

SUGGESTED SCRIPTING LANGUAGE FOR THE EVLA CONTROL SCRIPTS

Barry Clark, February 11, 2003

I am presuming that the evla control software will be written in python/java (Jython). I will describe things in a python-like syntax; these will eventually connect to things in java, one hopes in a reasonably transparent fashion.

The description here primarily concerns the eventual system - EVLA antennas being correlated by the WIDAR correlator. A good deal has to be added to handle unconverted VLA antennas, the VLA correlator, NMA antennas, or VLBA antennas. When we proceed along these lines, I think it will become fairly clear how we want to handle things for these other devices by the time we can contemplate implementation for them.

The division below between properties and methods is rather arbitrary, and subject to revision. In fact, most of the things listed below as properties, and used by python as properties, will be implemented in java as methods, and transmitted between python and java via the java beans naming convention, implemented in the Jython core.

Much of what follows is the detail necessary to allow an expert to do whatever the hardware is capable of doing. Much of this will not be seen by the observer, or even by the Observing Tool. There is something to be said for reading this document backwards, the Examples first, and then seeing what the commands you read there actually do with the hardware.

Behind the scenes of the classes described below, there needs to be an "array level" set of software, that does such things as starting things up, and allowing operators to assign antennas to subarrays, stowing antennas, and so forth. This software will also provide system-wide information resources - such things as the locations of all the antennas (for the delay calculations), how the subreflector of each antenna is to be set for each band, and what the outside temperature, barometric pressure, and dewpoint are. It may also provide a storage place for different control scripts to talk to each other, so that a preamble script can leave a note for other scripts saying what calibrator or phase referencing cycle length to use, for instance. It would also provide a dictionary giving the correspondence between station IDs and antenna IDs, so that the observer, or observing tool, can easily refer to antennas either way.

The names I give here are pretty lengthy. This is so I won't have to explain what they mean. However, it is likely they would make a rather cluttered control script, so we may want to shorten them considerably.

In describing things below, I say that they result in commands being sent to a module or modules. This is not to dictate how they get to modules. In fact, my preferred implementation is that the commands merely set a switch or value inside a software object, and then all the values are timetagged and sent at once, by the execute method I include below. There are further remarks about implementation in a separate note.

ANTENNA PROPERTIES AND METHODS

From the perspective of the Control Script, it will be presented with a list of 28 antenna objects (more when the NMA comes along). The script may then set properties and call functions at will on an individual antenna basis. It may also create one or more subarray objects, and slice and dice the list of antennas among them.

If it is important that the script know whether the antenna is actually being controlled (that is, is not in the barn, has not been stowed, and has not been assigned by the operator to another subarray running another script), the antenna object will have a property 'physical', which is true if this script has actual control over the antenna. This is a read-only property. (Actually, in the inimitable fashion of python, you can set it, but doing so will accomplish nothing but confusing yourself - that setting will not carry over into the java world.) It is recommended that this property be consulted only in case of real need, and the non-physical antennas be set up in the same fashion as the physical ones. If you do this, the operator can add an antenna to your subarray, and it will seamlessly be added to your observation.

ANTENNA PROPERTIES

LoIfSetup loIfSetup

The LoIfSetup class encapsulates most of the antenna command points. Its properties and methods are given below.

double secondLoOffsetAC1

Antenna idiocratic offset frequency for WIDAR use (must be zero if VLA correlator is being used). The frequency command actually sent to the L302 module is the sum of this frequency and the one specified by in the LOSetup object. This property probably would most usefully have units kHz (the corresponding item in the LoIfSetup has units GHz). This offset applies to the sampler 1 side of downconverters A and C, and to the eight bit sampler from those downconverters.

double secondLoOffsetAC2

double secondLoOffsetBD1

double secondLoOffsetBD2

Same as above for the other samplers.

double secondLoPhaseAC1

double secondLoPhaseAC2

double secondLoPhaseBD1

double secondLoPhaseBD2

Useful for test purposes. Units turns.

long phaseReversalPattern

Walsh function pattern for the transition array. First (lsb) bit applies to the first heartbeat interval following an integer 10 seconds. Pattern is 32 bits long. The antenna objects as presented to the script should already have a default for this, and I am unable to conceive of a reason for changing it, but why not. If this were a permanent feature, rather than a transition item, I'd supply separate patterns for AC and BD.

