PalmSens
Compact Electrochemical Interfaces

MethodSCRIPT ™

MethodSCRIPT v1.4

Version v1.4, 2023-02-01

Table of Contents

PalmSens

Chapter 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 instrument and generating scripts for supported devices.

Although the base of MethodSCRIPT is device-agnostic, there are differences between instruments that prevent identical scripts from running on multiple devices. These differences are indicated in their appropriate chapter. For documentation regarding detailed device capabilities please visit palmsens.com.

1.1. Terminology

PGStat	Potentiostat / Galvanostat
EmStat	PGStat device series by PalmSens
Cell	The electrochemical system to be analysed
CE	Counter Electrode
RE	Reference Electrode
WE	Working Electrode
SE	Sense Electrode
Technique	A standard electrochemical measurement technique
Iteration	A single execution of a loop
SI	International System of Units
Var	(MethodSCRIPT) variable (usually command input)
Var [out]	Variable that will be used for command output
Var [in/out]	Variable which value is both used as command input and output
HEX	Hexadecimal (= base 16) number (e.g. 0xA1)

Chapter 2. Features

2.1. Implemented 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[1]:
  - Chronoamperometry (CA)
  - Linear Sweep Voltammetry (LSV)
  - Cyclic Voltammetry (CV)
  - Differential Pulse Voltammetry (DPV)
  - Square Wave Voltammetry (SWV)
  - Normal Pulse Voltammetry (NPV)
  - Pulsed Amperometric Detection (PAD)
  - Electrochemical Impedance Spectroscopy (EIS)
  - Galvanostatic Electrochemical Impedance Spectroscopy (GEIS)
  - Open Circuit Potentiometry (OCP)
  - Chronopotentiometry (CP)
  - Linear Sweep Potentiometry (LSP)
  - AC Voltammetry (ACV)
  - Fast Cyclic Voltammetry (FCV)
- Storing of measurement data to onboard flash storage or SD card (if available on hardware).
- 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.
- Support for user code during a measurement step.
- Up to 26 variables or arrays can be stored and referenced to from within the script. This allows for fast burst measurements that are not slowed down by communication.
- A comprehensive set of MethodSCRIPT commands:
  - Basic math operations (addition, subtraction, multiplication, division).
  - Bitwise operations (and, or, xor, logical shift left/right, inversion).
  - Conditional statements (if, elseif, else, endif).
  - Support for loops.
  - Synchronization commands (wait amount of time, wait until interval).
- Exact timing control.
- Script syntax will be verified when loading. Runtime errors are checked during execution.

- Autorun script at start-up from persistent memory.
- Low-power[2] mode (hibernate).
- Direct control over GPIO and the I²C interface for communication with external sensors and actuators.

2.2. Planned future features

- The following techniques are planned:
  - Fast Chronoamperometry (FCA)
  - Stripping Chronopotentiometry (SCP)

2.3. Supported devices

- EmStat4
- EmStat Pico

[1] Not all techniques are supported by every instrument.
[2] The hibernate command is supported on all instruments, but only low-power on EmStat Pico.

Chapter 3. Script format

A script consists of a series of MethodSCRIPT commands. Each command starts with the command name and is followed by zero or more arguments. Arguments are separated by one or more spaces (or tabs). Tabs and spaces at the start and end of the line are ignored. Each command is terminated by a newline character (`'\n'`, ASCII code 10). Lines are limited to a maximum of 128 characters (including leading and trailing tabs and spaces and the newline character). Empty lines (including lines only containing spaces and tabs) are not allowed in MethodSCRIPT.

Comments can be added to a line by inserting a `#` character followed by the comment. A line containing only a comment is allowed.

Since MethodSCRIPT v1.4, comments may take up a tiny amount of storage and execution time to preserve line numbering.

The following small MethodSCRIPT example demonstrates the syntax.

```
# This is a comment
wait 100m  # Comments can also follow other text
if 1 < 2 then
  send_string "Hello world"
endif
```

3.1. Relation between MethodSCRIPT and communication protocol

MethodSCRIPTs are sent to the device using the *communication protocol*, which is described in detail in a separate document. Since there is a tight relationship between the two protocols, a brief summary and example are given below.

To send a script to the device:

- Send `e` (for *execute*) or `l` (for *load*), followed by a newline character (`\n`).
- Send the MethodSCRIPT, line by line, each line followed by a newline character (`\n`).
- Send an empty line (`\n`) to denote the end of the script.

The `e` and `l` command, as well as the empty line, are not part of the MethodSCRIPT language but are part of the device communication protocol.

The following example shows how the above MethodSCRIPT can be transmitted and executed using the device communication protocol. In this example, the newline characters are rendered as `\n`.

```
e\n
# This is a comment\n
wait 100m  # Comments can also follow other text\n
if 1 < 2 then\n
  send_string "Hello world"\n
endif\n
\n
```

The response of above script will be:

```
e\n
Thello world\n
\n
```

This response can be broken down into three parts:

1. The `"e"` followed by `\n` acknowledges that the *execute* command has been started.
2. The `"T"` followed by `"hello world"` is the output of the `send_string` command.
3. The empty line denotes the (successful) end of the script execution.

In the remainder of this document, only the MethodSCRIPT commands will be shown, without the `e` or `l` command, and without the empty line at the end. For readability, the `\n` will be omitted as well, except when needed for clarification.

In some example scripts provided on the web or in other documents, the `e` is included as the first line of the script. This allows for simple copy-pasting to a terminal application in order to directly execute the script. It should be clear from context when the `e` command should be added (if absent) or removed (if present).

Chapter 4. MethodSCRIPT variables

4.1. MethodSCRIPT variables

MethodSCRIPT variables represent numerical values that can be used within the script. They can be stored internally either in floating-point format or as signed integer. Some commands only accept integer variables, others will only accept floating-point variables (*floats*). In Chapter 14, *Script command description*, the arguments of each command are documented. See the "Arguments" table in each command section.

Floating-point variables are represented as a signed integer value followed by an SI prefix. 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 (= 100×10^{-3}).

Table 1. SI prefix conversion table

SI prefix	Text	Factor
`a`	atto	10^{-18}
`f`	femto	10^{-15}
`p`	pico	10^{-12}
`n`	nano	10^{-9}
`n`	nano	10^{-9}
`u`	micro	10^{-6}
`m`	milli	10^{-3}
(space)	*(none)*	10^{0}
`k`	kilo	10^{3}
`M`	mega	10^{-9}
`G`	giga	10^{6}
`T`	tera	10^{9}
`P`	peta	10^{12}
`E`	exa	10^{15}

Integer variables end with an `i` instead of an SI prefix. If no prefix is provided, the number is assumed to be a floating-point number. Integer variables can also be entered in hexadecimal or binary representation by prefixing the value with `0x` or `0b` respectively. In this case, the `i` at the end of the number is optional. Hexadecimal and binary representations are not allowed for floating-point variables.

Operations involving floating-point numbers often introduce (tiny) rounding errors. Consequently, testing for equality of floating-point numbers (e.g. testing if `x == 3`) might give unexpected results. This makes floating-point numbers less suitable when an exact integer value is expected, such as with counters in loops.

Integer variables are internally represented as 32-bit signed integers. They are not subject to rounding. However, integers have a limited range (roughly -2×10^{9} to $+2 \times 10^{9}$) and are

truncated when dividing. For example, when an integer number 10 is divided by 4, the result is 2 instead of 2.5.

Variables are not explicitly linked to a unit; instead the unit is implied by the associated *Variable Type*. Refer to section Chapter 7, *Variable types* for more information.

Some number input parameters are not MethodSCRIPT variables. These include *uint8*, *uint16* and *uint32*. For such integer parameters, it is allowed but not necessary to append an `i`. They do not accept SI prefixes.

The representation of MethodSCRIPT variables is different for scripts and script output. The format of the output is described in Chapter 5, *Interpreting measurement data packages*.

4.2. Script command variables

Variables that are part of the MethodSCRIPT are represented as a signed integer followed by a prefix for floating-point values, or `i` for integer values.

Integer variables

```
255i
0xFF
0b11111111
```

Above example shows the integer value of the decimal number `255` using decimal, hexadecimal and binary representation. In the example, the `i` is omitted in places where it is optional.

Float variables

```
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.

4.3. 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` is the hexadecimal value and `p` is the *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 be implemented by the user using, for example, a *lookup table* or a *switch case* to translate the prefix character to its corresponding factor. Example implementations for several programming languages and platforms can be found on our MethodSCRIPT Examples repository on GitHub.

Chapter 5. Interpreting measurement data packages

5.1. Package format

Measurement packages consist of a header, followed by up to 33 *variable* packages (each with their own *variable type*), followed by a terminating `\n` character. Consecutive packages are separated using a semicolon. The package format is shown in Table 2, “Measurement data package format.”. Section 5.2, “Variable sub package format” explains the format of the variable fields.

Table 2. Measurement data package format.

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

5.2. Variable sub package format

The format for a variable sub package is:

Table 3. Variable sub package format.

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

Where:

tt	Variable Type, represented as a base26 identifier that ranges from `aa` to `jv`. Variable Types are always lower case. See Chapter 7, *Variable types* for more information.
HHHHHHHp	MethodSCRIPT package variable. See Section 4.3, “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.

Table 4. 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 on EmStat Pico, <4% of max ADC value on EmStat4) 8 = overload warning (>80% of max ADC value) The *overload* and *timing not met* status flags mean that data is unreliable. When *overload warning* or *underload* is set, the data is probably fine, but ranging should be considered.
2	Range	2	Index of current range for current measurements (device-specific, see Section B.3, "Current ranges"), or any other range for other measurements (e.g. potential range for potential measurements). The range is just intended for diagnostic purposes, and is not used in any calculations during parsing. NOTE: Since originally only current ranges were implemented, this field is often referred to as *current range*. However, it does not always apply to currents anymore.
4	Noise	1	Noise level, intended for diagnostic purposes.

5.3. Package parsing example

A MethodSCRIPT device sends the following measurement data package:

```
Pda8000800u;ba8000800u,10,20B\n
```

This package contains two variables: `da8000800u` and `ba8000800u,10,20B`.

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

- The Variable Type is `da`, which corresponds to `VT_CELL_SET_POTENTIAL`.
- The value is `08000800` – `0x8000000` = `0x800` or 2048. The prefix is `u` which stands for *micro*. This makes the final value 2048 µV (= 2.048 mV).
- This variable has no metadata.

The variable sub package `ba8000800u,10,20B` can be broken down as follows:

- The Variable Type is `ba`, which corresponds to `VT_CURRENT`.
- The value is `08000800` – `0x8000000` = `0x800` or 2048. The prefix is `u`, which stands for *micro*. This makes the final value 2048 uA (= 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 `0B`. This indicates that it is a current range with the current range `0x0B` (= 11). For example, on the EmStat Pico, this refers to the *5 mA* current range. This current range is just for diagnostic purposes, and is not used in any calculations during parsing.

Chapter 6. Measurement loop commands

6.1. Introduction

Most 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 the responsibility of the user to ensure there is enough time between measurement iterations to execute the user commands inside the loop.

If the user code takes more time than there is available, the next iteration is started too late, which likely results in less accurate measurement results. This will be reflected in the metadata (see Table 4, “Metadata types.”), by setting the "timing not met" status flag, so it can be detected by inspecting the metadata. How much time is available for user code depends on many factors and should be determined empirically. For very fast measurement iterations it is recommended to keep the code inside the loop as short as possible so it does not take too long.

Often the communication data rate determines the minimum interval time for a measurement loop. If timing errors are caused by communication, it could be a solution to store the measurement results in a MethodSCRIPT array, and transmit the data after the measurement loop.

In contrast to measurement loops, *fast measurement techniques* have dedicated commands that will return all iterations at once. For example, a Fast CV measurement is performed using the `meas_fast_cv` command.

Limitation:
It is not possible to use a fast technique or another measurement loop inside of a measurement loop. However, measurement loops can be used freely inside of a normal loop and vice versa.

6.2. Measurement loop example

Below is an example of a MethodSCRIPT containing a measurement loop. This works as follows:

- The first five commands (before the `meas_loop_ca` command) are executed only once. These commands define the two variables that will be used in the loop, configure the potentiostat, and turn on the cell.
- The `meas_loop_ca` command starts a Chronoamperometry (CA) measurement. Based on the provided arguments, this will apply a DC potential of 100 mV and perform a current measurement iteration every 200 ms.
- After the measurement iteration has been performed, the MethodSCRIPT commands inside the measurement loop are executed. In this example, a data package is transmitted here, containing the set potential and measured current.
- When the `endloop` is reached, the firmware checks if another iteration should be performed. If this is the case, the script waits until it is time and then performs the next iteration.
- When the last iteration has been completed, the script continues after the `endloop` command. In this

example the loop stops after 5 iterations since an interval of 200 ms and a total run time of 1000 ms was specified.

```
var p
var c
# Select channel 0, set PGStat mode to low-speed and turn on the cell.
set_pgstat_chan 0
set_pgstat_mode 2
cell_on
# Run a measurement loop for the Chronoamperometry (CA) technique.
meas_loop_ca p c 100m 200m 1000m
   # The following commands are executed after each iteration (measurement).
   pck_start    # Start a new data packet.
   pck_add p    # Add the p variable (potential) to the packet.
   pck_add c    # Add the c variable (current) to the packet.
   pck_end      # Close and transmit the data packet.
   # At the endloop command, the script execution halts until it is time
   # for the next measurement loop iteration.
endloop
```

6.3. Measurement loop output

The start of a measurement loop is always indicated by a line in the format `MXXXX` where `XXXX` is the technique ID of the measurement loop (see Table 5). The end of a measurement loop is indicated by a line containing only an asterisk (`*`). In general, the output of a measurement loop would like something like this:

General output format of a measurement loop.

```
MXXXX
...output of user commands inside the loop
...(usually the data packages)
*
```

When the above example script is executed, the output could look like this.

Example output of the above measurement loop.

