Table of Contents

Description

GPS-MB devices with script programming support (Firmware 10.2 or higher) allow user-written programs to run on the device itself, making them more flexible and powerful.

The device will continue to operate normally while the script stored in its memory is running.

Connecting the GPS-MB to the PC for script programming

The RS232 port of the GPS-MB must be connected to the PC in order to load scripts into it.

This only works with a USB-to-RS232 cable using the FTDI FT232 chip. Only in this way will the Script Programmer software detect it when selecting USB connection

Then the RS232 (NMEA) port must be configured to 115200 bps from the configuration command console

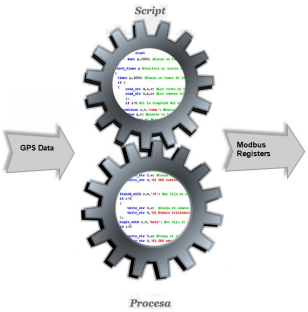
```
Current configuration:
----------------------
Model: SGWI-MB-GPS

NMEA Serial Port (A):
    Baud Rate: 115200
    Data Bits: 8 Bits
    Parity: NONE
    Parse Sentence: DISABLED
```

USB-to-RS232 cable connection (if it is an FTDI brand cable, you can follow the color column)

GPS-MB	Cable USB a RS232	Color (FTDI brand only)
RX (2)	TX (3)	Orange
TX (1)	RX (2)	Yellow
GND (6)	GND (5)	Black

Operations that can be performed from a script

- **Mathematical** operations
- **Logical** operations
- **Timing** operations
- Store values in non-volatile memory
- Reading GPS measurement inputs to perform logic or calculations
- Write new Modbus registers to a dedicated slave ID



How to learn more about script programming

Install the script programming software "Script Programmer", which you can download from www.exemys.com

In the Script Programmer help files or on the Exemys website you will find instructions for learning how to program the devices.

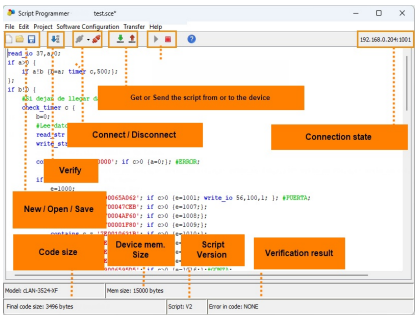
2026-05-05

Introduction

This program allows developing, compiling and transferring scripts to Exemys devices that support it.

Software Description

Below is a description of the main screen functions.



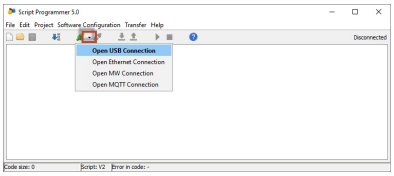
Device Connection

There are three ways to connect: via USB, via LAN/Ethernet, and via MW-XF depending on the device.

Device Connection - USB

First connect the device to the PC and make sure it is disconnected from ***"GRDconfig"***

Then expand the Connect button and choose ***"Open USB Connection"***

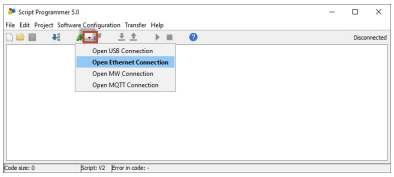


> ⚠ On some devices with a USB port, such as the RMS1-RM and wRemote-LoRa, you must disconnect and reconnect the device's USB port if you previously accessed the USB configuration console.

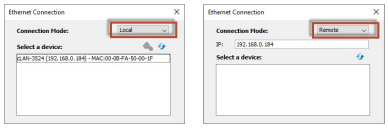
Device Connection - LAN/Ethernet

First connect the device to the same network as the PC and make sure it has a valid IP address configured as indicated in its user manual.

Then expand the Connect button and choose ***"Open Ethernet Connection"***

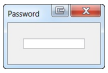


Doing so you should see the device(s) connected to your network, or you can choose the **"Remote"** mode to manually enter the device's IP address if it is on a different network.



Select the one you want to configure.

To connect you will need to enter the device password. It is the same one used to connect via MW.