double delayA1

An additional delay, given to the relevant correlator, for sampler A1, to be added to the other system and geometric delays by the correlator. In nanoseconds. This is an important and necessary

number for VLA antennas; we may not need it at all for eVLA antennas. In any event, default values should be installed in the antenna objects handed to the control script, and they will not need to change during a normal observation.

double delayA2
double delayA3
double delayB1
double delayB2
double delayB3
double delayC1
double delayC2
double delayC3
double delayD1
double delayD2
double delayD3

Similarly for the other samplers.

double rePhaseA1

An additional phasing to be applied by the correlator to the phase of fringes derived from the A1 sampler. It is expected that the autophasing system will work by setting this number. (Someday we might want two of these things, one to be used by autophasing, the other to be used by a WVR phase correction system.)

double rePhaseA2
double rePhaseA3
double rePhaseB1
double rePhaseB2
double rePhaseB3
double rePhaseC1
double rePhaseC2
double rePhaseC3
double rePhaseD1
double rePhaseD2
double rePhaseD3

As above for the other samplers.

integer addToSum

If the correlator is being instructed to produce a phased array output, non-zero specifies that this antenna is to be added to the output

Subarray subarray

This an object of type Subarray. This object is provided to serve two different functions, both having to do with keeping the correlator from becoming confused. First, the correlator is itself divided - the station boards on the input side of the correlator are most reasonably controlled by the Antenna objects, but the correlator backend needs the more global view provided by the Subarray objects. Making the subarray object available here is a mechanism for providing the antennas access to the CorrelatorSetup contained in the Subarray object. If the station boards and correlator backend have different ideas about the correlator setup, this can be disastrous, making the correlator output a confused muddle. This could be solved by storing a CorrelatorSetup object in the Antenna. A Subarray is called for for the second function, again concerned with the organization of the correlator, ensuring that an antenna belongs to one and only one Subarray. To maintain consistency

in these pointers, this property should not be set directly, but by calling the `addAntenna()` and `removeAntenna()` methods of the `Subarray` object, which will ensure consistency. Setting this property directly will surely cause bad things to happen.

Source source

This is an object of type `Source`. Methods and properties are given below. This is usually the same `Source` object as found in the `Subarray` object. I am at the moment unable to think of a good excuse for having it be different, but I am unwilling to swear that none exists, and including it here costs nothing.

MechanicalModel mechanicalModel

`MechanicalModel` is a class that holds the information about how to go from the location on the sky of the object you want to look at to the commands you give the antenna and subreflector servo systems. I'll discuss its methods below. The antenna object presented to the script will come with a `mechanicalModel`. Replacing this, or even using its methods is unlikely to be needed during a normal observation.

```
double pointingOffsetAz
double pointingOffsetEl
double pointingOffsetRa
double pointingOffsetDec
double pointingOffsetAzAuto
double pointingOffsetElAuto
double pointingReftime
double pointingRateAz
double pointingRateEl
double pointingRateRa
double pointingRateDec
```

These cause the antenna to be displaced relative to the phase tracking center. This is used for pointing scans, holography, and on-the-fly mosaicing. The sequence is: at a given time t , an ra , dec offset is calculated as $(t - Reftime) * RateRa$, $(t - Reftime) * RateDec$. These are added to `OffsetRa`, `OffsetDec`, and then to the RA, dec of the phase tracking center in the source object (see below). This will be converted to true azimuth and elevation by `CALC` or its equivalent. (Note that if these numbers are non-zero, `CALC` will be run twice, once to calculate delay tracking information for the correlator, and again for the antenna pointing.) To the results are added $(t - Reftime) * daz$, $(t - Reftime) * del$, and the two sets of `offsetAz` and `offsetEl`. (Two sets of `offsetAz` and `offsetEl` are provided so that one can be used from the script and the other can be used by the reference pointing analyser without interfering with each other.) These numbers are input to the `mechanicalModel`, which returns the corrected azimuth and elevation to send to the antenna servo system.

```
double focusOffset
```

Number (in mm) to be added to the focus setting returned by `MechanicalModel`.

```
double rotationOffset
```

Number (in fraction of a turn) to be added to the rotation setting returned by `MechanicalModel`.

```
InterferometerModel interferometerModel
```

