Table of Contents

Description

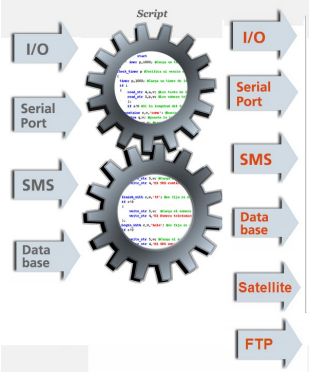
The GRD-XF allows the user to run programs on the device, making it more flexible and powerful.

The GRD-XF will continue to operate normally while the script loaded in its memory is running.

The following lists some of the operations that can be performed by script. See the rest of the documentation for additional features.

Operations that can be performed from a script

- Mathematical **operations**
- Logical **operations**
- **Timing** operations
- Reading the device's own inputs and Modbus variables
- Turning digital outputs on and off
- Sending and receiving **SMS**
- Sending through client **FTP**
- Parsing data from the **serial port**
- Sending data through an external **satellite** modem



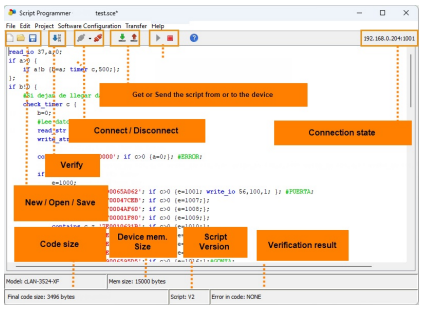
2026-04-13

Introduction

This program allows you to develop, compile and transfer the script to the GRD-XF. To use it, first make sure that the "GRDconfig" is working correctly and that you can communicate with the GRD-XF.

Software Description

The following is a description of the main screen functions.



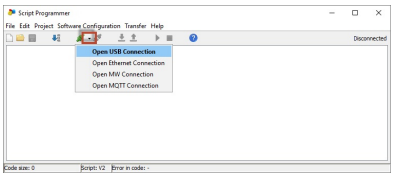
Device Connection

There are two ways to connect: via USB and via MW-XF.

Device Connection - GRD-XF via USB

First connect the GRD-XF to the PC and make sure it is disconnected from ***"GRDconfig"***

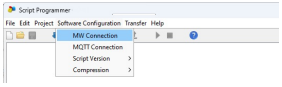
Then expand the Connect button and choose ***"Open USB Connection"***



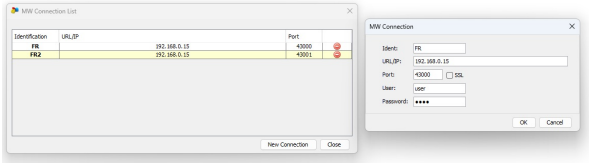
Device Connection - via MW-XF

The device must be connected to Middleware to be configured remotely. **The Middleware version must be 4.2.0 or later to support remote script upload/download.**

You must load in the configurator the parameters of the MW(s) you want to connect to in order to remotely access the GRDs connected to them. To do this, go to *"Software Configuration -> MW Connection"*

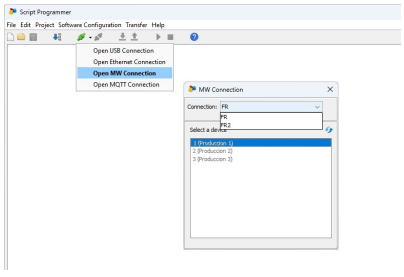


Then add as many URL/IP, port, and credentials as necessary by clicking "New Connection"



To connect to a device remotely, select *"Open MW Connection"* from the connection button. A window will then appear listing the configured MW connections and the available devices.

Note that some devices appear in gray and others in black; black devices are available, while gray devices are not connected to the MW at that moment.

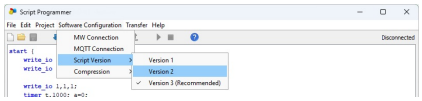


Script Versions 1, 2 and 3

In the ***"Project"*** menu, select ***"Properties"***, then the ***"Script"*** tab to choose script version 1, 2, or 3.

Only GRD-XF-2G and cLAN V1.x work with version 1. The others work with version 2, and from version 11.0 onward they automatically switch to version 3. Scripts written in version 3 are identical to version 2 scripts; they simply use less device memory.

Version 2/3 differs from version 1 because it allows twice as many variables, since they may be lowercase or uppercase.

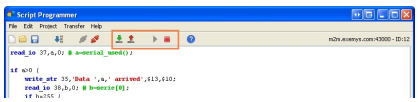


The script version is used by the Script Programmer in two cases: when it verifies a script, and when it tries to send it to the device.

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

Uploading and downloading scripts

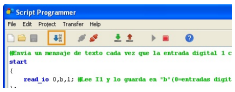
Once connected to the device, you can transfer and download scripts. You can also stop and start them.



Editing scripts

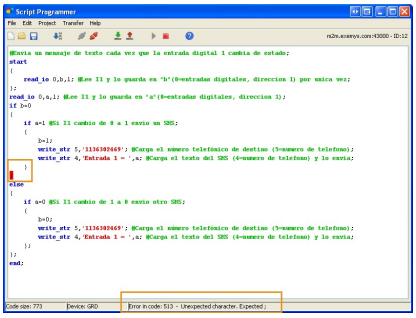
To develop a program, simply write the code in the editing panel. The environment provides function help and highlights correct syntax.