```
M0007
PdaDF5CB18n;ba9699F74p,14,218,40
PdaDF5CB18n;ba9699F74p,14,218,40
PdaDF5CB18n;ba9699F74p,14,218,40
PdaDF5CB18n;ba9699F74p,14,218,40
PdaDF5CB18n;ba9699F74p,14,218,40
*
```

As explained in Chapter 5, *Interpreting measurement data packages*, `daDF5CB18n` denotes a variable of type `CELL_SET_POTENTIAL` (i.e. the Set control value for WE potential) with a value of 0.099994392 [V]. Due to the

resolution of the DAC, the actual value is very close, but not exactly equal, to the specified value of 100 mV. The actual used value is returned by the measurement loop commands so they can be used in futher calculations.

Table 5. Measurement technique ID.

ID	Name
`0000`	Linear Sweep Voltammetry
`0001`	Differential Pulse Voltammetry
`0002`	Square Wave Voltammetry
`0003`	Normal Pulse Voltammetry
`0005`	Cyclic Voltammetry
`0007`	Chronoamperometry
`0008`	Pulsed Amperometric Detection
`000A`	Chronopotentiometry
`000B`	Open-Circuit Chronopotentiometry
`000D`	Electrochemical Impedance Spectroscopy
`000E`	Galvanostatic Electrochemical Impedance Spectroscopy
`000F`	Linear Sweep Potentiometery
`0010`	Fast Cyclic Voltammetry

See Chapter 14, *Script command description* to see which devices support which techniques.

Chapter 7. Variable types

Variable types (*VarTypes*) offer some context to MethodSCRIPT variables. They communicate the type and/or 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.

A complete list of all defined variable types is listed in Appendix C, *Variable types*

Chapter 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. *array*

For storing more than one element, *arrays* can be used. Like variables, arrays have to be defined before they can be used. Interaction with arrays happens via their reference (just like variables).

Table 6. Total storage for array elements

Instrument	Max array elements
ESPico	4096
EmStat4	32768

8.3. literal

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

8.4. *VarType*

See Chapter 7, *Variable types*.

8.5. integer types (*uint8*, *uint16*, *uint32*)

These are integer constants, these cannot be changed and do not accept SI prefixes. Minimum and maximum values for these variables are as follows:

Table 7. Data types

Variable	Min	Max
uint8	0	255
uint16	0	65,535
uint32	0	4,294,967,295

8.6. condition expressions

Condition expressions are used in the MethodSCRIPT commands `if`, `elseif` and `loop`. A condition expression always consists of an operator with two operands, in the form `operand1 operator operand2`, for example `i < 10`. The operators and operands must be separated by at least one space or tab. Both operands can be either a MethodSCRIPT variable or an (integer or floating-point) literal. The following operators are supported:

Operator	Description
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`&`	Bitwise AND
`|`	Bitwise OR

Notes:

- The comparison operators (`==`, `!=`, `>`, `>=`, `<`, `<=`) support integer and floating-point numbers. If any of the operands is a floating-point number, the other operand is converted to floating-point if neccesary.
- The result of any comparison with `NaN` (not-a-number) is always *false*.
- The bitwise operators (`&` and `|`) only support integer numbers.
- For bitwise operators, the condition is *true* if (and only if) the result of the bitwise operation is non-zero.
- For unsupported operations (i.e. a bitwise operation on a floating-point number), the condition is always *false*.

Beware of unexpected results due to rounding errors when using floating-point numbers. For example, the expression `100000001 == 99999999i` is *true*, because the integer number `99999999i` will be converted to floating-point format. In this case, both floating-point numbers are rounded to 100000000 and consequently the comparison evaluates to *true*. However, the expression `100000001i == 99999999i` is *false*, since both operands are integers, which are not rounded.

Do not forget to add the `i` suffix for integer literals (see Section 4.2, “Script command variables”) when using bitwise operators. For example, the condition `i & 1` will always be *false*, because `1` is a floating-point number, and bitwise operations on floating-numbers are not supported. However, the condition `i & 1i` will be *true* if bit 0 of variable `i` is set.

8.7. string

A sequence of characters, i.e. a piece of text. Strings are enclosed in double quotes, e.g. `"example string"`. Strings may only consist of printable ASCII characters (ASCII code 32–126), excluding the quotation mark (`"`), since that is used as delimiter.

In MethodSCRIPT, strings are always literals (constants). There are no commands to store or modify strings.

8.8. 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 9, *Optional arguments* for detailed information.

Chapter 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	Description
Channel	*uint8*	Channel of the additional working electrode.
Output current	*var* [out]	Output variable to store the measured current in.

The following code example performs an LSV measurement and sends 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.

```
# 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_range ba 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_range ba 1u
set_range_minmax da -500m 500m
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 10m 100m poly_we(1 b)
   pck_start
   pck_add p
   pck_add c
   pck_add b
   pck_end
endloop
cell_off
```

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	Description
Number of scans	*uint16*	The number of scans to perform (≥ 1).

This example CV performs a potential scan from 0 V to -500 mV to 500 mV and back to 0 V with steps of 10 mV at a rate of 1 V/s. Because of the `nscans(2)` parameter, this pattern is repeated two times.

```
meas_loop_cv p c 0 -500m 500m 10m 1 nscans(2)
   pck_start
   pck_add p
   pck_add c
   pck_end
endloop
```

Output for example with nscans 2

```
M0005
C0000
Pda8000000 ;ba9AE0ABCf,14,212,40
...
Pda899FAA9n;ba8100E0Dp,14,212,40
-
C0001
Pda8000000 ;ba9AE0ABCf,14,212,40
...
Pda899FAA9n;ba8100E0Dp,14,212,40
-
*
```

9.3. nscans_avg

Average the measured currents of multiple scans in a Cyclic Voltammetry measurement, keeping the same array length as when having only one scan.

Arguments

Name	Type	Description
Number of scans	*uint16*	The number of scans to average (1–30000).

For example, the following `meas_fast_cv` command will perform 7 scans which are averaged together. The result is stored in arrays `p` and `i` and printed using a loop.

Example

```
meas_fast_cv p i c 0 -100m 100m 100m 10 nscans_avg(7)
store_var x 0i ja
loop x < c
   pck_start meta_msk(0x00)
   # Add set potential to packet
   array_get p x t
   pck_add t
   # Add measured current to packet
   array_get i x t
   pck_add t
   pck_end
   add_var x 1i
endloop
```

The output contains 5 points, just like a scan without averaging would. In contrast with a regular scan without `nscans_avg`, the currents are averages over 7 scans.

Output

```
L
da8000000 ;ba801B85Cp
da20A34E8n;ba20C37E0p
da8000000 ;ba801B85Cp
daDF5CB18n;ba8018739n
da8000000 ;ba801DD0Fp
+
```

9.4. nscans_equil

Perform `n` amount of scans without measuring current, before the normal measured scans.

Arguments

Name	Type	Description
Number of scans	*uint16*	The number of scans to perform during the equilibration phase.

The following example illustrates the use of `nscans_equil` performing 3 equilibration scans. Output format is the same as without this optional parameter.

Example

```
meas_fast_cv p i c 0 -100m 100m 100m 10 nscans_equil(3)
```

9.5. 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	Description
Metadata mask	*uint8*	A bitwise mask used to enable/disable types of metadata packages. 0 = All metadata disabled 1 = Enable datapoint status package 2 = Enable current range package Values can be added to enable multiple types of metadata.

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.

```
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
```

9.6. eis_tdd

The `eis_tdd` optional parameter enables the transfer of time-domain data for an EIS or GEIS measurement.

This is not supported on the EmStat Pico.

Arguments

Name	Type	Description
Potential signal tdd	*array* [out]	The acquired time domain data of the potential signal of one EIS iteration. Minimum size required is 4096.
Current signal tdd	*array* [out]	The acquired time domain data of the current signal of one EIS iteration. Minimum size required is 4096.
Number of samples	*var* [out]	The number of acquired data points (samples) for both signals.
Sampling frequency	*var* [out]	The frequency at which the data points are acquired for both signals.
Averaging mode	*uint16*	Averaging mode. Future option, default = 0.

The following example perform an EIS measurement and send the EIS result data packets followed by the time-domain data for every iteration.

```
var h
var r
var j
var i
var n
var s
var d
var g
array u 4096
array c 4096
set_pgstat_chan 0
set_pgstat_mode 3
set_max_bandwidth 200k
set_range_minmax da 0 0
set_range ba 59m
set_autoranging ba 59n 59m
cell_on
meas_loop_eis h r j 50m 200k 1 11 0 eis_tdd(u c n s 0)
   pck_start
   pck_add h
   pck_add r
   pck_add j
   pck_add s
   pck_end
   store_var i 0i ja
   loop i < n
      array_get u i d
      array_get c i g
      pck_start
      pck_add d
      pck_add g
      pck_end
      add_var i 1i
   endloop
```

```
endloop
on_finished:
cell_off
```

9.7. eis_opt

The `eis_opt` optional parameter enables the user to control the acquisition properties for an EIS or GEIS measurement.

This is not supported on the EmStat Pico.

Arguments

Name	Type	Description
Minimum acquisition time	var / literal (float)	The minimum time for acquisition (for frequencies > (Min.Cycles / frequency). Must be a positive value.
Minimum nr. of cycles to acquire	*uint8*	The minimum number of cycles to acquire (for frequencies < 1/Min.Acq.Time). Must be a positive and non-zero value.

This example performs an EIS measurement with 10 ms minimal acquisition time and minimal 1 cycle to acquire.

```
var h
var r
var j
var i
var n
var s
var d
var g
set_pgstat_chan 0
set_pgstat_mode 3
set_max_bandwidth 200k
set_range_minmax da 0 0
set_range ba 59m
set_autoranging ba 59n 59m
cell_on
meas_loop_eis h r j 50m 200k 1 11 0 eis_opt(10m 1)
   pck_start
   pck_add h
   pck_add r
   pck_add j
   pck_add s
   pck_end
   store_var i 0i ja
   loop i < n
      array_get u i d
      array_get c i g
```

```
      pck_start
      pck_add d
      pck_add g
      pck_end
      add_var i 1i
   endloop
endloop
on_finished:
cell_off
```

9.8. eis_acdc

The `eis_acdc` optional parameter returns the AC and DC information for the potential and current signal.

This is not supported on the EmStat Pico.

Arguments

Name	Type	Description
E_AC	*var* [out] (float)	AC potential (in volts).
E_DC	*var* [out] (float)	DC potential (in volts).
I_AC	*var* [out] (float)	AC current (in amperes).
I_DC	*var* [out] (float)	DC current (in amperes).

Perform an EIS measurement and send the EIS result data packets followed by the E_AC, E_DC, I_AC, I_DC values.

```
var h
var r
var j
var i
var n
var s
var d
var g
var u
var c
set_pgstat_chan 0
set_pgstat_mode 3
set_max_bandwidth 200k
set_range_minmax da 0 0
set_range ba 59m
```

```
set_autoranging ba 59n 59m
cell_on
meas_loop_eis h r j 50m 200k 1 11 0 eis_acdc(u c n s)
   pck_start
   # add frequency, Z-real, Z-imaginary to the data packet
   pck_add h
   pck_add r
   pck_add j
   # add the E_AC, E_DC, I_AC, I_DC values to the data packet
   pck_add u
   pck_add c
   pck_add n
   pck_add s
   pck_end
endloop
on_finished:
cell_off
```

Chapter 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. on_finished:

The commands after this this tag will be executed when the script is aborted, or when normal script execution reaches the tag. A script can be aborted either by the MethodSCRIPT `abort` command, or by the abort (`Z`) command from the *communication protocol*. Note that the commands after the `on_finished:` tag are not executed if a script error has occurred, as no further commands are executed in this case.

The following example demonstrates the program flow when using `abort` and `on_finished:` in a script:

```
var i
store_var i 0i ja
loop i < 10i
    send_string "before if"
    if i == 2i
        send_string "abort"
        abort
    endif
    send_string "after if"
    add_var i 1i
endloop
on_finished:
send_string "finished"
```

Output:

```
L
Tbefore if
Tafter if
Tbefore if
Tafter if
Tbefore if
Tabort
+
Tfinished
```

The following scripts illustrates the use of the `on_finished:` tag in a more realistic use case. In this example, the cell will be switched off when the EIS loop is finished or when the script is aborted during the EIS loop.

```
# first configure channel and PGstat mode (not shown in this example)
# ...
cell_on
```

