EDY

GAT-1 / GAT-2

REMOTE CONTROL & PAGING
VIA MOBILE TELEPHONY NETWORK
& AUTOMATION CONTROLLER

PROGRAMMING MANUAL
for the firmware “GAT-x ver.1.2”

EN rev: 1.0, 2022-06-10

Legal Content Notice:

The company 'EDY electronics Ltd' follows a continuous development policy of its products and therefore reserves the right to make various alterations & improvements upon the goods described in this current document without any prior notice.

The use of GAT-1/ GAT-2 devices is at the sole discretion of the user. 'EDY electronics LTD' company cannot be held responsible for any loss of data, income, or whatsoever special, deposit, ancillary or indirect damage it may be caused by the (mis)use of the device.

Table of Contents

1: COMMUNICATING WITH THE GAT DEVICE, BASIC RULES & TERMINOLOGY

The GAT device firmware (**GAT-FW** for short) contains a set of features that can be activated and configured to cover the majority of remote control / paging cases.

Each function is activated and set with one or more **commands**, which are sent to the device with a **message**.

A message consists of Latin characters, numbers and symbols.

To be accepted by the device, each message must begin with a four-digit **Security Code**.

The default security code is "**0000**".

The security code is always followed by a blank character and at least one command with its **parameters**, if any.

Multiple commands can be sent in one message.

Commands are combinations of 2 or more uppercase Latin characters, which are initials or abbreviations of English words.

A simple example of a message sent to the device (remote control command, immediate output 1 activation) is:

```
0000 OA 1
```

In the example above the security code is "**0000**", the command is **OA** (initials of the English words "Output Activate") and the parameter that follows is the number **1** (index of first output).

Multiple commands can be sent in one message. A simple example of remote controlling could be:

```
0000 OA 1 OA 2
```

With this message there will be a simultaneous activation of outputs 1 and 2.

Space and line-change (NEW-LINE) characters are used to separate the elements of a message (i.e., the code, commands and parameters), while there is freedom to be used alternatively for the purpose of readability of the text.

The previous example could also be written:

```
0000
OA 1
OA 2
```

The **command parameters**:

- When it comes to **numbers** they consist of decimal digits (**0..9**).
- When it comes to **strings**, i.e., text, these are Latin characters (uppercase or lowercase), decimal places, spaces and symbols (+, -,!,%, etc.), except for the quotation mark ("), which is used to define the sequence boundaries.

The **""** sequence (two quotes) contains zero characters, so when displayed as a parameter defining a message, it means "**no message**".

For example, observe the following message:

```
0000 IM 1 1 "ALARM !!!" ""
```

Here, after the security code "0000" there is also the command "IM" ("Input event Message"),

which defines the messages the device will send to change the binary status of a voltage input.

The 4 parameters that follow the command are:

- The Input index **1** (first voltage input).
- Recipient index **1**, which means that the message will be sent to the first telephone number in the phone list (see **TL**).
- The character sequence **"ALARM !!!"** is the message that will be sent (without the quotation marks) when the input is activated.
- The empty character sequence **""**, which means no message will be sent the the inputs is deactivated.

Each time the device receives a message that starts with a valid security code followed by at least one space (or line break) character, it starts executing the commands sequentially, provided they are spelled correctly.

In case of error, the execution of the commands is stopped instantly, i.e., the rest of the message is rejected.

The device responds to the sender of the message with a confirmation or an error message (initial setting, see **CM** command).

For example, after receiving the message

```
0000 OA 1
```

the device will activate output 1 and respond to the sender [of the message***] with the confirmation message:

```
O.K. OA 1
```

The above description of messages and communication of the device does not only concern communication via SMS but also through the USB port,
when using the support software "**GAT communicator**" ("**GATcomm**" for short).

The **GAT-FW**, in addition to the command mechanism presented above, also supports the execution of an **automation program**.

The automation program ("**GAT Automation Program**", **GAT-AP** for short) is a structured programming language, which exists to meet the needs for the implementation of local automation, as well as the implementation of communication functions that are not contained ready-made in **GAT-FW**.

The **GAT-AP** language is described in detail in Chapter **4: GAT-AP AUTOMATION PROGRAM**.

2: DATA REPRESENTATION / TEXT FORMAT

In any text that can be sent via the device, some special characters can be inserted which will be instantly translated by the device software and replaced with data, text formatting or special characters.

For this to happen, we must enter the character "**%**" (percentage sign) followed by some characters that are the **description** of the data, so to appear in the preferred position in the text.

The **GAT-FW** understands the following descriptions:

2.1: %An : Representation of Analog Input value

Syntax: %An
n: [1..8], Input index.

Comment: Representation of the Analog Input value in the form of a decimal number.

Example: If the device sends the text "**A1 =% A1**" and the input value is 123 (1.23V), the recipient will receive a message that looks like:

```
A1 = 123
```

2.2: %In : Representation of a Binary Input value

Syntax: %In
n: [1..8], Input index.

Comment: Representation of the input binary value in the form of a digit '0' or '1'.

Example: If the device sends the text "**I1 =% I1**" and the input binary status is 0, the recipient will receive a message that looks like:

```
I1 = 0
```

2.3: %On : Representation of Binary Output state

Syntax: %On
n: [1..4], Output index.

Comment: Representation of the output binary value in the form of digit '0' or '1'.

Example: If the device sends the text "**O1 =% O1**" and output 1 is off, the recipient will receive a message that looks like:

```
O1 = 0
```

2.4: %Bn : Representation of Byte type Variable value

Syntax: %Bn
n: [1..32], Variable index.

Comment: Representation of a **Byte** type variable value in decimal format.

Example: If the device sends the text "**B1 =% B1**" and the value of variable **B1** is **100**, the recipient will receive a message that looks like:

```
B1 = 100
```

2.5: %Nn : Representation of a Number type Variable value

Syntax: %Nn
n: [1..64], Variable index.

Comment: Representation of the value of variable type Number, in the form of a decimal number.

Example: If a device sends the text "**N1 = %N1**" and the value of the variable **N1** is **1000**, the recipient will receive a message that looks like:

```
N1 = 1000
```

2.6: %f : Representation of a Variable value as floating point number

Syntax: %f[dp]Xn

[dp]: [1..4], Floating Point position.
This is optional and if it does not exist then it equals **2**,
i.e., a value of **1000** will be represented as "**10.00**"

Xn: Variable name. It may be:
Analog Input value "**A1**"..."**A8**",
Byte type variable value "**B1**".."**B32**" or
Number type variable value "**N1**".."**N64**".

Comment: The **GAT-AP** language does not support floating point numbers and all operations are performed in the field of integers.

The **%f** description is used to represent an integer in floating point format, which in some cases may be more understandable to the end user.

Examples:

1) If the voltage at input **1** is **4.75V** (**A1=475**) and the device sends the text "**Input # 1: %fA1 V**", the recipient will receive a message like this:

```
Input #1: 4.75 V
```

2) If the value of variable **N10** is **145234** and the device sends the text **"Result: %f4N10"**, the recipient will receive a message like this:

```
Result: 14.5234
```

2.7: %D : Representation of Date

Syntax: %D

Comment: Representation of the date in the format **YYYY/MM/DD**, where **YYYY** is the year, **MM** is the month and **DD** is the day of the month.

Example: If the device sends the text "**The date is: %D**", the recipient will receive a message like this:

```
The date is: 2020/06/15
```

2.8: %T : Representation of Time

Syntax: %T

Comment: Representation of the time in the format **HH:MM:SS**, where **HH** is the time, **MM** is the minutes and **SS** is the seconds.

Example: If the device sends the text "**The time is: %T**", the recipient will receive a message like this:

```
The time is: 18:35:12
```

2.9: %n : Line Break

Syntax: %n

Comment: If "**%n**" is present in a text, it is replaced by a line break at that point.

