Digatron power electronics

Battery Manager

Release R2016-1 (version 4.11) and later

OPC Interface

Programming Manual

Company

Address: Digatron Power Electronics GmbH
Tempelhofer Str. 12 – 14
52068 Aachen
Germany
Phone: +49-241-168090
Fax: +49-241-1680919
Email: info@digatron.de
Web: www.digatron.com

Battery Manager, OPC Interface Programming Manual

Trademarks

Table of contents

1 About this manual

1.1 Purpose

This manual describes the *OPC interface* of the Battery Manager.

1.2 Symbols

Within this document the following symbols might be used:

This symbol indicates helpful tips and information relating to the surrounding text.

This symbol indicates general information and/or warnings.
Ignoring these information/warnings can result in improper system behavior, malfunctions, physical damage, personal injury or even death of persons.

1.3 Typographical conventions

This document might use one or several of the following typographical conventions:

Presentation	Meaning
x	A key of the keyboard or a button on the screen to be pressed.
<placeholder>	A placeholder which has to be replaced by a real value (a string, number, etc.).
`source code`	Source code, program codes, names of variables and functions, text that has to be typewritten by the user, etc.
`sample`	Sample value within source code (which has to be replaced by a real value). While placeholders are difficult to be used within source code (like XML, HTML) `sample values` are used in these cases.

2 Introduction

2.1 Outline

To enable external systems to remote control functions of Digatron's Battery Manager (BM), an OPC interface is built into the BM. It provides the capability to receive commands (e.g. start or stop a battery formation or test) from an external program or PLC. Furthermore BM publishes status information about its circuits, batteries, program executions, etc. through this OPC interface.

The OPC interface is *not* suitable to exchange mass data (e.g. measurement data, logging data, etc.) since OPC is not designed for this!

Standard OPC access does *not* provide a secure communication ensuring that data has been received properly on the other side. OPC does also *not* provide any kind of handshaking or acknowledgements. All this would have to be implemented into a superior communication *protocol* – which would be no longer part of the OPC standard.

In the following chapters special advices explain how to ensure a proper communication using the OPC interface.

2.2 Architecture

Figure 2-1 shows a typical installation with BM (BM4 Workstation, CommServer, database and OPC client), an OPC server (installed on customer's computer or PLC), customer's applications, etc.

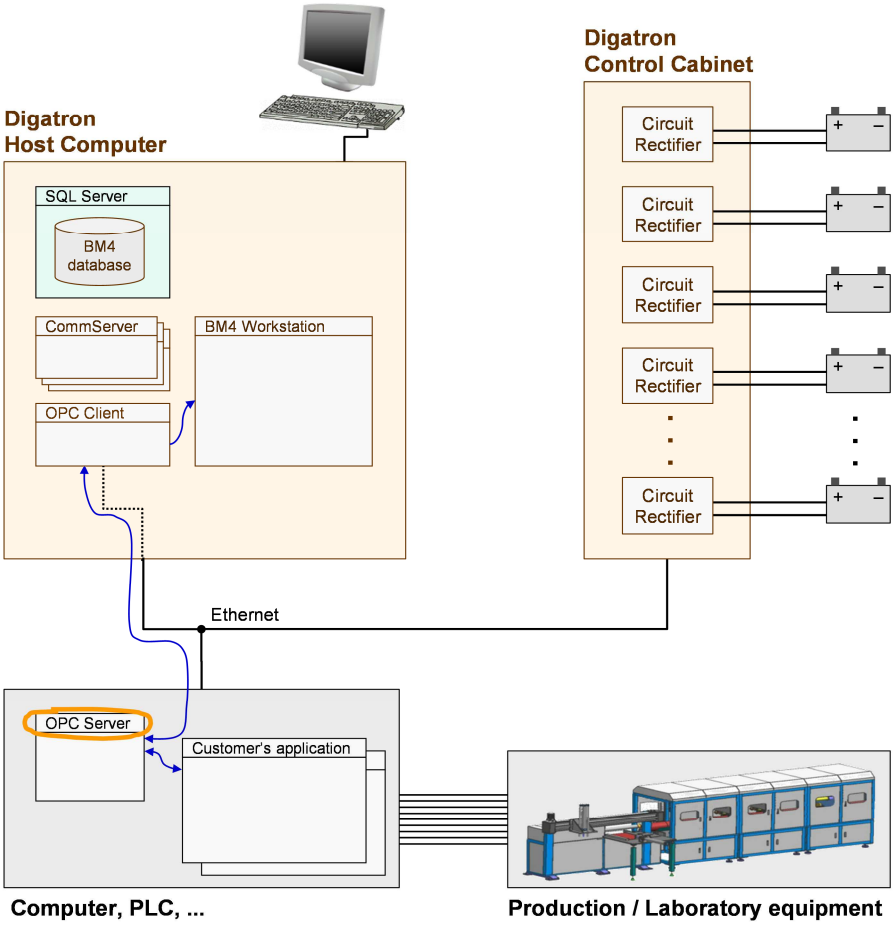

Figure 2-1: Typical installation

The communication between the PLC and the BM is performed through the OPC server. There is no direct interconnection between PLC and the BM. All data and information are exchanged through the OPC server.

The data being exchanged through the OPC server are data points, i.e. values of variables (integers, reals, strings, arrays, etc.) at a given time. Data points on an OPC server are called *OPC items*. Within this document the more general term *data point* will be used.

The Battery Manager does *not* include an OPC server, but an OPC client only.
OPC servers are available as third party products. The OPC server must be provided by the customer (see section 2.5 for details).

2.3 Configuration

The data points to be written to and to be read from the OPC server are defined via configuration of the BM *OPC Client*.

See Appendix B for details on the configuration of the BM *OPC Client*.

2.4 Usage

The OPC Interface allows customers to remote control and supervise the DIGATRON systems, e.g. select programs, start and stop circuits, get state information, etc.

2.5 OPC Server

DIGATRON does *not* provide OPC servers. An OPC server must fit to the appendant hardware (PLC, embedded control, etc.) and thus chosen by the customers.

If possible the OPC server should be installed on the host computer running the BM. The advantage is a highly reduced risk of insufficient rights for accessing the OPC server. If the OPC server is installed on another computer (as sketched in Figure 2-1) the customer must ensure:

- a stable Ethernet connected between the BM host computer and the OPC server computer *and*
- full access rights from the BM host computer to the designated computer and the installed OPC server.