```
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
```

Chapter 11. 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 and column number where the error occurred will be reported. During runtime, only the line number 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, refer to Appendix A, *Error codes* for a complete list of error codes. L = Line nr, starting at 1
C = Line character nr, starting at 1

Up to MethodSCRIPT v1.3, lines containing only comments were not counted for runtime errors. Since MethodSCRIPT v1.4, comment lines are also counted, so the line numbers do reflect the actual line number of the script, even during runtime.

Chapter 12. PGStat modes

PGStat modes (Potentiostat / Galvanostat 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 options with different properties. PGStat modes are device-specific, more information can be found in Section B.1, “PGStat mode properties”.

12.1. PGStat mode off

All measurement 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 frequency 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 sets the channel up to be used as an extra WE electrode that applies a potential relative to the WE of the main channel. This is also known as a bipot or a poly WE. This mode uses the RE and CE of the main channel, and does not use the RE and CE of the poly WE channel.

Chapter 13. Script command summary

13.1. Command summary

The following table lists all MethodSCRIPT commands, in which version they are introduced and which instruments are supported. In chapter Chapter 14, *Script command description* these commands are described in detail.

Table 8. MethodSCRIPT command summary

MethodSCRIPT command	version	EmStat Pico	EmStat4	Description
var	1.1	Y	Y	Declare a variable.
store_var	1.1	Y	Y	Store a value in a variable.
array	1.2	Y	Y	Declare an array.
array_set	1.2	Y	Y	Set a variable at the specified array index.
array_get	1.2	Y	Y	Get a variable from the specified array index.
copy_var	1.1	Y	Y	Copy a variable.
add_var	1.1	Y	Y	Add a value to a variable.
sub_var	1.1	Y	Y	Subtract a value from a variable.
mul_var	1.1	Y	Y	Multiply a variable.
div_var	1.1	Y	Y	Divide a variable.
bit_and_var	1.3	Y	Y	Perform a bitwise AND operation.
bit_or_var	1.3	Y	Y	Perform a bitwise OR operation.
bit_xor_var	1.3	Y	Y	Perform a bitwise XOR operation
bit_lsl_var	1.3	Y	Y	Logical Shift Left variable.
bit_lsr_var	1.3	Y	Y	Logical Shift Right variable.
bit_inv_var	1.3	Y	Y	Bitwise invert a variable.
int_to_float	1.3	Y	Y	Change the data type from *int* to *float*.
float_to_int	1.3	Y	Y	Change the data type from *float* to *int*.
set_e	1.1	Y	Y	Apply a variable or literal as the WE potential.
set_i	1.3	N	Y	Apply a variable or literal as the WE current in galvanostatic mode.
wait	1.1	Y	Y	Wait for the specified amount of time.
set_int	1.2	Y	Y	Configure the interval for the `await_int` command.

MethodSCRIPT command	version	EmStat Pico	EmStat4	Description
await_int	1.2	Y	Y	Wait for the next interval.
loop	1.1	Y	Y	Repeat a block of commands while some condition is fullfilled.
endloop	1.1	Y	Y	Signal the end of a loop.
breakloop	1.2	Y	Y	Break out of the current loop.
if, elseif, else, endif	1.2	Y	Y	Conditional statements allow the conditional execution of commands.
meas	1.1	Y	Y	Measure a data point of the specified type and store the result as a variable.
meas_loop_lsv	1.1	Y	Y	Perform a Linear Sweep Voltammetry (LSV) measurement.
meas_loop_lsp	1.3	N	Y	Perform a Linear Sweep Potentiometry (LSP) measurement.
meas_loop_cv	1.1	Y	Y	Perform a Cyclic Voltammetry (CV) measurement.
meas_loop_dpv	1.1	Y	Y	Perform a Differential Pulse Voltammetry (DPV) measurement.
meas_loop_swv	1.1	Y	Y	Perform a Square Wave Voltammetry (SWV) measurement.
meas_loop_npv	1.1	Y	Y	Perform a Normal Pulse Voltammetry (NPV) measurement.
meas_loop_ca	1.1	Y	Y	Perform a Chronoamperometry (CA) measurement.
meas_loop_cp	1.3	N	Y	Perform a Chronopotentiometry (CP) measurement.
meas_loop_pad	1.1	Y	Y	Perform a Pulsed Amperometric Detection (PAD) measurement.
meas_loop_ocp	1.1	Y	Y	Perform an Open Circuit Potentiometry (OCP) measurement.
meas_loop_eis	1.1	Y	Y	Perform a (potentiostatic) Electrochemical Impedance Spectroscopy (EIS) measurement.
meas_loop_geis	1.3	N	Y	Perform a Galvanostatic Electrochemical Impedance Spectroscopy (GEIS) measurement.
set_autoranging	1.1	Y	Y	Configure the autoranging for all `meas_loop_*` functions.
pck_start	1.1	Y	Y	Start a measurement data packet.

MethodSCRIPT command	version	EmStat Pico	EmStat4	Description
pck_add	1.1	Y	Y	Add a variable (or literal) to the measurement data package previously started with `pck_start`.
pck_end	1.1	Y	Y	Send the measurement data package previously started with `pck_start`, containing all variables added using `pck_add`.
set_max_bandwidth	1.1	Y	Y	Set maximum bandwidth of the signal being measured.
set_cr (deprecated)	1.1	Y	Y	Set the current range for the given maximum current.
set_range	1.3	Y	Y	Set the expected maximum absolute current or potential for a given *VarType*.
set_range_minmax	1.3	Y	Y	Set the expected minimum and maximum current or potential for a given *VarType*.
cell_on	1.1	Y	Y	Turn the cell on. This enables the WE potential or current regulation.
cell_off	1.1	Y	Y	Turn the cell off.
set_pgstat_mode	1.1	Y	Y	Set the PGStat hardware configuration to be used for measurements.
send_string	1.1	Y	Y	Send an arbitrary string as output of the MethodSCRIPT.
set_gpio_cfg	1.2	Y	Y	Set the GPIO pin configuration.
set_gpio_pullup	1.2	Y	Y	Enable or disable GPIO pin pull-ups.
set_gpio	1.1	Y	Y	Set the GPIO output values.
get_gpio	1.2	Y	Y	Get the GPIO input pin values.
set_pot_range (deprecated)	1.2	Y	Y	Set the expected potential range for the following measurements.
set_pgstat_chan	1.1	Y	Y	Select a PGStat channel.
set_poly_we_mode	1.1	Y	N	Select the mode of the additional working electrode.
get_time	1.2	Y	Y	Get the time since device startup in seconds.
file_open	1.2	Y	Y	Open a file on the persistent storage.
file_close	1.2	Y	Y	Close the currently open file.
set_script_output	1.2	Y	Y	Set the output mode for the script.
hibernate	1.2	Y	Y	Put the device in hibernate mode.

MethodSCRIPT command	version	EmStat Pico	EmStat4	Description
i2c_config	1.2	Y	Y	Setup I²C configuration.
i2c_write_byte	1.2	Y	Y	Transmit one byte to an I²C slave device.
i2c_read_byte	1.2	Y	Y	Receive one byte from an I²C slave device.
i2c_write	1.2	Y	Y	Write one or more bytes to an I²C slave device.
i2c_read	1.2	Y	Y	Read one or more bytes from an I²C slave device.
i2c_write_read	1.2	Y	Y	Write to and read from an I²C slave device.
abort	1.2	Y	Y	Abort the current script.
timer_start	1.2	Y	Y	Start the timer.
timer_get	1.2	Y	Y	Get the timer value.
set_channel_sync	1.3	N	Y	Enable or disable channel synchronization.
set_acquisition_frac	1.3	Y	Y	Set the fraction of the iteration time to use for measurement.
meas_fast_cv	1.4	N	Y	Perform a fast Cyclic Voltammetry (CV) measurement.
set_acquisition_frac_autoadjust	1.4	N	Y	Filter out the given frequency by automatically adjusting acquisition times.
set_e_aux	1.4	N	Y	Set the voltage on the AUX DAC.
set_gpio_msk	1.4	-	Y	Write to the GPIO pins indicated by the mask.
get_gpio_msk	1.4	-	Y	Get the GPIO input pin values with a mask.

13.2. MethodSCRIPT version on instruments

The below table lists the relationship between the instrument's firmware version and the MethodSCRIPT version.

Table 9. MethodSCRIPT and instrument firmware versions

MethodSCRIPT	EmStatPico	EmStat4
1.0	v1.0	-
1.1	v1.1	-
1.2	v1.2	v1.0
1.3	v1.3	v1.1
1.4	-	v1.2

Chapter 14. Script command description

14.1. var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Declare a variable. All MethodSCRIPT variables must be declared before use. Currently only names that consist of 1 lowercase character are allowed. When a variable is declared, it is initialized with the floating-point value 0 and *VarType* `aa`.

Arguments

Name	Type	Description
Variable name	*var*	Variable to declare.

Example

```
var a
```

14.2. store_var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Store a value in a variable.

Arguments

Name	Type	Description
Variable name	*var* [out] (*int*, *float*)	Variable to store value into.
Value	*literal* (*int*, *float*)	Literal value to store in the variable.
Variable Type	*VarType*	The type identifier for this value, see Chapter 7, *Variable types*.

Example

Store the value 200 as a floating-point number in the variable `a`, with *VarType* `VT_MISC_GENERIC1` (`ja`).

```
store_var a 200 ja
```

Same as above, but now as an integer value instead of floating-point value.

```
store_var a 200i ja
```

14.3. array

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Declare an array. Arrays can store multiple variables. All arrays must be declared before use. Currently only names that consist of one lower case character are allowed. The name may not be used by another array or variable.

Arrays have a fixed size and their memory is allocated when the command is first run. The minimum size is 1 and the maximum size is determined by the available memory on the device (see Table 6, “Total storage for array elements”). If there is not enough memory available, an error is generated.

It is allowed to declare the same array multiple times (with the same name). This makes it possible to declare an array inside a loop. However, when a variable is declared multiple times, the size must be the same, otherwise an error is generated. When redeclaring an array, the memory is reused.

Note that array memory is not freed until the end of the MethodSCRIPT, so it is best to avoid declaring many large arrays.

When this command is executed, all values in the array are initialized with the floating-point number 0.

Arrays are necessary for some MethodSCRIPT commands, but can also be used in general to store multiple variables, for example inside loops. Values can be written using `array_set` and read using `array_get`. Arrays use zero-based indexing, so the first element has index 0, the second element has index 1, and so on.

Arguments

Name	Type	Description
Variable name	*array*	Array reference.
Array size	*uint32*	The amount of variables this array can hold.

Example

Declare array with name `a` and size 10.

```
array a 10
```

variables and arrays with the same name cannot exist in the same script

14.4. array_set

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Set a variable at the specified array index.

Arguments

Name	Type	Description
Array variable	*array*	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. If a literal is used, the *VarType* will be set to `aa` (`UNKNOWN`).

Example

The following example declares an array `a` with 6 elements, and writes the value 0.02 to the last element (the variable at index 5).

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

To set the *VarType* as well, first define another variable, then store that variable in the array. The following example is similar to the example above, but also sets the *VarType* to `ja`.

```
array a 6i
var t
store_var t 20m ja
array_set a 5i t
```

14.5. array_get

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Get a variable from the specified array index.

Arguments

Name	Type	Description
Array variable	*array*	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

Get the value in the array at index 5 and store it in variable `b`.

```
array_get a 5i b
```

14.6. copy_var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Copy a variable. Copying includes the value, *VarType* and any metadata stored in a variable.

Arguments

Name	Type	Description
Source variable	*var* (*int*, *float*)	Variable to copy.
Destination variable	*var* [out] (*int*, *float*)	Variable to overwrite.

Example

Copies the variable `x` to `y`.

```
copy_var x y
```

14.7. add_var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Add a value to a variable.

The value of `arg2` is added to the variable specified by `arg1`. Both arguments must have the same data type (both `int` or both `float`). The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*, *float*)	Variable to be updated.
arg2	*var* / *literal* (*int*, *float*)	Value to add to `arg1`.

Example

Add 1 to variable `x` and store the result in `x`.

```
add_var x 1
```

14.8. sub_var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Subtract a value from a variable.

The value of `arg2` is subtracted from the variable specified by `arg1`. Both arguments must have the same data type (both `int` or both `float`). The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*, *float*)	Variable to be updated.
arg2	*var* / *literal* (*int*, *float*)	Value to subtract from arg1.

Example

Subtract 1 from the variable `x` and store the result in `x`.

```
sub_var x 1
```

14.9. mul_var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Multiply a variable.

The value of `arg1` is multiplied with the value of `arg2`. Both arguments must have the same data type (both `int` or both `float`). The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*, *float*)	The variable to be multiplied.
arg2	*var* / *literal* (*int*, *float*)	The value to multiply with.

Example

Multiply the variable `x` with 1.5 and stores the result in `x`.

```
mul_var x 1500m
```

14.10. div_var

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Divide a variable.

The value of `arg1` is divided by the value of `arg2`. Both arguments must have the same data type (both `int` or both `float`). The *VarType* and metadata of the variable(s) are not changed.

A floating-point division by zero results in *Not-a-Number*. An integer division by zero is not allowed and results in an error.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*, *float*)	The dividend (as input); the result (quotient) as output.
arg2	*var* / *literal* (*int*, *float*)	The divisor.

Example

Divide the variable `x` by 1.5 and stores the result in `x`.

```
div_var x 1500m
```

14.11. bit_and_var

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Perform a bitwise AND operation.

The value of `arg2` is bitwise ANDed to the variable specified by `arg1`. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*)	Argument 1 of the bit operation, and also the output variable.
arg2	*var* / *literal* (*int*)	Argument 2 of the bit operation.

Example

Perform a bitwise AND operation on `t` and 0x5555 and store it to `t`.

```
bit_and_var t 0x5555
```

14.12. bit_or_var

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Perform a bitwise OR operation.

The value of `arg2` is bitwise ORed to the variable specified by `arg1`. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*)	Argument 1 of the bit operation, and also the output variable.
arg2	*var* / *literal* (*int*)	Argument 2 of the bit operation.

Example

Perform a bitwise OR operation on `t` and 0x5555 and store it to `t`.

```
bit_or_var t 0x5555
```

14.13. bit_xor_var

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Perform a bitwise XOR operation

The value of `arg2` is bitwise XORed to the variable specified by `arg1`. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*)	Argument 1 of the bit operation; also the output variable.
arg2	*var* / *literal* (*int*)	Argument 2 of the bit operation.

Example

Perform a bitwise XOR operation on `t` and 0x5555 and store it to `t`.

```
bit_xor_var t 0x5555
```

14.14. bit_lsl_var

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Logical Shift Left variable.

Shift the variable specified by the first argument to the left by the number of bit positions specified in the second argument. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*)	The variable to shift.
arg2	*var* / *literal* (*int*)	Number of bits to shift.

Example

Perform a bitwise shift 4 places to the left on `t` and store it to `t`.

```
bit_lsl_var t 4i
```

14.15. bit_lsr_var

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Logical Shift Right variable.

Shift the variable specified by the first argument to the right by the number of bit positions specified in the second argument. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*)	The variable to shift.
arg2	*var* / *literal* (*int*)	Number of bits to shift.

Example

Perform a bitwise shift 4 places to the right on `t` and store it to `t`.

```
bit_lsr_var t 4i
```

14.16. bit_inv_var

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Bitwise invert a variable.

The sign bit is also inverted by this operation.

Arguments

Name	Type	Description
Variable	*var* [in/out] (*int*)	The variable to invert, the result is stored here.

Example

Perform a bitwise inverse operation on `t`.

```
bit_inv_var t
```

14.17. int_to_float

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Change the data type from *int* to *float*. Because of the nature of floats, this command will round to the nearest

value. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
Variable	*var* [in/out] (*int*)	Variable to convert.

Example

Convert variable `a` to float.

```
int_to_float a
```

14.18. float_to_int

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Change the data type from *float* to *int*. When changing the data type from floating-point to integer, the fractional part is discarded, i.e., the value is truncated towards zero. If the value is outside the range of an *int32* variable, the result is undefined. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
Variable	*var* [in/out] (*float*)	Variable to convert.

Example

Convert variable `a` to int.

```
float_to_int a
```

14.19. set_e

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Apply a variable or literal as the WE potential. The potential is limited by the potential range of the currently active *PGStat Mode* see Section B.1, “PGStat mode properties”.

Arguments

Name	Type	Description
Potential	*var / literal* (*float*)	The WE potential to apply in Volts.

Example

Set WE potential to 0.1 V.

```
set_e 100m
```

14.20. set_i

MethodSCRIPT	1.3
EmStat Pico	N
EmStat4	Y

Apply a variable or literal as the WE current in galvanostatic mode. Applied currents are limited by the selected CR. It is advised to use the `set_range` command before calling `set_i`.

Arguments

Name	Type	Description
Current	*var / literal* (*float*)	The WE current to apply in amperes.

Example

Sets control current value for the galvanostat loop to 0.1 A.

```
set_range ba 100m
set_i 100m
```

14.21. wait

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Wait for the specified amount of time.

Arguments

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

Example

Wait 100 milliseconds.

```
wait 100m
```

14.22. set_int

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Configure the interval for the `await_int` command. This also (re)starts the counter for the interval timer.

Arguments

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

Example

Set interval to 100 milliseconds.

```
set_int 100m
```

14.23. await_int

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

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

Arguments

-

Example

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.

```
var t
store_var t 0 aa
set_int 100m
# loop until wait time (t) is 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
```

14.24. loop

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Repeat a block of commands while some condition is fullfilled.

Each time the `loop` command is executed, the condition expression is evaluated. If the result is true, the commands between the `loop` and the corresponding `endloop` command are executed. The `endloop` command then jumps back to the `loop` command. If the result of the expression is false, the script continues after the corresponding `endloop` command.

For every `loop` command, there must be exactly one matching `endloop` command.

Arguments

Name	Type	Description
Operand 1	*var* / *literal* (*int*, *float*)	The left side of the conditional expression.
Operator	expression	The operator of the conditional expression.
Operand 2	*var* / *literal* (*int*, *float*)	The right side of the conditional expression.

Example

Add 1 to variable `i` until it reaches the value 10.

Note that the code between the `loop` and `endloop` commands is indented for readability, but this is not required. As described in Chapter 3, *Script format*, whitespace at the start of the line is ignored.

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

14.25. endloop

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Signal the end of a loop.

This command is used to end a `loop` command or any of the measurement loop commands. See the corresponding commands for more details.

Arguments

-

14.26. breakloop

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Break out of the current loop. The script will continue execution after the next `endloop`.

Arguments

-

14.27. if, elseif, else, endif

MethodSCRIPT	1.2
EmStat Pico	Y

EmStat4	Y

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	Description
Operand 1	*var* / *literal* (*int*, *float*)	The left side of the conditional expression.
Operator	expression	The operator of the conditional expression. See Section 8.6, "condition expressions".
Operand 2	*var* / *literal* (*int*, *float*)	The right side of the conditional expression.

Example

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

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

14.28. meas

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Measure a data point of the specified type and store the result as a variable. The data point 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 B.5, "Supported variable types for `meas` command".

Arguments

Name	Type	Description
Duration	*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	*VarType*	The type of variable to measure, see Chapter 7, *Variable types*.

Example

Measure the signal with the *VarType* `ba` (`VT_CURRENT`) for 100 ms and store the result in the variable `c`.

