API Documentation

API Documentation

July 20, 2006

Contents

1 Module DataDict

DataDict.py

The DataDict overloads the regular dictionary (but does not inherit from it) by allowing a user to call a special function with special arguments whenever any of its values change.

This code was based from the UserDict code. It could have inherited from UserDict; perhaps though it is easier to understand with it all right here.

NOTE: The latest Python versions allow derivation from built-in types. UserDict and DataDict are not needed anymore, and a rewrite should occur later on at some point.

The DataDict also allows attribute access to the dictionary.

Usage: Instantiating an instance of the class:

```
>>> mydict = DataDict()
```

Adding an item, simplest form:

```
>>> mydict.additem("simple")
>>> mydict["simple"] # default value is None
>>> print mydict["simple"]
None
```

```
>>> mydict["simple"] = 42
>>> mydict["simple"]
42
>>> mydict.simple
42
```

Reading an item that does not exist prints a message.

```
>>> mydict["notthere"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "DataDict.py", line 156, in __getitem__
    raise KeyError
KeyError: 'Key name notthere does not exist.'
>>> mydict.additem("withval",42)
>>> mydict["withval"]
42
```

What happens when you write an item that does not exist? Unlike ordinary dictionaries, we want that to complain.

```
>>> mydict["notthere"] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "DataDict.py", line 160, in __setitem__
    raise KeyError
KeyError: 'Key name notthere does not exist.'
```

Writing an unknown attribute should complain too; but it doesn't.

```
>>> mydict.notthere = 42
>>>
```

The action functions setfuncs and getfuncs hook assignment and retrieval into executable code. Lambda functions are used in this example and are handy for simple setfuncs and getfuncs.

```
>>> mydict.additem("withget",42,getfunc = lambda : "spam")
>>> mydict.withget
'spam'
>>> mydict.withget = "eggs" # try to write it, it seems to fail
>>> mydict["withget"] # a read returns value from function
'spam'
>>> mydict.data["withget"] # internally, it still knows last written value.
'eggs'
```

You wont use a write-only variable like this very often, but several dictionary items can share the same setter function using arguments.

```
>>> def setter(value, arg) : # setter function, two args.
...     print value, arg
...
>>> mydict.additem("withset",42,setfunc = setter, args = ("cheese",))
>>> mydict.additem("withset2",42,setfunc = setter, args = ("grail",))
>>> mydict["withset"]  = "baloney"
baloney cheese
>>> mydict.withset2  = "baloney"
baloney grail
```

A GUI hook for the whole dictionary can be set as well.

```
>>> mydict.set_widget = setter # we'll reuse setter for testing.
```

Now accesses to any item calls the set_widget function. currently, "_txt" is appended to each name. This reflects a pyshowall internal, and should probably be removed. The pyshowall set_widget function hook would be more appropriate for adding the "_txt"

```
>>> mydict["simple"] = "parrot"
simple_txt parrot
```

A namemap dictionary can alias a dict name to a widget name:

```
>>> mydict.namemap = {"simple":"complicated"}
>>> mydict["simple"] = "bridgekeeper"
complicated_txt bridgekeeper
```

1.1 Variables

Name	Description
__version__	**Value:** `'\n     $Id: DataDict.py 399 2006-06-04 20:02:1-7Z drew $\n'`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/DataDict.py $'`

1.2 Class DataDict

```
object ─┐
        │
        DataDict.DataDict
```

Known Subclasses: det.DetDict, run.RunDict

Smart dictionary. Very similiar to Python properties.

1.2.1 Methods

__cmp__(*self*, *dict*)

__contains__(*self*, *key*)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__delitem__(*self*, *key*)

__getattr__(*self*, *name*)

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__getitem__(*self*, *name*)

Recall a value from the smart dictionary.
If the name has a getfunc, that function is called (with arguments if arguments are defined for the name; the arguments are the same for setfunc and getfunc) The return value of getfunc is returned to the user.
If no getfunc exists, the internal value associated with that name is returned.

__hash__(*x*)

hash(x)

__init__(*self*, *dict*=`None`)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__iter__(*self*)

__len__(*self*)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*self*)
repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__setattr__(*self*, *name*, *value*)
x.__setattr__('name', value) <==> x.name = value

Overrides: object.__setattr__ extit(inherited documentation)

__setitem__(*self*, *name*, *val*)

Assign a value to the smart dictionary.
If a setfunc exists, that is called first (with args if they exist). Successful setfunc execution is followed by remembering the value that was written, then by an attempt to update a GUI via set_widget (the default set_widget function does nothing.) The namemap can map dictionary names to widget names if they are different; exceptions in namemap or set_widget are ignored.

__str__(*x*)

str(x)

additem(*self*, *name*, *val*=`None`, *setfunc*=`None`, *getfunc*=`None`, *args*=`()`, *kwds*=`None`, *docstring*=`None`)

Put a new entry in the smart dictionary.
name = string used to identify this thing val = initial value of item setfunc: callable. If defined, it is called with new value on writes getfunc: callable. If defined, it is called on reads args: if defined, passed as last argument for setfunc and getfunc. docstring: help string for the item.

clear(*self*)

copy(*self*)

get(*self*, *key*, *failobj*=`None`)

has_key(*self*, *key*)

items(*self*)

iteritems(*self*)

iterkeys(*self*)

itervalues(*self*)

keys(*self*)

popitem(*self*)

set_widget(*self*, **args*)

setdefault(*self*, *key*, *failobj*=`None`)

update(*self*, *dict*)

values(*self*)

1.2.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
namemap	**Value:** {}

2 Module autouser

autouser.py - automatic user object.

Simulates a user for automatically testing code that requires user input. It is also useful for scripting that same code for real operation. Check the source code in tests.py for an example.

AutoUser is a class that simulates a user at the keyboard.

Quick Start: To use in pydsp, break to the command prompt, then do:

```
>>> from autouser import AutoUser
>>> my_user = AutoUser("itime 4000","sscan","srun")
>>> cloop(my_user)
```

You can only use the autouser object once! It depletes its list of commands as it runs.

To write code that can use autouser objects just make sure the code that requires input is defined with raw_input as a keyword parameter that is passed in:

```
>>> def myfunc( self, raw_input=raw_input) :
>>> X = raw_input("are you sure?")
```

then in your test code use a simulated user!

Sample operation:

```
>>> autouser = AutoUser("Y","N","x")
>>> raw_input
<built-in function raw_input>
>>> raw_input = autouser
>>> answer = raw_input("format hard drive?")
format hard drive? Y
>>> answer
'Y'
>>> answer = raw_input("are you sure?")
are you sure? N
>>> answer
'N'
>>> answer = raw_input("hit x to continue")
hit x to continue x
>>> answer
'x'
>>> answer = raw_input("Raise an error at end of list")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "utils.py", line 14, in next
    raise StopIteration
StopIteration
>>> del raw_input
```

2.1 Variables

Name	Description
__version__	**Value:** `'$Id: autouser.py 401 2006-07-11 22:31:58Z drew-$ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/autouser.py $'`

2.2 Class AutoUser

Create a virtual user, executing a list of commands automatically (like a macro)

2.2.1 Methods

__init__(*self*, **responses*)

__call__(*self*, *prompt*=`''`)

3 Module det

det.py det.fth equivalent module..

The det module supports detector configuration, with persistence. This is the low level "static" configuration details. The names of the bias voltages and how to nominally set them (which dacs) The max number of rows and columns etc.

It is responsible for the detector files. it needs to know the detector name and the path to detector file directory

In that directory: 'detname'.map is maps signal names to dac numbers. 'detname'.bias maps names to voltages. (Actually, it does clock rails .) plain old 'detname' is the file that initializes the other stuff.

the 'run' module does the same thing at a higher level det stuff does not normally get persisted. run stuff does... Things that are to be restored in the next session are saved in 'lastrun.run'

The file used by the old system (det.fth) was order dependent. each line had a specific meaning. extra or missing lines are bad news.

