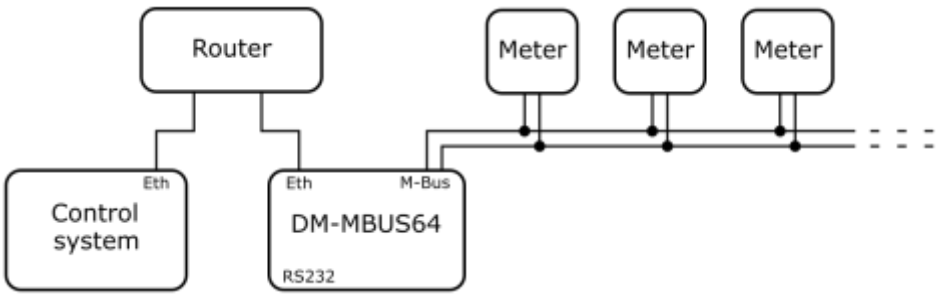

Fig. 1 – Connection of the control system to the converter (**DM-MBUS64**) via Ethernet

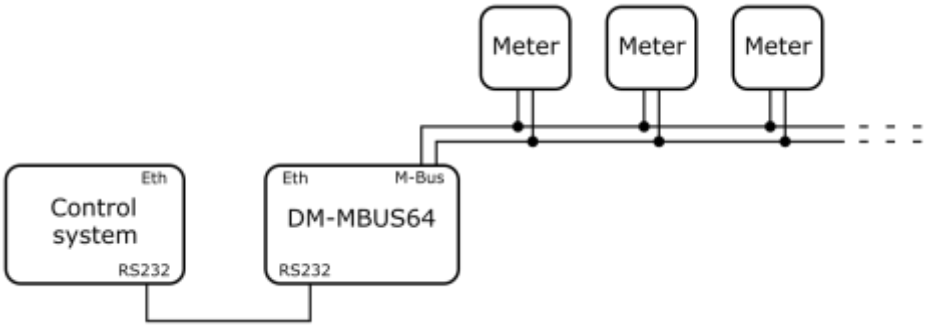

Fig. 2 – Connection of the control system to the converter (**DM-MBUS64**) via RS232

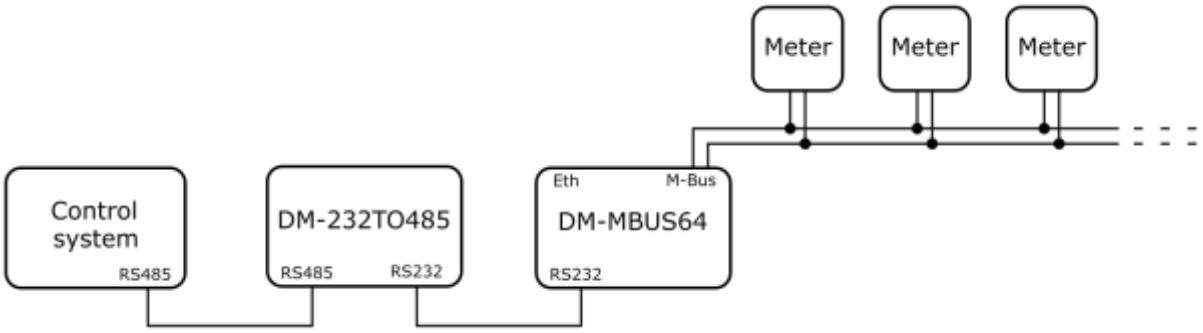

Fig. 3 – Connection of the control system to the converter (**DM-MBUS64**) via RS485 and the **DM-232TO485** converter

Algorithms described below can be used even when using converters of other manufacturers. Converters of other manufacturers have to support transparent data transfer (whatever they receive from the serial line or Ethernet, they send to M-Bus without any changes [in terms of SW] and vice versa).

Caution
A single serial line of the control system or HMI can use only one communication protocol. When using communication with a converter and via a serial line, the same communication protocol has to be selected for the converter as for other devices connected to the serial line.

