PalmSens

MethodSCRIPT™

MethodSCRIPT manual

Version 1.8, 2025-09-17

Table of Contents

1. Introduction . . . 1
    1.1. Terminology . . . 1
2. Features . . . 2
    2.1. Implemented features . . . 2
    2.2. Supported devices . . . 3
3. Script format . . . 4
    3.1. Relation between MethodSCRIPT and communication protocol . . . 4
4. MethodSCRIPT variables . . . 6
    4.1. MethodSCRIPT variables . . . 6
    4.2. Script command variables . . . 7
    4.3. Measurement data package variables . . . 7
        4.3.1. Invalid Values . . . 8
5. Interpreting measurement data packages . . . 9
    5.1. Package format . . . 9
    5.2. Variable sub package format . . . 9
    5.3. Package parsing example . . . 10
6. Measurement loop commands . . . 12
    6.1. Introduction . . . 12
    6.2. Measurement loop example . . . 12
    6.3. Measurement loop output . . . 13
7. Variable types . . . 15
8. Script argument types . . . 16
    8.1. *var* . . . 16
    8.2. *array* . . . 16
        8.2.1. Array Access Syntax . . . 17
    8.3. literal . . . 18
    8.4. *VarType* . . . 18
    8.5. integer types (*uint8*, *uint16*, *uint32*) . . . 18
    8.6. condition expressions . . . 19
    8.7. string . . . 20
        8.7.1. Interpolated strings . . . 20
    8.8. Optional arguments . . . 21
9. Optional arguments . . . 22
    9.1. poly_we . . . 22
    9.2. add_meas . . . 23
    9.3. nscans . . . 24
    9.4. nscans_avg . . . 25
    9.5. nscans_equil . . . 26
    9.6. meta_msk . . . 27
    9.7. eis_tdd . . . 27

9.8. eis_opt . . . 29
9.9. eis_acdc . . . 30
9.10. eis_dual_tdd . . . 31
9.11. eis_dual_acdc . . . 32
9.12. ms_eis_acdc . . . 33
9.13. window . . . 34
9.14. filter_type . . . 35
9.15. ocp . . . 35
9.16. qr_log . . . 36
10. Tags . . . 38
10.1. on_finished: . . . 38
11. Error handling . . . 40
12. PGStat modes . . . 41
12.1. PGStat mode off . . . 41
12.2. PGStat mode low speed . . . 41
12.3. PGStat mode high speed . . . 41
12.4. PGStat mode max range . . . 41
12.5. PGStat mode poly_we (deprecated) . . . 41
12.6. PGStat mode galvanostatic . . . 41
13. Script command summary . . . 42
13.1. Command summary . . . 42
13.2. MethodSCRIPT version on instruments . . . 49
14. Script command description . . . 50
14.1. Creating and manipulating variables . . . 50
14.1.1. var . . . 50
14.1.2. store_var . . . 50
14.1.3. copy_var . . . 51
14.2. Using arrays . . . 51
14.2.1. array . . . 51
14.2.2. array_set (deprecated) . . . 52
14.2.3. array_get (deprecated) . . . 53
14.2.4. subarray . . . 54
14.3. Mathematical operations . . . 55
14.3.1. add_var . . . 55
14.3.2. sub_var . . . 55
14.3.3. mul_var . . . 56
14.3.4. div_var . . . 56
14.3.5. mod_var . . . 57
14.3.6. pow_var . . . 58
14.3.7. log_var . . . 58
14.4. Logical operations . . . 59
14.4.1. bit_and_var . . . 59

PalmSens

14.4.2. bit_or_var . . . 60
14.4.3. bit_xor_var . . . 60
14.4.4. bit_lsl_var . . . 61
14.4.5. bit_lsr_var . . . 61
14.4.6. bit_inv_var . . . 62
14.5. Data type conversions . . . 62
14.5.1. int_to_float . . . 62
14.5.2. float_to_int . . . 63
14.5.3. alter_vartype . . . 63
14.6. Time, synchronization and hibernate . . . 64
14.6.1. rtc_get . . . 64
14.6.2. abort . . . 65
14.6.3. hibernate . . . 65
14.6.4. wait . . . 68
14.6.5. set_int . . . 69
14.6.6. await_int . . . 69
14.6.7. get_time . . . 70
14.6.8. timer_start . . . 71
14.6.9. timer_get . . . 71
14.6.10. set_channel_sync . . . 72
14.7. Conditional operations . . . 73
14.7.1. if, elseif, else, endif . . . 73
14.8. Loop constructs . . . 73
14.8.1. loop . . . 73
14.8.2. endloop . . . 74
14.8.3. breakloop . . . 75
14.9. Cell . . . 75
14.9.1. set_e . . . 75
14.9.2. set_i . . . 75
14.9.3. cell_on . . . 76
14.9.4. cell_off . . . 76
14.9.5. set_e_aux . . . 77
14.10. Measuring . . . 77
14.10.1. meas . . . 77
14.10.2. meas_ms_eis . . . 78
14.10.3. meas_fast_cv . . . 80
14.10.4. meas_fast_ca . . . 83
14.10.5. meas_scp . . . 84
14.11. Measurement loops . . . 86
14.11.1. set_scan_dir . . . 86
14.11.2. meas_loop_lsv . . . 86
14.11.3. meas_loop_acv . . . 88

14.11.4. meas_loop_lsp . . . 89
14.11.5. meas_loop_cv . . . 90
14.11.6. meas_loop_dpv . . . 91
14.11.7. meas_loop_swv . . . 93
14.11.8. meas_loop_npv . . . 94
14.11.9. meas_loop_ca . . . 95
14.11.10. meas_loop_ca_alt_mux . . . 96
14.11.11. meas_loop_cp . . . 98
14.11.12. meas_loop_cp_alt_mux . . . 99
14.11.13. meas_loop_pad . . . 101
14.11.14. meas_loop_ocp . . . 102
14.11.15. meas_loop_ocp_alt_mux . . . 103
14.11.16. meas_loop_eis . . . 104
14.11.17. meas_loop_eis_dual . . . 106
14.11.18. meas_loop_geis . . . 108
14.12. Script output . . . 109
14.12.1. pck_start . . . 109
14.12.2. pck_add . . . 110
14.12.3. pck_end . . . 110
14.12.4. file_open . . . 111
14.12.5. file_close . . . 111
14.12.6. set_script_output . . . 112
14.12.7. send_string . . . 112
14.13. Ranging . . . 113
14.13.1. set_pot_range (deprecated) . . . 113
14.13.2. set_cr (deprecated) . . . 114
14.13.3. set_range . . . 114
14.13.4. set_range_minmax . . . 116
14.13.5. set_autoranging . . . 117
14.13.6. trim_enable . . . 118
14.14. PGStat . . . 119
14.14.1. set_acquisition_frac . . . 119
14.14.2. set_acquisition_frac_autoadjust . . . 120
14.14.3. set_ir_comp . . . 121
14.14.4. set_pgstat_chan . . . 122
14.14.5. set_poly_we_mode (deprecated) . . . 122
14.14.6. set_pgstat_mode . . . 123
14.14.7. set_bipot_mode . . . 123
14.14.8. set_bipot_potential . . . 124
14.14.9. set_max_bandwidth . . . 124
14.15. GPIO . . . 125
14.15.1. set_gpio_cfg . . . 125

PalmSens

14.15.2. set_gpio_pullup . . . 126
14.15.3. set_gpio . . . 126
14.15.4. get_gpio . . . 127
14.15.5. set_gpio_msk . . . 127
14.15.6. get_gpio_msk . . . 128
14.16. I2C . . . 129
14.16.1. i2c_config . . . 129
14.16.2. i2c_write_byte . . . 129
14.16.3. i2c_read_byte . . . 130
14.16.4. i2c_write . . . 131
14.16.5. i2c_read . . . 132
14.16.6. i2c_write_read . . . 133
14.17. Multiplexers . . . 134
14.17.1. mux_config . . . 134
14.17.2. mux_get_channel_count . . . 135
14.17.3. mux_set_channel . . . 136
14.18. Misc . . . 136
14.18.1. notify_led . . . 136
14.18.2. smooth . . . 138
14.18.3. peak_detect . . . 139
14.18.4. beep . . . 140
14.18.5. battery_perc . . . 141
14.18.6. get_progress . . . 141
14.18.7. linear_fit . . . 142
14.18.8. mean . . . 143
14.18.9. qr_scan . . . 144
14.19. Display . . . 145
14.19.1. display_draw . . . 145
14.19.2. display_clear . . . 146
14.19.3. display_text . . . 146
14.19.4. display_icon . . . 146
14.19.5. display_progress . . . 147
14.19.6. display_btns . . . 148
14.19.7. display_inp_num . . . 149
14.19.8. display_scroll_add . . . 149
14.19.9. display_scroll_get . . . 150
14.19.10. display_keyboard . . . 151
15. MethodSCRIPT examples . . . 152
15.1. EIS example . . . 152
15.2. LSV example . . . 152
15.3. SWV example . . . 154
15.4. Fast CV example . . . 155

15.5. Fast CA example . . . 157
15.6. SCP example . . . 158
15.7. I²C example — temperature sensor . . . 160
15.8. I²C example — real time clock . . . 162
15.9. I²C example — EEPROM . . . 164
16. Document version changes . . . 167
Version 1.1 Rev 1 . . . 167
Version 1.1 Rev 2 . . . 167
Version 1.1 Rev 3 . . . 167
Version 1.1 Rev 4 . . . 167
Version 1.2 Rev 1 . . . 167
Version 1.2 Rev 2 . . . 168
Version 1.3 Rev 1 . . . 168
Version 1.4 Rev 1 . . . 169
Version 1.5 Rev 1 . . . 170
Version 1.6 Rev 1 . . . 171
Version 1.6 Rev 2 . . . 171
Version 1.7 Rev 1 . . . 171
Version 1.8 Rev 1 . . . 172
Appendix A: Error codes . . . 173
Appendix B: Device-specific information . . . 181
B.1. PGStat mode properties . . . 181
B.1.1. Nexus . . . 181
B.1.2. EmStat4 HR . . . 181
B.1.3. EmStat4 LR . . . 182
B.1.4. EmStat Pico . . . 182
B.1.5. Sensit Wearable . . . 183
B.2. EIS properties . . . 183
B.3. Current ranges . . . 184
B.3.1. Nexus . . . 184
B.3.2. EmStat4 LR . . . 185
B.3.3. EmStat4 HR . . . 186
B.3.4. EmStat Pico . . . 187
B.3.5. Sensit Wearable . . . 188
B.4. Potential ranges . . . 189
B.5. Supported variable types for `meas` command . . . 190
B.6. Device I/O pin configurations . . . 192
B.7. Measurement channels . . . 193
Appendix C: Variable types . . . 194

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)
  - Fast Chronoamperometry (FCA)
  - 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)
  - Multi-Sine Electrochemical Impedance Spectroscopy (MSEIS)
  - AC Voltammetry (ACV)
  - Fast Cyclic Voltammetry (FCV)
  - Stripping Chronopotentiometry (SCP)
- 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 50 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. Supported devices

- EmStat4
- EmStat Pico
- Sensit Wearable
- Nexus

[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 and Sensit Wearable.

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 256 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
  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`.

PalmSens

```
e\n
# This is a comment\n
wait 100m  # Comments can also follow other text\n
if 1 < 2\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}
`u`	micro	10^{-6}
`m`	milli	10^{-3}
(space)	*(none)*	10^{0}
`k`	kilo	10^{3}
`M`	mega	10^{6}
`G`	giga	10^{9}
`T`	tera	10^{12}
`P`	peta	10^{15}
`E`	exa	10^{18}

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.

4.3.1. Invalid Values

In the case that a value cannot be validly formatted, a `nan` will be returned instead. Such a value is formatted as 5 spaces followed by the text `nan` as follows:

```
     nan
```

This value will be returned in the following cases:

- A float's value is `NaN`
- A float's value is not finite
- An integer is out of the printable range (`-0x8000000` to `0x7FFFFFF`)

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 (<4% of max ADC value) 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. The value is defined as `15 + round(2 * log2(std / range))`. Where std is the standard deviation of the measured samples, and range is the value of the reported measurement range. Example: if the standard deviation is 16 µA, and the range is 1 mA, the noise level is: `15 + round(2 * log2(16e-6 / 1e-3)) = 3`.

The `Overload` flag is also affected by noise, and therefore may be set before the measured value reached the overload value.

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`.

PalmSens

- 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 potential
var current
# 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 potential current 100m 200m 1000m
   # The following commands are executed after each iteration (measurement).
   pck_start          # Start a new data packet.
   pck_add potential  # Add the 'potential' variable to the packet.
   pck_add current    # Add the 'current' variable 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

PalmSens

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 (LSV)
`0001`	Differential Pulse Voltammetry (DPV)
`0002`	Square Wave Voltammetry (SWV)
`0003`	Normal Pulse Voltammetry (NPV)
`0004`	AC Voltammetry (ACV)
`0005`	Cyclic Voltammetry (CV)
`0006`	Chronopotentiometric Stripping
`0007`	Chronoamperometry (CA)
`0008`	Pulsed Amperometric Detection (PAD)
`0009`	Fast Chronoamperometry (FCA)
`000A`	Chronopotentiometry (CP)
`000B`	Open Circuit Potentiometry (OCP)
`000D`	Electrochemical Impedance Spectroscopy (EIS)
`000E`	Galvanostatic Electrochemical Impedance Spectroscopy (GEIS)
`000F`	Linear Sweep Potentiometry (LSP)
`0010`	Fast Cyclic Voltammetry (FCV)
`0011`	Chronoamperometry (CA) with alternating mux
`0012`	Chronopotentiometry (CP) with alternating mux
`0013`	Open circuit potentiometry (OCP) with alternating mux
`0014`	Dual electrochemical impedance spectroscopy (Dual EIS)

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. Before a variable can be used, it first has to be declared to tell the instrument to reserve some memory. This can be done using the `var` command (see Section 14.1.1, “var”). Variable names must start with a lower case letter ('a' - 'z') and can for the rest consist of more lower case letters, numbers or underscores '_'.

For example, this allocates a few variables:

```
var a
var aa
var variable_3
var some_descriptive_name
```

The variable names are translated in the parsing stage so that their length or the amount of variables does not affect runtime. When choosing variable names, take the following into account: - The parser can only remember ~300 characters for all variable names combined. - Lines have a maximum length of 256 characters, this can be important for commands with multiple parameters.

There maximum amount of variables is 50. Variables are preserved during hibernation and exist for the duration of the script.

```
# Allocate variable with name my_number
var my_number
# Store PI in it
store_var my_number 3141m ja
# Send the content of var my_number to the user
pck_start
   pck_add my_number
pck_end
```

8.2. *array*

For storing more than one element, *arrays* can be used. This can be used with for example I²C data, fast techniques or generic measurements. Like variables, arrays have to be defined before they can be used (see Section 14.2.1, “array”). Interaction with arrays happens via their reference (just like variables). Arrays and variables denote distinct types, and cannot generally be substituted for one another in command arguments.

An example of defining an array, filling it with (squared) numbers and printing the content:

```
var temp
var i
store_var i 0i ja
# Define our array with size 10
array list_of_numbers 10
```

```
#
# Fill the array
#
loop i < 10
   copy_var i temp
   mul_var temp temp
   copy_var temp list_of_numbers[i]
   add_var i 1i
endloop
#
# Print the content to the user
#
store_var i 0i ja
loop i < 10
   copy_var list_of_numbers[i] temp
   pck_start
      pck_add i
      pck_add temp
   pck_end
   add_var i 1i
endloop
```

Table 6. Total storage for array elements

Instrument	Max array elements
EmStat Pico	4096
Sensit Wearable	4096
EmStat4	50000
Nexus	50000

On the EmStat Pico and Sensit Wearable, the contents of arrays are not preserved during hibernate, so they may contain random data afterwards.

8.2.1. Array Access Syntax

Array elements may be accessed with square bracket notation. Elements are zero-indexed, and the value used to index the array must be an integer (either an integer literal or an integer variable). Array accesses may not be nested - i.e. the index may not also be an array element.

An array element accessed in this way may be used in lieu of ordinary variables in command arguments, wherever a variable would be accepted.

```
array a 100 # Make the 100-element array "a"
# ...
# The initialisation of "a"s values is omitted
# ...
## Allowed:
```

```
# The 11th element will be used as the argument.
set_e a[10i]
#
# The 12th element will be used as the argument.
var x
store_var x 11i aa
set_e a[x]
#
# Write out all the items in the array
var i
store_var i 0i aa
loop i < 100i
  send_string f"{a[i]}"
  add_var i 1i
endloop
#
## Not allowed:
# "set_e" takes a variable, not an array.
set_e a
#
# Omitting the "i" after "10" makes it a float,
# but indices must be integers.
set_e a[10]
#
# Array accesses may not be nested.
set_e a[a[0i]]
```

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.

8.7.1. Interpolated strings

MethodSCRIPT does support limited string interpolation, allowing the values of variables to be included within a

PalmSens

string.

Interpolated strings are denoted by the letter `f` immediately preceding the opening quotation mark. Variables that are to be included in the string are surrounded by curly braces, e.g. `{varname}`. Curly braces that do not contain a valid variable name will cause an error.

The following example demonstrates how to print the value of a MethodSCRIPT variable:

```
var x
store_var x 10i ja
send_string f"x = {x}"
```

This will print the string `x = 10`.

A backslash (`\`) may be used to escape the following character, ensuring that it is included verbatim. The backslash itself will not be included in the output string.

Modifying the example above:

```
var x
store_var x 10i ja
send_string f"x = \{x}"
```

This will print the string `x = {x}` since the backslash escaped the opening curly brace, causing it to be included as-is rather than being interpolated.

If a backslash is required in the outputted string, write it as a double backslash (`\\`). The first backslash will escape the second, causing it to appear verbatim in the output:

```
var x
store_var x 10i ja
send_string f"x = {x} and then a backslash \\"
```

This will print the string `x = 10 and then a backslash \`.

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.

PalmSens

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

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, Nexus

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
```

The optional argument `poly_we` has been deprecated and may be removed in future releases. Use the optional argument `add_meas` instead. The argument `poly_we(1 b)` is equal to `add_meas(1 ba b)` and similar to `add_meas(0 bb b)` (the latter gives the same measurement result but with a different *VarType*).

9.2. add_meas

MethodSCRIPT	≥1.6
Supported instruments	EmStat Pico, Sensit Wearable, Nexus

Add an extra measurement to the command. This optional argument can be used multiple times for the same command. Depending on the instrument, different signals can be measured in parallel see Section B.7, “Measurement channels”

Arguments

Name	Type	Description
Channel	*uint8*	PGstat channel to use.
Var type	*VarType*	The type of variable to measure, see Chapter 7, *Variable types*.
Output variable	*var / array* [out] (*float*)	Output variable to store the measured data in. Must be an array if the loop outputs array data, such as for fast techniques.