The primary (and default) interferometerModel is a wrapper around CALC. Methods discussed below. There will also need to be a couple of utility models - one to just point in az-el (for tipping curves or just to park the antenna at a convenient place to work on it) and one for melting and dumping snow. We could conceivably have others - something that could track an earth satellite or something that would track a major planet by name for instance, but I regard providing these as fairly low priority.

integer correlator

Enums for WIDAR, VLA, BOTH, NONE. Various pieces of equipment, and especially the correlators themselves, need to know this.

string id

An identifier of the antenna. The script, if it wants to do something to a particular antenna, would search the set of antennas until it found one that matched. Alternately, the system could supply a dictionary that converts an antenna ID into the index into the provided list of antennas.

integer physical

A read-only property, which returns non-zero if the antenna is being controlled by this script at the time the property is being examined.

ANTENNA METHODS

precalculate (double timestamp)

Calculates delay and antenna pointing for the time given in the argument. It uses the Source object listed above. This delay and pointing are stored in an array, and become effective immediately after the time specified in the 'execute' method below. (This method could logically be part of the 'execute' method, but is not for technical reasons - CALC is more efficient if it is called for all antennas at a given time, rather than being called for several times for a given antenna.) This method is not normally executed from the script, but is called by the 'execute' method of the subarray.

calculate (double timestamp)

Calculates delay and antenna pointing for the time given in the argument. It uses the Source object moved into internal storage at the time specified in the 'execute' method below. The delay and pointing calculated are stored in the internal array behind the precalculated values. This method is used to extend an observation, and is called by a deeper level of the real-time system, not by the control script.

setPhasingSource(integer phasingBoardNo, Source source)

The WIDAR correlator will have the capability of simultaneously phasing up on more than one object within the antenna beam. The Sources provided here carry the information for doing so.

setPhasingDelay(integer phasingBoardNo, double value)

This is again something that may be necessary for individual subband phasing up. Added to the geometric delays and to the individual sampler delays mentioned above.

setPhasingPhase(integer phasingBoardNo, double phase)

Again added to the various sampler based phases mentioned above.

When we actually do this, we will probably need some sort of autophasing here, I think (but maybe not).

And finally, there is the method that actually does something.

execute(double timestamp)

This causes the properties of the Antenna object, as described by the properties and the properties of the contained objects, to be compared to the current state of the antenna hardware, and appropriate commands to be sent to the antenna, labeled with the execution time given. It also schedules, for the given time, moving the values from the 'precalculate' array into the internal, active, buffer, and moving the Source object into the internal, active, position, where it will be used by the 'calculate' method. Note that this is the *start time* of the observation, not the stop time that we are used to with the current VLA.

PROPERTIES OF THE LoIfSetup CLASS.

integer frontEnd

This could be an integer, as used by ALMA, or could be an enum with things like Q, KA, K, KU, etc. (Or whatever we can agree on in this contentious topic.) Software would examine this selection to causes commands to be sent to the F301s, the F302, and the switches associated with the L301s. It is also an input to the MechanicalModel methods that produce the focus and rotation settings for the subreflector. There are a lot of pieces of hardware being driven by the one value, but I think there is no point in providing separate control of all these switches. (It's probably worth having a code for "off"; possibly more than one - there are probably more than one place the signal path can be interrupted.) This value would also set the downconverter select switches (S1, S2, S3 in the current block diagram).

integer firstLO1

Sends commands to L301 #1. The L301 can be commanded to produce a frequency of $256\text{MHz} * \text{firstLO1} + 128\text{MHz}$. The output of the L301 may be used as a first LO, or, in a few cases as a second LO, or used as a first LO after being doubled or tripled.

double firstLO2

For the second L301. (Sorting out how one uses the two L301s is not trivial, and not discussed here.)

Downconverter downconverterAC

The Downconverter object includes the L302 LO settings used by the T304 module, and the sampler select switches for the D30x modules. This is for the A and C downconverters, usually, but not necessarily, connected as polarization pairs.

Downconverter downconverterBD

Same for the B and D downconverters.

CONSTRUCTORS OF THE LoIfSetup CLASS

LoIfSetup(integer bandcode, double ssloAC1, double ssloAC2, double ssloBD1, double ssloBD2)

Constructs an loIfSetup object with all the properties set for the given four signed sum lo frequencies (including making DownConverter objects) in the given band. This constructor would select 3bit samplers.

