PalmSens
Compact Electrochemical Interfaces

MethodSCRIPT v1.2

Rev. 1

Contents

PalmSens

1 Introduction

The MethodSCRIPT scripting language is designed to improve the flexibility of the PalmSens potentiostat and galvanostat devices for OEM users. It allows users to start measurements with arguments that are similar to the arguments in PSTrace.

PalmSens provides libraries and examples for handling low level communication with the EmStat Pico and generating scripts for supported devices.

Terminology

PGStat: Potentiostat / Galvanostat
CE: Counter Electrode
RE: Reference Electrode
WE: Working Electrode
RHS: Right hand side
LHS: Left hand side
Technique: A standard electrochemical technique
Iteration: A single execution of a loop

2 Features

2.1 Features

- Measurements can be tested in PSTrace and then exported to MethodSCRIPT. This allows for convenient testing of different measurements in PSTrace. The resulting MethodSCRIPT can then be easily imported as a text file and executed from within the user application. PSTrace can also run custom scripts and is able to plot the resulting measurement data.
- Support for the following electrochemical techniques:
  - Linear Sweep Voltammetry (LSV)
  - Cyclic Voltammetry (CV)
  - Differential Pulse Voltammetry (DPV)
  - Square Wave Voltammetry (SWV)
  - Normal Pulse Voltammetry (NPV)
  - Chronoamperometry (CA)
  - Open Circuit Potentiometry (OCP)
  - Electrochemical Impedance Spectroscopy (EIS)
  - Pulsed Amperometric Detection (PAD)
- Storing of measuring data to an SD card (if SD card is available).
- Support for BiPot / Poly WE
- Different measurements can be chained after one another in the same script, making it possible to combine multiple measurements without communication overhead.
- Conditional statements (if, else, elseif, endif)
- Variables can be stored and referenced to from within the script.
- Up to 26 variables can be declared. This enables temporary storage of measurement data to be sent later.
- Simple math can be performed on variables (add, sub, mul, div).
- Support for loops.
- Support for user code during a measurement step.
- Exact timing control.
- Script syntax will be verified when loading the script. Runtime errors are checked at execution. If there is an error, the location and error code of the error will be communicated.
- Autorun script stored in persistent memory at start up.
- Low power modes (sleep, hibernate).
- Variables can also be declared as arrays. Up to 4000 variables can be used. This allows for fast burst measurements that are not slowed down by communication.
- Direct control over the I2C interface for communication with external sensors and actuators.

2.2 Planned future features

- Checksum for measurement data packages to check their validity.

2.3 Supported devices

- EmStat Pico

3 Script format

The script consists of a series of pre-defined commands. Each command starts with the command string, followed by a pre-defined number of arguments. Arguments are separated by a ‘ ‘ (space) character. Each command is terminated by a ‘\n’ (newline) character. The ‘\n’ is omitted in most examples. Each line is limited to a maximum of 128 characters. Comments can be added by having the first non-whitespace character on the line be ‘#’.

To send a script to the device, first send “e\n”. This sets the device into MethodSCRIPT mode. To terminate the script, add a line containing only a ‘\n’.

The following example shows a short script that simply declares a variable, including the ‘\n’ characters:

```
e\n
#This is a comment\n
send_string “hello world”\n
\n
```

The response to this script will be:

```
e\n                       ←Ack of the execute script cmd ‘e’
Thello world\n            ←Reply of the “send_string “hello world”” cmd
\n                        ←End of script
```

4 MethodSCRIPT variables

MethodSCRIPT variables represent numerical values that can be used within the script. They can be stored internally as either floating point or as signed integer. Some commands only accept integer variables, others will only accept floating point variables, this is indicated in the command parameter table.

Floating point variables are represented as a signed integer value with an SI prefix. Despite the name, this SI prefix is added after the integer value. See “Table 1: SI prefix conversion table” for the available SI prefixes. Only SI prefixes available in this table can be used. For example, a variable with a value of “100” and a prefix of “m” translates to a floating point value of 0.1. Floating point variables cannot represent every possible value and will be rounded to the nearest representable number. This makes them less suitable for some purposes, such as counters in loops.

SI prefix	Text	Factor
'a'	atto	10^-18
'f'	femto	10^-15
'p'	pico	10^-12
'n'	nano	10^-9
'u'	micro	10^-6
'm'	milli	10^-3
' '	none	10^0
'k'	kilo	10^3
'M'	Mega	10^6
'G'	Giga	10^9
'T'	Tera	10^12
'P'	Peta	10^15
'E'	Exa	10^18

Table 1: SI prefix conversion table

Integer variables end with an 'i' instead of an SI prefix. They are represented as 32 bit signed integers. Integers are not subject to rounding, except when dividing two integers.

Variables are not explicitly linked to a unit; instead the unit is implied by the associated "Variable Type". Refer to section "Variable Types" for more information. Representation of MethodSCRIPT variables changes depending on whether the variable is part of a script command or part of a measurement data package.

Some number input parameters are not MethodSCRIPT variables. These include uint8, uint16, uint32, int8, int16, int32. For these integer parameters, it is allowed but not necessary to append an 'i'. They do not accept SI Prefixes.

4.1 Script command variables

Variables that are part of the MethodSCRIPT sent to the device are represented as a signed integer followed by a prefix for floating point values, or 'i' for integer values. Integer variables can also be entered as a hexadecimal or binary representation by prefixing the value with 0x or 0b respectively. Hexadecimal or binary representations are not allowed for floating point variables.

Example 1:

```
0xFFi
```

Above example shows the hexadecimal representation of the decimal number "255". It is stored internally as an integer because it ends with an 'i'.

Example 2:

```
500m
```

Above example shows the floating point number 0.5. It is stored internally as a floating point number because it has an SI prefix.

4.2 Measurement data package variables

Variables that are part of a measurement data package are represented as 28 bit unsigned hexadecimal values with an offset of 0x8000000 (2^27). A floating point variable has one of the SI prefixes shown in "Table 1: SI prefix conversion table", an integer variable ends with an 'i' instead.

This format looks as follows:

```
HHHHHHHp
```

Where:

`HHHHHHH`	= Hexadecimal value.
p	= Prefix character.

For example, a value of 0.01 would be represented as "800000Am" and a value of -0.01 would be represented as "7FFFFF6m". PalmSens provides source code examples that showcase how to parse measurement data.

To convert a MethodSCRIPT variable to a floating point value, the following pseudocode can be used:

```
(HexToUint32(HHHHHHH) - 2^27) * SIFactorFromPrefix(p)
```

To convert a floating point value to a MethodSCRIPT variable, the following pseudocode can be used:

```
Uint32ToHex(value) / SIFactorFromPrefix(p) + 2^27
```

Most programming languages have a built in way of converting a HEX string to an integer. The function SIFactorFromPrefix can use a hash table lookup or a switch case to translate the prefix character to its corresponding factor.

5 Interpreting measurement data packages

5.1 Package format

Measurement packages consist of a header, followed by any amount of “variable” packages (each with their own “variable type”), followed by a terminating ‘\n’ character. “Table 2: Measurement data package format” shows this format. Section “Variable sub package format” explains the format of the variable fields.

Header	Var 1	Var separator	Var 2	Var separator	Var X	Term
P	Variable	;	Variable	;	Variable	\n

Table 2: Measurement data package format

5.2 Variable sub package format

The format for a variable sub package is:

Var 1	Var 1 metadata 1	Var 1 metadata X
ttHHHHHHHp	,MV..V	,MV..V

Table 3: Variable sub package format

Where:

tt	= Variable Type, represented as a base26 identifier that ranges from “aa” to “zz”. Variable Types are always lower case. See section “Variable Types” for more information.
HHHHHHHp	= MethodSCRIPT package variable. See section “Measurement data package variables” for more information.
,	= Metadata separator.
M	= Metadata type ID, see “Table 4: Metadata types”.
V..V	= Metadata value as a hexadecimal value, length is determined by metadata type.

Metadata fields contain extra information about the variable. Each variable can have multiple metadata fields. See “Table 4: Metadata types” for the possible metadata types.

ID	Name	Length	Content
1	Status	1	0 = OK 1 = timing not met (custom commands in the measurement loop took too long for the specified interval of the measurement) 2 = overload (>95% of max ADC value) 4 = underload (<2% of max ADC value) 8 = overload warning (>80% of max ADC value) If an overload or underload is detected, the measured data can be unreliable.
2	Current range	2	Index of current range (device specific, see “Current ranges”). This current range is just intended for diagnostic purposes, and is not used in any calculations during parsing.

Table 4: Metadata types

5.3 Package parsing example

An EmStat Pico sends the following measurement data package:

```
Pda8000800u;ba8000800u,10,201\n
```

This package contains two variables: “da8000800u” and “ba8000800u,10,201”.

The variable sub package “da8000800u” can be broken down as follows:

- The Variable Type is “da”, this is variable type “VT_CELL_SET_POTENTIAL”.
- The value is “08000800 – 0x8000000” = 0x800 = 2048. The prefix is “u” which stands for “micro”. This makes the final value 2048 uV (or 2.048 mV).
- This variable has no metadata.

The variable sub package “ba8000800u,10,201” can be broken down as follows:

- The Variable Type is “ba”, this corresponds to Variable Type "VT_CURRENT”.
- The value is “08000800 – 0x8000000” = 0x800 = 2048. The prefix is “u” which stands for “micro”. This makes the final value 2048 uA (or 2.048 mA).
- This variable has two metadata packages, the first has an ID of “1” and a value of 0, indicating it is a status package with the value “OK”. The second metadata package has an ID of “2” and a value of 01. This indicates that it is a current range with the current range “1”. For the EmStat Pico, this refers to the “1.95 uA” current range. This current range is just for diagnostic purposes, and is not used in any calculations during parsing.