This example shows measuring the potential (which can deviate from the set potential), and the temperature in a CV measurement loop.

```
var p
var c
var m
var t
meas_loop_cv p c 0 -500m 500m 10m 1 add_meas(0 ab m) add_meas(0 ed t)
pck_start
pck_add p
pck_add c
```

```
pck_add m
pck_add t
pck_end
endloop
```

This example shows measuring the bipot current in a CV measurement loop.

```
array p 41
array c 41
array b 41
var n
meas_fast_cv p c n 0 100m -100m 10m 1 add_meas(0 bb b)
```

9.3. nscans

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 `Cnnnn` where `nnnn` is an integer ranging from `0000` to `9999`. 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 ≤ n ≤ 9999).

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
-
*
```

9.4. nscans_avg

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

```
var c
var x
array p 5
array i 5
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
pck_add p[x]
# Add measured current to packet
pck_add i[x]
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
+
```

9.5. nscans_equil

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

PalmSens

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)
```

9.6. meta_msk

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

9.7. eis_tdd

MethodSCRIPT	≥1.3

PalmSens

Supported instruments	EmStat4, Nexus

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

This is not supported on the EmStat Pico and Sensit Wearable.

Arguments

Name	Type	Description
Potential signal TDD	*array* [out]	The acquired time-domain data of the potential signal of one EIS iteration or MSEIS measurement. 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 performs an EIS measurement and sends 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
```


PalmSens

```
loop i < n
pck_start
pck_add u[i]
pck_add c[i]
pck_end
add_var i 1i
endloop
endloop
on_finished:
cell_off
```

9.8. eis_opt

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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. Setting the value below 1 ms will enable Fast EIS. Fast EIS enables performing EIS measurements as fast as possible, at the cost of accuracy. At frequencies of 10 kHz and above, the interval time is less than 1 ms.
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
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_end
endloop
on_finished:
cell_off
```

9.9. eis_acdc

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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
```

9.10. eis_dual_tdd

MethodSCRIPT	≥1.7
Supported instruments	Nexus

The `eis_dual_tdd` optional parameter enables the transfer of time-domain data (TDD) for an EIS dual measurement. It it similar to the eis_tdd optional parameter.

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.
Third signal TDD	*array* [out]	The acquired time-domain data of the third signal of one EIS iteration. Depending on the mode, this is the data of the BiPot, or second sense. Minimum size required is 4096.
Number of samples	*var* [out]	The number of acquired data points (samples) for all signals.
Sampling frequency	*var* [out]	The frequency at which the data points are acquired for all signals.
Averaging mode	*uint16*	Averaging mode. Future option, default = 0.

The following example shows the usage for an EIS dual measurement. A more complete example can be found in eis_tdd.

```
array p 4096
array c 4096
array b 4096
meas_loop_eis_dual 1 f z_r z_i b_r b_i 50m 200k 1 11 0 eis_dual_tdd(p c b n fs 0)
store_var i 0i ja
loop i < n
pck_start
pck_add p[i]
pck_add c[i]
pck_add b[i]
pck_end
add_var i 1i
endloop
endloop
```

9.11. eis_dual_acdc

MethodSCRIPT	≥1.7
Supported instruments	Nexus

The `eis_dual_acdc` optional parameter returns the AC and DC information of the 3 measured signals. It it similar to the eis_acdc optional parameter.

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).
3_AC	*var* [out] (*float*)	AC of third signal: BiPot, or second sense (depends on the mode).
3_DC	*var* [out] (*float*)	DC of third signal: BiPot, or second sense (depends on the mode).

The following example shows the usage for an EIS dual measurement. A more complete example can be found in eis_acdc.

```
meas_loop_eis_dual 1 f z_r z_i b_r b_i 50m 200k 1 11 0 eis_dual_acdc(e_ac e_dc i_ac i_dc
b_ac b_dc)
```

9.12. ms_eis_acdc

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

The `ms_eis_acdc` optional parameter returns the AC and DC information for the E and I signal of a MSEIS measurement. The user should make sure that the E_AC and I_AC argument are an array of sufficient length.

Table 8. Arguments

Name	Type	Description
E_AC	*array* [out] (*float*)	E signal AC value for each harmonic in volts
E_DC	*var* [out] (*float*)	E signal DC value in volts
I_AC	*array* [out] (*float*)	I signal AC value for each harmonic in amperes
I_DC	*var* [out] (*float*)	I signal DC value in amperes

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

```
array f 15
array r 15
array j 15
var i
var n
var s
array u 15
array c 15
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_ms_eis f r j 10m 10 180m 2 ms_eis_acdc(u n c s)
# First send the MSEIS results
store_var i 0i ja
loop i < 15i
```


PalmSens

```
pck_start
pck_add f[i]
pck_add r[i]
pck_add j[i]
pck_end
add_var i 1i
endloop
# Send AC voltage and current data for each harmonic
store_var i 0i ja
loop i < 15i
pck_start
pck_add u[i]
pck_add c[i]
pck_end
add_var i 1i
endloop
# Send the DC voltage and current data
pck_start
pck_add n
pck_add s
pck_end
cell_off
```

9.13. window

MethodSCRIPT	≥1.6
Supported instruments	Sensit Wearable, EmStat4, Nexus

Provide a window for peak detection.

Only peaks whose highest point is in the window (bounds inclusive) will be reported.

Arguments

Name	Type	Description
Left index	*var/ literal* (*int*)	The index of the leftmost element that may register as a peak
Right index	*var/ literal* (*int*)	The index of the rightmost element that may register as a peak

```
array indices 2
array heights 2
peak_detect data indices heights 0i 10u window(0i 5i)
```

9.14. filter_type

MethodSCRIPT	≥1.7
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Specify a filter type.

Sets the bandwidth of only the selected filter type.

Using `set_max_bandwidth` without this optional argument, will overwrite the bandwidth of all filter types.

Arguments

Name	Type	Description
Filter type	*uint32*	Select from the following list 1: Setpoint filter. Generates the setpoint for the set potential (or current in galvanostatic mode). 2: Control loop filter. Controls the CE so that the RE vs S (or current in galvanostatic mode) match the setpoint. 3: Current to voltage filter. Converts WE current to a voltage. This voltage is used both for galvanostatic control loop feedback, as well as measuring the current. 4: Measurement filter. Measures various voltages (including the converted WE current). 5: Second current to voltage filter. Converts WE2 current to a voltage. 6: iR compensation filter. Adds a portion of the converted WE current to the control loop feedback.

The following example sets the bandwidth of all filters to 10 kHz, and then sets the iR compensation filter to 100 kHz.

Example

```
set_max_bandwidth 10k
set_max_bandwidth 100k filter_type(6)
```

9.15. ocp

MethodSCRIPT	≥1.7
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Specify an open circuit potential.

Used by the `cell_on` command to set the initial voltage during the `cell_on` command in Galvanostatic mode. If the initial voltage is set to the OCP voltage, the initial applied current will settle from 0A to the requested current, preventing currents higher than the requested current.

Arguments

Name	Type	Description
Filter type	*var / literal (float)*	The open circuit potential in Volts.

The following example measures the OCP and uses it to reduce the initial applied current spike after the `cell_on` command.

Example

```
var p
set_pgstat_mode 6
set_range ab 1
set_range db 1m
set_i 1m
meas 100m p ab
cell_on ocp(p)
```

9.16. qr_log

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

When a QR code is successfully scanned, record its contents to the script output.

Any characters outside the set of visible Ascii characters will be written as `\XX`, where `XX` is the hexadecimal value of the character byte.

The backslash character itself will be written as a double backslash.

For example the following text:

```
Hello
Backslash\
```

Would appear in the output as:

`Hello\0ABackslash\\`

Arguments

Name	Type	Description

```
var count
array parsed_variables 5
qr_scan parsed_variables count  qr_log()
```

```
# When scanned, will write the barcode text to the log
```

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 (deprecated)

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.

> PGStat mode poly_we has been deprecated and may be removed in future releases. Instead, configure a channel for poly_we mode using the command `set_bipot_mode`.

12.6. PGStat mode galvanostatic

This mode is used to control the applied current, rather than the applied potential. This mode is required for all galvanostatic techniques and commands.

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 9. MethodSCRIPT command summary

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
Creating and manipulating variables						
var	≥1.1	Y	Y	Y	Y	Declare a variable.
store_var	≥1.1	Y	Y	Y	Y	Store a value in a variable.
copy_var	≥1.1	Y	Y	Y	Y	Copy a variable.
Using arrays						
array	≥1.2	Y	Y	Y	Y	Declare an array.
array_set (deprecated)	≥1.2	Y	Y	Y	Y	Set a variable at the specified array index.
array_get (deprecated)	≥1.2	Y	Y	Y	Y	Get a variable from the specified array index.
subarray	≥1.8	Y	Y	Y	Y	Declare an array that is a view into an existing array.
Mathematical operations						
add_var	≥1.1	Y	Y	Y	Y	Add a value to a variable.
sub_var	≥1.1	Y	Y	Y	Y	Subtract a value from a variable.
mul_var	≥1.1	Y	Y	Y	Y	Multiply a variable.
div_var	≥1.1	Y	Y	Y	Y	Divide a variable.
mod_var	≥1.5	Y	Y	Y	Y	Perform a modulo operation on a variable.
pow_var	≥1.7	Y	Y	Y	Y	Raise a variable to a power.
log_var	≥1.8	Y	Y	Y	Y	Take the natural logarithm of a variable.
Logical operations						
bit_and_var	≥1.3	Y	Y	Y	Y	Perform a bitwise AND operation.
bit_or_var	≥1.3	Y	Y	Y	Y	Perform a bitwise OR operation.
bit_xor_var	≥1.3	Y	Y	Y	Y	Perform a bitwise XOR operation
bit_lsl_var	≥1.3	Y	Y	Y	Y	Logical Shift Left variable.

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
bit_lsr_var	≥1.3	Y	Y	Y	Y	Logical Shift Right variable.
bit_inv_var	≥1.3	Y	Y	Y	Y	Bitwise invert a variable.
Data type conversions						
int_to_float	≥1.3	Y	Y	Y	Y	Change the data type from *int* to *float*.
float_to_int	≥1.3	Y	Y	Y	Y	Change the data type from *float* to *int*.
alter_vartype	≥1.5	Y	Y	Y	Y	Alter the *VarType* of a variable.
Time, synchronization and hibernate						
rtc_get	≥1.6	Y	Y	Y	Y	Read the current date and time from the real-time clock.
abort	≥1.2	Y	Y	Y	Y	Abort the current script.
hibernate	≥1.2	Y	Y	Y	N	Put the device in hibernate mode.
wait	≥1.1	Y	Y	Y	Y	Wait for the specified amount of time.
set_int	≥1.2	Y	Y	Y	Y	Configure the interval for the `await_int` command.
await_int	≥1.2	Y	Y	Y	Y	Wait for the next interval.
get_time	≥1.2	Y	Y	Y	Y	Get the time since device startup in seconds.
timer_start	≥1.2	Y	Y	Y	Y	Start the timer.
timer_get	≥1.2	Y	Y	Y	Y	Get the timer value.
set_channel_sync	≥1.3	N	N	Y	Y	Enable or disable channel synchronization.
Conditional operations						
if, elseif, else, endif	≥1.2	Y	Y	Y	Y	Conditional statements allow the conditional execution of commands.
Loop constructs						
loop	≥1.1	Y	Y	Y	Y	Repeat a block of commands while some condition is fullfilled.
endloop	≥1.1	Y	Y	Y	Y	Signal the end of a loop.
breakloop	≥1.2	Y	Y	Y	Y	Break out of the current loop.
Cell						
set_e	≥1.1	Y	Y	Y	Y	Apply a variable or literal as the WE potential.
set_i	≥1.3	N	N	Y	Y	Apply a variable or literal as the WE current in galvanostatic mode.

PalmSens

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
cell_on	≥1.3	Y	Y	Y	Y	Turn the cell on. This enables the WE potential or current regulation.
cell_off	≥1.3	Y	Y	Y	Y	Turn the cell off.
set_e_aux	≥1.4	N	N	Y	Y	Set the voltage on the AUX DAC.
Measuring						
meas	≥1.1	Y	Y	Y	Y	Measure a data point of the specified type and store the result as a variable.
meas_ms_eis	≥1.5	N	N	Y	Y	Perform a Multi-Sine EIS (MSEIS) measurement.
meas_fast_cv	≥1.4	N	N	Y	Y	Perform a Fast Cyclic Voltammetry (FCV) measurement.
meas_fast_ca	≥1.5	N	N	Y	Y	Perform a Fast Chronoamperometry (FCA) measurement.
meas_scp	≥1.8	N	N	N	Y	Perform a Stripping Chronopotentiometry (SCP) measurement.
Measurement loops						
set_scan_dir	≥1.5	Y	Y	Y	Y	Reverse the direction of the CV scan.
meas_loop_lsv	≥1.1	Y	Y	Y	Y	Perform a Linear Sweep Voltammetry (LSV) measurement.
meas_loop_acv	≥1.5	N	N	Y	Y	Perform a AC Voltammetry (ACV) measurement.
meas_loop_lsp	≥1.3	N	N	Y	Y	Perform a Linear Sweep Potentiometry (LSP) measurement.
meas_loop_cv	≥1.1	Y	Y	Y	Y	Perform a Cyclic Voltammetry (CV) measurement.
meas_loop_dpv	≥1.1	Y	Y	Y	Y	Perform a Differential Pulse Voltammetry (DPV) measurement.
meas_loop_swv	≥1.1	Y	Y	Y	Y	Perform a Square Wave Voltammetry (SWV) measurement.
meas_loop_npv	≥1.1	Y	Y	Y	Y	Perform a Normal Pulse Voltammetry (NPV) measurement.
meas_loop_ca	≥1.1	Y	Y	Y	Y	Perform a Chronoamperometry (CA) measurement.
meas_loop_ca_alt_mux	≥1.5	N	N	Y	Y	Perform a Chronoamperometry (CA) measurement in alternating multiplexer mode.
meas_loop_cp	≥1.3	N	N	Y	Y	Perform a Chronopotentiometry (CP) measurement.

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
meas_loop_cp_alt_mux	≥1.5	N	N	Y	Y	Perform a Chronopotentiometry (CP) measurement in alternating multiplexer mode.
meas_loop_pad	≥1.1	Y	Y	Y	Y	Perform a Pulsed Amperometric Detection (PAD) measurement.
meas_loop_ocp	≥1.1	Y	Y	Y	Y	Perform an Open Circuit Potentiometry (OCP) measurement.
meas_loop_ocp_alt_mux	≥1.5	N	N	Y	Y	Perform an Open Circuit Potentiometry (OCP) measurement in alternating multiplexer mode.
meas_loop_eis	≥1.1	Y	Y	Y	Y	Perform a (potentiostatic) Electrochemical Impedance Spectroscopy (EIS) measurement.
meas_loop_eis_dual	≥1.7	N	N	N	Y	Perform a dual (potentiostatic) Electrochemical Impedance Spectroscopy (EIS) measurement.
meas_loop_geis	≥1.3	N	N	Y	Y	Perform a Galvanostatic Electrochemical Impedance Spectroscopy (GEIS) measurement.
Script output						
pck_start	≥1.1	Y	Y	Y	Y	Start a measurement data packet.
pck_add	≥1.1	Y	Y	Y	Y	Add a variable (or literal) to the measurement data package previously started with `pck_start`.
pck_end	≥1.1	Y	Y	Y	Y	Send the measurement data package previously started with `pck_start`, containing all variables added using `pck_add`.
file_open	≥1.2	Y	Y	Y	Y	Open a file on the persistent storage.
file_close	≥1.2	Y	Y	Y	Y	Close the currently open file.
set_script_output	≥1.2	Y	Y	Y	Y	Set the output mode for the script.
send_string	≥1.1	Y	Y	Y	Y	Send an arbitrary string as output of the MethodSCRIPT.
Ranging						
set_pot_range (deprecated)	≥1.2	Y	Y	Y	Y	Set the expected potential range for the following measurements.
set_cr (deprecated)	≥1.1	Y	Y	Y	Y	Set the current range for the given maximum current.
set_range	≥1.3	Y	Y	Y	Y	Set the expected maximum absolute current or potential for a given *VarType*.

PalmSens

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
set_range_minmax	≥1.3	Y	Y	Y	Y	Set the expected minimum and maximum current or potential for a given *VarType*.
set_autoranging	≥1.1	Y	Y	Y	Y	Configure the autoranging for all `meas_loop_*` commands.
trim_enable	≥1.8	Y	Y	Y	Y	Enable or disable trimming for a given *VarType*.
PGStat						
set_acquisition_frac	≥1.3	Y	Y	Y	Y	Set the fraction of the iteration time to use for measurement.
set_acquisition_frac_auto adjust	≥1.4	N	N	Y	Y	Filter out the given frequency by automatically adjusting acquisition times.
set_ir_comp	≥1.5	N	N	Y	Y	Set resistance to be compensated by iR compensation.
set_pgstat_chan	≥1.1	Y	Y	Y	Y	Select a PGStat channel.
set_poly_we_mode (deprecated)	≥1.1	Y	Y	N	N	Select the mode of the additional working electrode.
set_pgstat_mode	≥1.1	Y	Y	Y	Y	Set the PGStat hardware configuration to be used for measurements.
set_bipot_mode	≥1.7	Y	Y	N	Y	Set the mode of the second working electrode.
set_bipot_potential	≥1.7	Y	Y	N	Y	Set the potential (offset) of the second working electrode.
set_max_bandwidth	≥1.1	Y	Y	Y	Y	Set maximum bandwidth of the signal being measured.
GPIO						
set_gpio_cfg	≥1.2	Y	Y	Y	Y	Set the GPIO pin configuration.
set_gpio_pullup	≥1.2	Y	Y	Y	Y	Enable or disable GPIO pin pull-ups.
set_gpio	≥1.1	Y	Y	Y	Y	Set the GPIO output values.
get_gpio	≥1.2	Y	Y	Y	Y	Get the GPIO input pin values.
set_gpio_msk	≥1.4	Y	Y	Y	Y	Write to the GPIO pins indicated by the mask.
get_gpio_msk	≥1.4	Y	Y	Y	Y	Get the GPIO input pin values with a mask.
I2C						
i2c_config	≥1.2	Y	Y	Y	Y	Setup I²C configuration.
i2c_write_byte	≥1.2	Y	Y	Y	Y	Transmit one byte to an I²C target device.

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
i2c_read_byte	≥1.2	Y	Y	Y	Y	Receive one byte from an I²C target device.
i2c_write	≥1.2	Y	Y	Y	Y	Write one or more bytes to an I²C target device.
i2c_read	≥1.2	Y	Y	Y	Y	Read one or more bytes from an I²C target device.
i2c_write_read	≥1.2	Y	Y	Y	Y	Write to and read from an I²C target device.
Multiplexers						
mux_config	≥1.4	Y	N	Y	Y	Configure a multiplexer to use in MethodSCRIPT.
mux_get_channel_count	≥1.4	Y	N	Y	Y	Get the number of channels on the multiplexer setup.
mux_set_channel	≥1.4	Y	N	Y	Y	Select channel on the multiplexer.
Misc						
notify_led	≥1.5	Y	Y	Y	Y	Notify the user of a user-defined event, using the LED.
smooth	≥1.6	N	Y	Y	Y	Apply Savitzky-Golay smoothing to data in an array.
peak_detect	≥1.6	Y	Y	Y	Y	Find peaks in the given data.
beep	≥1.7	N	N	N	Y	Make a beep, and wait for it to be finished.
battery_perc	≥1.7	N	Y	N	N	Read the battery's charge as a percentage.
get_progress	≥1.7	Y	Y	Y	Y	Read the progress through the current measurement, from 0 to 100.
linear_fit	≥1.8	N	Y	Y	Y	Perform a linear least squares regression on a set of data.
mean	≥1.8	N	Y	Y	Y	Take the mean of an array of data.
qr_scan	≥1.8	N	N	N	N	Trigger the QR code scanner.
Display						
display_draw	≥1.8	N	N	N	N	Immediately prompt the display to be updated.
display_clear	≥1.8	N	N	N	N	Remove all elements from the display.
display_text	≥1.8	N	N	N	N	Add a new line of text to the display, to be shown the next time the display is drawn (see Section 14.19.1, "display_draw").
display_icon	≥1.8	N	N	N	N	Add an icon on the display, to be shown the next time the display is drawn (see Section 14.19.1, "display_draw").

MethodSCRIPT command	Version	EmStat Pico	Sensit Wearable	EmStat4	Nexus	Description
display_progress	≥1.8	N	N	N	N	Add a progress bar on the display, to be shown the next time the display is drawn (see Section 14.19.1, “display_draw”).
display_btns	≥1.8	N	N	N	N	Show one or two buttons on the display, then immediately update the display and wait for the user to press one.
display_inp_num	≥1.8	N	N	N	N	Prompt the user for a numerical value, and wait until one is provided.
display_scroll_add	≥1.8	N	N	N	N	Add an entry to the scroll list on the display, to be shown using Section 14.19.9, “display_scroll_get”.
display_scroll_get	≥1.8	N	N	N	N	Show the scroll items (added by Section 14.19.8, “display_scroll_add”) to the user, and wait for a choice to be made.
display_keyboard	≥1.8	N	N	N	N	Get a line of text entered by the user and record it to the script output.

PalmSens

13.2. MethodSCRIPT version on instruments

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

Table 10. MethodSCRIPT and instrument firmware versions

MethodSCRIPT	EmStat Pico	Sensit Wearable	EmStat4	Nexus
1.0	v1.0	-	-	-
1.1	v1.1	-	-	-
1.2	v1.2	-	v1.0	-
1.3	v1.3	-	v1.1	-
1.4	-	-	v1.2	-
1.5	-	-	v1.3	-
1.6	-	v1.4	-	-
1.7	v1.5	-	-	v1.0
1.8	v1.6	v1.6	1.5	v1.1

Chapter 14. Script command description

14.1. Creating and manipulating variables

14.1.1. var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Declare a variable. All MethodSCRIPT variables must be declared before use. When a variable is declared, it is initialized with the floating-point value 0 and *VarType* `aa`. For details on naming and limitations see Chapter 8, *Script argument types*.

Arguments

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

Example

Define two variables with names `foo` and `bar`