In all cases the OPC server must be chosen, installed, operated and maintained by the customer.

DIGATRON has operating experience with OPC servers from the following companies:

- Siemens S7
- Schneider Telemechanique
- Allen-Bradley/Rockwell
- Matrikon
- Kepware
- CoDeSys

Other products may be supported upon request.

3 Communication elements

The BM *OPC Client* provides the following communication elements:

- *Heart beat* data points
  These data points are used to exchange live signals, indicating whether the respective counterpart is still alive and running.
- *Status* data points
  Status data points are show the current status of circuits. They are written from the BM *OPC Client* into the OPC server and are updated periodically. The update rate can be configured.
- *Command* data points
  Commands can be used from the PLC to invoke action on a circuit (e.g. starting and stopping a circuit). They are written from the PLC into the OPC server and read from the BM *OPC Client*.

ATTENTION
Commands sent from the PLC to the OPC server are active as long as no other command is sent to the circuit. From this a start command might be processed even after a reset on the circuits! It thus has to be overwritten (e.g. by the command 0 / NULL). The sending PLC has to read the status of a circuit and

- if the desired command has been executed *OR*
- if the desired command has not been processed in time

overwrite the existing commands for security reasons.

The following table shows the data types used for the data points:

Data type	Description
I1	Integer, 1 Byte (signed int)
I2	Integer, 2 Byte (signed int)
I4	Integer, 4 Byte (signed int)
R4	Real, 4 Bytes (“float”)
R8	Real, 8 Bytes (“double”)
S	String
A…	Array of one of the above

In principle all data points can have any data type (set via configuration of the data point). De facto the data type must fit the value to be represented (e.g. voltage should be a real value, not an integer).

The following chapters describe all available data points of the BM OPC interface. At the end of each chapter an example for the configuration of the data point (CommServer configuration file) is given.

4 Heart beat data points

4.1 HeartBeatRequest

This data point is written from the PLC and read by the BM *OPC Client*. When the value changes the BM *OPC Client* updates the data point *HeartBeat* (refer to Figure 4-1 and section 4.2).

The frequency of reading the data point *HeartBeatRequest* is set via group configuration (refer to Appendix B). Typical values are in between 100 ms and 1000 ms.

It is recommended that the PLC consecutively increases the value written into the data point *HeartBeatRequest* (i.e. writes a saw tooth signal). With such a signal a tracking and analysis of the Battery Manager reply (see section 4.2) is easier and more precise.

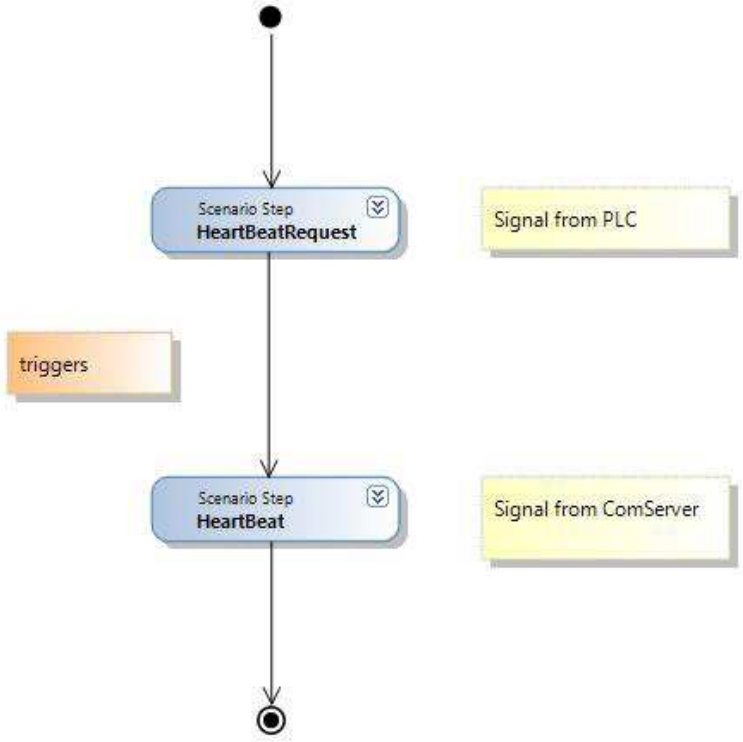

Figure 4-1: Heart beat sequence

Configuration:

```
; HeartBeatRequest needs to be a field of type R8/R4/I4
; value: arbitrary
```

4.2 HeartBeat

This data point shows the value read from *HeartBeatRequest* (see section 4.1). The value of *HeartBeatRequest* is either mirrored or increased by a configurable value (implementation on customer request).

5 Status data points

In this chapter status data points are described. Status data points are written by the BM *OPC Client* into the OPC server.

The frequency of writing the status data points is set via group configuration (refer to Appendix B). Typical values are in between 1 s and 10 min.

Status data points differentiate as follows:

- Circuit status as an aggregation for *all* circuits (see section 5.1 - 5.5)
  The status aggregation represents the status in which the *majority* of circuits are. It thus gives a system overview.
- Circuit status for each individual circuit (see section 5.6 - 5.20)
  These data points represent the status of each individual circuit.

5.1 Offset

This data point transfers values to internal circuit memory (available for special Digatron Tefeu versions only!).

Configuration:

```
;Temperature set value
Value_2="API_etuve_1!C1_Cons_Temp_Digatron"
ValueType_2=R4
ValueFlow_2=write
ValueEnabler_2=1
ValueMap_2=Real1
```

5.2 Status

This data point gives an overview of the status of all circuits. The value is a combination of all currently present status, i.e. a combination of flags. The following table shows the value of all Status.

Value	Description
0	no connection (offline)
1	all circuits in stop
2	all circuits active, in Interrupt or Error state
4	all circuits active, in SYNC state
8	all circuits active, active running in any other state like pause, charge etc.
16	some circuits in stop
32	some circuits active, in Interrupt or Error state
64	some circuits active, in SYNC state
128	some circuits active, active running in any other state like pause, charge etc.
256	some circuits are offline

Example: Some circuits are in Stop (16), some are Active, in Interrupt or Error (32). The value of the Status data point is 48 (16 + 32).