2.2 SW parametrisation

The following steps are usually employed to communicate with the meter:

1. send initialisation short frame,
2. wait for a single character response 0xE5,
3. a) in the case the response does not come, re-send the initialisation short frame,
   b) when the response comes, send a short frame with a request for usual data,
4. a) in the case the response with usual data does not come, re-send the short frame with a request for usual data,
   b) when the response with usual data comes, decode it,
5. in rare cases, when it is needed to retrieve more usual data from the meter, repeat step 3. b) with the other value of the main byte of the frame 0x5B/0x7B.

The **`fb_MBusDevice`** function block sends the initialisation frame and the usual data request. Furthermore, the function block deals with receiving data and with re-sending the last sent frame.

The **`fb_MVusDecode`** function block decodes received usual data to numeric and text values; it may also signalise potential issues that occur during decoding.

2.2.1 Function block fb_MBusDecode

Basic characteristics of the function block:

- decodes up to 40 numeric values from the meter – **`Values`** parameter,
- decodes up to 10 text values from the meter – **`ValuesStrings`** parameter,
- each value is accompanied by a read unit – **`UnitsStrings`** parameter,
- presentation of the numerical code of the meter medium – **`DeviceInfo`** parameter,
- decoding the three-letter manufacturer ID – **`DeviceInfo`** parameter,
- support of fixed standards of value formats: Int (1 to 8 bytes), Real (4 bytes) and BCD (2 to 12 digits),
- support of “LVAR” of variable value formats: ASCII string (up to 20 characters), positive and negative BCD value (up to 16 digits) and a binary number (up to 8 bytes),

- DIFE values are not processed in any way,
- support of standard VIF units: Wh, J, m^3, kg, sec., min., hours, days, W, J/h, m^3/h, m^3/min., m^3/s, kg/h, °C, K, bar, DateTime,
- support for extended VIFE units for VIF 0x7B and 0x7D: MWh, GJ, m^3, tons, MW, GJ/h, °C, W, Volts, Ampers,
- support for textual representation of units for VIF 0x7C,
- using a question mark character in the case the combination of VIF and VIFE is not one of the supported ones,
- frame inspection: header check, received address correctness check, CRC check, frame data completeness check, signalisation of non-decodable value or data types – **Errors** parameter.

The **fb_MBusDevice** function block operates in its body with an instance of the **fb_MBusDecode** function block. For correct operation, the **fb_MBusDecode** definition needs to be present in DetStudio in the “Project” window in the “Function blocks” node.

Complete description of the **fb_MBusDecode** function block is in the “Documentation” tab of the function block definition in DetStudio.

2.2.2 Function block fb_MBusDevice

Basic characteristics of the function block:
- used in processes with a period of 100 ms,
- launches communication with the meter at the leading edge of the **SyncMarkIn** input parameter,
- alternation of defined outbound communication frames,
- doubled repetition of outbound frames when no correct response comes,
- handover of received usual data to the **fb_MBusDecode** function block,
- saving the received usual data frame for potential follow-up user processing,
- presentation of decoded values, units and errors – parameters **Values**, **ValuesStrings**, **UnitsStrings**, **DeviceInfo** and **Errors**,
- existence of resources for sharing one communication interface for multiple meters defined by instances of **fb_MBusDevice** function blocks.

Meter addressing
Usually, one instance of a function block corresponds to one meter. This meter is defined by its primary address. When a negative address value is entered, the corresponding block is eliminated from the communication group.

Multiple-frame meters
In some cases, the meter can provide more usual data than a single frame can accommodate. For example, when the meter provides 600 byte of usual data, it is necessary to communicate with the meter by sending the frames with requests in the following order:
- initialisation short frame,
- request for usual data with the main frame byte 0x5B,
- request for usual data with the main frame byte 0x7B,
- request for usual data with the main frame byte 0x5B.