With this package, the file is more tolerant, and line items are key - value pairs.

notes: the device driver itself might eventually need to be insmod'ed from here! (it is the equivalent of the data program.. the driver might be enhanced/overhauled over time)

What is a bias voltage, at this level? well, it is tuple.. of.. a name, a dacnumber, a conversion from dac counts to volts, a sanity check for values.. and something else?

3.1 Functions

setclockpgm(*clockprogname*, *raw_input*=`<built-in function raw_input>`, *wfile*=`sys.stdout`)

Resets the DSP and reboots it with new clock program.
It tries 5 times after which it allows the program to continue even if boot fails.

init()

Initialize the detector smart dictionary. Normally called only once.

writeclockdac(*val*, **dacs*)

Writes val (in millivolts) to clock DACs. DACs is a tuple list of clock DAC numbers.
Checks that the voltage is ok. then converts the voltage into DAC counts and writes that out.

writebiasdacfunc(**args*, ***kwds*)

Create (curry) a function to write the bias DAC specified with the proper gain and offset.

setbiases(*biasrunfile*, *wfile*=`sys.stdout`, *zero*=`False`)

Loads biasrunfile and sets bias voltages accordingly.
Sets up the bias rails in the order that it finds them in the file. if zero==True, zero all the voltages in the reverse order of the file.

showbiases(*wfile*=`sys.stdout`)

List the bias names and their corresponding voltages

loadbiasmap(*biasmapfile*, *wfile*=`sys.stdout`)

Load a biasmapfile, which maps bias names to dac numbers.
each bias (programmable voltage) has: 1: a name we refer to it with. (the key) 2: a tuple of dacs that it is associated with (args for the set function) 3: a function that it calls to change the voltage. 4: the current value that it is set to.
a pseudo-bias (which may move around several biases in a coordinated manner) may be possible using this same thing..

get_detname()

Return the current detector name

savedet(*wfile*=`sys.stdout`)

Save the detector information in the "detfile."
The current det dictionary's detname is used for the name of the file. If a file of this name on the proper path can be opened for writing, the file is written using the current entries in the detector dictionary.

loaddet(*detfilename*, *wfile*=`sys.stdout`)

Load the detector configuration from the specified detfile.
First, it loads the biasmap. then it opens and reads the detfile itself, assigning the values into the det dictionary, which is 'smart' and runs code on some assignments.
The first item in the detfile (after detname) is the clock program name. assigning the clock program name into the dictionary actually resets the dsp and loads the named clock program It silently ignores keys in the detfile that it does not recognize.
After reading the detfile, loaddet sets the bias voltages.

powerup(*wfile*=`sys.stdout`)

Turn on all of the biases and clocks. use the order specified in detname.bias file

powerdown(*wfile*=`sys.stdout`)

Turn off all of the biases and clocks. use the order specified in detname.bias file, in reverse.

3.2 Variables

Name	Description
__version__	**Value:** `'$Id: det.py 400 2006-06-19 22:39:30Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/det.py $'`
dd	**Value:** `{'vbias': 0, 'vreset': 0, 'mybias': 0, 'voffset-': 0}`

continued on next page

Name	Description
biaslist	**Value:** `[]`
dac_mv_per_count	**Value:** `{0: 1.0, 1: 1.0, 12: 1.0, 13: 1.0, 26: 1.0, 27:-1.0}`
dac_mv_offset	**Value:** `{0: -1.0, 1: -1.0, 12: 2.0, 13: 2.0}`
maxdac	**Value:** `4500`
mindac	**Value:** `-8000`
dacfuncs	**Value:** `{'BIAS': <function writebiasdacfunc at 0xf6e594-c4>, 'CLOCK': <function writec...`
dacfuncnames	**Value:** `['BIAS', 'CLOCK']`

3.3 Class DetDict

```
object ─┐
        │
DataDict.DataDict ─┐
                   │
                   det.DetDict
```

Smart dictionary that contains the detector basic parameters.

Getting and setting values in this dictionary may actually read or write the physical hardware. Class is do-nothing, but properties can be added to the class on-the-fly outside of the class statement.

3.3.1 Methods

__cmp__(*self*, *dict*)

__contains__(*self*, *key*)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__delitem__(*self*, *key*)

__getattr__(*self*, *name*)

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__getitem__(*self*, *name*)

Recall a value from the smart dictionary.
If the name has a getfunc, that function is called (with arguments if arguments are defined for the name; the arguments are the same for setfunc and getfunc) The return value of getfunc is returned to the user.
If no getfunc exists, the internal value associated with that name is returned.

__hash__(*x*)

hash(x)

__init__(*self*, *dict*=`None`)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__iter__(*self*)

__len__(*self*)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*self*)

repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__setattr__(*self*, *name*, *value*)

x.__setattr__('name', value) <==> x.name = value

Overrides: object.__setattr__ extit(inherited documentation)

__setitem__(*self*, *name*, *val*)

Assign a value to the smart dictionary.
If a setfunc exists, that is called first (with args if they exist). Successful setfunc execution is followed by remembering the value that was written, then by an attempt to update a GUI via set_widget (the default set_widget function does nothing.) The namemap can map dictionary names to widget names if they are different; exceptions in namemap or set_widget are ignored.

__str__(*x*)

str(x)

additem(*self*, *name*, *val*=`None`, *setfunc*=`None`, *getfunc*=`None`, *args*=`()`, *kwds*=`None`, *docstring*=`None`)

Put a new entry in the smart dictionary.
name = string used to identify this thing val = initial value of item setfunc: callable. If defined, it is called with new value on writes getfunc: callable. If defined, it is called on reads args: if defined, passed as last argument for setfunc and getfunc. docstring: help string for the item.

clear(*self*)

copy(*self*)

get(*self*, *key*, *failobj*=`None`)

has_key(*self*, *key*)

items(*self*)

iteritems(*self*)

iterkeys(*self*)

itervalues(*self*)

keys(*self*)

popitem(*self*)

set_widget(*self*, **args*)

setdefault(*self*, *key*, *failobj*=`None`)

update(*self*, *dict*)

values(*self*)

3.3.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
namemap	**Value:** {}

4 Module detectors.default

default customization file

5 Module detectors.sipin226

customization file for silicon PIN array

5.1 Functions

get_vbias(**args*)

return the current bias voltage setting

set_vbias(*value*)

set the bias voltage by changing dsub

get_vreset(**args*)

return the current bias voltage setting

5.2 Variables

Name	Description
board	**Value:** `[<detectors.sipin226.Bunch object at 0xf6ba0c8c->]`
write_pin14	**Value:** `<function writebiasdac at 0xf6b905dc>`
set_vreset	**Value:** `<function set_vreset at 0xf6b9064c>`

5.3 Class Bunch

```
object ─┐
        └ detectors.sipin226.Bunch
```

5.3.1 Methods

__init__(*self*, ***kwds*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

5.3.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6 Module dsp

dsp.py Encapulates command level access to the running 56303 dsp program.

This module deals with the command table that appears in the clock program. Challenge faced here: different dsp programs need to be talked to in different ways!

It depends upon the ociw module, which handles the physical link (booting the dsp, serial communication, etc) ociw does not know or care what is inside the clock program. (well, should not anyway.) This dependency is somewhat hardcoded.. although that is kinda hard to do in Python. ;-)

Could we make this communication a MONITOR? We could just boot the monitor section, and replace clocking programs without replacing the monitor!

Module Todo: Check to see if clock program has been compiled. if not, do that from here. May want to break the command #defines out into a separate file, and read the same file into both C and Python. (Once and only once.) (Apply the "command" pattern here? Instead of writing this code, use template of command various templates could be defined in the dsp's header file. Populate command code in python dynamically by reading the C header file.) Check to see if the compiled clock program is already running on the dsp. If so, don't re-download.. just re-sync?? This module is a candidate for a separate thread.