LoIfSetup(integer bandcode, double ssloAC3, double ssloBD3)

Same as above, but selecting 8bit samplers.

LoIfSetup(integer bandcode, integer samplers)

Does the entire setup for the given band with a set of default frequencies. if 'samplers' is zero, the 3bit samplers are selected; if non-zero, the 8bit samplers.

PROPERTIES OF THE Downconverter CLASS

This class describes a polarization pair of downconverters, either AC or BD.

double secondLo1

Frequency of the first second LO (L302 #1 for downconverter A and C, L302 #2 for downconverter B and D). The L302 design contains a lot of stuff beyond this simple number. There are two DDS chips and two sideband selects. These must be exposed by the L302 MIB to the world in general (at least for the technician screen), but I see no reason to make these subpieces available to the Control Script. (But we might want to make them available anyway, just because it is such a bother to add the feature later if some reason arises.)
Units of GHz.

double secondLoTracking1

Specifies a rate of change of frequency with time. (Note: only useful with WIDAR correlator; VLA correlator/LO system can't handle this.)
Probably most useful units would be Hz/sec.

double secondLo2

double secondLoTracking2

As above for the other L302 involved.

integer transferSwitch

Non-zero causes the transfer switch to be thrown. (These switches are located just before the T304 modules.)

integer downConverterAInputLevelControlmode

I can see the usefulness of several control modes for the input attenuator, to be implemented in the MIB software. These can be set as defined enums. The control modes I see being useful (for testing if nothing else) are:

DONTSET leave the current attenuation, whatever it is.

SETATTEN set a given attenuation to a given value

SETLEVEL set the level as measured at the input total power detector to a given voltage at the time the command is sent

ALC continually monitor the input and adjust the attenuator to maintain a given input level
This word controls the A or B downconverter, the first of the polarization pair.

double downConverterAInputAttenuator

If the control mode is SETATTEN, this is the requested setting of the input attenuator in DB.

double downConverterAInputLevel

If the control mode is SETLEVEL or ALC, this is the voltage to be set on the input power detector in the BBC.

double downConverterAInputALCtimeConstant

If the control mode is ALC, this selects the time constant in the (software implemented) ALC loop.

integer downConverterAOutput1LevelControlMode

Same control modes as for input level. For this attenuator, we may also want a mode in which the level is set or ALCed to a given rms on the sampler output. I hesitate to specify such a thing until it is a little clearer how that information may be generated. Also, at this point we probably want a code for OFF (which grounds output of the module). This controls the sampler 1 output of the A or B downconverter.

double downConverterAOutput1Attenuator

As for input level.

double downConverterAOutput1LevelSet

As for input level.

double downConverterAOutput1ALCtimeConstant

As for input level.

integer downConverterAOutput2LevelControlMode

double downConverterAOutput2Attenuator

double downConverterAOutput2LevelSet

double downConverterAOutput2ALCtimeConstant

As above for the downconverter output 2.

integer downConverterAOutput3LevelControlMode

double downConverterAOutput3Attenuator

double downConverterAOutput3LevelSet

double downConverterAOutput3ALCtimeConstant

As above for the downconverter narrow band output to the 8 bit sampler.

integer downConverterCInputLevelControlmode

double downConverterCInputAttenuator

double downConverterCInputLevel

double downConverterCInputALCtimeConstant

integer downConverterCOutput1LevelControlMode

double downConverterCOutput1Attenuator

double downConverterCOutput1LevelSet

double downConverterCOutput1ALCtimeConstant

integer downConverterCOutput2LevelControlMode

double downConverterCOutput2Attenuator

double downConverterCOutput2LevelSet

double downConverterCOutput2ALCtimeConstant

integer downConverterCOutput3LevelControlMode

double downConverterCOutput3Attenuator
double downConverterCOutput3LevelSet
double downConverterCOutput3ALCtimeConstant
As above for downconverter C or D, the second of the polarization pair.

integer walshFunction
A transition system issue. Turns the walsh function on or off. If this system controls the VLA correlator, it would do so in both places.

integer sidebandFIRA
Zero selects upper sideband setup for the FIR in the transition DAC setup. Non-zero selects lower sideband setup. This is for the first downconverter of the pair, A or B.

integer sidebandFIRC
Similarly for the C or D IFs. (In practice all four FIRs will be set up in the same way. But hey.)

integer selectSamplerA
integer selectSamplerC
Zero selects two high speed three bit samplers. Non-zero selects one, lower speed, 8 bit sampler. For each of the two sampler modules of this polarization pair. Again, having these be different should not occur in practice.