The **CommType** parameter of the function block defines whether the given instance should send initialisation frames or not and which byte is to be used as the main byte of the frame. Possible values:
- 0 – an initialisation frame is sent as well as a request for values with the main byte 0x5B,
- 1 – an initialisation frame is sent as well as a request for values with the main byte 0x7B,
- 2 – only request for values is sent with the main byte 0x5B,
- 3 – only request for values is sent with the main byte 0x7B.

The mentioned multi-frame meter will be represented by three instances of the function block while the value of the address is always the same and the values of the **CommType** parameters are defined for individual instances of blocks at 0, 3 and 2.

Memory requirements
Each use of the function block uses 4.4 kB of RAM memory and 1.9 kB of FLASH memory. Control systems with 1 MB of RAM are therefore limited to roughly a 230 **fb_MBusDevice** blocks.

When it is necessary to use more blocks, it is possible to bypass this limit by lowering the number of rows of the output matrix parameters with decoded values:

- Function block **fb_MBusDevice**:
  - parameter **Values** with default dimensions set to 40×1,
  - parameter **UnitStrings** with default dimensions set to 40×20,
  - parameter **ValuesStrings** with default dimensions set to 10×20.

For example, if the first two matrices mentioned above with a default dimension of 40 rows are reduced to 20 rows and the third matrix with a default dimension of 10 rows is reduced to 2 rows, the RAM requirement drops to 3.1 kB. In the case of a control system with 1 MB of RAM, this would result in a rise of the maximum possible number of **fb_MBusDevice** function blocks to ca. 320.

Complete description of the **fb_MBusDevice** function block is in the “Documentation” tab of the function block definition in DetStudio.

2.2.3 Usage of function block fb_MBusDevice

The function block provides frames for transmission, defines occasions when they are to be sent or, in the case of received data, when they are to be read, when the block is to receive the number of bytes received in the frame and the frame as such.

This is represented by block parameters:

- **FrameReceived** – received frame,
- **BytesReceived** – amount of data in the received frame,
- **FrameSend** – frame to be sent,
- **Command** – definition of a communication action.

Based on the value of the **Command** parameter, one of the following actions is supposed to occur:

- 0, 1 – no action,
- 2 – send 5 characters of the frame readied in the **FrameSend** parameter and set the **Command** parameter to 1,
- 3 – save the received frame into the **FrameReceived** parameter, save the number of received bytes to the **BytesReceived** parameter and set the **Command** parameter to 1.

The function block does not perform any communication over any interface. To send or receive frames, the **UserComBinary** object is used – specifically its **Send()** method and parameters **BufferWrite**, **BufferRead** and **BytesReceived**. When sending the frame, it is recommended to erase the incoming buffer; when receiving a frame, it is recommended to erase the outgoing buffer. Whether the data are sent via serial line or Ethernet depends on the settings of the **UserComBinary** communication object parameters.

More information on the **UserComBinary** communication object can be found in the **DetStudio** IDE help.

Using shared variables
Usually, there is more than one meter connected to a converter. These meters are represented as instances of the **fb_MBusDevice** function block. At any given time, only one instance of the function block can be accessing a **UserComBinary** communication object and read data from one of the meters. Other instances of the **fb_MBusDevice** function block can access the same **USerComBinary** communication object only after the previous instance of **fb_MBusDevice** finishes its work. That means that variables that are used to set the parameters of the **fb_MBusDevice** function block need to be shared. Specifically, this concerns four variables of the following types and dimensions:

- ARRAY OF INT [1×262] – used in **FrameReceived** parameters,
- INT – used in **BytesReceived** parameters,
- ARRAY OF INT [1×5] – used in **FrameSend** parameters,
- INT – used in **Command** parameters.