6.1 Functions

dummymode()

Kick dsp into an offline mode for system use if no hardware available.

dummymodeOff(*device*)

read(*dataexpected*=`1`)

wrapper for read

readAdcs()

issue the read adc command

writeSeqBits(*bitpat*)

Write the 20 bit sequence register with a specific pattern.

setSeqBits(*bitpat*=`0x0FFFFF`)

set only the bits in bitpat. leave the others unchanged.

clearSeqBits(*bitpat*=`0x0FFFFF`)

clear only the bits that are SET in bitpat. leave the others unchanged.

testClockRails()

testBiases()

startingAlert()

Calls observers who want to know that clocking has started.

stoppingAlert()

Calls observers who want to know that clocking has stopped.

singlepix()

Tell clock program to do single pixel (no col/row clocks) multiple read

tweak_heater(*heater*)

Single word command to bump heater up or down a little bit. current heater voltage must be near the new voltage. +/- 100 dac counts.

set_heater_by_tweak(*desired_mv*)

adc2diode(*n_adc*)

Convert the hi res temp dac reading to a diode voltage

dspthreadable(*func*)

helper function to make a function threadable.
This can make functions threadable without having the dsp thread alive.

load_srec(**args*, ***kwds*)

Reset the target, download an srecord file, and execute it.

senddsp(**args*, ***kwds*)

Send command (cnum) and value (val), to the clocking program.

writedac(**args*, ***kwds*)

Write a voltage in daccounts off of zero to a clock DAC (0-31)
clips dac write if past rails

clockit(**args*, ***kwds*)

Acquire an image into the driver buffer.
first, tell device driver to get ready. then, tell clock program to go. then, sit and wait for the dsp (all the while reading and displaying pixels to go.) separate thread would be nice.

rnorm(**args*, ***kwds*)

Set reset to normal, use ron to set always-on reset

ron(**args*, ***kwds*)

Set reset to alwayson, use rnorm to revert to normal reset

rrow(**args*, ***kwds*)

Set to reset row mode

rglobal(**args*, ***kwds*)

Set to reset global mode

rampnext(**args*, ***kwds*)

Get the itime between each sample for the next image

getFrameTime(**args*, ***kwds*)

Return actual time of last frame. Requires clock code to cooperate.

getPedTime(**args*, ***kwds*)

Return actual time of nsamp pedestal frames.

timer(**args*, ***kwds*)

Return the current value of the dsp timer. no error checking.

read7888(**args*, ***kwds*)

Read and return last 7888 ADC conversion and setup next conversion

readBiasCurrent(**args*, ***kwds*)

read the bias current in microamps

readBiasVoltage(**args*, ***kwds*)

Read the bias voltage in volts

readClockCurrent(**args*, ***kwds*)

read the clock voltage in volts

readClockVoltage(**args*, ***kwds*)

read the clock current in microamps

vDiode(**args*, ***kwds*)

Read the diode voltage. currently just with the hi-res temp input

sciStepN(**args*, ***kwds*)

Send numsteps to the IM483 stepper motor controller connected to the SCI port on the DSP.

writebias(**args*, ***kwds*)

write a voltage in dac counts off of zero to a bias DAC (0-15)

seeclockpin(**args*, ***kwds*)

Show clock pin on monitor.
Outputs value of specified clock pin to black box output.

seebiaspin(**args*, ***kwds*)

Show bias pin on monitor.
Outputs value of specified clock bias to black box output.

6.2 Variables

Name	Description
__version__	**Value:** `'$Id: dsp.py 400 2006-06-19 22:39:30Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/dsp.py $'`
starttime	**Value:** `1153427614.751621`
clocks	**Value:** `[<dsp.ClockLine object at 0xf6e7906c>, <dsp.Cl-ockLine object at 0xf6e790ec>, ...`
clockpin2headcode	**Value:** `{1: 8, 2: 10, 3: 12, 4: 14, 9: 0, 10: 1, 11: 2,-12: 3, 17: 4, 18: 5, 19: 6, 2...`
biases	**Value:** `[<dsp.ClockLine object at 0xf6e792cc>, <dsp.Cl-ockLine object at 0xf6e792ec>, ...`
biaspin2headcode	**Value:** `{1: 0, 2: 1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 7-, 9: 8, 10: 9, 11: 10, 12: 11,...`
startfuncs	**Value:** `[]`
stopfuncs	**Value:** `[]`
b	**Value:** `<dsp.ClockLine object at 0xf6e7948c>`
c	**Value:** `<dsp.ClockLine object at 0xf6e7928c>`

6.3 Class ClockLine

```
object ─┐
        │
   dsp.ClockLine
```

Simple "bunch" object, Just holds related data.

6.3.1 Methods

__init__(*self*, ***args*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.3.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.4 Class ClockLine

```
object ─┐
        dsp.ClockLine
```

Simple "bunch" object, Just holds related data.

6.4.1 Methods

__init__(*self*, ***args*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

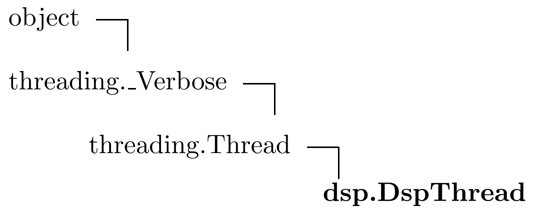
__str__(*x*)

str(x)

6.4.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.5 Class DspThread



Dsp object. singleton. runs as thread.

Inspired by page 284 of Python In a Nutshell.

6.5.1 Methods

__init__(*self*)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: threading.Thread.__init__

shutdown(*self*)

shut down in an orderly fashion.

do_nowait(*self*, *callable*, **args*, ***kwds*)

make a request on the dsp thread. when request is accepted, return a unique event that will allow caller to sleep until the task is done if request is rejected, raise StopIteration

blocking_do_nowait(*self*, *callable*, **args*, ***kwds*)

make a request on the dsp thread. if request is accepted, return a unique event that will allow caller to sleep until the task is done if request is rejected, raise StopIteration

do(*self*, *callable*, **args*, ***kwds*)

execute callable on dspthread
return its result, or raise its exception.

interrupt(*self*, *source*=`"unspecified"`)

Tell the dsp thread to halt execution.

run(*self*)

The main loop of the hardware serializer. do NOT call directly. This IS the dsp thread.

Overrides: threading.Thread.run

tc_in_acq(*self*, *current_temp*)

Temperature control during acquisition.
passes the current temp into the temp controller, adjusts the desired temperature (the carrot, ie goal we follow) determines the new heater power / voltage to send, and sends the heater command.

threadable(*self*, *func*)

helper method to make a function threadable, OOP style.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*self*)
repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

getName(*self*)

isAlive(*self*)

isDaemon(*self*)

join(*self*, *timeout*=`None`)

setDaemon(*self*, *daemonic*)

setName(*self*, *name*)

start(*self*)

6.5.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

7 Module dv

dv.py.. Socketed conection to DV program.

allows system to have DV automatically load in acquired images.