6 Measurement loop commands

All measurement techniques are implemented as “measurement loop commands”. This means that the command will execute one iteration of the measurement technique. After this, all MethodSCRIPT commands within the measurement loop are executed. When all commands have been executed, the device waits for the correct timing to start the next iteration of the measurement technique and the process begins again for the next iteration.

It is not possible to use a measurement loop inside of another measurement loop. Measurement loops can be used freely inside of a normal loop.

It is possible that the script steps in the loop take more time than is available between each iteration. If this happens, the next measurement iteration is delayed. It is the responsibility of the user to ensure there is enough time between measurement iterations to execute the user commands in the loop.

6.1 Measurement loop example

The following example shows a typical Chrono Amperometry measurement loop:

```
#Run a measurement loop for the Chrono Amperometry technique
meas_loop_ca p c 100m 100m 2
   #These user commands are executed after one measurement
   #iteration has been done
   pck_start
   pck_add p
   pck_add c
   pck_end
   #At “endloop”, the script execution halts until it is time for the
   #next measurement loop iteration
endloop
```

6.2 Measurement loop output

At the start of each measurement loop, the following line is sent from the device:

```
MXXXX
```

Where:

M	= The header for a measurement loop start package.
XXXX	= The technique ID of the measurement loop, see “Table 5: Measurement technique ID's”

ID	Name
0000	Linear Sweep Voltammetry
0001	Differential Pulse Voltammetry
0002	Square Wave Voltammetry
0003	Normal Pulse Voltammetry
0005	Cyclic Voltammetry
0007	Chrono Amperometry
0008	Pulsed Amperometric Detection
000B	Open-Circuit Chrono Potentiometry
000D	Electrochemical Impedance Spectroscopy

Table 5: Measurement technique ID's

When a measurement loop is completed the following line is sent:

```
*
```

The following example shows the output of a EIS measurement loop command:

```
M000D
… data packages …
*
```

7 Variable Types

Variable Types offer context to MethodSCRIPT variables. They communicate the unit and the origin of the variable. They are also used as an argument to some functions to measure a specific type of variable. For example, when the “meas” command is used, the type of variable to measure must be passed as an argument. Table 6: Variable Types shows the available variable types.

Measurable types	ID	Description
VT_UNKNOWN	aa	Unknown (not initialized)
VT_POTENTIAL	ab	Measured WE voltage vs RE
VT_POTENTIAL_CE	ac	Measured CE voltage vs GND
VT_POTENTIAL_RE	ae	Measured RE voltage vs GND
VT_POTENTIAL_WE_VS_CE	ag	Measured WE voltage vs CE
VT_POTENTIAL_AIN0	as	Measured Analog Input 0 voltage
VT_POTENTIAL_AIN1	at	Measured Analog Input 1 voltage
VT_POTENTIAL_AIN2	au	Measured Analog Input 2 voltage
VT_CURRENT	ba	Measured WE current
VT_PHASE	cp	Measured phase
VT_IMP	ci	Measured impedance
VT_ZREAL	cc	Measured real part of complex impedance
VT_ZIMAG	cd	Measured imaginary part of complex impedance
Appliable types	ID	Description
VT_CELL_SET_POTENTIAL	da	Set control value for cell potential
VT_CELL_SET_CURRENT	db	Set control value for cell current
VT_CELL_SET_FREQUENCY	dc	Set value for frequency
VT_CELL_SET_AMPLITUDE	dd	Set value for ac amplitude
Other	ID	Description
VT_TIME	eb	Time in seconds, referenced to the time since startup
VT_PIN_MSK	ec	Binary pin mask, indicating which pins are high / low
Generic types (reserved for user)	ID	Description
VT_CURRENT_GENERIC1	ha	
VT_CURRENT_GENERIC2	hb	
VT_CURRENT_GENERIC3	hc	
VT_CURRENT_GENERIC4	hd	
VT_POTENTIAL_GENERIC1	ia	
VT_POTENTIAL_GENERIC2	ib	
VT_POTENTIAL_GENERIC3	ic	
VT_POTENTIAL_GENERIC4	id	
VT_MISC_GENERIC1	ja	
VT_MISC_GENERIC2	jb	

VT_MISC_GENERIC3	jc	
VT_MISC_GENERIC4	jd	

Table 6: Variable Types

8 Script argument types

8.1 var

The argument “var” is a reference to a MethodSCRIPT variable. Variables can be changed during runtime.

8.2 literal

A literal is a constant value argument, it cannot change during runtime.

8.3 var_type

See section “Variable Types”

8.4 integer (int8, int16, int32, uint8, uint16, uint32)

These are integer constants, these cannot be changed and do not accept SI prefixes.

8.5 comparator

Comparator operator for Boolean logic, these include:

- The equals operator “==”
- The not equals operator “!=”
- The greater than operator “>”
- The greater than or equal to operator “>=”
- The smaller than operator “<”
- The smaller than or equal to operator “<=”
- The bitwise AND operator “&”
  (true if at least one bit of both sides matches and is ‘1’)
- The bitwise OR operator “|”
  (true if there is at least one bit of the left or right is set to ‘1’)
- The bitwise Exclusive OR operator “^”
  (true if at least one bit of the right and left operator differ in value)

8.6 string

A string constant argument, a string is always encapsulated in double quotes (").

8.7 Optional arguments

Some commands can have optional arguments to extend their functionality. For example most techniques support the use of a second working electrode (bipot or poly_we). See chapter “Optional arguments” for detailed information.

9 Optional arguments

Optional arguments are added after the last mandatory argument. The format is "cmd_name(arg1 arg2 arg3 ..)"

9.1 poly_we

Measure a current on a secondary WE. This secondary WE uses the CE and RE of the main WE, but can be offset in potential from the main WE or RE. WE’s that are used as poly WE must be configured as such using the command “set_pgstat_mode 5” for the channel the WE belongs to.

Arguments

Name	Type	
Channel	uint8	Channel of the additional working electrode
Output current	var[out]	Output variable to store the measured current in.

Example

```
e
#declare variable for output potential
var p
#declare variable for output current of main WE
var c
#declare variable for output current of secondary WE
var b
#enable bipot on ch 1
set_pgstat_chan 1
#set the selected channel to bipot mode
set_pgstat_mode 5
#set bp mode to offset or constant
set_poly_we_mode 1
#set offset or constant voltage
set_e 100m
#set the current-range of the secondary WE
set_cr 1u
#switch back to do actual measurement on ch 0
set_pgstat_chan 0
#set the main WE channel to low speed mode
set_pgstat_mode 2
set_cr 1u
set_pot_range 0m 0m
set_max_bandwidth 500
set_e -500m
cell_on
wait 1
#LSV measurement using channel 0 as WE1 and channel 1 as WE2
#WE2 current is stored in var b
meas_loop_lsv p c -500m 500m 5m 100m poly_we(1 b)
   pck_start
   pck_add p
   pck_add c
   pck_add b
   pck_end
endloop
cell_off
```

Perform an LSV measurement and send a data packet for every iteration. The data packet contains the set potential (p), the measured current of the main WE (c) and the measured current of the secondary WE (b). The LSV performs a potential scan from -500 mV to 500 mV with steps of 10 mV at a rate of 100 mV/s. This results in a total of 101 data points at a rate of 10 points per second.

9.2 nscans

Perform multiple potential sweeps (scans) during a Cyclic Voltammetry measurement, instead of sweeping only once. When nscans is used the cycle number will be printed at the start of every sweep. The number is formatted as “Cxxxx” where “xxxx” is a number starting from 0000. A special character (“-”) is printed at the end of every cycle. For the rest the output is the same as when nscans omitted. See output example below.

Arguments

Name	Type	
Number of scans	uint16	The number of scans to perform.

Example

```
meas_loop_cv p c 0 -500m 500m 10m 1 nscans(5)
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

This CV performs a potential scan from 0 V to -500 mV to 500 mV and back to 0V with steps of 10 mV at a rate of 1 V/s. Because of the “nscans(5)” parameter, this pattern is repeated 5 times.

Output example with nscans(2):

```
M0005
C0000
Pda7F8604Fu;ba475D0A8p,10,207
Pda7F9E6A6u;ba51FC060p,10,207
Pda7FB6CFCu;ba5C994C0p,10,207
-
C0001
Pda7F9E6A6u;ba51FC060p,10,207
Pda7FB6CFCu;ba5C994C0p,10,207
Pda7FCF353u;ba6731714p,10,207
-
*
```

9.3 meta_msk

Enable or disable metadata packages sent with the “pck_add” command. This can be used to reduce the amount of data sent by disabling packages, making it possible to achieve higher data rates.

Arguments

Name	Type	
Metadata mask	uint32	A bitwise mask used to enable/disable types of metadata packages. Values can be added to enable multiple types of metadata. 0 = All metadata disabled 1 = Enable datapoint status package 2 = Enable current range package

Example

```
e
var a
set_pgstat_mode 2
meas 100m a ba
pck_start meta_msk(0x03)
pck_add a
pck_end
pck_start meta_msk(0x01)
pck_add a
pck_end
```

This example measures a current and then sends two packages containing the measured current. The first package will include the current range and status metadata. The second package will only include the status metadata.

10 Tags

A script can have optional tags (or labels) to direct the execution flow in case of an event like aborting a running script.

10.1 Supported tags

Name	Description
on_finished:	The commands after this this tag will be executed when the script is aborted, or normal script execution reaches the tag. These commands are not executed if a script error has occured, as no further commands are executed in this case.

10.2 on_finished:

Example

```
meas_loop_eis h r j 10m 200k 100 17 0
   pck_start
   pck_add h
   pck_add r
   pck_add j
   pck_end
endloop
on_finished:
cell_off
```

The cell will be switched off when the EIS loop is finished or the script is aborted during the EIS loop. If a runtime script error occurs, these commands will not be executed.

11 Script commands

11.1 var

Declare a variable. All variables must be declared before use. Currently only names that consist of 1 lower case character are allowed.

Arguments

Name	Type	
Variable name	var	Variable reference (a-z).

Example

```
var a
```

Declare variable with name "a".

11.2 store_var

Store a value in a variable. This value can be referenced in following commands.

Arguments

Name	Type	
Variable name	var[out] (int, float)	Variable reference.
Value	literal (int, float)	Literal value to store in the variable.
Variable Type	var_type	The type identifier for this value, see section "Variable Types".

Example

```
store_var i 200 ja
```

Store a value of 200 in the variable 'i' as a floating point variable. This value is of type: "VT_MISC_GENERIC1".