```
meas 100m c ba
```

14.29. meas_loop_lsv

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Perform a Linear Sweep Voltammetry (LSV) measurement. An LSV measurement scans a potential range in small steps and measures the current at each step. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for information about measurement loops in general.

Arguments

Name	Type	Description
Set potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration.
Measured 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.

Name	Type	Description
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.

The set potential is not measured. The actually applied potential may clip if the set potential is outside the supported range.

Optional arguments

The following optional arguments are supported:

- poly_we

Example

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 sweep 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.

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

14.30. meas_loop_lsp

MethodSCRIPT	1.3
EmStat Pico	N
EmStat4	Y

Perform a Linear Sweep Potentiometry (LSP) measurement. An LSP measurement scans a range of currents in small steps and measures the potential at each step. Galvanostatic PGStat mode (6) is required for LSP. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

The resolution and maximum of the output current depend on the selected current range. Make sure to set the expected range before starting the LSP measurement.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Output potential	*var* [out] (*float*)	Output variable to store the measured potential in.
Output current	*var* [out] (*float*)	Output variable to store the set current for this iteration.
Begin current	*var* / *literal* (*float*)	The begin current for the LSP technique.
End current	*var* / *literal* (*float*)	The end current for the LSP technique.
Step current	*var* / *literal* (*float*)	The current 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 current" and "End current".
Scan rate	*var* / *literal* (*float*)	The scan rate of the LSP technique. This is the speed at which the applied current is ramped in A/s. Can only be positive.

Example

Perform an LSP measurement and send a data packet for every iteration. The data packet contains the set current and measured potential. The LSP performs a current sweep from -5 mA to 5 mA with steps of 100 µA at a rate of 1 mA/s. This results in a total of 101 data points at a rate of 10 points per second.

```
meas_loop_lsp p c -5m 5m 100u 1m
   pck_start
   pck_add c
   pck_add p
   pck_end
endloop
```

14.31. meas_loop_cv

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Set potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration.
Measured 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.

Optional arguments

The following optional arguments are supported:

- poly_we
- nscans

Example

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. It steps with 10 mV increments at a rate of 100 mV/s. This results in a total of 201 data points at a rate of 10 points per second.

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

14.32. meas_loop_dpv

MethodSCRIPT	1.1
EmStat Pico	Y

EmStat4	Y

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 "Measured current" variable. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Set potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration.
Measured 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".

Optional arguments

The following optional arguments are supported:

- poly_we

Example

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 20 mV is applied for 5 ms.

```
meas_loop_dpv p c -500m 500m 10m 20m 5m 100m
```

```
    pck_start
    pck_add p
    pck_add c
    pck_end
endloop
```

14.33. meas_loop_swv

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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 "Measured current" variable. The pulse length is "1 / Frequency / 2". A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Set potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration.
Measured 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.

Optional arguments

The following optional arguments are supported:

- poly_we

Example

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 30 mV (2 * 15 mV) is applied for 50 ms (1/Frequency/2).

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

14.34. meas_loop_npv

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Output potential	*var* [out] (*float*)	Output variable to store the pulse potential for this iteration.
Output current	*var* [out] (*float*)	Output variable to store the measured current in.
Begin potential	*var* / *literal* (*float*)	The base potential on which each iteration creates a step.

Name	Type	Description
End potential	*var / literal* (*float*)	The potential of the last pulse.
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".

Optional arguments

The following optional arguments are supported:

- poly_we

Example

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.

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

14.35. meas_loop_ca

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Perform a Chronoamperometry (CA) measurement. In a CA measurement, a DC potential is applied and the current is measured at the specified interval. The measured current is stored in the "Output current" variable. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
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.

Optional arguments

The following optional arguments are supported:

- poly_we

Example

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.

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

14.36. meas_loop_cp

MethodSCRIPT	1.3
EmStat Pico	N
EmStat4	Y

Perform a Chronopotentiometry (CP) measurement. In a CP measurement, a DC current is applied and the potential is measured at the specified interval. The measured potential is stored in the "Output potential" variable. Galvanostatic PGStat mode (6) is required for CP. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Output potential	*var* [out] (*float*)	Output variable to store the measured potential for this iteration.
Output current	*var* [out] (*float*)	Output variable to store the set current in.
DC current	*var* / *literal* (*float*)	The DC current 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.

Example

Perform a CP measurement and send a data packet for every iteration. The data packet contains the measured potential and set current. A DC current of 1 mA is applied. The 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.

```
meas_loop_cp p c 1m 100m 2
   pck_start
   pck_add c
   pck_add p
   pck_end
endloop
```

14.37. meas_loop_pad

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Perform a Pulsed Amperometric Detection (PAD) measurement. In a PAD measurement, potential pulses are periodically applied. Each iteration starts at the DC potential, the current is measured before the pulse (i_{dc}). Then the pulse potential is applied, and the current is measured at the end of the pulse (i_{pulse}). The output current returns a current value depending of one the 3 modes: dc (i_{dc}), pulse (i_{pulse}) or differential ($i_{pulse} - i_{dc}$). A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Output potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration.
Output current	*var* [out] (*float*)	Output variable, content depending on the value of the mode parameter DC mode: i_{dc} Pulse mode: i_{pulse} Differential mode: i_{pulse} - i_{dc}
DC potential	*var* / *literal* (*float*)	The DC 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

Optional arguments

The following optional arguments are supported:

- poly_we

Example

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.

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

14.38. meas_loop_ocp

MethodSCRIPT	1.1

EmStat Pico	Y
EmStat4	Y

Perform an Open Circuit Potentiometry (OCP) measurement. In an OCP measurement, the CE is disconnected so that no potential is applied. Therefore, the cell needs to be turned off (using the `cell_off` command) before starting this measurement. The open circuit RE potential is measured at the specified interval. The measured potential is stored in the "Output potential" variable. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
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.

Example

Perform an OCP measurement and send a data packet for every iteration. The data packet contains the 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.

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

14.39. meas_loop_eis

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Perform a (potentiostatic) Electrochemical Impedance Spectroscopy (EIS) measurement.

Perform a frequency scan and store the resulting Z-real and Z-imaginary in the given variables. High speed potentiostatic 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.

A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Output frequency	*var* [out] (*float*)	Output variable to store the applied frequency (Hz) for this iteration.
Output Z-real	*var* [out] (*float*)	Output variable to store the real part of the measured complex impedance. This field also contains the metadata of the I-signal (current)
Output Z-imaginary	*var* [out] (*float*)	Output variable to store the imaginary part of the measured complex impedance. This field also contains the metadata of the E-signal (potential)
Amplitude	*var* / *literal* (*float*)	Amplitude of the applied sine wave in V_{rms}
Start frequency	*var* / *literal* (*float*)	Start frequency of the scan in Hz
End frequency	*var* / *literal* (*float*)	End frequency of the scan in Hz
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 sine wave in Volt.

Optional arguments

The following optional arguments are supported:

- eis_tdd
- eis_opt
- eis_acdc

Example

Perform an EIS frequency scan from 100 kHz to 100 Hz with 10 mV amplitude and 200 mV DC offset. The frequency for each iteration is returned in variable `f`. The measured complex impedance is returned in 2 variables with Z-real in `r` and Z-imaginary in `i`. In total, 11 points will be measured at frequencies between 100 kHz and 100 Hz, divided on a logarithmic scale.

```
# mode 3= high speed mode
```

```
set_pgstat_mode 3
meas_loop_eis f r i 10m 100k 100 11i 200m
  pck_start
  pck_add f
  pck_add r
  pck_add i
  pck_end
endloop
```

14.40. meas_loop_geis

MethodSCRIPT	1.3
EmStat Pico	N
EmStat4	Y

Perform a Galvanostatic Electrochemical Impedance Spectroscopy (GEIS) measurement.

Perform a frequency scan and store the resulting Z-real and Z-imaginary in the given variables. Galvanostatic PGStat mode (6) is required for GEIS. The following commands currently have no effect on GEIS measurements:

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

A more detailed explanation on this technique can be found on the PalmSens knowledge base.

This is a measurement loop function and needs to be terminated with an `endloop` command. Refer to Chapter 6, *Measurement loop commands* for more information about measurement loops in general.

Arguments

Name	Type	Description
Output frequency	*var* [out] (*float*)	Output variable to store the applied frequency (in Hz) for this iteration.
Output Z-real	*var* [out] (*float*)	Output variable to store the real part of the measured complex impedance. This field also contains the metadata of the I-signal (current).
Output Z-imaginary	*var* [out] (*float*)	Output variable to store the imaginary part of the measured complex impedance. This field also contains the metadata of the E-signal (potential).
Amplitude	*var* / *literal* (*float*)	Amplitude of the applied sine wave in A_{rms}.
Start frequency	*var* / *literal* (*float*)	Start frequency of the scan in Hz.
End frequency	*var* / *literal* (*float*)	End frequency of the scan in Hz.

Name	Type	Description
Nr of points	*var / literal* (*int*, *float*)	Number of frequency points to be scanned.
DC current	*var / literal* (*float*)	DC current offset of the applied sine wave in ampere

Exceeding the maximum amplitude will throw an error, see Appendix B, *Device-specific information* for the maximum amplitude.

Optional arguments

The following optional arguments are supported:

- eis_tdd
- eis_opt
- eis_acdc

Example

Perform an GEIS measurement at frequency `f` with 10 mA_{rms} amplitude and 25 mA DC offset. The measured complex impedance is returned in 2 variables with Z-real in `r` and Z-imaginary in `i`. In total, 11 points will be measured at frequencies between 100 kHz and 100 Hz, divided on a logarithmic scale.

```
# mode 6= galvanostatic
set_pgstat_mode 6
meas_loop_geis f r i 10m 100k 100 11i 25m
  pck_start
  pck_add f
  pck_add r
  pck_add i
  pck_end
endloop
```

14.41. set_autoranging

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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

Arguments

Name	Type	Description
Var type	*VarType*	The type of variable to measure, see Chapter 7, *Variable types*.
Min	*var / literal (float)*	The minimum value in this measurement.
Max	*var / literal (float)*	The maximum value in this measurement.

The *VarType* argument is new in MethodSCRIPT v1.4. To provide backward compatibility with older scripts, the old syntax (with two arguments) is still supported as well. When the first argument is ommitted, the *VarType* `ba` (`VT_CURRENT`) is used. So, `set_autoranging 1u 1m` (old command) is the same as `set_autoranging ba 1u 1m` (new command). The old syntax might be removed in the future.

Example 1

Enable autoranging for currents between 1 µA and 1 mA.

```
set_autoranging ba 1u 1m
```

Example 2

Enable autoranging for potentials between 10 mV and 1 V.

```
set_autoranging ab 10m 1000m
```

14.42. pck_start

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Start a measurement data packet. Up to 33 variables can be added to the packet using the `pck_add` command. The complete packet is transmitted with the `pck_end` command.

Arguments

-

Optional arguments

The following optional arguments are supported:

- meta_msk

Example

Signal the start of a new measurement data package.

```
pck_start
```

14.43. pck_add

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Add a variable (or literal) to the measurement data package previously started with `pck_start`.

Arguments

Name	Type	Description
Variable	*var* / *literal* (*int*, *float*)	The variable to add to the data package.

Example

Add variable `i` to the measurement data package.

```
pck_add i
```

14.44. pck_end

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Send the measurement data package previously started with `pck_start`, containing all variables added using `pck_add`. The `pck_end` command may be called only once after each `pck_start` command.

Arguments

-

Example

Signal the end of a measurement data package.

```
pck_end
```

14.45. set_max_bandwidth

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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 the maximum bandwidth, the signal is attenuated by up to 1% of the potential or current step.

Arguments

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

Example

Set the max bandwidth to a frequency of 1 kHz.

```
set_max_bandwidth 1k
```

14.46. set_cr (deprecated)

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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

The `set_cr` command has been deprecated and may be removed in future releases. Use the `set_range` or `set_range_minmax` command instead.

This command is ignored when autoranging is enabled for `meas_loop_eis`.

Arguments

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

Example

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

```
set_cr 500n
```

It is recommended to use `set_range ba 500n` instead.

14.47. set_range

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Set the expected maximum absolute current or potential for a given *VarType*. The device will automatically configure itself, taking this maximum value into account. Unsupported *VarType*s are ignored without throwing an error.

The following variable types are currently supported:

- Measured current (`ba`): selects the lowest current range that can measure the "Max value" current without causing an overload. This command is ignored in galvanostatic mode.
- Measured potential (`ab`): selects the lowest potential range that can measure the "Max value" current without causing an overload. Devices that do not support potential ranging will ignore this command.
- Applied current (`db`): selects the lowest current range that can apply the "Max value" current without causing an overload. This command is ignored in non-galvanostatic modes.
- Applied potential (`da`): optimises the circuitry to be able to apply the "Max value" potential. This command is ignored in galvanostatic mode.

The following table shows which variable types are supported on which devices:

Variable type	EmStat Pico	Emstat4
`ba`	Yes	Yes
`ab`	No	Yes
`db`	No	Yes
`da`	Yes	No

This command is ignored when autoranging is enabled for `meas_loop_eis`.

Arguments

Name	Type	Description
Variable type	*VarType*	The type identifier for this value (see description above).
Max value	*var / literal* (*float*)	The maximum expected absolute current or potential.

Example

Set current range (`ba`) to be able to measure scurrent between -500 and 500 nA.

```
set_range ba 500n
```

14.48. set_range_minmax

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Set the expected minimum and maximum current or potential for a given *VarType*. The device will automatically configure itself, taking these values into account. Unsupported *VarType*s are ignored without throwing an error.

The following variable types are currently supported:

- Measured current (`ba`): selects the lowest current range that can measure the "Max value" current without causing an overload. This command is ignored in galvanostatic mode.
- Measured potential (`ab`): selects the lowest potential range that can measure the "Max value" current without causing an overload. Devices that do not support potential ranging will ignore this command.
- Applied current (`db`): selects the lowest current range that can apply the "Max value" current without causing an overload. This command is ignored in non-galvanostatic modes.
- Applied potential (`da`): optimises the circuitry to be able to apply the "Max value" potential. This command is ignored in galvanostatic mode. The EmStat Pico requires this command to reach its full applied potential.

The following table shows which variable types are supported on which devices:

Variable type	EmStat Pico	Emstat4
`ba`	Yes	Yes
`ab`	No	Yes
`db`	No	Yes

Variable type	EmStat Pico	Emstat4
`da`	Yes	No

This command is ignored when autoranging is enabled for `meas_loop_eis`.

Arguments

Name	Type	Description
Variable Type	*VarType*	The type identifier for this value (see description above).
Min value	*var / literal* (*float*)	The minimum expected current or potential.
Max value	*var / literal* (*float*)	The maximum expected current or potential.

Example

Set current range (ba) to be able to measure a current of -500 to 500 nA.