```
var foo
var bar
```

14.1.2. store_var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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*.

PalmSens

Example

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

```
store_var foo 200 ja
```

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

```
store_var foo 200i ja
```

14.1.3. copy_var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.2. Using arrays

14.2.1. array

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Declare an array. Arrays can store multiple variables. All arrays must be declared before use. The name may not be used by another array or variable. For details on naming and limitations see Chapter 8, *Script argument types*.

PalmSens

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. 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. Arrays use zero-based indexing, so the first element has index 0, the second element has index 1, and so on.

Arrays elements can be referenced using the `x[i]` style syntax described in Section 8.2.1, "Array Access Syntax".

Previous MethodScript releases advised the use of `array_set` and `array_get`, which have now been deprecated.

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

Arguments

Name	Type	Description
Variable name	*array*	Array reference.
Array size	*var* / *literal* (*int*)	The amount of variables this array can hold.

Example

Declare array with name `foo_bar_baz` and size 10.

```
array foo_bar_baz 10
```

Variables and arrays with the same name cannot exist in the same script.

14.2.2. array_set (deprecated)

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Set a variable at the specified array index.

The `array_set` command has been deprecated and may be removed in future releases. Use the `store_var` or `copy_var` commands instead, with `array access syntax`.

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 `foobar` with 6 elements, and writes the value 0.02 to the last element (the variable at index 5).

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

14.2.3. array_get (deprecated)

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Get a variable from the specified array index.

The `array_get` command has been deprecated and may be removed in future releases. Use the `copy_var` command instead, with `array access syntax`.

Arguments

Name	Type	Description
Array variable	*array*	Array reference.

Name	Type	Description
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
```

14.2.4. subarray

MethodSCRIPT	≥1.8
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Declare an array that is a view into an existing array. This does not allocate any new variables - it only allows access to existing data via a new variable binding.

You may encounter unexpected behaviour if the arrays passed to a MethodScript command point to the same underlying data.

Arguments

Name	Type	Description
Array variable	*array*	The new subarray to declare
Source array	*array*	The source array into which this subarray is a view
Array index	*var* / *literal* (*int*)	The start index in the source array, where the subarray starts
Length	*var* / *literal* (*int*)	The length of the subarray

Example

The following example shows how an array and a subarray pointing into it share the same underlying data.

```
# Declare a length 10 array called 'source'
array source 10i
# Declare a subarray which views elements 5 and 6 of 'source'
subarray view source 5i 2i
store_var source[5i] 3141m aa
```

```
store_var source[6i] 42i aa
send_string f"{view[0i]}, {view[1i]}"
# This logs "3.14" and "42", because view[0i] and view[1i]
# point to the same data as source[5i] and source[6i]
```

14.3. Mathematical operations

14.3.1. add_var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.3.2. sub_var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.3.3. mul_var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.3.4. div_var

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.3.5. mod_var

MethodSCRIPT	≥1.5
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Perform a modulo operation on a variable.

Calculate the remainder of dividing `arg1` by `arg2` and store the result in `arg1`. Both arguments must be integer variables. The *VarType* and metadata of the variable(s) are not changed.

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*)	The variable to be divided.
arg2	*var* / *literal* (*int*)	The value to divide by.

Example

Calculate the remainder of dividing the variable `a` by 4 and store the result in `a`.

```
mod_var a 4i
```

14.3.6. pow_var

MethodSCRIPT	≥1.7
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Raise a variable to a power.

Raise `arg1` to the power of `arg2` and store the result in `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.

> If a negative integer number is used for the exponent, an error will be raised. If mathematically invalid float parameters are used (such as 0^{-1} or $-1^{0.5}$) no error will be raised, but the result will be `NaN`

Arguments

Name	Type	Description
arg1	*var* [in/out] (*int*, *float*)	The variable to be updated and the base of the exponentiation.
arg2	*var / literal* (*int*, *float*)	The exponent.

Example

Take the square root of `x` and store the result in `x`.

```
pow_var x 500m
```

14.3.7. log_var

MethodSCRIPT	≥1.8
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Take the natural logarithm of a variable.

The value of `arg1` will be updated to be equal to its natural logarithm.

To convert to a logarithm with a base other than `e`, the standard transformation can be used:

`log_B(X) = \frac{log(X)}{log(B)}`

Nonpositive inputs will error

Arguments

Name	Type	Description
arg1	*var* [in/out] (*float*)	The variable to be updated

Example

Take the natural logarithm of x

```
log_var x
```

14.4. Logical operations

14.4.1. bit_and_var

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.4.2. bit_or_var

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.4.3. bit_xor_var

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.4.4. bit_lsl_var

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.4.5. bit_lsr_var

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.4.6. bit_inv_var

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.5. Data type conversions

14.5.1. int_to_float

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.5.2. float_to_int

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.5.3. alter_vartype

MethodSCRIPT	≥1.5
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Alter the *VarType* of a variable.

Arguments

Name	Type	Description
Variable	*var* [out] (*int*, *float*)	Variable reference.
Variable Type	*VarType*	The type identifier for this value, see Chapter 7, *Variable types*.

Example

Alter the type of variable `a` to `VT_MISC_GENERIC1`.

```
alter_vartype a ja
```

14.6. Time, synchronization and hibernate

14.6.1. rtc_get

MethodSCRIPT	≥1.6
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Read the current date and time from the real-time clock. This matches the behaviour of the `System date and time` register. On devices without a Real-Time Clock, this will return the system time relative to startup.

The following instruments support an external Real-Time Clock:

- The EmStat Pico does not have an RTC on the module, but does support the Ablic S-35390A RTC, which is incorporated in the the Sensit BT and on the EmStat Pico Development Kit. Support for it can be enabled in the `Peripheral configuration` register.
- The Sensit Wearable and EmStat4T incorporate an external RTC which is natively supported.

Arguments

Name	Type	Description
Year	*var* [out] (int)	The year, starting at 1 for 1AD
Month	*var* [out] (int)	The month, starting at 1 for January
Day	*var* [out] (int)	The day, starting at 1
Hour	*var* [out] (int)	The hour, 0 to 23
Minute	*var* [out] (int)	The minute, 0 to 59
Second	*var* [out] (int)	The second, 0 to 59

Example

Read and send out the current time

```
var yr
var mo
var dy
var hr
var mn
var sn
rtc_get yr mo dy hr mn sn
send_string f"{yr} {mo} {dy} {hr} {mn} {sn}"
```

14.6.2. abort

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

14.6.3. hibernate

MethodSCRIPT	≥1.2

Supported instruments	EmStat Pico, Sensit Wearable, EmStat4

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 and Sensit Wearable). 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 and Sensit Wearable, a high-to-low transition (falling edge) wakes up the instrument.
- **Timer:** The specified time has passed.
- **Double-tap:** A double-tap has been detected using the onboard accelerometer (Sensit Wearable only).

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

In MethodSCRIPT version 1.3 or lower, all channels settings were cleared, and channels were switched off in hibernate mode.

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

Since communication input is flushed during hibernation, it can be hard to abort scripts that have very little time between hibernations.

When automating the `hibernate` command on a MethodScript device, it is best to use a sync character - `\x16` - to wake the device from comms to ensure there is no reply. When using the serial interface manually, it is best to send a plain newline - `\n` - which may cause a newline to be echo'd from the device.

Arguments

Name	Type	Description
Wake-up source mask	*uint8*	Bitmask for wake-up sources: 0x01 = Communication 0x02 = WAKE pin 0x04 = Timer 0x08 = Double-tap (Sensit Wearable only) 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 14.15.1, “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 2
```

Supported PGStat modes

The EmStat Pico can only enter hibernation in PGStat modes "off (0)", "low speed (2)", and "poly_we (5)". In any other mode, an error (`0x0023`) will be thrown.

In modes low speed and poly_we mode, a potential can still be applied during hibernation. This will significantly increase power consumption by about 1 mA. For lowest power consumption, put both channels into PGStat mode off.

Prior to firmware version 1.4.00, the EmStat Pico would instead overwrite the PGStat mode to "off (0)".

Known limitations

- On the EmStat Pico, arrays are not preserved when a hibernate command is issued.
- The minimum hibernation time is 10 ms (125 ms in FW version 1.3 or lower). Error code `0x4205` will be thrown when the specified time value is too short.

Sensit Wearable

Wake pin

The WAKE pin can be activated by pushing on the top of the housing.

Double-tap

When enabled, a quick double tap can wake the device.

Double-tap detection is done by measuring acceleration. This means false positives can occur due to acceleration from other sources. A button press can also sometimes be interpreted as a double-tap, so it is not recommended to enable the double-tap and WAKE pin wakeup source at the same time, since this can cause WAKE pin events to be missed in rare cases.

BlueTooth

BlueTooth connections are maintained while hibernating. It is not possible the wake the device from BlueTooth. Received data will be handled when the device wakes up from other another wake source.

Supported PGStat modes

The Sensit Wearable can only enter hibernation in PGStat modes "off (0)", "low speed (2)", and "poly_we (5)". In any other mode, an error (`0x0023`) will be thrown.

In modes low speed and poly_we mode, a potential can still be applied during hibernation. This will significantly increase power consumption by about 1 mA. For lowest power consumption, put both channels into PGStat mode off.

Known limitations

- On the Sensit Wearable, arrays are not preserved when a hibernate command is issued.
- The minimum hibernation time is 10 ms. Error code `0x4205` will be thrown when the specified time value is too short.

EmStat4

The EmStat4 does not support deep-sleep in hardware, and so the `hibernate` command does not decrease power consumption.

For compatibility, the EmStat4 still accepts the `hibernate` command and will suspend MethodScript execution until the wakeup conditions is met.

Other, non-MethodScript functionality may remain responsive.

Nexus

The Nexus does not support the hibernate command

14.6.4. wait

MethodSCRIPT	≥1.1

PalmSens

Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.6.5. set_int

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.6.6. await_int

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.6.7. get_time

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

14.6.8. timer_start

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.6.9. timer_get

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 time
timer_start
# ...Do something interesting that takes a bit of time here...
timer_get time
pck_start
# Add a as a timestamp
pck_add time
# ...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.

14.6.10. set_channel_sync

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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.

When synchronization is enabled, the master will wait 100 ms before starting a measurement loop, to make sure the slave devices are ready to start.

Arguments

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

Example

```
# Enable syncing
```

```
set_channel_sync 1
```

14.7. Conditional operations

14.7.1. if, elseif, else, endif

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.8. Loop constructs

14.8.1. loop

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.8.2. endloop

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

-

PalmSens

14.8.3. breakloop

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

Arguments

-

14.9. Cell

14.9.1. set_e

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.9.2. set_i

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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
```

14.9.3. cell_on

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

-

Optional arguments

The following optional arguments are supported:

- `ocp`

Example

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

```
cell_on
```

14.9.4. cell_off

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.9.5. set_e_aux

MethodSCRIPT	≥1.4
Supported instruments	EmStat4, Nexus

Set the voltage on the AUX DAC.

Arguments

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

Example

```
set_e_aux a
```

14.10. Measuring

14.10.1. meas

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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*.

Optional arguments

The following optional arguments are supported:

- `add_meas`

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
```

14.10.2. meas_ms_eis

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

Perform a Multi-Sine EIS (MSEIS) measurement.

Multi-Sine EIS (MSEIS) can measure an impedance spectrum in less time then EIS at the cost of a reduced Signal-to-Noise Ratio (SNR). This command performs a potentiostatic multi-sine EIS measurement and stores the resulting frequencies, Z-real, and Z-imaginary in the given arrays.

The following commands currently have no effect on MSEIS measurements:

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

Arguments

Name	Type	Description
Applied frequencies	*array* [out] (*float*)	Output array to store the applied frequencies (Hz) of all harmonics.
Measured Z-real	*array* [out] (*float*)	Output array to store the real part of the measured complex impedances. This field also contains the meta-data of the I-signal (current)

Name	Type	Description
Measured Z-imaginary	*array* [out] (*float*)	Output array to store the imaginary part of the measured complex impedances. This field also contains the meta-data of the E-signal (potential)
Amplitude	*var / literal* (*float*)	Peak amplitude of the applied waveform in volt.
Base frequency	*var / literal* (*float*)	Base frequency of the applied waveform in Hz.
DC potential	*var / literal* (*float*)	DC potential offset of the applied waveform in volt.
Preset	*var / literal* (*int*)	Index of the waveform preset that should be used.

Optional arguments

`eis_tdd`

`eis_opt`

`ms_eis_acdc`

Presets

Depending on the expected impedance curve, a perturbation-preset can be chosen. A total of 6 presets are available with varying harmonics and amplitude distributions. Presets 1, 2, 4 and 5 feature a logarithmically decaying amplitude distribution, meaning that the base frequency has a relative amplitude of 1, and the highest included harmonic has a relative amplitude as specified in the table. The decrease of amplitude follows a logarithmic distribution, and can be benificial when the cell shows capacitive behavior.

	Flat	Logarithmic	
Multisine 5 (1-9x)	Preset 0	Preset 1 (min rel. amplitude = 0.7)	Preset 2 (min rel. amplitude = 0.3)
Multisine 15 (1-99x)	Preset 3	Preset 4 (min rel. amplitude = 0.5)	Preset 5 (min rel. amplitude = 0.1)

Example

Perform a MSEIS measurement using multisine preset 3 with 10 mV peak amplitude and 180 mV DC offset. The harmonic frequencies and complex impedances are stored in the arrays freqs, reals and imags. The user must ensure the supplied arrays are long enough to store the results of the chosen preset. When the measurement is done, the data is sent back point by point in a loop.

```
array freqs 15
array reals 15
array imags 15
var idx
```

```
meas_ms_eis freqs reals imags 10m 100m 180m 3
store_var idx 0i ja
loop idx < 15
pck_start
pck_add freqs[idx]
pck_add reals[idx]
pck_add imags[idx]
pck_end
add_var idx 1i
endloop
```

14.10.3. meas_fast_cv

MethodSCRIPT	≥1.4
Supported instruments	EmStat4, Nexus

Perform a Fast Cyclic Voltammetry (FCV) 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
Set potentials	*Array* [out] (*float*)	The array to store the set potentials in.
Measured currents	*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_COUNT` (`ee`).
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.

- `add_meas`
- `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 potentials 41
array currents 41
var npoints
meas_fast_cv potentials currents npoints 0 100m -100m 10m 1
```

Example 2: nscans

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

```
array potentials 205
array currents 205
var npoints
meas_fast_cv potentials currents npoints 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 potentials 41
array currents 41
var npoints
meas_fast_cv potentials currents npoints 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 `potentials`, `currents` and `npoints` variables is the same as if `nscans_avg` was not performed even though the values are averaged.

```
array potentials 41
array currents 41
var npoints
meas_fast_cv potentials currents npoints 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 potentials 123
array currents 123
var npoints
meas_fast_cv potentials currents npoints 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”.

14.10.4. meas_fast_ca

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

Perform a Fast Chronoamperometry (FCA) measurement.

This command is similar to the `meas_loop_ca` command, which is a measurement loop command. However, the fast measurement command is intended for short, (very) fast measurements with an accurate timing. The maximum data rate is 1 MS/s (1 million samples per second), using an interval time of 1 µs. Measurement points are averaged at maximum sample rate during the interval time, if possible. To achieve this, no other MethodSCRIPT commands can be performed during the measurement, and the results must be stored in an array. As a consequence, the number of data points to measure is limited to the maximum size of an array (50,000 on the EmStat4).

The `set_acquisition_frac` command does not apply for Fast CA measurements. Measurements are performed over the entire interval time.