Configuration:

```
;State of circuits
Value_1="API_etuve_1!C1_Digatron_Status"
ValueType_1=R8
ValueFlow_1=write
ValueMap_1=Status
```

5.3 StepNo

This data point gives the current step number of majority of circuits.

Configuration:

```
;Current step number
Value_5="API_etuve_1!C1_Digatron_Pas"
ValueType_5=I4
ValueFlow_5=write
ValueMap_5=StepNo
```

5.4 ProgNo

This data point gives the current program number of majority of circuits.

Configuration:

```
;Current program number
Value_6="API_etuve_1!C1_Digatron_Programme"
ValueType_6=I4
ValueFlow_6=write
ValueMap_6=ProgNo
```

5.5 Table

Database access (select, update, delete, insert) to a customer defined table (external documentation available).

Configuration:

```
Value_1="S7:[S7-Verbindung_1]Group_1.DB22.REAL8"
ValueType_1=AR8
ValueFlow_1=read
ValueMap_1=Table
ValueTable_1=tblCustomOPCData2Keys8Vals
ValueTableCount_1=8
```

5.6 FullStatus

This data point shows the status of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit. The coding of the Status is given in the following table.

Value	Description
0x01	no connection (offline)
0x02	Error
0x04	Stopped
0x08	Paused
0x10	Discharging
0x20	Charging
0x40	Synced
0x80	Interrupted
0x100	Transient
0x200	Unknown
0x400	Slave mode
0x800	Table mode

Limits: Array size of PLC data point.

Configuration:

```
;State of circuits as array for 2 servers ! Yes for two, since it is some special thing for a customer
Value_8="API_etuve_1!Digatron_Statut_word"
ValueType_8=AR8
ValueFlow_8=write
ValueMap_8=FullStatus
```

5.7 FullVoltage

This data point shows the voltage of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit.

Limits: Array size of PLC data point.

Configuration:

```
;Voltage of circuits as array for 2 servers ! See above
Value_9="API_etuve_1!Digatron_Tension"
ValueType_9=AR8
ValueFlow_9=write
ValueMap_9=FullVoltage
```

5.8 FullCurrent

This data point shows the current of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit.

Limits: Array size of PLC data point.

Configuration:

```
;Current of circuits as array for 2 servers ! See above
Value_10="API_etuve_1!Digatron_Courant"
ValueType_10=AR8
ValueFlow_10=write
ValueMap_10=FullCurrent
```

5.9 FullStepNo

This data point shows the step number of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit.

5.10 FullTemperature

This data point shows the step temperature of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit.

5.11 FullRemainingTime

This data point shows the estimated time (in seconds) to end of formation of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit.

ATTENTION

Working of this function depends on correct treatment of Battery Manager Program Editor field “Program Time”!

5.12 FullStartTime

This data point shows the start time of formation of all circuits of one particular CommServer (described by the according INI file). It is an array with each array element representing one circuit.

Important note: Currently only Siemens BCD format is supported!

Configuration:

```
; FullStartTime needs to be an array of R8
; where values are treated as OPC date values
Value_10="API_etuve_1!Digatron_FullStartTime"
ValueType_10=AR8
ValueFlow_10=write
ValueMap_10=FullStartTime
```

5.13 FullBatteryName

This data point shows the battery names of all batteries. It is an array of strings with each array element representing one battery. The values are the battery names.

The size of the array is dependent on the space reserved at PLC-side (in particular ArraySize and StringSize). The StringSize is automatically truncated by OPC. If there is no sufficient space for the complete list the last element (ArraySize) in the list will get the value “<#EOL#>” (indicating an incomplete list)!

The array size is identically to the array size of FullBatteryType and FullBatteryID (see section 5.14 and 5.15).

Since batteries typically don't change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdateBatteries request (see section 6.5).

Configuration:

```
Value_16="PLC1.Application.PLC_PRG.F_BattName"
ValueType_16=AS
ValueFlow_16=write
ValueMap_16=FullBatteryName
ValueArrayLowBound_16=1
ValueArrayHighBound_16=250
ValueGroupName_16=Battery
```

5.14 FullBatteryType

This data point shows the battery types of all batteries. It is an array of strings with each array element representing one battery. The values are the battery types.

The size of the array is dependent on the space reserved at PLC-side (in particular ArraySize and StringSize). The StringSize is automatically truncated by OPC. If there is no sufficient space for the complete list the last element (ArraySize) in the list will get the value "<#EOL#>" (indicating an incomplete list)!

The array size is identically to the array size of FullBatteryType and FullBatteryID (see section 5.13 and 5.15).

Since batteries typically don't change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdateBatteries request (see section 6.5).

Configuration:

```
Value_17="PLC1.Application.PLC_PRG.F_BattType"
ValueType_17=AS
ValueFlow_17=write
ValueMap_17=FullBatteryType
ValueArrayLowBound_17=1
ValueArrayHighBound_17=250
ValueGroupName_17=Battery
```

5.15 FullBatteryID

This data point shows the battery IDs of all batteries. It is an array of integer with each array element representing one battery. The values are the battery IDs.

The size of the array is dependent on the space reserved at PLC-side (in particular ArraySize). If there is no sufficient space for the complete list the last element (ArraySize) in the list will get the value 999999 (indicating an incomplete list)!

The array size is identically to the array size of FullBatteryType and FullBatteryID (see section 5.13 and 5.14).

Since batteries typically don't change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdateBatteries request (see section 6.5).

Configuration:

```
Value_18="PLC1.Application.PLC_PRG.F_BattID"
ValueType_18=AI4
ValueFlow_18=write
ValueMap_18=FullBatteryID
ValueArrayLowBound_18=1
ValueArrayHighBound_18=250
ValueGroupName_18=Battery
```

5.16 FullProgramsString

This data point shows all available programs. It is a string with all program names. For each name a fixed amount of characters is reserved; unused places are filled with space. No additional separators are used!

The number of characters for each program name as well as the size of the whole string is set via configuration.

Since programs typically don't change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdatePrograms request (see section 6.6).