```
set_range_minmax ba -500n 500n
```

14.49. cell_on

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Turn the cell on. This enables the WE potential or current regulation. Whether the WE is regulated for current or for potential depends on the selected *PGStat Mode*.

Arguments

-

Example

Turn the cell on. The instrument will start applying the configured potential or current.

```
cell_on
```

14.50. cell_off

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Turn the cell off.

Arguments

-

Example

Turn the cell off. This stops the instrument from applying a potential or current to the cell.

```
cell_off
```

14.51. set_pgstat_mode

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

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.

Arguments

Name	Type	Description
PGStat mode	*uint8*	0 = Off 2 = Low Speed mode 3 = High Speed mode 4 = Max Range mode 5 = Poly WE (BiPot) mode 6 = Galvanostatic mode

Example

Set hardware configuration to high speed mode.

```
set_pgstat_mode 3
```

14.52. send_string

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Send an arbitrary string as output of the MethodSCRIPT. This string is prepended by a `T`, which is the *text* package identifier.

Arguments

Name	Type	Description
Text	*string*	The text to send.

Example

Send the text "hello world".

```
send_string "hello world"
```

Output:

```
Thello world
```

14.53. set_gpio_cfg

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Set the GPIO pin 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 B.6, “Device I/O pin configurations” for available pin configurations per device.

Arguments

Name	Type	Description
Pin mask	*uint32*	Bitmask specifying which pins are configured with this command.
Mode	*uint8*	0 = Digital Input 1 = Digital Output 2 = Peripheral 1 (EmStat Pico only) 3 = Peripheral 2 (reserved for future use)

Example

Set pins 0 and 1 to digital output mode. The prefix *0b* means that the following value is expressed in a binary format.

```
set_gpio_cfg 0b11 1
```

14.54. set_gpio_pullup

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Enable or disable GPIO pin pull-ups.

Arguments

Name	Type	Description
Pin mask	*uint32*	Bitmask specifying which pins are configured with this command. Only input pins should be specified. Configuring the pull-up of an output pin will result in an error.
Pull-up	*uint8*	0 = Pull-up disabled 1 = Pull-up enabled

Example

Enable pull-up on pins 0 and 1. The prefix *0b* means that the following value is expressed in a binary format.

```
set_gpio_pullup 0b11 1
```

14.55. set_gpio

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Set the GPIO output values. This sets the output value of all pins. The output value only has effect when the pin is configured as digital output pin.

Arguments

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

Example

Set the output value of pin 0 and 1 to high and all other pins to low.

```
set_gpio 0b11
```

14.56. get_gpio

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Get the GPIO input pin values. This reads the input value of all GPIO pins, independent of the configured mode. For output pins, the input value will generally be equal to the output value. Bit operations could be used to filter out specific pin values.

Arguments

Name	Type	Description
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. The *VarType* of the variable will be set to `VT_PIN_MSK` (`ec`).

Example

Read the GPIO input values and store the values in variable `g`. Then check the output state of GPIO5.

```
var g
get_gpio g
if g & 0x20
  send_string "GPIO5 is high"
else
  send_string "GPIO5 is low"
endif
```

14.57. set_pot_range (deprecated)

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Set the expected potential range for the following measurements. Some devices cannot apply their full potential range in one measurement, but need to be set up beforehand to reach these potentials. This command 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 is a device-specific command. Currently only the EmStat Pico requires this command to reach its full potential range. The *dynamic potential window* is dependent on the PGStat mode and is defined in Section B.1, "PGStat mode properties".

The `set_pot_range` command has been deprecated and may be removed in future releases. Use the `set_range` or `set_range_minmax` command instead.

Arguments

Name	Type	Description
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

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

```
set_pot_range 0 1200m
```

It is recommended to use `set_range_minmax da 0 1200m` instead.

14.58. set_pgstat_chan

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	Y

Select a PGStat channel. If the device has multiple channels, they can be selected with this command. Both channels can be active at the same time, but the only way to measure both channels simultaneously is in bipotentiostat (bipot) mode, using the `poly_we` optional argument. Refer to the instrument's description document to see how many channels each device has.

Arguments

Name	Type	Description
Channel index	*uint8*	The PGStat channel index to select. A zero-based numbering is used, so the first channel has index 0.

Example

Select the first PGStat channel (channel 0).

```
set_pgstat_chan 0
```

14.59. set_poly_we_mode

MethodSCRIPT	1.1
EmStat Pico	Y
EmStat4	N

Select the mode of the additional working electrode.

Arguments

Name	Type	Description
Poly WE mode	*uint8*	The mode of the additional working electrode: 0 = fixed mode (Additional WE is kept fixed at the specified potential) 1 = offset mode (Additional WE will follow the main WE at a specified offset potential)

Example

Set the additional working electrode mode to offset mode.

```
set_poly_we_mode 1
```

14.60. get_time

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Get the time since device startup in seconds.

The resolution is dependent on the returned *time* value (see table below for estimated resolution). To measure time differences with a higher resolution, use the `timer_start` and `timer_get` commands instead.

Arguments

Name	Type	Description
Variable	*var* [out] (*float*)	The output variable to store the time in. The *VarType* of the variable will be set to `VT_TIME` (`eb`).

Example

Store the current time in variable `t`.

```
get_time t
```

Time accuracy

Internally, the system time is stored with a high resolution. MethodSCRIPT variables, on the other hand, use floating-point representation for which the resolution depends on the actual value. As a result, the resolution of the time returned by the `get_time` command gets lower when the device has been running for a longer time. The table below gives an indication of the resolution to expect for certain system time values. For example, between 10 an 100 days, the value may only distinguish between seconds, but not milliseconds. In a sense, it is comparable with a clock which arms only tick at whole seconds rather than move linearly.

System time	Resolution
< 1 hour	1 ms
1 to 24 hours	10 ms
1 to 10 days	100 ms
10 to 100 days	1 s
≥ 100 days	worse than 1 s

14.61. file_open

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Open a file on the persistent storage. This file can be used to store script output to, using the `set_script_output` command.

Arguments

Name	Type	Description
Path	*string*	The path to the file to open. The path may include folders. Folder names are separated by a slash (`/`).
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

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

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

14.62. file_close

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Close the currently open file. If output to file was enabled (see `set_script_output`), it will be disabled.

If no file is open, this command has no effect.

Arguments

-

Example

Close the currently open file.

```
file_close
```

14.63. set_script_output

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

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

Arguments

Name	Type	Description
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.

Output to file storage is only allowed when a file is currently open, otherwise an error occurs.

Example

Set the script output to be directed to file storage and normal output.

```
set_script_output 3
```

14.64. hibernate

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Put the device in hibernate mode. Hibernate is *deep sleep* mode in which many non-critical components of the instrument are disabled to reduce power consumption. The instrument remains functioning during hibernate, but suspends script execution until any of the enabled wake-up conditions is met. There are three wake-up conditions, that can be enabled individually:

- **Communication:** A character is received over the communication interface (typically UART or USB).
- **WAKE pin:** The WAKE pin is asserted. Each instrument has a dedicated WAKE pin (GPIO5 on the EmStat4, GPIO7 on the EmStat Pico). The pin must be configured correctly (as input pin) when this wake-up source is enabled. On the EmStat4, a low value on the input wakes up the instrument. On the EmStat Pico, a low-to-high transition (falling edge) wakes up the instrument.
- **Timer:** The specified time has passed.

If multiple wake-up sources are enabled, the instrument wakes up as soon as one condition is met.

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

During hibernate, the communication input is flushed, so any commands sent to the device during hibernate might get lost.

Arguments

Name	Type	Description
Wake-up source mask	*uint8*	Bitmask for wake-up sources: 0x01 = Communication 0x02 = WAKE pin 0x04 = Timer At least one wake-up source must be specified.
Wake-up time	*var* / *literal* (*float*)	Time in seconds after which the system is woken up by the system timer. (Must be >0 if the Timer is used as wake-up source.)

Example

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

```
hibernate 7i 60
```

Device-specific information

EmStat Pico

Disabling internal ADT7420 to save power

The hibernate command on the EmStat Pico will disable the on-board ADT7420 temperature sensor to save more power when GPIO8 and GPIO9 are configured for I²C. The current consumption with the temperature sensor enabled is about 250 µA 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 Section 14.53, “set_gpio_cfg” for more information on configuring GPIO.

Shutdown output pin

The EmStat Pico has the ability to set GPIO0 high when in hibernate. This behavior can be activated by configuring GPIO0 in mode 2 (see example below).

```
set_gpio_cfg 0x01 2i
```

Known limitations

- On the EmStat Pico, arrays are not preserved when a hibernate command is issued.
- The minimum hibernation time is 125 ms. Error code `0x4002` will be thrown when the specified time value is too short.

EmStat4

On the EmStat4, the `hibernate` command does not really put the device into hibernate mode, so it does

not decrease the power consumption. It is mainly implemented to be compatible with other MethodSCRIPT instruments. Except for the difference in power consumption, the commands act similarly on all instruments.

14.65. i2c_config

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

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	Description
Clock speed	*var / literal* (*int/float*)	I²C clock speed in Hz. 100 kHz (standard mode) and 400 kHz (fast mode) are officially supported.
Address mode	*literal* (*int/float*)	I²C addressing mode (7-bit or 10-bit)

Example

Configure I²C for standard mode (100 kHz) with 7-bit address.

```
i2c_config 100k 7
```

On the EmStat Pico, make sure the I²C GPIO pins are configured for I²C. *See Section 14.53, "set_gpio_cfg" for more information on configuring GPIO.*

14.66. i2c_write_byte

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Transmit one byte to an I²C slave device. This also generates the 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	Description
Device address	*var* / *literal* (*int*)	The (7-bit or 10-bit) address of the slave device.
Transmit data	*var* / *literal* (*int*)	Data byte to transmit.
ACK status	*var* [in/out] (*int*)	Result of the I²C operation. 0 = ACK received 1 = NACK received for address 2 = NACK received for data 3 = NACK received for address or data The value of the variable must be 0 before executing this command.

The variable passed for the *ACK status* argument should be initialized to 0. Otherwise this command will assume that the previous operation caused a NACK that was not handled by the script and will throw the error code `0x4011`.

Example

Write the value 3 to the device with address 0x48. Abort the script if the I²C operation failed.

```
var a
store_var a 0i ja
i2c_write_byte 0x48 0x03 a
if a != 0i
  abort
endif
```

14.67. i2c_read_byte

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Receive one byte from an I²C slave device. This also generates the 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	Description
Device address	*var* / *literal* (*int*)	The (7-bit or 10-bit) address of the slave device.

Name	Type	Description
Receive data	*var* (*int*)	Variable to store the received byte in.
ACK status	*var* [in/out] (*int*)	Result of the I²C operation. 0 = ACK received 1 = NACK received for address 2 = NACK received for data 3 = NACK received for address or data The value of the variable must be 0 before executing this command.

The variable passed for the *ACK status* argument should be initialized to 0. Otherwise this command will assume that the previous operation caused a NACK that was not handled by the script and will throw the error code `0x4011`.

Example

Read one byte of data from device 0x48 and store it in variable `d`. Abort the script if the I²C operation failed.

```
var a
var d
store_var a 0i ja
i2c_read_byte 0x48i d a
if a != 0i
  abort
endif
```

14.68. i2c_write

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Write one or more bytes to an I²C slave device. This also generates the 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	Description
Device address	*var / literal* (*int*)	The (7-bit or 10-bit) address of the slave device.
Transmit data	array (*int*)	Reference to an array that contains the data to transmit.

Name	Type	Description
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* [in/out] (*int*)	Result of the I²C operation. 0 = ACK received 1 = NACK received for address 2 = NACK received for data 3 = NACK received for address or data The value of the variable must be 0 before executing this command.

The variable passed for the *ACK status* argument should be initialized to 0. Otherwise this command will assume that the previous operation caused a NACK that was not handled by the script and will throw the error code `0x4011`.

Example

Write the values 12 and 34 to the I²C slave device with address 0x48.

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

14.69. i2c_read

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Read one or more bytes from an I²C slave device. This also generates the 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	Description
Device address	*var* / *literal* (*int*)	The (7-bit or 10-bit) address of the slave device.
Received data	array (*int*)	Reference to an array to store received data in.

Name	Type	Description
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* [in/out] (*int*)	Result of the I²C operation. 0 = ACK received 1 = NACK received for address 2 = NACK received for data 3 = NACK received for address or data The value of the variable must be 0 before executing this command.

The variable passed for the *ACK status* argument should be initialized to 0. Otherwise this command will assume that the previous operation caused a NACK that was not handled by the script and will throw the error code `0x4011`.

Example

Read 4 bytes from the I²C slave device with address 0x48 and store them in array `r`.

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

14.70. i2c_write_read

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Write to and read from an I²C slave device. This also generates the I²C start and stop conditions. In contrast with `i2c_read` and `i2c_write`, this command does not generate a STOP condition between writing and reading. 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	Description
Device address	*var* / *literal* (*int*)	The (7-bit or 10-bit) address of the slave device.
Transmit data	array (*int*)	Reference to an 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.

Name	Type	Description
Received data	array (*int*)	Reference to an array to store the 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* [in/out] (*int*)	Result of the I²C operation. 0 = ACK received 1 = NACK received for address 2 = NACK received for data 3 = NACK received for address or data The value of the variable must be 0 before executing this command.

The variable passed for the *ACK status* argument should be initialized to 0. Otherwise this command will assume that the previous operation caused a NACK that was not handled by the script and will throw the error code `0x4011`.

Example

Write 2 bytes to the I²C slave device with address 0x48, and then immediately read 4 bytes.

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

14.71. abort

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Abort the current script. If the script contains an `on_finished:` tag, execution will continue from there, otherwise the script is terminated immediately without error. If an `abort` command is executed inside a (measurement) loop, all `endloop` commands will still be executed. This means that the usual measurement loop output will be generated even when the measurement loop is aborted. Once the `on_finished:` tag has been processed, the `abort` command does not have any effect anymore, i.e. code after the `on_finished:` tag cannot be aborted.

Arguments

-

Example

```
var a
var d
store_var a 0i ja
i2c_read_byte 0x48i d a
if a != 0
  send_string "NACK received"
  abort
endif
# ...continue script here if I2C read succeeded
on_finished:
# ...always execute code after the on_finished: command
```

14.72. timer_start

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Start the timer.

A high-resolution timer is available to conveniently measure (execution) time. The timer is initialized at 0 when the script execution starts, and everytime the `timer_start` command is executed. Because of this, it is less susceptible to decreasing accuracy, and only one MethodSCRIPT variable is necessary to determine the time difference between two moments in the script. The timer value can be read using the `timer_get` command.

Arguments

-

Example