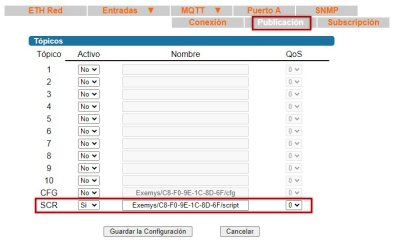


Device Connection via MQTT

Both the device and the Script Programmer must be connected to the same broker to send and receive scripts remotely via MQTT broker.

Device configuration

The SCR base topic used for publishing and subscribing must be configured. The device will append a text to the base as explained later.



Configuration topic - Device identification

The device will append the ClientID to the base configuration topic. The resulting subscribe and publish topics in the previous example will be:

Subscribe -> *Exemys/C8-F0-9E-1C-8D-6F/script/1/clientID*

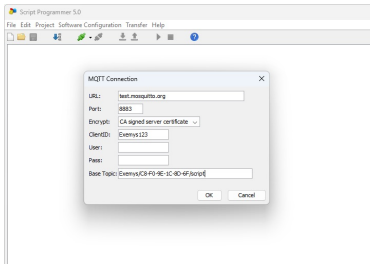
Publish -> *Exemys/C8-F0-9E-1C-8D-6F/script/0/clientID*

The clientID used by the device will be the one configured in the MQTT section.

ScriptProgrammer configuration

The broker parameters to connect to and the base topic to use for configuring remote devices must be loaded in the configurator.

To do this, go to *"Software Configuration -> MQTT Connection"*

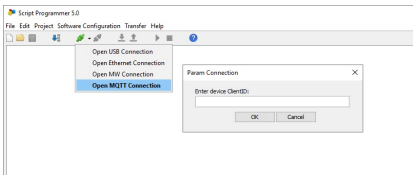


The *basic topic* must match the one configured on the devices.

Remote connection

Then expand the Connect button and choose ***"Open MQTT Connection"***

There you must enter the ClientID of the device to configure.



Once the connection is established, the "Download", "Upload", "Start" and "Stop" buttons can be used as if connected locally to the device.

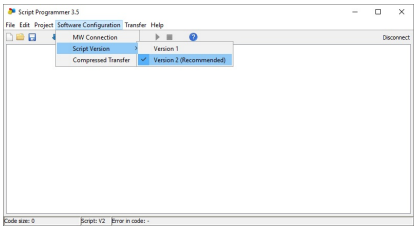
Script Versions 1, 2 and 3

In the ***"Project"*** menu, option ***"Properties"***, tab ***"Script"*** you can select between script version 1, 2 and 3.

Version 2 differs from version 1 in that it allows twice as many variables since they can be lowercase or uppercase.

All these devices work with version 2 or 3. From version 11.0 onwards they automatically switch to version 3. Programs written in version 3 are identical to those in version 2. They only take up less space in the device memory.

Version 2/3 differs from version 1 in that it allows twice as many variables since they can be lowercase or uppercase.

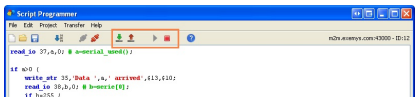


The script version will be used by Script Programmer in two situations: when **verifying** a script or when trying to **send it** to the device.

If when sending a script the selected version is not compatible with the target device, you will see a message indicating that error.

Upload and download of scripts

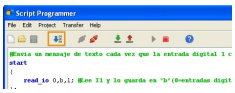
Once connected to the device, we can transfer and download scripts. We can also stop and start them.



Script editing

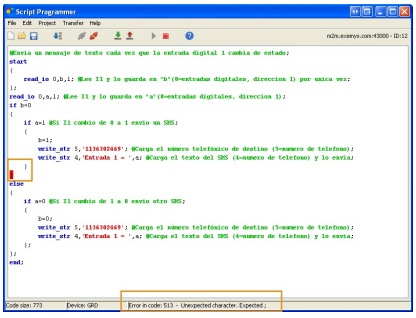
To develop a program, simply write the code in the editing panel. The environment provides autocomplete assistance for functions and highlights them to indicate correct syntax.

Once you finish writing the code, use the ***"Verify"*** button to compile the program and check for syntax errors.



At compile time, the bottom panel will indicate whether there are errors. If there is an error, the line will be highlighted in red and the ***"Error in Code:"*** field will show the line number. If there are no errors, a message will appear showing ***"Error in Code: NONE"***.