Configuration:

```
Value_21="PLC1.Application.PLC_PRG.F_ProgramName"
ValueType_21=AS
ValueFlow_21=write
ValueMap_21=FullProgramsString
ValueArrayLowBound_21=0
ValueArrayHighBound_21=999
ValueGroupName_21=Program
```

5.17 FullProgramName

AVAILABLE UPON CUSTOMER-REQUEST

This data point shows all available programs. It is an array of strings with each array element representing one program; the values are the program names. The program number can be read with FullProgramNo (see section 5.18), the system this program is according to with FullProgramSystem (see section 5.19).

The size of the array is dependent on the space reserved at PLC-side (in particular ArraySize and StringSize). The StringSize is automatically truncated by OPC. If there is no sufficient space for the complete list the last element (ArraySize) in the list will get the value "<#EOL#>" (indicating an incomplete list)!

Since programs typically don't change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdatePrograms request (see section 6.6).

5.18 FullProgramNo

AVAILABLE UPON CUSTOMER-REQUEST

This data point shows all available programs. It is an array of strings with each array element representing one program; the values are the program numbers. The program names can be read with FullProgramName (see section 5.17), the system this program is according to with FullProgramSystem (see section 5.19).

The size of the array is dependent on the space reserved at PLC-side (in particular ArraySize and StringSize). The StringSize is automatically truncated by OPC. If there is no sufficient space for the complete list the last element (ArraySize) in the list will get the value “<#EOL#>” (indicating an incomplete list)!

Since programs typically don’t change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdatePrograms request (see section 6.6).

5.19 FullProgramSystem

AVAILABLE UPON CUSTOMER-REQUEST

This data point shows all available programs. It is an array of integer with each array element representing one program; the values are the systems the programs are according to. The program names can be read with FullProgramName (see section 5.17), the program number can be read with FullProgramNo (see section 5.18).

The systems are identified by an integer value according to the following list:

```
SYS_UNKNOWN    0x00000000
SYS_BAFOS      0x00000001
SYS_BTS500     0x00000002
SYS_BTS550     0x00000004
SYS_BTS600     0x00000008
SYS_MF2000     0x00000010
SYS_PLT        0x00000020
SYS_MCFT       0x00000040
SYS_FSIII      0x00000080
SYS_FS_SLOT    0x00000100
SYS_CANMF      0x00000200
SYS_MCCD       0x00000400
```

The size of the array is dependent on the space reserved at PLC-side (in particular ArraySize). If there is no sufficient space for the complete list the last element (ArraySize) in the list will get the value 999999 (indicating an incomplete list)!

Since programs typically don’t change every now and then, this data point is automatically updated only after some time (time configurable). An instant update of the data point can be triggered by sending the UpdatePrograms request (see section 6.6).

5.20 FullValueEx

This data point shows a specific channel at all circuits. It is an array of integer or double with each array element representing one circuit; the values are the current values of the specified channel.

The channel is set via configuration (config-variable ValueChannelType/Number; see example below).

The array size (i.e. number of circuits) and the circuits to be represented in the array is also set via configuration (config-variables ValueArrayLowBound and ValueArrayHighBound; see example below).The maximum number of array elements (i.e. circuits) depend only on PLC array size.

Configuration:

```
;Channel value of circuits as array
Value_15="PLC1.Application.PLC_PRG.F_Capacity"
ValueType_15=AR4
ValueFlow_15=write
ValueMap_15=FullValueEx
ValueArrayLowBound_15=1
ValueArrayHighBound_15=204
ValueChannelType_15=7
ValueChannelNumber_15=1
ValueGroupName_15=Status
```

6 Command data points

In this chapter the *command* data points are described. Command data points are read by the BM *OPC Client* from the OPC server. The only exception is the *CommandExecutionReply* data point (see section 6.1) which is written from the BM *OPC Client* into the OPC server.

The frequency of reading the command data points is set via configuration. Typical values are in between 1 s and 5 s.

6.1 CommandExecutionReply

This data point shows the status of command executions. It is an array of integer or double with each array element representing one circuit result. Each time the BM *OPC Client* has read and executed a command (refer to section 6.2, 6.3 and 6.4), the resulting status of this action is written into this data point.

ATTENTION

This data point shows the status of the *most recent* command being read and executed.

It is up to the customer's program to store which command was sent at last. If a history of commands and their execution status is required, it is also up to the customer's program to record and store such.

The value of the data point is a numerical value according to the following table:

Value	Description
0	Last command read and executed without any errors.
1	Invalid command
10	Invalid battery ID
11	Invalid battery name
12	Invalid battery type
13	Missing battery ID
14	Both battery name and type have to be set for this system
20	Invalid program number
21	Invalid program name
22	Inconsistent program name and number
23	Battery Type has no default program
30	Invalid lower or upper circuit number
31	Lower circuit number is greater than upper circuit number
40	Invalid circuit or dispo entry
41	Circuit is blocked
42	Circuit cannot start because of faulty dispo
43	Circuit already running

50	Command ignored
51	Invalid StartFromStep
70	Syntax error in program
71	Compiled program too big
72	Program load failed

Configuration:

```
Value_4="PLC1.Application.PLC_PRG.F_CmdReplyI"
ValueType_4=AI2
ValueFlow_4=write
ValueMap_4=CommandExecutionReply
ValueArrayLowBound_4=1
ValueArrayHighBound_4=204
ValueGroupName_4=Status
```

6.2 Command

This data point provides the possibility to start, stop, interrupt and continue circuits. It is an array of four DWord (4-byte integer) with the following structure and usage:

Element	Usage	
1	Command:	0 (NULL), 1 (Start), 2 (Break), 3 (Stop), 4 (Continue)
2	BatteryID:	ID of a battery (see section 5.15 for details)
3	ProgramNo:	The program number, defined within the BM
4	Circuits:	4 · 8 Bit = 32 Bits for 32 circuits Bit = 0 ⇒ no command send for the according circuit Bit = 1 ⇒ command (see #1) is send to the circuit

Error checking

Before executing a command, the following is checked:

- *Command*
  If *Command* has an invalid value an error will be set.
- *BatteryID*
  - If *BatteryID* has a negative or invalid value (non-existing battery) an error will be set.
  - If *BatteryID* is 0 (zero) and the connected system requires an existing battery (e.g. BTS600) an error will be set.
- *ProgramNo*
  - If *ProgramNo* is positive but an invalid value (non-existing program) an error will be set.
  - If *ProgramNo* has a negative value the following will be checked:
    If *BatteryID* is a valid ID *and* the according battery has a battery type (defined in BM) *and* for this battery type a default program has been defined (within BM), then this default program will be used (see below) – otherwise an error will be set.

