APSpy Documentation

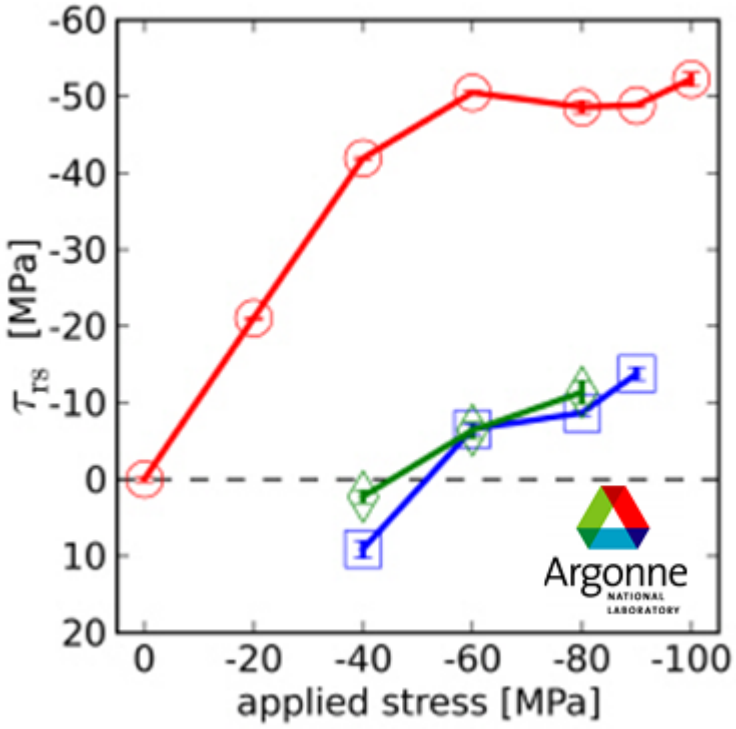
Release 1.2.1.trunk

Brian Toby, Lakhsmipriya Sukumar, Jon Almer, Pete Jemian

June 21, 2013

CONTENTS

Support for scripting experiment operations at APS beam lines using Python and EPICS (via PyEpics).



Brief Contents

CHAPTER

ONE

COMPLETE TABLE OF CONTENTS

1.1 Design and Theory of Operation

-tba-

1.2 Installation

- subversion repository, development trunk:
  - https://subversion.xray.aps.anl.gov/bcdaext/APSpy/trunk

```
svn co https://subversion.xray.aps.anl.gov/bcdaext/APSpy/trunk APSpy
```

- current documentation:
  - https://subversion.xray.aps.anl.gov/bcdaext/APSpy/trunk/docs/build/html/index.html
- TRAC site (bug reports, feature requests):
  - https://subversion.xray.aps.anl.gov/trac/bcdaext
  - This TRAC site is shared between multiple projects.
- Python Python *distutils* [1] installation:
  - download latest *.zip* (or *.tar.gz*) file from:
    - https://subversion.xray.aps.anl.gov/bcdaext/APSpy/trunk/dist
    - uncompress the archive to a directory and open the directory on a command-line
    - `python ./setup.py install`

Note: This package uses the Python *PyEpics* [2] package. Without the PyEpics package present, this package will operate in simulation mode.

Note: This package requires the Python *NumPy* [3] package.

[1] distutils: http://docs.python.org/2/distutils/
[2] PyEpics: http://cars9.uchicago.edu/software/python/pyepics3/
[3] NumPy: http://www.numpy.org/

In addition the PyEpics package must be installed order to control an instrument. However, if PyEpics is not installed, all routines documented here can still be run.

In simulation mode, EPICS interactions will be simulated and print statements will report what the Python code is attempting to do.

Simulation mode can be obtained (and no communication with EPICS is attempted) if PyEpics is installed by either of these methods:

1. Do not call `APSpy.spec.EnableEPICS()`
2. call `APSpy.spec.EnableEPICS(False)`

This allows scripts to be developed and tested without access to the instrument.

1.3 Configuration

-tba-

Caution: Need to verify.

example setup for operations at 1-ID: *brown.py*

```
import time
import datetime as dt
import epics as ep  #from epics import PV
import spec
import macros as mac
PV = ep.PV

################################################################################################################
# misc beamline macros
#PV Objects
spec.EnableEPICS()
spec.DefineMtr('mts_x2','1idc:m4','MTS_X2')
spec.DefineMtr('mts_y','1idc:m6','MTS_Y')
spec.DefineScaler('1id:scaler1',16)

GE_prefix = 'GE2:cam1:'
#GE_fnum = ep.PV(GE_prefix+"FileNumber")
GE_fname = ep.PV(GE_prefix+"FileName")
GE_tframe = ep.PV(GE_prefix+"AcquireTime")
GE_Nframe = ep.PV(GE_prefix+'NumImages')
GE_Acquire = ep.PV(GE_prefix+'Acquire')
GE_address = ep.PV(GE_prefix+'DetectorState_RBV')

def sleep(timesec):
    time.sleep(timesec)


################################################################################################################

mac.init_logging()
mac.add_logging_PV('GE_fname',GE_prefix+"FileName",as_string=True)
mac.add_logging_PV('GE_fnum',GE_prefix+"FileNumber")
```

```
mac.add_logging_PV('GE_tframe', GE_prefix+"AcquireTime")
mac.add_logging_PV('GE_Nframe',GE_prefix+'NumImages')
mac.add_logging_Global('S0','spec.S[0]')
mac.add_logging_Global('S1','spec.S[1]')
mac.add_logging_Global('S2','spec.S[2]')
mac.add_logging_Global('S8','spec.S[8]')
mac.add_logging_Global('S9','spec.S[9]')
mac.add_logging_Global('S10','spec.S[10]')
mac.add_logging_Global('S11','spec.S[11]')
mac.add_logging_PV('p1Hs',"1idc:m62.RBV")
mac.add_logging_PV('p1Vs',"1idc:m64.RBV")
mac.add_logging_PV('Iring',"BL01:srCurrent")
mac.add_logging_PV('energy',"1id:userTran3.A")
#mac.add_logging_PV('energy_cal',"1id:userTran3.A")
mac.add_logging_PV('preamp1',"1idc:A3sens_num.VAL")
mac.add_logging_PV('preamp2',"1idc:A4sens_num.VAL")
mac.add_logging_motor(spec.mts_x2)
mac.add_logging_motor(spec.mts_y)
#mac.add_logging_motor() # sammy_z,sammy_x2,sammy_z2,sammy_phi
mac.add_logging_PV('keyence1',"1id:Keyence:1:ch1.VAL")
mac.add_logging_PV('keyence2',"1id:Keyence:1:ch2.VAL")
mac.add_logging_PV('cross',"1id:D2Ch1_calc.VAL")
mac.add_logging_PV('load',"1id:D2Ch2_calc.VAL")
mac.add_logging_PV('mts3',"1id:D2Ch3_calc.VAL")
mac.add_logging_PV('mts4',"1id:D2Ch4_calc.VAL")
mac.add_logging_PV('temp1',"1id:ET_RI:Temp1")
mac.add_logging_PV('temp2',"1id:ET_RI:Temp2")
mac.add_logging_PV('temp3',"1id:ET_RI:Temp3")

def scan_xyN (fname, Nframe, tframe, stX, nX, dX, stY, nY, dY, nLoop, logname):
    ''' Usage: scan_xyN [fname] [Nframe] [tframe] [x0] [Nx] [dx] [y0] [Ny] [dy] [nLoop] [logname]
    '''
    # define PVs to be recorded

#p date(),GE_fname,GE_fnum,GE_tframe,GE_Nframe,S[0],S[1],S[2],S[8],S[9],S[10],S[11],
#   p1Hs,p1Vs,Iring,energy,energy_cal,preamp1,preamp2,
#    sammy_x,sammy_y,sammy_z,sammy_x2,sammy_z2,sammy_phi,
#    keyence1,keyence2,cross,load,mts3,mts4,temp1,temp2,temp3,temp4

    mac.write_logging_header(logname)
    for iLoop in range(nLoop):
        spec.umv(spec.mts_y,stY)
        for yLoop in range(nY):
            spec.umv(spec.mts_x2,stX)
            for xLoop in range(nX):
                GE_expose(fname, Nframe, tframe)
                #record beamline and exposure data in parameter file after exposure
                mac.write_logging_parameters(logname)
                spec.umvr(spec.mts_x2,dX)
            spec.umvr(spec.mts_y,dY)
    mac.beep_dac()


def GE_expose(fname, Nframe, tframe):
    """Collect data with a GE detector, defined by PV objects, GE_fname, GE_tframe and GE_Nframe.
    This checks the shutters are open for A & C and closes the fast shutter and puts it
    under remote control. The GE is loaded with the function's parameters and is started.
    While the GE collects data, the default scaler is counted. When the GE indicates it is done
```

```
    the fast shutter is taken out of remote control mode.

    :param str fname: Defines the file name used by the GE for data collection
    :param int Nframe: Defines the number of frames that will be collected
    :param float tframe: Defines the time per frame (sec)
    """

    delt=0.05 #wait time between epics commands
    mac.Cclose() #arm shutters
    mac.check_beam_shutterA()
    mac.check_beam_shutterC() #arm shutters
    mac.shutter_sweep() #shutter will be controlled remotely (by GE TTL)

    # set GE collection params
    GE_fname.put(fname)
    GE_tframe.put(tframe)
    GE_Nframe.put(Nframe)
    sleep(delt)

    # start collection
    GE_Acquire.put(1)
    sleep(delt) # probably serves no purpose, may as well start count immediately
    # start a count on scaler to last while GE collects data plus one second
    spec.ct(Nframe*tframe+1)  #count scalers for absorption/i0 info

    GE_wait_for_idle() #return prompt once GE is idle
    mac.shutter_manual() #reset to allow Copen/Cclose shutter commands

def GE_wait_for_idle(wait=250):
    '''check if GE is idle by checking the Epics PV driven by the Area Detector Control application
    checks every 0.1 sec (set by poll_time) up to wait seconds. If the wait is exceeded, the
    GE is assumed to be hung up and a Exception is triggered.

    :param int wait: time to wait before triggering an Exception. Defaults to 250 sec.
    '''

    poll_time=0.1
    Ncheck_max= wait/poll_time
    icheck=0
    sleep(poll_time)

    while GE_address.get() !=  0: # value of 0 = idle
        icheck += 1
        if icheck > Ncheck_max:
            raise Exception,"waited too long for GE - check detector computer and restart programs /
        sleep(poll_time) #extra safety factor
```