Arguments

Name	Type	Description
Set potential	*var* [out] (*float*)	Variable to store the set potential in. This is a single value because the set potential is the same for all data points.
Measured currents	*array* [out] (*float*)	Array to store the measured currents in. The array must be large enough to store all data points. The number of data points is determined by the run time and interval time.
Points count	*var* [out] (*int*)	Variable to store the number of measurement points in. The *VarType* of the variable will be set to `VT_COUNT` (`ee`).
DC potential	*var / literal* (*float*)	The DC potential to set.
Interval time	*var / literal* (*float*)	The interval time (i.e. the time between measurements). The minimum interval time is 1 µs. The maximum interval time is 1 minute.
Run time	*var / literal* (*float*)	The total measurement time. This must be greater than or equal to the interval time.

Optional arguments

The following optional arguments are supported:

- `add_meas`

> On the Nexus, `add_meas` only supports channels 1, 2 and 3 for the `meas_fast_ca` command. See Section B.7, “Measurement channels” for information on what can be measured on these channels.

Example

The following example performs a Fast CA measurement of 1 ms with an interval time of 1 µs and an applied potential of 200 mV.

```
var potential
array currents 1000
var num_points
meas_fast_ca potential currents num_points 200m 1u 1m
```

A more comprehensive example can be found in Section 15.5, “Fast CA example”.

14.10.5. meas_scp

MethodSCRIPT	≥1.8
Supported instruments	Nexus

Perform a Stripping Chronopotentiometry (SCP) measurement.

This command performs Stripping Chronopotemiometry also known as Potentiometric Stripping Analysis (PSA). It assumes this command is preceded by a deposition stage, where a potential has been applied for some time.

If the stripping current is set to 0 ampere then the cell will be switched off, and an OCP measurement will be performed. After the measurement, the device will be potentiostatic mode, and the cell remains off. If the stripping current is not 0, the device will quickly switch to galvanostatic mode, apply the current, and measure potential. After the measurement, it will remain in galvanostatic mode, and keep applying the current.

The measured potential over time should be monotonic: the potential should either only go up, or only go down.

Several things can be done to reduce the time to switch to galvanostatic mode:

- Set the range of `VT_CELL_SET_CURRENT` to the same value as `VT_CURRENT`.
- Increase max bandwidth: this will reduce settling time.

The result is the inverse derivative dt / dE. This value is calculated for each potential interval: a bin. The width of a bin can be calculated by: `(bins_end_potential - bins_start_potential) / bin_count`. The center potential of a bin can be calculated by: `bins_start_potential + bin_width / 2 + index * bin_width`. Where index is the index of the bins array, starting at 0.

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

Arguments

Name	Type	Description
Derivative bins	*array* [out] (*int*)	Array to store the derivative dt / dE values in. The *VarType* of will be set to `VT_DT_DE`. The range and status of `VT_POTENTIAL` will be added.
Bin count	*var* [out] (*int*)	Variable to store the number of derivative bins in. The *VarType* of the variable will be set to `VT_COUNT`.
Bins start potential	*var* [out] (*float*)	Variable to store the lowest potential of the first bin. The range of `VT_POTENTIAL` will be added.
Bins end potential	*var* [out] (*float*)	Variable to store the highest potential of the last bin. The range of `VT_POTENTIAL` will be added.
Set current	*var* [out] (*float*)	Variable variable to store the set current in. The set current is the actual current setpoint, which is close to the specified stripping current, but rounded to the nearest achievable current based on the device resolution.
Stripping current	*var / literal* (*float*)	The stripping current to apply, as absolute value. The sign of the current will be picked such that the potential goes towards the end potential. If this is 0, an OCP measurement will be performed.
End potential	*var / literal* (*float*)	The measurement will stop when the measured potential has passed the end potential.
Run time	*var / literal* (*float*)	The maximum measurement time, in case the end potential is not reached. This should be maximum 2100 seconds (35 minutes).

Optional arguments

No optional arguments are supported.

Example

The following example performs a SCP measurement using a 1mA stripping current, an end potential of 0.5V, and a maximum measurement time of 10s.

```
array bins 4096
var bin_count
var bins_start_pot
var bins_end_pot
var current
meas_scp bins bin_count bins_start_pot bins_end_pot current 1m 500m 10
```

A more elaborate example can be found in Section 15.6, “SCP example”.

PalmSens

14.11. Measurement loops

14.11.1. set_scan_dir

MethodSCRIPT	≥1.5
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Reverse the direction of the CV scan.

This command allows the CV loop to skip some portion of its potential sweep and change direction early. If the loop is already stepping in the desired direction, this command does nothing.

Arguments

Name	Type	Description
Direction	*var / literal* (*int/float*)	>0: Set the loop to increase the potential with each step <0: Set the loop to decrease the potential with each step 0: Set the loop to reverse its direction

When using this command with *Direction* equal to 0, care must be taken to avoid double reversals on successive loop iterations. If possible, a value greater than 0 or less than 0 should be used instead.

Example

```
var current
var potential
meas_loop_cv potential current 0 1 -1 100m 1
if current > 10m
# If more than 10 mA current, start scanning downwards immediately
set_scan_dir -1
endif
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.2. meas_loop_lsv

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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.
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:

- `add_meas`
- `poly_we` (*deprecated*)

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 potential current -500m 500m 10m 100m
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.3. meas_loop_acv

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

Perform a AC Voltammetry (ACV) measurement. In a ACV measurement, a potentialscan is performed with a superimposed sine wave. At each step, the ac-potential and ac-current are measured and the complex impedance is calculated.

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.

Arguments

Name	Type	Description
Measured DC potential	*var* [out] (*float*)	Output variable to store the measured DC potential for this iteration.
Measured DC current	*var* [out] (*float*)	Output variable to store the measured DC current for this iteration.
Measured AC potential	*var* [out] (*float*)	Output variable to store the measured AC potential for this iteration.
Measured AC current	*var* [out] (*float*)	Output variable to store the measured AC current for this iteration.
Measured 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)
Measured 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)
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.
Scan rate	*var / literal* (*float*)	The scan rate of the ACV technique. This is the speed at which the applied potential is ramped in V/s. Can only be positive.
Amplitude	*var / literal* (*float*)	Sine wave amplitude in RMS voltage.
Frequency	*var / literal* (*float*)	Sine wave frequency in Hz. This must be chosen such that 4 cycles at this frequency fit in each step period. The step period may be calculated as the step potential divided by the scan rate.

Example

```
meas_loop_acv dc_pot dc_cur ac_pot ac_cur z_real z_imag -500m 500m 10m 20m 10m 15
pck_start
pck_add dc_pot
pck_add ac_cur
pck_end
endloop
```

Perform an ACV measurement and send a data packet for every iteration, with each packet containing the set potential and AC current.

The ACV performs a potential scan from -500 mV to 500 mV with steps of 10 mV, a scanrate of 20 mV/s and an amplitude of 10 mV at 15 Hz. This results in a total of 101 data points at a rate of 2 points per second.

14.11.4. meas_loop_lsp

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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
Measured potential	*var* [out] (*float*)	Output variable to store the measured potential in.
Current setpoint	*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".

Name	Type	Description
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.

Optional arguments

The following optional arguments are supported:

- `add_meas`

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 potential current -5m 5m 100u 1m
pck_start
pck_add current
pck_add potential
pck_end
endloop
```

14.11.5. meas_loop_cv

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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.

Name	Type	Description
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:

- `add_meas`
- `poly_we` (*deprecated*)
- `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 potential current 0 500m -500m 10m 100m
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.6. meas_loop_dpv

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 potential must be an absolute value, the direction of the pulse depends on scan direction.
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".

On the EmStat Pico and Sensit Wearable, `pulse time` may not be larger than 50% of the iteration, otherwise the instrument will throw an error.

Optional arguments

The following optional arguments are supported:

- `add_meas`
- `poly_we` (*deprecated*)

Both `add_meas` and `poly_we` will report the measured value at the pulse minus the measured value just before the pulse: forward - reverse.

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 potential current -500m 500m 10m 20m 5m 100m
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.7. meas_loop_swv

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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.

Name	Type	Description
Frequency	*var / literal* (*float*)	The frequency of the pulses.

Optional arguments

The following optional arguments are supported:

- `add_meas`
- `poly_we` (*deprecated*)

Both `add_meas` and `poly_we` will report the measured value at the pulse minus the measured value just before the pulse: forward - reverse.

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 potential current forward reverse -500m 500m 10m 15m 10
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.8. meas_loop_npv

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
Set potential	*var* [out] (*float*)	Output variable to store the pulse potential for this iteration.
Measured 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.
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:

- `add_meas`
- `poly_we` (*deprecated*)

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 potential current -500m 500m 10m 20m 100m
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.9. meas_loop_ca

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
Set potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration. The set potential is the DC potential setpoint, rounded to the nearest achievable potential based on the device resolution.
Measured 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:

- `add_meas`
- `poly_we` (*deprecated*)

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 potential current 100m 100m 2
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.10. meas_loop_ca_alt_mux

MethodSCRIPT	≥1.5

PalmSens

Supported instruments	EmStat4, Nexus

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

During the interval time, all selected multiplexer channels are measured for an equal amount of time. The measured current is stored in the "Output current" array. This array should be large enough to hold all sampled multiplexer channels. Before this alternating multiplexer command can be used, the multiplexer has to be configured using `mux_config`.

> Some settling time (5 ms) is required after switching a multiplexer channel, make sure the interval time is long enough.

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. The set potential is the DC potential setpoint, rounded to the nearest achievable potential based on the device resolution.
Measured currents	*Array* [out] (*float*)	Output array to store the measured currents for the current iteration. The first value in the array is the measured current on the first multiplexer channel.
DC potential	*var / literal* (*float*)	The DC potential to be applied.
Interval time	*var / literal* (*float*)	The interval between measured data points. Note that the time per multiplexer channel is the interval time divided by the number of multiplexer channels.
Run time	*var / literal* (*float*)	The total run time of the measurement.
First multiplexer channel	*var / literal* (*int*)	The first multiplexer channel to measure (starting at 1).
Last multiplexer channel	*var / literal* (*int*)	The last multiplexer channel to measure (starting at 1).

Optional arguments

The following optional arguments are supported:

- `add_meas`

Example

The following example performs a CA measurement on multiplexer channels 1 to 3. Apply a potential of 1 V, use an interval of 300 ms, and run for 9 seconds.

```
var potential
var time
array currents 3
# NB: first configure the multiplexer using "mux_config"
timer_start
meas_loop_ca_alt_mux potential currents 1 300m 9000m 1i 3i
timer_get time
pck_start
pck_add time
pck_add potential
pck_add currents[0i]
pck_add currents[1i]
pck_add currents[2i]
pck_end
endloop
```

14.11.11. meas_loop_cp

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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
Measured potential	*var* [out] (*float*)	Output variable to store the measured potential for this iteration.
Set current	*var* [out] (*float*)	Output variable to store the set current in. The set current is the actual current setpoint, which is close to the specified DC current, but rounded to the nearest achievable current based on the device resolution.
DC current	*var / literal* (*float*)	The DC current to be applied.
Interval time	*var / literal* (*float*)	The interval between measured data points.

Name	Type	Description
Run time	*var / literal* (*float*)	The total run time of the measurement.

Optional arguments

The following optional arguments are supported:

- `add_meas`

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 potential current 1m 100m 2
pck_start
pck_add current
pck_add potential
pck_end
endloop
```

14.11.12. meas_loop_cp_alt_mux

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

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

During the interval time, all selected multiplexer channels are measured for an equal amount of time. The measured potential is stored in the "Output potential" array. This array should be large enough to hold all sampled multiplexer channels. Before this alternating multiplexer command can be used, the multiplexer has to be configured using `mux_config`.

> Some settling time (5 ms) is required after switching a multiplexer channel, make sure the interval time is long enough.

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
Measured potentials	*Array* [out] (*float*)	Output array to store the measured potentials for the current iteration. The first value in the array is the measured potential on the first multiplexer channel.
Set current	*var* [out] (*float*)	Output variable to store the set current for this iteration. The set current is the actual current setpoint, which is close to the specified DC current, but rounded to the nearest achievable current based on the device resolution.
DC current	*var / literal* (*float*)	The DC current to be applied.
Interval time	*var / literal* (*float*)	The interval between measured data points. Note that the time per multiplexer channel is the interval time divided by the number of multiplexer channels.
Run time	*var / literal* (*float*)	The total run time of the measurement.
First multiplexer channel	*var / literal* (*int*)	The first multiplexer channel to measure (starting at 1).
Last multiplexer channel	*var / literal* (*int*)	The last multiplexer channel to measure (starting at 1).

Optional arguments

The following optional arguments are supported:

- `add_meas`

Example

The following example performs a CP measurement on multiplexer channels 1 to 3. Apply a current of 1 uA, use an interval of 300 ms, and run for 9 seconds.

```
var current
var time
array potentials 3
# NB: first configure the multiplexer using "mux_config"
timer_start
meas_loop_cp_alt_mux potentials current 1u 300m 9000m 1i 3i
timer_get time
pck_start
pck_add time
pck_add current
pck_add potentials[0i]
pck_add potentials[1i]
pck_add potentials[2i]
pck_end
```

```
endloop
```

14.11.13. meas_loop_pad

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
Set potential	*var* [out] (*float*)	Output variable to store the set potential for this iteration. The set potential is the potential setpoint, rounded to the nearest achievable potential based on the device resolution. The reported potential depends on the mode used: DC mode: E_{dc} Pulse mode: E_{pulse} Differential mode: E_{pulse} - E_{dc}
Measured 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*	1 = DC 2 = Pulse 3 = Differential

Optional arguments

The following optional arguments are supported:

- `add_meas`
- `poly_we` (*deprecated*)

Both `add_meas` and `poly_we` will report the measured value the same way as the measured current:

- DC mode: measured value before the pulse.
- Pulse mode: measured value at the end of the pulse.
- Differential mode: DC - pulse.

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 potential current 500m 1500m 10m 50m 10050m 2
pck_start
pck_add potential
pck_add current
pck_end
endloop
```

14.11.14. meas_loop_ocp

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
Measured potential	*var* [out] (*float*)	Output variable to store the measured RE potential in.

Name	Type	Description
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:

- `add_meas`

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 potential 100m 2
pck_start
pck_add potential
pck_end
endloop
```

14.11.15. meas_loop_ocp_alt_mux

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

Perform an Open Circuit Potentiometry (OCP) measurement in alternating multiplexer mode. 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. A more detailed explanation on this technique can be found on the PalmSens knowledge base.

During the interval time, all selected multiplexer channels are measured for an equal amount of time. The measured potential is stored in the "Output potential" array. This array should be large enough to hold all sampled multiplexer channels. Before this alternating multiplexer command can be used, the multiplexer has to be configured using `mux_config`.

> Some settling time (5 ms) is required after switching a multiplexer channel, make sure the interval time is long enough.

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
Measured potentials	*Array* [out] (*float*)	Output array to store the measured potentials for the current iteration. The first value in the array is the measured potential on the first multiplexer channel.
Interval time	*var* / *literal* (*float*)	The interval between measured data points. Note that the time per multiplexer channel is the interval time divided by the number of multiplexer channels.
Run time	*var* / *literal* (*float*)	The total run time of the measurement.
First multiplexer channel	*var* / *literal* (*int*)	The first multiplexer channel to measure (starting at 1).
Last multiplexer channel	*var* / *literal* (*int*)	The last multiplexer channel to measure (starting at 1).

Optional arguments

The following optional arguments are supported:

- `add_meas`

Example

The following example performs an OCP measurement on multiplexer channels 1 to 3. Use an interval of 300 ms, and run for 9 seconds.

```
array potentials 3
var time
# NB: first configure the multiplexer using "mux_config"
timer_start
meas_loop_ocp_alt_mux potentials 300m 9000m 1i 3i
timer_get time
pck_start
pck_add time
pck_add potentials[0i]
pck_add potentials[1i]
pck_add potentials[2i]
pck_end
endloop
```

14.11.16. meas_loop_eis

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
Applied frequency	*var* [out] (*float*)	Output variable to store the applied frequency (Hz) for this iteration.
Measured 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)
Measured 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 `freq`. The measured complex impedance is returned in 2 variables with Z-real in `z_real` and Z-imaginary in `z_imag`. 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 freq z_real z_imag 10m 100k 100 11i 200m
pck_start
pck_add freq
pck_add z_real
pck_add z_imag
pck_end
endloop
```

14.11.17. meas_loop_eis_dual

MethodSCRIPT	≥1.7
Supported instruments	Nexus

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

Similar to `meas_loop_eis`, but measures an extra signal, resulting in a second impedance being measured.

Arguments

Name	Type	Description
Mode	*uint8*	1 = Bipot: the second impedance is E / BiPot 2 = RE vs second sense : the second impedance is S2 / I 3 = SE vs second sense : the second impedance is S2 / I
Applied frequency	*var* [out] (*float*)	Output variable to store the applied frequency (Hz) for this iteration.
Measured 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).
Measured 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).
Second measured Z-real	*var* [out] (*float*)	Output variable to store the real part of the second measured complex impedance. This field also contains the metadata of: mode 1: The measured bipot current. mode 2, or 3: The I-signal (current).
Second measured Z-imaginary	*var* [out] (*float*)	Output variable to store the imaginary part of the second measured complex impedance. This field also contains the metadata of: mode 1: The E-signal (potential). mode 2, or 3: The measured second sense potential.

Name	Type	Description
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_opt`
- `eis_dual_acdc`
- `eis_dual_tdd`

Example

Perform a dual EIS frequency scan from 100 kHz to 100 Hz with 10 mV amplitude and 200 mV DC offset. We use mode 1, measuring the BiPot current as third signal. The frequency for each iteration is returned in variable `freq`. The measured complex impedance is returned in 2 variables with Z-real in `z_r` and Z-imaginary in `z_i`. The measured complex impedance of the BiPot is returned in 2 variables with Z-real in `b_r` and Z-imaginary in `b_i`. In total, 11 points will be measured at frequencies between 100 kHz and 100 Hz, divided on a logarithmic scale.

```
var freq
var z_r
var z_i
var b_r
var b_i
set_pgstat_mode 3
set_bipot_mode 2
cell_on
meas_loop_eis_dual 1 freq z_r z_i b_r b_i 10m 100k 100 11i 200m
pck_start
pck_add freq
pck_add z_r
pck_add z_i
pck_add b_r
pck_add b_i
pck_end
endloop
on_finished:
```


PalmSens

```
cell_off
```

14.11.18. meas_loop_geis

MethodSCRIPT	≥1.3
Supported instruments	EmStat4, Nexus

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.
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 `freq` with 10 mA_{rms} amplitude and 25mA DC offset. The measured complex impedance is returned in 2 variables with Z-real in `z_r` and Z-imaginary in `z_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 freq z_r z_i 10m 100k 100 11i 25m
pck_start
pck_add freq
pck_add z_r
pck_add z_i
pck_end
endloop
```

14.12. Script output

14.12.1. pck_start

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Start a measurement data packet. Up to 129 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
```

14.12.2. pck_add

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.12.3. pck_end

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.12.4. file_open

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

To include variables in your string, see Section 8.7.1, “Interpolated strings”

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 (`/`). As of MethodSCRIPT version 1.5: With mode 2, a counter will be added where "&i" is in the path. This counter will be increased until a file with that path does not exist.
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 named "measurement<count>.txt", where <count> is a counter that increases to make the filename unique.

```
file_open "measurement&i.txt" 2
```

14.12.5. file_close

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.12.6. set_script_output

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.12.7. send_string

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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

> To include variables in your string, see Section 8.7.1, "Interpolated strings"

Arguments

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

Example

Send the text "hello world".

```
send_string "hello world"
```

Output:

```
Thello world
```

14.13. Ranging

14.13.1. set_pot_range (deprecated)

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 and Sensit Wearable require 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.