PROPERTIES OF THE MechanicalModel CLASS.

double X
double Y
double Z
The location of the antenna.

string axistype
"altz" for all our antennas, but why not.

double axis_off_x
double axis_off_y
double axis_off_z
axis_off_x is the quantity the current VLA software calls 'k'. Terminology taken from CALC descriptions.

ifTransmissionDelay
The time taken for the digitized data to get from the antenna to the control building.

loTransmissionDelay
Effectively, the time offset between the control building and the antenna.

CONSTRUCTORS OF THE MechanicalModel CLASS

MechanicalModel(Integer id)
MechanicalModel(String id)
Reads a system file, or otherwise consults a system catalog or database, searching for the given id, and extracts all the usual gang of pointing

parameters, focus/rotation parameters, antenna location, etc. I suspect we will end up with some antennas identified with integers and some with strings.

METHODS OF THE MechanicalModel CLASS.

double getRotation(integer bandcode, double elev)
Returns the recommended subreflector rotation angle for the given receiver and elevation. Causes the associated collimation values to be stored within the object for use by getAzimuth(), getElevation().

double getFocus(enum bandcode, double elev)
Returns the recommended subreflector focus setting for the given receiver and elevation. Causes the associated collimation values to be stored within the object for use by getAzimuth(), getElevation().

double getAzimuth(double az, double el)
Returns the azimuth command to give to the antenna given the input position, and using the collimation values created by the last call to getRotation() or getFocus().

double getElevation(double az, double el)
Returns the elevation command to give to the antenna given the input position, and using the collimation values created by the last call to getRotation() or getFocus().

PROPERTIES OF THE Subarray CLASS

CorrelatorSetup correlatorSetup
A CorrelatorSetup object describing the correlator setup.

METHODS OF THE Subarray CLASS

addAntenna(Antenna antenna)
Adds the named antenna to this subarray. If that antenna already belongs to a subarray (has a non-null in its subarray property) removeAntenna() would be called for that subarray before it is added to this. This ensures that an antenna can belong to only one subarray. 'self' would be installed in antenna.subarray.

removeAntenna(Antenna antenna)
Removes the named antenna from the list of antennas belonging to this subarray, and puts null into antenna.subarray.

setLoIfSetup(LoIfSetup loIfSetup)
For each of the antennas belonging to this subarray, puts the named loIfSetup into its loIfSetup property.

setCorrelator(enum {WIDAR, VLA, BOTH, NONE} correlator)
Sets the correlator property of all the antennas belonging to this subarray.

setSource(Source source)
For each of the antennas belonging to this subarray, puts the named source into its source property.

```
setPhasingSource(integer phasingBoardNo, Source source)
setPhasingDelay(integer phasingBoardNo, double value)
setPhasingPhase(integer phasingBoardNo, double phase)
    Calls the relevant method for each antenna belongin to this subarray.
```

```
execute(time)
    For a suitable selection of times (I believe time - 5dec, time + 5sec,
    time + 15sec would work well), call the precalculate() method of each
    antenna in the subarray. Then call the execute(time) method of each
    antenna in the subarray. Then suspend the current thread until 'time'.
    An interesting possibility would be to have the thread, before
    suspending itself, look for input from the array operator, so that the
    operator could talk to the array as if he were living inside the
    script.
```

NOTE

The description above has nothing in it about how the output datasets are to be labeled, and associated with Program, Project, and Proposal. This clearly needs to be made known to the correlator backend in some fashion or other. It would be appropriate for the developers of the Scheduling Tool and the Data Reduction System to specify what form that should take.