- In case of an error the according error number will be written into the *CommandExecutionReply* data point (see section 6.1). No further actions will be executed (no circuit will be changed).

Command execution

If the command is without any error it will be executed. The following applies:

- *ProgramNo*
  - If *ProgramNo* has a valid value the corresponding program will be used.
  - If *ProgramNo* has a negative value the following will be checked:
    If *BatteryID* is a valid ID *and* the according battery has a battery type (defined in BM) *and* for this battery type a default program has been defined (within BM), then this default program will be used (see below).
- The 4th element (Circuits) is interpreted as a map of 4 · 8 = 32 flags (4 bytes, each byte = 8 bits), representing the first 32 circuits of the corresponding rectifiers/s.
  The command is thus limited to 32 circuits.
- After execution of the command an "OK" will be written into the *CommandExecutionReply* data point (see section 6.1).

After processing a command it is recommended to observe whether circuit(s) move into the new state and stay in the desired state.

Commands sent from PLC to the OPC server are active as long as no other command is written into the OPC data point. If the command is not removed after the command was executed, the command will be executed again and again (in case of CommServer restart scenarios)!

As an example: If a circuit was started and the command was not removed, the circuits will be directly restarted as soon as the program on that circuit has finished. Even after a *reset* or after a *restart* of the according CommServer the circuit will be directly started again!

To prevent such, the PLC *must* operate as follows:

1. Write desired command into the command data point
2. Observe the according circuits (refer to *FullStatus*, see section 5.6)
   As soon as
   - circuits are in the desired state *OR*
   - command has not been processed within the expected time (timeout error)

   remove the command, i.e. write *0* (*NULL*) into the data point.

Configuration:

```
; Command needs to be an array of 4* R8/R4/I4
Value_7="API_etuve_1!Digatron_Command_Socket"
ValueType_7=AI4
ValueFlow_7=read
ValueMap_7=Command
ValueGroupName_7=Command
```

6.3 Command2

This data point provides the possibility to start, stop, interrupt and continue circuits. It is similar to *Command* (see chapter 0), the only difference is the type of circuits-definition.

Command2 is an array of five DWord (4-byte integer) with the following structure and usage:

Element	Usage	
1	Command:	0 (NULL), 1 (Start), 2 (Break), 3 (Stop), 4 (Continue)
2	BatteryID:	ID of a battery (see section 5.15 for details)
3	ProgramNo:	The program number, defined within the BM
4	LowerCircuitNo:	Circuit number (not the circuit ID)
5	UpperCircuitNo:	Circuit number (not the circuit ID)

Error checking

Before executing a command, the following is checked:

- *Command*
  If *Command* has an invalid value an error will be set.
- *BatteryID*
  - If *BatteryID* has a negative or invalid value (non-existing battery) an error will be set.
  - If *BatteryID* is 0 (zero) and the connected system requires an existing battery (e.g. BTS600) an error will be set.
- *ProgramNo*
  - If *ProgramNo* is positive but an invalid value (non-existing program) an error will be set.
  - If *ProgramNo* has a negative value the following will be checked:
    If *BatteryID* is a valid ID *and* the according battery has a battery type (defined in BM) *and* for this battery type a default program has been defined (within BM), then this default program will be used (see below) – otherwise an error will be set.
- *LowerCircuitNo* and *UpperCircuitNo*
  - If *LowerCircuitNo* and/or *UpperCircuitNo* have negative values *OR*
    if *LowerCircuitNo* and/or *UpperCircuitNo* have invalid values *OR*
    if *LowerCircuitNo* is greater than *UpperCircuitNo*
    then an error will be set.
- In case of an error the according error number will be written into the *CommandExecutionReply* data point (see section 6.1). No further actions will be executed (no circuit will be changed).

Command execution

If the command is without any error it will be executed. The following applies:

- *ProgramNo*
  - If *ProgramNo* has a valid value the corresponding program will be used.
  - If *ProgramNo* has a negative value the following will be checked:
    If *BatteryID* is a valid ID *and* the according battery has a battery type (defined in BM) *and* for this battery type a default program has been defined (within BM), then this default program will be used (see below).
- *LowerCircuitNo* and *UpperCircuitNo*

- If *LowerCircuitNo* and *UpperCircuitNo* are equal this particular circuit is operated.
- If *LowerCircuitNo* and *UpperCircuitNo* specify a range, all circuits within this range are operated.

- After execution of the command an "OK" will be written into the *CommandExecutionReply* data point (see section 6.1).

After processing a command it is recommended to observe whether circuit(s) move into the new state and stay in the desired state.

Commands sent from PLC to the OPC server are active as long as no other command is written into the OPC data point. If the command is not removed after the command was executed, the command will be executed again and again (in case of CommServer restart scenarios)!

As an example: If a circuit was started and the command was not removed, the circuits will be directly restarted as soon as the program on that circuit has finished. Even after a *reset* or after a *restart* of the according CommServer the circuit will be directly started again!

To prevent such, the PLC *must* operate as follows:

1. Write desired command into the command data point
2. Observe the according circuits (refer to *FullStatus*, see section 5.6)
   As soon as
   - circuits are in the desired state *OR*
   - command has not been processed within the expected time (timeout error)

   remove the command, i.e. write *0* (*NULL*) into the data point.

Configuration:

```
; Command2 needs to be an array of 5*R8/R4/I4
Value_7="API_etuve_1!Digatron_Command_Socket"
ValueType_7=AI4
ValueFlow_7=read
ValueMap_7=Command2
ValueGroupName_7=Command
```

6.4 Command3

This data point provides the possibility to start, stop, interrupt and continue circuits. It is similar to *Command* (see section 6.2) and *Command2* (see section 6.3). The difference is that *Command3* provides more flexibility and features.