7.1 Functions

send_command(*cmd*)

Send command to dv over a socket.

send2dv(*cmd*, *wfile*=`sys.stdout`)

Send the command to dv and echo it at the terminal

view(*filename*)

View the image in file 'filename' in dv window

bmax(*bufnum*, *wfile*=`sys.stdout`)

Set the maximum buffer to be used in dv.
Can use numbers or letters. bmax [0-7] or bmax ["a-h"|"A-H"]

math(*mathstr*)

Send math string to DV. Not implemented

advance_fbuf()

Advance the "current" source buffer to the next one.

load_bkg(*bkgfilename*)

Tell dv to load the bkg file into buffer F

load_src(*srcfilename*, *wfile*=`sys.stdout`)

Tell dv to load the src file into next src buffer

src_minus_bkg()

Tell dv to subtract the background from the current source.
We pre-advance the frame buffer in pydsp, then write the file. so it is pointing to the last source buffer we wrote. (This is unlike dspsys, which wrote the buffer, then advanced.)

dv_path(*path*)

Tell dv the path it should use for data files.

dv_dofile()

Legacy. either do a macro or tell dv where macros are. ?? not sure what this does in fthsys.

active(*canvasnum*)

set the active canvas. Canvas = GUI Display, 0 - 8 in L mode last one is the big one.

buffer(*bufferletter*)

set the buffer for the active canvas. (A-H in L mode) buffer = data source. Each canvas can refer to any buffer

vfmode()

turn auto-vf mode on or off. if turning on, make sure paths are cool. (fthsys did it this way) todo: implement?

7.2 Variables

Name	Description
__version__	**Value:** `'$Id: dv.py 400 2006-06-19 22:39:30Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/dv.py $'`
hostname	**Value:** `'localhost'`
port	**Value:** `30123`
packsize	**Value:** `160`
autoupdatemode	**Value:** `0`
bgsubmode	**Value:** `0`
buf	**Value:** `['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']`

8 Module filterBase

filterBase.py Base Filter Classes. (Replaces FILTERWH.FTH)

contains both fixed and variable filters. they *could* both inherit from a common "filter" base class.. but at the current complexity level, it doesn't really seem helpful to do so.

Filters support a simple interface... You basically set the filter to some value.

CV filters are set by wavelength. fixed filters are set by filter name.

The lowest level of this ultimately moves the wheel to some angle.. This module has a dummy move method that gets overridded with the real move method. A single wheel is assumed in the current implementation.

8.1 Functions

move(*degrees*)

8.2 Variables

Name	Description
__version__	**Value:** `'$Id: filterBase.py 399 2006-06-04 20:02:17Z drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp/trunk/pydsp/filterBase.py $'`
__author__	**Value:** `'$Author: drew $'`

8.3 Class cvFilter

Continuously variable filter base class.

8.3.1 Methods

__init__(*self*, ***kw*)

set(*self*, *nm*)

Set the filter to the requested wavelength.

moveto(*self*, *degrees*)

8.4 Class fixedFilter

Fixed filter base class.

fixed filters have named filters that know their positions. a move method is mixed in, so they can accept move requests.

8.4.1 Methods

__init__(*self*, ***kw*)

set(*self*, *name*)

set filter to named position by finding the corresponding angle and moving there.

moveto(*self*, *degrees*)

9 Module filterII

filterII.py Filter Routines. (Replaces FILTERWH.FTH) concrete implementation of abstract filter. This is style II.

9.1 Functions

get_wavelength()

get_bandwidth()

get_filtername()

set(*value*)

9.2 Variables

Name	Description
__version__	**Value:** `'$Id: filterII.py 247 2004-10-23 18:52:25Z dsp -$ '`
__author__	**Value:** `'$Author: dsp $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/filterII.py $'`
dialzero	**Value:** `27.75`
cdszero	**Value:** `960`
cvfII	**Value:** `<filterBase.cvFilter instance at 0xf6b7c08c>`
cvfIII	**Value:** `<filterBase.cvFilter instance at 0xf6b7c06c>`
cvfs	**Value:** `[<filterBase.cvFilter instance at 0xf6b7c08c>,-<filterBase.cvFilter instance ...`
fix	**Value:** `<filterBase.fixedFilter instance at 0xf6b7ebec->`
filtername	**Value:** `None`
bandwidth	**Value:** `None`
wavelength	**Value:** `None`
__test__	**Value:** `{'usage': '\n>>> fix.set("cds")\nmoving wheel -to 335.0 degrees\n>>> fix.set("...`

10 Module filterwh

filterII.py Filter Routines. (Replaces FILTERWH.FTH) concrete implementation of abstract filter. This is style II.

10.1 Functions

get_wavelength()

get_bandwidth()

get_filtername()

set(*value*)

10.2 Variables

Name	Description
__version__	**Value:** `'$Id: filterII.py 247 2004-10-23 18:52:25Z dsp -$ '`
__author__	**Value:** `'$Author: dsp $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/filterII.py $'`
dialzero	**Value:** `27.75`
cdszero	**Value:** `960`
cvfII	**Value:** `<filterBase.cvFilter instance at 0xf6e2a40c>`
cvfIII	**Value:** `<filterBase.cvFilter instance at 0xf6e25fac>`
cvfs	**Value:** `[<filterBase.cvFilter instance at 0xf6e2a40c>,-<filterBase.cvFilter instance ...`
fix	**Value:** `<filterBase.fixedFilter instance at 0xf6e22dec->`
filtername	**Value:** `None`
bandwidth	**Value:** `None`
wavelength	**Value:** `None`
__test__	**Value:** `{'usage': '\n>>> fix.set("cds")\nmoving wheel -to 335.0 degrees\n>>> fix.set("...`

11 Module fits

fits.py.. wrapper for pyfits, puts good header stuff in fits files.

11.1 Functions

interleaved_descrambler(*hdu*)

For interleaved or single output, there is no need to descramble

block_descrambler(*hdu*)

descramble any number of outputs into block stripe formatted array.

quadrant_descrambler(*hdu*)

descramble a four output array in quadrant format.

fitsheader(*header*)

Update the header with the current information from the system.

write_image(*filename*, *obuf*)

update a new buffer with the proper fits stuff and write to disk.
This is a mix of two responsibilities. getting the proper time and temp data is one thing.. sending the image to the disk is another. they 'should' be separated. Not a biggie though.

11.2 Variables

Name	Description
__version__	**Value:** `'$Id: fits.py 400 2006-06-19 22:39:30Z drew $ '`
__revision__	**Value:** `'$Id: fits.py 400 2006-06-19 22:39:30Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/fits.py $'`
descramble	**Value:** `{'quadrant': <function quadrant_descrambler at -0xf6de8454>, None: <function i...`

11.3 Class fitsbuffer

11.3.1 Methods

__init__(*self*, *buf*=`None`)

set_size(*self*, *height*, *width*)

12 Module ociw

ociw.py: lowest-level interface to the device driver and the running DSP.

no real dependencies, but some of it assumes a certain data protocol supported by the running dsp program.

It also assumes a single PCI card. A more generic approach would create an object for each card. The driver would need a revisit in this situation.

this module, and sload.py, replaced the ociw.so Python extension module that was written in C.

12.1 Functions

open(*devname*=`"/dev/ociw0"`)

Open the device. (perhaps a bad name, Python has a built-in named open.)

write(*i*)

Write one 16 bit word to the device.

read(*expected*=`True`)

Read one 16 bit word from the device.

reset()

Reset the device.

command(*cmd*, *addr*)

Write 8 bit command and 24 bit address to the clock program. wasteful for commands with only 8 bits of data.

data24(*data*)

Write a 24 bit data word to the clock program.

clear_fifo()

make sure nothing is in the fifo

peek_fifo()

see how much is in the fifo, without reading it.

interrupted()

RawFrame objects call interrupted in all data acquisition loops to see if an interrupt has occurred. This is how acquisition is aborted.
The default implementation does nothing.
Override this function with something that actually works.

12.2 Variables

Name	Description
__revision__	**Value:** `'$Id:$'`
fd	**Value:** `None`
dev	**Value:** `None`
SET_DATA_MS	**Value:** `1`
SET_DATA_NS	**Value:** `2`
SET_DATA_LS	**Value:** `3`
SET_ADDR_MS	**Value:** `4`
SET_ADDR_NS	**Value:** `5`
SET_ADDR_LS	**Value:** `6`

12.3 Class AbstractFrame

unused design concept. left here as food for thought.