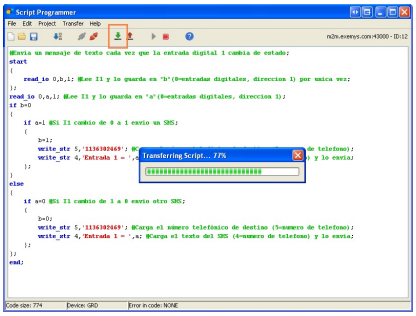
The following image shows a program with an error; in this case a ***";"*** is missing.

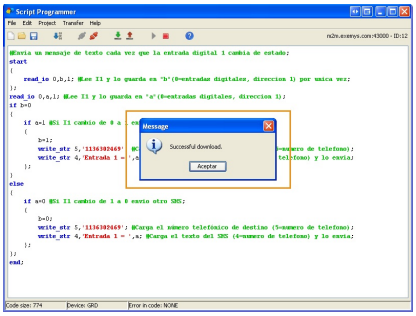


The semicolon is missing in the brace before the one marked in red; this is because the compiler indicates it found an unexpected character.

The following image shows a program without errors.

If there are no errors, we can transfer the program to the device by clicking ***"Download to device"***. A window will appear showing the download status and then a message indicating whether the download was successful or not.





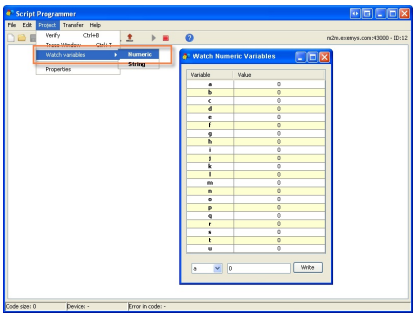
Script debugging

The Script Programmer has two tools for debugging written scripts.

Variable monitoring

With this tool you can view the value of numeric or text variables while the program is running. You can also modify the value of variables to simulate script working conditions.

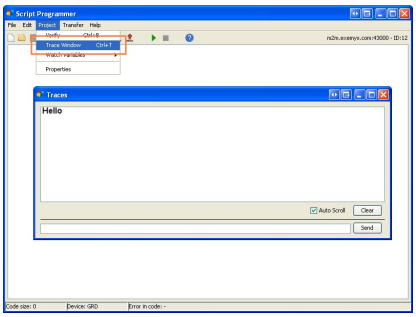
Once connected to the device, go to the **"Project"** menu, option **"Watch variables"** and then **"Numeric"** or **"String"** depending on the type of variable to monitor.



Transmission and reception of traces

With this tool you can send text from the script to the Script Programmer to follow the script's operation. Text can also be sent to simulate script working conditions.

Once connected to the device, go to the **"Project"** menu, option **"Trace Window"**



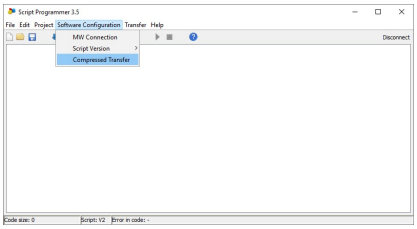
Script compression

The device has limited space for storing scripts. In almost all devices the maximum is 15,000 characters. There are some cases where it is limited to 5,000.

If this space is not sufficient for your application, you can use the script compression option. When enabled, the Script Programmer will remove all comments and tabs before sending the program.

If you want to keep the comments in your program, save a copy on your computer.

To enable compression, go to the **"Software Configuration"** menu, option **"Compressed Transfer"**



2026-05-05

Introduction

The ***Exemys Script*** programming language is a loop-type language, meaning it executes all code up to the last line and then starts again.

There are no statements to create loops within the program, so the program flow cannot be stopped at a specific code section. In this regard it resembles PLC programming, although its syntax is closer to C or Pascal.

Script Versions 1, 2 and 3

There are 3 script versions. Version 2 differs from version 1 in that it allows twice as many variables since they can be lowercase or uppercase. In version 1 variables are only written in lowercase. Version 3 saves between 20 tond 30% of the space occupied by the program in the device memory, allowing longer programs to be loaded. You can notice the difference by pressing the verify button.