Command3 is an array of seven to nine *strings* with the following structure and usage:

Element	Usage	
1	Command:	0 (NULL), 1 (Start), 2 (Break), 3 (Stop), 4 (Continue)
2	BatteryName:	*Name* of a battery (see section 5.13 for details)
3	BatteryType:	*Type* of a battery (see section 5.14 for details)
4	ProgramNo:	The program number, defined within the BM
5	ProgramName:	The program name, defined within the BM

6	LowerCircuitNo:	Circuit number (not the circuit ID)
7	UpperCircuitNo:	Circuit number (not the circuit ID)
8 *	OrderNo:	Serial number inserted in Session header
9 *	StartFromStep:	Starting step of program (*not* supported for all circuit types!)

Elements marked with *: Optional element; might be empty or even non-existent!

Error checking

Before executing a command, the following is checked:

- *Command*
  If *Command* has an invalid value an error will be set.
- *BatteryName* and *BatteryType*
  - If *BatteryName* has an invalid value (non-existing battery) an error will be set.
  - If *BatteryName* is empty and the connected system requires an existing battery (e.g. BTS600) an error will be set.
  - If *BatteryType* has an invalid value (non-existing battery type) an error will be set.
    If both *BatteryName* and *BatteryType* are defined, but *BatteryType* is not identical to the battery type defined in BM for the given *BatteryName* an error will be set.
- *ProgramNo* and *ProgramName*
  - If *ProgramNo* has an invalid value (non-existing program) an error will be set.
  - If *ProgramName* has an invalid value (non-existing program) an error will be set.
  - If *ProgramNo* and *ProgramName* are not empty but don't match an error will be set.
  - If *ProgramNo* and *ProgramName* are empty the following will be checked:
    If *BatteryName* is given but no battery type defined for this battery *OR*
    if *BatteryName* is given, a battery type is defined, but no default program for this battery type *OR*
    if *BatteryName* is empty and *BatteryType* is empty too *OR*
    if *BatteryName* is empty, *BatteryType* is given, but no default program for this battery type
    then an error will be set.
- *LowerCircuitNo* and *UpperCircuitNo*
  - If *LowerCircuitNo* and/or *UpperCircuitNo* have negative values *OR*
    if *LowerCircuitNo* and/or *UpperCircuitNo* have invalid values *OR*
    if *LowerCircuitNo* is greater than *UpperCircuitNo*
    then an error will be set.
- In case of an error the according error number will be written into the *CommandExecutionReply* data point (see section 6.1). No further actions will be executed (no circuit will be changed).

Command execution

If the command is without any error it will be executed. The following applies:

- *ProgramNo*
  - If *ProgramNo* and/or *ProgramName* have a valid value then this program will be used.
  - If *ProgramNo* and *ProgramName* are empty the following will be checked:
    If *BatteryName* is valid *and* the according battery has a battery type (defined in BM) *and* for this battery type a default program has been defined (within BM) *OR*
    if *BatteryType* is valid *and* for this battery type a default program has been defined (within BM),

then this default program will be used (see below).

- *LowerCircuitNo* and *UpperCircuitNo*
  - If *LowerCircuitNo* and *UpperCircuitNo* are equal this particular circuit is operated.
  - If *LowerCircuitNo* and *UpperCircuitNo* specify a range, all circuits within this range are operated.
- After execution of the command an “OK” will be written into the *CommandExecutionReply* data point (see section 6.1).

After processing a command it is recommended to observe whether circuit(s) move into the new state and stay in the desired state.

Commands sent from PLC to the OPC server are active as long as no other command is written into the OPC data point. If the command is not removed after the command was executed, the command will be executed again and again (in case of CommServer restart scenarios)!

As an example: If a circuit was started and the command was not removed, the circuits will be directly restarted as soon as the program on that circuit has finished. Even after a *reset* or after a *restart* of the according CommServer the circuit will be directly started again!

To prevent such, the PLC *must* operate as follows:

1. Write desired command into the command data point
2. Observe the according circuits (refer to *FullStatus*, see section 5.6)
   As soon as
   - circuits are in the desired state *OR*
   - command has not been processed within the expected time (timeout error)

   remove the command, i.e. write *0* (*NULL*) into the data point.

Configuration:

```
Value_3="PLC1.Application.PLC_PRG.F_Command3"
ValueType_3=AS
ValueFlow_3=read
ValueMap_3=Command3
ValueGroupName_3=Command
```

6.5 UpdateBatteries

This data point triggers the update mechanism of the data points *FullBatteryName*, *FullBatteryType* and *FullBatteryID* (see section 5.13, 5.14 and 5.15). It is a scalar type R8, R4 or I4 with the values 0 or 1.

The *OPC Client* triggers on a rising edge 0 ↗ 1:
When the PLC writes a 1 (or more precisely: any value except 0) into *UpdateBatteries*, the BM *OPC Client* will write the current values into *FullBatteryName*, *FullBatteryType* and *FullBatteryID* (refer to Figure 6-1). Afterwards the PLC has to write a 0 (zero) back into the data point *UpdateBatteries*.

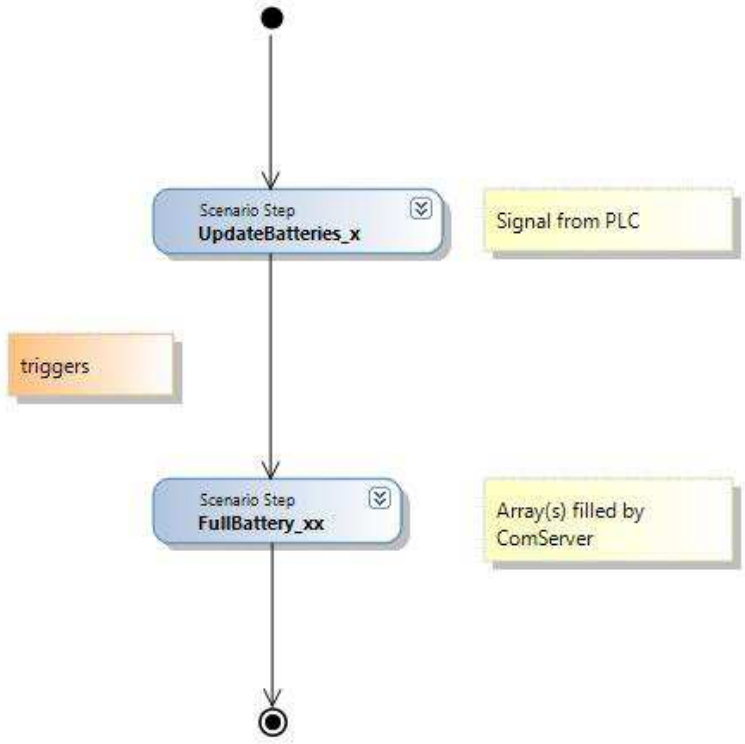

Figure 6-1: UpdateBatteries sequence

Configuration:

```
; where content is 0=NULL, 1=Trigger update
Value_20="PLC1.Application.PLC_PRG.F_BattTrigger"
ValueType_20=I4
ValueFlow_20=read
ValueMap_20=UpdateBatteries
ValueGroupName_20=Command
```

6.6 UpdatePrograms

This data point triggers the update mechanism of the data points *FullProgamsString*, *FullProgramName*, *FullProgramNo* and *FullProgramSystem* (see section 5.16 - 5.19). It is a scalar type R8, R4 or I4 with the values 0 or 1.