Example: If the device sends the text "**First Line%nSecond Line**", the recipient will receive a message like this:

```
First Line
Second Line
```

2.10: %% : The character "%"

Syntax: %%

Comment: If "**%%**" is present in a text, it is replaced with the character "**%**" at that point.

Example: If the device sends the text "**100%% True**",
the recipient will receive a message like this:

```
100% True
```

3: COMMANDS UNDERSTOOD BY THE GAT-FW

The commands described in the following chapters work on both types of the device, GAT-1 & GAT-2. There are differences in the maximum values of some parameters, when they refer to an input or output index number.

Commands are divided into the following categories:

- **Functional Parameters** setup commands. These commands configure all functions of the device. As soon as the device receives such a command, it saves the settings and operates according to them, from there on.
- **Immediate Execution** commands. These include e.g., the remote control commands. Their characteristic is that they cause some action (e.g. activation of output) the moment they are received by the device.
- **Information Query** commands. These are questions to the device, which cause an immediate answer.

3.1: FUNCTIONAL PARAMETERS SETUP COMMANDS

These commands configure the parameters of the following functions:

- Communication
- Voltage inputs
- Relay outputs
- Automation with telephone calls
- Special events management

3.1.1: SC: Security Code Setup

Syntax: SC n1 n2

n1: [0000..9999], Four-digit Security Code.

n2: Same as **n1**, for verification.

Default value: 0000

Comment: With this command we program a new security code, so that the device becomes inaccessible to anyone who does not know it.

ATTENTION: the password can only be changed by someone who knows it, or return to the initial value 0000 with general parameters initialization via the "reset" button, where all settings are initialized and the automation program is erased.

Example: To set the security code to **5678**, one must send the message:

```
0000 SC 5678 5678
```

Directly after this message, the device will only react to messages starting with the code **5678**.

3.1.2: TN : Telephone Numbers List

Syntax: TN tn1 ... tn8

tn1 ... tn8: List of **0** to **8** telephone numbers, up to **16** digits each.

Default value: The phone number list is empty.

Comment: Define a list of users with 0 to 8 phone numbers.

The list is used to identify or access users through their index number in this list.
Index numbers range from 1 to 8.

“Users” (or “clients”) are those who receive messages, when some device functions are activated, and these are sent as a response to events, such as changing the status of an input, restarting the system and more. The same telephone numbers are also used in telephone calling functions.

Each time this command is sent, the previous phone numbers are deleted.

Example: To define **3** users with telephone numbers **6911222333**, **6944555666** and **6977888999**, we must send the message:

```
0000 TN 6911222333 6944555666 6977888999
```

After that, the device will contain a list of these 3 phone numbers.

To clear the list, we need to send the command without parameters:

```
0000 TN
```

3.1.3: CM: Confirmation Method of incomming message

Syntax: CM n1

n1: [0..3], Confirmation method:

0: The device does not send any confirmation.

1: The device sends a confirmation message to the sender of the incoming message.

2: The device sends a confirmation message to the first number in the **TN** list.

3: Correct message confirmation by phone call and error report with message to the sender of the incoming message.

Default value: n1 = 1

Comment: When **n1** is equal to **1** or **2**, the device responds for confirmation with a message similar to the received one, with the following differences:

If the message is fully accepted, the first 4 characters (the security code) are replaced with "**O.K.**", and the rest of the message is returned unchanged.

In case of an error (syntax error or parameter value out of bounds), the first 4 characters are replaced with "**Ennn**", where "**E**" is the initial of "**Error**" and **nnn** is a three-digit number, indicating the location of the error in the message.

Following is the message received, with the character "**>**" inserted before the character where the error was detected.

Example: If the security code is "**0000**" and we send the following message to change it to "**1234**" (see SC):

```
0000 SC 1234 2234
```

the device will respond with:

```
E014 SC 1234 >2234
```

which means that an error was detected in position #14 (wrong second code in this case).

Now, if we send the message correctly:

```
0000 SC 1234 1234
```

the device will respond with:

```
O.K. SC 1234 1234
```

3.1.4: HD: message HeaDer

Syntax: HD str1

str1: Heading text with a maximum length of 150 characters.

Default value: str1 = "", The text str1 is empty.

Comment: With this command we have the ability to define a header text, preceding any other text sent by the device in response to various events. It can be used to add the device ID, to mark the message with the date/time of sending and more.

Example: To set the message header to "**GAT device # 1**" followed by the time and date information, we can enter:

```
0000 HD "GAT device #1 %T - %D"
```

Suppose, we have enabled the sending of the message "**Input 1 ON**" on activation of Input 1 (see **IM**).

On Input 1 activation the device will send a message like:

```
GAT device #1 15:35:25 - 2020/08/15
Input 1 ON
```

3.1.5: GR: GSM network status Report

Syntax: GR n1 n2

n1: [0..4], Output control mode:

0: No output is controlled.

1..4: Output with index **n1** is activated in case of **GSM** network interruption, and is deactivated when the network is restored.

n2: [0..9], Client index:

0: No message is sent after network restoration.

1..8: Client with index **n2** (see **TN**) is notified with message after network restoration.

9: All clients (see **TN**) are notified with message after network restoration.

Default values: n1 = 0, n2 = 0, function deactivated

Comment: When parameter **n1** is not **0**, the device notifies for change on the GSM network connection by controlling the output with index **n1**. The output used continues to obey to remote control commands, while it can also be set for pulsed operation (see **OP**).

When parameter **n2** is not **0**, the device notifies the recipient specified with index **n2** with a message when the network is restored. With **n2 = 9** all recipients are notified.

The message sent is like the following:

```
GSM RESTORED
was off-line for X'
```

where **X** is the time in minutes that the device was off-line.

Example: To program the device to activate output **1** in the event of a **GSM** network outage, the command is:

```
0000 GR 1 0
```

To send an alert message to all SMS recipients without changing any output, the command is:

```
0000 GR 0 9
```

3.1.6: RR: Restart Report

Syntax: RR n1

n1: [0..9], Restart Report function mode:

0: The device does not send a message after device restart (function is disabled).

1..8: The device sends a message after device restart to client with index **n1** (see **TN**).

9: The device sends a message after device restart to all of the **TN** list clients.

Default value: n1 = 0, function deactivated.

Comment: When this feature is enabled, the device notifies with a message after each reboot of its firmware. The message describes the reason of the restart, so that in each case the appropriate measures can be taken:

"RESTART REPORT: POWER UP" : power failure

"RESTART REPORT: BLOCK" : software crash (electric noise or bug)

"RESTART REPORT: BUTTON" : operation of the RESET button

Example: With the command

```
0000 RR 9
```

we activate the restart report function. Thus, e.g., if there is a power outage, after reset the device will send the message:

```
RESTART REPORT: POWER UP
```

to all the users on the list.

3.1.7: IM: Input event Messages

Syntax: IM n1 n2 str1 str2

n1: [1..8], Input index.

n2: [1..9], Which clients to notify when the input changes state:

1..8: Client with index **n2** (see **TN**) is notified with **SMS**.

9: All of the **TN** list clients are notified with **SMS**.

str1: Input activation message with a maximum length of 150 characters.

str2: Input deactivation message with a maximum length of 150 characters.

Default values: str1 = "", str2 = "", function deactivated.

Comment: By activating this function, when a state change on the input defined by paramater **n1** is detected, the device sends the appropriate messages to the users defined with parameter **n2**.

Examples:

1) To match the message "**ALARM ON!**" on activation and the message "**ALARM OFF**" on deactivation of input **1** and to define that this message will be sent to all **TN** list clients, the command is:

```
0000 IM 1 9 "ALARM ON!" "ALARM OFF"
```

After this command, each time input **1** is activated, the device will send the following SMS to all the phone numbers in the **TN** list:

```
ALARM ON!
```

While, when input **1** is deactivated it will send:

```
ALARM OFF
```

2) To program the device to send the message "**Room Temperature O.K.**", each time input **2** is activated and the message to be sent to the first client of the **TN** list:

```
0000 IM 2 1 "Room Temperature O.K." ""
```

Notice the two sticky quotes that follow the activation message, they are necessary to define that there is no deactivation message.