```
store_var i 200i ja
```

Store a value of 200 in the variable 'i' as an integer variable. This value is of type: "VT_MISC_GENERIC1".

11.3 array

Declare a variable array. All variables must be declared before use. Currently only names that consist of 1 lower case character are allowed.

Arguments

Name	Type	
Variable name	var	Array reference (a-z).
Array size	uint32	The amount of variables this array can hold.

Example

```
array a 10
```

Declare array with name "a" and size 10.

11.4 array_set

Set a variable at the specified index in the array.

Arguments

Name	Type	
Array variable	var	Array reference.
Array index	var / literal (int)	The index in the array to store the value to.
Variable	var / literal (int, float)	The variable to store in the array.

Example

```
array a 6i
array_set a 5i 20m
```

Declare array with name “a”. Then store the value “0.02” in the array at index 5.

11.5 array_get

Get a variable from the specified index in the array.

Arguments

Name	Type	
Array variable	var	Array reference.
Array index	var / literal (int)	The index in the array to get the value from.
Variable	var[out] (int, float)	The output variable to store the data from the array in.

Example

```
array_get a 5i b
```

Get the value in the array at index 5 and stores it in variable “b”.

11.6 copy_var

Copies value from the source address to the destination address.

Arguments

Name	Type	
Source variable	var (int, float)	Variable reference to copy from.
Destination variable	var[out] (int, float)	Variable reference to copy to.

Example

```
copy_var i j
```

Copies the variable ‘i‘ to ‘j‘.

11.7 add_var

Add “lhs” to “rhs” and store the result in “lhs”. Metadata of lhs is maintained. Accepts either integer or floating point variables, but both arguments must match.

Arguments

Name	Type	
Lhs	var (int, float)	The lhs variable, the result is stored here.
Rhs	var / literal (int, float)	Literal or variable to add to lhs var.

Example

```
add_var i 1
```

Adds 1 to variable ‘i’ and stores it to ‘i’.

11.8 sub_var

Subtract “rhs” from “lhs” and store the result in “lhs”. Metadata of lhs is maintained. Accepts either integer or floating point variables, but both arguments must match.

Arguments

Name	Type	
Lhs	var (int, float)	The lhs variable, the result is stored here.
Rhs	var / literal (int, float)	Literal or variable to subtract from lhs var.

Example

```
sub_var i 1
```

Subtracts 1 from the variable ‘i’ and stores it to ‘i’

11.9 mul_var

Multiply “lhs” with “rhs” and store the result in “lhs”. Metadata of lhs is maintained. Accepts either integer or floating point variables, but both arguments must match.

Arguments

Name	Type	
Lhs	var (int, float)	The lhs variable, the result is stored here.
Rhs	var / literal (int, float)	Literal or variable to multiply lhs by.

Example

```
mul_var i 1500m
```

Multiplies the variable ‘i’ with 1.5 and stores it to ‘i’

11.10 div_var

Divide “lhs” by “rhs” and store the result in “lhs”. Metadata of lhs is maintained. Accepts either integer or floating point variables, but both arguments must match.

Arguments

Name	Type	
Lhs	var (int, float)	The lhs variable, the result is stored here.
Rhs	var / literal (int, float)	Literal or variable to divide lhs by.

Example

```
div_var i 1500m
```

Divides the variable ‘i’ by 1.5 and stores it to ‘i’

11.11 set_e

Apply a variable or literal as the cell potential. This determines the potential (WE vs RE). The potential is limited by the potential range of the currently active “pgstat mode” see section “PGStat mode properties”.

Arguments

Name	Type	
Potential	var / literal (float)	The cell potential to apply in volts.

Example

```
set_e 100m
```

Sets control value for the potentiostat loop to 0.1V.

11.12 wait

Wait for the specified amount of time.

Arguments

Name	Type	
Time	var / literal (float)	The amount of time to wait in seconds.

Example

```
wait 100m
```

Wait 100 milliseconds.

11.13 set_int

Configure the interval for the “await_int” command.

Arguments

Name	Type	
Interval	var / literal (float)	The interval time in seconds.

Example

```
set_int 100m
```

Set interval to 100 milliseconds.

11.14 await_int

Wait for the next interval. This command allows the use of an asynchronous background timer to synchronize the script to a certain interval.

Arguments

No arguments

Example

```
var t
store_var t 0 aa
set_int 100m
#loop until t wait time is higher than 50 ms
loop t <= 50m
   #wait for next interval of 100ms
   await_int
   #add 10 ms to wait time
   add_var t 10m
   #wait variable amount of time
   wait t
endloop
```

Set interval to 100 ms. Then execute a loop every 100 ms using await_int to synchronize the start of each loop. Even though the loop takes a variable amount of time because of the variable “wait” command, the loop will execute once every 100 ms.

11.15 loop

Repeat all commands up to the next “endloop” until the specified condition is matched. All loops must be terminated with an “endloop”. Accepts either integer or floating point variables, but if argument types don’t match, they are compared as floats.

Arguments

Name	Type	
Stop condition lhs	var / literal (int, float)	Literal or variable to be compared with the rhs variable.
Stop condition comparator	comparator	Comparator indicating the type of comparison to make.
Stop condition rhs	var / literal (int, float)	Literal or variable to be compared with the lhs variable.

Example

```
var i
store_var i 0i aa
loop i < 10i
   add_var i 1i
endloop
```

Add 1 to i until variable “i” reaches 10. This example uses integer variables.

11.16 endloop

Signals the end of a loop, see “loop” command.

Arguments

No arguments.

11.17 breakloop

Breaks out of the current loop. The script will continue execution from the next “endloop”.

Arguments

No arguments.

11.18 if, elseif, else, endif

Conditional statements allow the conditional execution of commands. Every "if" statement must be terminated by an "endif" statement. In between the "if" and "endif" statements can be any number of "elseif" statements and/or one "else" statement. Accepts either integer or floating point variables, but if argument types don't match, they are compared as floats.

Arguments for if, elseif commands

Name	Type	
Condition lhs	var / literal (int, float)	Literal or variable to be compared with the rhs variable.
Condition comparator	comparator	Comparator indicating the type of comparison to make.
Condition rhs	var / literal (int, float)	Literal or variable to be compared with the lhs variable.

Example

```
if a > 5
  send_string "a is bigger than 5"
elseif a >= 3
  send_string "a is lower than 5 but bigger than or equal to 3"
else
  send_string "a is lower than 3"
endif
```

One of the send_string commands will be executed, depending on the value of variable 'a'.

11.19 meas

Measure a datapoint of the specified type and store the result as a variable. The datapoint will be averaged for the specified amount of time at the maximum available sampling rate.

For supported value types of each device, refer to section "Supported variable types for meas command".

Arguments

Name	Type	
Time to measure	var / literal (float)	The amount of time to spend averaging measured data.
Destination	var[out] (float)	Variable to store the measured data in.
Var type	var_type	The type of variable to measure, see section "Variable Types".

Example

```
meas 100m c ba
```

Measure the signal with the var_type: ba (VT_CURRENT) for 100ms and store the result in the variable 'c'.

11.20 meas_loop_lsv

Perform a Linear Sweep Voltammetry (LSV) measurement and store the resulting current in a variable. An LSV measurement scans a potential range in small steps and measures the current at each step.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section "Measurement loop" for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store the measured current in.
Begin potential	var / literal (float)	The begin potential for the LSV technique.
End potential	var / literal (float)	The end potential for the LSV technique.
Step potential	var / literal (float)	The potential increase for each step. Affects the amount of data points per second, together with the scan rate. This is an absolute step. The direction of the scan is determined by "Begin potential" and "End potential".
Scan rate	var / literal (float)	The scan rate of the LSV technique. This is the speed at which the applied potential is ramped in V/s. Can only be positive.
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we

Example

```
meas_loop_lsv p c -500m 500m 10m 100m
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Perform an LSV measurement and send a data packet for every iteration. The data packet contains the set potential and measured current. The LSV performs a potential scan from -500 mV to 500 mV with steps of 10 mV at a rate of 100 mV/s. This results in a total of 101 data points at a rate of 10 points per second.