The *OPC Client* triggers on a rising edge 0 ↗ 1:
When the PLC writes a 1 (or more precisely: any value except 0) into *UpdatePrograms*, the BM *OPC Client* will write the current values into *FullProgamsString*, *FullProgramName*, *FullProgramNo*

and *FullProgramSystem* (refer to Figure 6-2). Afterwards the PLC has to write a 0 (zero) again into *UpdatePrograms*.

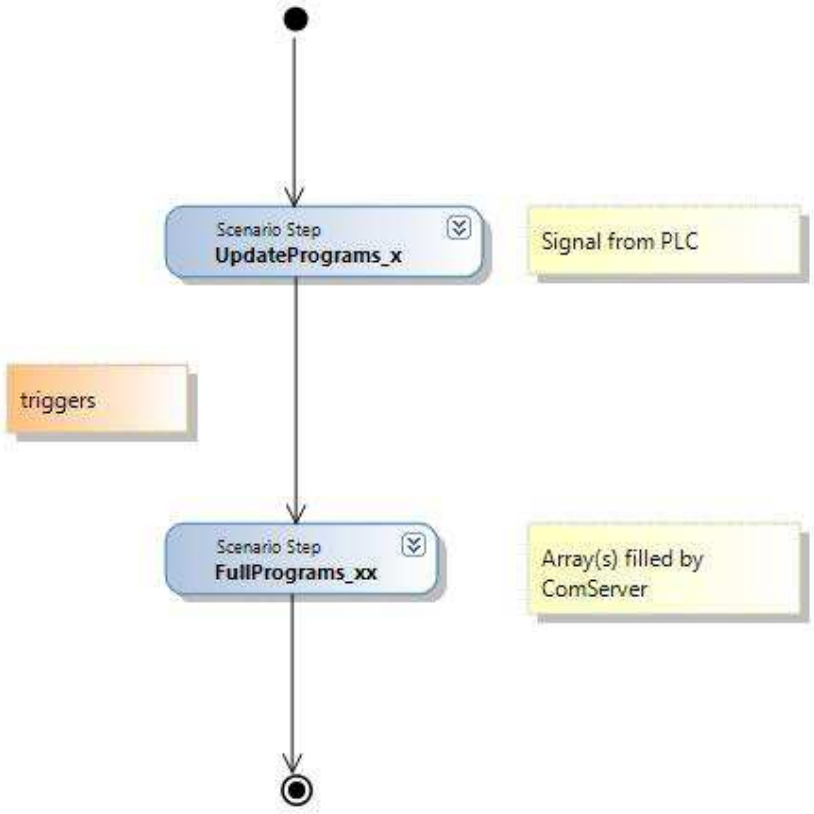

Figure 6-2: UpdatePrograms sequence

Configuration:

```
; where content is 0=NULL, 1=Trigger update
Value_10="PLC1.Application.PLC_PRG.F_ProgramTrigger"
ValueType_10=I4
ValueFlow_10=read
ValueMap_10=UpdatePrograms
ValueGroupName_10=Command
```

6.7 ScanInput, ScanResponse, ScanInsertResponse

The data points ScanInput, ScanResponse and ScanInsertResponse support undocumented features (customer-specific functionalities).

Configuration:

```
Value_8="[Formation_electrique].Code_Element1"
ValueType_8=S
ValueFlow_8=read
ValueMap_8=ScanInput

Value_9="[Formation_electrique].Scan_response1"
ValueType_9=S
ValueFlow_9=write
ValueMap_9=ScanResponse

Value_10="[Formation_electrique].Insert_response1"
ValueType_10=S
ValueFlow_10=write
ValueMap_10=ScanInsertResponse
```

7 Possible extensions

In this chapter possible extensions to the command communication elements are listed.

7.1 UpdateProgramsFilterBatteryID *AVAILABLE UPON CUSTOMER-REQUEST*

This data point triggers the update mechanism for the data points *FullProgamXxx* (refer to the section 5.16 - 5.19). It is an array of R8, R4 or I4. The first array element stores the trigger value, the second array element specifies the battery ID.

The *OPC Client* triggers on a rising edge 0 ↗ 1:
When the PLC writes a 1 (one) into *UpdateProgramsFilterBatteryID*[1], the BM *OPC Client* will write the current values into *FullProgamsString*, *FullProgramName*, *FullProgramNo* and *FullProgramSystem* (refer to Figure 6-2). Afterwards the PLC has to write a 0 (zero) again into *UpdateProgramsFilterBatteryID*.

Configuration:

```
; UpdateProgramsFilterBatteryID needs to be an array of type 2*R8/R4/I4
; where 1: is 0=NULL, 1=Trigger update
;       2: batteryID to query
```

7.2 UpdateProgramsFilterBatteryName *AVAILABLE UPON CUSTOMER-REQUEST*

This data point triggers the update mechanism for the data points FullProgamXxx (refer to the sections 5.16, 5.17, 5.18 and 5.19). It is an array of R8, R4 or I4. The first array element stores the trigger value, the second array element specifies the battery name.

The *OPC Client* triggers on a rising edge 0 ↗ 1:
When the PLC writes a 1 (one) into *UpdateProgramsFilterBatteryName*[1], the BM *OPC Client* will write the current values into *FullProgamsString*, *FullProgramName*, *FullProgramNo* and *FullProgramSystem* (refer to Figure 6-2). Afterwards the PLC has to write a 0 (zero) again into *UpdateProgramsFilterBatteryName*.