1.4 Examples

-tba-

1.4.1 Interactive session (ipython)

```
#!/usr/bin/env python


'''Test of spec module used by an interactive session'''


########### SVN repository information ###################
# $Date: 2013-04-24 18:41:03 -0500 (Wed, 24 Apr 2013) $
# $Author: jemian $
# $Revision: 1281 $
# $URL: https://subversion.xray.aps.anl.gov/bcdaext/APSpy/trunk/src/testing/test_spec_session.py $
# $Id: test_spec_session.py 1281 2013-04-24 23:41:03Z jemian $
########### SVN repository information ###################


from APSpy.spec import *
import epics as PyEpics
import logging
from pprint import pprint


def wait_motors():
    '''this is useful but specific to the development IOC'''
    while not alldone.get():
        sleep(1)

alldone = PyEpics.PV('como:alldone')


EnableEPICS()
DefineMtr('samX', 'como:m1',  'sample X position (mm) + outboard')
DefineMtr('samZ', 'como:m2',  'sample Z position (mm) + up')
DefineMtr('phi',  'como:m3',  'sample rotation (deg)')
DefineMtr('j1',   'como:m4',  'sample table N jack')
DefineMtr('j2',   'como:m5',  'sample table SE jack')
DefineMtr('j3',   'como:m6',  'sample table SW jack')

m = motors.get_keys_enum()
print 'samX', m.samX, wm(m.samX)
print 'phi',  m.phi,  wm(m.phi)
print 'phi',  m.phi,  wm('phi')
try:
    # this should fail since 'phi' is not a
    # known symbol in our namespace
    print 'phi',  m.phi,  wm(phi)
except NameError, s:
    print s
exec( DefineMotorSymbols(mtrDB) )
print 'phi',  m.phi,  wm(phi)
print
wa()
print '\n again \n'
wa(True)

print "j2 motor's dictionary:"
pprint(GetMtrInfo( m.j2 ))
```

```

DefinePseudoMtr({
    # define pseudo motor position
    'jack': '(A[j1] + A[j2] + A[j3])/3.',
    # define motor movements in terms of pseudo motor target position
    'j1': 'A[j1] + T[jack] - A[jack]',
    'j2': 'A[j2] + T[jack] - A[jack]',
    'j3': 'A[j3] + T[jack] - ((A[j1] + A[j2] + A[j3])/3)',
    },'composite motion including j1,j2 & j3')
DefinePseudoMtr({
    # define pseudo motor positions
    'samLX': 'cosd(A[phi])*A[samX] + sind(A[phi])*A[samZ]',
    'samLZ': '-sind(A[phi])*A[samX] + cosd(A[phi])*A[samZ]',
    # define motor movements in terms of pseudo motor target position
    'samX' : 'cosd(A[phi])*T[samLX] - sind(A[phi]) * T[samLZ]',
    'samZ' : 'sind(A[phi])*T[samLX] + cosd(A[phi]) * T[samLZ]',
    },'sample displacement in diffractometer axes')

wa(True)

mmv([(m.samX,0),(m.samZ,0),(m.phi,0),(m.j1,0),(m.j2,0),(m.j3,0)])
wait_motors()

print "moving samX to 2.0";    umv( m.samX,   2.)
print "moving samZ to 0.5";    umv( m.samZ,   0.5)
wait_motors()
print "moving samX by -1.0";   umvr(m.samX,  -1.)
print "moving j1   to 2.05";   umv( m.j1,     2.05)
print "moving j2   to 2.11";   umv( m.j2,     2.11)
print "moving j3   to 2.2";    umv( m.j3,     2.2)
print "moving phi  to 30.0";   umv( m.phi,    30.0)
wait_motors()
wa(True)



print 'done'
```

1.4.2 Python file

```
#!/usr/bin/env python


'''Test of spec module used by another module'''


############ SVN repository information ###################
# $Date: 2013-04-24 18:41:03 -0500 (Wed, 24 Apr 2013) $
# $Author: jemian $
# $Revision: 1281 $
# $URL: https://subversion.xray.aps.anl.gov/bcdaext/APSpy/trunk/src/testing/test_spec_macro.py $
# $Id: test_spec_macro.py 1281 2013-04-24 23:41:03Z jemian $
############ SVN repository information ###################


import APSpy.spec
import epics as PyEpics
```

```
import logging
from pprint import pprint


MOTOR_CONFIGURATION = '''
  samX   como:m1     sample X position (mm) + outboard
  samZ   como:m2     sample Z position (mm) + up
  phi    como:m3     sample rotation (deg)
  j1     como:m4     sample table N jack
  j2     como:m5     sample table SE jack
  j3     como:m6     sample table SW jack
'''

alldone = PyEpics.PV('como:alldone')

def wait_motors():
    while not alldone.get():
        APSpy.spec.sleep(1)
    logging.info("keep waiting for motor(s) to stop? " + str(alldone.get() != 1) )

def ImportMotorSymbols():
    exec( APSpy.spec.DefineMotorSymbols(APSpy.spec.mtrDB, make_global=True) )


def print_title(title):
    logging.info( 70*'=' )
    logging.info( title )
    logging.info( 70*'=' )


def setup():
    """exercise the definition routines"""
    print_title('setup')
    logging.info("*** define the motors we will use")
    #APSpy.spec.DEBUG = True

    # test applies when using development IOC *only*
    motors = MOTOR_CONFIGURATION.strip().splitlines()
    for row in motors:
        items = row.split()
        mne  = items[0]
        pv   = items[1]
        desc = ' '.join(items[2:])
        item = APSpy.spec.DefineMtr(mne, pv, desc)
        #print item, mne, pv, APSpy.spec.mtrDB[item], mne in APSpy.spec.mtrDB, mne in APSpy.spec.__d

    APSpy.spec.DefinePseudoMtr({
        # define pseudo motor position
        'jack': '(A[j1] + A[j2] + A[j3])/3.',
        # define motor movements in terms of pseudo motor target position
        'j1': 'A[j1] + T[jack] - A[jack]',
        'j2': 'A[j2] + T[jack] - A[jack]',
        'j3': 'A[j3] + T[jack] - ((A[j1] + A[j2] + A[j3])/3)',
        },'composite motion for j1,j2 & j3')
    APSpy.spec.DefinePseudoMtr({
        # define pseudo motor positions
        'samLX': 'cosd(A[phi])*A[samX] + sind(A[phi])*A[samZ]',
        'samLZ': '-sind(A[phi])*A[samX] + cosd(A[phi])*A[samZ]',
```

```
        # define motor movements in terms of pseudo motor target position
        'samX' : 'cosd(A[phi])*T[samLX] - sind(A[phi]) * T[samLZ]',
        'samZ' : 'sind(A[phi])*T[samLX] + cosd(A[phi]) * T[samLZ]',
        },' sample displacement in difractometer axes')

    ImportMotorSymbols()

    logging.info('*** ListMtrs()')
    logging.info(APSpy.spec.ListMtrs())

    import rst_table
    t = rst_table.Table()
    for mne in APSpy.spec.ListMtrs():
        logging.info('*** GetMtrInfo(%s)' % mne)
        info = APSpy.spec.GetMtrInfo(APSpy.spec.Sym2MtrVal(mne))
        if len(t.labels) == 0:
            t.labels = info.keys()
        t.rows.append([str(info[item]).strip() for item in t.labels])
    print t.reST(add_tabularcolumns=False)


def test_motors():
    """exercise some motor movement routines"""
    print_title('test_motors')

    logging.info("move all motors to 0")
    for mne in APSpy.spec.ListMtrs():
        mtr = APSpy.spec.Sym2MtrVal(mne)
        APSpy.spec.mv(mtr, 0.0)
    wait_motors()

    logging.info("*** perform some moves")
    logging.info("moving samX to 2.0");    APSpy.spec.umv( samX,   2.)
    logging.info("moving samZ to 0.5");    APSpy.spec.umv( samZ,   0.5)
    wait_motors()
    logging.info("moving samX by -1.0");   APSpy.spec.umvr(samX,  -1.)
    logging.info("moving j1   to 2.05");   APSpy.spec.umv( j1,     2.05)
    logging.info("moving j2   to 2.11");   APSpy.spec.umv( j2,     2.11)
    logging.info("moving j3   to 2.2");    APSpy.spec.umv( j3,     2.2)
    logging.info("moving phi  to 30.0");   APSpy.spec.umv( phi,    30.0)

    logging.info('Before move of jack')
    for mtr in 'j1','j2','j3','jack':
        print('  ', mtr, APSpy.spec.ReadMtr(APSpy.spec.Sym2MtrVal(mtr)))
    APSpy.spec.umv(jack,3.12)
    wait_motors()
    logging.info('After move of jack')
    for mtr in 'j1','j2','j3','jack':
        print '  ', mtr, APSpy.spec.ReadMtr(APSpy.spec.Sym2MtrVal(mtr))

    logging.info('\n\nX,Z = ' + str(APSpy.spec.wm(samX)))
    logging.info(APSpy.spec.wm(samZ))
    print 'sq', APSpy.spec.wm(samX)**2 + APSpy.spec.wm(samZ)**2
    for ang in (0, 30, 45, 90, 180, 270, -30):
        APSpy.spec.mv(phi, ang)
        wait_motors()
    logging.info('ang, LX,LZ = ' + str(ang))
    logging.info( str(APSpy.spec.wm(samLX)) + '  ' + str(APSpy.spec.wm(samLZ)) )
```

```
134     logging.info('sq ' + str((APSpy.spec.wm(samLX) **2 + APSpy.spec.wm(samLZ)**2)) )
135     APSpy.spec.sleep(1)
136 
137     logging.info('*** move jack, samLX & samLZ together')
138     APSpy.spec.mmv([
139               [jack,  0],
140               [samLX, 0],
141               [samLZ, 0]],)
142 
143     logging.info('*** move back in samLX & samLZ together in 5 steps')
144     APSpy.spec.MoveMultipleMtr([
145                           [samLX, 0.616025403784],
146                           [samLZ, 0.933012701892]],
147                          5)
148 
149     logging.info("move all motors back to 0")
150     wait_motors()
151     # refer to motor names in spec module namespace
152     APSpy.spec.mmv([
153               [samX, 0],
154               [samZ, 0],
155               [phi,  0],
156               [j1,   0],
157               [j2,   0],
158               [j3,   0],
159               ],)
160     wait_motors()
161 
162 
163 def show_motors():
164     """exercise motor introspection routines"""
165     print_title('show_motors')
166 
167     logging.info('*** ExplainMtr() by string')
168     for mne in APSpy.spec.ListMtrs():
169         mtr = APSpy.spec.Sym2MtrVal(mne)
170         logging.info( '\t'.join([mne,  str(APSpy.spec.wm(mtr)),  APSpy.spec.ExplainMtr(mtr)]) )
171     logging.info('*** ExplainMtr() by reference')
172     logging.info(str(samX)+'\t' + APSpy.spec.ExplainMtr(samX))
173     logging.info('*** wm(samX,samZ):')
174     logging.info(APSpy.spec.wm(samX, samZ))
175     logging.info('*** wa()')
176     APSpy.spec.wa()
177     logging.info('*** wa() long form')
178     APSpy.spec.wa(True)
179 
180 
181 def test_scalers():
182     """exercise scaler routines"""
183     print_title('test_scalers')
184     APSpy.spec.DefineScaler('ioc:scaler1', 16)
185     APSpy.spec.DefineScaler('ioc:scaler2', 8, index=1)
186     logging.info("Count on default scaler, default count time")
187     print APSpy.spec.ct()
188     print "Count on scaler 1, count time=2.5"
189     print APSpy.spec.ct(2.5, index=1)
190     print "Start async count on scaler 1, count time=5 sec"
191     APSpy.spec.count_em(5, index=1)
```

```
192     print "read immediately"
193     print APSpy.spec.get_counts()
194     sleep(1)
195     print "after a second"
196     print APSpy.spec.get_counts()
197     APSpy.spec.wait_count()
198     logging.info("after a  wait_count()")
199     logging.info(APSpy.spec.get_counts())
200
201
202 if __name__ == '__main__':
203     logging.basicConfig(format='%(asctime)s %(message)s',
204                         level=logging.INFO)
205     APSpy.spec.EnableEPICS()
206     setup()
207
208     test_motors()
209     show_motors()
210     #test_scalers()
211     logging.info("wait for all motors to complete their moves")
212     wait_motors()
213     print_title("all tests complete")
```