11.21 meas_loop_cv

Perform a Cyclic Voltammetry (CV) measurement. In a CV measurement, the potential is stepped from the begin potential to the vertex 1 potential, then the direction is reversed and the potential is stepped to the vertex 2 potential and finally the direction is reversed again and the potential is stepped back to the begin potential. The current is measured at each step.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section “Measurement loop” for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store the measured current in.
Begin potential	var / literal (float)	The begin potential for the CV technique.
Vertex 1 potential	var / literal (float)	The vertex 1 potential. First potential where direction reverses.
Vertex 2 potential	var / literal (float)	The vertex 2 potential. Second potential where direction reverses.
Step potential	var / literal (float)	The potential increase for each step. Affects the amount of data points per second, together with the scan rate. This is an absolute step that does not affect the direction of the scan.
Scan rate	var / literal (float)	The scan rate of the CV technique. This is the speed at which the applied potential is ramped in V/s. Can only be positive.
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we
nscans

Example

```
meas_loop_cv p c 0 500m -500m 10m 100m
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Perform a CV measurement and send a data packet for every iteration. The data packet contains the set potential and measured current. The CV performs a potential scan from 0 mV to 500 mV to -500 mV to 0 mV. The steps of 10 mV at a rate of 100 mV/s. This results in a total of 201 data points at a rate of 10 points per second.

11.22 meas_loop_dpv

Perform a Differential Pulse Voltammetry (DPV) measurement. In a DPV measurement, the potential is stepped from the begin potential to the end potential. At each step, the current (reverse current) is measured, then a potential pulse is applied and the current (forward current) is measured. The forward current minus the reverse current is stored in the "Output current" variable.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section "Measurement loop" for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store "forward current – reverse current" in.
Begin potential	var / literal (float)	The begin potential for the potential scan.
End potential	var / literal (float)	The end potential for the potential scan.
Step potential	var / literal (float)	The potential increase for each step. Affects the amount of data points per second, together with the scan rate. This is an absolute step that does not affect the direction of the scan.
Pulse potential	var / literal (float)	The potential of the pulse. This is added to the currently applied potential during a step.
Pulse time	var / literal (float)	The time the pulse should be applied.
Scan rate	var / literal (float)	The speed at which the applied potential is ramped in V/s. Can only be positive. Scan rate must be lower than "Step potential / Pulse time / 2".
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we

Example

```
meas_loop_dpv p c -500m 500m 10m 20m 5m 100m
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Perform a DPV measurement and send a data packet for every iteration. The data packet contains the set potential and "forward current – reverse current". The DPV performs a potential scan from -500 mV to 500 mV with steps of 10 mV at a rate of 100 mV/s. This results in a total of 101 data points at a rate of 10 points per second. At every step a pulse of 20mV is applied for 5ms

11.23 meas_loop_swv

Perform a Square Wave Voltammetry (SWV) measurement. In a SWV measurement, the potential is stepped from the begin potential to the end potential. At each step, the current (reverse current) is measured, then a potential pulse is applied and the current (forward current) is measured. The forward current minus the reverse current is stored in the “Output current” variable. The pulse length is “1 / Frequency / 2”.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section “Measurement loop” for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store “forward current – reverse current” in.
Output forward current	var[out] (float)	Output variable to store forward current in.
Output reverse current	var[out] (float)	Output variable to store reverse current in.
Begin potential	var / literal (float)	The begin potential for the potential scan.
End potential	var / literal (float)	The end potential for the potential scan.
Step potential	var / literal (float)	The potential increase for each step. This is an absolute step that does not affect the direction of the scan.
Amplitude potential	var / literal (float)	The amplitude of the pulse. This value times 2 is added to the currently applied potential during a step.
Frequency	var / literal (float)	The frequency of the pulses.
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we

Example

```
meas_loop_swv p c f r -500m 500m 10m 15m 10
    pck_start
    pck_add p
    pck_add c
    pck_end
endloop
```

Perform a SWV measurement and send a data packet for every iteration. The data packet contains the set potential and “forward current – reverse current”. The SWV performs a potential scan from -500 mV to 500 mV with steps of 10 mV at a frequency of 10 Hz. This results in a total of 101 data points at a rate of 10 points per second. At every step a pulse of 30mV (2*15mV) is applied for 50ms (1/Frequency/2).

11.24 meas_loop_npv

Perform a Normal Pulse Voltammetry (NPV) measurement. In an NPV measurement, the pulse potential is stepped from the begin potential to the end potential. At each step the pulse potential is applied and the current is measured at the top of this pulse. The potential is then set back to the begin potential until the next step. The measured current is stored in the “Output current” variable.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section “Measurement loop” for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store the measured current in.
Begin potential	var / literal (float)	The begin potential for the potential scan.
End potential	var / literal (float)	The end potential for the potential scan.
Step potential	var / literal (float)	The pulse potential increase for each step. Affects the amount of data points per second, together with the scan rate. This is an absolute step that does not affect the direction of the scan.
Pulse time	var / literal (float)	The time the pulse should be applied.
Scan rate	var / literal (float)	The speed at which the applied potential is ramped in V/s. Can only be positive. Scan rate must be lower than “Step potential / Pulse time / 2”.
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we

Example

```
meas_loop_npv p c -500m 500m 10m 20m 5m 100m
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Perform an NPV measurement and send a data packet for every iteration. The data packet contains the set potential and measured pulse current. The NPV performs a potential scan from -500 mV to 500 mV with steps of 10 mV at a rate of 100 mV/s. This results in a total of 101 data points at a rate of 10 points per second. At every step a potential pulse of “step index * step potential” mV is applied for 5ms.

11.25 meas_loop_ca

Perform a Chrono Amperometry (CA) measurement. In a CA measurement, a DC potential is applied the current is measured at the specified interval. The measured current is stored in the "Output current" variable.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section "Measurement loop" for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store the measured current in.
DC potential	var / literal (float)	The DC potential to be applied.
Interval time	var / literal (float)	The interval between measured data points.
Run time	var / literal (float)	The total run time of the measurement.
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we

Example

```
meas_loop_ca p c 100m 100m 2
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Perform a CA measurement and send a data packet for every iteration. The data packet contains the set potential and measured current. A DC potential of 100 mV is applied. The current is measured every 100 ms for a total of 2 seconds. This results in a total of 20 data points at a rate of 10 points per second.

11.26 meas_loop_pad

Perform a Pulsed Amperometric Detection (PAD) measurement. In a PAD measurement, potential pulses are applied to a DC potential. Each iteration starts at the DC potential, the current is measured before the pulse (idc). Then the pulse potential is applied, and the current is measured at the end of the pulse (ipulse). The output current returns a current value depending of one the 3 modes: dc (idc), pulse (ipulse) or differential (ipulse – idc).

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section "Measurement loop" for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the set potential for this iteration.
Output current	var[out] (float)	Output variable to store "forward current – reverse current" in.
DC potential	var / literal (float)	The begin potential for the potential scan.
Pulse potential	var / literal (float)	The potential of the pulse. This is the potential that is set during a pulse. It is not referenced to the DC potential.
Pulse time	var / literal (float)	The time the pulse should be applied.
Interval time	var / literal (float)	The time of the pulse interval
Run time	var / literal (float)	Total run time of the measurement
mode	uint8	PAD mode : 1= dc , 2 = pulse , 3 = differential
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Optional arguments

poly_we

Example

```
meas_loop_pad p c 500m 1500m 10m 50m 10050m 2
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Perform a PAD measurement and send a data packet for every iteration. The data packet contains the set potential and measured current. A DC potential of 500 mV is applied. A pulse potential of 1500mV is applied every 50 ms for 10 ms and the current is measured on the pulse (mode = pulse). The measurement is 10.05 seconds in total. This results in a total of 201 data points at a rate of 20 points per second.

11.27 meas_loop_ocp

Perform an Open Circuit Potentiometry (OCP) measurement. In an OCP measurement, the CE is disconnected so that no potential is applied. The open circuit RE potential is measured at the specified interval. The measured potential is stored in the “Output potential” variable.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section “Measurement loop” for more information.

Arguments

Name	Type	
Output potential	var[out] (float)	Output variable to store the measured RE potential in.
Interval time	var / literal (float)	The interval between measured data points.
Run time	var / literal (float)	The total run time of the measurement.
<opt. argument>	Optional arg.	See chapter 9 for detailed information

Example

```
meas_loop_ocp p 100m 2
   pck_start
   pck_add p
   pck_end
endloop
```

Perform an OCP measurement and send a data packet for every iteration. The data packet contains the set measured RE potential. The RE potential is measured every 100 ms for a total of 2 seconds. This results in a total of 20 data points at a rate of 10 points per second.

11.28 meas_loop_eis

Perform an EIS frequency scan and store the resulting Z-real and Z-imaginary in the given variables. EIS does not currently support autoranging. High speed PGStat mode is required for EIS. The following commands currently have no effect on EIS measurements:

- set_max_bandwidth: bandwidth is taken from frequency scan ranges.
- set_pot_range: pot range is taken from amplitude and DC potential arguments.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to section "Measurement loop" for more information.

Arguments

Name	Type	
Output frequency	var[out] (float)	Output variable to store the set frequency for this iteration.
Output Z-real	var[out] (float)	Output variable to store the measured phase in.
Output Z-imaginary	var[out] (float)	Output variable to store the measured impedance in.
Amplitude	var / literal (float)	Amplitude of the applied sinewave.
Start frequency	var / literal (float)	Start frequency of the scan.
End frequency	var / literal (float)	End frequency of the scan.
Nr of points	var / literal (int, float)	Number of frequency points to be scanned.
DC potential	var / literal (float)	DC potential offset of the applied sinewave.

Example

```
meas_loop_eis f r i 10m 100k 100 11i 0
  pck_start
  pck_add f
  pck_add r
  pck_add i
  pck_end
endloop
```

Perform an EIS measurement a frequency f with 10mV amplitude and stores the Z-real result in r and the Z-imaginary result in j. 11 points will be measured at frequencies between 100 kHz and 100 Hz, divided on a logarithmic scale.

11.29 set_autoranging

Enable or disable autoranging for all meas_loop_* functions. Autoranging selects the most appropriate current range for the current measured in the last measurement loop iteration. The selected current range is limited by the min and max current arguments. If min expected current and max expected current are the same value, autoranging is disabled.

Arguments

Name	Type	
Min current	literal (float)	The min current in this measurement.
Max current	literal (float)	The max current in this measurement.

Example

```
set_autoranging 1u 1m
```

Enable autoranging for currents between 1 uA and 1 mA.

11.30 pck_start

Signal the start of a measurement data packet.

Arguments

No arguments.

Example

```
pck_start
```

Signal the start of a new measurement package.

11.31 pck_add

Add a stored variable to be sent in this data packet.

Arguments

Name	Type	
Variable	var (int, float)	The variable to add to the data packet.

Example

```
pck_add i
```

Add variable ‘i’ to the data packet.

11.32 pck_end

Signal the end of a measurement data package.

Arguments

No arguments.

Optional arguments

meta_msk

Example