Variables

There are 2 types of variables: numeric of type ***"long"*** and text of type ***"string"***.

It is not necessary to define variables since there is a fixed number.

In script version 2, numeric variables are 42, from ***"a"*** to ***"u"*** and from ***"A"*** to ***"U"***. Text variables are 10, from ***"v"*** to ***"z"*** and from ***"V"*** to ***"Z"***, with a maximum length of 100 characters each.

Since numeric variables are of type "long", any operation that yields a decimal result will be truncated.

The initial value of numeric variables is 0; for text variables it is an empty string.

Assign a value to a variable:

- Numeric variables:

```
a = 652;
```

- Text variables:

```
v = 'Hello'
```

Note that to assign text, it must be placed in single quotes.

String concatenation:

To concatenate variables, simply place them one after the other, separated by commas.

For example:

```
a = 20;
u = 'The temperature is: ';
v = ' °C';
```

If we want to form the string 'The temperature is 20 °C ' and store it in another variable we do the following:

```
w = u, a, v;
```

Another way to do it would be:

```
w = 'The temperature is: ', a,' °C';
```

Concatenation can only be done in text assignment to string variables and in the write_str function

Insertion of ASCII values in strings:

To insert an ASCII value in a string, the $ operator can be used. After the operator, the ASCII code in decimal must be specified. ASCII 0 is not allowed.

For example:

```
z = 'Hello',$13,$10;
```

ASCII value insertion can only be done in text assignment to string variables and in the write_str function

Variable Aliases

From version **6.1** of the Script Programmer, variable "aliases" can be created to make the code easier to read.

This code can be transferred to ALL Exemys devices that support Script Programmer, since before sending the code to the device, the alias is replaced by the corresponding variable.

Since aliases are sent as comments within the code, when reading the script loaded on the device, the variables will be replaced back by the aliases, unless the compression option was used, which removes comments.

It is ALWAYS recommended to keep a copy of the scripts loaded on the devices.

Example:

Before	Now
	#[a=SP1];
read_io 2,a,1;	#[b=presion];
if a>b {	read_io 2,presion,1;
write_io 1,8,1;	if presion>SP1 {
};	write_io 1,8,1;
end;	};
	end;

If you already have a written script, you can add comments indicating the aliases, send it to the device (uncompressed) and then read it back. When doing so, all variables with aliases will be replaced.

Arithmetic operators

Below are the operators that can be used in the script. Note that it only handles "long" type values, so any operation yielding decimal results will be truncated - only the integer part will be returned.

Operator	Meaning
=	Equals
^	Exponential
|	Or
&	And
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

For example:

```
a = 130;
b = a+5;
```

Resultado: La variable *b* vale 135.

Program structure

All instructions end with a semicolon ***";"***.

It is very important to keep in mind that the program runs in a loop: when the last statement is reached, it starts over. Because of this, there is a statement called ***"start"*** that runs only once when the program starts. Inside this block, initial values for variables or any other initialization can be placed.

Every program must end with the instruction ***"end;"***.

Below is the typical structure of a program:

```
start
{
    a = 10;
};
a = a + 1;
end;
```

This simple example shows how the ***"start"*** function is used to initialize variable ***"a"*** to 10 only once and then increment it by 1 continuously.
As we can see, all instructions after the ***"start"*** block and before ***"end;"*** are executed cyclically.

Comments:

To add a comment to the program, prepend the text with the ***"#"*** character and end it with ***";"***.

For example:

```
start
{
    a = 10;  #Initialize variable a = 10;
};
a = a + 1;    #Increment variable a;
end;
```

Flow control functions

Flow control specifies the order in which certain lines of code will be executed.
In our case there are 3 functions for flow control: ***"start"***, ***"if-else"*** and ***"end"***.

Function ***"start"***

Used to execute lines of code only once when the program starts.
The instructions to execute are placed between braces, and after the closing brace ***";"*** must be placed.

Syntax:

```
start
{
...;
...;
};
```

Function ***"if-else"***:

This function is used for decision-making. If the condition placed after ***"if"*** is valid, the instructions in the braces below will execute; otherwise the instructions inside ***"else"***.