1.5 APSpy Source Code Documentation

SPEC Simulation module

1.5.1 *Module spec: SPEC-like emulation*

The Python functions in this module are designed to emulate similar commands/macros in SPEC or provide similar functionality. They require the PyEpics [4] package.

Motor interface routines

Description	Relative	Absolute
move motor	`mvr()`	`mv()`
move motor with wait	`umvr()`	`umv()`
move multiple motors [5]		`mmv()`
move multiple w/wait [2]		`ummv()`
where is this motor?		`wm()`
where are all motors?		`wa()`

[4] PyEpics: http://cars9.uchicago.edu/software/python/pyepics3/
[5] These commands implement capabilities not present in SPEC.

Scaler routines

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

More spec-like capabilities

description	command
Turn simulation mode on	`onsim()`
Turn simulation mode off	`offsim()`
array of motor positions	*A[]*
array of last count values	*S[]*

Routines not in spec

Routine	Description
`sleep()`	Delay for a specified amount of time
`EnableEPICS()`	Turns simulation mode on or off
`UseEPICS()`	Show if EPICS should be accessed
`DefineMtr()`	Define a motor to be accessed
`DefinePseudoMtr()`	Define pseudo motors from previously defined motors
`GetMtrInfo()`	Retrieves all motor info from a key
`DefineMotorSymbols()`	Used to define motor symbols in caller's namespace
`DefineScaler()`	Define a scaler to be accessed
`GetScalerInfo()`	Retrieves all scaler info from an index
`ListMtrs()`	Returns a list of motor symbols
`Sym2MtrVal()`	Retrieves the motor entry key from a symbol
`ExplainMtr()`	Retrieves the motor description from a key or symbol
`ReadMtr()`	Returns the motor position from a key
`PositionMtr()`	Moves a motor
`MoveMultipleMtr()`	Move several motors together
`GetScalerLastCount()`	Returns the last set of counts that have been read for a scaler
`GetScalerLastTime()`	Returns the counting time for the last use of a scaler
`GetScalerLabels()`	Returns the labels that have been retrieved for a scaler
`SetMon()`	Set the monitor channel for the scaler
`GetMon()`	Return the monitor channel for the scaler
`SetDet()`	Set the main detector channel for the scaler
`GetDet()`	Return the main detector channel for the scaler
`setCOUNT()`	Sets the default counting time
`initElapsed()`	Initialize the elapsed time counter
`setElapsed()`	Update the elapsed time counter
`setRETRIES()`	Sets the maximum number of EPICS retries
`setDEBUG()`	Sets debugging mode (printing lots of stuff) on or off

Global variables

As described below, these variables can be read from outside of the package, but should be set with care.

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 APSpy.spec import *`, as setting the variable directly has problems:

This will sort-of work:

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

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

but this will not work:

```
>>> from APSpy.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.

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 or `sleep()` is called.

SIMSPEED When in simulation mode, scripts are sped up by decreasing delays (calls to `spec.sleep()`) by a factor of `SIMSPEED`. Be sure to change `spec.SIMSPEED` if you want to change this.

A[]

A As in spec, A[mtr1] provides the current position of mtr1. A is not actually implemented as a global array, but can be indexed as one.

S[]

S As in spec, S[*i*] provides the last read intensity from scaler channel *i*. This is a python list and is thus indexed starting at 0. The first channel, `S[0]`, is expected to be configured as the count-time reference channel.

Complete Function Descriptions

The functions available in this module are listed below.

exception APSpy.spec.**APSpyException**
any exception from this module

exception APSpy.spec.**APSpyMotorException**
any motor exception

exception APSpy.spec.**APSpyScalerException**
any scaler exception

exception APSpy.spec.**APSpyUndefinedMotorException**
requested a mne or mtrsym that was not defined

exception APSpy.spec.**APSpyUndefinedScalerException**
(one instance occurs) - needs work to demonstrate usage

APSpy.spec.**DefineMotorSymbols**(*db={}*, *make_global=False*)
make definitions of the motor symbolic names

Returns a string listing the motor symbols and values. This string can be executed in the local namespace (by `exec()`) to define (or redefine) these names locally for convenience. This is recommended for interactive (command-line session) use only.

```
>>> from spec import *
>>> DefineMtr('samX', 'como:m1',  'sample X position (mm) + outboard')
>>> DefineMtr('samZ', 'como:m2',  'sample Z position (mm) + up')
>>> samX
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'samX' is not defined
>>> exec( DefineMotorSymbols(mtrDB) )
>>> samX
'mtr1000'
>>>
```

Note: Using `exec()` is the only known and reliable way to import the motor symbol definitions into the local namespace.

To import the motor symbols into a module's *global* namespace, (non-interactive, as used in a script or macro file) one way would be to include the following function in the module:

```
import spec
# ...
def ImportMotorSymbols():
    exec( spec.DefineMotorSymbols(spec.mtrDB, make_global=True) )
# ...
ImportMotorSymbols()
```

APSpy.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 `APSpyException` 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 APSpy.spec import * ‘‘ python command to import these routines into the current module’s name space, it is necessary to repeat this command after `DefineMtr()` 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 APSpy.spec
>>> APSpy.spec.EnableEPICS()
>>> APSpy.spec.DefineMtr('mtrXX1','ioc1:mtr98','Example motor #1')
>>> APSpy.spec.DefineMtr('mtrXX2','ioc1:mtr99','Example motor #2')
>>> APSpy.spec.mv(spec.mtrXX1, 0.123)
```

It is also possible to mix the two styles:

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

`APSpy.spec.`**`DefinePseudoMtr`**(*inpdict*, *comment=''*)

Define one or more pseudo motors in terms of previously defined motors. Adds the new pseudo motor definition(s) to the motor table.