METHOD OF THE InterferometerModel CLASS

```
Pointing calc(double time, Source source, MechanicalModel place)
    The 'Pointing' object is simply a container for three doubles,
    with delay, azimuth, and elevation.
```

CONSTRUCTOR OF THE InterferometerModel CLASS

```
InterferometerModel(double time)
    Constructs a CALC type interferometer mode, and then consults system
    files or the USNO directly for current EOPs and derivatives thereof,
    extrapolated to the given MJD.
```

PROPERTIES OF THE Source CLASS

```
double ra
    J2000 right ascension, radians
```

```
double dec
    J2000 declination, radians
```

```
double parallax
    Geoparallax (that is, reciprocal of the distance in earth radii)
```

```
double dra
double ddec
double dparallax
    Time derivatives of the above (turns per day for the first two, just
    per day for the third are probably the most useful units)
```

CONSTRUCTORS OF THE Source CLASS

Besides the empty constructor, we can profitably have
Source(String ra, String dec)

Equivalent to `x = Source(); x.ra=ra; x.dec=dec`

NOTE

As defined above, the Source class does not contain anything to convey the observers intent from the Observing Tool to the Pipeline. The current VLA does a rudimentary job with a source name, a one character calcode, and a numeric qualifier. It is generally agreed that something better is needed for eVLA. It seems likely that something will be needed to be transmitted through the Control Script to convey that intent. There may be other routes for such information as well. In any case it appears appropriate for the designers of the Observing Tool and the Pipeline tool to specify what is needed in the Control Script.

PROPERTIES OF THE CorrelatorSetup CLASS

Pulsar pulsar

An object telling how to set up the pulsar timer.

METHODS OF THE CorrelatorSetup CLASS

The correlator is an immensely powerful and flexible device. The object here is to make available as much as possible of the flexibility in a reasonably comprehensible form. But it is only reasonably comprehensible; in practice one would almost always use a 'canned' setup (that can be invoked from the script). Actually using the features below, to develop a new canned setup, would be a rare exercise even for an expert.

The viewpoint taken here is that the correlator consists of the set 64 correlator chip quadrants per baseline. Each chip correlator quadrant may be connected to any two sampler-subbands. The quadrant itself may be configured to produce: 128 lags of full polarization products if the two sampler-subbands connected are a polarization pair; 256 lags of each of the two inputs; or 512 lags of the first input. (These numbers assume recirculation is not in effect; if it is, they are multiplied by the recirculation factor.) The observer specifies to which two sampler-subbands he wants the given correlator chip quadrant connected, and to which internal configuration he wants it set in the properties of a Lagset object. For the case in which the observer wishes to assign more than one correlator chip quadrant to a given subband (to increase spectral resolution), he would simply insert the same LagSet object into more than one slots of the lagSet array. The system software will notice this, and arrange to string them end-to-end. Although this array of lagSets is the sort of thing that is passed as a property elsewhere in this document, a 'set' method is suggested here for technical reasons (if it were a property, the lagSet objects would be passed across the python/java interface as Objects, without type checking).

The correlator is so flexible that it is sometimes difficult to remember that it is not infinitely flexible. Using the descriptions below it is possible to construct illegal setups. Someday, somebody will have to

write down the rules for checking the legality.

setLagSet(integer n, LagSet lagSet)

Stores the lagSet object in slot n of the lagSet descriptor array. Nominally tells what quadrant (bits 0:1) and subband correlator (bits 2:5) is being assigned. In fact, there may be another level of abstraction in this mapping to allow transparent substitution of a malfunctioning chip. The same lagSet object may be stored in several slots, meaning that that number of correlator chip quadrants will be assigned to the processing described by the lagset.

LagSet getLagSet(integer n)

Must be provided to that one can create a canned setup and then easily modify it.

setPhasing(n, PhasingSetup phasingSetup)

Stores the phasingSetup object in slot n of the phasingSetup array.

PROPERTIES OF THE Pulsar CLASS

double pulsarFrequency

For setting up the pulsar timer. Frequency in Hz.

double pulsarPhase

A phase (zero to 1) of the pulsar at the time given below.

double pulsarEpoch

A time for which the given phase and frequency are extrapolated

PROPERTIES OF THE LagSet CLASS

String sampler1

This gives the sampler to which the correlator chip quadrant is to be connected. This is a two character string, the first is a letter for the downconverter, the second is a number ("1" or "2") for the sampler. Whether "1" denotes the first 3bit sampler or the only 8bit sampler is decided by the LoIfSetup properties "selectSamplerA" and "selectSamplerC"

integer subband1

There are 16.