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



When compiling, the lower section will indicate whether there are errors. If there is an error, the line will be marked in red and the ***"Error in Code:"*** field will show the line. If there are no errors, a message will appear and it will show ***"Error in Code: NONE"*** .

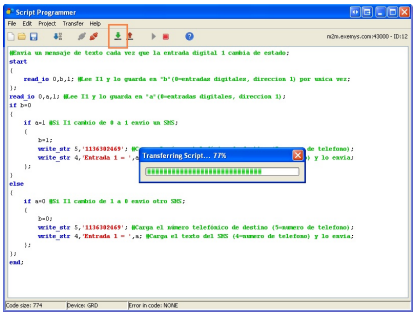
The image below shows a program with an error; in this case a ***";"*** is missing.

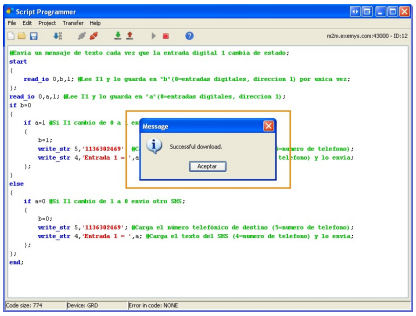


The semicolon is missing in the bracket before the one marked in red. This happens because the compiler encountered an unexpected character.

The image below shows a program without errors.

If there are no errors, you can transfer the program to the device by clicking ***"Download to device"***. A window will show the download status, and then a message will indicate whether the download was successful.





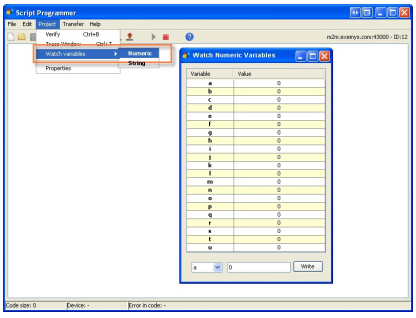
Script debugging

The Script Programmer provides two tools for debugging written scripts. GRD devices must have firmware version 5.2.0 or higher to support these options. The cLAN always has these options available.

Variable monitoring

With this tool you can see numeric or text variable values while the program is running. You can also modify variable values to simulate script operating conditions.

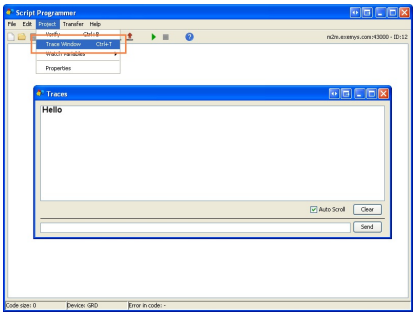
Once connected to the device, go to the **"Project"** menu, select **"Watch variables"**, and then choose **"Numeric"** or **"String"** depending on the type of variable to monitor.



Trace sending and receiving

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

Once connected to the device, go to the **"Project"** menu and select **"Trace Window"**.



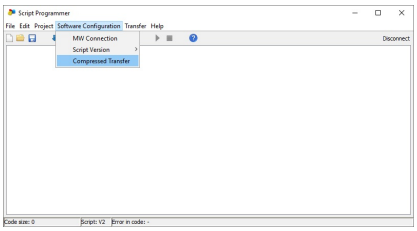
Script compression

The device has limited space for storing the script. The maximum is 15,000 characters.

If this space is not enough 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 your program comments, save a copy on your computer.

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



2026-04-13

Introduction

The Exemys Script programming language is loop-based, which means it executes all code to the last line and then starts again.

There are no statements to create loops within the program, so you cannot stop the program flow in a section of code. In this sense it resembles PLC programming, although its syntax is closer to C or Pascal.

In addition to reading this manual, we recommend reading example scripts. You can download examples from this link www.exemys.com/GRDscriptsExamples

All these examples also work on cLAN devices, except those that use SMS.

Script Versions 1, 2 and 3

There are 3 script versions. Version 2 differs from version 1 because 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 and 30% of the memory used by the program in the device, allowing longer programs to be loaded. You can notice the difference by pressing the verify button.

GRD-XF-2G and cLAN V1.x devices work with version 1

GRD-XF-3G and cLAN V2.x devices work with version 2 (up to version 10.x) and version 3 from 11.0 onward

GRD-MQ and cLAN MQ devices work with version 2 (up to version 10.x) and version 3 from 11.0 onward

Version 3 is fully compatible with version 2 and switches automatically to 3 on compatible devices (V11.0 or later).

Programs written in version 3 are identical to those in version 2. They simply occupy less space in device memory.

Variables

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

It is not necessary to define variables because there is a fixed amount.

In script **version 1** numeric variables are 21, from ***"a"*** to ***"u"***. Text variables are 5, from ***"v"*** to ***"z"***, with a maximum length of 100 characters each.

In script **version 2/3** 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.

Because numeric variables are of type "long", any operation that produces a result with decimals will be truncated.

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

Numeric variables can be "mapped" to be read in some way. In the GRD they can be linked to input and output channels.

Assign a value to variable:

- Variables numéricas:

```
a = 652;
```

- Variables de texto:

```
v = 'Hello World'
```

Please note that text must be placed between single quotes.

Concatenation de strings:

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

For example:

```
a = 20;
u = 'Temperature ';
v = ' °C';
```

If we want to form the phrase 'Temperature 20 °C ' and store it in another variable, we do the following:

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

Another way to do it would be:

```
w = 'Temperature ', a,' °C';
```

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

Inserting ASCII values into strings:

If you want to insert an ASCII value into a string, you can use the $ operator. After the operator you must specify the ASCII code in decimal. ASCII 0 is not allowed.