Parameters

- **inpdict** (*dict*) – defines a dictionary that defines pseudo motor postions in terms of real motor positions and maps pseudo-motor target positions into real motor target positions. Dictionary entries that do not correspond to previously defined motors are used to define new pseudo-motors.
- **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).

For computations in the dictionary, motor positions may be referenced in one of two ways, A[mtr] or T[mtr]. A[mtr] provides the actual position of the motor while T[mtr] provides the target position for the move, i.e., the value of the motor or pseudo-motor after the move, if it will be changed. For definitions of pseudo motors, use of A[] is usually correct, but for entries that compute target positions of real motors, one almost always wishes to use T[] to compute from target positions (this is most important for use with `MoveMultipleMtr()`, where

multiple target positions are updated prior to any motor movement.). See the examples, below. Note also that these expressions are computed in the spec namespace, so the prefix 'spec.' on motor names (etc.) is not needed.

Note that all the routines in math and numpy are available for use in these calculations (but must be prefixed by math or numpy or np (such as `math.log10()` or `np.exp2()` or `numpy.exp2()` or constant `math.pi`). In addition, for convenience the following functions are also defined without a prefix: `sind()` (sine of angle in degrees), `cosd()` (cosine of angle in degrees), `tand()` (tangent of angle in degrees), `asind()` (inverse sine, returns angle in degrees), `acosd()` (inverse cosine, returns angle in degrees), `atand()` (inverse tangent, returns angle in degrees), `abs()`, `sqrt()` and `exp()`.

Examples:

```
>>> DefineMtr('j1','1idc:j1','sample table N jack')
>>> DefineMtr('j2','1idc:j2','sample table SE jack')
>>> DefineMtr('j3','1idc:j3','sample table SW jack')
>>> APSpy.spec.DefinePseudoMtr({
...     # define pseudo motor position
...     'jack': '(A[j1] + A[j2] + A[j3])/3.',
...     # map motor movements in terms of pseudo motor target position
...     'j1': 'A[j1] + T[jack] - A[jack]',
...     'j2': 'A[j2] + T[jack] - A[jack]',
...     'j3': 'A[j3] + T[jack] - ((A[j1] + A[j2] + A[j3])/3)',
...     })
```

The above definition a new pseudo motor, *jack* is defined in terms of three motors that are already defined, *j1*, *j2*, and *j3*. Note that 'T[jack] - A[jack]' (or equivalently 'T[jack] - ((A[j1] + A[j2] + A[j3])/3)', both are used here as a pedagogical example) computes the difference between the target position for jack and its current position and then adds that difference to the positions for *j1*, *j2*, and *j3*, thus, the motors move relative to their initial positions. Note that the comments placed in the input are only a guide to the reader, the fact that 'jack' is new and *j1*, *j2*, and *j3* are defined indicates that *jack* is to be defined.

```
>>> DefineMtr('samX','1idc:m77','sample X position (mm) + outboard')
>>> DefineMtr('samZ','1idc:m78','sample Z position (mm) + up')
>>> DefineMtr('phi','1idc:mphi','sample rotation (deg)')
>>> APSpy.spec.DefinePseudoMtr({
...     # define pseudo motor positions
...     'samLX': 'cosd(A[phi])*A[samX] + sind(A[phi])*A[samZ]',
...     'samLZ': '-sind(A[phi])*A[samX] + cosd(A[phi])*A[samZ]',
...     # define motor movements in terms of pseudo motor target position
...     'samX' : 'cosd(T[phi])*T[samLX] - sind(T[phi]) * T[samLZ]',
...     'samZ' : 'sind(T[phi])*T[samLX] + cosd(T[phi]) * T[samLZ]',
...     })
```

In the above definition two new pseudo motors, *samLX* and *samLZ* are defined in terms of three motors that are already defined, *samX*, *samZ*, and *phi*. This maps the axes defined by the sample translations *samX*, *samZ* which are rotated by motor *phi* relative to the diffractometer coordinate system into a static frame of reference. Note that use of T[samLX] and T[samLY] is necessary in the latter expressions, but A[phi] could be used in place of T[phi] as long as one does not try to move phi along with *samLX* and/or *samLY* in a single call to `MoveMultipleMtr()`.

As described for `DefineMtr()`, if you will use the '' from APSpy.spec import * '' python command to import these routines into the current module's name space, it is necessary to repeat this import command after defining all motors and pseudo motors to import the newly defined global symbols into the top namespace.

`APSpy.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 APSpy.spec import *
>>> EnableEPICS()
>>> DefineScaler('id1:scaler1',16)
>>> DefineScaler('id1: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 APSpy.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]
```

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

Call to enable communication with EPICS.

This must be called to enable communication with EPICS before initializing motors. If not called then APSpy will function in simulation mode only. If the PyEpics module cannot be loaded, then this function has no effect.

Parameters **state** (*bool*) – if False is specified, then EPICS communication is disabled (default value, True).

APSpy.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

APSpy.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

APSpy.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

APSpy.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

APSpy.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

APSpy.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

APSpy.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

APSpy.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

APSpy.spec.**ListMtrs**()
Returns a list of the variables defined as motor symbols.

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

APSpy.spec.**MoveMultipleMtr**(*mtrposlist*, *nsteps=1*, *wait=True*)
Launch movement of several motors together. If a motor would be moved more than one time (for example because it is referenced in more than on pseudo-motor), only the last move is actually performed. The target for each motor is included in subsequent computations, so that when motor positions are computed from postions of more than one pseudo-motor, the performed move will represent the positions from the cummulative move of all previous motors. To deal with the case where motor speeds or movements are unequal, the requested moves can be broken down into a series of nsteps steps, where each motor will be moved an increment of 1/nsteps times the total requested change in position. This will not keep the movement on exactly the requested trajectory, but it will stay close.

Parameters

- **mtrposlist** (*list*) – A list of motor keys and target positions, for example [(samLX,1.1),(samLZ,0.25)]
- **nsteps** (*int*) – the number of steps to be used to break down the requested move. The default, 1, means that all motors are launched at the same time for the entire requested movement range, but a value of 2 indicates that all motors will launched to the mid-point

of the requested movement range and only after all motors have reached that point, will the subsequent set of moves be started.

- **wait** (*bool*) – When *wait* is False, moves are started, but the routine returns immediately, but *wait* is True (default), the routine returns after all motors have stopped moving. If :*nsteps* is greater than 1, this parameter is ignored and the routine returns only after all requested moves are completed.

Example:

```
>>> MoveMultipleMtr([(samLX,1.1),(samLZ,0.25)],5,wait=True)
```

APSpy.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 (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.

APSpy.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).

APSpy.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

- **Detector** (*int*) – channel number. If omitted the Detector 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.

APSpy.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.

APSpy.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 (such as 'phi')

Returns **str** motor value (such as *mtr1002*)

Raises *APSpyException* if the value does not correspond to a motor entry.

APSpy.spec.**UseEPICS**()

Show if use of EPICS is allowed or disabled, see EnableEPICS(), onsim() and offsim().

Returns True if PyEpics has been loaded and enabled (see EnableEPICS()) and simulate mode is False (see onsim() and offsim()), False otherwise.

APSpy.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** (*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()
```

APSpy.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]
```

APSpy.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 instananeously

Returns a list of channels values

Example:

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

APSpy.spec.**get_mtrsym**(*key*)
return motor mtrsym given either mtrsym or mne

mne: motor menmonic symbol, such as *phi*

mtrsym: motor symbol, such as *mtr1002*. *mtrsym* is the key used to index mtrDB

Parameters **key** (*str*) – either a motor mne or a mtrsym

Returns str a motor key

Raises APSpyUndefinedMotorException if mtrsym is not found

APSpy.spec.**initElapsed**()
Initialize the elapsed time counter

APSpy.spec.**mmv**(*mtrposlist*, *nsteps=1*, *wait=False*)
Launch movement of several motors together. By default, does not wait for all motion to complete. See the equivalent function, MoveMultipleMtr(), for a complete description.

Parameters

- **mtrposlist** (*list*) – A list of pairs of motor keys and target positions
- **nsteps** (*int*) – the number of steps to be used to break down the requested move. The default, 1, means that all motors are launched at the same time for the entire requested movement range, but a value of 2 indicates that all motors will launched to the mid-point of the requested movement range and only after all motors have reached that point, will the subsequent set of moves be started.
- **wait** (*bool*) – When *wait* is False, moves are started, but the routine returns immediately, but *wait* is True, the routine returns after all motors have stopped moving. the default is to not wait. Note that if *nsteps* is greater than 1, this parameter is ignored and the routine returns only after all requested moves are completed.

Example:

```
>>>  mmv([(samLX,1.1),(samLZ,0.25)])
```

APSpy.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)
```