```
timer_start
```

14.73. timer_get

MethodSCRIPT	1.2
EmStat Pico	Y
EmStat4	Y

Get the timer value. This returns the time relative to the last call to `timer_start` (or to the start of the script otherwise). This method can be called multiple times without changing the starting moment.

Arguments

Name	Type	Description
Relative time	*var* [out] (*float*)	The time relative to the last `timer_start` command. The *VarType* of this variable will be set to `VT_TIME` (`eb`).

Example

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

Due to floating-point number limitations the resolution is dependent on the returned time value. For a time resolution of less than 1 ms, the measured time should not exceed 1 hour.

14.74. set_channel_sync

MethodSCRIPT	1.3
EmStat Pico	N
EmStat4	Y

Enable or disable channel synchronization.

On multi-channel devices that support it, the `set_channel_sync` can be used to synchronize measurements between multiple channels. When synchronization is enabled the *slave* device will wait until the *master* enables synchronisation. After that, the slave and master will synchronize their measurement loop start and iterations.

Arguments

Name	Type	Description
Sync enable	*uint8*	0: Disable syncing 1: Enable syncing

Example

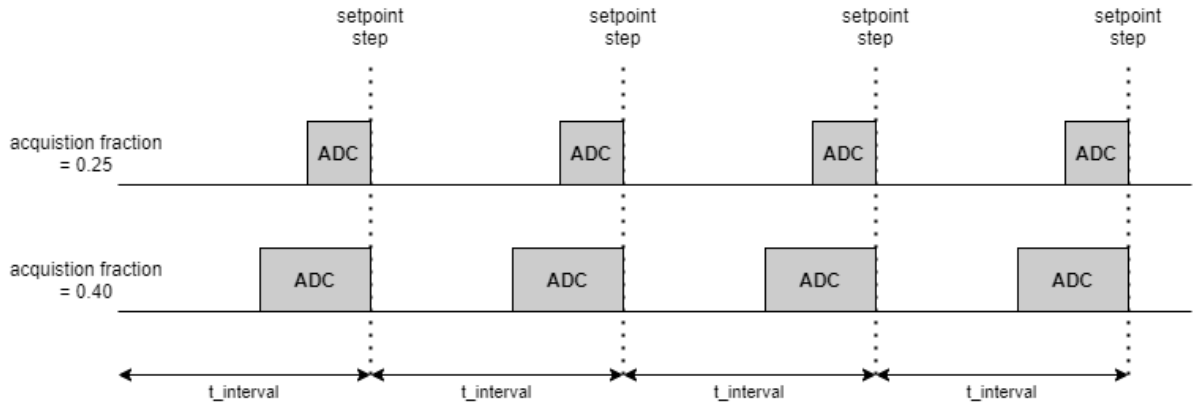
```
# Enable syncing
set_channel_sync 1
```

14.75. set_acquisition_frac

MethodSCRIPT	1.3
EmStat Pico	Y
EmStat4	Y

Set the fraction of the iteration time to use for measurement. This only applies to measurement loops, and the iteration time is determined by the measurement loop command arguments. When multiple signals are to be measured, the acquisition time is shared between them. The fraction must be greater than 0 and smaller than 1.

The following figure shows the time that the Analog-to-Digital Conversion (ADC) is active, for two different settings of the acquisition fraction:



The actual applied fraction could be influenced by the `set_acquisition_frac_autoadjust` command. To prevent this, disable the auto adjustment by setting the frequency to 0.

The `set_pgstat_mode` command initializes the fraction to the default value of 0.25 (= 25%). To change the fraction, this command should therefore be used *after* `set_pgstat_mode`.

A larger fraction means that less time is available for other commands in the measurement loop to be executed, which could result in timing issues if the remaining time is too short. Make sure to check the "status" metadata (see Table 4, “Metadata types.”) to verify that the loop timing was met.

Arguments

Name	Type	Description
Fraction	*var / literal* (*float*)	The fraction (a value between 0 and 1) of the iteration time to use for measurement.

Example

Set acquisition fraction to 25%.

```
set_acquisition_frac 250m
```

14.76. meas_fast_cv

MethodSCRIPT	1.4
EmStat Pico	N
EmStat4	Y

Perform a fast Cyclic Voltammetry (CV) measurement. In a CV measurement, the potential is stepped from the begin potential to the vertex 1, vertex 2 and back to the begin potential. For each step, the current is measured. Contrary to the `meas_loop_cv` function, the fast CV is not implemented as a measurement loop. That means that the script cannot execute other commands during fast CV. Measurement data is stored in arrays and can be transmitted afterwards.

Arguments

Name	Type	Description
Potential	*Array* [out] (*float*)	The array to store the set potentials in.
Current	*Array* [out] (*float*)	The array to store the measured currents in.
Points count	*var* [out] (*int*)	The number of measurement points. The *VarType* of the variable will be set to `VT_MISC_GENERIC1` (`ja`).
Begin potential	*var* / *literal* (*float*)	The potential to start at (and eventually, to end at).
Vertex 1 potential	*var* / *literal* (*float*)	The potential of the first point to change direction in.
Vertex 2 potential	*var* / *literal* (*float*)	The potential of the second point to change direction in.
Step potential	*var* / *literal* (*float*)	The potential step size.
Scan rate	*var* / *literal* (*float*)	The speed at which the scan is performed (in V/s).

The instrument will round its step size to its DAC resolution (see device description document). As a result, the number of points can vary between instruments and may be slightly different than expected. The actual number of points measured will be stored in the *Points count* variable.

Optional arguments

For fast CV, these optional arguments can be combined freely.

- nscans
- nscans_avg
- nscans_equil

`nscans` defines the number of scans to perform sequentially, the result is stored in the *Current* array. The first and last measured sample are both measured at the *begin potential* for symmetry. Splitting the output into multiple scans is quite straightforward. The number of samples per scan is equal to the total number of samples divided by the number of scans.

Currents measured at the last point of one scan are copied and used as first point for the next scan. This is done for convenience and avoids applying the same potential twice in a row.

Index in array	Measurement index	Scan	Potential	Description
0	0	1	0 mV	*Begin potential*
1	1	1	100 mV	*Vertex 1 potential*
2	2	1	0 mV	
3	3	1	-100 mV	*Vertex 2 potential*
4	4	1	0 mV	*Begin potential*
5	**4**	2	0 mV	*Begin potential*, copy of previous point, no extra measurement.
6	5	2	100 mV	*Vertex 1 potential*
7	6	2	0 mV	
8	7	2	-100 mV	*Vertex 2 potential*
9	8	2	0 mV	*Begin potential*

`nscans_equil` steps through all vertexes, just like a regular CV scan. The equillibration scans do not measure the current and are intended to prepare the cell before a the first scan.

`nscans_avg` takes the average of all points over multiple scans while making sure that every potential is set exactly once. This allows averaging more samples to achieve a better signal-to-noise ratio, while still maintaining a low step potential. However, care should be taken that these multiple scans overlap.

Example 1

The following example performs a fast CV without optional arguments. It will start at 0 V, go to vertex 1 at 100 mV before going to -100 mV and back to 0 V. The step size is 10 mV and the scan rate is 1 V/s.

```
array p 41i
array i 41i
var c
meas_fast_cv p i c 0 100m -100m 10m 1
```

Example 2: nscans

The following example performs a fast CV with `nscans` argument to perform 5 scans sequentially.

```
array p 205i
array i 205i
var c
meas_fast_cv p i c 0 100m -100m 10m 1 nscans(5)
```

Example 3: nscans_equil

The following example illustrates fast CV with `nscans_equil` argument to perform 2 scans before actual measurements. After the 2 equilibration scans, a single fast CV scan is performed.

```
array p 41i
array i 41i
var c
meas_fast_cv p i c 0 100m -100m 10m 1 nscans_equil(2)
```

Example 4: nscans_avg

The following example performs a fast CV with `nscans_avg` argument to perform averaging over 3 scans. The format of `p`,`i` and `c` variables is the same as if `nscans_avg` was not performed even though the values are averaged.

```
array p 41i
array i 41i
var c
meas_fast_cv p i c 0 100m -100m 10m 1 nscans_avg(3)
```

Example 5: nscans_equil, nscans and nscans_avg

The following example performs a fast CV with all 3 optional arguments. After equillibrating for 1 scan, 3 scans are performed which are averaged twice each.

```
array p 123i
array i 123i
var c
meas_fast_cv p i c 0 100m -100m 10m 1 nscans_equil(1) nscans(3) nscans_avg(2)
```

An example with an entire fast CV script can be found in Section 15.4, “Fast CV example”.

14.77. set_acquisition_frac_autoadjust

MethodSCRIPT	1.4
EmStat Pico	N
EmStat4	Y

Filter out the given frequency by automatically adjusting acquisition times. The acquisition time is the time in which the signal is actually measured during an iteration. This works on the principle that by adjusting this time to a multiple of the period of a frequency, this frequency is filtered out.

The `set_pgstat_mode` command sets the filtered frequency to a default value of 10 Hz, which will filter out both 50 and 60 Hz. It is recommended to set the frequency to the area's power grid frequency, so that it can be enabled at lower acquisition times. To turn off the auto adjustment, a frequency of 0 Hz can be set. The adjustment will only be applied if the set frequency is lower than `1 / (acquisition time * 2)`. For CA and OCP, it is applied if the frequency is at least equal to `1 / acquisition time`.

The acquisition time is determined by:

- the `set_acquisition_frac` command (by default 25%),
- the interval of the measurement, and
- the number of variables to be measured.

This command does not apply to the `meas`, `meas_loop_eis` and `meas_loop_geis` commands.

Arguments

Name	Type	Description
Frequency	*var* / *literal* (*float*)	The acquisition auto adjust frequency.

Example

Set acquisition auto adjust frequency to filter out 50 Hertz.

```
set_acquisition_frac_autoadjust 50
```

14.78. set_e_aux

MethodSCRIPT	1.4
EmStat Pico	N
EmStat4	Y

Set the voltage on the AUX DAC.

Arguments

Name	Type	Description
Voltage	*var / literal* (*float*)	Output voltage.

Example

```
set_e_aux a
```

14.79. set_gpio_msk

MethodSCRIPT	1.4
EmStat Pico	-
EmStat4	Y

Write to the GPIO pins indicated by the mask. Both *value* and *mask* are bit masks with on bit per pin.

Some pins may be protected on certain instruments or configurations. Writing to these pins will result in an error.

Arguments

Name	Type	Description
Mask	*var / literal* (*int*)	Mask indicating which pins to change, one bit per pin with `1` meaning enabled.
Values	*var / literal* (*int*)	Values to write to masked pins, one bit per pin.

Example

Set the output value of pins 0 and 2 to `1`, and pins 1 and 3 to `0`.

```
set_gpio_msk 0b00001111 0b101
```

14.80. get_gpio_msk

MethodSCRIPT	1.4
EmStat Pico	-
EmStat4	Y

Get the GPIO input pin values with a mask. This reads the input value of all GPIO pins specified by the mask, independent of the configured mode. This is especially useful when multiple things are connected to the GPIO, but only a few pins are relevant. Both returned value and mask have one bit per pin, where a bit with value `1` in the mask means enabled.

Arguments

Name	Type	Description
Mask	*var / literal* (*int*)	Mask indicating which pins to read, one bit per pin with `1` meaning enabled.
Values	var [out] (*int*)	Bitmask that represents the state of the bits specified by the first argument. Bits that are high correspond with a high input signal. The *VarType* of the variable will be set to `VT_PIN_MSK` (`ec`).

Example

Read the input value of GPIO5 and store the value in variable `g`. Then check the output state of GPIO5.

```
var g
get_gpio_msk 0x20 g
if g == 0x20
  send_string "GPIO5 is high"
else
  send_string "GPIO5 is low"
endif
```

Chapter 15. MethodSCRIPT 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.

15.1. 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 10 mV and no DC potential is applied.

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

Example output

```
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
```

15.2. LSV example

The following example script runs an LSV from -0.5 V to 1.5 V in approximately 200 steps of 10 mV. The scan

rate is set to 100 mV/s. After each step, a data packet will be sent containing the set WE potential and the measured WE current. The measured WE current will be used to autorange.

```
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_range_minmax da -500m 1500m
# Set current range to 1 mA.
set_range ba 1m
# Enable autoranging, between current of 100 uA and 5 mA.
set_autoranging ba 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

```
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
```

15.3. SWV example

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

```
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 1.1 V.
set_range_minmax da -500m 690m
# Set current range for a maximum expected current of 2 uA.
set_range ba 2u
# Disable autoranging.
set_autoranging ba 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, and 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

```
M0002
Pda7F85E36u;ba8030DDCp,10,202;ba7FB6915p,10,202;ba7F85B39p,10,202
...
Pda807A1CAu;ba8030EB6p,10,202;ba80AB012p,10,202;ba807A15Cp,10,202
*
```

15.4. Fast CV example

The following example performs a fast CV with 3 scans with 2 averaging passes each. The `meas_fast_cv` command stores the set potential and measured current in arrays which are sent using a loop. This example is intended to run on a 1 kΩ resistor so the current range is set accordingly.

The output can be split into separate scans quite easily because each scan has the same number of points. The number of points per scan is equal to the total number of points divided by the number of scans. In this case, we have 15 points and 3 scans resulting in gives 5 points per scan. The variable `c` holds the total number of points, so splitting could be done in MethodSCRIPT. The second loop in the example does just that.

```
# Variable for number of points measured
var c
# Variable used as loop iterator for points within a scan
var x
# Variable used to store temporary data
var t
# Array to store set potentials
array p 15i
# Array to store measured currents
array i 15i
var s
# Variable used as loop iterator for total points processed
var n
# Configure instrument to perform this measurement on 1k ohm
set_pgstat_chan 0
set_pgstat_mode 2
set_max_bandwidth 1M
set_range_minmax da -110m 110m
set_range_minmax ba -110u 110u
# Set the potentiostat at e_begin and let it settle a bit before applying it on the cell
set_e 0
wait 50m
cell_on
# Perform the actual measurement. Note that this does not have a measurement loop
meas_fast_cv p i c 0 -100m 100m 100m 10 nscans(3) nscans_avg(2)
# Points per scan (s) is points total (c) / nscans (3)
copy_var c s
div_var s 3i
store_var n 0i ja
```

```
# Loop through scans
loop n < c
   store_var x 0i ja
   send_string "scan separator"
   # Loop through points in scan
   loop x < s
      pck_start meta_msk(0x00)
      # Add index to packet
      pck_add n
      # Add set potential to packet
      array_get p n t
      pck_add t
      # Add measured current to packet
      array_get i n t
      pck_add t
      pck_end
      # Increase indexes
      add_var x 1i
      add_var n 1i
   endloop
endloop
cell_off
```

Example output

```
L
Tscan separator
L
Pja8000000i;da8000000 ;ba8022674p
Pja8000001i;da20A34E8n;ba20CCAA8p
Pja8000002i;da8000000 ;ba8024B26p
Pja8000003i;daDF5CB18n;ba801875Fn
Pja8000004i;da8000000 ;ba8024B26p
+
Tscan separator
L
Pja8000005i;da8000000 ;ba8024B26p
Pja8000006i;da20A34E8n;ba20CEF58p
Pja8000007i;da8000000 ;ba8022674p
Pja8000008i;daDF5CB18n;ba801875Fn
Pja8000009i;da8000000 ;ba8024B26p
+
Tscan separator
L
Pja800000Ai;da8000000 ;ba8024B26p
Pja800000Bi;da20A34E8n;ba20CEF58p
Pja800000Ci;da8000000 ;ba8024B26p
Pja800000Di;daDF5CB18n;ba801875Fn
Pja800000Ei;da8000000 ;ba8024B26p
```

```
+
+
```

Our output has the following format: `index;potential;current` Scans are separated by the text "scan separator". MethodSCRIPT also prints an `L` at the start of each loop and and `+` at the end of them.

15.5. I²C example — temperature sensor

The following example script demonstrates how to communicate with the ADT7420 temperature sensor (see datasheet) using I²C. This is the temperature sensor on the EmStat Pico Module. Note that the sensor has I²C bus address 0x48.

The script will first check the ID of the sensor, then configure it for 16-bit continuous mode, and read and log 40 temperature measurements. This will take approximately 10 seconds. If the script is executed using PSTrace, a plot of the temperature over time will be shown.