For example:

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

Inserting ASCII values can only be done when assigning text to string variables and in the write_str function

Variable Aliases

Since Script Programmer version **6.1**, you can create variable "aliases" to make the code easier to read.

This code can be transferred to ALL Exemys devices that support Script Programmer, because before sending the code to the device, the alias will be replaced with the corresponding variable.

Because aliases are sent as comments inside the code, when reading the loaded script from the device, the variables will be replaced by the aliases again, 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:

Antes

```
read_io 2,a,1;
if a>b {
write_io 1,8,1;
};
end;
```

Ahora

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

If you have an existing script, you can add comments specifying the aliases, send it to the device (uncompressed), and then read it back. Once this is done, all variables with assigned aliases will be replaced.

Arithmetic operators

Operator	Description
=	Assignment
^	Exponential
|	Bitwise Or
&	Bitwise And
+	Addition
-	Subtraccion
*	Multiplication
/	Division
%	Modulo

Example:

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

Result: *b* variable value is 135.

Program structure

Al instruction must end with the ***";"*** symbol.

The program runs in a loop. This means that it will run until the last program line and start from the beginning again.

The script last instruction must be ***"end;"***

```
a = a + 1;
end;
```

On this example "a" variable will be incremented constantly. Its initial value is 0.

On-line comments:

If you wish to add a comment line you must use the ***"#"***. On-line comments must also en with the ***";"*** symbol.

Flow control functions

"start" function

It marks a block that will be executed only once. It must be written at the beginning of the script.

Syntax:

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

Example:

```
start
{
    a = 10;  #a initial value is 10;
};
a = a + 1;   #a is incremented by 1 constantly;
end;
```

"if-else" function

The script will decide the script execution flow based on condition. If the condition is true the code in the block next to the "***if***" instruction will be executed. You can add also a code block that will be executed if the condition is no true.

The condition operators are the following ones:

Operator	Description
=	Equals to
!	Not equal to
>	Greater than
<	Less than

Syntax:

Single ***"if"***:

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

A **";"** symbol is required to close the block.

"if-else":

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

The **";"** is only requiered on the ***"else"*** block.

"end" function

This function is used to mark the end of the program. When the interpreter finds this line it will jump to the first line of the script.

Syntax:

```
end;
```

Interface functions

"read_io" funcion

With ***read_io*** you can get values from different sources like I/O channels, the real time clock, etc

The ***"source"*** is indicated with a number. Some sources will require an index number to point an address inside that source.

The result of this function will be loaded in the indicated numeric variable.

Syntax:

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

Available sources may change depending on the device where you are running the script and the script version. New sources can be added in the future.
Browse "Sources-Destinations" section for the currently available ones.

"write_io" function

With ***write_io*** you can set values in different destinations, like digital output channels, pulse channels, etc.

The ***"destination"*** is indicated with a number. Some destinations will require an index number to point an address inside that destination.

The value to be written can be a number or a numeric variable.

Syntax:

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

Available destinations may change depending on the device where you are running the script and the script version. New destinations can be added in the future.

Browse "Sources-Destinations" section for the currently available ones.

"read_str" function

With ***read_str*** you can get incoming strings from different sources like the serial port or a SMS.

The ***"source"*** is indicated with a number.

The result of this function will be loaded in the indicated string and numeric variables. The numeric variable will contain the string length. If the value is 0 it means that there isn't a new incoming string from that source.

Syntax:

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

Available sources may change depending on the device where you are running the script and the script version. New sources can be added in the future.

Browse "Sources-Destinations" section for the currently available ones.

"write_str" function

With ***write_str*** you can send strings to different destinations, like an SMS or the serial port. The ***"destination"*** is indicated with a number.

Syntax:

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

Available destinations may change depending on the device where you are running the script and the script version. New destinations can be added in the future.

Browse "Sources-Destinations" section for the currently available ones.

The string can be a variable string or a text typed between single quotes. **This function support string concatenation and including ASCII values.**

String functions

"is_equal" function

Compares one string variables with a text (variable string or a text typed between single quotes). The numeric variable will contain the result, 1 if they are equal or 0 if they are different.

Syntax:

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

Example:

```
v='PUMP RUN';
is_equal c,v,'PUMP RUN';
if c=1 {
        #texts are equal;
};
```

"finish_with" function

Compares the end of one string variables with a text (variable string or a text typed between single quotes). The numeric variable will contain the result, 1 if they match or 0 if they don't.

Syntax:

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

Example:

```
v='PUMP RUN';
finish_with c,v,'RUN';
if c=1 {
        #The string ends with 'RUN';
};
```

"begin_with" function

Compares the beginning of one string variables with a text (variable string or a text typed between single quotes). The numeric variable will contain the result, 1 if they match or 0 if they don't.

Syntax:

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

Example:

```
v='PUMP RUN';
end_with c,v,'RUN';
if c=1 {
        #The string ends with 'RUN';
};
```

"contains" function

Determines if one string (fixed text or string variable) is contained by a string variable. The numeric variable will contain the position where the string if found or 0 if its not contained.

Syntax:

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

Example:

```
v='PUMP RUN';
contains c,v,'MP';
if c>0 {
        #The variable v contains the text 'MP' ;
};
```

"upper" function

Converts all character is one string variable to uppercase.