12.3.1 Methods

__init__(*self*, ***kwds*)

grab(*self*)

12.4 Class RawFrame

Image data acquisition object. Acquires a single frame. Instantiate it with nrows, ncols, prepix, and postpix. possibly add special funcs to the base class. have the object acquire by calling its grab function. it returns the image data in a string. tag it with metadata. The public interface is the grab method. Interested objects can subscribe to events before and after the data has come in.

12.4.1 Methods

__init__(*self*, *nrows*, *ncols*, *prepix*=`0`, *postpix*=`0`)

create a new acquisition object. each frame has a size expressed in rows and columns, and a few pixels that always precede or follow the main frame.
Each instance makes a copy of the base class' list of subscribers and allows the user to add new functions to this list or perhaps delete the ones that are there.

preframeAlert(*self*)

Publish-subscribe hook. Called BEFORE frame is started.

frameAlert(*self*)

Publish-subscribe hook. Called AFTER frame is successfully in.

grab_predata(*self*)

Acquire the preceding pixels for the image frame.

grab_data(*self*)

Acquire the main data for the frame.

grab_postdata(*self*)

Acquire the trailing pixels for the image frame.

grab(*self*)

Acquire the complete frame. Most users should just call this directly.

12.4.2 Class Variables

Name	Description
prefuncs	**Value:** `[]`
funcs	**Value:** `[]`

13 Module pydsp

pydsp.py – the almost top level file.

Usage was originally: python -i pydsp.py Now, start_pydsp.py is the top level. Usage is effectively: python -i start_pydsp.py

Importing this file again when it was the top level caused undesired execution of stuff (first import.) Plus, it seemed an architectural error to re-import the top file.

Alias startup to 'pydsp' by inserting the line: alias pydsp='python -tt -i $DSPHOME/pydsp/start_pydsp.py' in ~/.bashrc with this alias, just type 'pydsp' at the command line.

it emulates dspsys.fth in the dspsys forth-based system

TODO: It has a mishmash of things and could use a refactor. Its primary focus should be defining the namespace that the user sees. Sections of that namespace might be sensibly broken out to separate files. The help functions need access to that namespace, so they wound up here.

13.1 Functions

help(*thing*=`None`, *raw_input*=`<built-in function raw_input>`, *wfile*=`sys.stdout`)

Wrapper for python's pydoc help function.
It checks input; if it is a string, check all the likely places for a match: the run dict, the det dict, and the globals. If it is not a string, just pass it to pydoc.

see(*thing*, *wfile*=`sys.stdout`)

See the source code associated with an item, if possible.
It accepts strings or python objects. 'see' checks a string argument against the run and det dictionaries, then against the global namespace. If a match is found, it converts to a python object if possible and subsequently tries to get the source code for python objects.

words(*filt*=`None`, *wfile*=`sys.stdout`)

Prints three columns of words: funcs, run dict, det dict

rdclockcurrent(*wfile*=`sys.stdout`)

Internal function, do not use.

rdclockrail(*wfile*=`sys.stdout`)

Internal function, do not use.

rdbiascurrent(*wfile*=`sys.stdout`)

Internal function, do not use.

rdbiasvoltage(*wfile*=`sys.stdout`)

Internal function, do not use.

rbv(*wfile*=`sys.stdout`)

loaduser(*name*=`""`, *wfile*=`sys.stdout`)

Not implemented. To load in a user program use execuser instead.

userload(*name*=`""`, *wfile*=`sys.stdout`)

Not implemented. To load in a user program use execuser instead.

execuser(*filename*=`None`, *raw_input*=`<built-in function raw_input>`, *wfile*=`sys.stdout`)

Run a user program from the acq directory.
User program can be a straight python script, or file that defines a bunch of new words.
Any new words are added to the globals dictionary.

runuser(*wfile*=`sys.stdout`)

Not implemented. To load in a user program use execuser instead.

tcstart()

Activate temp control. Use 'tcgoto' to set the desired temp.
NOTE: Hardware temperature control not yet implemented.

tcstop()

Stop temp control.
NOTE: Hardware temperature control not yet implemented.

tcgoto(*temp*)

Set a new goal temperature.
NOTE: Hardware temperature control not yet implemented.

tcgoal()

Print the current temperature goal. (Final temp)
NOTE: Hardware temperature control not yet implemented.

tccarrot()

Print the current instantaneous temperature carrot.
NOTE: Hardware temperature control not yet implemented.

tcwait()

Wait for the desired temperature (carrot) to reach the goal temperature.
NOTE: Hardware temperature control not yet implemented.

temp2file(*filename*=`"/scratch/data/tempmon"`)

startTempControl()

connect the automatic temperature control up to the system.

try_int(*token*)

Try to make the token into an integer
If it can be made an int, return the int. else, return the original token.

execcmd(*s*, *raw_input*=`<built-in function raw_input>`, *wfile*=`sys.stdout`)

Process s, a text command string.
It might be a smart dictionary (run or det) word, or it might be a function.
Check out ipython from scipy before rewriting this.

pyshowall_connect()

Connect pyshowall to the system.
Pyshowall attempts to be generic. It is the standard interface for dealing with camera arrays. It does not know about the electronics. The main app may have different ways of hooking into the pyshowall funcs, so in this function, we plug in the application specific hooks. The main app conforms to it more than it conforms to the main app.

cloop(*raw_input*=`<built-in function raw_input>`, *wfile*=`sys.stdout`)

The main pydsp command loop.
Very simple. Replace this with ipython someday. It gets and sets things in the run dict or the data dict. or, runs one of the commands in this file.

doCompleter()

Handles tab completion.
Hook up the readline module so that all commands are logged to a history file and are available after the system shuts down and gets restarted.

startWebServer()

Start up a simple embedded web server in its own thread.

startpydsp()

Iinitialize the system from the base defaults.
Then recover the data that is in lastrun.run then hook up a control-c interrupt handler, and then the filter wheel.

13.2 Variables

Name	Description
__revision__	**Value:** `'$Id: pydsp.py 401 2006-07-11 22:31:58Z drew $'`
__version__	**Value:** `'$Id: pydsp.py 401 2006-07-11 22:31:58Z drew $'`

continued on next page

Name	Description
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/pydsp.py $'`
tmps	**Value:** `<function tmps at 0xf6c2772c>`
pagetop	**Value:** `' \n<html>\n<head>\n<title>Pydsp on Itchy</t-itle>\n</head>\n<body>\n<h1>Pydsp...`
pageend	**Value:** `'\n</body>\n</html>\n'`
ifuncs	**Value:** `[]`

13.3 Class pydspWebServer

SocketServer.BaseRequestHandler
SocketServer.StreamRequestHandler
BaseHTTPServer.BaseHTTPRequestHandler
pydsp.pydspWebServer

The class that defines the internals for the pydsp web server.

13.3.1 Methods

do_GET(*self*)

do_POST(*self*)

log_request(*self*, **args*)
Log an accepted request.
This is called by send_reponse().

Overrides: BaseHTTPServer.BaseHTTPRequestHandler.log_request extit(inherited documentation)

log_message(*self*, **args*)
Log an accepted request.
This is called by send_reponse().

Overrides: BaseHTTPServer.BaseHTTPRequestHandler.log_message extit(inherited documentation)

do_HEAD(*self*)

__init__(*self*, *request*, *client_address*, *server*)

address_string(*self*)

Return the client address formatted for logging.
This version looks up the full hostname using gethostbyaddr(), and tries to find a name that contains at least one dot.

date_time_string(*self*)

Return the current date and time formatted for a message header.

end_headers(*self*)

Send the blank line ending the MIME headers.

finish(*self*)
Overrides: SocketServer.BaseRequestHandler.finish

handle(*self*)

Handle multiple requests if necessary.