APSpy.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)
```

APSpy.spec.**offsim**()
Turns simulation mode off. Note that unlike `EnableEPICS()`, `onsim()` and `offsim()` can be used at any time.

APSpy.spec.**onsim**()
Turns simulation mode on. Note that unlike `EnableEPICS()`, `onsim()` and `offsim()` can be used at any time.

APSpy.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.

APSpy.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.

APSpy.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 scalers or `sleep()` is called.

Returns the elapsed time in sec (float)

APSpy.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.

APSpy.spec.**sleep**(*sec*)
Causes the script to delay for *sec* seconds.

This method is replaced when plotting is loaded by an alternate method (see `sleepWithYield()` in `macros._makePlotWin()`).

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

APSpy.spec.**ummv**(*mtrposlist*, *nsteps=1*, *wait=True*)

Launch movement of several motors together. By default, waits for all motion to complete. See the equivalent function, MoveMultipleMtr(), for a complete description.

Parameters

- **mtrposlist** (*list*) – A list of pairs of motor keys and target positions
- **nsteps** (*int*) – the number of steps to be used to break down the requested move. The default, 1, means that all motors are launched at the same time for the entire requested movement range, but a value of 2 indicates that all motors will launched to the mid-point of the requested movement range and only after all motors have reached that point, will the subsequent set of moves be started.
- **wait** (*bool*) – When *wait* is False, moves are started, but the routine returns immediately, but *wait* is True (default), the routine returns after all motors have stopped moving. If *nsteps* is greater than 1, this parameter is ignored and the routine returns only after all requested moves are completed.

Example:

```
>>>  ummv([(samLX,1.1),(samLZ,0.25)])
```

APSpy.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)
```

APSpy.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)
```

APSpy.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()
===== ========
motor position
===== ========
samX  1.0
samZ  0.5
===== ========
>>> wa(True)
===== ======== ================================
motor position description
===== ======== ================================
samX  1.0      sample X position (mm) + outboard
samZ  0.5      sample Z position (mm) + up
===== ======== ================================
```

APSpy.spec.**wait_count**()
Wait for scaler to finish, must follow count_em

Returns None

Example:

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

APSpy.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]
```

SPEC Simulation: Macros module

1.5.2 *Module macros: Additional SPEC-like emulation*

Python functions listed below are designed to implement functionality for data collection similar to that available in spec. Routines are divided into sections, *General Purpose Routines*, *Logging*, *Plotting*, *Monitoring* and *Macros specific to 1-ID*

General Purpose Routines

Note that the `ascan()` and `dscan()` are affected by what is used in `SetDet()` and possibly `spec.SetMon()` as well as the *Logging* configuration.

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

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

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

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.

Monitoring

Monitoring of PVs is used to record values of selected PVs when any designated PV changes. Optionally, only when that PV changes to a specific value or the recording can be limited to not occur more than a maximum frequency. It may be best to perform monitoring in a process separate from the one making changes to EPICS PVs.

Monitoring routines	Description
`DefMonitor()`	Set up a PV to be monitored
`StartAllMonitors()`	Start the monitoring operation

Monitor definition examples:

```
>>> spec.EnableEPICS()
>>> macros.DefMonitor('/tmp/tst','1ide1:m1.VAL',
...                ('1id:scaler1.S2','1id:scaler1.S3','1ide1:m1.RBV','1ide1:m1.VAL')
...                )
>>> macros.StartAllMonitors()
```

This will report the values of four PVs every time that PV 1ide1:m1.VAL is changed.

```
>>> macros.DefMonitor('/tmp/tst','1ide1:m1.RBV',
...                ('1id:scaler1.S3','1ide1:m1.RBV','1ide1:m1.VAL'),
...                pvvalue=0.0)
>>> macros.StartAllMonitors()
```

This will report three PVs, but only when PV 1ide1:m1.RBV is changed to 0.0 (within 0.00001)

```
>>> macros.DefMonitor('/tmp/tst','1ide1:m1.RBV',
...               ('1id:scaler1.S2','1id:scaler1.S3','1ide1:m1.RBV','1ide1:m1.VAL'),
...                delay=1.0)
>>> macros.StartAllMonitors()
```

This will report three PVs, every time that PV 1ide1:m1.RBV is changed, but only a maximum of one change will be reported each second.

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

Complete Function Descriptions

The functions available in this module are listed below.

APSpy.macros.**Cclose**()
Close 1-ID fast shutter in B hutch

APSpy.macros.**Copen**()
Open 1-ID fast shutter in B hutch

APSpy.macros.**DefMonitor**(*fileprefix*, *pv*, *monitorlist*, *pvvalue=None*, *delay=None*)
Write values of PVs in monitorlist each time that PV pv changes, values are written to a file named by fileprefix + timestamp optionally, values are written only if the PV is set to value pvvalue (if not None) and optionally only recording the first change in a period of delay seconds (if not None):

Monitoring starts when StartAllMonitors() is called.

Parameters

- **fileprefix** (*str*) – defines name of file to use
- **pv** (*str*) – PV to monitor
- **monitorlist** (*list*) – list of PVs to report
- **pvvalue** (*?*) – report monitored PV only if this value is obtained
- **delay** (*float*) – do not log changes more frequently than this frequency in seconds

see *Monitoring* for an example of use.

APSpy.macros.**DefineLoggingPlot**(*tablbl*, *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 be 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

- **tablbl** (*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 `APSpy.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 `APSpy.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 APSpy.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

APSpy.macros.**InitLoggingPlot**()

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

APSpy.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

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

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

Parameters **FitClass** (*class*) – a class that defines a minumum 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])
```

APSpy.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

APSpy.macros.**SendTextEmail**(*recipientlist*, *msgtext*, *subject='APSpy auto msg'*, *recipientname=None*, *senderemail='1ID@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 "APSpy 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 "1ID@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')
```

APSpy.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

APSpy.macros.**ShowPlots**()
Pause to show plot screens. Call this at the end of a script, if needed.

APSpy.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.

APSpy.macros.**StartAllMonitors**(*sleep=True*)
Start the monitoring defined in DefMonitor. Optionally delay until control-C is pressed. The control-C operation closes all files and clears the monitoring information.

Parameters **sleep** (*bool*) – if True (default) start an infinite loop of one second delays

see *Monitoring* for an example of use.

APSpy.macros.**UpdateLoggingPlots**()
Read all current values in plot and display in plots

see *Plotting* for an example of use.

APSpy.macros.**UserIn**(*parname*, *default*, *typ*)
Prompt a user for input.

For reasons unclear, this is not raising an Exception on Control-C on Linux, but Control D does raise an exception, so use that.

Parameters

- **parname** (*str*) – a string to be given to the user to tell them what to input
- **default** (*(any)*) – a default value, if no input is provided, use None to force user input
- **typ** (*type*) – a data type, such as int, float, or str

Returns the value provided by the user in the selected type

Examples:

```
>>> UserIn('test',2.0,float)
test (2.0): x
Invalid, try again
test (2.0): 3.1
3.1

>>> UserIn('test',None,float)
test (None):
Invalid, try again
test (None): 4
4.0

>>> UserIn('test',2,int)
test (2): 2.0
Invalid, try again
```

```
test (2): 2
2
```

APSpy.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.

APSpy.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.

APSpy.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.

APSpy.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.

APSpy.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 be defined (see DefineScaler()). The default (0) is used if not specified.

see *Logging* for an example of use.

APSpy.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).

APSpy.macros.**beep_dac**(*beeptime=1.0*)
Set the 1-ID beeper on for a fixed period, which defaults to 1 second uses PV object beeper (defined as 1id:DAC1_8.VAL) makes sure that the beeper is actually turned on and off throws exception if beeper fails

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

APSpy.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.

APSpy.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.

APSpy.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`).

`APSpy.macros.`**`init_logging`**`()`
Initialize the list of data items to be logged

see *Logging* for an example of use.

`APSpy.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()`

`APSpy.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.

`APSpy.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.

`APSpy.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.

`APSpy.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.

`APSpy.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 be defined (see `DefineScaler()`). The default (0) is used if not specified.

see *Logging* for an example of use.

`APSpy.macros.`**`show_logging`**()

Show the user the current logged items

`APSpy.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()

`APSpy.macros.`**`shutter_sweep`**()

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

`APSpy.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'
```

APSpy.macros.**write_logging_header**(*filename=''*)
Write a header for parameters recorded when `write_logging_parameters()` is called.

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

see *Logging* for an example of use.

APSpy.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 be used for output. If not specified, the output is sent to the terminal window.

see *Logging* for an example of use.

Area Detector Implementation Module

1.5.3 *Module AD: Area-Detector access*

These routines provide a general framework for control of Area Detectors (AD). Also included is a specific implementation for the commonly-used area detectors in 1-ID, which are summarized in *Defined Commands*, below.

Kludged: is added to all string caput calls (in `AD_set()`)

Detector Access Routines

These routines are used to change or read parameters for detectors, or to show information about how these commands have been configured.

Access routines	Description
`AD_get()`	Read an area detector parameter
`AD_set()`	Set an area detector parameter
`AD_acquire()`	Set the filename, count time and frames and collect
`AD_done()`	Test if the detector(s) have completed data collection
`AD_show()`	Shows commands options for `AD_get()` and `AD_set()`
`AD_cmds()`	Returns but does not print commands options for `AD_get()` and `AD_set()`

Detector Setup Routines

These routines are used inside the module and are likely only changed by beamline staff.