```
pck_end
```

Signal the end of a measurement data package.

11.33 set_max_bandwidth

Set maximum bandwidth of the signal being measured. Any signal of significant higher frequency than the set bandwidth will be filtered out. There is no defined lower bound to the bandwidth. At max bandwidth the signal is attenuated by up to 1% of the potential or current step.

Arguments

Name	Type	
Max bandwidth	var / literal (float)	The maximum expected bandwidth expected. Anything below this frequency will not be filtered out.

Example

```
set_max_bandwidth 1k
```

Set the max bandwidth to a frequency of 1 kHz.

11.34 set_cr

Set the current range for the given maximum current. The device will select the lowest current range that can measure this current without overloading.

Arguments

Name	Type	
Max current	var / literal (float)	The maximum expected current.

Example

```
set_cr 500n
```

Set current range to be able to measure a current of 500nA

Note: This command is ignored when autoranging is enable for meas_loop_eis.

11.35 cell_on

Turn the cell on, any settings set when the cell was off will be applied here.

Arguments

No arguments.

Example

```
cell_on
```

Turn the cell on. The potentiostat will start applying the configured potential.

11.36 cell_off

Turn the cell off.

Arguments

No arguments.

Example

```
cell_off
```

Turn the cell off. This stops the potentiostat from applying a potential to the cell.

11.37 set_pgstat_mode

Set the pgstat hardware configuration to be used for measurements. Setting the pgstat mode initializes all channel settings to the default values for that mode. See section “PGStat Modes" for more information.

Arguments

Name	Type	
PGStat mode	uint8	Set pgstat mode: 0 = off 2 = low speed 3 = high power 4 = max range 5 = poly_we

Example

```
set_pgstat_mode 3
```

Set hardware configuration to high power mode.

11.38 send_string

Send an arbitrary string as output of the MethodSCRIPT. This string is prepended by a ‘T’, this is the “text” package identifier. Avoid sending a ‘\n’ character or non-ASCII characters.

Arguments

Name	Type	
String	string	An arbitrary string. Surrounded by quotes (")

Example

```
send_string "hello world"
```

Sends string “Thello world\n” as output of the MethodSCRIPT.

11.39 set_gpio_cfg

Set GPIO pins configuration. Pins can be configured as one of multiple supported modes. To use a pin in a specific mode, it must be configured for that mode. See section “Device GPIO pin configurations ” for available pin configurations per device.

Arguments

Name	Type	
Pin mask	uint32	Bitmask that represents pins that will be configured with this command.
Mode	uint8	0 = GPIO Input 1 = GPIO Output 2 = Peripheral 1 3 = Peripheral 2

Example

```
set_gpio_cfg 0b11 1
```

Set pins 0 and 1 to GPIO output mode. The “0b” means that the following value is expressed in a binary format.

11.40 set_gpio_pullup

Enable or disable GPIO pin pullups.

Arguments

Name	Type	
Pin mask	uint32	Bitmask that represents pins that will be configured with this command.
Pullup	uint8	0 = Pullup disabled 1 = Pullup enabled

Example

```
set_gpio_pullup 0b11 1
```

Enables pullup on pins 0 and 1. The “0b” means that the following value is expressed in a binary format.

11.41 set_gpio

Set GPIO pins. Pins with multiple roles that are not configured as GPIO output pins are ignored.

Arguments

Name	Type	
Pin mask	var / literal (int)	Bitmask that represents the state of the bits. Bit 0 is for GPIO0, bit 1 for GPIO1, etc. Bits that are high correspond with a high output signal.

Example

```
set_gpio 0b11i
```

Sets pin 0 and 1 high, the rest of the GPIO output pins is set low. The added ‘i’ is needed because “set_gpio” only accepts integer variables.

11.42 get_gpio

Get GPIO pin values. Pins with multiple roles that are not configured as GPIO input pins are ignored.

Arguments

Name	Type	
Pin mask	var[out] (int)	Bitmask that represents the state of the bits. Bit 0 is for GPIO0, bit 1 for GPIO1, etc. Bits that are high correspond with a high input signal.

Example

```
get_gpio g
```

Read all GPIO pins configured as input and stores the bit mask representation of the high pins in variable g.

11.43 set_pot_range

Set the expected potential range for this script. Some devices cannot apply their full potential range in one measurement, but need to be set up to reach these potentials beforehand. This function lets you communicate to the device what the voltage range is you expect in your measurement. The device will automatically configure itself to be able to reach these potentials. This function will return an error if the expected voltage range is greater than the dynamic potential range of the device, or if the expected voltage range exceeds the maximum potential limits of the device.

This is a device specific command. The following devices require this command to reach their full potential range:

- EmStat Pico

For these devices the voltage range that can be applied without changing the expected potential range is defined in section “PGStat Modes” as the “dynamic potential range”.

Arguments

Name	Type	
Potential 1	var / literal (float)	Bound 1 of the expected voltage range for this measurement.
Potential 2	var / literal (float)	Bound 2 of the expected voltage range for this measurement.

Example

```
set_pot_range 0 1200m
```

Ensure that the next measurement can apply potentials between 0 V and 1.2 V.

11.44 set_pgstat_chan

Select a potentiostat channel. If the device has multiple parallel potentiostat channels, they can be selected with this command. In the future it will be possible to use these two channels parallel to each other, but this feature is not yet available. Refer to section "Other device specific properties" to see how many channels each device has.

Arguments

Name	Type	
Channel index	uint8	The pgstat channel index to select.

Example

```
set_pgstat_chan 0
```

Selects pgstat channel 0.

11.45 set_poly_we_mode

Selects the mode of the additional working electrode.

Arguments

Name	Type	
Poly_we_mode	uint8	The mode of the additional working electrode: 0 = fixed mode (Additional WE is relative to RE) 1= offset mode (Additional WE is relative to main WE)

Example

```
set_poly_we_mode 1
```

The additional working electrode mode is set to offset mode.

11.46 get_time

Retrieves current time in seconds from the internal device clock. Resolution is dependent on the returned *time* value (see table below for estimated resolution).

Arguments

Name	Type	
Time	var[out] (float)	The output variable to store the time in.

Example

```
get_time t
```

Stores current time in variable 't'.

System time	Resolution
<1 hour	≤1ms
1 to 24 hours	≤10ms
1 to 10 days	≤100ms
10 to 100 days	≤1s
≥100 days	>1s

11.47 file_open

Opens file on persistent storage. This file can be used to store script output to. To store script output to this file, use the “set_script_output” command.

Arguments

Name	Type	
Path	string	The path to the file to open. May include folders.
Open mode	uint8	0 = Create new file, if a file with the same name exists, it is overwritten. 1 = Create new file, if a file with the same name exists, new data is appended to it. 2 = Create new file, if a file with the same name exists, the file is not opened and an error is returned.

Example

```
file_open "measurement.txt" 0
```

Creates a new file, overwriting any existing file with the same name.

11.48 file_close

Closes currently opened file on persistent storage. If no file is opened, the command is skipped.

Arguments

No arguments

Example

```
file_close
```

Closes the currently opened file.

11.49 set_script_output

Sets the output mode for the script. This affects where the measurement packages and other script output are sent to.

Arguments

Name	Type	
Output mode	uint8	0 = Disable the output of the script completely. 1 = Output to the normal output channel (Default) 2 = Output to file storage 3 = Output to both normal channel and file storage

Example

```
set_script_output 3
```

Script output is directed to file storage and normal output.

11.50 hibernate

Puts the device in hibernate mode (deep sleep).

Arguments

Name	Type	
Wakeup source mask	uint8	Bitmask for wakeup sources 0x01 = UART 0x02 = Wakeup pin 0x04 = Wakeup timer
Wakeup time	var / literal (float)	Time in seconds after which the system is woken up by the system timer. Time resolution is 125ms, STATUS_SCRIPT_BAD_ARG (4002) will be thrown when time < 125 millisecond.

Example

```
hibernate 0x07i 60
```

Hibernate until the system is woken by the wake-up pin, UART or after 60 seconds.

NOTE:
The hibernate command will disable the internal ADT7420 temperature sensor on the EmStatPico when GPIO8 and GPIO9 are configured for I2C to save more power. Power consumption with the temperature sensor enabled is about 250µA higher that it would be with the sensor disabled. It is up to the user to configure these pins for I2C prior to entering hibernate or disable the temperature sensor manually. See 11.39 set_gpio_cfg for more information on configuring GPIO.

NOTE:
All channels settings are cleared, and channels are switched off in hibernate mode

11.51 i2c_config

Setup I2C configuration. This is required before using any other I2C command from MethodSCRIPT. The I2C interface supported by MethodSCRIPT always works as master. Multi master mode is currently not supported.

Arguments

Name	Type	
Clock speed	var / literal (int)	I2C clock speed. 100k (standard mode) and 400k (fast mode) are officially supported.
Address mode	literal (int)	I2C addressing mode (7 or 10 bit)

Example

```
i2c_config 100k 7
```

Configure I2C for standard mode with 7 bit address.

NOTE:
Make sure the I2C GPIO pins are configured for I2C. *See 11.39 set_gpio_cfg for more information on configuring GPIO.*

11.52 i2c_write_byte

Transmits one byte over I2C. Also generates I2C start and stop conditions. If a NAck (Not Acknowledge) was received from the slave device the user should handle this and reset the *Ack status* variable.

Arguments

Name	Type	
Device address	var / literal (int)	Address of the slave device.
Transmit data	var / literal (int)	Data byte to transmit.
Ack status	var[out]	Result of the I2C operation. 0 = Ack received 1 = NAck received for address 2 = NAck received for data NOTE: the variable passed for this argument should be initialized to 0. Otherwise it will assume that the previous operation caused a NAck that was not handled by the script and will throw the error: "*STATUS_SCRIPT_I2C_UNHANDLED_NACK*".

Example