After this command, after activation of input **2** the device will send:

```
Room Temperature O.K.
```

while it will ignore its deactivation.

3) To disable all messages related to state changes on input **1**, the command is:

```
0000 IM 1 1 "" ""
```

Parameter **n2** does not matter in this case.

3.1.8: IT: Input reaction Time

Syntax: IT n1 n2

n1: [1..8], Input index

n2: [0..65500], Time in tenths of a second (0.1 sec) required for the level to be stable on the input, to accept its state.

0: The minimum reaction time of a voltage input is **0.02 sec**.
65500: The maximum response time is **6550 sec**, i.e., **109.17 minutes**.

Default value: n2=10 (1 sec)

Comment: This command is useful to filter-out too fast changes on the inputs of the device, and also insert time delay between the input trigger and the generated event.

Example: To set the response times of Input #1 to **20 seconds** and of Input #2 to **0.5 second**:

```
0000  IT 1 200  IT 2 5
```

3.1.9: AIC: Analog Input Calibration

Syntax: AIC n1 n2 n3

n1: [1..8], Input index.

n2: [-10000..10000], Calibration Offset.

n3: [-9990..9990], Calibration Factor.

Default values: n1=0, n2=1000

Comment: The analog value "**An**" for each voltage input is calculated by the formula:

An = (Input * Factor) + Offset

"**Input**" is a floating point value that moves in the range **0.00 .. 1.00**, for input voltage in the range **0 .. 10 Volt**. "**Offset**" and "**Factor**" are the parameters **n2** and **n3** respectively.

With the default values for "**Offset**"=**0** and "**Factor**"=**1000**, the formula is done:

An = (Input * 1000) + 0

and so the value "**An**" fluctuates in the range of **0 .. 1000**, with a resolution of **1 unit per 0.01 Volt**.

Examples:

1) To calibrate the analog value of input **1** so that it fluctuates in the range **0 .. 100**, for input voltage in the range of **0.00V**..**10.00V**:

```
AIC 1 0 100
```

2) To calibrate the analog value of input **2** so that it fluctuates in the range **-500 .. + 500**, for input voltage in the range of **0.00V**..**10.00V**:

```
AIC 2 -500 1000
```

3.1.10: AIT: Analog Input Thresholds, convert analog to binary value

Syntax: AIT n1 n2 n3

n1: [1..8], Input index.

n2: [-9990..9990], Low threshold.

n3: [-9990..9990], High threshold.

Default values: n1 = 0, n2 = 1000

Comment: The binary value or "state" of the input can be **0** (low) or **1** (high).
This value is determined by the following rules:

***** If the value "**An**" is less than "**Low threshold**", then the value "**In**" changes to **0**.

***** If the value "**An**" is greater than "**High threshold**", then the value "**In**" changes to **1**.

The change of the binary value can trigger events such as sending messages or making phone calls.

By default, the input state changes to **0** when "**An**"**<200** (input voltage drops below **2.00V**) and changes to **1** when "**An**"**>400** (input voltage increases above **4.00V**). Each input has its own high and low thresholds. The maximum threshold is **9.99V**.

Example: To setup (with the default input calibration settings) the state of input **1** to change to **0** when the input voltage drops below **4.00V** and change to **1** when it rises above **8.00V**:

```
AIT 1 400 800
```

3.1.11: OC: Output Command, define output control commands

Syntax: OC n1 str1 str2

n1: [1..4], Output index.

str1: Output activation command with a maximum length of **150** characters.

str2: Output deactivation command with a maximum length of **150** characters.

Default values: str1 = "", str2 = "" : the function is disabled

Comment: With this command we can define texts that will serve as remote control commands for the outputs. Remote control commands set with this command are sent as messages without the need for security code.

When the device receives such a command it will respond with a confirmation message, depending on the **CM** command setting.

Example: To program output **1** its activation command to be the text "**OPEN DOOR**" and the deactivation command to be the text "**CLOSE DOOR**":

```
0000 OC 1 "OPEN DOOR" "CLOSE DOOR"
```

After this command, output **1** is activated with the message

```
OPEN DOOR
```

and turns off with the message

```
CLOSE DOOR
```

3.1.12: OP: Output Pulse mode

Syntax: OP n1 n2

n1: [1..4], Output index.

n2: [0..65500], Mode of operation:
0: Continuous Output operation
1..65500: Output pulse duration in seconds

Default value: n2 = 0, continuous output function

Comment: In some applications we need a pulse from an output, either because there is a device that is activated by a short pulse or it must work for a certain time. In this mode, when an output is turned on it stays on for a certain amount of time and then turns off automatically.

Example: To program output **1** to produce a pulse lasting **5** seconds:

```
0000 OP 1 5
```

After this command, when output **1** is activated by remote control command or other function, it will remain active for **5** seconds and then will be deactivated.

3.1.13: TCI: Tel. Call on Input activation

Syntax: TCI n1 n2

n1: [1..8], Input index

N2: [0..9], Mode of operation:

0: Function disabled.

1..8: Phone call to the client with index **n2** in the **TN** list.

9: The device makes telephone calls to all clients in the **TN** list.

Default value: n2 = 0 : the function is disabled.

Comment: When input **n1** is activated, the device will make a phone call to one or more recipients specified by parameter **n2**.

Example: To program the device so that an input **1** activation triggers a phone call to the first user in the **TN** list:

```
0000 TCI 1 1
```

To turn off this feature:

```
0000 TCI 1 0
```

3.1.14: TCO: Tel. Call Output, remote control with phone call

Syntax: TCO n1 n2

n1: [1..4], Input index.

n2: [0..9], Mode of operation:

0: The function is disabled.

1..8: Output **n1** is activated with a phone call from the client with index **n2** in the **TN** list.

9: Output **n1** is activated with a phone call from any client contained in the **TN** list.

Default value: n2 = 0, The function is disabled.

Comment: This function is used for output remote control with unanswered phone calls from specific numbers. The output used can also be set to pulse mode (see **OP**).

Example: To setup the device to activate output **1** with a call from the first client and output **2** with a call from any client contained in the **TN** list:

```
0000 TCO 1 1  TCO 2 9
```

To turn off this feature:

```
0000 TCO 1 0  TCO 2 0
```

3.1.15: TCA: Tel. Call Answer, functionality check

Syntax: TCA n1

n1: [0..9], Mode of operation:

0: The function is disabled.

1..8: The device answers the incoming telephone call with a telephone call, only for the phone number of client **n1** in the **TN** list.

9: The device answers the incoming telephone call with a telephone call, if the phone number is included in the **TN** list.

Default value: n1 = 0, The function is disabled.

Comment: This command allows us to confirm that an installed device is on-line and ready, with the use of unanswered phone calls.

Combined with remote control by phone calls (see **TCO**), it enables reliable and inexpensive remote control.

Example: To program the device to answer with phone call to any phone number in the **TN** list:

```
0000 TCA 9
```

3.1.16: TCD: Tel. Call Duration, setup the duration of outgoing phone calls

Syntax: TCD n1

n1: [4..40], Duration in seconds of outgoing telephone call caused by the **TCI** / **TCA** functions or the automation program.

Default value: n1 = 15

Comment: The **n1** parameter sets the maximum time in seconds that the device allows to elapse after the start of the telephone call. The time it takes to make a GSM phone call varies, depending on the signal strength and the GSM network load. Also, some voicemails that the GSM network provider may interfere with, may delay the execution of the call.

