
SPECpy Documentation

Release 1

Lakshmipriya Sukumar and Brian Toby

October 25, 2012

CONTENTS

1	<i>Module spec: SPEC-like emulation</i>	3
1.1	<i>Motor interface routines</i>	3
1.2	<i>Scaler routines</i>	3
1.3	<i>Other routines in module spec</i>	4
1.4	<i>Global variables</i>	4
1.5	<i>Complete Function Descriptions</i>	5
2	<i>Module macros: Additional SPEC-like emulation</i>	13
2.1	<i>Logging</i>	13
2.2	<i>Plotting</i>	14
2.3	<i>Macros specific to 1-ID</i>	15
2.4	<i>Complete Function Descriptions</i>	16
	Index	25

Note that this package requires the Python NumPy and PyEpics packages be installed order to control an instrument. However, if PyEpics is not installed, all routines documented here can still be run. However, in this case EPICS interactions will be simulated and print statements will report what the Python code is attempting to do. Likewise, if PyEpics is installed, but `:func:~spec.EnableEPICS` is not called (or is called with a value of False), again no communication with EPICS is attempted. This allows scripts to be developed and tested without access to the instrument.

MODULE SPEC: SPEC-LIKE EMULATION

The Python functions listed below are designed to emulate similar commands/macros in SPEC.

1.1 Motor interface routines

Description	Relative	Absolute
move motor	<code>mvr ()</code>	<code>mv ()</code>
move motor with wait	<code>umvr ()</code>	<code>umv ()</code>
where is this motor?		<code>wm ()</code>
where are all motors?		<code>wa ()</code>

1.2 Scaler routines

description	command
start and readout scaler after completion	<code>ct ()</code>
start scaler and return	<code>count_em ()</code>
wait for scaler to complete	<code>wait_count ()</code>
read scaler	<code>get_counts ()</code>

1.3 Other routines in module spec

routine	Description
<code>sleep()</code>	Delay for a specified amount of time
<code>EnableEPICS()</code>	Turns simulation mode on or off
<code>ShowEnabled()</code>	Show if use of use of EPICS is available
<code>DefineMtr()</code>	Define a motor to be accessed
<code>GetMtrInfo()</code>	Retrieves all motor info from a key
<code>DefineScaler()</code>	Define a scaler to be accessed
<code>GetScalerInfo()</code>	Retrieves all scaler info from an index
<code>ListMtrs()</code>	Returns a list of motor symbols
<code>Sym2MtrVal()</code>	Retrieves the motor entry key from a symbol
<code>ExplainMtr()</code>	Retrieves the motor description from a key or symbol
<code>ReadMtr()</code>	Returns the motor position from a key
<code>PositionMtr()</code>	Moves a motor
<code>GetScalerLastCount()</code>	Returns the last set of counts that have been read for a scaler
<code>GetScalerLastTime()</code>	Returns the counting time for the last use of a scaler
<code>GetScalerLabels()</code>	Returns the labels that have been retrieved for a scaler
<code>SetMon()</code>	Set the monitor channel for the scaler
<code>GetMon()</code>	Return the monitor channel for the scaler
<code>SetDet()</code>	Set the main detector channel for the scaler
<code>GetDet()</code>	Return the main detector channel for the scaler
<code>setCOUNT()</code>	Sets the default counting time
<code>initElapsed()</code>	Initialize the elapsed time counter
<code>setElapsed()</code>	Update the elapsed time counter
<code>setRETRIES()</code>	Sets the maximum number of EPICS retries
<code>setDEBUG()</code>	Sets debugging mode (printing lots of stuff) on or off

1.4 Global variables

`COUNT` defines the default counting time (sec) when `ct` is called without an argument. Defaults to 1 sec. Use `setCOUNT()` to set this when using `from spec import *`, as setting the variable directly has problems:

This will sort-of work:

```
>>> from spec import *
>>> import spec
>>> spec.COUNT=3
```

however, `COUNT` in the local namespace will still have the old value.

but this will not work:

```
>>> from spec import *
>>> COUNT=3
```

This fails because the local copy of `COUNT` gets replaced, but the copy of `COUNT` actually in the `spec` module is left unchanged.

S `S` is a list that contains the last count values measured during the last call to `ct()` or `get_counts()`.

MAX_RETRIES Number of times to retry an EPICS operation (that are nominally expected to work on the first try) before generating an exception. Use `setRETRIES()` to set this or care when changing this (see comment on `COUNT`, in this section.)

DEBUG When set to True lots of print statements to be executed. Use for code development/testing. Use `setDEBUG()` to set this or care when changing this (see comment on `COUNT`, above in this section.)

ELAPSED Contains the time that has elapsed between when the spec module was loaded (or `initElapsed()` was called) and when `setElapsed()` was last called, which happens when motors are moved or counting is done.

1.5 Complete Function Descriptions

The functions available in this module are listed below.

`spec.DefineMtr(symbol, prefix, comment='')`

Define a motor for use in this module. Adds a motor to the motor table.

Parameters

- **symbol** (*string*) – a symbolic name for the motor. A global variable is defined in this module's name space with this name, This must be unique; exception `specException` is raised if a name is reused.
- **prefix** (*string*) – the prefix for the motor PV (`ioc:mnnn`). Omit the motor record field name (`.VAL`, etc.).
- **comment** (*string*) – a human-readable text field that describes the motor. Suggestion: include units and define the motion direction.

Returns key of entry created in motor table (str).