14.13.2. set_cr (deprecated)

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Set the current range for the given maximum current. The device will select the lowest current range that can measure this current without overloading. Note that the current range has an impact on the potentiostat's bandwidth, please consult the instrument's datasheet for more information.

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.

14.13.3. set_range

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Set the expected maximum absolute current or potential for a given *VarType*. This value will be interpreted as a range between -"Max value" and "Max value". The device will automatically configure itself to best handle values within this range. 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 ensures the WE current can be measured at the best available resolution and accuracy. Note that the current range has an impact on the potentiostat's bandwidth, please consult the instrument's datasheet for more information. This command is ignored in galvanostatic mode.
- Measured potential (`ab`): selects the lowest potential range that can measure the +/- "Max value" potential without causing an overload. This ensures the WE/SE vs RE potential can be measured at the best available resolution and accuracy.
- Applied current (`db`): selects the lowest current range that can apply the +/- "Max value" current without causing an overload. This ensures the WE current can be applied at the best available resolution and accuracy. This command is ignored in non-galvanostatic modes.
- Applied potential (`da`): using `set_range` is not recommended for "Applied potential (`da`)". For the EmStat Pico and Sensit Wearable, consider using `set_range_minmax` instead.

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

Variable type	EmStat Pico	Sensit Wearable	Emstat4	Nexus
`ba`	Yes	Yes	Yes	Yes
`ab`	No	No	Yes	Yes
`db`	No	No	Yes	Yes
`da`	Not recommended	Not recommended	No	No

This command is ignored when autoranging is enabled for `meas_loop_eis`, `meas_loop_acv` and `meas_ms_eis`.

Calling `set_range` with "Max value" is equivalent to calling `set_range_minmax` with -"Max value" and "Max value".

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
```

14.13.4. set_range_minmax

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Set the expected minimum and maximum current or potential for a given *VarType*. The device will automatically configure itself to best handle values within the range between the specified minimum and maximum value. Unsupported *VarTypes* are ignored without throwing an error.

The following variable types are currently supported:

- Measured current (`ba`): selects the lowest current range that can measure both the "Min value" and "Max value" current without causing an overload. This ensures the WE current can be measured at the best available resolution and accuracy. Note that the current range has an impact on the potentiostat's bandwidth, please consult the instrument's datasheet for more information. This command is ignored in galvanostatic mode.
- Measured potential (`ab`): selects the lowest potential range that can measure both the "Min value" and "Max value" potential without causing an overload. This ensures the WE/SE vs RE potential can be measured at the best available resolution and accuracy.
- Applied current (`db`): selects the lowest current range that can apply both the "Min value" and "Max value" current without causing an overload. This ensures the WE current can be applied at the best available resolution and accuracy. This command is ignored in non-galvanostatic modes.
- Applied potential (`da`): configures the device to be able to apply both the "Min value" and the "Max value" potential. The EmStat Pico and Sensit Wearable require this command to reach its full applied potential, as it has a limited "Dynamic potential window" that can moved around with this command. See Section B.1, "PGStat mode properties" for more information.

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

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

This command is ignored when autoranging is enabled for `meas_loop_eis`, `meas_loop_acv` and `meas_ms_eis`.

PalmSens

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
```

14.13.5. set_autoranging

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Configure the autoranging for all `meas_loop_*` commands. Autoranging selects the most appropriate range for the measured value in the last measurement loop iteration.

The range selected during autoranging is limited by the min and max arguments. If min and max are the same value, autoranging is disabled.

> No autoranging is performed on calling this command.

> The `set_range` and `set_range_minmax` commands are not affected by the min and max arguments of `set_autoranging`.

Arguments

Name	Type	Description
Var type	*VarType*	The type of variable to measure, see Chapter 7, *Variable types*.
Min	*var / literal* (*float*)	The minimum absolute value to use for autoranging. Can be used to exclude lower ranges. Must be positive.
Max	*var / literal* (*float*)	The maximum absolute value to use for autoranging. Can be used to exclude higher ranges. Must be positive.

> The *VarType* argument is new in MethodSCRIPT v1.3. 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
```

14.13.6. trim_enable

MethodSCRIPT	≥1.8
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Enable or disable trimming for a given *VarType*. Enabling trimming will allow the use of lower ranges, by removing (trimming) a DC offset. However, it may lead to longer settling times.

By default, and after setting pgstat mode, trimming is disabled.

The following variable types are currently supported:

- Measured potential (`ab`)
- Measured second sense potential (`ah` and `ai`)
- Measured current (`ba`)

Unsupported variable types are ignored without throwing an error.

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

Variable type	EmStat Pico	Sensit Wearable	Emstat4	Nexus
`ab`	No	No	Yes	Yes
`ah`	No	No	No	Yes
`ai`	No	No	No	Yes
`ba`	No	No	Yes	No

Arguments

Name	Type	Description
Variable type	*VarType*	The type identifier for this value.
Enable	*var / literal* (*int*)	1 to enable, 0 to disable

Example

Enable trimming for measured potential:

```
trim_enable ab 1
```

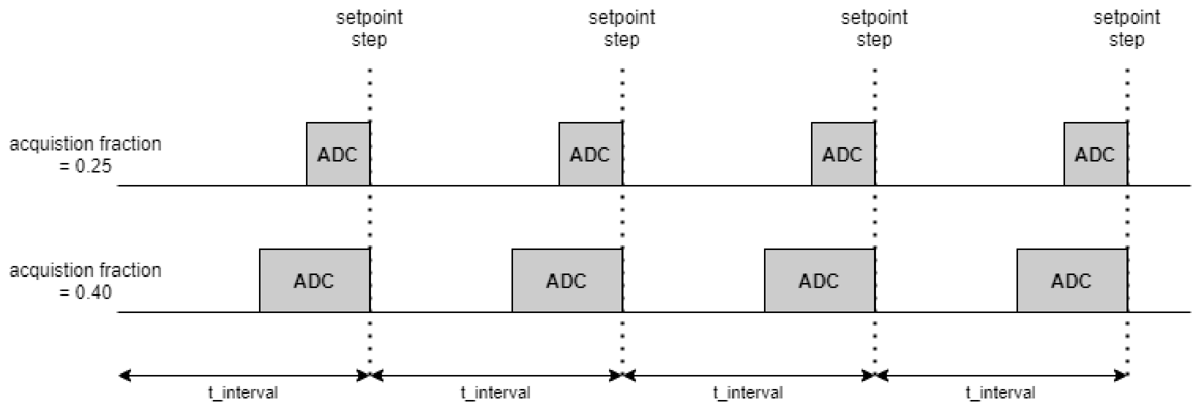
14.14. PGStat

14.14.1. set_acquisition_frac

MethodSCRIPT	≥1.3
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.14.2. set_acquisition_frac_autoadjust

MethodSCRIPT	≥1.4
Supported instruments	EmStat4, Nexus

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.

PalmSens

Example

Set acquisition auto adjust frequency to filter out 50 Hertz.

```
set_acquisition_frac_autoadjust 50
```

14.14.3. set_ir_comp

MethodSCRIPT	≥1.5
Supported instruments	EmStat4, Nexus

Set resistance to be compensated by iR compensation.

Compensate an ohmic drop (also known as iR drop) by increasing the WE potential based on the WE current.

This can be used to correct for an unwanted voltage drop between the WE and RE electrodes. It is only necessary when the ohmic drop is significant when compared to the WE potential. iR compensation is only possible if the resistance over which this voltage drop occurs is known and constant.

The EIS technique can be used to determine frequency independent impedances between RE and WE. This is a way of isolating the impedance that behaves like a pure resistor (at least over frequency), which implies it is eligible for iR compensation. In most cells, this is the lowest impedance point in the Nyquist plot where the imaginary impedance (*Z''*) is zero.

Compensating for large iR drops can cause the system to become unstable.

INFO: iR compensation is not available in galvanostatic mode, or for high frequency measurements like EIS.

EmStat4

iR compensation is only supported on an EmStat4X that is licensed for iR compensation.

Arguments

Name	Type	Description
Resistance	*var / literal* (*float*)	The resistance to compensate for in ohms (Ω)

Example

Compensate for the voltage drop over a resistance of 100 Ω between RE and WE.

```
set_ir_comp 100
```

14.14.4. set_pgstat_chan

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 `add_meas` 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
```

14.14.5. set_poly_we_mode (deprecated)

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable

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
```


PalmSens

The `set_poly_we_mode` command has been deprecated and may be removed in future releases. Use the `set_bipot_mode` command instead.

14.14.6. set_pgstat_mode

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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. Additionally if there is a bipot channel configured, it will be cleared by this command.

Arguments

Name	Type	Description
PGStat mode	*uint8*	0 = Off // 1 = OBSOLETE, this was OCP but dropped later 2 = Low Speed mode 3 = High Speed mode 4 = Max Range mode 5 = Poly WE (BiPot) mode (*deprecated*) 6 = Galvanostatic mode

Example

Set hardware configuration to high speed mode.

```
set_pgstat_mode 3
```

14.14.7. set_bipot_mode

MethodSCRIPT	≥1.7
Supported instruments	EmStat Pico, Sensit Wearable, Nexus

Set the mode of the second working electrode. Can only be changed while cell is off. The second cell will be switched on and off together with the main PGStat.

On the EmStat Pico this command sets the hardware configuration for the non-active channel to Poly WE (BiPot) mode. Consequently, this also initializes all channel settings to the default values for that mode. This is similar to calling `set_pgstat_mode` on that channel (with the now deprecated mode 5), which was the only way to configure the bipot channel before MethodSCRIPT v1.7.

Changing the bipot mode may not preserve the potential on the bipot. As such, setting a non-disabled mode should almost always be followed by `set_bipot_potential` to ensure the potential is as desired.

Arguments

Name	Type	Description
BiPot mode	*uint8*	0 = Disabled 1 = Fixed: additional WE is kept fixed at the specified potential (WE2 is offset from RE) 2 = Offset: additional WE will follow the main WE at a specified offset potential (WE2 is offset from S, or WE1 if S is not available)

Example

Set the BiPot in offset mode.

```
set_bipot_mode 2
```

14.14.8. set_bipot_potential

MethodSCRIPT	≥1.7
Supported instruments	EmStat Pico, Sensit Wearable, Nexus

Set the potential (offset) of the second working electrode. The second electrode must have already been enabled using `set_bipot_mode`.

Arguments

Name	Type	Description
Potential	*var / literal (float)*	Potential (offset) of the second working electrode in Volt.

Example

Set the potential offset of the second working electrode to 0.1V.

```
set_bipot_mode 2
set_bipot_potential 100m
```

14.14.9. set_max_bandwidth

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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. The actual bandwidth is dependent on multiple factors, such as the current range, please consult the instrument's datasheet for more information.

If bipot mode is enabled (e.g. using the `set_bipot_mode` command), this command also applies to the bipot channel.

Arguments

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

Optional arguments

The following optional arguments are supported:

- `filter_type`

Example

Set the max bandwidth to a frequency of 1 kHz.

```
set_max_bandwidth 1k
```

14.15. GPIO

14.15.1. set_gpio_cfg

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 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
```

14.15.2. set_gpio_pullup

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.15.3. set_gpio

MethodSCRIPT	≥1.1
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 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
```

14.15.4. get_gpio

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Get the GPIO input pin values. This reads the input value of all GPIO pins configured as input. Pins that are not configured as input will always return a bit value of `0`. 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
```

14.15.5. set_gpio_msk

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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
```

14.15.6. get_gpio_msk

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Get the GPIO input pin values with a mask. This reads the input value of all GPIO pins specified by the mask. Any pins that are not configured as input or outside of the specified mask will return a bit value of `0`. 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
```

14.16. I2C

14.16.1. i2c_config

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

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 (only 7-bit mode is currently supported)

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 14.15.1, "set_gpio_cfg" for more information on configuring GPIO.*

14.16.2. i2c_write_byte

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Transmit one byte to an I²C target device. This also generates the I²C start and stop conditions. If a NACK (Not Acknowledge) was received from the target device, the user should handle this and reset the *ACK status* variable.

Arguments

Name	Type	Description
Device address	*var / literal* (*int*)	The address of the target device.

Name	Type	Description
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 ack
store_var ack 0i ja
i2c_write_byte 0x48 0x03 ack
if ack != 0i
abort
endif
```

14.16.3. i2c_read_byte

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Receive one byte from an I²C target device. This also generates the I²C start and stop conditions. If a NACK (Not Acknowledge) was received from the target device, the user should handle this and reset the *ACK status* variable.

Arguments

Name	Type	Description
Device address	*var* / *literal* (*int*)	The (7-bit) address of the target device.
Receive data	*var* (*int*)	Variable to store the received byte in.

Name	Type	Description
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 `data`. Abort the script if the I²C operation failed.

```
var ack
var data
store_var ack 0i ja
i2c_read_byte 0x48i data ack
if ack != 0i
abort
endif
```

14.16.4. i2c_write

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Write one or more bytes to an I²C target device. This also generates the I²C start and stop conditions. If a NACK (Not Acknowledge) was received from the target device, the user should handle this and reset the *ACK status* variable.

Arguments

Name	Type	Description
Device address	*var / literal* (*int*)	The (7-bit) address of the target 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
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 target device with address 0x48.

```
var ack
store_var ack 0i ja
array w_array 2
store_var w_array[0i] 12i aa
store_var w_array[1i] 34i aa
i2c_write 0x48 w_array 2 ack
```

14.16.5. i2c_read

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Read one or more bytes from an I²C target device. This also generates the I²C start and stop conditions. If a NACK (Not Acknowledge) was received from the target device, the user should handle this and reset the *ACK status* variable.

Arguments

Name	Type	Description
Device address	*var / literal* (*int*)	The (7-bit) address of the target device.
Received data	array (*int*)	Reference to an array to store received data in.
Receive count	*var / literal* (*int*)	Number of bytes to receive. Minimum value = 1, maximum value is 255 or size of the array.

Name	Type	Description
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 target device with address 0x48 and store them in array `r_array`.

```
var ack
store_var ack 0i ja
array r_array 4
i2c_read 0x48 r_array 4 ack
```

14.16.6. i2c_write_read

MethodSCRIPT	≥1.2
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Write to and read from an I²C target 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 target device, the user should handle this and reset the *ACK status* variable.

Arguments

Name	Type	Description
Device address	*var / literal* (*int*)	The (7-bit) address of the target 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.
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.

Name	Type	Description
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 target device with address 0x48, and then immediately read 4 bytes.

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

14.17. Multiplexers

14.17.1. mux_config

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, EmStat4, Nexus

Configure a multiplexer to use in MethodSCRIPT. This tells the instrument which multiplexer (mux) is connected and which settings to set. Configuring the multiplexer will configure GPIO pins designated for that particular multiplexer. When the multiplexer type is set to *none*, the designated GPIO pins for the previously selected mux are switched back to input.

Arguments

Name	Type	Description
Mux type	*var / literal* (*int*)	The multiplexer type, see Table 11, “Mux type values”
Config	(*uint32*)	MUX configuration as bit mask, see Table 12, “Mux configuration fields”

Table 11. Mux type values

ID	Multiplexer type
0	None
1	Original MUX8
2	Original MUX16
3	MUX8-R2
4	Multiplexer for EmStat Pico, 16 channel
5	Multiplexer for EmStat Pico, 256 channel matrix

Configuration options are defined to be standard across all multiplexers. However, not all options can be set (automatically) on all multiplexer. Please resort to the manual of the particular multiplexer to find out which options are available.

Table 12. Mux configuration fields

Mask	Option
0x0002	Switch box 1
0x0004	Switch box 2
0x0008	OCP mode enable
0x0010	Common RE and CE
0x0020	Connect RE to CE
0x0040	Connect SE to WE
0x0180	WE mode (0x0000 = float, 0x0100 = GND, 0x0180 = standby voltage)

Example

The following example demonstrates configuring the MUX8-R2 to be enabled with RE connected to CE, and WE to GND.

```
mux_config 3i 0x0120
```

14.17.2. mux_get_channel_count

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, EmStat4, Nexus

Get the number of channels on the multiplexer setup. Different multiplexers can have a different number of channels and this command should help making scripts more universal. The returned number of channels is the number provided by the multiplexer rather than the number of channels actually connected to a solution.

In case of the MUX8-R2, this command will give the total number of channels available in the chain. So for three MUX8-R2s in daisy-chain configuration, it will return 24 channels.

Arguments

Name	Type	Description
Number of channels	*var* [out] (*int*)	Variable to store the total available number of channels The *VarType* of this variable will be set to `VT_UNKNOWN` (`aa`).

Example

Store the number of available mux channels in variable `n`.

```
var n
mux_get_channel_count n
```

14.17.3. mux_set_channel

MethodSCRIPT	≥1.4
Supported instruments	EmStat Pico, EmStat4, Nexus

Select channel on the multiplexer. The multiplexer has to be configured with `mux_config` before selecting.

Arguments

Name	Type	Description
Channel	*var / literal* (*int*)	The channel to select (starting from 1)

Example

Select channel 3 on the MUX.

```
mux_set_channel 3i
```

14.18. Misc

14.18.1. notify_led

MethodSCRIPT	≥1.5
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Notify the user of a user-defined event, using the LED. This is intended as a generic way to notify the user of test results, errors, the progress of the measurement, or other events. Because different devices have different LED (color) availability, the device will choose the best way to signal each event type. Notifications are persistent between script runs.

Arguments

Name	Type	Description
Notify mode	*uint16*	Notify type. See tables below for device specific behavior. 0 = Clear notifications 1 = Idle 2 = Busy 3 = Attention 4 = Test pass 5 = Test fail 6 = Warning 7 = Error

EmStat4

The EmStat4 will use the multicolor LED for all status notifications. Default LED behavior is overridden by notifications.

Notify type	Mode	Behavior description
0	Clear notifications	Default LED behavior
1	Idle	Solid blue LED
2	Busy	Solid red LED
3	Attention	Solid white LED
4	Test pass	Solid green LED
5	Test fail	Solid red LED
6	Warning	Solid yellow LED
7	Error	Solid yellow LED

EmStat Pico

The EmStat Pico will use the blue and red LED for all status notifications. Default LED behavior is overridden by notifications.

Notify type	Mode	Behavior description
0	Clear notifications	Default LED behavior
1	Idle	Red LED off, solid blue LED
2	Busy	Solid red LED, solid blue LED
3	Attention	Solid red LED, blue LED off
4	Test pass	Red LED off, solid blue LED
5	Test fail	Solid red LED, blue LED off

PalmSens

Notify type	Mode	Behavior description
6	Warning	Solid red LED, blue LED off
7	Error	Solid red LED, blue LED off

Sensit Wearable

The Sensit Wearable will use the blue LED for all status notifications. Default LED behavior is overridden by notifications.

Notify type	Mode	Behavior description
0	Clear notifications	Default LED behavior
1	Idle	Blinking blue LED (0.5 Hz)
2	Busy	Solid blue LED
3	Attention	Pulse blue LED (100 ms on, 400 ms off)
4	Test pass	Pulse blue LED (900 ms on, 100 ms off)
5	Test fail	Blinking blue LED (4 Hz)
6	Warning	Blinking blue LED (4 Hz)
7	Error	Blinking blue LED (4 Hz)

Example

Notify the user that a measurement is ongoing. On the EmStat4 and EmStat Pico this turns on the red LED.