Example: To program the device to hold the phone call for **20** seconds:

```
0000 TCD 20
```

3.1.17: RTC: Real Time Clock setup

Syntax: RTC str1

str1: Text containing date / time information.
Must have the format "**YY/MM/DD,hh:mm:ss**", where:
"**YY**" = year (00..99)
"**MM**" = month (01..12)
"**DD**" = day (01..31)
"**hh**" = time (00..23)
"**mm**" = minutes (00..59)
"**ss**" = seconds (00..59)

Comment: The device's real-time clock is automatically updated by the GSM network when it is online. When the power is off, it continues to operate for some hours thanks to the built-in super capacitor. In some cases, the user may need to update it with a command.

Example: The message

```
0000 RTC "20/04/15,10:30:00"
```

sets the date to April 15, 2020 and the time to 10:30:00.

3.1.18: APR: Automation Program Run, Automation Program execution control

Syntax: APR n1

n1: [0..2], Control value

0: The automation program stops.
1: The automation program starts with initial conditions:
Application variables reset and execution of the “Prologue” module.
2: The automation program continues running the “Main” module,
without resetting the variables and execution of the “Prologue” module.

Comment: Stop, start and resume running the automation program.

Example: To start the automation program, the command is:

```
0000 APR 1
```

3.1.19: RS: ReSet, Device initialization

Syntax: RS

Comment: Resets all programmable device settings to the default settings, shuts off the automation program, and frees up the volatile variables. This command also resets the security code to "**0000**".

Example:

```
0000 RS
```

3.2: IMMEDIATE EXECUTION COMMANDS

These commands cause some action the moment they are received by the device, without any functional parameter change.

3.2.1: OA: Output Activate

Syntax: OA n1

n1: [1..4], Output index.

Comment: The device immediately activates the output defined by parameter **n1**.
The output remains active until a shutdown command is received, or until the pulse time has elapsed in pulsed output mode (see **OP**).

This command is not accepted (it is considered an error), if a special remote controlling command is programmed for this output (see **OC**).

Example: To activate output **1**:

```
0000 OA 1
```

3.2.2: OD: Output Deactivate

Syntax: OD n1

n1: [1..4], Output index.

Comment: The device immediately switches off the output defined by parameter n1. This also applies to pulsed output mode (see **OP**).

This command is not accepted (it is considered an error), if a special remote controlling command is programmed for this output (see **OC**).

Example: To turn off output **1**:

```
0000 OD 1
```

3.2.3: SMS: send SMS immediately

Syntax: SMS tn1 str1

tn1: Telephone number to which the SMS will be sent, up to **16** digits.

str1: Text to be sent, with a maximum size of **150** characters.
As an exception to the general syntax rules, only the beginning of the message is defined with quotation mark.

Comment: Send an SMS immediately to a phone number. The text to be sent extends to the end of the message. This implies that this command must be the last in a message.

Example:

To send the device the message "HELLO!" at the telephone number 691234567890:

```
0000 SM 691234567890 "HELLO!
```

Note that the quotation mark character at the end of the text is missing.

3.3: INFORMATION QUERY COMMANDS

These commands cause the device to respond immediately in the form of one or more messages.

3.3.1: ST: functional STatus query

Syntax: ST

Comment: The device responds to the sender of the order with a message in the following format:

```
ST:
I=iiiiiiii O=oooo
SIG=signal
RTC=date,time
MSG_SZ=message_size
AP_SZ=ap_size NVV_SZ=nvv_size
AP=ap_checksum,ap_state
ERROR=now:gen_error/ap_error,last:last_ gen_error
MO=mo_ok_cnt/mo_fail_cnt TO=to_ok_cnt/to_fail_cnt
AICAL=an_inp_cal
```

where the red words are the data returned by the device and are explained below:

iiiiiiii: The binary state of the voltage inputs in the form of digits **1** and **0.**

oooo: The binary state of the relay outputs in the form of digits **1** and **0**.

signal: When the device is connected to a GSM network the signal strength is **1 .. 5**, when there is no GSM connection it is '**X**'.

date: The date in the format **YYYY/MM/DD**, where:
YYYY: 2000 ... 2099, year
MM: 01 ... 12, month of the year
DD: 01 ... 31, day of the month

time: The time in the format **HH:MM:SS** όπου

HH: 00...24, hours
MM: 00...59, minutes
SS: 00...59, seconds

message_size: Size of non-volatile memory used for texts.

ap_size: Size of non-volatile memory used for the Automation Program code.

nvv_size: Size of memory used for the Automation Program non-volatile variables.

ap_checksum: Automation Program Checksum

ap_state: Execution state of the Automation Program (**RUNNING** or **STOPPED**)

gen_error: General error code. May be:
0: No Error
1: GSM module startup
2: GSM module communication #1
3: GSM module communication #2
4: Sending SMS
5: Functional parameters
6: Text memory
7: Application program memory
8: Automation Program execution

ap_error: Automation Program error code.
Used to diagnose / clarify the error that occurs when uploading the automation program.

last_ gen_error: Keeps the last non-zero general error code.

mo_ok_cnt: Number of successful SMS sends via GSM.

mo_fail_cnt: Number of failed SMS sends via GSM.

to_ok_cnt: Number of successful phone calls via GSM.

to_fail_cnt: Number of failed phone calls via GSM.

an_inp_cal: Result of factory calibration of analog inputs.
When "**OK**" then the analog inputs have the precision specifications that refer to the technical characteristics of the device.
If it is "**FAIL**", then the analog inputs accuracy falls to 3%.

Statistics from **gen_error** onwards are valid for the period from the last boot of the device.

Example: After a status query with the command:

```
0000 ST
```

a GAT-2 device can respond with the message:

```
ST:
I=11100000 O=1100
SIG=4
RTC=2020/04/25,01:18:31
```

```
MSG_SZ=144
AP_SZ=213 NVV_SZ=9
AP=78B8,RUNNING
ERROR=now:0/0,last:0
MO=0/0 TO=0/0
AICAL=OK
```

which means:
* inputs 1, 2 and 3 are on, outputs 1 and 2 are on,
* the GSM signal strength is 4 out of 5,
* the date is 04/25/2020 and the time is 01:18:31,
* the text size for messages used is 144 bytes,
* the size of the automation program is 213 bytes,
* the size of the indelible variable memory used is 9 bytes,
* the control sum of the automation program is 78B8 and it is currently running,
* error codes are 0 / 0.0 which means that everything is fine,
* no SMS or phone calls have been made from the start of the device until now
* and the calibration of the analog inputs is OK.

3.3.2: ID: Identification Data query

Syntax: ID

Comment: The device responds to the sender of the command with the message:

```
ID:
GAT-n version v.v SN:xxxxxx
```

where **n** is the **device type** (**1** or **2**), **v.v** is the **firmware version** and **xxxxxx** is the **device serial number**.

Example: After sending the next message to a GAT device:

```
0000 ID
```

the device can answer:

```
ID:
GAT-1 version 1.2 SN:10440A
```

which means that the device type is **GAT-1**,
the software version is **1.2**
and the device serial number is **10440A**.

3.3.3: FP: Functional Parameters query

Syntax: FP n1

n1: [0..1], Answer method:

0: Send only the functional parameters that differ from the default settings.

1: Send all functional parameters.

The parameter **n1** is optional, if omitted, it's as if it equals **0**.

Comment: This command is used to revoke all operating parameters of the device, except the security code. The number of messages that the device will return varies, depending on its type and volume of data.