The execution condition can use the following operators:

Operator	Function
=	Equal
!	Different
	Greater

>	thatn
<	Less than

The condition is written leaving a space after ***"if"***

Syntax:

Only ***"if"***:

```
if condition
{
   ...;
   ...;
};
```

After the closing brace of ***"if"*** a ***";"*** must be placed.

"if-else":

```
if condition
{
   ...;
   ...;
}
else
{
   ...;
   ...;
};
```

In this case, the ***";"*** is placed after the closing brace of ***"else"***, not after ***"if"*** .

Function ***"end"***

Used only to indicate the end of the program; after executing it, the program returns to the first line of code.

Syntax:

```
end;
```

Interface functions (sources/destinations)

These functions allow reading data from various sources and sending data to various destinations.

Function ***"read_io"***

Allows reading indexed parameters from various sources, such as values of digital input channels, digital output channels, analog inputs, etc.

The data "source" must be indicated with a number. If applicable, the "index" parameter specifies the position within that source.
The reading result is stored in a numeric variable.

Syntax:

```
read_io source,numeric_variable,index;
```

The available sources depend on the **device** where the script runs and the script version. New sources may be added in the future.
Refer to the sources/destinations section for more details.

Function ***"write_io"***

Allows writing indexed parameters to various destinations, such as values of digital output channels, pulse outputs, etc.

The "destination" where to write must be indicated with a number. If applicable, the "index" parameter specifies the position within that destination.

The "value" to write can be a number or a numeric variable.

Syntax:

```
write_io destination,index,value;
```

The available destinations depend on the **device** where the script runs and the script version. New destinations may be added in the future.
Refer to the sources/destinations section for more details.

Function ***"read_str"***:

Allows reading strings from various sources, such as data received from a serial port or by SMS. The ***"source"*** must be indicated with a number.

The reading result is stored in two variables: one of type string and one of type numeric, indicating the string length. If there is no data, this parameter will be 0.

Syntax:

```
read_str source,numeric_variable,string_variable;
```

The available sources depend on the **device** where the script runs and the script version. New sources may be added in the future.
Refer to the sources/destinations section for more details.

Function ***"write_str"***:

Allows writing strings to various "destinations", such as the serial port or SMS. The ***"destination"*** must be indicated with a number and the string to send.

Syntax:

```
write_str destination,string;.
```

The available destinations depend on the **device** where the script runs and the script version. New destinations may be added in the future.
Refer to the sources/destinations section for more details.

The string can be a variable or a string written in the function. **This function supports concatenation and inclusion of ASCII values.**

String functions

These functions allow performing various operations on string-type program variables.

Function ***"is_equal"***

Compares a string variable with a string (fixed text or string variable). Returns 0 if they differ and 1 if they are equal.

Syntax:

```
is_equal numeric_variable,string_variable,string;
```

Example:

```
v='Turn off';
is_equal c,v,'Turn off';
if c=1 {
        #The texts are equal;
};
```

Function ***"finish_with"***

Determines if a string variable ends with a particular string (fixed text or string variable). Returns 0 if false or 1 if true

Syntax:

```
finish_with numeric_variable,string_variable,string;
```

Example:

```
v='Turn off';
finish_with c,v,'off';
if c=1 {
        #The string stored in v ends with 'off';
};
```

Function ***"begin_with"***

Determines if a string variable begins with a particular string (fixed text or string variable). Returns 0 if false or 1 if true

Syntax:

```
begin_with numeric_variable,string_variable,string;
```

Example:

```
v='Turn off';
begin_with c,v,'Turn';
if c=1 {
        #The string stored in v starts with 'Turn';
};
```

Function ***"contains"***:

Determines if a string variable contains a particular string (fixed text or string variable). Returns 0 if not found, or the position where it was found if found.

Syntax:

```
contains numeric_variable,string_variable,string;
```

Example:

```
v='Turn off';
contains c,v,'off';
if c>0 {
        #The string stored in v contains the text 'off';
};
```

Function ***"upper"***

Converts all characters of a string variable to uppercase, storing the result in the same variable.

Syntax:

```
upper string_variable;
```

Example:

```
v='Off';
upper v;
#Now v equals 'OFF';
```

Function *"lower"*

Converts all characters of a string variable to lowercase, storing the result in the same variable.

Syntax:

```
lower string_variable;
```

Example:

```
v='Off';
lower v;
#Now v equals 'off';
```

Function *"strlen"*

Returns in a numeric variable the length of a *string* variable.