```
notify_led 2
```

14.18.2. smooth

MethodSCRIPT	≥1.6
Supported instruments	Sensit Wearable, EmStat4, Nexus

Apply Savitzky-Golay smoothing to data in an array.

Apart from their float value, variables in the output array will be identical to those in the input array (noise, vartype, etc).

If the output array is longer than the input, excess variables will be left unchanged.

The length of the data arrays must be sufficiently long to apply the requested smoothing strength. If the data is too short, error code `0x420F` will be returned. In this case, either a lower strength or larger array must be used.

Arguments

Name	Type	Description
Input array	*Array* (*float*)	Input data
Output array	*Array* [out] (*float*)	Output data. Must be at least as long as the input. It may be the same array as the input, in which case the original data will be overwritten.
Smoothness	*var / literal* (*int*)	Smoothing strength 0 = Low 1 = Medium 2 = High 3 = Very High

Example

Smooth the `invals` array and store the outcome in the `outvals` array

```
smooth invals outvals 2i
```

14.18.3. peak_detect

MethodSCRIPT	≥1.6
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Find peaks in the given data.

Multiple peaks may be detected, and their info is stored in the `Output indices` and `Output heights` arrays. The peaks in these arrays are sorted in descending order of peak height. If there is not enough room to store all the peaks detected in the output arrays, the rest will be ignored.

Arguments

Name	Type	Description
Input array	*Array* (*float*)	Input data
Output indices	*Array* [out] (*int*)	The indices of the peaks, sorted in descending order of height. If fewer peaks were present in the data than can be stored in the array, excess values will be -1.
Output heights	*Array* [out] (*float*)	The heights of the peaks, sorted in descending order of height. If fewer peaks were present in the data than can be stored in the array, excess values will be 0. Heights are always absolute positive values, even if the detected peaks are negative.

Name	Type	Description
Direction	*var / literal* (*int*)	Direction of peaks to detect 0 = Positive going 1 = Negative going
Threshold	*var / literal* (*float*)	Threshold of peak heights to detect. Lower peaks are ignored.

Optional arguments

The following optional arguments are supported:

- `window`

Example

Detect the two highest positive peaks in an input array, larger than 10e-6.

```
array indices 2
array heights 2
peak_detect data indices heights 0i 10u
```

14.18.4. beep

MethodSCRIPT	≥1.7
Supported instruments	Nexus

Make a beep, and wait for it to be finished. If the device is muted then no tone will play, but the command will still wait.

Arguments

Name	Type	Description
Tone	*uint8*	The note to play - Higher values are higher pitched. The exact note played is counted in semitones above C3. e.g. 12 is C4 (~262Hz), 2 is D3 (~147Hz)
Volume	*uint8*	From 0 to 100. If the device does not support volume control, this is ignored.
Duration	*var /literal* (*float*)	Seconds to play the tone for.

Example

Play C4 at half volume for a second

```
beep 12 50 1000
```

14.18.5. battery_perc

MethodSCRIPT	≥1.7
Supported instruments	Sensit Wearable

Read the battery’s charge as a percentage.

If the battery’s state of charge cannot be measured, `battery_perc` will return -1. For example this happens on the Sensit Wearable during charging.

Arguments

Name	Type	Description
Measured Percentage	*var* [out] (*float*)	Variable to store value into.

Example

Read the current battery charge, and report it.

```
var p
battery_perc p
if p < 0
send_string "I'm a Sensit Wearable and the battery is charging!"
else
send_string f"Battery at {p}%"
endif
```

14.18.6. get_progress

MethodSCRIPT	≥1.7
Supported instruments	EmStat Pico, Sensit Wearable, EmStat4, Nexus

Read the progress through the current measurement, from 0 to 100.

It is an error to use this outside a measurement loop

Arguments

Name	Type	Description
Measured Percentage	*var* [out] (*int*)	Variable to store value into.

Example

Write out progress through a measurement as it iterates

```
var prog
var ocp
meas_loop_ocp ocp 100m 10
get_progress prog
send_string f"{prog}%"
endloop
```

14.18.7. linear_fit

MethodSCRIPT	≥1.8
Supported instruments	Sensit Wearable, EmStat4, Nexus

Perform a linear least squares regression on a set of data.

Arguments

Name	Type	Description
X datapoints	*Array* (*float*)	Array containing X datapoints as floats
Y datapoints	*Array* (*float*)	Array containing Y datapoints as floats
Slope	*var* [out] (*float*)	Slope of the best fit line
Intercept	*var* [out] (*float*)	Y-intercept of the best fit line

Example

Perform an LSV, and then perform a best fit on the data. For a resistive load, the slope of this data equals the resistance.

```
e
array ps 101i
array cs 101i
```

```
var p
var c
var ix
store_var ix 0i ja
set_pgstat_mode 2
cell_on
set_autoranging ba 1a 1
meas_loop_lsv p c 0 1 10m 1
    copy_var p ps[ix]
    copy_var c cs[ix]
    add_var ix 1i
endloop
var slope
var offset
linear_fit cs ps slope offset
pck_start
    pck_add slope
    pck_add offset
pck_end
```

14.18.8. mean

MethodSCRIPT	≥1.8
Supported instruments	Sensit Wearable, EmStat4, Nexus

Take the mean of an array of data.

The returned mean will have the same *VarType* as the first element of the input data.

Arguments

Name	Type	Description
Data	*Array* (*float*)	Array containing datapoints as floats
Mean	*var* [out] (*float*)	Mean of the data in the input array

Example

Perform an OCP measurement and report the mean of the last 100 data points

```
e
array data 200i
var p
var ix
store_var ix 0i ja
```

```
meas_loop_ocp p 10m 2
    copy_var p data[ix]
    add_var ix 1i
endloop
subarray last100 data 100i 100i
var avg
mean last100 avg
send_string f"{avg}"
```

14.18.9. qr_scan

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Trigger the QR code scanner.

The scanner recognises QR codes in the following format:

`[Arbitrary Data]#!PS[(integer)|(float),]`

After the text `#!PS`, all remaining text must be a comma separated list of integer or float values. For example the QR text:

`Leading text #!PS3,10,2.0,10e-1,50`

would produce the values:

`3 (int), 10 (int), 2.0 (float), 1.0 (float), 50 (int)`.

Output count is set according to the following rules:

- -3 if no code was scanned.
- -2 if a code was scanned and contained a `#!PS` marker but with invalid formatting.
- -1 if a code was scanned but contained no `#!PS` marker.
- The number of values parsed following the `#!PS` marker.
  - This can be zero for a code that ends immediately.
  - This can be greater than the size of the output array. In this case the output array will contain as many values as it can hold, and the output count will be set to the number of values that were parsed.

Arguments

Name	Type	Description
Output Array	*Array* [out] (*float*)	Array to store scanned values into
Output count	*var* (*int*)	The number of variables scanned, or -1 in the case of an error

Optional arguments

The following optional arguments are supported:

- `qr_log`

Example

Scan a QR code, print the number of values scanned and then print them out.

```
array outputs 5i
var out_count
qr_scan outputs out_count
var ix
store_var ix 0i aa
send_string f"Parsed {out_count} values"
if out_count < 0i
  # This is some kind of error. For this example we will just end the script.
  abort
end if
loop ix < out_count
  if ix >= 5i
    # There were more values in the QR code than we could store. So
    # only print the first 5.
    break
  end if
  send_string f"{ix}: {outputs[ix]}"
  add_var ix 1i
endloop
```

14.19. Display

14.19.1. display_draw

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Immediately prompt the display to be updated.

The display will be occassionally updated regardless - however it is advised that the user includes `display_draw` when they have added all the elements they wish to be shown, in order to make the interface as responsive as possible.

If this command is not used, the user may feel a noticeable lag before items appear on the display.

Arguments

-

14.19.2. display_clear

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Remove all elements from the display.

This won’t be reflected on-screen until it is next drawn, see Section 14.19.1, “display_draw”.

Arguments

-

14.19.3. display_text

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Add a new line of text to the display, to be shown the next time the display is drawn (see Section 14.19.1, “display_draw”).

If the text is too long, it will be truncated. If there is not room on the screen, this will have no effect.

This text will appear alongside elements added by the Section 14.19.4, “display_icon”, Section 14.19.6, “display_btns” and Section 14.19.5, “display_progress” commands.

Arguments

Name	Type	Description
Text	*string*	The line to display
Size	*var / literal* (int / float)	0 = Small 1 = Large

Example

Add the lines "Hello" and "World" to the display.

```
display_text "Hello" 0
display_text "World" 0
display_draw
```

14.19.4. display_icon

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Add an icon on the display, to be shown the next time the display is drawn (see Section 14.19.1, “display_draw”).

This icon will appear alongside elements added by the Section 14.19.3, “display_text”, Section 14.19.6, “display_btns” and Section 14.19.5, “display_progress” commands.

Arguments

Name	Type	Description
Icon type	*var / literal* (int / float)	1 = Okay 2 = Warn 3 = Query 4 = Error 5 = Info 6 = QR Code 7 = Spinner 8 = Add droplet 9 = Insert SPE 10 = User Icon 1 11 = User Icon 2 12 = User Icon 3 13 = User Icon 4 14 = User Icon 5

Example

Display a warning to the user

```
display_icon 1i
display_text "This is a warning!"
display_draw
```

14.19.5. display_progress

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Add a progress bar on the display, to be shown the next time the display is drawn (see Section 14.19.1, “display_draw”).

This icon will appear alongside elements added by the Section 14.19.3, “display_text”, Section 14.19.4, “display_icon” and Section 14.19.6, “display_btns” commands.

Arguments

Name	Type	Description
Progress value	*var / literal* (int / float)	The progress to show between 0 and 100. Any value beyond these limits will hide the progress bar.

Example

Display a progress bar filling from 0 to 100 to the user, then hide it.

```
var x
store_var x 0 ja
loop x < 100
  display_progress x
  display_draw
  add_var x 1i
endloop
# Hide the progress bar
display_progress -1
display_draw
```

14.19.6. display_btns

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Show one or two buttons on the display, then immediately update the display and wait for the user to press one.

These buttons will be shown at the bottom of the display, below any elements added by the Section 14.19.3, “display_text”, Section 14.19.4, “display_icon” and Section 14.19.5, “display_progress” commands.

Note that since this command waits for the user to press a button, you must add other elements (text, icons, progress) *before* this command.

After a button is pressed, the screen will be cleared.

Arguments

Name	Type	Description
Pressed button ID	*var* [out]	Indicates the pressed button. If the left button is pressed or there is only one button, this will be `0i`. If the right button is pressed, it will be `1i`.
Left Text	*string*	The text to be shown on the left button
Right Text	*string*	The text to be shown on the right button. If this is the empty string `""`, only the left button will be shown.

Example

Show the user two buttons, and tell them which one they pressed.

```
var v
display_text "Pick a button!"
display_btns v "Left" "Right"
if v == 0i
send_string "You pressed left!"
elseif v == 1i
send_string "You pressed right!"
endif
```

14.19.7. display_inp_num

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Prompt the user for a numerical value, and wait until one is provided.

Everything else on the screen will be removed to show the keypad. When this command completes, the display will be cleared.

Arguments

Name	Type	Description
Prompt	*string*	The prompt to show while waiting for input
Value	*var* [out]	The int/float value inputted by the user
Int or Float	*literal* (int)	0 = Input an integer 1 = Input a float

Example

Ask the user for a value, and then display it back to them.

```
var v
display_inp_num "Please enter a number" v 0
display_text f"You entered {v}"
display_draw
```

14.19.8. display_scroll_add

MethodSCRIPT	≥1.8

Supported instruments	EmStat4T

Add an entry to the scroll list on the display, to be shown using Section 14.19.9, “display_scroll_get”.

Up to ten scroll entries may be added, after which adding more will have no effect.

This command will have no immediate effect on the display, until Section 14.19.9, “display_scroll_get” is used.

Arguments

Name	Type	Description
Text	*string*	The line to show to the user

Example

Give the user 4 items to choose between

```
var choice
display_scroll_add "Entry A"
display_scroll_add "Entry B"
display_scroll_add "Entry C"
display_scroll_add "Entry D"
display_scroll_get "Make a choice" choice
# If the user picks "Entry C", then `choice` will equal `2`.
```

14.19.9. display_scroll_get

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Show the scroll items (added by Section 14.19.8, “display_scroll_add”) to the user, and wait for a choice to be made.

At least one entry must have been added since the last time this command was called.

Everything else on the screen will be removed to show the scroll selection. When this command completes, the display will be cleared.

Arguments

Name	Type	Description
Prompt	*string*	The prompt to show while the user chooses
Chosen	*var* [out] (*int*)	The index of the value chosen by the user (starting at 0)

Example

Give the user 4 items to choose between

```
var choice
display_scroll_add "Entry A"
display_scroll_add "Entry B"
display_scroll_add "Entry C"
display_scroll_add "Entry D"
display_scroll_get "Make a choice" choice
# If the user picks "Entry C", then `choice` will equal `2`.
```

14.19.10. display_keyboard

MethodSCRIPT	≥1.8
Supported instruments	EmStat4T

Get a line of text entered by the user and record it to the script output.

Everything else on the screen will be removed to show the keyboard. When this command completes, the display will be cleared.

Arguments

Name	Type	Description
Text	*string*	The prompt to show the user

Example

Ask the user to record their name in the log

```
display_keyboard "Enter user name:"
```

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 freq
var z_real
var z_imag
# 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 freq z_real z_imag 10m 200k 200 11 0
    # Send measurement package containing frequency, Z-real and Z-imaginary.
    pck_start
    pck_add freq
    pck_add z_real
    pck_add z_imag
    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 current
var potential
# 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.
# If you want autoranging before the measurement, but no datapoints,
# remove the pck_ commands from the loop.
meas_loop_ca potential current -500m 500m 5
    pck_start
    pck_add potential
    pck_add current
    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 potential current -500m 1500m 10m 100m
    # Send package containing set potential and measured WE current.
    pck_start
    pck_add potential
    pck_add current
    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 current
var potential
var forward
var reverse
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 potential current forward reverse -500m 500m 10m 100m 10
    # Send package with set potential, forward current - reverse current,
    # forward current, and reverse current.
    pck_start
    pck_add potential
    pck_add current
    pck_add forward
    pck_add reverse
    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 total_points
var scan_points
# Variable used as loop iterator for total points processed
var i
# Variable used as loop iterator for points within a scan
var j
# Array to store set potentials
array potentials 15
# Array to store measured currents
array currents 15
# Configure instrument to perform this measurement on 1 kohm
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 potentials currents total_points 0 -100m 100m 100m 10 nscans(3) nscans_avg(2)
# Points per scan (scan_points) is points total (total_points) / nscans (3)
copy_var total_points scan_points
div_var scan_points 3i
store_var i 0i ja
```

```
# Loop through scans
loop i < total_points
   store_var j 0i ja
   send_string "scan separator"
   # Loop through points in scan
   loop j < scan_points
      pck_start meta_msk(0x00)
      # Add index to packet
      pck_add i
      # Add set potential to packet
      pck_add potentials[i]
      # Add measured current to packet
      pck_add currents[i]
      pck_end
      # Increase indexes
      add_var i 1i
      add_var j 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. Fast CA example

The following example performs a Fast CA measurement of 1 ms with an interval time of 1 µs. A potential step from 100 mV to 200 mV is performed before starting the measurement.

Timestamps are calculated in MethodSCRIPT and added to the data packages, so PSTrace can automatically plot the current versus time. Note that the timestamps are calculated using multiplication, not addition. Mathematically it would be the same to add 1 us to the timestamp every iteration. However, due to accumulation of rounding errors, such an approach could lead to very inaccurate timestamps, and as a consequenc, a potentially misleading plot. Because the index variable is an integer, it can be incremented without any rounding issues. The `int_to_float` command is then necessary to convert a variable from integer to floating-point format before it can be multiplied with another floating-point number. Finally, the *VarType* will be set to `eb` (`VT_TIME`) so the host software (e.g. PSTrace) can identify that the variable contains a time.

```
array currents 1000
var current
var potential
var num_points
var index
var time
set_pgstat_mode 2
set_range ba 200u  # set current range to +/- 200 uA
set_max_bandwidth 1G  # set bandwidth to 1 GHz
set_e 100m
cell_on
wait 100m
meas_fast_ca potential currents num_points 200m 1u 1m
cell_off
store_var index 0i ja
loop index < num_points
  copy_var index time
  int_to_float time
  alter_vartype time eb
  mul_var time 1u
  pck_start
  pck_add time
  pck_add currents[index]
  pck_end
  add_var index 1i
endloop
```

Example output

```
L
Peb8000000 ;baDF23478p,10,212
Peb80F423Fp;baDF2EBF8p,10,212
Peb81E847Fp;baE064608p,10,212
...
Peb80F3688n;ba8030D34n,10,212
Peb80F3A70n;ba8030D7Fn,10,212
Peb80F3E58n;ba8030DA4n,10,212
+
```

15.6. SCP example

The following example performs an SCP measurement using a 1mA stripping current, an end potential of 0.5V, and a maximum measurement time of 10s. The current range for set and measured current are set to the same value, to speed up the switch to galvanostatic mode. It is preceded by a deposition stage, where a potential of 2V has been applied for 3s. After the measurement, the cell is switched off, to avoid the cell going too far passed the end potential. Lastly, the potential is sent for every bin with non-zero dt / dE, so the host software (e.g. PSTrace) can plot the potential on the x-axis, and dt / dE on the y-axis.