Overrides: SocketServer.BaseRequestHandler.handle

handle_one_request(*self*)

Handle a single HTTP request.
You normally don't need to override this method; see the class __doc__ string for information on how to handle specific HTTP commands such as GET and POST.

log_date_time_string(*self*)

Return the current time formatted for logging.

log_error(*self*, **args*)

Log an error.
This is called when a request cannot be fulfilled. By default it passes the message on to log_message().
Arguments are the same as for log_message().
XXX This should go to the separate error log.

parse_request(*self*)

Parse a request (internal).
The request should be stored in self.raw_requestline; the results are in self.command, self.path, self.request_version and self.headers.
Return True for success, False for failure; on failure, an error is sent back.

send_error(*self*, *code*, *message*=`None`)

Send and log an error reply.
Arguments are the error code, and a detailed message. The detailed message defaults to the short entry matching the response code.
This sends an error response (so it must be called before any output has been generated), logs the error, and finally sends a piece of HTML explaining the error to the user.

send_header(*self*, *keyword*, *value*)

Send a MIME header.

send_response(*self*, *code*, *message*=`None`)

Send the response header and log the response code.
Also send two standard headers with the server software version and the current date.

setup(*self*)
Overrides: SocketServer.BaseRequestHandler.setup

version_string(*self*)

Return the server software version string.

13.3.2 Class Variables

Name	Description
server_version	**Value:** `'pydspHTTP/1.0'`
error_message_format	**Value:** `'<head>\n<title>Error response</title>\n</head>\n<body>\n<h1>Error response</...`
monthname	**Value:** `[None, 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', ...`
protocol_version	**Value:** `'HTTP/1.0'`
rbufsize	**Value:** `-1`
responses	**Value:** `{400: ('Bad request', 'Bad request syntax or unsupported method'), 401: ('Una...`
sys_version	**Value:** `'Python/2.3.4'`
wbufsize	**Value:** `0`
weekdayname	**Value:** `['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']`

14 Module pyshowall.pyshowall

Python version of gshowall (much much smaller!!)

uses XML output from glade-2 UI builder. Basically does two things: 1: supports a command box, in which the user can enter commands. 2: allows the user to edit some boxes.

Requires pyGTK, the python bindings to the GIMP toolkit.

pyGTK is 1.99 or 2.xx GTK is 2.0, 2.2, 2.4 ish..

we have gtk-2.0 working here.

14.1 Functions

on_focus_out_event(**args*)

activatecmd(*cmd*)

Called (by on_command_activate) when the user enters a command into the command box in pyshowall. cmd is the string that was entered.
the default is a do-nothing command. Gets overridden. pyshowall.activatecmd = xxxx

on_command_activate(**args*)

called from gtk when a command is entered in the command box.
hook from the command box widget into the system. gets the text property of the command box (the request that was typed in.) Clears the command box, and calls pyshowall.activatecmd with the request.

on_enter_notify_event(**args*)

When focus enters an edit box, show help in status bar

on_leave_notify_event(**args*)

when focus leaves an edit box, clear status bar
rewrite box, too. Rewrite only needs to be done if the box was editable, and edited.

gui_raw_input(*prompt*=`""`)

gui version of raw_input function.
Pass in the prompt. It blocks and returns a response to the caller.

abortfunc()

scanfunc()

runfunc()

showbiases()

on_Abort_clicked(**args*)

on_Scan_clicked(**args*)

on_Run_clicked(**args*)

on_Voltages_clicked(**args*)

on_activate(**args*)

generic handler for editable run and det dict widgets.
hook FROM pyshowall into the system. detects that the widget has been changed. extracts the name and value from the widget and forms a text command. Then calls command processor on this text string.

scantime()

dummy function. will be patched in when system comes up.

totaltime()

return the total frame time expected. Smart UI. so what??

mytimer()

restart_timer()

stop_timer()

init_showall()

do_update()

call gtk and process events till no more events are pending.
makes sure that anything in the gui that needs updating gets updated.

set_widget(*name*, *value*)

sets the value of a named widget.

startguithread()

create a global instance of pyshowall in a separate thread.

14.2 Variables

Name	Description
__version__	**Value:** `'$Id$ '`
__author__	**Value:** `'$Author$'`

continued on next page

Name	Description
__URL__	**Value:** `'$URL$'`
verbose	**Value:** `0`
rd	**Value:** {}
dd	**Value:** {}
starttime	**Value:** `1153427616.001482`

14.3 Class gui_writer

A file-like object that accepts and prints strings.

14.3.1 Methods

write(*self*, *printstring*)

14.4 Class GuiThread

```
object ─┐
        │
threading._Verbose ─┐
                    │
        threading.Thread ─┐
                          │
                          pyshowall.pyshowall.GuiThread
```

The gui runs in this object, which is a separate thread.

14.4.1 Methods

__init__(*self*)
Overrides: threading.Thread.__init__

shutdown(*self*)

shut down in an orderly fashion.

run(*self*)
Overrides: threading.Thread.run

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

    a new object with type S, a subtype of T

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*self*)
repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

getName(*self*)

isAlive(*self*)

isDaemon(*self*)

join(*self*, *timeout*=`None`)

setDaemon(*self*, *daemonic*)

setName(*self*, *name*)

start(*self*)

14.4.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

15 Module run

run.py. Handles the run dictionary–

This smart dictionary contains the things that are persisted and restored. when the system is shut down and restarted.

it is hard to say what is "detector" and what is "last run" some things might be shared between them. bias voltages?? clock rails? detector might have defaults

This seems to go against the single responsibility principle.

15.1 Functions

init(*wfile*=`sys.stdout`)

asksavedet(*wfile*=`sys.stdout`)

XXX check if the detector configuration has changed.

savesetup(*filename*=`"lastrun.run"`, *wfile*=`sys.stdout`)

Saves the current configuration to filename. Default is "lastrun.run."
It uses the detpath from xdir, saves the rundict stuff and also saves the detector file.
NOTE: Does NOT save bias and clock voltages, except for a few exceptions.

setup(*filename*, *wfile*=`sys.stdout`)

Recover system state from rundata file.
using detpath, load in rundata file (usually lastrun.run) loop through this file and parse lines into key/value pairs. check the run dict for the key and assign there. if the key is in 'detkeys' then it is a special det file value we save and restore as an exception.

recover(*filename*=`"lastrun.run"`)

Read in the contents of filename (the run file) and reinitialize things.
The detector filename, when restored, causes the det variables to be recovered as well.

15.2 Variables

Name	Description
__version__	**Value:** `'$Id: run.py 400 2006-06-19 22:39:30Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/run.py $'`
detkeys	**Value:** `['voffset', 'vreset']`
rd	**Value:** {}

15.3 Class RunDict

```
object ─┐
        │
DataDict.DataDict ─┐
                   │
                run.RunDict
```

15.3.1 Methods

__cmp__(*self*, *dict*)

__contains__(*self*, *key*)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__delitem__(*self*, *key*)

__getattr__(*self*, *name*)

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__getitem__(*self*, *name*)

Recall a value from the smart dictionary.
If the name has a getfunc, that function is called (with arguments if arguments are defined for the name; the arguments are the same for setfunc and getfunc) The return value of getfunc is returned to the user. If no getfunc exists, the internal value associated with that name is returned.

__hash__(*x*)

hash(x)

__init__(*self*, *dict*=`None`)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__iter__(*self*)

__len__(*self*)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*self*)
repr(x)

Overrides: object.__repr__ extit(inherited documentation)

__setattr__(*self*, *name*, *value*)
x.__setattr__('name', value) <==> x.name = value

Overrides: object.__setattr__ extit(inherited documentation)

__setitem__(*self*, *name*, *val*)

Assign a value to the smart dictionary.
If a setfunc exists, that is called first (with args if they exist). Successful setfunc execution is followed by remembering the value that was written, then by an attempt to update a GUI via set_widget (the default set_widget function does nothing.) The namemap can map dictionary names to widget names if they are different; exceptions in namemap or set_widget are ignored.