Configuration:

```
; UpdateProgramsFilterBatteryName needs to be an array of type 2*string
; where 1: is 0=NULL, 1=Trigger update
;       2: battery name to query
```

Appendix A Definitions and Abbreviations

Term / Abbreviation	Explanation
BM	Battery Manager
OPC	OLE for Process Control
OPC client	A computer program reading and writing data from/to an OPC server
OPC server	A server providing the possibility to exchange plant data between control devices from different manufacturers

Appendix B OPC Configuration

The configuration of the BM *OPC Client* is done in the respective CommServer configuration files (e.g. “RailPcComSvr.custom.ini”).

Within the INI file the following three sections are of importance:

- OPC
- OPCGroups
- OPCValues

Appendix B.1 Section OPC

The following code shows the default settings for the configuration section “OPC”. Normally only the keys *HostName* and *ServerName* must be adjusted.

```
[OPC]
;local server=empty
;remote server=192.168.0.72
HostName=192.168.9.1

;server name e.g. like "Matrikon.OPC.Simulation.1"
ServerName=CoDeSys.OPC.DA

;this gives an extensive log file with access, reads and writes
TraceToFile=1
ContinuousUpdate=0
;set BrowseServer to 1 to see ALL known elements of that OPC server
BrowseServer=0
LogStartup=1
BulkWrite=1

;NullValue defines a value which is used to indicate NULL values for
doubles in database
NullValue=-1
```

Appendix B.2 Section OPCGroups

The section *OPCGroups* defines groups of variables with different update scenarios (e.g. write/read frequency).

The keys must be enumerated consecutively! Start with “_1” and increment by one for each new group name. Otherwise the system will react with unpredictable results!

```
[OPCGroups]
// UpdateRate < 100 -> seconds >= 100 -> milliseconds
GroupName_1=Status
GroupAsync_1=1
GroupUpdateRate_1=5

GroupName_2=Program
GroupAsync_2=1
GroupUpdateRate_2=1200000

GroupName_3=Command
GroupAsync_3=1
GroupUpdateRate_3=1000

GroupName_4=Puls
GroupAsync_4=1
GroupUpdateRate_4=300

GroupName_5=Battery
GroupAsync_5=1
GroupUpdateRate_5=1100000
```

Appendix B.3 Section OPCValues

In the following example only FullStatusEx is activated. The remaining parts are only for reference.

The keys must be enumerated consecutively! Start with "_1" and increment by one for each new group name. Otherwise the system will react with unpredictable results!

```
[OPCValues]
Value_1="PLC1.Application.PLC_PRG.F_Status"
ValueType_1=AI2
ValueFlow_1=write
ValueMap_1=FullStatusEx
ValueArrayLowBound_1=1
ValueArrayHighBound_1=204
ValueGroupName_1=Status

Value_3="PLC1.Application.PLC_PRG.F_Command3"
ValueType_3=AS
ValueFlow_3=read
ValueMap_3=Command3
ValueGroupName_3=Command

Value_4="PLC1.Application.PLC_PRG.F_CmdReplyI"
ValueType_4=AI2
ValueFlow_4=write
ValueMap_4=CommandExecutionReply
ValueArrayLowBound_4=1
ValueArrayHighBound_4=204
ValueGroupName_4=Status

;OPC name in flat mode e.g. like "Formation.Chambre3.Temperature1.Real8"
;type of value: allowed R4,R8,I1,I2,I4,S and Array version AR8,AI4,AS
;An A in front of type means ARRAY of type e.g. AR4 is an ARRAY of Real4
;direction of data transfer: allowed read, write
;read means FROM OPC server
```

```
;write means TO OPC server
;enable var to allow writing to OPC = offset in XION digital out (1-32)
;map value to internal XION array (1-32) or {Status, StepNo, ProgNo}
;Temperature set value
Value_2="API_etuve_1!C1_Cons_Temp_Digatron"
ValueType_2=R4
ValueFlow_2=write
ValueEnabler_2=1
ValueMap_2=Real1
```

The following table gives an overview about the usage of special values for definition of data items.

Name	Enabler	Table	Tablesize	ChanType	ChanNum	Remarks
Command	X	-	-	-	-	EnableOffset = count of circuits to start
Command2	-	-	-	-	-	
Command3	-	-	-	-	-	
CommandExecReplyEx	-	-	-	-	-	
FullBatteryID	-	-	-	-	-	
FullBatteryName	-	-	-	-	-	
FullBatteryType	-	-	-	-	-	
FullCurrent	-	-	-	-	-	
FullCurrentEx	-	-	-	-	-	
FullProgramsLinesEx	-	-	-	-	-	
FullProgramsNoEx	-	-	-	-	-	
FullProgramTimeEx	-	-	-	-	-	
FullProgramsLines	-	-	-	-	-	
FullProgramsString	X	-	-	-	-	create "space" separated list of programs in one string LowBound = width of single string HighBound = count of strings(beginning from 0) EnableOffset = count of spaces in front of string
FullRemainingTimeEx	-	-	-	-	-	
FullStartTimeEx	-	-	-	-	-	Only Siemens BCD format supported
FullStatus	-	-	-	-	-	
FullStatusEx	-	-	-	-	-	
FullStepNoEx	-	-	-	-	-	
FullStepTimeEx	-	-	-	-	-	
FullTemperatureEx	-	-	-	-	-	
FullValueEx	-	-	-	X	X	ChannelType and Number from tblCircuit
FullVoltage	-	-	-	-	-	
FullVoltageEx	-	-	-	-	-	
HeartBeat	-	-	-	-	-	
HeartBeatRequest	-	-	-	-	-	
ProgNo	-	-	-	-	-	
ScanInput	-	-	-	-	-	
ScanInsertResponse	-	-	-	-	-	
ScanResponse	-	-	-	-	-	
Status	-	-	-	-	-	
StatusErrorSft	X	-	-	-	-	Enabler = serverID of circuits Bounds = Address range of circuits
StepNo	-	-	-	-	-	
Table	-	X	X	-	-	The real table name in database: ValueTable_1 = tblCustomOPCData2Keys8Vals ValueTableCount_1 = 8
UpdateBatteries	-	-	-	-	-	
UpdatePrograms	-	-	-	-	-	