Example: After requesting the functional parameters from a **GAT-1** device that has the default settings, with the command:

```
0000 FP 1
```

the device will respond with the next messages:

Message #1:

```
TN
CM 1
HD ""
GR 0 0
RR 0
IM 1 1 "" ""
IM 2 1 "" ""
IM 3 1 "" ""
IM 4 1 "" ""
IT 1 10
IT 2 10
IT 3 10
IT 4 10
AIC 1 0 1000
AIC 2 0 1000
*
```

Message #2:

```
FP p2
AIC 3 0 1000
AIC 4 0 1000
AIT 1 200 400
```

```
AIT 2 200 400
AIT 3 200 400
AIT 4 200 400
OC 1 "" ""
OC 2 "" ""
OP 1 0
OP 2 0
TCI 1 0
TCI 2 0
TCI 3 0
*
```

Message #3:

```
FP p3
TCI 4 0
TCO 1 0
TCO 2 0
TCA 0
TCD 15
```

End of messages

Each message starts with "**FP pn**", where "**pn**" is "**p1**", "**p2**" etc. and shows the message serial number. Messages #1 and #2 end with the '*****' symbol, which means that another message is following.

If we now ask the question with zero (or without) parameter:

```
0000 FP
```

the device will respond:

```
FP p1
```

that is, it will return an empty list.

This means that the device has no settings other than the default ones.

3.3.4: AV: Analog input Values query

Syntax: AV n1

n1: [1..8], Analog value query of input with index **n1**.
Without parameter, the device returns the analog values of all inputs.

Comment: The device responds with a message stating the analog value of one or all of the inputs.

Examples:

1) To ask the analog value of input #2 we must send:

```
0000 AV 2
```

If the device has the default calibration and a voltage equal to **5.00V** is connected to input **#2**, the device will respond:

```
A2=500
```

2) To query the analog values of all inputs of the device we must send:

```
0000 AV
```

The device will return a list of all analog input values (here a GAT-1 device, with 4 inputs):

```
A1=497
A2=1000
A3=0
A4=740
```

3.3.5: APL: Automation Program Listing query

Syntax: APL

Comment: After this command, the device responds with one or more messages containing the automation program. Each message starts with "APL pn", where "pn" is "p1", "p2" etc., and shows the message serial number. When another one is to follow, the message ends with the '*' symbol.

Example: To ask the automation program we must enter:

```
0000 APL
```

The device can respond with the following message:

```
APL p1
P() M()
```

In the above case the automation program is empty.

4: THE AUTOMATION PROGRAM “GAT-AP”

The GAT device firmware supports a structured text programming language, the “**GAT-AP**” (GAT Automation Program). In cases where the device needs to perform some automation or we need to create communication functions that do not exist ready-made, we need to write an automation program.

The automation program is organized into modules and sub-modules that run under conditions. A module can contain statements that can be expressions with system and general use variables, system function calls and sub-modules, while it is able to perform logical and arithmetic operations and text management.

Editing an automation program is a process done on a computer through the "**GATcomm**" support software.

With the automation program, specialized functions can be implemented that are not ready-made in the software. The following example will give you an idea.

4.1: Example of an automation program

Suppose we have the following scenario: The device measures the voltage at input 1. A user, known to the device by his phone number, can make an unanswered call to the device, whenever he wants to know the voltage at input 1. The device replies with an SMS like the following:

Input #1 Voltage: 5.48V

The automation program needed for the above operation is the following:

```
;Application with simple automation program
TN 1111111111 ;Telephone Numbers list with 1 client

P() ; “Prologue” module, empty

M( ; “Main” module
      IF TCR==1 ( ;telephone call reception from client #1
            ;Send report message to client #1
            SM( 1, "Input #1 Voltage: %fA1V" )
      )
)
;end of application
```

In the text above, the **green text** that is written after the character "**;**" (semicolon) are comments useful for understanding the program, which are not transmitted to the device but are stored in the application file. The text transmitted to the device looks like the following:

```
TN 1111111111
P()M(IF TCR==1 (SM(1,"Input #1 Voltage: %fA1V")))
```

The "**GATcomm**" software checks the automation program while it is being transmitted and sends it to the device "condensed", without comments and unnecessary characters.

4.2: Program Modules & Program Cycle

Program modules are the groups of expresions that are executed at different times or under different conditions. Syntactically, a program module line is enclosed in a pair of parentheses "**(**" and "**)**".

A program module may be empty.

An automation program consists of two basic and mandatory modules.

There is the module that is executed on the **first program cycle** and we call it "**Prologue**", and the "**Main**" module that is executed on **every program cycle**.

The Prologue section runs just once, during the (re-) start of the automation program. Its main utility is the preparation of some elements of the program that will be used in the main module.

On each **program cycle**, the GAT-FW starts at the beginning of a module and executes sequential operations, branching commands and functions it encounters, until it reaches the end of the module.

The Main module is the one that runs on each program cycle (and also in the first one, immediately after the Prologue section).

Program cycles are performed at a regular interval of 10 milliseconds.
Thus, 100 program cycles are performed every second.

4.3: Syntactic Rules & Comments

At this point we will explain a very simple automation program.

In the automation program text, whatever is written after the symbol '**;**' (semicolon) and until the end of a line is a **comment** and exists only to clarify the program. The "GATcomm" text editor accepts comments (which can be written in any language, t.m. with characters other than Latin that are allowed to write the program) and saves them in the program text file.

The automation program is as follows:

```
P(   ;beginning of Prologue module
)    ;end of prologue module

M(   ;beginning of Main module
     O1 = I1 a I2      ;output1 = input1 AND input2
)    ;end of Main module
```

The program you see does a very simple job:
Activates output 1 when input 1 AND input 2 are activated simultaneously.
When one of these inputs is turned off, output 1 is also turned off.

In the first row, we see the representation "**P(**", which is the character "**P**" (from Prologue) and the symbol "**(**" (opening parenthesis). This marks the **beginning of the prologue module**.

In the next line we see the symbol "**)**" (closing parenthesis). This marks the **end of the prologue module**.

In this program the prologue section is module, ie, nothing is contained between the beginning and its ending. That's okay, a module can be empty.

In the next row we see the representation "**M(**", ie, the character "**M**" (from Main) and the symbol "**(**" (opening parenthesis). This marks the **beginning of the main module**.

In the next row we see an **expression** in which some **system variables** are combined with **operators**. All these new elements will be presented in the following chapters, but here we will mention that "**O1**" symbolizes output **1**, "**=**" is the assignment operator, "**I1**" and "**I2**" symbolize inputs **1** and **2** respectively and "**a**" symbolizes the operator for the logical operation **AND**.

In the next row we see the symbol "**)**" (close parenthesis). This marks the **end of the main module**.

The program presented could well be written, for character economy, as follows:

```
P()M(O1=I1aI2)
```

Blank characters are not required between the various elements, when operators are inserted. The program is still understandable by the GAT-FW.

Certainly, there is no need to write a program in such an incomprehensible way, when you write it through the "GATcomm" software. Write it with spaces, identation, line breaks and comments so that it is understandable. "GATcomm" will deal with the character economy, when the program needs to be sent to the device.

4.4: Variables

Variables are an important element in any programming language.

A variable is a "storage" that contains a value which represents a state, a quantity etc. and as the name implies, it may change. In the example of the simple program we already saw, **O1** represents the state of output 1. In the case of outputs, the device recognizes the variables **O1**, **O2**, **O3** and **O4**.

These are **system variables**, because the device firmware undertakes to transfer the values (which the automation program assigns to them) to the corresponding outputs, thus, translating the result of an expression in the program (**O1 = I1 a I2**) into a physical result (**relay output control**).

In addition to system variables, the device supports **general purpose application variables**.

The **type** of the variable O1 is **binary**, i.e, this variable can take only two values, the value **0** which translates to disabled output and the value **1**, which translates to activated output. In addition to the binary type, the GAT-AP language supports variables that are integer decimal numbers.

The last characteristic of a variable is, whether the application program can assign a value to that variable, i.e., whether that variable is **writable** or not. In the case of the outputs, of course we can "write them". In other cases, such as the variables representing the state of the inputs, this is prohibited because they represent an existing state.