Sample usage for three instances of the **fb_MBusDevice** function block defining three meters:

```
fb_MBusDevice1(
     FrameReceived = MBusFrameRec,
     BytesReceived = MBusBytesRec,
     FrameSend = MBusFrameSend,
     Command = MBusCommand
);

fb_MBusDevice2(
     FrameReceived = MBusFrameRec,
     BytesReceived = MBusBytesRec,
     FrameSend = MBusFrameSend,
     Command = MBusCommand
);

fb_MBusDevice3(
     FrameReceived = MBusFrameRec,
     BytesReceived = MBusBytesRec,
     FrameSend = MBusFrameSend,
     Command = MBusCommand
);
```

In this example, this concerns variables **MBusFrameRec**, **MBusBytesRec**, **MBusFrameSend** and **MBusCommand**.

Algorithm of operation with the UserComBinary object

When the aforementioned parameters of **fb_MBusDevice** function blocks are linked to shared variables, it is possible to create a simple algorithm that operates the **UserComBinary** communication object based on the value of the shared variable defined for the **Command** parameter.

Resulting code:

```
CASE MBusCommand OF
     2:
          FOR i = 0 TO 4 DO
               UserComBinary1.BufferWrite[0,i] = Int_To_Byte(MBusFrameSend[0,i]);
          ENDFOR;
          UserComBinary1.Send(5);
          UserComBinary1.ClearReadBuffer();
          MBusCommand = 1;
     3:
          FOR i = 0 TO UserComBinary1.BytesReceived DO
               MBusFrameRec[0,i] = UserComBinary1.BufferRead[0,i];
          ENDFOR;
          MBusBytesRec = UserComBinary1.BytesReceived;
          UserComBinary1.ClearWriteBuffer();
          MBusCommand = 1;
ENDCASE;
```

The **Command** parameter is used by the function blocks not only for writing a required action, it is also read. This prevents collisions when more instances try to send or accept inbound data at the same time.

3 Sample application

The application note includes a sample application designed for communication with meters. This application includes the definition of two **`UserComBinary`** communication objects. Each is defined with a different IP address representing a different communication converter. Each **`UserComBinary`** object uses three instances of the **`fb_MBusDevice`** function block.

Individual communications are divided into subroutines that are called from the periodic process with a period of 100 ms. Communication via the communication object is allowed by setting the **`AutomatRun`** local variable to a positive, non-zero value. Reading as such is launched by a synchronisation bit from the **`SyncMark1`** local block. By default, the communication is to occur every hour. The frequency of the reads can be changed by adjusting the values of parameters of the **`SyncMark1`** block.

More information on the **`SyncMark`** block can be found in the **DetStudio** IDE help.

The sample application is attached to the application note in the “mbus_p1_cz_xxx.dsox” file. This project has been created for control system **AMR-OP87 RevA**. However, it can be modified to suit any control system fitted with the Ethernet interface using the DetStudio menu “Tools/Change station…”.

Caution
The sample application was designed for DetStudio 2.2.43 or newer!

4 Appendix A

4.1 Communication via MODBUS protocols

When it is not possible, for any reason, to use communication with a converter via the **`UserComBinary`** communication object, it is possible to communicate with the **DM-MBUS64** converter via MODBUS protocols. That is either MODBUS RTU via one of the serial lines or MODBUS TCP via the Ethernet interface.

4.1.1 HW parametrisation

Parametrisation of the **DM-MBUS64** converter is usually done via a web site while it is in the so-called service mode. The corresponding converter interface needs to be set to MODBUS.

4.1.2 SW parametrisation

To communicate with the converter, the following holding registers are used:

- 100 – setting the converter action,
- 101 to 361 – outbound frame to the M-Bus network,
- 400 – reading the state of the inbound frame,
- 402 – number of inbound frames from the M-Bus network,
- 403 to 663 – inbound frame from the M-Bus network.

Detailed description of the converter actions and communication registers can be found in its operation manual.

To initialise a meter or a request for usual data, a short, 5-byte frame is used. One of the functions of the **DM-MBUS64** converter is the automatic checksum value calculation and appending the end character 0x16. It is therefore sufficient to write only 3 bytes of the outbound frame and to communicate only registers 101 to 103. Given that there may be issues in the future, the sample application operates with a communication definition with 20 registers in the range of 101 to 121.