Syntax:

```
upper string_variable;
```

Example:

```
v='Turn ON';
upper v;
#v equals 'TURN ON';
```

"lower" function

Converts all character is one string variable to lowercase.

Syntax:

```
lower string_variable;
```

Example:

```
v='Turn ON';
upper v;
#v equals 'turn on';
```

"strlen" function

Gets the string length and stores it on a numeric variable.

Syntax:

```
strlen numeric_variable,string_variable;
```

Example:

```
v='PUMP RUN';
strlen c,v;
#c equals 8 ;
```

"substr" function

Returns part of a string within the same string variable

Syntax:

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

```
v='PUMP RUN';
substr 2,3,v;
#v equals 'UMP';
```

Conversion functions

"point" function

Converts a numeric variable to string and places a decimal point on a fixed position.

Syntax:

```
point string_variable,numeric_variable,decimals;
```

Example:

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

"aton" function

Converts number inside a string variable to a numeric variable. It starts at the beginning of the string and ends where it finds a non-numeric character or reaches the end of the string.

Syntax:

```
aton numeric_variable,string_variable;
```

Example:

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

"day","month","year","hs","min","sec" and *"nday"* functions

These functions will convert a ***time_stamp*** to day, month, year, hour, minute, seconds or day of the week.

Current data/time can be read using ***read_io*** with source #7.

Syntax:

```
day day,timestamp;

month mont,timestamp;

year year,timestamp;

hs hour,timestamp;

min minutes,timestamp;

sec seconds,timestamp;

nday dayoftheweek,timestamp;
```

"nday" function will return the day of the week number starting with Sunday=0s.

Example:

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

Mathematical and logic functions

"neg" function

It will invert the value of a numeric variable bitwise.

Syntax:

```
neg result,initialvalue;
```

Example:

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

# b equals 4294934972 (FFFF81BC);
```

"sqtr" function

Calculates the square root of a numeric variable. As numeric values are integers the fractional part will be truncated. Multiply the number before calculation if you need higher precision.

Syntax:

```
sqrt result,initialvalue;
```

Example:

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

"scale" function

Scales a number using the two point form of the linear equation.

Syntax:

```
scale result,initialvalue,x0,x1,y0,y1;
```

Example: Scale a 4-20mA signal on input AN1 to a number between 0 and 500