__str__(*x*)

str(x)

additem(*self*, *name*, *val*=`None`, *setfunc*=`None`, *getfunc*=`None`, *args*=`()`, *kwds*=`None`, *docstring*=`None`)

Put a new entry in the smart dictionary.
name = string used to identify this thing val = initial value of item setfunc: callable. If defined, it is called with new value on writes getfunc: callable. If defined, it is called on reads args: if defined, passed as last argument for setfunc and getfunc. docstring: help string for the item.

clear(*self*)

copy(*self*)

get(*self*, *key*, *failobj*=`None`)

has_key(*self*, *key*)

items(*self*)

iteritems(*self*)

iterkeys(*self*)

itervalues(*self*)

keys(*self*)

popitem(*self*)

set_widget(*self*, **args*)

setdefault(*self*, *key*, *failobj*=`None`)

update(*self*, *dict*)

values(*self*)

15.3.2 Class Variables

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
namemap	**Value:** {}

16 Module runrun

runrun.py, parallel file of runrun.fth.

runrun interfaces the run data of run.py with the hardware of dsp.py not to mention the fits file, DV, and showall code. it is kinda the glue code that brings things together

16.1 Functions

burst()

Turns off crun mode and returns to the normal acquisition mode.

minItime()

Computes and returns the minimum value of the itime. type "print minItime()" to see the result at the command loop.

scantime()

Compute and return the total amount of time it will take to acquire an image.

src()

Set a flag to make the current DV buffer store a source image. NOTE: This function should not be used by the user directly, use only if you know what you are doing. This function is called by the 'sscan' and 'srun' commands.

bkg()

Set a flag to make the current DV buffer store a background image. NOTE: This function should not be used by the user directly, use only if you know what you are doing. This function is called by the 'bscan' and 'brun' commands.

vdiode(*wfile*=`sys.stdout`)

Print out the current diode voltage.

preCheckItime(*raw_input*=`<built-in function raw_input>`, *wfile*=`sys.stdout`)

This calls the scantime command with current settings to see if the set itime is appropriate. If itime is too short, a warning is presented to the user and waits for 3 possible response: * A carriage return, a 'n' or 'N' in the input string which aborts back to the command loop. * A 'y' or 'Y' where the itime will be set to the minimum possible. * If the user types a number, it will attempt to use that number instead as a correction to the original input value.

calibrateItime()

A Finds the magic calibration constants.
Does a small number of frames w many columns and few rows does a small number of frames many rows and few columns does many frames with few rows and few columns uses linear algebra to figure out solution. modifies values for the rest of the run of pydsp. constants are persisted by hand editing runrun.py. adding them to the run dict is an exercise for the user. ;-)

bscan()

This acquires a fowler pedestal and signal and writes the normalized difference into the bkg buffer (in DV this is always buffer F).
NOTE: This command does not save the image to disk, a subsequent invocation of this command will overwrite the buffer. Image will be displayed in the next available DV buffer.

pedscan()

This acquires a pedestal frame writes it into a ped buffer.
NOTE: This command does not save the image to disk, a subsequent invocation of this command will overwrite the buffer. Image will be displayed in the next available DV buffer.

sigscan()

This acquires a signal frame and writes it into a sig buffer.
NOTE: This command does not save the image to disk, a subsequent invocation of this command will overwrite the buffer. Image will be displayed in the next available DV buffer.

srun()

This acquires a fowler pedestal and signal and writes the normalized difference into a src buffer.
NOTE: This command will save the image to disk using an incrementing filename scheme and displayed in the next available DV buffer.

brun()

This acquires a fowler pedestal and signal and writes the normalized difference into a bkg buffer.
NOTE: This command will save the image to disk using an incrementing filename scheme and displayed in the next available DV buffer.

pedrun()

This acquires a pedestal frame and writes it into a ped buffer.
NOTE: This command will save the image to disk using an incrementing filename scheme and displayed in the next available DV buffer.

rrun()

OBSOLETE: Do not use, will be removed in future versions.

sigrun()

This acquires a signal frame and writes into into a sig buffer.
NOTE: This command will save the image to disk using an incrementing filename scheme and displayed in the next available DV buffer.

crun(*wfile*=`sys.stdout`)

Run the array continuously. Type 'burst' to abort. This command will return immediatly and run in the background. Type 'burst' to enter 'burst' mode which ends 'crun' mode.

scandir(*names*=`["itime","nsamp","vreset","dsub","TEMP","COMMENT1",]`, *wfile*=`sys.stdout`)

Writes a scandir text log for the acquired images in a run. It defaults to cataloging the following fields: "itime","nsamp","vreset","dsub","TEMP","COMMENT1".
The list of names is a default one, and a different list could be passed in.

bgscan()

Perform a 'sscan' in a background thread. When invoked, the command will return immediately to a command prompt.

16.2 Variables

Name	Description
scan	**Value:** `<function scan at 0xf6c8c844>`
run	**Value:** `<function run at 0xf6c8c8b4>`
sscan	**Value:** `<function sscan at 0xf6c8c924>`
rscan	**Value:** `<function rscan at 0xf6c8c994>`
sutr	**Value:** `<function sutr at 0xf6c8cbc4>`
rsutr	**Value:** `<function rsutr at 0xf6c8cc34>`

17 Module sload

sload.py

handles sending S-records to the DSP bootloader.

17.1 Class sloader

load s-records down to the DSP

17.1.1 Methods

__init__(*self*)

syncup(*self*)

write a zero to the dsp, return non-zero if nothing comes back

waitsync(*self*)

wait for sync, >256 tries max

srec2ints(*self*, *srec*)

convert a single s-record to an array of 16 bit ints.

sendvals(*self*, *values*)

send a single srecord (as a list of values) to the DSP.

sendsrec(*self*, *srec*)

send a single srecord (character string)

sendsrecs(*self*, *srecs*)

send a list of srecords to the DSP

file2srecs(*self*, *filename*)

open the srecord file, read all the lines, and make a list

sendfile(*self*, *filename*)

Open a srecord file and download srecords to dsp.

load_srec(*self*, *filename*)

Reset the target, download an srecord file, and execute it.

17.1.2 Class Variables

Name	Description
X	**Value:** `1`
Y	**Value:** `2`
P	**Value:** `4`
GO	**Value:** `8`
ignore_checksums	**Value:** `0`

18 Module start_pydsp

The top level file of the pydsp system. all execution starts here. A shell-bang could have been used, but we used an alias instead.

Basically, start the DSP thread, initialize the runtime hooks, start the gui thread Start some extra stuff like runline completion, then bring the control system to life. finally, dump into the command loop.

18.1 Functions

initHardware()

Initialize the dsp device.
Open up the device driver. start the singleton dsp thread.

loadDetAndRun()

Load the det (static) and run (volatile) dictionaries.
This effectively fires up the system, for instance: setting the clock program attribute is a "smart" operation, and loads the clock program into the DSP as well.

startPyshowall()

Attempt to start the pyshowall GUI.
The gui is coupled to the det and run dictionaries. Each dictionary has a set_widget method that is called when one of the attribuetes is changed. This updates the GUI. Some widget names are different from attribute names, so a namemap translates from one to the other.

18.2 Variables

Name	Description
__version__	**Value:** `'$Id: start_pydsp.py 398 2006-06-03 22:21:52Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp/trunk/pydsp/start_pydsp.py $'`

19 Module tempcontrol

tempcontrol.py

Control temperature by varying power to a heater resistor.