Setup routines	Description
`DefineAreaDetector()`	Define an area detector for later use
`defADcmd()`	Define parameters to set up an area detector command

Defined Commands

Below are lists of the commands that can be used in `AD_get()` and `AD_set()`, for each detector (*GE*, *Retiga*, *ScintX*):

Defined commands for GE detectors

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None
trigger_mode	Triggering: Angio=0, Rad=1, UserSingle=2, MultiDet=3	int	(0, 1, 2, 3)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None
trigger_mode	Triggering: Angio=0, Rad=1, UserSingle=2, MultiDet=3	int	(0, 1, 2, 3)

Defined commands for Retiga detectors

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
transfer	Transfer EPICS values to FPGA	int	(1,)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
transfer	Transfer EPICS values to FPGA	int	(1,)
trigger_mode	Triggering: free=0, edge: Hi=1, low=2, Pulse: Hi=3, low=5, soft=5, Strobe: Hi=6, low=7, last=8	int	0 <= val <= 8

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
transfer	Transfer EPICS values to FPGA	int	(1,)
trigger_mode	Triggering: free=0, edge: Hi=1, low=2, Pulse: Hi=3, low=5, soft=5, Strobe: Hi=6, low=7, last=8	int	0 <= val <= 8

Defined commands for ScintX detectors

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None
trigger_mode	Triggering mode: Internal=0, External=1	int	(0, 1)

command	Explanation	data type	validator
acquire	Trigger Data collection	int	(1,)
acquire_time	Data collection time/frame (sec)	float	val > 0
autoincrement	Overwrite current file or increment filenumber	int	(0, 1)
autosave	Save images to file: No=0, Yes=1	int	(0, 1)
autostore	Save images to file: No=0, Yes=1	int	(0, 1)
filename	Set image filename	str	None
filenumber	Next file number	int	val > 0
filepath	Full path for data file	str	None
frames	number of frames	int	0 < val < 300
state	Data collection state	int	None
trigger_mode	Triggering mode: Internal=0, External=1	int	(0, 1)
video_mode	Format: 0=4024x2680, 2=2012x1340	int	(0, 2)

Complete Function Descriptions

The functions available in this module are listed below.

APSpy.AD.**AD_acquire**(*detsyms*, *filename*, *counttime*, *frames=1*, *wait=False*)

Set parameters for an area detector and collect image(s)

Parameters

- **detsyms** (*object*) – An area detector variable (or name as a string), as defined in `DefineAreaDetector()`. Alternately, a list of area detectors variable or names (as strings) can be supplied.
- **filename** (*str*) – The name of the data file to be used
- **counttime** (*float*) – The data collection pre frame time to be used (sec)
- **frames** (*int*) – The number of images to be recorded
- **wait** (*bool*) – If False (default) return immediately; if True, return after waiting the appropriate amount of time and when the `state` command (if defined) indicates the data collection is done.

APSpy.AD.**AD_cmds**(*detsym=None*)

Returns all the commands defined for a particular detector, or with any detector. Does not print.

Parameters **detsym** (*object*) – An area detector variable (or name as a string), as defined in `DefineAreaDetector()`. Default is to only list commands that can be used with all detectors.

Returns a list of allowed commands

APSpy.AD.**AD_done**(*detsyms*, *wait=True*)

Test and optionally wait for the detector(s) have completed data collection

Parameters

- **detsyms** (*object*) – An area detector variable (or name as a string), as defined in DefineAreaDetector(). Alternately, a list of area detectors variable or names (as strings) can be supplied.
- **wait** (*bool*) – If False test and return immediately; if True (default), return after the state command (if defined) indicates the data collection is done for each detector.

Returns True if all detector(s) are done; False is wait is False and any detectors are not done; or None if after 30 seconds, any detector is not complete

APSpy.AD.**AD_get**(*detsyms*, *cmd*, *ignoreOK=False*)

Read a parameter from an area detector

Parameters

- **detsyms** (*object*) – An area detector variable (or name as a string), as defined in DefineAreaDetector() or a list of area detector variables or strings. If a list (or tuple) of detectors is used, the function may return a list of values (but only if they differ.)
- **cmd** (*str*) – a command string that has been defined using defADcmd()
- **ignoreOK** (*bool*) – if ignoreOK is False (default) an exception will be raised if command cmd is not defined for a detector. If True, the command will be ignored

Returns the as-read parameter. The type will be determined by the PV associated with the command. If detsyms is a list and the read values differ, then a list of values is returned. Otherwise, only the (common) value is return.

Examples:

```
>>> AD.DefineAreaDetector('GE1', 'GE', 'GE1:cam1')
>>> val = AD.AD_get(AD.GE1,'acquire_time')
```

or

```
>>> val = AD.AD_get('GE1','acquire_time')
```

also

```
>>> hydra = (AD.GE1,AD.GE2,AD.GE3,AD.GE4)
>>> val =  AD.AD_get(hydra,'trigger_mode')
>>> try:
>>>     if len(val) == 4 and not isinstance(val,str):
>>>         print 'values disagree'
>>> except TypeError:
>>>     pass
```

APSpy.AD.**AD_set**(*detsyms*, *cmd*, *value*, *ignoreOK=False*)

Set a parameter for an area detector. *This routine has been patched to add a to strings, to fix a problem in EPICS.*

Parameters

- **detsyms** (*object*) – An area detector variable (or name as a string), as defined in DefineAreaDetector(). Alternately, a list of area detectors variable or names (as strings) can be supplied. The command (cmd) must be defined for all supplied detectors, or an exception occurs.
- **cmd** (*str*) – a command that has been defined using defADcmd()

- **value** (*str*) – The value to set the parameter. This value will be set to the type defined for the command from `defADcmd()` if possible and will be checked against the enumeration range, if one is supplied. If the type conversion fails or the check fails, an exception is raised.
- **ignoreOK** (*bool*) – if ignoreOK is False (default) an exception will be raised if command cmd is not defined for a detector. If True, the command will be ignored

Returns the as-read parameter. The type will be determined by the PV associated with the command.

Examples:

```
>>> AD.DefineAreaDetector('GE1', 'GE', 'GE1:cam1')
>>> val = AD.AD_set(AD.GE1,'acquire_time',3)
```

or

```
>>> val = AD.AD_set('GE1','acquire_time',3)
```

also

```
>>> hydra = (AD.GE1,AD.GE2,AD.GE3,AD.GE4)
>>> val =  AD.AD_set(hydra,'trigger_mode',0)
```

APSpy.AD.**AD_show**(*detsym=None*, *allowprint=True*)

Shows all the commands defined for a particular detector, or with any detector.

Parameters

- **detsym** (*object*) – An area detector variable (or name as a string), as defined in `DefineAreaDetector()`. Default is to only list commands that can be used with all detectors.
- **allowprint** (*bool*) – If True (default) prints a list of the allowed commands

Returns a list of allowed commands

APSpy.AD.**DefineAreaDetector**(*detsym*, *detectortype*, *controlprefix*, *imageprefix=None*, *comment=''*)

Define an area detector for use in this module

Parameters

- **detsym** (*str*) – a symbolic name for the detector. 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.
- **detectortype** (*str*) – the type of the detector. This must match one of the entries in global variable detectorTypeList (case sensitive).
- **controlprefix** (*str*) – the prefix for the detector PV (dev:camN). Omit the detector record field names (.NumImages, etc.). Inclusion of a final colon (':') is optional.
- **imageprefix** (*str*) – the prefix for the detector PV (dev:fmt). Omit the detector record field names (.FileNumber, etc.). Inclusion of a final colon (':') is optional. If not specified, defaults to the value for controlprefix
- **comment** (*string*) – a optional human-readable text field that describes the detector.

Returns detector object created for the detector

Example:

```
>>> DefineAreaDetector('GE1', 'GE', 'GE1:cam1', comment='bottom')
>>> DefineAreaDetector('GE2', 'GE', 'GE2:cam1', comment='left')
>>> DefineAreaDetector('GE3', 'GE', 'GE3:cam1', comment='top')
>>> DefineAreaDetector('GE4', 'GE', 'GE4:cam1', comment='right')
>>> DefineAreaDetector('ScintX', 'ScintX', 'ScintX:cam1', 'ScintX:TIFF1:')
>>> DefineAreaDetector('Retiga1', 'Retiga', 'QIMAGE1:cam1:', 'QIMAGE1:TIFF1:')
>>> DefineAreaDetector('Retiga2', 'Retiga', 'QIMAGE2:cam1:', 'QIMAGE2:TIFF1:')
```

APSpy.AD.**defADcmd**(*command*, *setsuffix*, *readsuffix*, *comment=''*, *valtyp=<type 'int'>*, *det=None*, *enum=None*)

This is called to create a table of actions to be used for writing to detectors. This will normally only be used by beamline staff and only inside this routine. Define detector-specific commands first, if they should take precedence over generic ones.