```
read_io 2,a,1; #a = AN1;
scale c,a,400,2000,0,500;
#c equals scaled number
```

Timming functions

This functions will allow you to control the program flow using timers.

"timer" and *"check_timer"* function

Use ***"timer"*** to store on a numeric variable the time you want to wait (in milliseconds)

Use ***"check_timer"*** to check if the time has expired or not.

Syntax:

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

Once the time has expired the code inside the *check_timer* block will be executed. This code will be executed on every program loop until the timer is loaded again. Typically you will be reloaded the timer inside the *check_timer* block.

Note: The timing functions are no recommend on applications where precision timing is required because timers can have some dispersion.

2026-03-19

Introduction

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

List of sources/destinations for GRD-XF

The 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	2G	3G/4G
0	1 to 100	-	read_io	Digital input channel (Ix)	Channels	Yes	Yes
1	1 to 100	-	read/write_io	Digital output channel (Ox)		Yes	Yes
2	1 to 100	-	read_io/write_io	Analog input channel (ANx)		Yes/5.1.3	Yes
3	1 to 100	-	read/write_io	Pulse input channel (PIx)		Yes	Yes
305	1 to 100	-	read/write_io	Channel memory		-	1.9
35	-	-	read/write_str	Script Programmer traces	Traces	5.2.0	Yes
21	1 to 20	-	read/write_io	Non-volatile memory for numbers	Non-volatile Memory	5.1.1	Yes
121 to 125	-	-	read/write_str	Non-volatile memory for text 1 to 5		5.2.2	Yes
270	1 to 100	-	read_io	Read value from Modbus query MB-xxN	Modbus Master - Direct query read	-	1.8
271	1 to 100	-	read_io	Read status of Modbus query MB-xxN (0 Failed, 1 OK)		-	1.8
59	0	-	write_io	Change multiplier of read_io 23/36/72/68/69/73/74 from 1000 to another value (repeat line)	Modbus Master - Float 32 Channels	5.2.5B3	Yes
23	1 to 100	-	read_io	Channel P with Modbus Float32 query. Returns integer part of value multiplied by 1000. See write_io 59.		5.1.2	Yes
36	1 to 100	-	read_io	Channel P with Modbus Float32 query with swapped words. Returns integer part of value multiplied by 1000. See write_io 59.		5.2.0	Yes
59	0	-	write_io	Change multiplier of read_io 23/36/72/68/69/73/74 from 1000 to another value (repeat line)	-	5.2.5B3	Yes
9	0	Table	read_io	GSM connection status	GSM	Yes	Yes
280	0	0 / 1	read_io	Roaming status (0 no, 1 yes)		-	1.8
10	0	Table	read_io	GPRS connection status	GPRS	Yes	Yes
22	0	0 / 1	read/write_io	Device configuration enabling MW connection (0 or 1)	MW	5.2.2	Yes
11	0	Table	read_io	MW connection status		Yes	Yes
39	0	-	read_io	Number of SMS in outbox	SMS	5.2.4	Yes
4	-	-	read_str	SMS text		Yes	Yes
5	-	-	read_str	SMS phone number		Yes	Yes
101 to 108	-	-	read_str	Phone book names 1 to 8		5.1.3	Yes
111 to 118	-	-	read_str	Phone book numbers 1 to 8		5.1.3	Yes
4	-	-	write_str	SMS text		Yes	Yes
5	-	-	write_str	SMS phone number		Yes	Yes
17	1 to 100	0	write_io	From DIx channel	Reports, forced	Yes	Yes
18	1 to 100	0	write_io	From DOx channel		Yes	Yes
19	1 to 100	0	write_io	From ANx channel		Yes	Yes
20	1 to 100	0	write_io	From PIx channel		Yes	Yes
48	0	-	write_io	Disable sending historical records to MW (1 disabled, 0 enabled)	Historical records	5.2.2	Yes
12	1 to 100	Value	write_io	By time of channel ANx	Historical records, generation	Yes	Yes
13	1 to 100	Value	write_io	By time of PIx channel		Yes	Yes
14	1 to 100	Value	write_io	Maximum alarm of channel ANx		Yes	Yes
15	1 to 100	Value	write_io	Minimum alarm of channel ANx		Yes	Yes
16	1 to 100	Value	write_io	Normal alarm of ANx channel		Yes	Yes
56	1 to 100	0/1	write_io	From DIx channel change		5.2.2	1.6
57	1 to 100	0/1	write_io	From DOx channel change		5.2.2	1.6
8	0	-	read_io	Number of unsent historical records stored in memory	Historical records, memory access	Yes	Yes
60	-	-	write_io	Read in memory the data of a specific record from historical memory (use together with read_io 61 to 65)		5.2.2	Yes
61	0	-	read_io	'channel type' of record read with write_io 60		5.2.2	Yes
62	0	-	read_io	'timestamp' of record read with write_io 60		5.2.2	Yes
63	0	-	read_io	'historical type' of record read with write_io 60		5.2.2	Yes
64	0	-	read_io	'channel number' of record read with write_io 60		5.2.2	Yes
65	0	-	read_io	'value' of record read with write_io 60		5.2.2	Yes
66	-	-	write_io	Delete the first N records from historical memory		5.2.2	Yes
7	0	-	read_io	Current time (seconds since 1/1/2000)	Clock	Yes	Yes
7	0	-	write_io	Set current time (seconds since 1/1/2000)		-	1.8
37	0	-	read_io	Amount of data in serial port buffer (Data is cleared from buffer with write_io 37)	Serial port	5.2.0	Yes
37	0	-	write_io	Delete first N data from serial port buffer (use together with read_io 37 and read_io 38)		5.2.0	Yes
38	0 to 199	0-255	read_io	Read the binary value from the indicated position in the serial port buffer		5.2.0	Yes
38	0	0-255	write_io	Send a byte to serial port		5.2.5B3	Yes
6	-	-	read/write_str	Serial port send/receive	Serial port, text mode	Yes	Yes
32	0	-	write_io	Check if data was sent to satellite modem (consumes data)	Iridium Satellite SBD	-	1.3
54	0	-	write_io	Starts sending historical records via satellite modem		5.2.2	Yes
55	0	Table	read_io	Satellite modem send status.		5.2.2	Yes
32	-	-	read_str	Reception of text via SFIELD sent through transparent serial port		-	1.3
29	-	-	write_str	Load string transmission via satellite modem to transparent port (use with write_io 31)		-	1.9
31	0	-	write_io	Trigger string transmission via satellite modem to transparent port (use with write_str 29) MW 5.1.0		-	1.9
50	-	-	write_str	Load process buffer (use together with read_str 51)	Parsing	5.2.0	Yes
51	-	-	read_str	Read process buffer with added start, end and NMEA checksum (load process buffer first with write_str 50)		5.2.0	Yes
44	0	-	write_io	Start client connection FTP	FTP Client	5.2.2	10.1
46	0	-	write_io	Close file and terminate client connection FTP		5.2.2	10.1
47	0	Table	read_io	FTP client status		5.2.2	10.1
40	-	-	write_str	Load URL for FTP client		5.2.2	10.1
41	-	-	write_str	Load FTP user		5.2.2	10.1
42	-	-	write_str	Load password FTP		5.2.2	10.1
43	-	-	write_str	Load filename for FTP client		5.2.2	10.1
45	-	-	write_str	Load text line in FTP file and send it		5.2.2	10.1

2026-04-09

Read/Write input/output channels

Source/ Destination	Index	Value	R/W	Description	Function
0	1 a 100	-	read_io	Digital input channel (Ix)	Channels
1	1 a 100	-	read/write_io	Digital output channel (Ox)	
2	1 a 100	-	read_io/write_io	Analog input channel (ANx)	
3	1 a 100	-	read/write_io	Pulse input channel (PIx)	
305	1 a 100	-	read/write_io	Channel memory	

Sources 0 to 3 return the values of the device's different input/output channels. The **"index"** indicates the channel number to read.

Example: Read the value of analog input channel 4 (AN4) and save it in variable c

```
read_io 2,c,4;
```

Destination 0 allows activating the device outputs. The **"index"** indicates the channel number to write.

Example: Turn off digital output channel 3 (O3)