```
# I2C ACK status
var a
# byte
var b
# loop counter
var i
# Status register value
var s
# MSB of temperature
var m
# LSB of temperature
var l
# Time
var t
# Read buffer
array r 2
# Write buffer
array w 2
# Configure GPIO8-9 for I2C (Mode 2)
set_gpio_cfg 0x0300 2
# Configure I2C peripheral to 100 kHz clock, 7-bit address.
i2c_config 100k 7
# Initialize ACK status at 0.
store_var a 0i ja
# Read and check device ID.
array_set w 0i 0x0B
i2c_write_read 0x48 w 1i r 1i a
if a != 0i
  abort
endif
array_get r 0i b
if b != 0xCB
  send_string "ERROR: Invalid ID (not an ADT7420 device)"
```

```
  abort
endif
# Configure the sensor for 16-bit mode with continuous conversion
# by writing value 0x80 to address 0x03 (configuration register).
array_set w 0i 0x03
array_set w 1i 0x80
i2c_write 0x48 w 2i a
if a != 0i
  abort
endif
# Start timer and logging temperature measurements.
timer_start
store_var i 0i ja
loop i < 40i
  # Read status register until measurement ready.
  array_set w 0i 0x02
  store_var s 0x80 ja
  loop s & 0x80
    i2c_write_read 0x48 w 1i r 1i a
    if a != 0i
      abort
    endif
    array_get r 0i s
  endloop
  # Read timer.
  timer_get t
  # Read temperature value.
  array_set w 0i 0x00
  i2c_write_read 0x48 w 1i r 2i a
  if a != 0i
    abort
  endif
  # Convert temperature.
  array_get r 0i m
  array_get r 1i l
  # Combine MSB + LSB in one variable.
  bit_lsl_var m 8i
  bit_or_var m l
  # Handle negative temperatures.
  if m & 0x8000
    sub_var m 65536i
  endif
  # Convert to float and divide by 128 to get temperature in degrees Celsius.
  int_to_float m
  div_var m 128
  pck_start
  pck_add t
  pck_add m
  pck_end
  add_var i 1i
```

```
endloop
on_finished:
if a == 1i
  send_string "ERROR: I2C address NACK"
elseif a == 2i
  send_string "ERROR: I2C data NACK"
elseif a == 3i
  send_string "ERROR: I2C data or address NACK"
endif
```

Example output

```
L                       ← Start of outer loop (i < 40)
L                       ← Start of wait loop (wait until measurement ready)
+                       ← End of wait loop
Peb803B5BDu;aa934837Cu  ← Data package containing time and temperature
L                       ← Start of wait loop (wait until measurement ready)
+                       ← End of wait loop
Peb80767C9u;aa934C086u  ← Data package containing time and temperature
L                       ← Start of wait loop (wait until measurement ready)
+                       ← End of wait loop
Peb80B1A11u;aa93464F8u  ← Data package containing time and temperature
...                     ← Inner loop repeated 37 more times
+                       ← End of outer loop
```

15.6. I²C example — real time clock

The below example script demonstrates the use of I²C in combination with the ABLIC S-35390A RTC that can be found on the EmStat Pico Development Kit. It sets the time and date to the arbitrary value of 2:14 AM 29-08-2097. Then it will wait 10 seconds and read back the time. See the datasheet of the RTC for a description of the register formats and how to use it correctly.

```
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
```

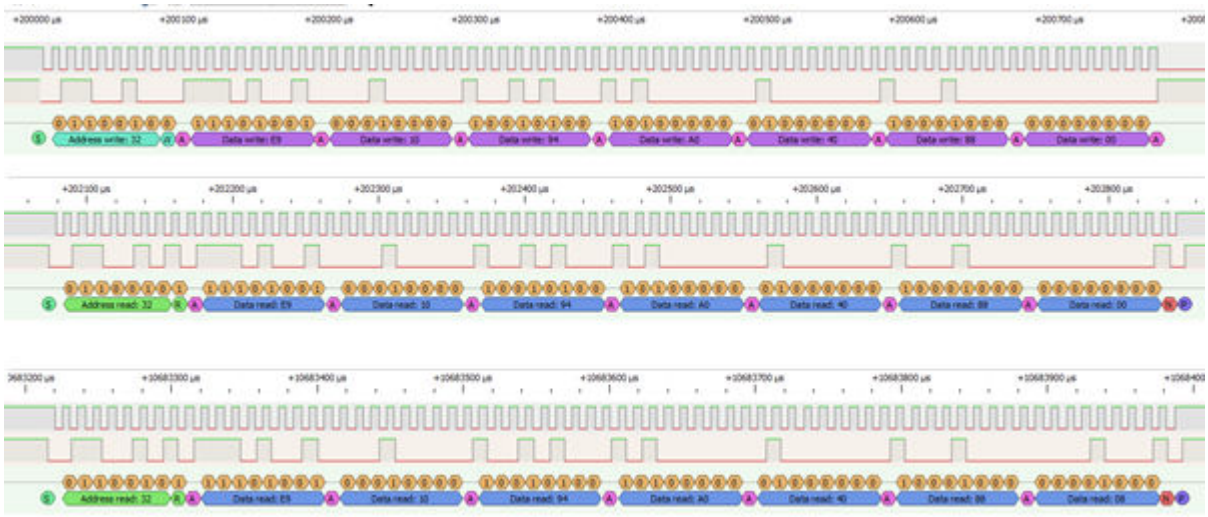
```
array_set w 4i 0x40i
# Minute = 14
array_set w 5i 0x88i
# Seconds = 0
array_set w 6i 0x00i
# Configure I2C GPIOs and set it to 100 kHz clock, 7-bit address
set_gpio_cfg 0x0300i 2
i2c_config 100k 7
# Write data to real-time data registers
i2c_write 0x32i w 7i a
# Printing the time as it was written.
i2c_read 0x32i r 7i a
store_var i 0i ja
loop i < 7i
    array_get r i d
    pck_start
    pck_add d
    pck_end
    add_var i 1i
endloop
# 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
store_var i 0i ja
loop i < 7i
    array_get r i d
    pck_start
    pck_add d
    pck_end
    add_var i 1i
endloop
```

Example output

```
L
Paa80000E9i
Paa8000010i
Paa8000094i
Paa80000A0i
Paa8000040i
Paa8000088i
Paa8000000i
+
TWaiting for the time to change.
L
Paa80000E9i
Paa8000010i
```

```
Paa8000094i
Paa80000A0i
Paa8000040i
Paa8000088i
Paa8000090i
+
```

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.



15.7. I²C example — EEPROM

The following example demonstrates writing to and reading from the 24LC32A EEPROM on the EmStat Pico Development Kit. It will write a counter to the EEPROM and read it back later. Note that the EEPROM may require some time to finish the write operation before a read will be successful.

```
# Acknowledge value
var a
var b
# Loop variable
var i
# Temporary value
var v
store_var a 123i ja
# Write array, 2 bytes address + 32 bytes data
array w 34i
# Read array, 32 bytes data
array r 32i
# Configure I2C with 400 kHz clock and 7-bit address
set_gpio_cfg 0x0300i 2i
i2c_config 400k 7i
# EEPROM register address MSB (1) and LSB (64) to form 320
array_set w 0i 1i
```

```
array_set w 1i 64i
# Write data values 0-32 to bytes 2-34 of the array
store_var i 2i ja
store_var v 0i ja
loop i < 34i
    array_set w i v
    add_var i 1i
    add_var v 1i
endloop
# Write to device
store_var b 0i ja
i2c_write 0x50i w 34i b
# Handle ACK/NACK
if b != 0i
    send_string "FAILED to write to EEPROM"
    abort
endif
# Read EEPROM. Will generate NACK until write is completed.
# Variable b is set to 1 to enter the loop.
store_var b 1i ja
loop b != 0i
    # Reset var b so I2C will not fail when receiving b NACK
    store_var b 0i ja
    # Note the address from the write array is reused
    i2c_write_read 0x50i w 2i r 32i b
    send_string "reading EEPROM"
endloop
# Print the received data
store_var i 0i ja
loop i < 32i
    pck_start
    array_get r i v
    pck_add v
    pck_end
    add_var i 1i
endloop
```

Example output

```
L
+
L
Treading EEPROM
Treading EEPROM
Treading EEPROM
Treading EEPROM
Treading EEPROM
Treading EEPROM
Treading EEPROM
```