```
array bins 4096
var bin_count
var bins_start_pot
var bins_end_pot
var current
var potential
set_pgstat_mode 3
set_max_bandwidth 500k
set_range ab 1    # Set VT_POTENTIAL range.
set_range db 1m   # Set VT_CELL_SET_CURRENT range.
set_range ba 1m   # Set VT_CURRENT range the same as VT_CELL_SET_CURRENT.
#
# Deposition stage
set_e 2
cell_on
wait 3
#
# The SCP measurement
meas_scp bins bin_count bins_start_pot bins_end_pot current 1m 500m 10
cell_off  # Quickly turn off the cell, otherwise we keep applying current.
#
# Find the first and last non zero value in the bins.
var index
var index_start
var index_end
store_var index_start 0i ja
store_var index_end 0i ja
```


PalmSens

```
store_var index 0i ja
loop index < bin_count
    if bins[index] > 0
        copy_var index index_start
        breakloop
    endif
    add_var index 1i
endloop
copy_var bin_count index
sub_var index 1i
loop index >= 0
    if bins[index] > 0
        copy_var index index_end
        add_var index_end 1i
        breakloop
    endif
    sub_var index 1i
endloop
#
# Precompute the bin width: (bins_end_pot - bins_start_pot) / bin_count
var bin_width
var bin_count_flt
copy_var bin_count bin_count_flt
int_to_float bin_count_flt
copy_var bins_end_pot bin_width
sub_var bin_width bins_start_pot
div_var bin_width bin_count_flt
#
# Precompute the center value of the first bin: bins_start_pot + bin_width / 2.
var center_start
copy_var bin_width center_start
div_var center_start 2
add_var center_start bins_start_pot
#
# Loop over the bins, skipping the zero values at the start and end.
var index_flt
copy_var index_start index
loop index < index_end
    # Calculate center bin potential: center_start + index * bin_width
    copy_var index index_flt
    int_to_float index_flt
    copy_var bin_width potential
    mul_var potential index_flt
    add_var potential center_start
    #
    # Send the bin center potential and dt / dE for that potential.
    pck_start
      pck_add potential
      pck_add bins[index]
    pck_end
```


PalmSens

```
    add_var index 1i
endloop
on_finished:
cell_off
```

Example output

```
L
Pab807A900u,200;eg80F42FFn,10,200
Pab807BC87u,200;eg80F42FFn,10,200
Pab807D00Eu,200;eg80F42FFn,10,200
...
Pab81CA312u,200;eg80C3599n,10,200
Pab81CB698u,200;eg80F42FFn,10,200
Pab81CCA1Fu,200;eg80C3599n,10,200
+
```

15.7. 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 ack
# loop counter
var i
var temperature
var time
# 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 ack 0i ja
# Read and check device ID.
store_var w[0i] 0x0B aa
i2c_write_read 0x48 w 1i r 1i ack
if ack != 0i
  abort
```

```
endif
if r[0i] != 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).
store_var w[0i] 0x03 aa
store_var w[1i] 0x80 aa
i2c_write 0x48 w 2i ack
if ack != 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.
  store_var w[0i] 0x02 aa
  store_var r[0i] 0x80 ja
  loop r[0i] & 0x80
    i2c_write_read 0x48 w 1i r 1i ack
    if ack != 0i
      abort
    endif
  endloop
  # Read timer.
  timer_get time
  # Read temperature value.
  store_var w[0i] 0x00 aa
  i2c_write_read 0x48 w 1i r 2i ack
  if ack != 0i
    abort
  endif
  # Convert temperature.
  # Store MSB
  copy_var r[0i] temperature
  # Combine MSB + LSB in one variable.
  bit_lsl_var temperature 8i
  bit_or_var temperature r[1i]
  # Handle negative temperatures.
  if temperature & 0x8000
    sub_var temperature 65536i
  endif
  # Convert to float and divide by 128 to get temperature in degrees Celsius.
  int_to_float temperature
  div_var temperature 128
  pck_start
  pck_add time
  pck_add temperature
```

```
  pck_end
  add_var i 1i
endloop
on_finished:
if ack == 1i
  send_string "ERROR: I2C address NACK"
elseif ack == 2i
  send_string "ERROR: I2C data NACK"
elseif ack == 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
```

15.8. 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 ack
store_var ack 0i ja
var i
store_var i 0i ja
array r 7
array w 7
# Year = '97
store_var w[0i] 0xE9 aa
# Month = August
store_var w[1i] 0x10 aa
# Day = 29
store_var w[2i] 0x94 aa
# Day of week = friday
store_var w[3i] 0xA0 aa
```

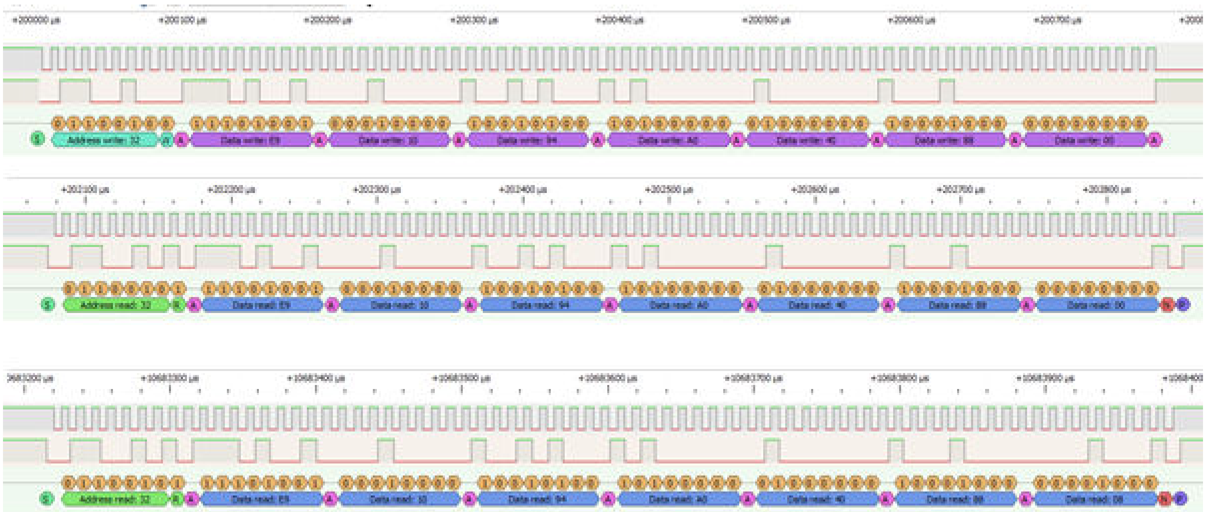
```
# Hour = 2 AM
store_var w[4i] 0x40 aa
# Minute = 14
store_var w[5i] 0x88 aa
# Seconds = 0
store_var w[6i] 0x00 aa
# Configure I2C GPIOs and set it to 100 kHz clock, 7-bit address
set_gpio_cfg 0x0300i 2
i2c_config 100k 7
# Write data to real-time data registers
i2c_write 0x32i w 7i ack
# Printing the time as it was written.
i2c_read 0x32i r 7i ack
store_var i 0i ja
loop i < 7i
    pck_start
    pck_add r[i]
    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 ack
store_var i 0i ja
loop i < 7i
    pck_start
    pck_add r[i]
    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.



15.9. 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 ack
# Loop variable
var i
# Temporary value
var v
# Write array, 2 bytes address + 32 bytes data
array w 34
# Read array, 32 bytes data
array r 32
# 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
store_var w[0i] 1i aa
store_var w[1i] 64i aa
# 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
    copy_var v w[i]
    add_var i 1i
    add_var v 1i
endloop
# Write to device
store_var ack 0i ja
i2c_write 0x50i w 34i ack
# Handle ACK/NACK
if ack != 0i
    send_string "FAILED to write to EEPROM"
    abort
endif
# Read EEPROM. Will generate NACK until write is completed.
# Variable ack is set to 1 to enter the loop.
store_var ack 1i ja
loop ack != 0i
    # Reset var ack so I2C will not fail when receiving NACK
    store_var ack 0i ja
    # Note the address from the write array is reused
    i2c_write_read 0x50i w 2i r 32i ack
    send_string "reading EEPROM"
endloop
# Print the received data
store_var i 0i ja
loop i < 32i
    pck_start
    pck_add r[i]
    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 (FCV) measurement 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`)

PalmSens

Version 1.5 Rev 1

- Increased array size on EmStat4 from 32768 to 50000 variables
- Added new MethodSCRIPT commands:
  - Mux commands: `mux_config`, `mux_get_channel_count`, and `mux_set_channel`
  - AC Voltammetry (ACV) measurement technique: `meas_loop_acv`
  - Multi-Sine EIS (MSEIS) measurement technique: `meas_ms_eis`
  - Fast CA (FCA) measurement technique: `meas_fast_ca`
  - Alternating mux measurement techniques:
    - CA: `meas_loop_ca_alt_mux`
    - CP: `meas_loop_cp_alt_mux`
    - OCP: `meas_loop_ocp_alt_mux`
  - iR compensation: `set_ir_comp`
  - Modulo operation: `mod_var`
  - Alter the *VarType* of a MethodSCRIPT variable: `alter_vartype`
  - Output user notifications using the device LED: `notify_led`
  - Set scan direction for Cyclic Voltammetry (CV): `set_scan_dir`
- Added support for interpolated strings (*f-strings*), see Section 8.7.1, “Interpolated strings”
- Added support for array access syntax, see Section 8.2.1, “Array Access Syntax”
- Added support for auto-incrementing number in file, see Section 14.12.4, “file_open”
- Added support for multicharacter variable names, see Section 8.1, “*var*”
- Updated error codes var types
- Fixed example scripts in chapter 3
- Updated `eis_opt` command to support fast fixed frequency EIS measurements
- Command `set_autoranging` now responds with an error when given negative inputs
- Added missing galvanostatic mode in Chapter 12, *PGStat modes* and clarified Section B.1, “PGStat mode properties”

Version 1.6 Rev 1

- Added new MethodSCRIPT commands:
  - Added `smooth` command, enabling data smoothing within MethodSCRIPT
  - Added `peak_detect` command, enabling peak searching within MethodSCRIPT
  - Added `rtc_get` command, enabling RTC date and time to be retrieved within MethodSCRIPT
- Added explanation of noise level in chapter Section 5.2, “Variable sub package format”
- Fixed wrong factors in Table 1, “SI prefix conversion table”
- Updated examples to use new MethodSCRIPT features
- Updated inconsistent PAD output potential variable documentation
- Updated hibernate command documentation for the EmStat Pico to reflect firmware changes
- Added Sensit Wearable
- Added Nexus

Version 1.6 Rev 2

- Fixed wrong `VT_POTENTIAL_AIN1` ID for the Sensit Wearable.

Version 1.7 Rev 1

- Easier way of using bipot:
  - Added command `set_bipot_mode` (replaces `set_poly_we_mode`, which had to be used from the bipot channel)
  - Added command `set_bipot_potential` (replaces `set_e`, which had to be used from the bipot channel)
  - Added optional argument `add_meas`
  - Added new *VarTypes*, such as `bb` (bipot current)
  - Deprecated command `set_poly_we_mode` and PGStat mode 5 (poly_we) in favor of the new `set_bipot_mode` command
  - Deprecated optional argument `poly_we` in favor of `add_meas`
- Added dual EIS measurement technique (Nexus only):
  - Added `meas_loop_eis_dual` MethodSCRIPT command.
  - Added optional arguments `eis_dual_acdc` and `eis_dual_tdd`
- Added `battery_perc` MethodSCRIPT command
- Added `beep` MethodSCRIPT command
- Added optional argument `filter_type`
- Added optional argument `ocp`
- Added `pow_var` MethodSCRIPT command
- Added `get_progress` MethodSCRIPT command

- The EmStat Pico now uses the external Ablic S-35390A RTC for its system date and time if enabled in the peripheral configuration register.

Version 1.8 Rev 1

- Added `meas_scp` MethodSCRIPT command
- Added `trim_enable` MethodSCRIPT command
- Added `linear_fit` MethodSCRIPT command
- Added `subarray` MethodSCRIPT command
- Added `mean` MethodSCRIPT command
- Added `log_var` MethodSCRIPT command
- Added display commands for the EmStat4T
  - display_btns
  - display_clear
  - display_draw
  - display_icon
  - display_inp_num
  - display_keyboard
  - display_progress
  - display_scroll_add
  - display_scroll_get
  - display_text
- Added qr scan for the EmStat4T
  - With new optional command qr_log

Appendix A: Error codes

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

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 13. 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
0x0019	Authentication error
0x001A	Calibration invalid
0x001B	This command or part of this command is not supported by the current device
0x001C	Step Potential must at least 1 DAC LSB for this technique

Error code	Description
0x001D	Pulse Potential must at least 1 DAC LSB for this technique
0x001E	Amplitude must at least 1 DAC LSB this technique
0x001F	Product is not licensed for this technique
0x0020	Cannot have more than one high speed and/or max range mode enabled
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.
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

Error code	Description
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	The multiplexer was not found.
0x0047	The filesystem has to be mounted to complete this action.
0x0048	This device is not a multi-device, no serial available.
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 selected GPIO pin mode is not allowed (by NVM config or device type).
0x0057	The on-board flash module has timed out.
0x0058	Timing error during fast measurement (possibly caused by communication).
0x005A	The instrument cannot meet the requested measurement timing.
0x005B	The variable type is already being measured.
0x006D	The COMM command expected an hexadecimal value, but received something else.
0x006E	The COMM command expected a decimal value, but received something else.
0x0071	The provided key does not fit the lock on this register.
0x0072	I2C port expander did not acknowledge a command

Error code	Description
0x0073	Filesystem module not supported
0x0074	The IP address is not available (yet).
0x007A	There is no measurement channel left for the requested measurement.
0x007B	Temperature measurements during EIS with > 8 kHz are not supported.
0x007C	The specified mode is unknown
0x007D	The ADXL367 did not acknowledge an I2C command
0x007E	An unexpected error occurred during an I2C operation.
0x007F	I2C bus timeout during I2C operation (probably caused by I2C target device).
0x0080	The CE is oscillating.
0x0082	Operation requires system warnings to be cleared.
0x0083	Filesystem operations are not supported on this device.
0x0084	The requested variable type does not support ranging.
0x0085	The selected GPIO pin does not support harware synchronisation.
0x0086	Hardware select must be disabled before the `role` pin can be disabled.
0x0087	The `role` pin must be configured before the hadware select can be enabled
0x0088	The instrument has reserved this GPIO pin to be controlled by hardware (e.g. file system or HW-sync).
0x0089	This GPIO pin cannot be unlocked, as it was not locked in the first place
0x008A	This GPIO pin can only be used for interfacing with a specific external memory
0x008B	The BiPot should be disabled.
0x008C	iR compensation should be disabled.
0x008D	The key provided for the reset command is incorrect.
0x008E	The SPI interface is not confgiured while it is required for the filesystem
0x008F	The SPI interface requires the SPI pins to be configured to 'peri 1'
0x0091	The GPIO is locked for a multiplexer (e.g. Mux8R2 or PicoMux)
0x0092	The GPIO is locked for external storage (e.g. SD-card or NAND flash)
0x0093	The GPIO is locked for an external LED
0x0094	The GPIO is locked for hardware synchronisation
0x0095	The GPIO is locked for external PGStat signals
0x0096	The GPIO is locked for some special purpose on this instrument
0x0097	An attempt was made to access a GPIO using a key while it is unlocked
0x0098	The configuration set using the Peripheral configuration (0x01) register is invalid

Error code	Description
0x0099	Filesystem file is corrupt
0x009A	Filesystem failed to format
0x009B	Filesystem I/O error
0x009C	Filesystem didn't have enough memory to perform the operation
0x009D	Filesystem path was too long to handle
0x009E	Filesystem the path was not valid
0x009F	Filesystem could not find the file specified
0x00A0	Filesystem FM not supported
0x00A1	Filesystem doesn't have a listing
0x00A2	Filesystem is not initialized
0x00A3	Filesystem file is open, but it should not have been
0x00A4	Filesystem file is not open
0x00A5	Filesystem does not support this feature
0x00A6	Filesystem expected something which is not true.
0x00A7	Filesystem could not find the path
0x00A8	Access denied due to prohibited access or directory full
0x00A9	The file/directory object is invalid
0x00AA	The physical drive is write protected
0x00AB	The logical drive number is invalid
0x00AC	There is no valid filesystem volume
0x00AD	The format operation was aborted due to any problem
0x00AE	The operation is rejected according to the file sharing policy
0x00AF	Working buffer could not be allocated
0x00B0	Too many files opened at once by filesystem
0x00B1	Parameter given to the filesystem is invalid
0x00B2	The file mode is invalid (should be readonly, new, append, overwrite).
0x00B3	The pin mode requred for the LED mapping is not allowed for this pin
0x00B4	The pin mode requred for the HW-sync role is not allowed for this pin
0x00B5	The pin mode requred for the HW-sync start mapping is not allowed for this pin
0x00B6	Use of the encrypted filesystem failed
0x00B7	The user key is not in a valid state for this cmmand

Error code	Description
0x00B8	The communications protocol is not in valid lock state for this command
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
0x4017	Arrays must be the same size but are not
0x4018	The script has ended unexpectedly.
0x4019	The script command is only valid for a multichannel (combined) device
0x401A	The script command 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.
0x401D	The file path is too long for the file system.
0x401E	Insufficient memory to store array index
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
0x4024	The resistance value is to big for the whole autorange range
0x4025	The resistance value is to big for current current range

PalmSens

Error code	Description
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` command
0x4029	The technique requires that at least one step should be made
0x402A	The variable names do not fit in memory anymore, try using shorter names.
0x402B	The variable name did not start with 'a'-'z' or otherwise contained anything other than 'a'-'z', '0'-'9' and '_'.
0x402C	The variable name is too long to be processed.
0x402D	The file mode is invalid.
0x402E	The file mode does not support a counter in the file path.
0x402F	The file path with the maximum counter value already exists.
0x4030	There are too many files open already.
0x4031	The specified multi device type is not defined.
0x4032	Cannot set the potential (or potential range) within the active measurement loop.
0x4033	Cannot set the current (or current range) within the active measurement loop.
0x4034	The used feature is not licensed on this product.
0x4035	The given filter type is unknown or not supported.
0x4036	The given command is only allowed within measurement loops.
0x4037	A computation has resulted in an overflow
0x4038	The array access was not correctly formed
0x4039	The literal argument was not correctly formed
0x403A	The subarray declaration was out of bounds for the source array
0x403B	A file needs to be opened before it can be written to
0x403C	The MethodScript output mode is unknown
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

Error code	Description
0x4208	MScript argument reference was invalid (not 'a' - 'z')
0x4209	MScript argument variable type is invalid or 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
0x4210	An f-string contains an opening brace that is never closed
0x4211	MScript argument is an array element, which is not supported by this command
0x7FFF	A fatal error has occurred, the device must be reset

Appendix B: Device-specific information

For more information, please consult the instrument datasheet.

B.1. PGStat mode properties

This section shows the most important changes in specifications depending on the selected PGStat mode. See Chapter 12, *PGStat modes* for a description of all PGStat modes.

B.1.1. Nexus

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

Table 14. Potentiostat mode properties for Nexus.

Parameter	Min. value	Max. value
Bandwidth	-	1 MHz
Potential range	-10.0 V	10.0 V
Dynamic potential window	-10.0 V	10.0 V

Table 15. Galvanostat mode properties for Nexus.

Parameter	Min. value	Max. value
Bandwidth	-	1 MHz
Current range	-1.0 A	1.0 A

B.1.2. EmStat4 HR

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

Table 16. 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 17. Galvanostat mode properties for EmStat4 HR.

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

B.1.3. EmStat4 LR

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

Table 18. 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 19. Galvanostat mode properties for EmStat4 LR.