```
write_io 1,3,0;
```

Destination 2 of **analog input channels** allows writing only if they are linked to Modbus queries. Calling write_io will generate a Modbus write command.

Destination 3 of **pulse channels** accepts the same values supported by the device on those channels (physical counters or Modbus query of length 2).

Sources 23 and 36 allow converting Modbus Float 32 registers linked to pulse channels to integer values.

Channel memory

This volatile memory area with 100 positions can function as a "Source" for all channels.

The value of this memory can be read and written with `read_io/write_io 305`

This allows freeing script numeric variables that were previously used to link to channels.

2026-04-09

Sending/Receiving messages to the Script Programmer ("Traces")

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

Destination 35 allows you to send text to the "Traces" window in the Script Programmer (available since Script Programmer V2.0). 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.

2026-04-09

Read/Write non-volatile memory

For numbers

Source/ Destination	Index	Value	R/W	Description	Function
21	1 a 20	-	read/write_io	Non-volatile memory for numbers	Non-volatile memory

The source/destination 21 allows reading and writing up to 20 numeric values in the device's non-volatile memory.

In devices with firmware 5.1.1, do not invoke this function permanently, only when the value changes (there is a risk of damaging the memory).

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

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

For text

Source/ Destination	Index	Value	R/W	Description	Function
121 a 125	-	-	read/write_str	Non-volatile memory for text 1 to 5	Non-volatile memory

The 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' in the third position of the non-volatile memory for texts.

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

2026-04-09

Direct reading of Modbus queries

read_io	Description	Index
270	Direct reading of Modbus queries - Value	1 to 100
271	Direct reading of Modbus queries - Status	1 to 100

Sources 270 and 271 allow reading the value of a Modbus query without previously mapping that query to a channel.

This avoids wasting channels by using them as an intermediate point before processing values in the script.

Source 271 returns 0 if there is a Modbus communication failure or 1 if communication has no errors.

Example: Read the value of Modbus query 4 and store it in variable c

```
read_io 270,c,4;
```

2026-04-09

[Reading GSM/GPRS Status in GRD]

Status Reading

Source/ Destination	Index	Value	R/W	Description	Function
9	0	Table	read_io	GSM connection status	GSM
280	0	0 / 1	read_io	Roaming status (0 no, 1 yes)	
10	0	Tabla	read_io	GPRS connection status	GPRS

From sources 9, 10 and 280 you can read the GSM and GPRS status. The possible states are listed below.

GSM States

#	GSM State
0	OFF
1	ATTACHING
2	SIM NOT INSERTED
3	PIN REQUIRED
4	PIN ERROR
5	PIN OK
6	BLOCKED
7	**LOW SIGNAL**
8	ACCESS DENIED
9	**READY**

GPRS States

#	GPRS State
0	OFF
1	WAITING FOR GSM READY
2	ATTACHING
3	**CONNECTED**
4	ERROR
5	WAITING FOR RECONNECTION

2026-04-09

[Reading MW status and MW connection configuration]

Source/ Destination	Index	Value	R/W	Description	Function
22	0	0 / 1	read/write_io	Device configuration for MW connection enablement (0 or 1)	MW
11	0	Table	read_io	MW connection status	

Reading states

Middleware States

#	Middleware Status
0	OFF
1	WAIT GPRS READY (GRD only)
2	CONNECTING
3	CONNECTION REFUSED
4	CONNECTION FAILED
5	HOST UNREACHABLE
6	HOST CLOSED CONNECTION (GRD only)
7	**CONNECTED**
8	ERROR
9	WAIT RECONNECTION
10	DNS FAILURE
11	LOGGING IN

MW connection configuration

The source/destination 22 allows configuring whether the device connects to the MW or not.

Note that this configuration is saved in the device's non-volatile memory, and once disconnected from the MW you will not be able to access it remotely anymore.

Disable sending historical records to MW

read_io / write_io	Description	Index
48	Disable sending historical records to MW (1 disabled, 0 enabled)	0

The source/destination 48 allows disabling the sending of historical records to the MW.

Note that this configuration is saved in the device's non-volatile memory, and if you want the device to send historical records again, you must re-enable the sending.

This function is useful when you want to send historical records through some alternative means.

Example: Disable sending historical records to MW.

```
write_io 48,0,1;;
```

2026-04-09

Sending/Receiving SMS. Phone book

Source/ Destination	Index	Value	R/W	Description	Function
39	0	-	read_io	Number of SMS in outbox	SMS
4	-	-	read_str	SMS text	
5	-	-	read_str	SMS phone number	
101 a 108	-	-	read_str	Names of the phone book 1 to 8	
111 a 118	-	-	read_str	Numbers of the phone book 1 to 8	
4	-	-	write_str	SMS text	
5	-	-	write_str	SMS phone number	

Reception

With sources 4 and 5 you can receive text messages. To know if a message arrived you must read the message text constantly until its length is different from 0.

Example: Check if a text message arrives

```
if 1 {
     read_str 4,a,v;
     read_str 5,b,w;
};
if a!0 {
     #An SMS was received;
};
```

Sending

Destinations 4 and 5 can be used to send text messages. The message is sent when writing the text to destination 4. Previously you must load the destination number in destination 5.

If a message was just received and you want to reply, you can write the response directly to destination 4 without first writing the phone number.

Example: Send a text message

```
     write_str 5,'1166041241'; #Load the destination phone number;
     write_str 4,'Hello'; #Load the SMS text and send it;
```

Allows knowing how many SMS are waiting to be sent.

Phone book

Sources 101 to 108 allow reading the names of the GRD phone book

Sources 111 to 118 allow reading the phone numbers of the GRD phone book

These sources are useful if you want to change the recipient of a message sent from the script without having to edit the program.

Example: Read the phone number in position 5 of the GRD phone book