If you will use the “ `from <module> import *` ” python command to import these routines into the current module's name space, it is necessary to repeat this command after `DefineScaler()` to import the globals defined within in the top namespace:

Example (recommended for interactive use):

```
>>> from spec import *
>>> EnableEPICS()
>>> DefineMtr('mtrXX1', 'ioc1:mtr98', 'Example motor #1')
>>> DefineMtr('mtrXX2', 'ioc1:mtr99', 'Example motor #2')
>>> from spec import *
>>> mv(mtrXX1, 0.123)
```

Note that if the second `from ... import *` command is not used, the variables `mtrXX1` and `mtrXX2` cannot be accessed and the final command will fail.

Alternate example (this is a cleaner way to code scripts, since namespaces are not mixed):

```
>>> import spec
>>> spec.EnableEPICS()
>>> spec.DefineMtr('mtrXX1', 'ioc1:mtr98', 'Example motor #1')
>>> spec.DefineMtr('mtrXX2', 'ioc1:mtr99', 'Example motor #2')
>>> spec.mv(spec.mtrXX1, 0.123)
```

It is also possible to mix the two styles:

```
>>> import spec
>>> spec.EnableEPICS()
>>> spec.DefineMtr('mtrXX1', 'ioc1:mtr98', 'Example motor #1')
>>> spec.DefineMtr('mtrXX2', 'ioc1:mtr99', 'Example motor #2')
>>> from spec import *
>>> mv(mtrXX1, 0.123)
```

`spec.DefineScaler` (*prefix*, *channels=8*, *index=0*)

Defines a scaler to be used for this module

Parameters

- **prefix** (*string*) – the prefix for the scaler PV (ioc:mnnn). Omit the scaler record field name (.CNT, etc.)
- **channels** (*int*) – the number of channels associated with the scaler. Defaults to 8.
- **index** (*int*) – an index for the scaler, if more than one will be defined. The default (0) is used to define the scaler that will be used when `ct()` is called with one or no arguments.

Example (recommended for interactive use):

```
>>> from spec import *
>>> EnableEPICS()
>>> DefineScaler('idl:scaler1', 16)
>>> DefineScaler('idl:scaler2', index=1)
>>> ct()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

Alternate example (preferred for use in code):

```
>>> import spec as s
>>> s.EnableEPICS()
>>> s.DefineScaler('ioc1:3820:scaler1', 16)
>>> s.DefineScaler('ioc1:3820:scaler2', index=1)
>>> s.ct()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
>>> s.ct(index=1)
[1, 2, 3, 4, 5, 6, 7, 8]
```

`spec.EnableEPICS` (*state=True*)

Call to enable communication with EPICS.

If not called then the module will function in simulation mode only. If the PyEpics module cannot be loaded, then simulation will also be used.

Parameters *state* (*bool*) – if False is specified, then simulation mode is used (default value, True)

`spec.ExplainMtr` (*mtr*)

Show the description for a motor, as defined in `DefineMtr()`

Parameters *mtr* (*various*) – symbolic name for the motor, can take two forms: a motor key or a motor symbol.

Returns motor description (str) or '?' if not defined

`spec.GetDet` (*index=0*)

Return the main detector channel for the scaler or none if not defined. (See `SetDet()`) This is used for ASCAN, etc.

Parameters `index` (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns the channel number of the Detector

`spec.GetMon` (*index=0*)

Return the monitor channel for the scaler or none if not defined. (See `SetMon()`) This is used for counting on the Monitor.

Parameters `index` (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns the channel number of the Monitor

`spec.GetMtrInfo` (*mtr*)

Return a dictionary with motor information.

Parameters `mtr` (*str*) – a key corresponding to an entry in the motor table. If the value does not correspond to a motor entry, an exception is raised.

Returns dictionary with motor information

`spec.GetScalerInfo` (*index=0*)

returns information about a scaler based on the index

Parameters `index` (*int*) – an index for the scaler, if more than one is be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns a dictionary with information on the scaler

`spec.GetScalerLabels` (*index=0*)

returns the labels that have been retrieved for a scaler

Parameters `index` (*int*) – an index for the scaler, if more than one is be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns a list of labels

`spec.GetScalerLastCount` (*index=0*)

returns the last set of counts that have been read for a scaler

Parameters `index` (*int*) – an index for the scaler, if more than one is be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns a list of the last counts

`spec.GetScalerLastTime` (*index=0*)

returns the count time for the last read from a scaler

Parameters `index` (*int*) – an index for the scaler, if more than one is be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns a single float with the last elapsed time for that scaler (initialized at 0) of the last counts

`spec.ListMtrs` ()

Returns a list of the variables defined as motor symbols.

Returns a python list of defined motor symbols (list of str values).

`spec.PositionMtr` (*mtr, pos, wait=True*)

Move a motor

Position a motor associated with `mtr` to position `pos`, wait for the move to complete if `wait` is `True`, or else return immediately. The function attempts to verify the move command has been acted upon.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **pos** (*float*) – a value to position the motor. If the value is invalid or outside the limits an exception occurs (todo: are hard limits checked?).
- **wait** (*bool*) – a flag that specifies if the move should be completed before the function returns. If False, the function returns immediately.

`spec.ReadMtr(mtr)`

Return the motor position associated with the passed motor value.

Parameters **mtr** (*int*) – a key corresponding to an entry in the motor table. If the value does not correspond to a motor entry, an exception is raised.

Returns motor position (float).

`spec.SetDet(Detector=None, index=0)`

Set the main detector channel for the scaler. The default is to restore this to the initial setting, where this is undefined. This is used for ASCAN, etc.

Parameters

- **Monitor** (*int*) – channel number. If omitted the Monitor is set as undefined. The valid range for this parameter is 0 through one less than the number of channels.
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

`spec.SetMon(Monitor=None, index=0)`

Set the monitor channel for the scaler. The default is to restore this to the initial setting, where this is undefined. This is needed for counting on the Monitor.

Parameters

- **Monitor** (*int*) – channel number. If omitted the Monitor is set as undefined. The valid range for this parameter is 0 through one less than the number of channels.
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

`spec.ShowEnabled()`

Show if use of EPICS is allowed or disabled, see `EnableEPICS()`.

Returns True if PyEpics has been loaded, False otherwise

`spec.Sym2MtrVal(mtrsym)`

Converts a motor symbol (as a string) to the motor value (key) as assigned in `DefineMtr()`

Parameters **mtrsym** (*str*) – a motor symbol as supplied in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.

Returns motor value (str).

`spec.count_em(count=None, index=0)`

Cause scaler to start counting for specified period, but return immediately. On the first use, this will take the scaler out of autocount mode and put it into one-shot mode (this is because if one does not read the scaler shortly after a count when in autocount mode, the scaler returns to autocount and the values are lost.) If put in one-shot mode, then autocount will be restored when the python interpreter is exited.

Counting is on time if count is 0 or positive; Counting is on monitor if count < 0

Parameters

- **count-time** (*float*) – time (sec) to count, if omitted COUNT is used (see *Global variables* section)
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.

Returns None

Example:

```
>>> count_em()
>>> # do other commands
>>> wait_count()
>>> get_counts()
```

`spec.ct` (*count=None, index=0, label=False*)

Cause scaler to count for specified period or to a specified number of counts on a prespecified channel (see `SetMon()`)

Counting is on time if count is 0 or positive; Counting is on monitor if count < 0

Global variable S is set to the count values for the n channels (set in `DefineScaler()`) to provide functionality similar to `spec`.

Parameters

- **count** (*float*) – time (sec) to count, if omitted COUNT is used (see *Global variables* section)
- **index** (*int*) – an index for the scaler, if more than one is defined (see `DefineScaler()`). The default (0) is used if not specified.
- **label** (*bool*) – indicates if counts should be printed along with their labels The default (False) is to not print counts

Returns count values for the channels (see `DefineScaler()`)

Example:

```
>>> ct()
[10000000.0, 505219.0, 359.0, 499.0, 389.0, 356.0, 114.0, 53.0]
>>> SetMon(3)
>>> ct(-1000)
[20085739.0, 1011505.0, 719.0, 1000.0, 781.0, 715.0, 226.0, 105.0]
```

`spec.get_counts` (*wait=False*)

Read scaler with optional delay, must follow `count_em`

reads count values for the channels (see `DefineScaler()`)

Parameters **wait** (*bool*) – True causes the routine to wait for the scaler to complete; False (default) will read the scaler instantaneously

Returns a list of channels values

Example:

```
>>> get_counts()
[1, 2, 3, 4, 5, 6, 7, 8]
```

`spec.initElapsed` ()

Initialize the elapsed time counter

`spec.mv(mtr, pos)`

Move motor without wait

If the move cannot be made, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **pos** (*float*) – a value to position the motor. If the value is invalid or outside the limits, an exception occurs.

Example:

```
>>> mv(samX, 0.1)
```

`spec.mvr(mtr, delta)`

Move motor relative to current position without wait.

If the move cannot be made, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **delta** (*float*) – a value to offset the motor. If the resulting value is invalid or outside the limits, an exception occurs.

Example:

```
>>> mvr(samX, 0.1)
```

`spec.setCOUNT(count)`

Sets the default counting time, see global variable `COUNT` (see *Global variables* section). Used in `ct()`.

Parameters **count** (*float*) – default time (sec) to count.

`spec.setDEBUG(state=True)`

Sets the debug state on or off, see global variable `DEBUG` (see *Global variables* section)

Parameters **state** (*bool*) – `DEBUG` is initialized as `False`, but the default effect of `setDEBUG`, if no parameter is specified is to turn the debug state on.

`spec.setElapsed()`

Measure time from the last call to `initElapsed()`. Global variable `ELAPSED` is set to this value. This is called after motors are moved and when counting is done with scalars.

Returns the elapsed time in sec (float)

`spec.setRETRIES(count=20)`

Sets the maximum number of times to retry an EPICS operation (that would nominally be expected to work on the first try) before generating an exception. See global variable `MAX_RETRIES` (in *Global variables* section)

Parameters **count** (*float*) – maximum number of times to retry an EPICS operation. Defaults to 20.

`spec.sleep(sec)`

Causes the script to delay for *sec* seconds. This routine gets replaced when plotting is loaded by an alternate routine (see `sleepWithYield()` in `macros._makePlotWin()`)

Parameters **sec** (*float*) – time to delay in seconds

`spec.umv(mtr, pos)`

Move motor with wait.

If the move cannot be completed, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **pos** (*float*) – a value to position the motor. If the value is invalid or outside the limits, an exception occurs.

Example:

```
>>> umv(samX, 0.1)
```

`spec.umvr(mtr, delta)`

Move motor relative to current position with wait.

If the move cannot be completed, an exception is raised.

Parameters

- **mtr** (*int*) – a value corresponding to an entry in the motor table, as defined in `DefineMtr()`. If the value does not correspond to a motor entry, an exception is raised.
- **delta** (*float*) – a value to offset the motor. If the resulting value is invalid or outside the limits, an exception occurs.

Example:

```
>>> umvr(samX, 0.1)
```

`spec.wa(label=False)`

Print positions of all motors defined using `DefineMtr()`.

Parameters **label** (*bool*) – a flag that specifies if the list should include the motor descriptions. If omitted or `False`, the descriptions are not included.

Example:

```
>>> wa()
samX          1.0
samZ          0.0
>>> wa(True)
samX          1.0      sample X position (mm) + outboard
samZ          0.0      sample Z position (mm) + up
```

`spec.wait_count()`

Wait for scaler to finish, must follow `count_em`

Returns `None`

Example:

```
>>> wait_count()
```

`spec.wm(*mtrs)`

Read out specified motor(s).

Arguments one or more motor table entries that are defined in `DefineMtr()`.

Returns a single float if a single argument is passed to `wm`. Returns a list of floats if more than one argument is passed.

Example:

```
>>> wm(samX, samZ)
[1.0, 0.0]
```

MODULE MACROS: ADDITIONAL SPEC-LIKE EMULATION

Python functions listed below are designed to implement functionality similar to that in spec.

General purpose routines	Description
<code>specdate()</code>	Returns the date/time formatted like Spec
<code>SetScanFile()</code>	Open a file for scan output
<code>ascan()</code>	Scan a single motor on a fixed range
<code>dscan()</code>	Scan a single motor on a range relative to current position
<code>RefitLastScan()</code>	Fit a user-supplied function to a user-supplied function
<code>SendTextEmail()</code>	Sends an e-mail message to one or more addresses

2.1 Logging

An important set of configuration parameters is that which determine what values are recorded. During data collection, for example, after each `ascan()` or `dscan()` data point. Also, for use in defining macros, the values can also be saved to a log file using `write_logging_parameters()`.

Logging routines	Description
<code>init_logging()</code>	Initializes the list of items to be reported
<code>show_logging()</code>	Displays a list of the items that will be logged
<code>make_log_obj_PV()</code>	Define Logging Object that records a PV value
<code>make_log_obj_Global()</code>	Define Logging Object that records a global variable
<code>make_log_obj_PVobj()</code>	Define Logging Object that records a value from a PVobj object
<code>make_log_obj_motor()</code>	Define Logging Object that records a motor position.
<code>make_log_obj_scaler()</code>	Define Logging Object that records a scaler channel value.
<code>log_it()</code>	Adds a Logging Object to the list of items to be reported
<code>add_logging_PV()</code>	Adds a PV to the list of items to be reported
<code>add_logging_Global()</code>	Adds a Global variable to the list of items to be reported
<code>add_logging_PVobj()</code>	Adds a PV object to the list of items to be reported
<code>add_logging_motor()</code>	Adds a motor reference to the list of items to be reported
<code>add_logging_scaler()</code>	Adds a scaler channel to the list of items to be reported
<code>write_logging_header()</code>	Writes a header line with labels for each logged item
<code>write_logging_parameters()</code>	Write the current value of each logged variable

Two examples for setting up logging (new method):

```
>>> import macros
>>> macros.init_logging()
```

```
>>> GE_prefix = 'GE2:cam1:'
>>> macros.log_it(macros.make_log_obj_PV('GE_fname', GE_prefix+"FileName", as_string=True))
>>> macros.log_it(macros.make_log_obj_PV('GE_fnum', GE_prefix+"FileNumber"))
>>> macros.log_it(macros.make_log_obj_motor(spec.samX))
>>> macros.log_it(macros.make_log_obj_scaler(9))
>>> macros.log_it(macros.make_log_obj_Global('var S9', 'spec.S[9]'))
>>> macros.log_it(macros.make_log_obj_PV('p1Vs', "1idc:m64.RBV"))
```

Note that the *make_log_obj_scaler* and *make_log_obj_Global* calls above will record the same value (though with different headings), but the *make_log_obj_scaler* is a better choice as the second option could produce the wrong value if use of a second scaler is later added to a script.

Old method (does the same as the previous) is:

```
>>> import macros
>>> macros.init_logging()
>>> GE_prefix = 'GE2:cam1:'
>>> macros.add_logging_PV('GE_fname', GE_prefix+"FileName", as_string=True)
>>> macros.add_logging_PV('GE_fnum', GE_prefix+"FileNumber")
>>> macros.add_logging_motor(spec.samX)
>>> macros.add_logging_scaler(9)
>>> macros.add_logging_Global('var S9', 'spec.S[9]')
>>> macros.add_logging_PV('p1Vs', "1idc:m64.RBV")
```

Example for use of logging in a script:

```
>>> mac.write_logging_header(logname)
>>> spec.umv(spec.mts_y, stY)
>>> for iLoop in range(nLoop):
>>>     spec.umvr(spec.mts_y, dY)
>>>     count_em(Nframe*tframe)
>>>     GE_expose(fname, Nframe, tframe)
>>>     wait_count()
>>>     get_counts()
>>>     mac.write_logging_parameters(logname)
>>> mac.beep_dac()
```

This code step-scans motor *mts_y*. It writes a header to the log file at the beginning of the operation and then logs parameters after each measurement. Measurements are done in *GE_expose* and the default scaler, which are run at the same time.

Note that it can be useful to put differing sets of logging configurations into files where they can be invoked as needed using `execfile(xxx.py)` [where *xxx.py* is the name of the file to be read]. Do not use `import` for this task because `import` will process the file when it is referenced first, but will not do anything if one attempts to import the file again (to reset values back after a different setting has been used). One must use `reload` to force that.

2.2 Plotting

Similar to logging, it is also possible to designate that values can be plotted as part of a script. A Logging Object (from the `make_log_obj_...()` routines) is needed for each item that will be plotted.

Plotting routines	Description
<code>make_log_obj_PV()</code>	Define Logging Object that records a PV value
<code>make_log_obj_Global()</code>	Define Logging Object that records a global variable
<code>make_log_obj_PVobj()</code>	Define Logging Object that records a value from a PVobj object
<code>make_log_obj_motor()</code>	Define Logging Object that records a motor position.
<code>make_log_obj_scaler()</code>	Define Logging Object that records a scaler channel value.
<code>DefineLoggingPlot()</code>	Creates a plot (if needed) or tab on tab to display values and register items to be plotted.
<code>UpdateLoggingPlots()</code>	Read and display all parameters added to plot in <code>DefineLoggingPlot()</code> .
<code>InitLoggingPlot()</code>	Clear out plotting definitions from previous calls to <code>DefineLoggingPlot()</code> .

Examples:

```
>>> macros.DefineLoggingPlot (
...     'I vs pos',
...     macros.make_log_obj_motor(spec.samX),
...     macros.make_log_obj_scaler(2),
... )
>>> spec.umv(spec.samX,2)
>>> for iLoop in range(30):
...     spec.umvr(spec.samX,0.05)
...     spec.ct(1)
...     macros.UpdateLoggingPlots()
```

In the above example, a scaler channel is read and plotted against a motor position.