Parameters

- **command** (*str*) – A string to be used in `AD_get()` and `AD_set()` to be used to read or set an area detector parameter
- **setsuffix** (*str*) – The PV suffix to be used to set the parameter. This is appended to the end of controlprefix (if setsuffix begins with one % sign) or imageprefix (if setsuffix begins with twp % signs). If this is blank, the PV cannot be set.
- **readsuffix** (*str*) – The PV suffix to be used to read the parameter. This is appended to the end of controlprefix (if readsuffix begins with one % sign) or imageprefix (if readsuffix begins with twp % signs). If this is blank, the PV cannot be read.
- **comment** (*string*) – a optional human-readable text field that describes the command.
- **valtyp** (*type*) – a data type for the PV. Should be str, float or int (default)
- **detectortype** (*str*) – The type of the detector, if the command is not generic. The default is to define a command that can be used with all area detectors.
- **enum** (*str*) – A list of allowed values for the command, or a statement that must evaluate as True for the value to be accepted, typically a logical test on variable val. The default is allow all values.

  enum examples:

  `enum=(0,1,2)` – defines three specific allowed values (0, 1 and 2). No others are valid.

  `enum='val > 0'` – requires that the value must be greater than 0.0

  `enum='0 <= val <= 10'` – requires the value be 0 or 10 or any value in between

Examples:

```
>>> # GE detector specific
>>> defADcmd('trigger_mode', '%TriggerMode', '%TriggerMode_RBV',
...    'Triggering: Angio=0, Rad=1, UserSingle=2, MultiDet=3',
...    det='GE', enum=(0,1,2,3))
>>> defADcmd('state', '', '%DetectorState_RBV', 'Data collection state', det='GE')
>>> defADcmd('autostore', '%%AutoStore', '%%AutoStore_RBV', 'Save images to file: No=0, Yes=1',
...    det='GE', enum=(0,1)) # overrides generic, below
>>> defADcmd('autosave', '%%AutoStore', '%%AutoStore_RBV', 'Save images to file: No=0, Yes=1',
...    det='GE', enum=(0,1))
>>> #ScintX detector specific
>>> defADcmd('trigger_mode', '%TriggerMode', '%TriggerMode_RBV',
...    'Triggering mode: Internal=0, External=1',
...    det='ScintX', enum=(0,1))
```

```
>>> defADcmd('video_mode', '%CCDVideoMode', '%CCDVideoMode_RBV',
...    'Format: 0=4024x2680, 2=2012x1340', det='ScintX',enum=(0,2))
>>> #Retiga detector specific
>>> defADcmd('transfer', '%qInitialize', '', 'Transfer EPICS values to FPGA', det='Retiga', enum
>>> defADcmd('trigger_mode', '%TriggerMode', '%TriggerMode_RBV',
...     'Triggering: free=0, edge: Hi=1, low=2, Pulse: Hi=3, low=5, soft=5, Strobe: Hi=6, low=7,
...     det='Retiga', enum='0 <= val <= 8')
>>> # Generic
>>> defADcmd('acquire', '%Acquire', '', 'Trigger Data coll.',enum=(1,))
>>> defADcmd('acquire_time', '%AcquireTime', '%AcquireTime_RBV', 'Data coll. time (sec)', float,
>>> defADcmd('frames', '%NumImages', '%NumImages_RBV', 'number of frames (int)', enum='0 < val <
>>> defADcmd('filename', '%%FileName', '%%FileName_RBV', 'Set data filename', str)
>>> defADcmd('filenumber', '%%FileNumber', '%%FileNumber_RBV', 'Next file number', int)
>>> defADcmd('autoincrement', '%%AutoIncrement', '%%AutoIncrement_RBV',
...   'Overwrite current file or increment filenumber', int, enum=(0,1))
>>> defADcmd('filepath', '%%FilePath', '%%FilePath_RBV', 'Full path for data file',str)
```

GE Image processing module

1.5.4 *Module GE: GE Image processing*

This is a module for reading files and quick processing of data from the GE angiography detector in use at sector 1. It requires the NumPy package. It does not require the PyEpics package. Most functions in this module work directly with data files created by the GE detectors, with the exceptions of `PlotGEimage()` and `PlotROIsums()`, which plot images and ROI values, respectively, from other functions in this module.

Summary

Routines	Description
`Count_Frames()`	Determine the number of frames in a GE image file
`getGEimage()`	Read a single entire GE image file
`getGE_ROI()`	Read a section (region of interest) of a GE image file
`PlotGEimage()`	Plot an image or ROI
`sumGE_ROIs()`	Report the average intensity for ROIs in a GE image frame
`sumAllGE_ROIs()`	Reports the average intensity for ROIs for all frames in a file
`PlotROIsums()`	Plots the ROIs values from `sumAllGE_ROIs()`

Complete Function Descriptions

The functions available in this module are listed below.

APSpy.GE.**Count_Frames**(*filename*)

Determine the number of frames in a GE file by looking at the file size.

Parameters **filename** (*str*) – The filename containing the as-recorded GE images

Returns the number of frames (int).

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.Count_Frames(ifil)
220
```

APSpy.GE.**PlotGEimage**(*img*, *title*, *tablbl*, *plotlist*, *region=None*, *size=(700, 700)*, *imgwin=None*)

Create a plot of an image in tabbed window

Parameters

- **img** (*array*) – An image, as a numpy array or matplotlib compatible object. Usually this will be created by `getGEimage()` or `getGE_ROI()`.
- **title** (*str*) – A string with a title for the window
- **tablbl** (*str*) – A string with the title for the new tab (should be short)
- **plotlist** (*list*) – A list of _ImagePlot objects. As new plots are created in this routine they are added to this list. The list is used to assign color maps.
- **region** (*list*) – A list for four numbers which describes the ROI location for use in adding offsets for the plot axes labeling. The numbers are:

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

  The default is to label the pixels starting from zero.
- **size** (*list*) – A list, tuple or wx.size object with the size of the window to be created in pixels. The default is (700,700)
- **imgwin** (*object*) – A plotnotebook object that has been created using `plotnotebook.MakePlotWindow()`, usually in a prior call to `PlotGEimage()`. A value of None (default) causes a new frame (window) to be created.

Returns A reference to the plot window (a plotnotebook object), which will be either imgwin or the new one created in `plotnotebook.MakePlotWindow()`.

Examples:

```
>>> import plotnotebook
>>> import GE
>>> plotlist = []
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> img = GE.getGEimage(ifil,2)
>>> imgwin = GE.PlotGEimage(img,'image window','full image',plotlist)
>>> plotnotebook.ShowPlots()

>>> import plotnotebook
>>> import GE
>>> plotlist = []
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> ROI = GE.getGE_ROI(ifil,2,(100,200,5,7))
>>> imgwin = GE.PlotGEimage(ROI ,'','ROI',plotlist, (100,200,4,6))
>>> plotnotebook.ShowPlots()
```

APSpy.GE.**PlotROIsums**(*datarray*, *tablbl='ROIs'*, *title=''*, *captions=None*, *size=(700, 700)*, *imgwin=None*)

Plots a series of ROIs

Parameters

- **datarray** (*array*) – a list of MxN array of average intensity values, as returned by `sumAllGE_ROIs()`, where M is the number of frames and N is the number of ROI region(s).

- **tablbl** (*str*) – A string with the title for the new tab. (Should be short; default is "ROIs".)
- **title** (*str*) – A string with a title for the window. Defaults to blank.
- **captions** (*list*) – A list of N strings, where each string specifies a legend caption for each of the N ROI regions. (Default is "ROI #".)
- **size** (*list*) – A list, tuple or wx.size object with the size of the window to be created in pixels. The default is (700,700)
- **imgwin** (*object*) – A plotnotebook object that has been created using `plotnotebook.MakePlotWindow()`, usually in a prior call to `PlotGEimage()`. A value of None (default) causes a new frame (window) to be created.

Returns A reference to the plot window (a plotnotebook object), which will be either imgwin or the new one created in `plotnotebook.MakePlotWindow()`.

Example:

```
>>> regionlist = [ROI_rect(1335,1525,50,50),ROI_rect(1435,1525,50,50),
...               ROI_rect(335,1525,50,50),ROI_rect(1935,1525,50,50)]
>>> caps = [str(i[0])+','+str(i[1]) for i in regionlist]
>>> ROIarr = GE.sumAllGE_ROIs(imgfile,regionlist)
>>> GE.PlotROIsums(ROIarr, captions=caps)
>>> import plotnotebook
>>> plotnotebook.ShowPlots()
```

class APSpy.GE.**ROI_rect**(*xmid*, *ymid*, *xwid*, *ywid*)

Defines the rectangle of a region of interest (ROI) by midpoint and width

Each ROI region consists of 4 elements:

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

get_bounds()

return the boundaries (start and end) of an ROI

APSpy.GE.**getGE_ROI**(*filename*, *frame*, *region*)

Read a section (region of interest) of a GE image from a file. This is usually faster than reading an entire image.

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images
- **frame** (*int*) – the image number on the file, counted starting at 1
- **region** (*list*) – describes the region to be extracted

element #	label	description
0	xmid	x value for central pixel
1	ymid	y value for central pixel
2	xwid	half-width of ROI in pixels
3	ywid	half-width of ROI in pixels

The extracted ROI will be pixels img[ymid-ywid:ymid+ywid,xmid-xwid:xmid+xwid] where img is the full image.

Returns An image as a (2*ywid)x(2*xwid) numpy memmap (behaves like an array) of intensities

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.getGE_ROI(ifil,2,(100,200,4,6))
memmap([[1755, 1762, 1763, 1761, 1766, 1762, 1761, 1756],
   [1761, 1763, 1760, 1764, 1765, 1769, 1755, 1758],
   [1762, 1762, 1763, 1758, 1769, 1769, 1757, 1756],
   [1760, 1767, 1764, 1763, 1763, 1765, 1762, 1756],
   [1760, 1764, 1760, 1763, 1763, 1762, 1758, 1758],
   [1761, 1760, 1766, 1762, 1761, 1767, 1761, 1761],
   [1754, 1761, 1765, 1754, 1760, 1768, 1760, 1759],
   [1763, 1764, 1764, 1763, 1766, 1762, 1765, 1761],
   [1760, 1757, 1761, 1765, 1766, 1766, 1761, 1759],
   [1761, 1761, 1761, 1761, 1761, 1763, 1757, 1758],
   [1757, 1765, 1760, 1767, 1764, 1768, 1758, 1760],
   [1762, 1765, 1764, 1760, 1764, 1766, 1761, 1761]], dtype=uint16)
```

APSpy.GE.**getGEimage**(*filename*, *frame*)

Read a single entire GE image from a file

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images
- **frame** (*int*) – the image number on the file, counted starting at 1 An exception is raised if frame is greater than the number of frames in the file.