```
    read_str 115,a,v;
```

2026-04-09

Forced reports

Source/ Destination	Index	Value	R/W	Description	Function
17	1 a 100	0	write_io	Of channel DIx	Reports, forced
18	1 a 100	0	write_io	Of channel DOx	
19	1 a 100	0	write_io	Of channel ANx	
20	1 a 100	0	write_io	Of channel PIx	

The destinations 17 to 20 allow forcing the sending of reports, in addition to those generated by the device itself. The value of the report is the one the channel has at that moment. The value field is ignored.

It is recommended to use this function carefully to not generate GPRS traffic permanently (GRD only)

Example: Generate a report of channel AN3 every 10 seconds with its current value

```
check_timer t
{
    timer t,10000;
    write_io 19,3,0;
};
```

2026-04-09

Historical records - Generation

Source/ Destination	Index	Value	R/W	Description	Function
12	1 a 100	Valor	write_io	By time of channel ANx	Historical generation
13	1 a 100	Valor	write_io	By time of channel PIx	
14	1 a 100	Valor	write_io	Of maximum alarm of channel ANx	
15	1 a 100	Valor	write_io	Of minimum alarm of channel ANx	
16	1 a 100	Valor	write_io	Of normal alarm of channel ANx	
56	1 a 100	0/1	write_io	Of change of channel DIx	
57	1 a 100	0/1	write_io	Of change of channel DOx	

These destinations allow generating historical records from the script, in addition to those generated by the device itself. The value of the historical record must be indicated in the value field.

It is recommended to use this function carefully to not generate historical records permanently.

Example: Generate a historical record by time of channel AN2 every 10 seconds with the value 457

```
check_timer t
{
    timer t,10000;
    write_io 12,2,457;
};
```

2026-04-09

Historical records - Access to historical records memory

Source/ Destination	Index	Value	R/W	Description	Function
8	0	-	read_io	Number of unsent historical records stored in memory	Historical records, memory access
60	-	-	write_io	Read in memory the data of a specific record from the historical records memory (use together with read_io 61 to 65)	
61	0	-	read_io	'channel type' of record read with write_io 60	
62	0	-	read_io	'timestamp' of record read with write_io 60	
63	0	-	read_io	'historical type' of record read with write_io 60	
64	0	-	read_io	'channel number' of record read with write_io 60	
65	0	-	read_io	'value' of record read with write_io 60	
66	-	-	write_io	Deletes the first N records from the historical records memory	

The sources/destinations 60 to 66 allow reading the device's historical records memory. These functions are used to be able to send the content of this memory through a means other than sending to the Middleware, for example, via FTP.

Before trying to read a record you can see how many records there are in memory using *read_io 8*. Then you can invoke *write_io 60* to read a particular record and *read_io* 61 to 65 to obtain the values of the fields of the read record. Finally you can use *write_io 66* to delete the read record(s).

Example: Read the historical records from the device's memory and send them to the "Traces" console of the Script Programmer

```
read_io 8,a,0;
if a>0
{
    write_io 60,0,0; #Read the first record from memory (the 2nd 0 indicates the first position);
    read_io 61,b,0; #Load in b the channel type;
    read_io 62,c,0; #Load in c the timestamp;
    read_io 63,d,0; #Load in d the historical type;
    read_io 64,e,0; #Load in e the channel number;
    read_io 65,f,0; #Load in f the value;
    wrb,"-",b,"-",c,"-",d,"-",e,"-",f,$13,$10;
    write_str 35,w; #Send the record text to the console;
    write_io 66,0,1; #Delete the sent record from the device memory;
};
```

For a more complete example of the use of these functions see the FTP usage example.

⚠	*If you are using read_io 8 / write_io 66 to keep the memory with new records use the amount 90000 instead of 100000.*

2026-04-09

Historical records - Disable sending to MW

Source/ Destination	Index	Value	R/W	Description	Function
48	0	-	write_io	Disable sending historical records to MW (1 disabled, 0 enabled)	Historical

The source/destination 48 allows disabling the sending of historical records to MW.

Note that this configuration is saved in the device's non-volatile memory, and that if you want the device to send historical records again, you must re-enable sending.

This function is useful when you want to send historical records through some alternative means.

Example: Disable sending historical records to MW.

```
write_io 48,0,1;;
```

2026-04-09

Reading real time clock

Source/ Destination	Index	Value	R/W	Description	Function
7	0	-	read_io/write_io	Current time (seconds since 1/1/2000)	Clock

The source 7 allows reading the device's current date/time. The number can be converted to text using the conversion functions.

Example: Get the current month in variable g

```
read_io 7,e,0;
month g,e;
```

In recent firmware versions it is also possible to configure the time from the script with `write_io 7`

2026-04-09

Access to serial port in text mode

Source/ Destination	Index	Value	R/W	Description	Function
6	-	-	read/write_str	Sending/Reception of the Serial Port	Serial port, text mode

The source/destination 6 allows receiving and sending text from and to the device's serial port. To know if text arrived at the serial port you must read source 6 constantly until its length is different from 0.

To send a text simply write to destination 6.

For it to work correctly you must configure the serial port in "Script" mode

If you want to send binary characters you can use the $ operator

Example: Make "echo" of the text received by the serial port.