```
var a
store_var a 0 ja
i2c_write_byte 0x48i 0x03i a
```

Write the value 3 to the device with address 0x48.

11.53 i2c_read_byte

Receive one byte over I2C. Also generates I2C start and stop condition. If a NAck (Not Acknowledge) was received from the slave device the user should handle this and reset the *Ack status* variable.

Arguments

Name	Type	
Device address	var / literal (int)	Address of the slave device.
Receive data	var (int)	Variable to store received byte in.
Ack status	var[out] (int)	Result of the I2C operation. 0 = Ack received 1 = NAck received for address 2 = NAck received for data NOTE: the variable passed for this argument should be initialized to 0. Otherwise it will assume that the previous operation caused a NAck that was not handled by the script and will throw the error: "*STATUS_SCRIPT_I2C_UNHANDLED_NACK*".

Example

```
var a
var d
store_var a 0 ja
i2c_read_byte 0x48i d a
```

Receive one byte of data from device 0x48 and store it in variable "d".

11.54 i2c_write

Write the contents of an array over I2C. Also generates I2C start and stop conditions. If a NAck (Not Acknowledge) was received from the slave device the user should handle this and reset the *Ack status* variable.

Arguments

Name	Type	
Device address	var / literal (int)	Address of the slave device.
Transmit data	array (int)	Reference to array that contains the data to transmit.
Transmit count	var / literal (int)	Number of bytes to transmit. Minimum value = 1, maximum value is 255 or size of the array.
Ack status	var[out] (int)	Result of the I2C operation. 0 = Ack received 1 = NAck received for address 2 = NAck received for data NOTE: the variable passed for this argument should be initialized to 0. Otherwise it will assume that the previous operation caused a NAck that was not handled by the script and will throw the error: "*STATUS_SCRIPT_I2C_UNHANDLED_NACK*".

Example

```
var a
array w 2
array_set w 0i 12i
array_set w 1i 34i
store_var a 0 ja
i2c_write 0x48i w 2 a
```

Transmit the values 12 (0x0C) and 34 (0x22) to the device with address 0x48.

11.55 i2c_read

Read a specified number of bytes from I2C. Also generates I2C start and stop conditions. If a NAck (Not Acknowledge) was received from the slave device the user should handle this and reset the *Ack status* variable.

Arguments

Name	Type	
Device address	var / literal (int)	Address of the slave device.
Received data	array (int)	Reference to array to store received data in.
Receive count	var / literal (int)	Number of bytes to receive. Minimum value = 1, maximum value is 255 or size of the array.
Ack status	var[out] (int)	Result of the I2C operation. 0 = Ack received 1 = NAck received for address 2 = NAck received for data NOTE: the variable passed for this argument should be initialized to 0. Otherwise it will assume that the previous operation caused a NAck that was not handled by the script and will throw the error: "*STATUS_SCRIPT_I2C_UNHANDLED_NACK*".

Example

```
var a
array r 4
store_var a 0 ja
i2c_read 0x48i r 4 a
```

Receive 4 bytes from device 0x48 and store them in array "r".

11.56 i2c_write_read

Transmit the contents of an array over I2C directly followed by reading multiple bytes to a second array. Also generates I2C start and stop conditions. If a NAck (Not Acknowledge) was received from the slave device the user should handle this and reset the *Ack status* variable. In contrast with i2c_read and i2c_write this command does not generate a STOP-condition between writing and reading.

Arguments

Name	Type	
Device address	var / literal (int)	Address of the slave device.
Transmit data	array (int)	Reference to array that contains the data to transmit.
Transmit count	var / literal (int)	Number of bytes to transmit. Minimum value = 1, maximum value is 255 or size of the array.
Received data	array (int)	Reference to array to store received data in.
Receive count	var / literal (int)	Number of bytes to receive. Minimum value = 1, maximum value is 255 or size of the array.
Ack status	var[out] (int)	Result of the I2C operation. 0 = Ack received 1 = NAck received for address 2 = NAck received for data NOTE: the variable passed for this argument should be initialized to 0. Otherwise it will assume that the previous operation caused a NAck that was not handled by the script and will throw the error: “*STATUS_SCRIPT_I2C_UNHANDLED_NACK*”.

Example

```
var a
array w 2
array r 4
array_set w 0i 12i
array_set w 1i 34i
store_var a 0 ja
i2c_write_read 0x48i w 2 r 4 a
```

Write 2 bytes to device 0x48 followed by reading 4 bytes.

11.57 abort

Aborts current code. If the “on_finished:” tag is used it will continue from there. Otherwise the script is terminated without error.

Arguments

This method has no arguments.

Example

```
var a
var d
store_var a 0 ja
i2c_read_byte 0x48i d a
if a != 0
  send_string "NAck received"
  abort
endif
# Do something interesting with the data in ‘d’
```

11.58 timer_start

For precise timing between two moments a timer can be set. This this timer can be (re)started with the timer_start command after which timer_get will return a time relative to this start moment.

Arguments

This method has no arguments.

Example

```
var a
timer_start
# Do something interesting here
```

11.59 timer_get

Read the time relative to the last call to “timer_start”. This method can be called multiple times without changing the starting moment.

Arguments

Name	Type	
Relative time	var[out] (float)	The time relative to the last “timer_start” command

Example

```
var a
timer_start
# Do something interesting that takes a bit of time here
timer_get a
pck_start
# Add a as a timestamp
pck_add a
# Add other package data
pck_end
```

NOTE:
Due to floating point number limitations the resolution is dependent on the returned time value. For a time resolution of <1ms the relative time should not exceed 1 hour.

12 PGStat Modes

PGStat modes are device wide configurations that affect which hardware is used during measurements. This is necessary for devices that have a choice between multiple measurement hardware with different properties. PGStat modes are device specific, more information can be found in “PGStat mode properties”.

12.1 PGStat mode off

All hardware is turned off to save power, no measurements can be done.

12.2 PGStat mode low speed

The hardware configuration that has the best properties for low speed measurements is picked. Usually this means it is less sensitive to high frequency noise and consumes less power. However the maximum bandwidth is limited.

12.3 PGStat mode high speed

The hardware configuration that has the best properties for high speed measurements is used. In general, this will consume more power and be more sensitive to noise. However, it will allow higher frequencies measurements to be done.

12.4 PGStat mode max range

This mode uses a hardware configuration having the highest possible potential range by combining the high and low speed mode. In general, this will consume more power and be more sensitive to noise. The bandwidth is limited to the bandwidth of the low speed mode.

12.5 PGStat mode poly_we

This mode combines the two channels forming a poly_we (bipot) device. In this mode one channel is setup as the main potentiostat and the other as an additional working electrode (bipot).

13 Script examples

Note: The command terminators (\n) are not shown in the examples. These examples can be used on any device that supports MethodSCRIPT, but they contain some commands that are device specific for the EmStat Pico. These commands will be ignored on devices that do not use them.

13.1 EIS example

The following example script runs an EIS scan from 200 kHz down to 200 Hz over 11 points. After each point a data packet will be sent containing the: frequency, Z-real, Z-imaginary variables. The amplitude of the sine is set to 10m and no DC potential is applied.

```
e
var h
var r
var j
#Select channel 0
set_pgstat_chan 0
#High speed mode is required for EIS
set_pgstat_mode 3
#Autorange starting at 1mA down to 10uA
set_autoranging 10u 1m
#Cell must be on to do measurements
cell_on
#Run actual EIS measurement
meas_loop_eis h r j 10m 200k 200 11 0
   #Send measurement package containing frequency, Z-real and Z-imaginary
   pck_start
   pck_add h
   pck_add r
   pck_add j
   pck_end
endloop
#Turn cell off when finished or aborted
on_finished:
cell_off
```

Example output:

```
e                                                         ← ack of ‘e’ cmd
M000D                                                     ← start of measurement loop
Pdc8030D40 ;ccAAE483Fm,14,288;cd7FD3127 ,14,288           ← data package
…                                                         ← more data packages
Pdc8030D3Fm;cc80EDA04 ,14,287;cd9751491m,14,287           ← data package
*                                                         ← end of measurement loop
                                                          ← newline indicating end of script
```

13.2 LSV example

The following example script runs an LSV from -0.5 V to 1.5 V with steps of 10 mV in 201 steps. The scan rate is set to 100 mV/s. After each step, a data packet will be sent containing the set cell potential and the measured WE current. The measured WE current will be used to autorange.

```
e
var c
var p
#Select channel 0
set_pgstat_chan 0
#Low speed mode is fast enough
set_pgstat_mode 2
#Select bandwidth of 40 for 10 points per second
set_max_bandwidth 40
#Set up potential window between -0.5 V and 1.5 V, otherwise
#the max potential would be 1.1 V for low speed mode
set_pot_range -500m 1500m
#Set current range to 1 mA
set_cr 1m
#Enable autoranging, between current of 100 uA and 5 mA
set_autoranging 100u 5m
#Turn cell on for measurements
cell_on
#equilibrate at -0.5 V for 5 seconds, using a CA measurement
meas_loop_ca p c -500m 500m 5
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
#Start LSV measurement from -0.5 V to 1.5 V, with steps of 10 mV
#and a scan rate of 100 mV/s
meas_loop_lsv p c -500m 1500m 10m 100m
   #Send package containing set potential and measured WE current.
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
#Turn off cell when done or aborted
on_finished:
cell_off
```

Example output:

```
e                                       ← ack of ‘e’ cmd
M0007                                   ← start of measurement loop (CA)
Pda7F85E36u;ba7F77484p,14,20B           ← data package
…                                       ← more data packages
Pda7F85E36u;ba7F77484p,14,20B           ← data package
*                                       ← end of measurement loop (CA)
M0000                                   ← start of measurement loop (LSV)
Pda816E55Fu;ba816DB89p,14,207           ← data package
…                                       ← more data packages
Pda816E55Fu;ba816DB89p,14,207           ← data package
*                                       ← end of measurement loop (LSV)
                                        ← newline indicating end of script
```

13.3 SWV example

The following example script runs a SWV from -0.5V to 0.5V with steps of 10 mV in 101 steps. After each step, a data packet will be sent containing the cell potential for that step and current resulting from the SWV measurement.