```
>>> macros.DefineLoggingPlot (
...     'I vs time',
...     macros.make_log_obj_Global('time (sec)', 'spec.ELAPSED'),
...     macros.make_log_obj_scaler(2),
...     macros.make_log_obj_scaler(3),
... )
>>> spec.initElapsed()
>>> for iLoop in range(30):
...     spec.ct(1)
...     macros.UpdateLoggingPlots()
```

In the above example, two scaler channels are plotted against elapsed time.

2.3 Macros specific to 1-ID

These macros reference 1-ID PV's or are customized for 1-ID in some other manner.

1-ID specific routines	Description
<code>beep_dac()</code>	Causes a beep to sound
<code>Cclose()</code>	Close 1-ID fast shutter in B hutch
<code>Copen()</code>	Open 1-ID fast shutter in B hutch
<code>shutter_sweep()</code>	Set 1-ID fast shutter to external control
<code>shutter_manual()</code>	Set 1-ID fast shutter to manually control
<code>check_beam_shutterA()</code>	Open 1-ID Safety shutter to bring beam into 1-ID-A
<code>check_beam_shutterC()</code>	Open 1-ID Safety shutter to bring beam into 1-ID-C
<code>Sopen()</code>	Same as <code>check_beam_shutterC()</code> , bring beam into 1-ID-C
<code>MakeMtrDefaults()</code>	Create a file with default motor assignments
<code>SaveMotorLimits()</code>	Create a file with soft limits for off-line simulations

2.4 Complete Function Descriptions

The functions available in this module are listed below.

`macros.Cclose()`

Close I-ID fast shutter in B hutch

`macros.Copen()`

Open I-ID fast shutter in B hutch

`macros.DefineLoggingPlot (tblbl, Xvar, *args)`

Creates a plot window (if needed) or tab on plot to display values. Parameters include a label for the tab, a Logging Object that will be used as an x-value and as many Logging Object as desired (minimum 1) that will define y-values. Each time this routine is called, a new plot tab is called. As many plot tabs can be created and populated as desired.

see [Plotting](#) for an example of use.

Parameters

- **tblbl** (*str*) – a label to place on the plot tab
- **Xvar** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`
- **Yvar** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`
- **Yvar1** (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`

`class macros.FitClass (x, y)`

Defines a prototype class for deriving fitting class implementations. A fitting class should define at least two method: `__init__` and `Eval`.

`__init__(x,y)` computes a list of very approximate values for the fit parameters, good enough to be used as the starting values in the fit. The number of terms computed determines the number of parameter values that will be fit.

`Eval(parms,x)` provides the function to be fit.

optionally, `Format(parms)` is used to return a nicely-formatted text string with the fitted parameters.

`Eval (parm, x)`

Evaluate the fitting function and return a “y” value computed for each value in x. Ideally this expression computes all values in a single NumPy expression, but looping is allowed. Both parameters should be lists, tuples or numpy arrays.

Parameters

- **parm** (*list,tuple,etc.*) – parameters in the same order as returned by `StartParms()`
- **x** (*list,tuple,etc.*) – values of the independent parameter (scanned variable) for evaluation of the function.

`Format (parm)`

This prints the parameters, potentially in a way that explains what they mean. If not overridden, one gets “Parameter values = <list>”

Parameters `parm` (*list,tuple,etc.*) – parameters in the same order as returned by `StartParms()`

StartParms ()

Return the starting parameter values determined in `__init__()`

class macros.FitGauss (*x*, *y*)

Define a function for fitting with a Gaussian.

Parameters are defined as:

index	value
[0]	location of peak
[1]	function value at maximum, less parm[3]
[2]	width as FWHM
[3]	added to all points

Eval (*parm*, *x*)

Evaluate the Gaussian

Format (*parm*)

Prints the parameters

class macros.FitSawtooth (*x*, *y*, *Symmetric=True*)

Define a function for fitting with a symmetric or asymmetric saw-tooth function.

Parameters are defined as:

index	value
[0]	location of peak
[1]	function value at maximum
[2]	added to all points
[3]	asymmetric: slope on leading side of peak (+ is rising) symmetric: slope on both sides of peak
[4]	asymmetric: slope on trailing side of peak (+ is falling)

Parameters Symmetric (*bool*) – determines if the SawTooth is symmetric (True) or asymmetric (False), meaning that the leading side and the trailing side of the peak can have different slopes.

Eval (*parm*, *x*)

Evaluate the sawtooth function

macros.InitLoggingPlot ()

Clear out plot definitions from previous calls to `DefineLoggingPlot()`. Prevents updates from occurring in `UpdateLoggingPlot()`, but does not delete any tabs or the window.

macros.MakeMtrDefaults (*fil=None*, *out=None*)

Routine in Development: Creates an initialization file from a spreadsheet describing the 1-ID beamline motor assignments

Parameters

- **fil** (*str*) – input file to read. By default opens file `../1ID/1ID_stages.csv` relative to the location of the current file.
- **out** (*str*) – output file to write. By default writes file `../1ID/mtrsetup.py.new` Note that if the default file name is used, the output file must be renamed before use to `mtrsetup.py`

macros.RefitLastScan (*FitClass*, ***kwargs*)

Fit and plot an arbitrary equation to data from the last ascan

Parameters FitClass (*class*) – a class that defines a minimum of two methods, one to define a fitting function and the other to determine rough starting values for the fitting function. See `FitGauss` or `FitSawtooth` for examples of Fitting classes.

Optional: additional keyword parameters will be passed for the creation of a FitClass object.

Returns an optimized list of parameters or None if the fit fails

Example:

```
>>> macros.RefitLastScan(macros.FitSawtooth)
Parameter values =1.45, 28.5, 1.5, 2.1053
array([ 1.44999999, 28.50005241, 1.4999749 , 2.10525894])
```

or

```
>>> macros.RefitLastScan(macros.FitSawtooth, Symmetric=False)
Parameter values =1.45, 28.5, 1.5, 2.1053, 2.1053
array([ 1.44999999, 28.5000524 , 1.49997491, 2.10525896, 2.10525891])
```

`macros.SaveMotorLimits` (*out=None*)

Routine in Development: Creates an initialization file for simulation use with the limits for every motor PV that is found in the current 1-ID beamline motor assignments. import `mtrsetup.py` or equivalent first. Scans each PV from 1 to the max number defined.

Parameters *out* (*str*) – output file to write, writes file `motorlimits.dat.new` in the same directory as this file by default. Note that if the default file name is used, the output file must be renamed before use to `motorlimits.dat`

`macros.SendTextEmail` (*recipientlist*, *msgtext*, *subject='specpy auto msg'*, *recipientname=None*, *senderemail='IID@aps.anl.gov'*)

Send a short text string as an e-mail message. Uses the APS outgoing email server (`apsmail.aps.anl.gov`) to send the message via SMTP.

Parameters

- **recipientlist** (*str*) – A string containing a single e-mail address or a list or tuple (etc.) containing a list of strings with e-mail addresses.
- **msgtext** (*str*) – a string containing the contents of the message to be sent.
- **subject** (*str*) – a subject to be included in the e-mail message; defaults to “specpy auto msg”.
- **recipientname** (*str*) – a string to be used for the recipient(s) of the message. If not specified, no “To:” header shows up in the e-mail. This should be an e-mail address or `@aps.anl.gov` is appended.
- **senderemail** (*str*) – a string with the e-mail address identified as the sender of the e-mail; defaults to “IID@aps.anl.gov”. This should be an e-mail address or `@aps.anl.gov` is appended.

Examples:

```
>>> msg = 'This is a very short e-mail'
>>> macros.SendTextEmail(['toby@sigmaxi.net', 'brian.h.toby@gmail.com'], msg, subject='test')
```

or with a single address:

```
>>> msg = """Dear Brian,
...   How about a longer message?
...   Thanks, Brian
...   """
>>> to = "toby@anl.gov"
>>> macros.SendTextEmail(to, msg, recipientname='spamee@anl.gov', senderemail='spammer@anl.gov')
```

A good way to use this routine is in a try/except block:

```
>>> userlist = ['user@univ.edu', 'contact@anl.gov']
>>> try:
>>>     macros.write_logging_header(logname)
>>>     spec.umv(spec.mts_y, stY)
>>>     for iLoop in range(nLoop):
>>>         spec.umv(spec.mts_x2, stX)
>>>         for xLoop in range(nX):
>>>             GE_expose(fname, Nframe, tframe)
>>>             macros.write_logging_parameters(logname)
>>>             spec.umvr(spec.mts_x2, dX)
>>>             spec.umvr(spec.mts_y, dY)
>>>         macros.beep_dac()
>>> except Exception:
>>>     import traceback
>>>     msg = "An error occurred at " + macros.specdate()
>>>     msg += " in file " + __file__ + "\n\n"
>>>     msg += str(traceback.format_exc())
>>>     macros.SendTextEmail(userlist, msg, 'Beamline Abort')
```

`macros.SetScanFile(outfile=None)`

Set a file for output from ascan, etc. The output is intended to closely mimic what spec produces in ascan and dscan.

Parameters `outfile` (*str*) – the file name to be opened. If not specified, output is sent to the terminal.

If the file is new (or is the not specified) a header listing all motors, etc. is printed

`macros.ShowPlots()`

Pause to show plot screens. Call this at the end of a script, if needed.

`macros.Sopen()`

If not already open, open 1-ID-C Safety shutter to bring beam into 1-ID-C. Keep trying in an infinite loop until the shutter opens.

`macros.UpdateLoggingPlots()`

Read all current values in plot and display in plots

see [Plotting](#) for an example of use.

`macros.add_logging_Global(txt, var)`

Define a global variable to be recorded when `write_logging_parameters()` is called.

Parameters

- `txt` (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- `var` (*str*) – defines a Python variable that will be logged each time `write_logging_parameters()` is called. Note that this is read inside the macros module so the variable must be defined inside that module or must be prefixed by a reference to a module referenced in that module, e.g. `spec.S[0]`

see [Logging](#) for an example of use.

`macros.add_logging_PV(txt, PV, as_string=False)`

Define a PV to be recorded when `write_logging_parameters()` is called.

Parameters

- `txt` (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.

- **PV** (*str*) – defines an EPICS Process Variable that will be read and logged each time `write_logging_parameters()` is called.
- **as_string** (*bool*) – if True, the PV will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

macros.**add_logging_PVobj** (*txt, PVobj, as_string=False*)

Define a PVobj to be recorded when `write_logging_parameters()` is called.

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **PV** (*epics.PV*) – defines a PyEpics PV object that is connected to an EPICS Process Variable. The PV method `.get()` will be used to read that PV to log it each time `write_logging_parameters()` is called.
- **as_string** (*bool*) – if True, the PV value will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

macros.**add_logging_motor** (*mtr*)

Define a motor object to be recorded when `write_logging_parameters()` is called. Note that the heading text string is defined as the motor's symbol (see `spec.DefineMtr()`).

Parameters **mtr** (*str*) – a reference to a motor object, returned by `spec.DefineMtr()` or defined in the motor symbol. The position of the motor will be read and logged each time `write_logging_parameters()` is called.

see *Logging* for an example of use.

macros.**add_logging_scaler** (*channel, index=0*)

Define a scaler channel to be recorded when `write_logging_parameters()` is called. Note that the heading text string is defined as the scaler's label (which is read from the scaler when `spec.DefineScaler()` is run).

Parameters

- **channel** (*str*) – a channel number for a scaler, which can be any value between 0 and one less than the number of channels. The last-read value of that scaler logged each time `write_logging_parameters()` is called.
- **index** (*int*) – an index for the scaler, if more than one is to be defined (see `DefineScaler()`). The default (0) is used if not specified.

see *Logging* for an example of use.

macros.**ascan** (*mtr, start, finish, npts, count, index=0, settle=0.0, _func='ascan'*)

Scan one motor and record parameters set with logging to the scanfile (see `func:SetScanFile`).

Parameters

- **mtr** (*str*) – a reference to a motor object, returned by `spec.DefineMtr()` or defined in the motor symbol.
- **start** (*float*) – starting position for scan
- **finish** (*float*) – ending position for scan

- **npts** (*int*) – number of points for scan
- **count** (*float*) – count time. Counting is on time (sec) if count is 0 or positive; Counting is on monitor if count < 0
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.
- **settle** (*float*) – a time to wait (sec) after the motor has been moved before counting is starting. Default is 0.0 which means no delay

Example:

```
>>> spec.SetDet(2)
>>> macros.ascan(spec.samX, 1, 2, 21, 1, settle=.1)
```

It is recommended that if `ascan` will be run in command line, where python commands are typed into a console window, that ipython be used in pylab mode (`ipython --pylab`).

```
macros.beep_dac (beep_time=1.0)
```

Set the 1-ID beeper on for a fixed period, which defaults to 1 second uses PV object beeper (defined as lid:DAC1_8.VAL) makes sure that the beeper is actually turned on and off throws exception if beeper fails

Parameters `beep_time` (*float*) – time to sound the beeper (sec), defaults to 1.0

```
macros.check_beam_shutterA()
```

If not already open, open 1-ID-A Safety shutter to bring beam into 1-ID-A. Keep trying in an infinite loop until the shutter opens.

```
macros.check_beam_shutterC()
```

If not already open, open 1-ID-C Safety shutter to bring beam into 1-ID-C. Keep trying in an infinite loop until the shutter opens.

```
macros.dscan (mtr, start, finish, npts, count, index=0, settle=0.0)
```

Relative scan of motor, see func:`ascan`,

Parameters

- **mtr** (*str*) – a reference to a motor object, returned by `spec.DefineMtr()` or defined in the motor symbol.
- **start** (*float*) – starting position for scan, relative to current motor position
- **finish** (*float*) – ending position for scan, relative to current motor position
- **npts** (*int*) – number of points for scan
- **count** (*float*) – count time. Counting is on time (sec) if count is 0 or positive; Counting is on monitor if count < 0
- **index** (*int*) – an index for the scaler, if more than one will be defined (see `DefineScaler()`). The default (0) is used if not specified.
- **settle** (*float*) – a time to wait (sec) after the motor has been moved before counting is starting. Default is 0.0 which means no delay

Example:

```
>>> spec.SetDet(2)
>>> macros.dscan(spec.samX, -1, 1, 21, 1, settle=.1)
```

It is recommended that if `dscan` will be run in command line, where python commands are typed into a console window, that ipython be used in pylab mode (`ipython --pylab`).

`macros.init_logging()`

Initialize the list of data items to be logged

see *Logging* for an example of use.

`macros.log_it(LogObj)`

Add a Logging Object into list to be recorded when `write_logging_parameters()` is called.

Parameters `LogObj` (*object*) – a reference to a Logging Object created by `make_log_obj_PV()`, `make_log_obj_Global()`, `make_log_obj_PVobj()`, `make_log_obj_motor()` or `make_log_obj_scaler()`

`macros.make_log_obj_Global(txt, var)`

Define Logging Object that records a global variable

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **var** (*str*) – defines a Python variable that will be logged each time `write_logging_parameters()` is called. Note that this is read inside the macros module so the variable must be defined inside that module or must be prefixed by a reference to a module referenced in that module, e.g. `spec.S[0]`

see *Logging* for an example of use.

`macros.make_log_obj_PV(txt, PV, as_string=False)`

Define Logging Object that records a PV value

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **PV** (*str*) – defines an EPICS Process Variable that will be read and logged each time `write_logging_parameters()` is called.
- **as_string** (*bool*) – if True, the PV will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

`macros.make_log_obj_PVobj(txt, PVobj, as_string=False)`

Define Logging Object that records a value from a PVobj object

Parameters

- **txt** (*str*) – defines a text string, preferably short, to be used when `write_logging_header()` is called as a header for the item to be logged.
- **PV** (*epics.PV*) – defines a PyEpics PV object that is connected to an EPICS Process Variable. The PV method `.get()` will be used to read that PV to log it each time `write_logging_parameters()` is called.
- **as_string** (*bool*) – if True, the PV value will be translated to a string. When False (default) the native data type will be used. Use of True is of greatest for waveform records that are used to store character strings as a series of integers.

see *Logging* for an example of use.

`macros.make_log_obj_motor(mtr)`

Define Logging Object that records a motor position. Note that the heading text string is defined as the motor's symbol (see `spec.DefineMtr()`).

Parameters `mtr (str)` – a reference to a motor object, returned by `spec.DefineMtr()` or defined in the motor symbol. The position of the motor will be read and logged each time `write_logging_parameters()` is called.

see *Logging* for an example of use.

`macros.make_log_obj_scaler(channel, index=0)`

Define Logging Object that records a scaler channel value. Note that the heading text string is defined as the scaler's label (which is read from the scaler when `spec.DefineScaler()` is run).

Parameters

- **channel (str)** – a channel number for a scaler, which can be any value between 0 and one less than the number of channels. The last-read value of that scaler logged each time `write_logging_parameters()` is called.
- **index (int)** – an index for the scaler, if more than one is to be defined (see `DefineScaler()`). The default (0) is used if not specified.

see *Logging* for an example of use.

`macros.show_logging()`

Show the user the current logged items

`macros.shutter_manual()`

Set 1-ID fast shutter so that it will not be controlled by the GE TTL signal and can be manually opened and closed with `Copen()` and `Cclose()`

`macros.shutter_sweep()`

Set 1-ID fast shutter so that it will be controlled by an external electronic control (usually the GE TTL signal)

`macros.specdate()`

format current date/time as produced in `Spec`

Returns the current date/time as a string, formatted like “Thu Oct 04 18:24:14 2012”

Example:

```
>>> macros.specdate()
'Thu Oct 11 16:16:39 2012'
```

`macros.write_logging_header(filename='')`

Write a header for parameters recorded when `write_logging_parameters()` is called.

Parameters `filename (str)` – a filename to be used for output. If not specified, the output is sent to the terminal window.

see *Logging* for an example of use.

`macros.write_logging_parameters(filename='')`

Record the current value of all items tagged to be recorded in `add_logging_PV()`, `add_logging_Global()`, `add_logging_PVobj()`, `add_logging_motor()` or `add_logging_scaler()`.

Parameters `filename (str)` – a filename to be used for output. If not specified, the output is sent to the terminal window.

see *Logging* for an example of use.

INDEX

A

add_logging_Global() (in module macros), 19
add_logging_motor() (in module macros), 20
add_logging_PV() (in module macros), 19
add_logging_PVobj() (in module macros), 20
add_logging_scaler() (in module macros), 20
ascan() (in module macros), 20

B

beep_dac() (in module macros), 21

C

Cclose() (in module macros), 16
check_beam_shutterA() (in module macros), 21
check_beam_shutterC() (in module macros), 21
Copen() (in module macros), 16
COUNT, 4
count_em() (in module spec), 8
ct() (in module spec), 9

D

DEBUG, 5
DefineLoggingPlot() (in module macros), 16
DefineMtr() (in module spec), 5
DefineScaler() (in module spec), 6
dscan() (in module macros), 21

E

ELAPSED, 5
EnableEPICS() (in module spec), 6
Eval() (macros.FitClass method), 16
Eval() (macros.FitGauss method), 17
Eval() (macros.FitSawtooth method), 17
ExplainMtr() (in module spec), 6

F

FitClass (class in macros), 16
FitGauss (class in macros), 17
FitSawtooth (class in macros), 17
Format() (macros.FitClass method), 16
Format() (macros.FitGauss method), 17

G

get_counts() (in module spec), 9
GetDet() (in module spec), 6
GetMon() (in module spec), 7
GetMtrInfo() (in module spec), 7
GetScalerInfo() (in module spec), 7
GetScalerLabels() (in module spec), 7
GetScalerLastCount() (in module spec), 7
GetScalerLastTime() (in module spec), 7

I

init_logging() (in module macros), 22
initElapsed() (in module spec), 9
InitLoggingPlot() (in module macros), 17

L

ListMtrs() (in module spec), 7
log_it() (in module macros), 22

M

make_log_obj_Global() (in module macros), 22
make_log_obj_motor() (in module macros), 22
make_log_obj_PV() (in module macros), 22
make_log_obj_PVobj() (in module macros), 22
make_log_obj_scaler() (in module macros), 23
MakeMtrDefaults() (in module macros), 17
MAX_RETRIES, 5
mv() (in module spec), 9
mvr() (in module spec), 10

P

PositionMtr() (in module spec), 7

R

ReadMtr() (in module spec), 8
RefitLastScan() (in module macros), 17

S

S, 4
SaveMotorLimits() (in module macros), 18
SendTextEmail() (in module macros), 18

setCOUNT() (in module spec), 10
setDEBUG() (in module spec), 10
SetDet() (in module spec), 8
setElapsed() (in module spec), 10
SetMon() (in module spec), 8
setRETRIES() (in module spec), 10
SetScanFile() (in module macros), 19
show_logging() (in module macros), 23
ShowEnabled() (in module spec), 8
ShowPlots() (in module macros), 19
shutter_manual() (in module macros), 23
shutter_sweep() (in module macros), 23
sleep() (in module spec), 10
Sopen() (in module macros), 19
specdate() (in module macros), 23
StartParms() (macros.FitClass method), 16
Sym2MtrVal() (in module spec), 8

U

umv() (in module spec), 10
umvr() (in module spec), 11
UpdateLoggingPlots() (in module macros), 19

W

wa() (in module spec), 11
wait_count() (in module spec), 11
wm() (in module spec), 11
write_logging_header() (in module macros), 23
write_logging_parameters() (in module macros), 23