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

B.1.4. EmStat Pico

Table 20. 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 21. 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 22. 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.1.5. Sensit Wearable

Table 23. Sensit Wearable 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 24. Sensit Wearable 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 25. Sensit Wearable 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 26. Nexus potentiostatic EIS properties.

Parameter	Value
Max. amplitude (V_{RMS})	0.300 V
Max. frequency	1 MHz

Table 27. EmStat4 potentiostatic EIS properties.

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

Table 28. 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 29. EmStat Pico potentiostatic EIS properties.

Parameter	Value
Max. amplitude (V_{RMS})	0.429 V
Max. frequency	200 kHz

Table 30. Sensit Wearable potentiostatic EIS properties.

Parameter	Value
Max. amplitude (V_{RMS})	0.429 V
Max. frequency	200 kHz

B.3. Current ranges

B.3.1. Nexus

Table 31. Nexus potentiostat current ranges.

Index	Range name	Underload	Overload warning	Overload	Maximum
0x10	100pA	20.48 pA	409.6 pA	486.4 pA	512 pA
0x00	1nA	204.8 pA	4.096 nA	4.864 nA	5.12 nA
0x01	10nA	2.048 nA	40.96 nA	48.64 nA	51.2 nA
0x02	100nA	20.48 nA	409.6 nA	486.4 nA	512 nA
0x03	1uA	204.8 nA	4.096 µA	4.864 µA	5.12 µA
0x04	10uA	2.048 µA	40.96 µA	48.64 µA	51.2 µA
0x05	100uA	20.48 µA	409.6 µA	486.4 µA	512 µA
0x06	1mA_tia	204.8 µA	4.096 mA	4.864 mA	5.12 mA
0x07	10mA_tia	2.048 mA	40.96 mA	48.64 mA	51.2 mA
0x08	1mA	204.8 µA	4.096 mA	4.864 mA	5.12 mA
0x09	10mA	2.048 mA	40.96 mA	48.64 mA	51.2 mA
0x0a	100mA	20.48 mA	409.6 mA	486.4 mA	512 mA
0x0b	1A	204.8 mA	-	-	1.1 A

Table 32. Nexus BiPot current ranges.

Index	Range name	Underload	Overload warning	Overload	Maximum
0x10	100pA	20.55 pA	411 pA	488 pA	513.7 pA

PalmSens

Index	Range name	Underload	Overload warning	Overload	Maximum
0x00	1nA	205.5 pA	4.11 nA	4.88 nA	5.137 nA
0x01	10nA	2.055 nA	41.1 nA	48.8 nA	51.37 nA
0x02	100nA	20.55 nA	411 nA	488 nA	513.7 nA
0x03	1uA	205.5 nA	4.11 µA	4.88 µA	5.137 µA
0x04	10uA	2.055 µA	41.1 µA	48.8 µA	51.37 µA
0x05	100uA	20.55 µA	411 µA	488 µA	513.7 µA
0x06	1mA	205.5 µA	4.11 mA	4.88 mA	5.137 mA
0x07	10mA	2.055 mA	41.1 mA	48.8 mA	51.37 mA

Table 33. Nexus galvanostat current ranges.

Index	Current range name
0x20	1 nA
0x21	10 nA
0x22	100 nA
0x23	1 uA
0x24	10 uA
0x25	100 uA
0x26	1 mA (tia)
0x27	10 mA (tia)
0x28	1 mA
0x29	10 mA
0x2A	100 mA
0x2B	1 A

B.3.2. EmStat4 LR

Table 34. EmStat4 LR potentiostat current ranges.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x03	1 nA	123 pA	2.46 nA	2.92 nA	3 nA
0x06	10 nA	1.23 nA	24.6 nA	29.2 nA	30 nA
0x09	100 nA	12.3 nA	246 nA	292 nA	300 nA
0x0C	1 µA	123 nA	2.46 µA	2.92 µA	3 µA

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x0F	10 µA	1.23 µA	24.6 µA	29.2 µA	30 µA
0x12	100 µA	12.3 µA	246 µA	292 µA	300 µA
0x15	1 mA	123 µA	2.46 mA	2.92 mA	3 mA
0x18	10 mA	1.23 mA	24.6 mA	29.2 mA	30 mA

Table 35. EmStat4 LR galvanostat current ranges.

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

B.3.3. EmStat4 HR

Table 36. EmStat4 HR potentiostat current ranges.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x09	100 nA	12.3 nA	246 nA	292 nA	300 nA
0x0C	1 µA	123 nA	2.46 µA	2.92 µA	3 µA
0x0F	10 µA	1.23 µA	24.6 µA	29.2 µA	30 µA
0x12	100 µA	12.3 µA	246 µA	292 µA	300 µA
0x15	1 mA	123 µA	2.46 mA	2.92 mA	3 mA
0x18	10 mA	1.23 mA	24.6 mA	29.2 mA	30 mA
0x1B	100 mA	12.3 mA	246 mA	292 mA	200 mA

Table 37. EmStat4 HR galvanostat current ranges.

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

B.3.4. EmStat Pico

Table 38. EmStat Pico low speed mode.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x0	100 nA	2.44 nA	48.9 nA	58.1 nA	60.0 nA
0x1	1.95 µA	35.1 nA	956 nA	1.13 µA	1.17 µA
0x2	3.91 µA	95.6 nA	1.91 µA	2.27 µA	2.34 µA
0x3	7.81 µA	191 nA	3.82 µA	4.54 µA	4.68 µA
0x4	15.63 µA	382 nA	7.65 µA	9.08 µA	9.38 µA
0x5	31.25 µA	764 nA	15.3 µA	1.82 µA	18.7 µA
0x6	62.5 µA	1.53 µA	30.6 µA	36.3 µA	37.5 µA
0x7	125 µA	3.06 µA	61.2 µA	72.6 µA	75.0 µA
0x8	250 µA	6.12 µA	122 µA	145 µA	150 µA
0x9	500 µA	12.2 µA	245 µA	291 µA	300 µA
0xA	1 mA	24.4 µA	489 µA	581 µA	600 µA
0xB	5 mA	122 µA	2.45 mA	2.91 mA	3.00 mA

Table 39. EmStat Pico high speed mode.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x80	100 nA	2.44 nA	48.9 nA	58.1 nA	60.0 nA
0x81	1 µA	24.4 nA	489 nA	581 nA	600 nA
0x82	6.25 µA	153 nA	3.06 µA	3.63 µA	3.75 µA
0x83	12.5 µA	306 nA	6.12 µA	7.26 µA	7.50 µA
0x84	25 µA	612 nA	12.2 µA	14.5 µA	15.0 µA
0x85	50 µA	1.22 µA	24.5 µA	29.1 µA	30.0 µA
0x86	100 µA	2.44 µA	48.9 µA	58.1 µA	60.0 µA

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x87	200 µA	4.90 µA	97.9 µA	116 µA	120 µA
0x88	1 mA	24.4 µA	489 µA	581 µA	600 µA
0x89	5 mA	122 µA	2.45 mA	2.91 mA	3.00 mA

Table 40. EmStat Pico max range mode.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x80	100 nA	2.44 nA	48.9 nA	58.1 nA	60.0 nA
0x81	1 µA	24.4 nA	489 nA	581 nA	600 nA
0x82	6.25 µA	153 nA	3.06 µA	3.63 µA	3.75 µA
0x83	12.5 µA	306 nA	6.12 µA	7.26 µA	7.50 µA
0x84	25 µA	612 nA	12.2 µA	14.5 µA	15.0 µA
0x85	50 µA	1.22 µA	24.5 µA	29.1 µA	30.0 µA
0x86	100 µA	2.44 µA	48.9 µA	58.1 µA	60.0 µA
0x87	200 µA	4.90 µA	97.9 µA	116 µA	120 µA
0x88	1 mA	24.4 µA	489 µA	581 µA	600 µA
0x89	5 mA	122 µA	2.45 mA	2.91 mA	3.00 mA

B.3.5. Sensit Wearable

Table 41. Sensit Wearable low speed mode.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x0	100 nA	2.44 nA	48.9 nA	58.1 nA	60.0 nA
0x1	1.95 µA	35.1 nA	956 nA	1.13 µA	1.17 µA
0x2	3.91 µA	95.6 nA	1.91 µA	2.27 µA	2.34 µA
0x3	7.81 µA	191 nA	3.82 µA	4.54 µA	4.68 µA
0x4	15.63 µA	382 nA	7.65 µA	9.08 µA	9.38 µA
0x5	31.25 µA	764 nA	15.3 µA	1.82 µA	18.7 µA
0x6	62.5 µA	1.53 µA	30.6 µA	36.3 µA	37.5 µA
0x7	125 µA	3.06 µA	61.2 µA	72.6 µA	75.0 µA
0x8	250 µA	6.12 µA	122 µA	145 µA	150 µA
0x9	500 µA	12.2 µA	245 µA	291 µA	300 µA

Index	Current range name	Underload	Overload warning	Overload	Maximum
0xA	1 mA	24.4 µA	489 µA	581 µA	600 µA
0xB	5 mA	122 µA	2.45 mA	2.91 mA	3.00 mA

Table 42. Sensit Wearable high speed mode.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x80	100 nA	2.44 nA	48.9 nA	58.1 nA	60.0 nA
0x81	1 µA	24.4 nA	489 nA	581 nA	600 nA
0x82	6.25 µA	153 nA	3.06 µA	3.63 µA	3.75 µA
0x83	12.5 µA	306 nA	6.12 µA	7.26 µA	7.50 µA
0x84	25 µA	612 nA	12.2 µA	14.5 µA	15.0 µA
0x85	50 µA	1.22 µA	24.5 µA	29.1 µA	30.0 µA
0x86	100 µA	2.44 µA	48.9 µA	58.1 µA	60.0 µA
0x87	200 µA	4.90 µA	97.9 µA	116 µA	120 µA
0x88	1 mA	24.4 µA	489 µA	581 µA	600 µA
0x89	5 mA	122 µA	2.45 mA	2.91 mA	3.00 mA

Table 43. Sensit Wearable max range mode.

Index	Current range name	Underload	Overload warning	Overload	Maximum
0x80	100 nA	2.44 nA	48.9 nA	58.1 nA	60.0 nA
0x81	1 µA	24.4 nA	489 nA	581 nA	600 nA
0x82	6.25 µA	153 nA	3.06 µA	3.63 µA	3.75 µA
0x83	12.5 µA	306 nA	6.12 µA	7.26 µA	7.50 µA
0x84	25 µA	612 nA	12.2 µA	14.5 µA	15.0 µA
0x85	50 µA	1.22 µA	24.5 µA	29.1 µA	30.0 µA
0x86	100 µA	2.44 µA	48.9 µA	58.1 µA	60.0 µA
0x87	200 µA	4.90 µA	97.9 µA	116 µA	120 µA
0x88	1 mA	24.4 µA	489 µA	581 µA	600 µA
0x89	5 mA	122 µA	2.45 mA	2.91 mA	3.00 mA

B.4. Potential ranges

Table 44. Nexus galvanostat potential ranges.

PalmSens

Index	Range name	Underload	Overload warning	Overload	Maximum
0x00	1V	409.6 mV	8.192 V	9.728 V	10.24 V
0x01	100mV	40.96 mV	819.2 mV	972.8 mV	1.024 V
0x02	10mV	4.096 mV	81.92 mV	97.28 mV	102.4 mV
0x03	1mV	409.6 µV	8.192 mV	9.728 mV	10.24 mV

Table 45. EmStat4 LR galvanostat potential ranges.

Index	Potential range name	Underload	Overload warning	Overload	Maximum
0	10 mV	1.26 mV	25.2 mV	29.9 mV	30 mV
1	20 mV	2.52 mV	50.4 mV	59.9 mV	60 mV
2	50 mV	6.30 mV	126 mV	150 mV	150 mV
3	100 mV	12.6 mV	252 mV	299 mV	300 mV
4	200 mV	25.2 mV	504 mV	599 mV	600 mV
5	500 mV	63.0 mV	1.26 V	1.50 V	1.5 V
6	1 V	126 mV	2.52 V	2.99 V	3 V

Table 46. EmStat4 HR galvanostat potential ranges.

Index	Potential range name	Underload	Overload warning	Overload	Maximum
0	10 mV	2.53 mV	50.6 mV	60.1 mV	60 mV
1	20 mV	5.06 mV	101 mV	120 mV	120 mV
2	50 mV	12.7 mV	253 mV	300 mV	300 mV
3	100 mV	25.3 mV	506 mV	601 mV	600 mV
4	200 mV	50.6 mV	1.01 V	1.20 V	1.2 V
5	500 mV	127 mV	2.53 V	3.00 V	3 V
6	1 V	253 mV	5.06 V	6.01 V	6 V

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

Table 47. Supported variable types Nexus.

ID	Name
`ab`	`VT_POTENTIAL`
`ac`	`VT_POTENTIAL_CE`
`ag`	`VT_POTENTIAL_WE_VS_CE`

ID	Name
`as`	`VT_POTENTIAL_AIN0`
`ah`	`VT_POTENTIAL_S2_VS_RE`
`ai`	`VT_POTENTIAL_SE_V2_S2`
`ba`	`VT_CURRENT`
`bb`	`VT_CURRENT_BIPOT`
`ed`	`VT_TEMPERATURE`
`ef`	`VT_TEMPERATURE_BOARD`

Table 48. Supported variable types EmStat4.

ID	Name
`ab`	`VT_POTENTIAL`
`ac`	`VT_POTENTIAL_CE`
`ae`	`VT_POTENTIAL_RE`
`ag`	`VT_POTENTIAL_WE_VS_CE`
`as`	`VT_POTENTIAL_AIN0`
`ba`	`VT_CURRENT`

Table 49. Supported variable types EmStat Pico.

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

Table 50. Supported variable types Sensit Wearable.

ID	Name
`ab`	`VT_POTENTIAL`
`ac`	`VT_POTENTIAL_CE`
`ae`	`VT_POTENTIAL_RE`
`ag`	`VT_POTENTIAL_WE_VS_CE`
`at`	`VT_POTENTIAL_AIN1`

ID	Name
`ba`	`VT_CURRENT`
`ef`	`VT_TEMPERATURE_BOARD`

`VT_POTENTIAL_AIN1` should not be used on a standard Sensit Wearable. However if the device has been customised to use an external NTC, `VT_POTENTIAL_AIN1` measures the voltage on that NTC. In such cases, the `VT_TEMPERATURE_BOARD` parameter is no longer useful.
For further information on customising the NTC, please contact PalmSens directly.

B.6. Device I/O pin configurations

Table 51. 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 52. 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
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 that is always available, such as the Sensit BT, GPIO1–4 are reserved and cannot be used as general-purpose I/O pins.
◊ Using the I2C lines as digital I/O is strongly discouraged and will disable the instrument's internal and external I2C bus along with the on board temperature sensor.

Table 53. Sensit Wearable I/O pin configuration.

Bitmask	Pin name	Mode 0	Mode 1	Mode 2
0x0002	SPI_MISO	Not available	Not available	SPI flash memory*
0x0004	SPI_CLK	Not available	Not available	SPI flash memory*
0x0008	SPI_MOSI	Not available	Not available	SPI flash memory*
0x0010	SPI_CS0	Not available	Not available	SPI flash memory*
0x0080	GPIO7_WAKE	Digital Input	Not available	Wake from sleep (Active low)
0x0100	I2C_SCL	Not available	Not available	I²C*
0x0200	I2C_SDA	Not available	Not available	I²C*

* It is not necessary to configure the SPI or I2C pins on the Sensit Wearable, they are always assigned to SPI and I2C.

B.7. Measurement channels

Some instruments (such as the Nexus) are capable of measuring multiple channels simultaneously. These parallel input channels can be added to another MethodSCRIPT command using the `add_meas()` optional argument.

Table 54. Nexus signal distribution

Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
VT_CURRENT	VT_POTENTIAL	VT_CURRENT_BIPOT	VT_POTENTIAL_CE	VT_TEMPERATURE	VT_TEMPERATURE_BOARD	VT_POTENTIAL_AIN0
	VT_POTENTIAL_RE	VT_POTENTIAL_S2_VS_RE	VT_POTENTIAL_WE_VS_CE			
		VT_POTENTIAL_SE_VS_S2				

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 55. 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_S2_VS_RE`	`ah`	Measured second sense voltage vs RE.
`VT_POTENTIAL_SE_VS_S2`	`ai`	Measured SE voltage vs second sense.
`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_CURRENT_BIPOT`	`bb`	Measured WE2 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_ZREAL_BIPOT`	`cl`	Measured real part of complex bipot impedance
`VT_ZIMAG_BIPOT`	`cm`	Measured imaginary part of complex bipot impedance

Name	ID	Description
`VT_ZREAL_S2_VS_RE`	`cn`	Measured real part of complex second sense impedance
`VT_ZIMAG_S2_VS_RE`	`co`	Measured imaginary part of complex second sense impedance
`VT_ZREAL_SE_VS_S2`	`cp`	Measured real part of complex second sense impedance
`VT_ZIMAG_SE_VS_S2`	`cq`	Measured imaginary part of complex second sense impedance
`VT_EIS_TDD_BIPOT`	`cr`	Measured bipot current time domain data
`VT_EIS_TDD_S2_VS_RE`	`cs`	Measured S2 vs RE potential time domain data
`VT_EIS_TDD_SE_VS_S2`	`ct`	Measured SE vs S2 potential time domain data
`VT_EIS_AC_BIPOT`	`cu`	Measured AC bipot current
`VT_EIS_DC_BIPOT`	`cv`	Measured DC bipot current
`VT_EIS_AC_S2_VS_RE`	`cw`	Measured AC S2 vs RE potential
`VT_EIS_DC_S2_VS_RE`	`cx`	Measured DC S2 vs RE potential
`VT_EIS_AC_SE_VS_S2`	`cy`	Measured AC SE vs S2 potential
`VT_EIS_DC_SE_VS_S2`	`cz`	Measured DC SE vs S2 potential
`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_CHANNEL`	`ea`	The channel
`VT_TIME`	`eb`	Time in seconds
`VT_PIN_MSK`	`ec`	Binary pin bitmask, indicating which pins are high / low
`VT_TEMPERATURE`	`ed`	CPU temperature in degrees Celsius
`VT_COUNT`	`ee`	Count (e.g. number of data points)
`VT_TEMPERATURE_BOARD`	`ef`	Board temperature in degrees Celsius
`VT_DT_DE`	`eg`	Time vs potential derivative, dt/dE in s/V.
`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

Name	ID	Description
`VT_POTENTIAL_GENERIC4`	`id`	Generic potential 4
`VT_MISC_GENERIC1`	`ja`	Miscellaneous value 1 (reserved for user code)
`VT_MISC_GENERIC2`	`jb`	Miscellaneous value 2 (reserved for user code)
`VT_MISC_GENERIC3`	`jc`	Miscellaneous value 3 (reserved for user code)
`VT_MISC_GENERIC4`	`jd`	Miscellaneous value 4 (reserved for user code)