4.5: System Variables

Most interaction of the automation program with the real world, is done via the **system variables.** The GAT-FW updates the system input variables before a program cycle is executed, and updates the relay outputs and other functionalities based on the system output variables.

The automation program has access to the inputs and outputs of the system through the following variables:

4.5.1: An, In: Analog & Binary Voltage Input Value

An: Αναλογική τιμή εισόδων

Syntax:	**A1 ... A8**
Range:	**-32768 … 32767**
Writable:	**NO**

Comment: Analog values of voltage inputs. They are automatically updated before each program cycle.

In: Δυαδική τιμή εισόδων

Syntax:	**I1 ... I8**
Range:	**0 … 1**
Writable:	**NO**

Comment: Binary values of voltage inputs. They are automatically updated before each program cycle.

4.5.2: On: Output state

Syntax:	**O1 ... O4**
Range:	**0 … 1**
Writable:	**YES**

Comment: State of the relay outputs. The relays are automatically updated by these variables, after each program cycle.

4.5.3: Timer variables

The GAT-FW has **8 independent timers**, i.e, timekeeping mechanisms with a resolution of 1 sec and an accuracy of 0.1sec. Each timer is associated with 2 system variables:

TEn: Timer Enable

Syntax:	**TE1 ... TE8**
Range:	**0 … 1**
Writable:	**YES**

Comment: Timer activation flags. When the value 1 is assigned to one of them in the automation program, the corresponding timer is activated. When the value 0 is assigned, the timer stops counting.

TCn: Time Counter

Syntax:	**TC1 ... TC8**
Range:	**-2147483648 ... 2147483647**
Writable:	**YES**

Comment: When a timer is activated, the value of the corresponding counter automatically increases every second. When the timer is off, the value of the corresponding meter remains constant.

4.5.5: Real Time Clock

There is a set of system variables, which represent date and time information in the form of integers and are used to implement automations that depend on this information.

These variables all start with "**RT**", they are not writable and are automatically updated before each program cycle:

Name:	**Real Time clock Hour**
Syntax:	**RTHO**
Range:	**0 ... 23**

Name:	**Real Time clock Minutes**
Syntax:	**RTMI**
Range:	**0 ... 59**

Name:	**Real Time clock Seconds**
Syntax:	**RTSE**
Range:	**0 ... 59**

Name:	**Real Time clock Year**
Syntax:	**RTY**
Range:	**0 ... 99**

Name:	**Real Time clock Month**
Syntax:	**RTM**
Range:	**1 ... 12**

Name:	**Real Time clock Day**
Syntax:	**RTD**
Range:	**1 ... 31**

4.5.5: GSM state variables

There are two system variables, which are used so that the automation program can know the current state of the GSM network. These variables are not writable and are automatically updated before each program cycle.

Name:	**GSM state**
Syntax:	**GST**
Range:	**0 ... 8**

Comment: Current GSM network status.

The table below explains the meaning of the values this variable can have:

0	Off-line
1	On-line, idle
2	Outgoing telephone call active
3	Telephone communication active
4	Incoming phone call ringing
5	Incoming message process active
6	Outgoing message process active
7	The outgoing message queue is full
8	GSM network error

Name:	**GSM signal strength**
Syntax:	**GSI**
Range:	**0 ... 5**

Comment: The value of this variable represents the current signal strength of the GSM network.

4.5.6: Diagnostic variables

Name:	**Function Result**
Syntax:	**FR**
Range:	**0 ... 255**
Writable:	**YES**

Comment: The value of this variable is set by the last program function that was executed (see Functions).

Name:	**Incoming Message information**
Syntax:	**MSR**
Range:	**0 ... 255**
Writable:	**YES**

Comment: This variable can be:

0 when no message has been received,
1..8 when the message comes from a TN list user,
9 for a message from an unknown sender and
255 for a message received via USB

Name:	**Incoming phone call information**
Syntax:	**TCR**
Range:	**0 ... 255**
Writable:	**YES**

Comment: This variable can be:

0 when there is no incoming call,
1..8 when the call comes from a user in the TN list and
9 for a call from an unknown sender

Name:	**Message-Out Success Counter**
Syntax:	**MOSC**
Range:	**0 ... 255**
Writable:	**YES**

Comment: This variable is 0 at device startup and increases by 1 with each message send success.

Name:	Message-Out Failure Counter
Syntax:	MOFC
Range:	0 ... 255
Writable:	YES

Comment: This variable is 0 at device startup and increases by 1 with each message send failure.

Name:	Telephone Call Failure Counter
Syntax:	TOFC
Range:	0 ... 255
Writable:	YES

Comment: This variable is 0 at device startup and increases by 1 with each outgoing telephone call failure.

Name:	Error code
Syntax:	ERR
Range:	0 ... 4
Writable:	NO

Comment: Error code.
The table below explains the values it can have.

0	No error
1	GSM unit startup error
2	Communication error with GSM module #1
3	Communication error with GSM module #2
4	SMS send error

4.6: Application variables Bn & Nn

There are two groups of general purpose variables that can be used for calculations, data storage and other uses that will be presented below.

When the automation program starts, these variables are all reset, except for those defined by the automation program as "non volatile variables", which hold their values even when the device is not being powered on (see Functions – NVV).

Name:	**Byte type variable**
Syntax:	**B1 ... B32**
Range:	**0 ... 255**
Writable:	**YES**

Name:	**Number type variable**
Syntax:	**N1 ... N64**
Range:	**-2147483648 ... 2147483647**
Writable:	**YES**

Application variables can be used either individually or as groups called **variable lists** that can be accessed by index, i.e, the position of the variable in the list (see Functions - VG, VS, VE).

4.7: Operators

These are the symbols which define the operations performed between the variables and/or constants which participate in an expression.

4.7.1: The assignment operator '='

The operations defined after (to the right of) the assignment operator (equal to) are performed and the result is assigned to the variable before it (to its left).

```
B1 = I1             ; assign the state of input 1 to variable B1
N2 = 100            ; assign the value 100 to the variable N2
N3 = N2 + 1         ; if N2 is equal to 100, N3 will now be equal to 101
N3 = N3 + 1         ; if N3 is equal to 101, N3 will now be equal to 102
O1 = N3==102        ; if N3 equals 102, output O1 will be activated
```

4.7.2: The Numerical Operators '+' '-' '*' '/' '%'

All arithmetic operations are performed between expressions in the integer field.

Symbol	Description
+	PLUS, Addition
-	MINUS, Subtraction
*	BY, Multiplication
/	DIV, Division
%	MOD, Division remainder

Some examples will clarify how these operators are used:

```
N1 = 5       ; N1 is set to 5
N2 = N1/2    ; N2 will now be equal to 2 (the whole part of 2.5)
N3 = N1%2    ; N3 will now be equal to 1 (the remainder of division 5/2)
```

4.7.3: The Comparison Operators '==' '!=' '>' '>=' '<' '<='

These operators perform numerical comparisons between expressions and produce a binary result, equal to 1 if the comparison was true and 0 if it was false.

Symbol	Description
==	Is Equal
!=	Is Not Equal
>	Is Greater
>=	Is Greater than or Equal
<	Is Less
<=	Is Less than or Equal

Some examples will show how these operators are used:

```
B1 = N1==10  ; B1 will become 1 if N1 equals 10, otherwise it will become 0
B2 = N1!=N2  ; B2 will become 1 if N1 differs from N2, otherwise it will become 0
O2 = N1>=500 ; output 2 will be activated if N1 is greater than or equal to 500
```

4.7.4: The Boolean Logical Operators 'a' 'o' 'x'

These operators perform logical operations (Boolean algebra) between expressions and produce a binary result. Any expression which is different from zero is considered as 'True' and any expression which is equal to zero is considered as 'False'. The binary results of the logical operations are 1 for 'True' and 0 for 'False'.