For initialisation of reading, error monitoring and value decoding, the previously described **`fb_MBusDevice`** function block can be used. To use it, all communication with the converter has to be set as event-based – writing into registers will be performed after the corresponding variable is set, and registers will be read after the **`Refresh()`** method of the corresponding variable was called. This is the reason why all priorities of the register communication are set to **`None`**.

The MODBUS protocol supports reading of up to 125 registers. An inbound frame from the M-Bus network can potentially have up to 261 characters. That is why registers 403 to 663 need to be separated into three communicated groups. The sample application uses division to 90, 90 and 81 registers. The following figure shows the list of used definitions of MODBUS registers. This applies to communication via MODBUS RTU protocol and MODBUS TCP protocol.

ModbusMaster1.ModbusDevice1

Name	Type	Row	Col	Registers	Kind	Priority	MB Read function	MB Write function	Comment
MBus_Flags	WORD			100	Holding register	None	Function_03	Function_16	
MBus_OutTlg	WORD[.]	1	20	101	Holding register	None	Function_03	Function_16	
MBus_ComStat	WORD			400	Holding register	None	Function_03	Function_16	
MBus_RecChars	WORD			402	Holding register	None	Function_03	Function_16	
MBus_InTlg0	WORD[.]	1	90	403	Holding register	None	Function_03	Function_16	
MBus_InTlg1	WORD[.]	1	90	493	Holding register	None	Function_03	Function_16	
MBus_InTlg2	WORD[.]	1	81	583	Holding register	None	Function_03	Function_16	

Fig. 4 – List of used MODBUS registers' definitions

More information on communication objects **`ModbusMaster`**, **`ModbusDevice`**, **`ModbusMasterTCP`** and **`ModbusDeviceTCP`** can be found in the **DetStudio** IDE help.

As soon as the corresponding **`ModbusDevice`** or **`ModbusDeviceTcp`** communication object is defined, it is possible to adjust the communication algorithm. To communicate via the MODBUS protocol with the **DM-MBUS64** converter, the following method is recommended:

1. save the outbound frame to the converter and reset the number of received characters,
2. pause for the communication to occur,
3. write 1 to the action register of the converter to send a frame into the M-Bus network,
4. pause for the communication to occur,
5. read the value of the inbound frame status register,
6. pause for the communication to occur,
7. reading the status of the inbound frame; possible variants:
   a) the inbound frame has yet to come, repeat from step 4,
   b) the inbound frame came – read values of the converter frame, the number of incoming bytes and reset the value of the inbound frame state, continue with step 8,
   c) the converter is indicating an error, the algorithm therefore interrupts the communication prematurely and rests the value of the inbound frame status,
8. pause for the communication to occur,
9. processing the values and clearing the buffer of the incoming frame from the converter's M-Bus network by writing the decade value 203 to the converter's action register.