Returns An image as a 2048x2048 numpy array of intensities

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.getGEimage(ifil,2)
array([[1699, 1713, 1713, ..., 1701, 1697, 1695],
   [1708, 1717, 1717, ..., 1708, 1703, 1705],
   [1715, 1719, 1719, ..., 1708, 1707, 1707],
   ...,
   [1714, 1720, 1714, ..., 1698, 1702, 1697],
   [1714, 1718, 1716, ..., 1702, 1703, 1702],
   [1701, 1704, 1697, ..., 1684, 1685, 1687]], dtype=uint16)
```

APSpy.GE.**sumAllGE_ROIs**(*filename*, *regionlist*, *processes=1*)

Computes the average intensity for each ROI specified in the regionlist for every frame in a raw GE image file.

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images
- **regionlist** (*[ROI_rect]*) – A list of ROI_rect objects
- **processes** (*int*) – specifies the number of simultaneous processes that can be used to perform ROI integration using the Python *multiprocessing* module. The default, 1, will not use this module and all computations are done in the current thread. Values >1 can show significant gains in speed on multicore/multicpu computers.

Returns a list of MxN array of average intensity values, where M is the number of frames and N is the number of ROI region(s) in regionlist.

Examples:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> GE.sumAllGE_ROIs(ifil, [(100,200,4,6), (1335,1525,50,50)])
array([[ 1794.57291667,  1801.2036    ],
```

```
[ 1761.80208333,  1792.6894    ],
[ 1760.5       ,  1791.7353    ],
[ 1760.36458333,  1791.4961    ],
[ 1760.03125   ,  1791.6162    ],
...
[ 1760.0625    ,  1779.0867    ],
[ 1759.72916667,  1779.1182    ],
[ 1759.5       ,  1779.2508    ]])
```

In the example above, two ROIs are integrated for all frames in a file in the current Python interpreter.

```
>>> import GE
>>> import numpy as np
>>> import time
>>> imgfile = '/tmp/AZ91_01306'
>>> regionlist = [ROI_rect(1335,1525,50,50),ROI_rect(1435,1525,50,50),
...    ROI_rect(335,1525,50,50),ROI_rect(1935,1525,50,50)]
>>> nframe = GE.Count_Frames(imgfile)
>>> l = {}
>>> for proc in range(10):
...    st = time.time()
...    l[proc] = GE.sumAllGE_ROIs(imgfile,regionlist, proc)
...    print 'sec per frame, processors=',proc,(time.time()-st)/float(nframe)
...    assert(np.allclose(l[0],l[proc]))
```

The example above integrates 4 ROIs and compares running with all computations in the current Python thread (processes=0 and 1) with running with up to 9 concurrent processes. Usually one sees a speed-up with ~1.5 times the actual number of cores for multiprocessing. The assert is used to confirm the computation returns the same results independent of the number of processes.

APSpy.GE.**sumGE_ROIs**(*filename*, *frame*, *regionlist*)

Reads a frame from a raw GE image file and returns a list of the average intensity for each ROI specified in the regionlist.

Parameters

- **filename** (*str*) – The filename containing as-recorded GE images
- **frame** (*int*) – the image number on the file, counted starting at 1
- **regionlist** (*[ROI_rect]*) – A list of ROI_rect objects

Returns a list of N average intensity values, one for each ROI region in regionlist.

Example:

```
>>> ifil = '/Users/toby/software/work/1ID/data/AZ91_01306.ge2'
>>> regionlist = [ROI_rect(1335,1525,50,50),ROI_rect(1435,1525,50,50),]
>>> GE.sumGE_ROIs(ifil,2,regionlist)
[1792.6894, 1780.4342]
```

APSpy.GE.**sumGE_ROIs_wrapper**(*args*)

Provides an interface to `sumGE_ROIs()` that allows it to be called with a single argument. This is needed for use with the multiprocessing module and `sumAllGE_ROIs()`

Parameters **args** (*tuple*) – A tuple or list containing a filename, frame, and regionlist, as defined in `sumGE_ROIs()`.

Returns a list of N average intensity values, one for each ROI region in regionlist.

1.5.5 *Module CrossReference: simple support for cross-references*

simple support for cross-references

class APSpy.cross_ref.**CrossReference**(*name='CrossReferenceList'*)
maintain a dictionary of cross-references

add(*key*, *value*)
add key = value to the cross-reference dictionary

get(*key*)
return the value of a specific key

get_keys()
return the list of keys

get_keys_dict()
return the xref dict

get_keys_enum()
return a typesafe enum of the keys and values

This is equivalent to defining a class named with the value of self.name.

See http://docs.python.org/2/library/functions.html#type

get_values()
return the list of values

1.5.6 *Module motor: (internal) motor support classes*

motor support classes

class APSpy.motor.**MotorObject**(*symbol*, *mtrpv*, *info*, *tolerance*)
internal data structure with configuration of an EPICS motor

get_dict()
Return a dictionary with motor information.

1.5.7 *Module rst_table: format a table in restructured text (reST) format*

Nicely format a table, output in restructured text (reST) format

class APSpy.rst_table.**Table**
Construct a table in reST (using no row or column spans)

Each cell may have multiple lines, separated by a newline (n) character.

For example, this code:

```
from rst_table import Table
t = Table()
t.labels = ('Name', 'Type', 'Units', 'Description', )
t.rows.append( ['uno', 'dos', 'tres', 'quatro', ] )
t.rows.append( ['class', 'NX_FLOAT', '..', '..', ] )
t.rows.append( ['1', '2', '3', '4', ] )
print t.reST()
```

generates this output:

```
1  ===== ======== ===== ===========
2  Name  Type     Units Description
3  ===== ======== ===== ===========
4  uno   dos      tres  quatro
5  class NX_FLOAT ..    ..
6  1     2        3     4
7  ===== ======== ===== ===========
```

which looks like this when formatted:

Name	Type	Units	Description
uno	dos	tres	quatro
class	NX_FLOAT		
1	2	3	4

class methods

complex_table(*indentation=''*, *add_tabularcolumns=True*)
return the table in *complex* rest format

Name and Attributes	Type	Units	Description (and Occurrences)
one, two	buckle my	shoe. three, four	...
uno	dos	tres	quatro
class	NX_FLOAT		
1	2	3	4

find_widths()
measure the maximum width of each column, considering possible line breaks in each cell

list_table(*indentation=''*, *add_tabularcolumns=True*, *title='Table'*)
return the table in *list-table* rest format (experimental)

from: http://docutils.sourceforge.net/docs/ref/rst/directives.html

Table 1.1: Table

Name and Attributes	Type	Units	Description (and Occurrences)
one, two	buckle my	shoe. three, four	...
uno	dos	tres	quatro
class	NX_FLOAT		
1	2	3	4

reST(*indentation=''*, *format='simple'*, ***kws*)
return the table in reST format

format may be one of these: *simple* (default), *complex*, or *list*

simple_table(*indentation=''*, *add_tabularcolumns=True*)
return the table in *simple* rest format

Name and Attributes	Type	Units	Description (and Occurrences)
one,	buckle my	shoe.	...
two			
		three, four	
uno	dos	tres	quatro
class	NX_FLOAT		
1	2	3	4

This documentation was compiled June 21, 2013.

1.6 About APSpy

1.6.1 License

This software is licensed using the Argonne OPEN SOURCE LICENSE, see `LICENSE` file for details

1.6.2 Change History

This describes user-visible changes between the versions. Refer to the TRAC site for details on the tickets noting *APSpy* in the title. (https://subversion.xray.aps.anl.gov/trac/bcdaext/report/1?sort=ticket&asc=1)

Version 1.2 (not released)

- -tba-

Version 1.1.7Nov (2012-11-07 as SPECpy)

- add simulation mode
- Add new area detector module
- add pseudo motors and simultaneous start for motor moves
- milestone: https://subversion.xray.aps.anl.gov/trac/spec1ID/milestone/Enhancement%3A%20Misc%20specpy%20capabiliies

Version 1.0 (2012-08-24 as SPECpy)

- upload initial version of 8/15 (not fully working and needs changes to work with the coming pyepics-3.2.1)

This documentation was assembled June 21, 2013.

Warning: This code **must** be modified in order to use it at other locations.
This code was developed to support operations at station 1-ID of the Advanced Photon Source.
At this time, configuration details for 1-ID are woven throughout the source code.

This document is also available in a PDF.

PYTHON MODULE INDEX

a

INDEX

I

L

M

O

P

R

S

T

U

W