Syntax:

```
strlen numeric_variable,string_variable;
```

Example:

```
v='Apagar';
strlen c,v;
#Now c equals 6;
```

Function *"substr"*

Returns a part of a string variable within the same variable

Syntax:

```
substr start,end,string_variable;
```

Example:

```
v='APAGAR BOMBA';
substr 2,3,v;
#Now v equals 'PA';
```

Conversion functions

These functions convert variables from one type to another.

Function *"point"*

Places the decimal point on a numeric variable and converts it to a *string*.
Parameters are: the numeric variable, the string variable, and the number of decimal places.

Syntax:

```
point string_variable,numeric_variable,decimal_places;
```

Example:

```
c=123;
point v,c,1;
#Now v equals '12.3';
```

Function *"aton"*

Converts a number within a string to a numeric variable. Starts at the beginning of the string and ends where a non-numeric value or end of string is found.

Syntax:

```
aton numeric_variable,string_variable;
```

Example:

```
v='123 RPM';
aton c,v;
#Now c equals 123;
```

Funciones *"day","month","year","hs","min","sec"* y *"nday"*

These functions convert a ***time_stamp*** to day, month, year, hour, minute, second and day of week number, respectively.
As seen previously, the current date/time is read with the ***read_io*** type **7** function and stored in a numeric variable representing the number of seconds since 01/01/2000.

Syntax:

```
day day,timestamp;

month month,timestamp;

year year,timestamp;

hs hour,timestamp;

min minutes,timestamp;

sec seconds,timestamp;

nday day_of_week,timestamp;
```

The ***"nday"*** function returns the day of the week number starting from Sunday = 0.

Example:

```
read_io 7,e,0; #Read the current time into e;
day f,e;
month g,e;
year h,e;
hs i,e;
min j,e;
sec k,e;
#The current date and time is f/g/h i:j:k;
```

Mathematical and logical functions

These functions perform certain special mathematical operations.

Function *"neg"*

Used to negate a numeric variable. Negation is performed at the bit level.

Syntax:

```
neg resultado,inicio;
```

First place the variable in which to obtain the result, then the variable to negate.

Example:

```
a=32323; #7E43h
neg b,a;

# b vale 4294934972 (FFFF81BC);
```

Function *"sqtr"*

Performs the square root. Since Exemys Script only handles integer variables of type "long", the decimal part of the result will be truncated. To avoid this, it is recommended to multiply before performing the operation.

Syntax:

```
sqrt result,base;
```

Example:

```
a=225;
sqrt b,a;
#Now b equals 15;
```

Function *"scale"*

Allows scaling a variable using the equation of a line passing through 2 points.

Syntax:

```
scale resultado,inicio,x0,x1,y0,y1;
```

Example: Scale the 4-20mA signal from input AN1 to a value from 0 to 500

```
read_io 2,a,1; #a = AN1;
scale c,a,400,2000,0,500;
#Now c holds the scaled value;
```

Timer functions

These functions allow controlling program flow based on time intervals.

Functions *"timer"* and *"check_timer"*

These functions allow defining a time period and executing instructions when it elapses.
The usage is as follows: first, execute the ***"timer"*** function, passing a numeric variable and a time in milliseconds as parameters. Then query that variable with the ***"check_timer"*** function.

Syntax:

```
timer numeric_variable,time_in_milliseconds;
...
check_timer numeric_variable
{
    ...
    ...
```

```
};
```

As we can see, with the ***"timer"*** function we define the time period, and with ***"check_timer"*** we query the previously loaded variable. When the period elapses, the instructions placed between braces execute.
Keep in mind that once the period elapses, the condition will always be true, and if ***"check_timer"*** is called again, the instructions will execute again. The variable is normally reloaded inside the ***"check_timer"*** block.

Note: Timer functions should not be used in time-precision applications, as the timer may exhibit some jitter.

2026-05-05

Introduction

Using the read_io, write_io, read_str and write_str functions, multiple additional functions can be accessed. This section of the manual lists the different sources/destinations and then explains their use by function.

Source/destination list

⚠ The firmware version from which a source/destination is available is indicated in case it was added after the initial version.