This method described above is representing the following code:

```
CASE Modbus_step OF
     1:
          tmpOutTlg[0,0] = Int_To_Byte(MBusFrameSend[0,0]);
          tmpOutTlg[0,1] = Int_To_Byte(MBusFrameSend[0,1]);
          tmpOutTlg[0,2] = Int_To_Byte(MBusFrameSend[0,2]);
          ModbusMaster1.ModbusDevice1.MBus_OutTlg = tmpOutTlg;
          MBusBytesRec = 0;
          Modbus_step = 2;
     2:
          Modbus_step = 3;
     3:
          ModbusMaster1.ModbusDevice1.MBus_Flags = 1;
          Modbus_time = 0;
          Modbus_step = 4;
     4:
          Modbus_time = Modbus_time + 1;
          Modbus_step = Modbus_time >= 2 ? 5 : 4;
```

```
      5:
            ModbusMaster1.ModbusDevice1.MBus_ComStat.Refresh();
            Modbus_time = 0;
            Modbus_step = 6;
      6:
            Modbus_time = Modbus_time + 1;
            Modbus_step = Modbus_time >= 2 ? 7 : 6;
      7:
            IF ModbusMaster1.ModbusDevice1.MBus_ComStat.0 THEN
                  Modbus_time = 0;
                  Modbus_step = 4;
            ELSE
                  IF ModbusMaster1.ModbusDevice1.MBus_ComStat.1 THEN
                        ModbusMaster1.ModbusDevice1.MBus_RecChars.Refresh();
                        ModbusMaster1.ModbusDevice1.MBus_InTlg0.Refresh();
                        ModbusMaster1.ModbusDevice1.MBus_InTlg1.Refresh();
                        ModbusMaster1.ModbusDevice1.MBus_InTlg2.Refresh();
                        ModbusMaster1.ModbusDevice1.MBus_ComStat = 0;
                        Modbus_time = 0;
                        Modbus_step = 8;
                  ELSE
                        ModbusMaster1.ModbusDevice1.MBus_ComStat = 0;
                        Modbus_step = 0;
                  ENDIF;
            ENDIF;
      8:
            Modbus_time = Modbus_time + 1;
            Modbus_step = Modbus_time >= 6 ? 9 : 8;
      9:
            MBusBytesRec = ModbusMaster1.ModbusDevice1.MBus_RecChars;
            j = 0;
            FOR i = 0 TO 89 DO
                  MBusFrameRec[0,j] = ModbusMaster1.ModbusDevice1.MBus_InTlg0[0,i];
                  j = j + 1;
            ENDFOR;
            FOR i = 0 TO 89 DO
                  MBusFrameRec[0,j] = ModbusMaster1.ModbusDevice1.MBus_InTlg1[0,i];
                  j = j + 1;
            ENDFOR;
            FOR i = 0 TO 80 DO
                  MBusFrameRec[0,j] = ModbusMaster1.ModbusDevice1.MBus_InTlg2[0,i];
                  j = j + 1;
            ENDFOR;
            ModbusMaster1.ModbusDevice1.MBus_Flags = 203;
            Modbus_step = 0;
ENDCASE;
```

Let's reiterate that the **fb_MBusDevice** function block determines the occasion when messages are sent and when read data are accepted via the **Command** parameter. The algorithm for value 2 (sending data) gets simplified to start the algorithm described above by setting the **Modbus_step** variable to 1. The algorithm for value 3 (data handover) is completely deleted as the data are handed over in the MODBUS reading algorithm in step 9. The resulting code for processing control values:

```
CASE MBusCommand OF
      2:
            Modbus_step = 1;
            MBusCommand = 1;
      3:
            MBusCommand = 1;
endcase;
```

4.1.3 Sample application

The application note includes a sample application designed for communication with meters via the MODBUS RTU and MODBUS TCP protocols. Every communication uses three instances of the **`fb_MBusDevice`** function block.

Individual communications are divided into subroutines that are called from the periodic process with a period of 100 ms. Communication via the communication object is allowed by setting the **`AutomatRun`** local variable to a positive, non-zero value. Reading as such is launched by a synchronisation bit from the **`SyncMark1`** local block. By default, the communication is to occur every hour. The frequency of the reads can be changed by adjusting the values of parameters of the **`SyncMark1`** block.

More information on the **`SyncMark`** block can be found in the **DetStudio** IDE help.

The sample application is attached to the application note as the “mbus_p2_cz_xxx.dsox” file. This project has been created for control system **AMR-OP87 RevA**. However, it can be modified to suit any control system fitted with the Ethernet interface using the DetStudio menu “Tools/Change station…”.

Caution
The sample application was designed for DetStudio 2.2.43 or newer!

5 Technical support

For all information regarding the communication in a network of the M-Bus in AMiT control systems, please contact AMiT Technical Support. Technical support can be contacted preferably by e-mail at **support@amit.cz**

6 Notice

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.

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.