```
Treading EEPROM
Treading EEPROM
Treading EEPROM
Treading EEPROM
+
L
Paa8000000i
Paa8000001i
Paa8000002i
Paa8000003i
Paa8000004i
Paa8000005i
Paa8000006i
Paa8000007i
Paa8000008i
Paa8000009i
Paa800000Ai
Paa800000Bi
Paa800000Ci
Paa800000Di
Paa800000Ei
Paa800000Fi
Paa8000010i
Paa8000011i
Paa8000012i
Paa8000013i
Paa8000014i
Paa8000015i
Paa8000016i
Paa8000017i
Paa8000018i
Paa8000019i
Paa800001Ai
Paa800001Bi
Paa800001Ci
Paa800001Di
Paa800001Ei
Paa800001Fi
+
```

Chapter 16. Document 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

Version 1.2 Rev 2

- Added EEPROM example
- Moved EmStat Pico specific information to chapter "device-specific information"
- Added reference to comparator in `loop` and `if` command documentation
- Removed outdated warning that `meas_loop_eis` does not support autoranging

Version 1.3 Rev 1

- Added I²C generic NACK for address or data (for devices that cannot distinguish)
- Added EmStat4 information
- `set_autoranging` changed having additional *VarType* parameter
- Added `eis_tdd` command to retrieve EIS time domain data
- Replaced `set_cr` and `set_potential_range` commands with more generic `set_range` and `set_range_minmax` commands
- Added CP technique
- Added LSP technique
- Added Galvanostatic EIS technique
- Added `set_i` command
- Updated error codes
- Updated features section
- Updated terminology
- `set_pgstat_mode` now resets all mode settings to default values
- Added `set_channel_sync` command
- Added bitwise operation commands

- Added `float_to_int` and `int_to_float` commands
- Added galvanostat pgstat mode
- Added `set_acquisition_frac` command
- Added potential ranges in metadata

Version 1.4 Rev 1

- General document changes:
  - Rearranged chapters, moved large tables to appendix
  - Updated document formatting
- Chapter 3:
  - Clarified relation between device communication protocol and MethodSCRIPT
- Chapter 14:
  - Added list of supported instruments and MethodSCRIPT versions for each command
  - Updated documentation of some commands
- Chapter 15:
  - Updated I²C example scripts
  - Added links to datasheets of S-35390A (RTC) and ADT7420 (temperature sensor)
  - Added EEPROM example
- Appendix A:
  - Updated error codes
  - Added table mapping instrument firmware versions to MethodSCRIPT versions
  - Updated variable types
- MethodSCRIPT changes:
  - Updated line numbers to also include comments
  - Updated behavior of `pck_start`/`pck_add`/`pck_end` commands
  - Added fast Cyclic Voltammetry technique (`meas_fast_cv` command)
  - Added frequency filtering with `set_acquisition_frac_autoadjust` command
  - Added `set_e_aux` command
  - Added masked versions of GPIO commands (`set_gpio_msk` and `get_gpio_msk`)

Appendix A: Error codes

The following table lists all error codes that can be returned by MethodSCRIPT instruments.

Note that some of these error codes are part of the communication protocol (e.g. 0004–0006, 0008, 0009), while others are only returned during MethodSCRIPT loading (e.g. 0003, 4001) or MethodSCRIPT execution (e.g. 0028, 400F). Some more generic error codes (e.g. 0001) are applicable for both. In this table, no distinguishment is made between the source of the error codes. Instead, all codes are included and sorted by number so they can be quickly referenced when troubleshooting.

The error codes and their meaning are the same for all instruments and firmware versions. However, in some cases, the same error condition could result in a different error code when using another instrument or firmware version.

Table 10. Error code lookup table

Error code	Description
0x0001	An unspecified error has occurred
0x0002	An invalid *VarType* has been used
0x0003	The command was not recognized
0x0004	Unknown register
0x0005	Register is read-only
0x0006	Communication mode invalid
0x0007	An argument has an unexpected value
0x0008	Command exceeds maximum length
0x0009	The command has timed out
0x000B	Cannot reserve the memory needed for this var
0x000C	Cannot run a script without loading one first
0x000E	An overflow has occurred while averaging a measured value
0x000F	The given potential is not valid
0x0010	A variable has become either "NaN" or "inf"
0x0011	The input frequency is invalid
0x0012	The input amplitude is invalid
0x0014	Cannot perform OCP measurement when cell on
0x0015	CRC invalid
0x0016	An error has occurred while reading / writing flash
0x0017	The specified flash address is not valid for this device
0x0018	The device settings have been corrupted

Error code	Description
0x0019	Authentication error
0x001A	Calibration invalid
0x001B	This command or part of this command is not supported by the current device
0x001C	Step Potential cannot be negative for this technique
0x001D	Pulse Potential cannot be negative for this technique
0x001E	Amplitude cannot be negative for this technique
0x001F	Product is not licensed for this technique
0x0020	Cannot have more than one high speed and/or max range mode enabled (EmStat Pico)
0x0021	The specified PGStat mode is not supported
0x0022	Channel set to be used as Poly WE is not configured as Poly WE
0x0023	Command is invalid for the selected PGStat mode
0x0024	The maximum number of vars to measure has been exceeded
0x0025	The specified PAD mode is unknown
0x0026	An error has occurred during a file operation
0x0027	Cannot open file, a file with this name already exists
0x0028	Variable divided by zero
0x0029	GPIO pin mode is not known by the device
0x002A	GPIO configuration is incompatible with the selected operation
0x002B	CRC of received line was incorrect (CRC16-ext)
0x002C	ID of received line was not the expected value (CRC16-ext)
0x002D	Received line was too short to extract a header (CRC16-ext)
0x002E	Settings are not initialized
0x002F	Channel is not available for this device
0x0030	Calibration process has failed
0x0032	Critical cell overload, aborting measurement to prevent damage.
0x0033	FLASH ECC error has occurred
0x0034	Flash program operation failed
0x0035	Flash Erase operation failed
0x0036	Flash page/block is locked
0x0037	Flash write operation on protected memory
0x0038	Flash is busy executing last command.

Error code	Description
0x0039	Operation failed because block was marked as bad
0x003A	The specified address is not valid
0x003B	An error has occurred while attempting to mount the filesystem
0x003C	An error has occurred while attempting to format the filesystem memory
0x003D	A timeout has occurred during SPI communication
0x003E	A timeout has occurred somewhere
0x003F	The calibrations registers are locked, write actions not allowed.
0x0040	Memory module not supported.
0x0041	Flash memory format not recognized or supported.
0x0042	This register is locked for current permission level.
0x0043	Register is write-only
0x0044	Command requires additional initialization
0x0045	Configuration not valid for this command
0x0046	No mux was found during auto-detect.
0x0047	The filesystem has to be mounted to complete this action.
0x0048	This device is not a multi-device, no serial available.
0x0049	GPIO configuration is incompatible with the ES4X IO AUX port
0x004A	MCU register access is not allowed, only RAM and peripherals are accessible.
0x004B	Runtime (comm) command argument too short to be valid.
0x004C	Runtime (comm) command argument has an invalid format.
0x004E	Hibernate wake up source is invalid
0x004F	Hibernate requires at least one wake up source, none was given.
0x0050	Wake pin for hibernate not configured as `input`
0x0051	The code provided to the permission register was not valid.
0x0052	An overrun error occurred on a communication interface (e.g. UART).
0x0053	Argument length incorrect for this register.
0x0055	The GPIO pins requested to change do not exist on this instrument
0x0056	The GPIO pin is reserved for a special purpose (by NVM config or device type)
0x0057	The on-board flash module has timed out.
0x0200	COMM argument value cannot be negative for this command
0x0201	COMM argument value cannot be positive for this command

Error code	Description
0x0202	COMM argument value cannot be zero for this command
0x0203	COMM argument value must be negative for this command (also not zero)
0x0204	COMM argument value must be positive for this command (also not zero)
0x0205	COMM argument value is outside the allowed bounds for this command
0x0206	COMM argument value cannot be used for this specific instrument
0x0207	An unexpected additional COMM argument was provided
0x4001	The script command is unknown
0x4004	An unexpected character was encountered
0x4005	The script is too large for the internal script memory
0x4008	This optional argument is not valid for this command
0x4009	The stored script is generated for an older firmware version and cannot be run
0x400B	Measurement loops cannot be placed inside other measurement loops
0x400C	Command not supported in current situation
0x400D	Scope depth too large
0x400E	The command had an invalid effect on scope depth
0x400F	Array index out of bounds
0x4010	I2C interface was not initialized with i2c_config command
0x4011	This is an error, NAck flag not handled by script
0x4012	Something unexpected went wrong.
0x4013	I2C clock frequency not supported by hardware
0x4014	Non integer SI vars cannot be parsed from hex or binary representation
0x4016	RTC was selected as wake-up source and selected time is not supported
0x4018	The script has ended unexpectedly.
0x4019	The script command is only valid for a multichannel (combined) device
0x401A	Fast_cv cannot be called from within a measurement loop.
0x401B	the pck sequence is called wrong
0x401C	The maximum amounts of variables per packet has been exceeded.
0x4020	A timeout has occurred for one of the script commands
0x4021	The mux is not initialized/configured.
0x4022	Measurement loop timing is too fast to use with multiplexer
0x4023	The script command is only valid for a device with IR compensation

Error code	Description
0x4024	The resistance value is to big for the whole autorange range
0x4025	The resistance value is to big for current current range
0x4026	The variable already exists when declared
0x4027	This command requires the cell to be enabled with the `cell_on` command
0x4028	This command requires the cell to be disabled with the `cell_off` comma
0x4029	The technique requires that at least one step should be made
0x4200	MScript argument value cannot be negative for this command
0x4201	MScript argument value cannot be positive for this command
0x4202	MScript argument value cannot be zero for this command
0x4203	MScript argument value must be negative for this command (also not zero)
0x4204	MScript argument value must be positive for this command (also not zero)
0x4205	MScript argument value is outside the allowed bounds for this command
0x4206	MScript argument value cannot be used for this specific instrument
0x4207	MScript argument datatype (float/int) is invalid for this command
0x4208	MScript argument reference was invalid (not 'a' - 'z')
0x4209	MScript argument variable type is not supported for this command
0x420A	An unexpected, additional, (optional) MScript argument was provided
0x420B	MScript argument variable is not declared
0x420C	MScript argument is of type var, which is not supported by this command
0x420D	MScript argument is of type literal, which is not supported by this command
0x420E	MScript argument is of type array, which is not supported by this command
0x420F	MScript argument array size is insufficient
0x7FFF	A fatal error has occurred, the device must be reset

Appendix B: Device-specific information

B.1. PGStat mode properties

Low Speed mode For low frequency, low noise applications.

High Speed mode For high scan rates and frequencies.

Max Range mode A combination of the Low and High Speed modes for optimal dynamic DC potential range.

The EmStat4 accepts the *Low Speed*, *High Speed*, and *Max Range* modes, but there is no functional difference between these modes.

B.1.1. EmStat4 HR

Table 11. Potentiostat mode properties for EmStat4 HR.

Parameter	Min. value	Max. value
Bandwidth	-	500 kHz
Potential range	-6.0 V	6.0 V
Dynamic potential window	-6.0 V	6.0 V

Table 12. Galvanostat mode properties for EmStat4 HR.

Parameter	Min. value	Max. value
Bandwidth	-	500 kHz
Current range	-200 mA	200 mA

B.1.2. EmStat4 LR

Table 13. Potentiostat mode properties for EmStat4 LR.

Parameter	Value min	Value max
Bandwidth	-	500 kHz
Potential range	-3.0 V	3.0 V
Dynamic potential window	-3.0 V	3.0 V

Table 14. Galvanostat mode properties for EmStat4 LR.

Parameter	Min. value	Max. value
Bandwidth	-	500 kHz
Current range	-30 mA	30 mA

B.1.3. EmStat Pico

Table 15. EmStat Pico low speed mode properties.

Parameter	Min. value	Max. value
Bandwidth	0.016 Hz	100 Hz
Potential range	-1.25 V	2.0 V
Dynamic potential window	2.2 V	2.2 V

Table 16. EmStat Pico high speed mode properties.

Parameter	Min. value	Max. value
Bandwidth	0.016 Hz	200 kHz
Potential range	-1.7 V	2.0 V
Dynamic potential window	1.214 V	1.214 V

Table 17. EmStat Pico max range mode properties.

Parameter	Min. value	Max. value
Bandwidth	0.016 Hz	100 Hz
Potential range	-1.7 V	2.0 V
Dynamic potential window	2.6 V	2.6 V

B.2. EIS properties

Table 18. EmStat4 potentiostatic EIS properties.

Parameter	Value
Max. amplitude (V_{RMS})	0.900 V
Max. frequency	200 kHz

Table 19. EmStat4 galvanostatic EIS (GEIS) properties.

Parameter	Value
Max. amplitude (A_{RMS})	0.9 x CR[1]
Max. frequency	200 kHz

[1] With GEIS, the maximum amplitude is a factor of the selected current range, e.g., at 10 mA CR the max. (RMS) amplitude is 9 mA.

Table 20. EmStat Pico potentiostatic EIS properties.

Parameter	Value
Max. amplitude (V_{RMS})	0.429 V

Parameter	Value
Max. frequency	200 kHz

B.3. Current ranges

B.3.1. EmStat4 LR

Table 21. EmStat4 LR potentiostat current ranges.

Current range	Index
1 nA	0x03
10 nA	0x06
100 nA	0x09
1 µA	0x0C
10 µA	0x0F
100 µA	0x12
1 mA	0x15
10 mA	0x18

Table 22. EmStat4 LR galvanostat current ranges.

Current range	Index
10 nA	0x06
1 µA	0x0C
100 µA	0x12
10 mA	0x18

B.3.2. EmStat4 HR

Table 23. EmStat4 HR potentiostat current ranges.

Current range	Index
100 nA	0x09
1 µA	0x0C
10 µA	0x0F
100 µA	0x12
1 mA	0x15
10 mA	0x18
100 mA	0x1B

Table 24. EmStat4 HR galvanostat current ranges.

Current range	Index
1 µA	0x0C
100 µA	0x12
10 mA	0x18
100 mA	0x1B

B.3.3. EmStat Pico

Table 25. EmStat Pico low speed mode.

Current range	Index
100 nA	0x0
1.95 µA	0x1
3.91 µA	0x2
7.81 µA	0x3
15.63 µA	0x4
31.25 µA	0x5
62.5 µA	0x6
125 µA	0x7
250 µA	0x8
500 µA	0x9
1 mA	0xA
5 mA	0xB

Table 26. EmStat Pico high speed mode.

Current range	Index
100 nA	0x80
1 µA	0x81
6.25 µA	0x82
12.5 µA	0x83
25 µA	0x84
50 µA	0x85
100 µA	0x86
200 µA	0x87

Current range	Index
1 mA	0x88
5 mA	0x89

Table 27. EmStat Pico max range mode.

Current range	Index
100 nA	0x80
1 µA	0x81
6.25 µA	0x82
12.5 µA	0x83
25 µA	0x84
50 µA	0x85
100 µA	0x86
200 µA	0x87
1 mA	0x88
5 mA	0x89

B.4. Potential ranges

Table 28. EmStat4 HR/LR galvanostat potential ranges.

Potential range	Index
10 mV	0
20 mV	1
50 mV	2
100 mV	3
200 mV	4
500 mV	5
1 V	6

B.5. Supported variable types for `meas` command

Table 29. Supported variable types EmStat4.

Name	ID
`VT_POTENTIAL`	`ab`
`VT_POTENTIAL_CE`	`ac`

Name	ID
VT_POTENTIAL_RE	ae
VT_POTENTIAL_WE_VS_CE	ag
VT_POTENTIAL_AIN0	as
VT_CURRENT	ba

Table 30. Supported variable types EmStat Pico.

Name	ID
VT_POTENTIAL	ab
VT_POTENTIAL_CE	ac
VT_POTENTIAL_RE	ae
VT_POTENTIAL_WE_VS_CE	ag
VT_POTENTIAL_AIN0	as
VT_POTENTIAL_AIN1	at
VT_POTENTIAL_AIN2	au
VT_CURRENT	ba

B.6. Device I/O pin configurations

Table 31. EmStat4 I/O pin configuration.

Bitmask	Pin name	Mode 0	Mode 1
0x0001	GPIO0	Digital Input	Digital Output
0x0002	GPIO1	Digital Input	Digital Output
0x0004	GPIO2*	Digital Input	Digital Output
0x0008	GPIO3	Digital Input	Digital Output
0x0010	GPIO4	Digital Input	Digital Output
0x0020	GPIO5_WAKE	Digital Input	Digital Output
0x0040	GPIO6_PWM	Digital Input	Digital Output

* On some devices, such as the EmStat4R / EmStat4 Go, GPIO2 is used for the external cell LED and cannot be used as general-purpose I/O pin.

Table 32. EmStat Pico I/O pin configuration.

Bitmask	Pin name	Mode 0	Mode 1	Mode 2
0x0001	GPIO0_PWM	Digital Input	Digital Output	Shutdown (output)
0x0002	GPIO1_SPI_MISO†	Digital Input	Digital Output	SPI flash memory
0x0004	GPIO2_SPI_CLK†	Digital Input	Digital Output	SPI flash memory

Bitmask	Pin name	Mode 0	Mode 1	Mode 2
0x0008	GPIO3_SPI_MOSI†	Digital Input	Digital Output	SPI flash memory
0x0010	GPIO4_SPI_CS0†	Digital Input	Digital Output	SPI flash memory
0x0020	GPIO5	Digital Input	Digital Output	
0x0040	GPIO6*	Digital Input	Digital Output	
0x0080	GPIO7_WAKE	Digital Input	Digital Output	Wake from sleep (Active low)
0x0100	I2C_SCL	Digital Input	Digital Output	I²C
0x0200	I2C_SDA	Digital Input	Digital Output	I²C

* On some devices, such as the Sensit BT, GPIO6 is used for the external cell LED and cannot be used as general-purpose I/O pin.
† For devices with on-board storage memory, such as the Sensit BT, GPIO1–4 are reserved and cannot be used as general-purpose I/O pins.

Appendix C: Variable types

The following table lists all variable types that are defined in MethodSCRIPT. All IDs not listed in this table are reserved for future use. It is not recommended to use other variable types than the ones listed in this table.

Table 33. Variable types lookup table

Name	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_SE	ad	Measured SE voltage vs GND
VT_POTENTIAL_RE	ae	Measured RE voltage vs GND
VT_POTENTIAL_WE	af	Measured WE 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	ca	Measured phase
VT_IMP	cb	Measured impedance
VT_ZREAL	cc	Measured real part of complex impedance
VT_ZIMAG	cd	Measured imaginary part of complex impedance
VT_EIS_TDD_E	ce	Measured RE potential Time Domain Data
VT_EIS_TDD_I	cf	Measured WE current Time Domain Data
VT_EIS_FS	cg	Sampling frequency used for EIS measurement
VT_EIS_E_AC	ch	Measured AC potential
VT_EIS_E_DC	ci	Measured DC potential
VT_EIS_I_AC	cj	Measured AC current
VT_EIS_I_DC	ck	Measured DC current
VT_CELL_SET_POTENTIAL	da	Set control value for WE potential
VT_CELL_SET_CURRENT	db	Set control value for WE current
VT_CELL_SET_FREQUENCY	dc	Set value for frequency
VT_CELL_SET_AMPLITUDE	dd	Set value for ac amplitude
VT_TIME	eb	Time in seconds

Name	ID	Description
VT_PIN_MSK	ec	Binary pin bitmask, indicating which pins are high / low
VT_TEMPERATURE	ed	Temperature in degrees Celsius
VT_CURRENT_GENERIC1	ha	Generic current 1
VT_CURRENT_GENERIC2	hb	Generic current 2
VT_CURRENT_GENERIC3	hc	Generic current 3
VT_CURRENT_GENERIC4	hd	Generic current 4
VT_POTENTIAL_GENERIC1	ia	Generic potential 1
VT_POTENTIAL_GENERIC2	ib	Generic potential 2
VT_POTENTIAL_GENERIC3	ic	Generic potential 3
VT_POTENTIAL_GENERIC4	id	Generic potential 4
VT_MISC_GENERIC1	ja	Miscellaneous value 1
VT_MISC_GENERIC2	jb	Miscellaneous value 2
VT_MISC_GENERIC3	jc	Miscellaneous value 3
VT_MISC_GENERIC4	jd	Miscellaneous value 4