Source / Destination	Index	Value	R/W	Description	Function	Firmware
2	1 to 100	-	read_io	GPS registers	GPS	10.2
5	2	0 / 1	write_io	Enable internal Modbus map	Modbus slave	10.2
	3	1 to 247	write_io	Internal Modbus ID - must differ from the internal table ID		10.2
	4	1 to 64000	write_io	Shifts the start address of the internal Modbus map (default 1)		10.2
402	3	1 to 1000	write_io	Modbus buffer - Select and position (Auto-increment for writing)		10.2
403	1	10^N -2 -> 0.01 -1 -> 0.1 0 -> 1 1 -> 10 2 -> 100	write_io	Buffers - Configure exponent for floating-point values		10.2
	2	0-> MSB / 1 ->LSB 0-> MSW / 1 ->LSW	write_io	Buffers - Configure byte/word order		10.2
404	1	Number according to format, exponent, and order	read/write_io	Buffers - Unsigned 8-bit integer		10.2
	2		read/write_io	Buffers - Signed 8-bit integer		10.2
	3		read/write_io	Buffers - Unsigned 16-bit integer		10.2
	4		read/write_io	Buffers - Signed 16-bit integer		10.2
	6		read/write_io	Buffers - Signed 32-bit integer		10.2
	7		read/write_io	Buffers - 32-bit floating-point value		10.2
35		-	read/write_str	Script Programmer traces	Traces	10.2
21	1 to 20	0 to 2147483647	read/write_io	Non-volatile memory for numbers	Mem. Non-volatile	10.2
121 to 125		-	read/write_str	Non-volatile memory for text 1 to 5		10.2

2026-05-05

GPS value readings

Source / Destination	Index	Value	R/W	Description	Function
2	1 to 100	-	read_io	Read GPS registers	GPS

Source 2 returns the values read from the GPS

The address of each register matches the corresponding Modbus register.

For example, 1=Latitude, 3=Longitude, 11=Speed.

Refer to the device manual to see the remaining values

Example: Read latitude and store it in variable a

```
read_io 2,a,1;
```

2026-05-05

Internal Modbus Slave

The GPS-MB has groups of Modbus registers. One is the standard group described in the user manual.

The other is dedicated to the script and mainly allows calculations to be performed using GPS register values and made available to the user in Modbus registers.

It has a memory area of 1000 words that can be read or written from the script.

With the factory configuration, the registers range from Holding Register 1 to 1000 (40001 to 41000).

Destination 5 / index 2 enables the internal script slave

Destination 5 / index 3 indicates which ID will be used for this slave. It must be different from those configured in the console.

Destination 5 / index 4 allows shifting the start address of this map. The default value is 1 (40001)

Sources/destinations 403 and 404 used to access the buffer may be shared by other functions.

After choosing the buffer and position with destination 402, the address auto-increments as data is loaded.

Source / Destination	Index	Value	R/W	Description	Function
5	2	0 / 1	write_io	Enable internal Modbus map	Modbus slave
	3	1 to 247	write_io	Internal Modbus ID, it cannot be the same as the internal table ID	
	4	1 to 64000	write_io	Shift the start of the internal Modbus map (default 1)	
402	3	1 to 1000	write_io	Modbus buffer - Select and position (Auto-increment for writing)	
403	1	10^N -2 -> 0.01 -1 -> 0.1 **0 -> 1** 1 -> 10 2 -> 100	write_io	Buffers - Configure exponent for floating-point values	
	2	**0-> MSB** / 1 ->LSB **0-> MSW** / 1 ->LSW	write_io	Buffers - Configure byte/word order	
404	3	Number according	read/write_io	Buffers - Unsigned 16-bit integer	
	4	to format,	read/write_io	Buffers - Signed 16-bit integer	
	6	exponent, and	read/write_io	Buffers - Signed 32-bit integer	
	7	order.	read/write_io	Buffers - 32-bit floating-point value	

Write example: Enable the internal script slave with ID 10 tond initial address 1. Then load the following values into Modbus registers starting from 40001