```
read_str 6,a,v;
if a!0 {
    write_str 6,v;
};
```

Access to serial port in binary mode

Source/ Destination	Index	Value	R/W	Description	Function
37	0	-	read_io	Number of data in the serial port buffer (Data is deleted from the buffer with write_io 37)	Serial port
37	0	-	write_io	Delete the first N data from the serial port buffer (use together with read_io 37 and read_io 38)	
38	0 a 199	0-255	read_io	Reads the binary value of the indicated position of the serial port buffer	
38	0	0-255	write_io	Send a byte to the serial port	

The source/destination 37, plus source 38 allow interpreting binary data received in the serial port.

For it to work correctly you must configure the serial port in "Script" mode

If you want to send 'binary' data you can use the destination you can use write_str 6 together with the $ operator

Example: Wait to receive more than 2 bytes. Then check if the third received byte is the binary 126. Finally delete 3 bytes from the buffer

```
read_io 37,a,0;
if a>2 {
    read_io 38,b,3;
    if b=126 {
        #3rd byte received is binary 126;
    };
    write_io 37,0,3;
};
```

Second serial port in script mode

From firmware version 11.3 both serial ports can work in script mode. The sources destinations of the serial port configured in "Script Sec" mode are the same as the main port but adding 1000.

For example, to send data in text mode you will use write_str 1006

2026-01-06

Iridium SBD satellite modem

Source/ Destination	Index	Value	R/W	Description	Function
32	0	-	write_io	Checks if data was sent to the satellite modem (consumes data)	Iridium SBD satellite
54	0	-	write_io	Initiates sending of historical records via satellite modem	
55	0	Table	read_io	Status of sending via satellite modem.	
32	-	-	read_str	Reception of text by SFIELD sent through transparent serial port	
29	-	-	write_str	Loads sending of string via satellite modem to transparent port (use with write_io 31)	
31	0	-	write_io	Triggers sending of string via satellite modem to transparent port (use with write_str 29) MW 5.1.0	

Sending status

#	Sending status
-7	Does not send. Connected to MW
-6	Error sending records
-5	Initializing modem
-4	Error in serial port configuration (Satellite mode and at 19200 bps)
-3	No records to send
-2	Error reading records memory
-1	Sending records
0	Ready to send
> 0	Number of records sent

If you have an Iridium SBD modem (EDGE/ITAS) connected to the GRD/cLAN serial port, you can send the historical records in its memory to the MW using the Iridium satellite network. In the device manual you can see more details of the solution.

It is recommended to use with discretion and know the cost of sending data via satellite modem.

The destination 54 allows initiating the sending of the records that the device has in memory. To initiate sending the modem must be in "Ready to send" state. The GRD/cLAN will send all records it can in a single satellite message.

From version 1.3 of GRD-XF-3G/4G and 2.2 of cLAN data can also be received with the satellite modem. Data does not arrive spontaneously, the GRD/cLAN must check if new data arrived. This is done automatically every time data is sent. A reception check can also be initiated using write_io 32. Note that each time it is checked, Iridium generates a charge. For sending data from MW to GRD/cLAN, the Iridium service called "Static IP addresses for Mobile Terminated SBD" must be contracted and configured in the MW.

Using read_str 32 you can read the text sent using the satellite modem. This text is introduced to the MW using the transparent serial port connection.

Strings can also be sent to the MW's transparent serial port connection by loading the string with write_str 29 and triggering the sending with write_io 31

Every time data is sent (write_io 54) or checked (write_io 32), commands can also be received from the MW generated from the writings table in the database. In this way, digital output channels, Modbus query analogs or channels linked to script variables can be modified remotely.

As an example please look at the satelite.sce file that you can download along with the other script usage examples www.exemys.com/GRDscriptsExamples

2026-04-09

Calculation of checksums and CRCs

Source/ Destination	Index	Value	R/W	Description	Function
50	-	-	write_str	Load process buffer (use together with read_str 51)	Parsing

The destination 50 allows loading a string into the process buffer to then apply some process to the loaded text.

NMEA protocol process

Source/ Destination	Index	Value	R/W	Description	Function
51	-	-	read_str	Reads process buffer with addition of start, end and NMEA checksum (load the process buffer first with write_str 50)	Parsing

The source 51 will save in the variable the NMEA frame previously loaded in the process buffer with *write_str 50*.

This source adds to the original frame the start, end and NMEA checksum. This allows simulating an NMEA "talker" with the device.

Example: Send the NMEA sentence *GPMWV,145.8,R,87.2,K,A* through the device's serial port, after adding the start character, end character and checksum.

```
write_str 50,'GPMWV,145.8,R,87.2,K,A';
read_str 51,a,w;
write_str 6,w;
```

2026-04-09

FTP Client

Source/ Destination	Index	Value	R/W	Description	Function
44	0	-	write_io	Start client connection	FTP Client
46	0	-	write_io	Close file and end client connection	
47	0	Table	read_io	Client status	
40	-	-	write_str	Load URL for client	
41	-	-	write_str	Load username	
42	-	-	write_str	Load password	
43	-	-	write_str	Load filename for client	
45	-	-	write_str	Load text line into file and send it	

States (read_io 47)

#	FTP client status
0	**IDLE**
2	CONNECTING
3	CONNECTION FAIL
7	ERROR
8	SENDING FILE
9	WAIT READY TO SEND
10	**READY TO SEND**

The sources/destinations listed in these tables allow implementing an FTP client for uploading text files to a server. Normally this function will be used to send historical records from the device memory together with the sources/destinations that allow accessing the historical record memory.

To be able to send data via FTP the GRD must be registered on the GPRS network (GRD only).

As an example please see the file *ftp.sce* which you can download along with the other script usage examples from www.exemys.com/GRDscriptsExamples

2026-04-09