Symbol	Description
a	AND, Logical Conjunction
o	OR, Logical Disjunction
x	EXOR, Logical Exclusive Disjunction

Some examples will show how these operators work:

```
B1=1 B2=1 B3=0 B4=0 ; assign initial values to these variables
B5 = B1 a B2        ; B5 is now 1 (1 AND 1)
B6 = B1 a B3        ; B6 is now 0 (1 AND 0)
B7 = B3 a B4        ; B7 is now 0 (0 AND 0)
B8 = B1 o B2        ; B8 is now 1 (1 OR 1)
B9 = B1 o B3        ; B9 is now 1 (1 OR 0)
B10 = B3 o B4       ; B10 is now 0 (0 OR 0)
B11 = B1 x B2       ; B11 is now 0 (1 EXOR 1)
B12 = B1 x B3       ; B12 is now 1 (1 EXOR 0)
B13 = B3 x B4       ; B13 is now 0 (0 EXOR 0)
```

4.7.5: The Bitwise Logical Operators '&' '|' '^'

These operators perform bit-per-bit logical operations between integer expressions and produce integer result.

Symbol	Description
&	AND, Bitwise Conjunction
|	OR, Bitwise Disjunction
^	EXOR, Bitwise Exclusive Disjunction

Some examples will show how these operators work:

```
B1=1 B2=2 B3=3    ; assign initial values to these variables
B4 = B1 & B2      ; B4 is now 0 (00000001 AND 00000010)
B5 = B1 & B3      ; B5 is now 1 (00000001 AND 00000011)
B6 = B1 | B2      ; B6 is now 3 (00000001 OR 00000010)
B7 = B1 ^ B3      ; B7 is now 2 (00000001 EXOR 00000011)
```

4.7.6: The Unary prefix Operators '-' '!' '~'

The numeric unary operator which is accepted is the negative sign "-", i.e:

```
N1 = 100
N2 = -N1    ; variable N2 is now equal to -100
```

The logical unary operator recognized is that of negation (NOT), symbolized by the exclamation mark "**!**". He reverses the logical state of the quantity that follows, i.e., if it is different to 0 it returns 0, while if it equals 0 it returns 1:

```
B1=1  B2=0  N1=100  N2=0      ; assign initial values to the variables
B3 = !B1                      ; B3 is now 0 ( NOT 1 )
B4 = !B2                      ; B4 is now 1 ( NOT 0 )
B5 = !N1                      ; B5 is now 0 ( NOT 100 )
B6 = !N2                      ; B6 is now 1 ( NOT 0 )
```

The bitwise logical unary operator recognized is the Binary One's Complement operator (BINARY NOT), symbolized by the tilde "**~**", which has the effect of 'flipping' the bits of a value, t.m. every 1 turns to 0 and every 0 turns to 1:

```
B1 = 170    ; 170 is binary 10101010
B2 = ~B1    ; B2 is now 85 ( binary 01010101 )
```

4.8: Operator priorities

Unlike other programming languages, here the operators do not have other different priorities. The system simply starts up after the assignment operator and calculates the result, performing the operations one by one, until it reaches the end of the expression.

Example 1:

```
N1 = 50                ; assign 50 to N1
N2 = 10 + N1 * 20      ; N2 is now (10+N1)*20 = 1200
```

To enforce the priorities of the operations, we use parentheses:

```
N2 = 10 + (N1 * 20)   ; N2 is now 10+(50*20) = 1010
```

Example 2: Suppose we want to check if the value of variable **B5** is from 5 to 10. The result of the test will be stored in variable **B10**.

```
B10 = ( B5>=5 ) a ( B5<=10 )
```

Example 3: Suppose we want to check if the value of variable **B5** is from 5 to 10 or from 20 to 30. The result of the test will be stored in variable **B10**.

```
B10 = ( ( B5>=5 ) a ( B5<=10 ) ) o ( ( B5>=20 ) a ( B5<=30 ) )
```

4.9: Branch commands

There are cases where some modules of commands have to be executed or not, depending on whether a condition is fulfilled.

In these cases we must utilize the branching commands **IF - ELIF - ELSE**

The general structure of these commands is:

```
IF condition1 (
      sub-module1
)
ELIF condition2 (
      sub-module2
)
ELSE (
      sub-module3
)
```

In the above text, "**condition1**"and "**condition2**" can be any expression which produces result. They are “fulfilled”, when the result of this expression is different of zero. In this case the sub-module that follows is executed. When the result of this expression is zero, the sub-module that follows is skipped.

In place of "**sub-module1**", "**sub-module2**" and "**sub-module3**" can be any expressions, even other branching commands in depth of up to 8 levels.

ELIF and **ELSE** commands are optional, while there can be more than one **ELIF** but only one **ELSE,** always at the end of the branch chain.

At this point an example is needed, to clarify the operation of the branching commands:

```
IF  N1==1
(     ; N1 is equal to 1
      O1=1 ; activate output 1
)
ELIF  N1==2
(     ; N1 is equal to 2
      O2=1 ; activate output 2
)
ELSE
(     ; N1 is different from 1 and 2
      O1=0  O2=0  ; turn off outputs 1 and 2
      O3=1        ; and activate output 3
)
```

Here, there are three sub-modules and in each program cycle only one of them is executed, depending on the value of **N1**.

4.10: GAT-AP Functions

The GAT-AP language provides some functions that can be performed through the automation program.

The general structure of a function is: **FUNCTION(param1, param2 ...)**

Functions have parameters, based on which they perform a task and / or return a result.

The parameters of a function are contained in parentheses, which immediately follow its name and are separated by the comma character ("**,**").

Functions that return a result can participate in expressions with arithmetic or logical operations.

4.10.1: NVV: Define a Non Volatile Variables list

Syntax: NVV(va1, n1, n2, n3)

va1: Name of the first variable in the list. It can be Byte or Number Variable.

n1: [1..32], Number of items in list. It must be a integer constant.

n2: [1..8], Storage update rate. It must be a integer constant.
- **1:** Save every 512 sec.
- **2:** Save every 256 sec.
- **3:** Save every 128 sec.
- **4:** Save every 64 sec.
- **5:** Save every 32 sec.
- **6:** Save every 16 sec.
- **7:** Save every 8 sec.
- **8:** Save every 4 sec.

n3: [1..15], Number of backup copies. It must be a integer constant.

Comment: This function is used to define an area (list) of application variables as "**non volatile**". This means that at regular intervals and as long as they have changed, these variables will be stored in the non volatile memory of the device. When the power is turned off and restored, these variables, instead of starting at zero, will maintain the values they had, before the power outage.

This function is only allowed to be executed in the Prologue section. It does not return a value, so it can not be a member of any arithmetic or logical expresion.

With the parameters **va1** and **n1** we define the range of variables. The range of variables to be defined can have a maximum size of 32 Bytes or 8 Number of variables. A program can define up to 4 lists of non volatile variables.

With the parameter **n2**, we define the time interval between the storage and with the parameter **n3**, we define the copies of the backup copies, thus multiplying by the same factor the memory wear-off endurance by 100,000 rewrites.

After executing this function, the system variable **FR** is set to 1 if all went well, or 0 in case of error.

This function reserves space in the non volatile memory of the device, which is deducted from the total space available for the program and message texts. The space reserved is returned by the **ST** information query command. If, during the development of an automation program, this is loaded and executed with different parameters in the **NVV** function, the GAT-AP does not release the areas reserved in previous tests, resulting in memory areas being blocked. The blocked memory is released with the GAT-FW command **RS** (reset), which must be executed before uploading the automation program to the device.

Example: To define that variables **B10**, **B11**, **B12**, **B13** and **B14** will become non volatile with **8** seconds update rate and with endurance of at least 400,000 rewrites, we must write:

```
NVV(  B10, 5, 7,  4 )
```

4.10.2: RMC: Received Message Contains, Search for text in incoming message

Syntax: RMC(str1)

str1: Text to search

Comment: This function is used to check the existence and location of specific text to be searched in an incoming message.