Modbus register	Value (orden byte/word)	Hexadecimal value
40001	-3000 (MSB)	F448
40002	1000 (MSB)	03E8
40003:4	-70000 (MSW)	FFFEEE90
40005:6	70000 (MSW)	00011170
40007:8	-70000 (LSB)	EE90FFFE
40009:10	70000 (LSB)	11700001
40011	1000 (LSB)	E803

```
start {
  write_io 5,2,1; #Enable Modbus slave;
  write_io 5,3,10; #Modbus slave ID 10;
  write_io 5,4,1; #Initial address 1 (40001);
};

write_io 402,3,1; #Select Modbus buffer and position at address 1 (41001);

write_io 403,2,0;#MSB;
write_io 404,4,-3000; #40001;
write_io 404,4,1000; #40002;
write_io 403,2,0;#MSW;
write_io 404,6,-70000; #40003:4;
write_io 404,6,70000; #40005:6;
write_io 403,2,1;#LSW;
write_io 404,6,-70000; #40007:8;
write_io 404,6,70000; #40009:10;
write_io 403,2,1;#LSB;
write_io 404,4,1000; #40011;

end;
```

Read example: Enable the internal script slave with ID 10 tond initial address 1. Read the values of Modbus registers 40001 to 40011 and load them into variables with different formats.

Modbus register	Hexadecimal value	Variable value
40001	F448	a=-3000
40002	03E8	b=1000
40003:4	FFFEEE90	c=-70000
40005:6	00011170	d=70000
40007:8	EE90FFFE	e=-70000
40009:10	11700001	f=70000
40011	E803	g=1000

```
start {
  write_io 5,2,1; #Enable Modbus slave;
  write_io 5,3,10; #Modbus slave ID 10;
  write_io 5,4,1; #Initial address 1 (40001);
};

write_io 402,3,1; #Select Modbus buffer and position at address 1 (41001);

write_io 403,2,0;#MSB;
write_io 402,3,1; read_io 404,a,4; #41001;
write_io 402,3,2; read_io 404,b,4; #41002;
write_io 403,2,0;#MSW;
write_io 402,3,3; read_io 404,c,6; #41003:4;
write_io 402,3,5; read_io 404,d,6; #41005:6;
write_io 403,2,1;#LSW;
write_io 402,3,7; read_io 404,e,6; #41007:8;
write_io 402,3,9; read_io 404,f,6; #41009:10;
write_io 403,2,1;#LSB;
write_io 402,3,11; read_io 404,g,4; #41011;

end;
```

Complete example: Reading GPS data, calculating, and writing the result to the Modbus slave.

```
start {
  write_io 5,2,1; #Enable Modbus slave;
  write_io 5,3,10; #Modbus slave ID 10;
  write_io 5,4,1; #Initial address 1 (40001);
};

read_io 2,a,11; #a=velocidad en kph x 10;

#supera los 60.0 kph?;
if a>600 { b=1; } else { b=0; };

write_io 402,3,1; #Select Modbus buffer and position at the initial address;
write_io 403,2,0;#MSB;
write_io 404,4,b; #40001;

end;
```

2026-05-05

Transmission/Reception of messages to Script Programmer ("Traces")

Source /Destination	R/W	Description	Function
35	read/write_str	Script Programmer Traces	Traces

Destination 35 allows sending text to the "Traces" window in the Script Programmer. This is particularly useful when testing a new script.

There is a consideration regarding space and underscore characters. The space character will be replaced by an underscore before reading with *read_str 35*. The underscore will be replaced by a space after sending with *write_str 35*.

If the Script Programmer is not connected when writing to destination 35, the text will simply be lost but will not affect the operation of the program.

2024-04-23

Non-volatile memory read/write

The device can store text or numbers in non-volatile memory to preserve values even when the device is powered off.

Source / Destination	Index	Value	R/W	Description	Function
21	1 to 20	0 to 2147483647	read/write_io	Non-volatile memory for numbers	Mem. Non-volatile
121 to 125	-		read/write_str	Non-volatile memory for text 1 to 5	

For numbers

Source/destination 21 allows reading and writing numeric values in the device's non-volatile memory.

Example: Read the numeric value stored at position 15 of non-volatile memory into variable g

```
read_io 21,g,15;
```

For text

Sources/destinations 121 to 125 allow reading and writing up to 5 texts in the device's non-volatile memory.

Example: Write the word 'hello into the third position of non-volatile text memory.

```
write_str 123,'hello';
```

2026-05-05