```
e
var c
var p
var f
var g
set_pgstat_chan 0
set_pgstat_mode 2
#Set maximum required bandwidth based on frequency * 4,
#however since SWV measures 2 datapoints, we have to multiply the
#bandwidth by 2 as well
set_max_bandwidth 80
#Set potential window.
#The max expected potential for SWV is EEnd + EAmp * 2 - EStep.
#This measurement would also work without this command since it
#stays within the default potential window of -1.1 V to 1.1V
set_pot_range -500m 690m
#Set current range for a maximum expected current of 2 uA
set_cr 2u
#Disable autoranging
set_autoranging 2u 2u
#Turn cell on for measurement
cell_on
#Perform SWV
meas_loop_swv p c f g -500m 500m 10m 100m 10
   #Send package with set potential,
   #"forward current - reverse current",
   #"forward current"
   #"reverse current"
   pck_start
   pck_add p
   pck_add c
   pck_add f
   pck_add g
   pck_end
endloop
#Turn off cell when done or aborted
on_finished:
cell_off
```

Example output:

```
e                                                          ← ack of 'e' cmd
M0002                                                      ← start of measurement loop
Pda7F85E36u;ba8030DDCp,10,202;ba7FB6915p,10,202;ba7F85B39p,10,202  ← data package
…                                                          ← more data packages
Pda807A1CAu;ba8030EB6p,10,202;ba80AB012p,10,202;ba807A15Cp,10,202 ← data package
*                                                          ← end of measurement loop
                                                           ← newline indicating end of script
```

13.4 I2C example – temperature sensor

The example script below reads the 16bit temperature value from the ADT7420 sensor using I2C. This is the internal temperature sensor on the Pico. Note that the senor has an I2C address 0x48.

```
e
# Most significant bits
var m
# Least significant bits
var l
# Acknowledge
var a
# Status / buffer register
var s
# Array with Write data
array w 2
# Array with Read data
array r 2
store_var a 0 ja
# Configure I2C GPIOs and set it to 100k clock, 7 bit address
set_gpio_cfg 0x0300i 2
i2c_config 100k 7
# Configure the sensor for 16bit mode with continuous conversion
array_set w 0i 0x03i
array_set w 1i 0x80i
i2c_write 0x48i w 2 a
# Read back value
i2c_write_read 0x048i w 1 r 1 a
array_get r 0i s
if s != 0x80i
   send_string "ERROR: register did not change."
   abort
endif
# Wait for temperature measurement to become ready
# This takes about 250ms and can be read from bit 7 in register 0x02
wait 250m
store_var s 0x80i ja
array_set w 0i 0x02i
loop s & 0x80i
   i2c_write_byte 0x48i 0x02i a
   i2c_read_byte 0x48i s a
endloop
# Read temperature values
i2c_write_byte 0x48i 0x00i a
i2c_read 0x48i r 2 a
array_get r 0i m
array_get r 1i l
# Send values to user
pck_start
   pck_add m
   pck_add l
pck_end
```

Example output:

```
e                          ← ack of ‘e’ cmd
L                          ← Start of loop
+                          ← End of loop
Paa800000Ai;aa80000E9i     ← Temperature data package
```

13.5 I2C example – Real time clock

The below example script demonstrates the use of I2C in combination with the S-35390 RTC that can be found on the EmStat Pico development board. It sets the time and date to the arbitrary value of 2:14Am 29-8-97. Then It will wait 10 seconds and reads back the time.

```
e
var a
var d
store_var a 0i ja
var i
store_var i 0i ja
array r 7i
array w 7i
# Year = '97
array_set w 0i 0xE9i
# Month = August
array_set w 1i 0x10i
# Day = 29
array_set w 2i 0x94i
# Day of week = friday
array_set w 3i 0xA0i
# Hour = 2 AM
array_set w 4i 0x40i
# Minute = 14
array_set w 5i 0x88i
# Seconds = 0
array_set w 6i 0x00i
# Configure I2C GPIOs and set it to 100k clock, 7 bit address
set_gpio_cfg 0x0300i 2
i2c_config 100k 7
# Write data to real-time data registers
i2c_write 0x32i w 7i a
# Printing the time as it was written.
i2c_read 0x32i r 7i a
pck_start
store_var i 0i ja
loop i < 7i
   array_get r i d
   pck_add d
   add_var i 1i
endloop
pck_end
# Wait ~10 seconds
send_string "Waiting for the time to change."
wait 9500m
# Read data from real-time data registers
i2c_read 0x32i r 7i a
pck_start
store_var i 0i ja
loop i < 7i
   array_get r i d
   pck_add d
   add_var i 1i
endloop
pck_end
```

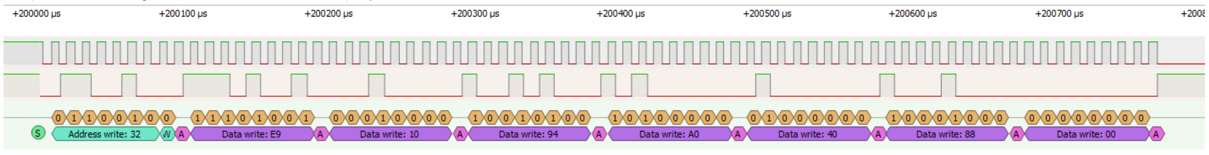
PalmSens

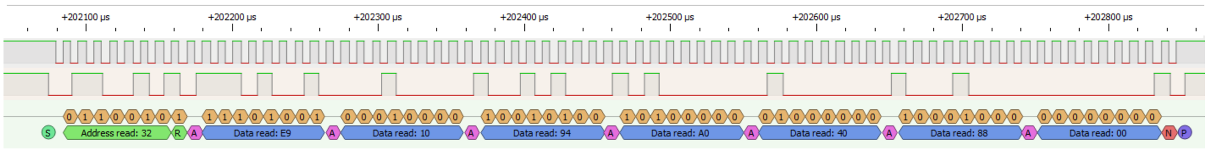
Example output:

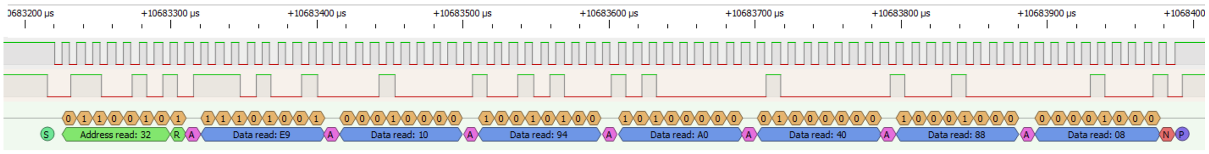
```
e
PL
aa80000E9i;aa8000010i;aa8000094i;aa80000A0i;aa8000040i;aa8000088i;aa8000000i+

TWaiting for the time to change.
PL
aa80000E9i;aa8000010i;aa8000094i;aa80000A0i;aa8000040i;aa8000088i;aa8000008i+
```

The raw communication over I2C is displayed below. The top line contains the SCL, the line below that is SDA. The bottom lines of each row represent the interpreted data.







14 Error handling

Errors can occur that prevent the execution of the MethodSCRIPT. These errors can occur either during the parsing of the script or during the execution of the script (runtime). If the error occurs during parsing, the line nr and character nr where the error occurred will be reported. During runtime, only the line nr will be reported. A command that returns an error will not return an extra newline '\n' after the newline of the error message.

Parsing error format:

```
!XXXX: Line L, Col C\n
```

Runtime error format:

```
!XXXX: Line L\n
```

Where:
XXXX = The error code, see "Table 7: Error codes"
L = Line nr, starting at 1
C = Line character nr, starting at 1

The reported line number for runtime errors does not count comment lines. For parsing errors, the comment lines do count.

Code (Hex)	Name	Description
0001	STATUS_ERR	An unspecified error has occurred
0002	STATUS_INVALID_VT	An invalid Value Type has been used
0003	STATUS_UNKNOWN_CMD	The command was not recognized
0004	STATUS_REG_UNKNOWN	Not applicable for MethodSCRIPT
0005	STATUS_REG_READ_ONLY	Not applicable for MethodSCRIPT
0006	STATUS_WRONG_COMM_MODE	Not applicable for MethodSCRIPT
0007	STATUS_BAD_ARG	An argument has an unexpected value
0008	STATUS_CMD_BUFF_OVERFLOW	Command exceeds maximum length
0009	STATUS_CMD_TIMEOUT	The command has timed out
000A	STATUS_REF_ARG_OUT_OF_RANGE	A var has a wrong identifier
000B	STATUS_OUT_OF_VAR_MEM	Cannot reserve the memory needed for this var
000C	STATUS_NO_SCRIPT_LOADED	Cannot run a script without loading one first
000D	STATUS_INVALID_TIME	The given (or calculated) time value is invalid for this command
000E	STATUS_OVERFLOW	An overflow has occurred while averaging a measured value
000F	STATUS_INVALID_POTENTIAL	The given potential is not valid
0010	STATUS_INVALID_BITVAL	A variable has become either "NaN" or "inf"
0011	STATUS_INVALID_FREQUENCY	The input frequency is invalid
0012	STATUS_INVALID_AMPLITUDE	The input amplitude is invalid
0013	STATUS_NVM_ADDR_OUT_OF_RANGE	Not applicable for MethodSCRIPT

0014	STATUS_OCP_CELL_ON_NOT_ALLOWED	Cannot perform OCP measurement when cell on
0015	STATUS_INVALID_CRC	Not applicable for MethodSCRIPT
0016	STATUS_FLASH_ERROR	An error has occurred while reading / writing flash
0017	STATUS_INVALID_FLASH_ADDR	An error has occurred while reading / writing flash
0018	STATUS_SETTINGS_CORRUPT	The device settings have been corrupted
0019	STATUS_AUTH_ERR	Not applicable for MethodSCRIPT
001A	STATUS_CALIBRATION_INVALID	Not applicable for MethodSCRIPT
001B	STATUS_NOT_SUPPORTED	This command or part of this command is not supported by the current device
001C	STATUS_NEGATIVE_ESTEP	Step Potential cannot be negative for this technique
001D	STATUS_NEGATIVE_EPULSE	Pulse Potential cannot be negative for this technique
001E	STATUS_NEGATIVE_EAMP	Amplitude cannot be negative for this technique
001F	STATUS_TECH_NOT_LICENCED	Product is not licenced for this technique
0020	STATUS_MULTIPLE_HS	Cannot have more than one high speed and/or max range mode enabled (EmStat Pico)
0021	STATUS_UNKNOWN_PGS_MODE	The specified PGStat mode is not supported
0022	STATUS_CHANNEL_NOT_POLY_WE	Channel set to be used as Poly WE is not configured as Poly WE
0023	STATUS_INVALID_FOR_PGSTAT_MODE	Command is invalid for the selected PGStat mode
0024	STATUS_TOO_MANY_EXTRA_VARS	The maximum number of vars to measure has been exceeded
0025	STATUS_UNKNOWN_PAD_MODE	The specified PAD mode is unknown
0026	STATUS_FILE_ERR	An error has occurred during a file operation
0027	STATUS_FILE_EXISTS	Cannot open file, a file with this name already exists
0028	STATUS_ZERO_DIV	Variable divided by zero
0029	STATUS_UNKNOWN_GPIO_CFG	GPIO pin mode is not known by the device
002A	STATUS_WRONG_GPIO_CFG	GPIO configuration is incompatible with the selected operation
4000	STATUS_SCRIPT_SYNTAX_ERR	The script contains a syntax error
4001	STATUS_SCRIPT_UNKNOWN_CMD	The script command is unknown
4002	STATUS_SCRIPT_BAD_ARG	An argument was invalid for this command
4003	STATUS_SCRIPT_ARG_OUT_OF_RANGE	An argument was out of range
4004	STATUS_SCRIPT_UNEXPECTED_CHAR	An unexpected character was encountered
4005	STATUS_SCRIPT_OUT_OF_CMD_MEM	The script is too large for the internal script memory

4006	STATUS_SCRIPT_UNKNOWN_VAR_TYPE	The variable type specified is unknown
4007	STATUS_SCRIPT_VAR_UNDEFINED	The variable has not been declared
4008	STATUS_SCRIPT_INVALID_OPT_ARG	This optional argument is not valid for this command
4009	STATUS_SCRIPT_INVALID_VERSION	The stored script is generated for an older firmware version and cannot be run
400A	STATUS_SCRIPT_INVALID_DATATYPE	The parameter datatype (float/int) is not valid for this command
400B	STATUS_SCRIPT_NESTED_MEAS_LOOP	Measurement loops cannot be placed inside other measuments loops
400C	STATUS_SCRIPT_UNEXPECTED_CMD	Command not supported in current situation
400D	STATUS_SCRIPT_MAX_SCOPE_DEPTH	Scope depth too large
400E	STATUS_SCRIPT_INVALID_SCOPE	The command had an invalid effect o scope depth (for example "if" directly followed by an "endif" statement)
400F	STATUS_SCRIPT_INDEX_OUT_OF_RANGE	Array index out of bounds
4010	STATUS_SCRIPT_I2C_NOT_CONFIGURED	I2C interface was not initialized with i2c_config command
4011	STATUS_SCRIPT_I2C_UNHANDLED_NACK	NAck flag not handled by script
4012	STATUS_SCRIPT_I2C_ERR	Something unexpected went wrong. Could be a bug in the firmware
4013	STATUS_SCRIPT_I2C_INVALID_CLOCK	I2C clock frequency not supported by hardware
4014	STATUS_SCRIPT_HEX_OR_BIN_FLT	Non integer SI vars cannot be parsed from hex or binary representation
4015	STATUS_INVALID_WAKEUP_SOURCE	The selected (combination of) wake-up source is invalid
4016	STATUS_WAKEUP_TIME_INVALID	RTC was selected as wake-up source with invalid time argument
7FFF	STATUS_FATAL_ERROR	A fatal error has occurred, the device must be reset

Table 7: Error codes

15 Device specific information

15.1 PGStat mode properties

EmStat Pico

Low speed mode	Value min	Value max
Bandwidth	0.016 Hz	100 Hz
Potential range	-1.25 V	2.0 V
Dynamic potential window	2.2 V	2.2 V
High speed mode	Value min	Value max
Bandwidth	0.016 Hz	200 kHz
Potential range	-1.7 V	2.0 V
Dynamic potential window	1.214 V	1.214 V
Max range mode	Value min	Value max
Bandwidth	0.016 Hz	100 Hz
Potential range	-1.7 V	2.0 V
Dynamic potential window	2.6 V	2.6 V

Table 8: EmStat Pico PGStat mode properties (see EmStat Pico datasheet for more information)

15.2 Current ranges

EmStat Pico

Low speed mode current ranges	Current follower resistor	Current range index
100 nA	10 MOhm	0x0
1.95 uA	512 kOhm	0x1
3.91 uA	256 kOhm	0x2
7.81 uA	128 kOhm	0x3
15.63 uA	64 kOhm	0x4
31.25 uA	32 kOhm	0x5
62.5 uA	16 kOhm	0x6
125 uA	8 kOhm	0x7
250 uA	4 kOhm	0x8
500 uA	2 kOhm	0x9
1 mA	1 kOhm	0xA
5 mA	200 Ohm	0xB
High speed mode current ranges	Current follower resistor	Current range index
100 nA	10 MOhm	0x80
1 uA	1 MOhm	0x81
6.25 uA	160 kOhm	0x82
12.5 uA	80 kOhm	0x83
25 uA	40 kOhm	0x84
50 uA	20 kOhm	0x85
100 uA	10 kOhm	0x86
200 uA	5 kOhm	0x87
1 mA	1 kOhm	0x88
5 mA	200 Ohm	0x89
Max range mode current ranges	Current follower resistor	Current range index
100 nA	10 MOhm	0x80
1 uA	1 MOhm	0x81
6.25 uA	160 kOhm	0x82
12.5 uA	80 kOhm	0x83
25 uA	40 kOhm	0x84
50 uA	20 kOhm	0x85
100 uA	10 kOhm	0x86
200 uA	5 kOhm	0x87
1 mA	1 kOhm	0x88
5 mA	200 Ohm	0x89

Table 9: EmStat Pico current ranges

15.3 Supported variable types for meas command

EmStat Pico

Variable types
VT_POTENTIAL
VT_POTENTIAL_CE
VT_POTENTIAL_RE
VT_POTENTIAL_WE_VS_CE
VT_POTENTIAL_AIN0
VT_POTENTIAL_AIN1
VT_POTENTIAL_AIN2
VT_CURRENT

Table 10: EmStat Pico measurable variable types

15.4 Device GPIO pin configurations

EmStat Pico

Bitmask	Pin name	Mode 0	Mode 1	Mode 3
0x0001	GPIO0_PWM	GPIO Input	GPIO Output	PWM (Not implemented)
0x0002	GPIO1_SPI_MISO	GPIO Input	GPIO Output	SD card
0x0004	GPIO2_SPI_CLK	GPIO Input	GPIO Output	SD card
0x0008	GPIO3_SPI_MOSI	GPIO Input	GPIO Output	SD card
0x0010	GPIO4_SPI_CS0	GPIO Input	GPIO Output	SD card
0x0020	GPIO5	GPIO Input	GPIO Output	
0x0040	GPIO6	GPIO Input	GPIO Output	
0x0080	GPIO7_WAKE	GPIO Input	GPIO Output	Wake from sleep (Active low)
0x0100	I2C_SCL	GPIO Input	GPIO Output	I2C
0x0200	I2C_SDA	GPIO Input	GPIO Output	I2C

Table 11: EmStat Pico GPIO pin configurations

15.5 Other device specific properties

Property	EmStat Pico
Number of pgstat channels	2
File storage	SD card (SPI)

Table 12: Other device specific properties

16 Version changes

Version 1.1 Rev 1

- Added support for EmStat Pico firmware v1.1
- Added “Tags” chapter
- Added Max range pgstat mode for the EmStat Pico
- Added BiPot / Poly WE support
- Added PAD technique
- The ‘e’ command now replies with an extra ‘\n’ to separate the script response from the ‘e’ command response
- Added ability to use whitespace in script (tabs and spaces)
- Added error code documentation

Version 1.1 Rev 2

- Corrected EIS auto ranging information
- Added information about loop command output

Version 1.1 Rev 3

- Corrected OCP parameters, does not have set potential
- Corrected set_pgstat_chan command example
- Corrected SWV example comment about bandwidth
- Correct loop example “add” command should be “add_var”
- Corrected inconsistent names for low power / low speed mode

Version 1.1 Rev 4

- Corrected endloop command was sometimes called end_loop

Version 1.2 Rev 1

- Added conditional statements (if, else, elseif, endif)
- Added abort command
- Added breakloop command
- Added external storage (SD Card) commands
- Added new variable types
- Added supported variable types table
- Added bitwise operators
- Added new GPIO commands (get_gpio, set_gpio_cfg, set_gpio_pullup)
- Added support for integer variables
- Updated error codes
- Added get_time command
- Added timer_start and timer_get commands
- Added set_int, await_int commands
- Added ability to input hexadecimal or binary values
- Added support for arrays
- Added support for specifying what metadata to send in measurement packages
- Added nscans optional parameter for Cyclic Voltammetry
- Added hibernate command
- Added I2C interface
- Added I2C example