When an incoming message arrives, containing the text to be searched, the function returns a number, which is the shift of the text that starts at the beginning of the message, plus one. In any other case the function returns 0.

So if the incoming message is **"TEST MESSAGE"**,

RMC("TEST") will return **1**,

RMC("MESSAGE") will return **6**

and **RMC("XXX")** will return **0**.

Also, **RMC("")** will return **1** to any incoming message.

Examples:

1) Activate output 1 when a message containing the text "HEATER ON" is received:

```
IF RMC("HEATER ON") (
     O1 = 1
)
```

2) Check output 1 when a message starting with "HEATER" is received. If "ON" follows, the output is activated, otherwise if "OFF" follows, the output is deactivated:

```
IF RMC("HEATER") == 1 ( ; received message starting (== 1) with "HEATER"
     IF RMC("ON") (          ; "ON" follows
          O1 = 1                  ; activate output 1
     )
     ELIF RMC("OFF ") (      ; "OFF" follows
          O1 = 0                  ; deactivate output 1
     )
)
```

4.10.3: RMN: Received Message Numbers, Collect numbers from incoming message

Syntax: RMN(va1, n1)

va1: First variable in the list, to store the sequence of numbers collected from the incoming message. It can be Byte or Number Variable.

n1: Maximum number of numbers to collect from the incoming message. Must be constant integer.

Comment: This function collects the consecutive numbers contained in the text of an incoming message and saves them in the list of variables defined by parameters vn1 and n1. It returns the number of collected items (which is also saved in system variable **FR).**

Example: In some case we want the device to receive via a message, a twenty-four hour time ("time: minutes") and a temperature value. The message must start with the text "Heater Setup" (see **RMC**).

Suppose the device just received the message:

```
Heater Setup, Time: 15:30, Temperature: 25
```

To store the 3 numbers contained in this message in variables **N10**, **N11** and **N12**, we can write:

```
IF RMC( "Heater Setup") (RMN( N10, 3 ))
```

After the execution of the above code line, the values of the variables will be:
N10=15, N11=30, N12=25

4.10.4: SM: Send Message

Syntax: SM(tn1, str1)

tn1: Can be a integer constant or an application variable with value **0..9**
or a phone number in text format.
0: Send to the sender of the last message arrived.
1..8: Send to the client in the **TN** list with this index.
9: Send to all clients in the **TN** list.
"12345678": Telephone number as text, up to **16 digits.**

str1: Text to send, with a maximum length of 150 characters.

Comment: With this function we can send messages through the automation program. If the GSM unit of the device is not ready to send the message, the automation program pauses at the point of the function, until the message is sent. If the automation program is not allowed to pause, this function must be executed after checking the **GST** system variable (indicating the status of the GSM unit). When this variable is equal to **1** (which means on-line and idle), the function can be executed instantly without delaying the automation program.

Examples:

```
SM( 1, "Hello" )   ; send a "Hello" message to client #1 in the TN list

SM( "11111111", "Hello" )   ; send a "Hello" message to phone number 11111111

B1=2                       ;
SM( B1, "Hello" )          ; send a "Hello" message to client #2 in the TN list
```

4.10.5: TCE: Telephone Call Execute, Execute outgoing telephone call

Syntax: TCE(tn1)

tn1: Can be a integer constant or an application variable with value **0..9** or a phone number in text format.
0: Send to the sender of the last message arrived.
1..8: Send to the client in the **TN** list with this index.
9: Send to all clients in the **TN** list.
"12345678": Telephone number as text, up to **16 digits.**

Comment: With this function we can make phone calls through the automation program. The function is executed instantaneously, even if it cannot start the phone call operation (e.g. because the GSM unit is busy at the moment).

This function changes the value of the system variable **FR**. This becomes 1 when the call is successfully started or 0 in case of failure.

Examples:

```
TCE( 1 )          ; make a phone call to client #1 in the TN list

TCE( 9 )          ; make phone calls to all clients in the TN list

TCE( "11111111" ) ; make a phone call to tel. Number 11111111

B1=2              ;
TCE( B1 )         ;  make a phone call to client #2 in the TN list
```

4.10.6: VG: Get Value from Variables List item

Syntax: VG(va1, cv2)

va1: First variable of the variables list. Can be of type Byte or Number.

cv2: Offset from the first variable in the list. Can be integer constant or a Byte or Number variable.

Comment: Returns the value of the variable which is **cv2** positions after the first variable **va1** in the list.

Example:

If **B1 = 2** and **N3 = 150**, then the expression

```
N10 = VG( N1, B1 )
```

will assign to variable **N10** the value of **N3**, i.e. **150**.

4.10.7: VS: Set Value in Variables List item

Syntax: VS(va1, cv2, cv3)

va1: First variable of the variables list. Can be of type Byte or Number.

cv2: Offset from the first variable in the list. Can be integer constant or a Byte or Number variable.

cv3: The value to be assigned to the variable defined by the previous parameters.

Comment: Sets the value of the variable which is **cv2** positions after the first variable **va1** in the list, to the value **cv3**.

Example:

If **B1 = 2** and **N10 = 150**, then the expression

```
VG( N1, B1, N10 )
```

will assign to variable **N3** the value of **N10**, i.e. 150.

4.10.8: VE: Search for Value in Variables List

Syntax: VE(va1, cv2, cv3)

va1: First variable of the variables list. Can be of type Byte or Number.

cv2: Size of the variables list. Can be integer constant or a Byte or Number variable.

cv3: Value to be searched in the list defined by the previous parameters.

Comment: This function is used to search for a value in a list of variables.

If the value **cv3** exists in the list defined by the first variable **va1** and the list size **cv2**, then **VE** returns the offset from the first variable plus 1, otherwise it returns 0.

Example:

If the variable **N1** is equal to 150 and **N13** is equal to 150 while **N10**, **N11** and **N12** are not equal to 150, then the expression

```
VE( N10, 5, N1 )
```

will return the number 4 (3 plus 1) because the value 150 was found in variable **N13**, which is 3 positions after **N10**.

4.11: The preprocessor macro “def”

A feature which helps in the development of a program, is the possibility that the various components that make it up, such as variables and constant values, have a name related to their use.

The **GAT-AP** language does not innately have the ability to name variables. The gap is filled with the **GATcomm** software and his **program preprocessor**, which checks the contents of the text editor for syntactic and other errors and notifies the user about them, while enhancing the rendering with coloring and structured alignment of the text.

To name the various elements so that the program becomes more understandable to the user, the preprocessor understands the macro '**def**' (from define, definition).

Its syntax is as follows:

```
def NAME expression
```

where **NAME** is the name that will replace **expression** within the program. The **NAME** can contain Latin characters, numbers and underscores '_', while it is not allowed to start with a number.

The **expression** can be anything, such as a numeric constant, a variable name or a complex expression, as long as it is on the same line of text.

The following example shows the capabilities of **def**, which can prove useful in large programs using many variables and constant values that are repeated in text.

Example: Suppose we have the automation program:

```
P()
M(
    IF (A1>45) a !B1 (
        SM(1,“ALARM!!!”)
        B1=1
    )
)
```

The above, with the help of **def** can be written as follows:

```
def TEMPERATURE_INP       A1
def MAX_TEMPERATURE       45
def OVER_HEAT             ( TEMPERATURE_INP > MAX_TEMPERATURE )
def ALARM_MESSAGE_SENT    B1
def SEND_ALARM_MESSAGE    SM( 1, “ALARM!!!” )

P()
M(
    IF OVER_HEAT a !ALARM_MESSAGE_SENT (
        SEND_ALARM_MESSAGE
        ALARM_MESSAGE_SENT=1
    )
)
```