19.1 Variables

Name	Description
__version__	**Value:** `'$Id: tempcontrol.py 314 2004-12-07 15:25:51Z d-sp $ '`
__author__	**Value:** `'$Author: dsp $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/tempcontrol.py $'`

19.2 Class tcontroller

Generic temperature controller class.

Ideally, the thermal modelling is accurate, and little extra is needed to close the loop.

19.2.1 Methods

__init__(*self*, *temp_reader*=`None`, *heat_writer*=`None`, *K_per_min*=`0.1`, *decay_minutes*=`55`, *equilib_K*=`29.6`, *equilib_V*=`3.0`, *gain*=`10`, *wfile*=`sys.stdout`)

create a new temp controller object.

watts(*self*, *error*=`None`)

Given all else, what power should we put out?

ac_watts(*self*)

K_per_min(*self*)

given the carrot, goal, and ramp rate, yield desired thermal velocity.

r_power(*self*, *volts*)

Return resistor power for a given voltage

dc_watts(*self*, *temp*=`None`)

watts needed to sustain temp (defaults to current carrot)

watts_per_K(*self*, *temp*=`None`)

Return slope of watts vs temp curve at the specified temp. thermal conductivity varies wildly with temperature.. also, the array may self-heat differently at different temps.

set_dc_calibration(*self*, *reading1*, *reading2*)

using two known control points, where each reading is a tuple (voltage, temp), set the slope and equilibrium points.

load_r(*self*)

Return the load resistance. Why a function? R may vary with temp. (property might be good here) the current monitor could actually help here, and measure the current, and we could compute the proper voltage to deliver that power.

volts(*self*, *watts*=`None`)

power is V squared over R. V = sqrt(power * resistance)

readTemp(*self*)

return the current temperature in degrees Kelvin. MUST OVERRIDE!

setHeater(*self*, *voltage*)

set the heater voltage in volts. MUST OVERRIDE!

adjustCarrot(*self*)

Move the carrot. carrot == desired instantaneous temperature. (picture a horse with a carrot hanging in front of it)

do(*self*)

goto(*self*, *newgoal*)

20 Module tests

The main unit test suite for pydsp.

Somewhat of a summary of the different modules as well. run at the Pydsp command, probably WITHOUT a device connected.

20.1 Functions

test_DataDict()

test_det()

test_run()

test_xdir()

test_dsp()

test_runrun()

test the functionality of the runrun module

test_dv()

test_filter()

test_scans()

Automatically test the system. The actual system.
Puts in known commands, and checks to see that proper responses are obtained. This suite can be run after code is changed to make sure that things were not broken.

test_autouser()

test_pydsp()

test_start_pydsp()

testall()

20.2 Variables

Name	Description
__version__	**Value:** `'$Id: tests.py 399 2006-06-04 20:02:17Z drew $ -'`

continued on next page

Name	Description
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/tests.py $'`

21 Module tmptr

tmptr_curve10.py TS temperatures for fanout+work surface

Reads lake shore diode voltage-temperature characteristic into dictionary (vcurv, tcurv). the temperatures are calculated via linear regression. If the voltage indicates a temperature that is either above or below the characteristic, None is returned.

21.1 Functions

tdiode(*v*, *table*=`curve10`)

Convert diode voltage to Kelvin using table (curve10 default).

21.2 Variables

Name	Description
__version__	**Value:** `'$Id: tmptr_curve10.py 400 2006-06-19 22:39:30Z - drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp- /trunk/pydsp/tmptr_curve10.py $'`
curve10	**Value:** `[(320.0, 0.47069), (310.0, 0.49484), (300.0, 0.- 51892000000000005), (290.0, 0....`

22 Module tmptr_curve10

tmptr_curve10.py TS temperatures for fanout+work surface

Reads lake shore diode voltage-temperature characteristic into dictionary (vcurv, tcurv). the temperatures are calculated via linear regression. If the voltage indicates a temperature that is either above or below the characteristic, None is returned.

22.1 Functions

tdiode(*v*, *table*=`curve10`)

Convert diode voltage to Kelvin using table (curve10 default).

22.2 Variables

Name	Description
__version__	**Value:** `'$Id: tmptr_curve10.py 400 2006-06-19 22:39:30Z - drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp- /trunk/pydsp/tmptr_curve10.py $'`
curve10	**Value:** `[(320.0, 0.47069), (310.0, 0.49484), (300.0, 0.- 51892000000000005), (290.0, 0....`

23 Module wheel

wheel.py Filter Wheel Movement Routines & Parameters Version 1.20

filter wheel movement, geg stepper motor slo-syn m061-fd-301 model, 200 steps/360deg 10:1 anti-backlash worm reduction ==> 2000 steps/360deg of fw 0-9999 10-turn fwp indicator ==> 5 fw steps/motor step 0.18 deg/motor step, 0.036 deg/fw step

IM483 stepper driver takes 4 steps at its input to make the dial indicator move one count. (this is coarsest resolution of the IM483.)

23.1 Functions

move(*motorsteps*)

default motor move method. override with a real motor move method.

backlash()

getdial()

Return the current position of the dial

presetdial(*dialpos*)

getdialsteps(*dialpos*)

Return the number of motor steps it will take to get to dialpos by shortest path.

setdial(*dialpos*)

Move the filter wheel to make the dial read dialpos. Use backlash

setdialnb(*dialpos*)

Move the filter wheel to make the dial read dialpos. No backlash

angle(*degrees*=`None`)

if degrees passed in, set filter wheel to that angle. Return position.

23.2 Variables

Name	Description
__version__	**Value:** `'$Id: wheel.py 399 2006-06-04 20:02:17Z drew $ -'`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/wheel.py $'`

continued on next page

Name	Description
curdial	**Value:** `None`
dialzero	**Value:** `0`

23.3 Class Wheel

23.3.1 Methods

__init__(*self*, *value*=`0.0`)

setdegrees(*self*, *degrees*)

getsteps(*self*)

steps_to(*self*, *other*)

24 Module xdir

xdir.py is a translation of xdir.fth.

It keeps track of the paths to various system-related files. It seems trivial enough that most of its functionality should probably go into start_pydsp.py and run.py

24.1 Functions

paths()

prints out the det, data, and clock paths.

noobject()

clear out the current object name and number

nonight()

clear out the current night name, also clear the current object

get_nightpath()

Return the full path to the current night directory, or None

get_objpath()

Return the full path to the current object directory

get_objfilename(*num*=`None`)

Return the full path to the object file number num

get_nextobjfilename()

return the name of the next file we can write to. leave objnum at the last file that was written, or zero if none have been written yet. objnum is always less than or equal to the number of objects in the directory.

setclockpath(*newclockpath*)

setdatapath(*newdatapath*)

set a new data path. if the path is a change to the current one, clear out the night and object.

setdetpath(*newdetpath*)

set_night(*newnightname*)

Create a new night diretory under the datapath we allow setting to a previously existing night.. although that may be strange outside the lab. (in the lab, a ”night” is usually a particular detector under test.

set_object(*newobjname*)

Set the name of the current object. if it does not exist, make it exist. if it does exist, cd to it and advance the number to the first unused used object number.

24.2 Variables

Name	Description
__version__	**Value:** `'$Id: xdir.py 399 2006-06-04 20:02:17Z drew $ '`
__author__	**Value:** `'$Author: drew $'`
__URL__	**Value:** `'$URL: http://astro.pas.rochester.edu/svn/pydsp-/trunk/pydsp/xdir.py $'`
datapath	**Value:** `''`
nightname	**Value:** `''`
objname	**Value:** `''`
objnum	**Value:** `0`
dsphome	**Value:** `'/home/dsp/dsp'`
detpath	**Value:** `'/home/dsp/dsp/det'`
clockpath	**Value:** `'/home/dsp/dsp/56300'`
driverpath	**Value:** `'/home/dsp/dsp/ociwpci'`

Index