String sampler2

integer subband2

The second input, with meaning as for sampler1, subband1.

integer connection

1, 2, 4 for the number of concatenated correlator chip cells: 1 indicates full polarization, 2 indicates two channel parallel hand processing, 4 indicates a concatenated correlation of the single sampler-subband indicated by sampler1, subband1.

Below are parameters of the station board setup. It is perhaps a bit illogical to put them here, but it is convenient.

integer decimation

Governs the bandwidth of the subBand.

integer slot

Selects which of the frequency slots of the sampler bandwidth this occupies (there are 16*decimation) slots

integer phasedArrayOutputFormat

Phased array summation is not really a part of the correlator proper, but this is a convenient place to specify it, without having to create a new object type. The number would just select one of the various output formats supported by the phasing board output formatting FPGA.

Below tells what happens after the data leave the correlator chip. Again, a perhaps slightly illogical, but quite convenient, place to put this.

integer integrationTime

This is the LTA integration time; further integration may be done in the correlator backend, so an additional integration time is given in the Processing object. In milliseconds.

Processing processing

An object describing such things as the synthesized channel bandwidth and channel shape (for example Hanning, or uniform if there are not enough lags assigned to get the requested bandwidth any other way). It would also give the number of output channels and their selection (if the whole subband need not be saved). It may also include the specification of an interference excision algorithm, etc.

PulsarBinning pulsarBinning

An object describing the pulsar ephemeris and the usage of the correlator memory for the pulsar bins.

PROPERTIES OF THE PulsarBinning CLASS

integer numbBins

Number of pulsar bins to be implemented. An entry of 1 indicates pulsar gating with a single bin.

integer binSize

In milliseconds

double gateStart

Phase of the pulsar timing generator at which the first bin is to start receiving data. (Fractions between zero and 1).

PROPERTIES OF THE Processing CLASS

integer integration

This number specifies how many of the LTA dumps are to be added together for outputting. So the total integration time is this number times the integration time specified in the lagSet class.

integer channelShape

This selects an apodizing scheme for making the output spectrum from the lag set. First to be implemented should be uniform weighting and Hanning smoothing. It may be that we want square channels for the continuum.

integer excisionType

This is intended to be a hook upon which one might hang an interference excision algorithm. The main point of doing interference excision in the backend processing is to be able to do it on a timescale shorter than the final output integration time (that is, this option is only useful if 'integration' (above) is a number greater than 1).

integer excisionDuplicate

Output a datastream with the interference excision turned on and a second with the excision turned off.

double excisionParameter

It's bound to need one.

SYSTEM SUPPLIED STATIC FUNCTIONS

The system would provide the service of converting human readable angle measurements to radians. If I understand the Jython documentation correctly it is easy to do this transparently in some cases. That is,

```
>>>mysource.ra ="03h47m16.384"
```

Would automatically convert to radians. But

```
>>>print mysource.ra
```

```
0.9916658
```

would show the radian value. However, a toRad() function would be supplied to make the translation explicit, if desired. And a toHms() and toDms() would be supplied to do the conversion the other way, for logging purposes.

And it is convenient to have a toTurn() function, which differs from the toRad() by the factor of two pi, because time is measured in fraction of a day. There would be a system supplied time() function, returning the current time. So the equivalent of a current VLA observe file in durations is to start the file with a statement

```
>>>starttime = time()
```

And thereafter, calls to the execute() functions have argument

```
>>>mysubarray.execute(starttime + sum_of_durations)
```

The eVLA will probably run on UTC (but possibly on TAI as does the current VLA). So a facility to convert LST to MDJ+UTC is needed. So mjd(x),

where x is a floating or String representation of a sidereal time (less than 24 hours) returns the UTC time nearest in the future (or possibly in the very recent past, to avoid rounding problems) that that sidereal time will occur. So an observe file that runs in fixed LSTs would have things like

```
>>>mysubarray.execute(mjd("03:02:00"))
```

A utility function to permit (sort of) observing in dwell times rather than in durations would be easy to implement:

```
double moveTime(Double time, Source source1, Source source2)
```

would return the answer to the question "If I leave source1 at <time>, when will I get settled on source2?" (For a hetrogenous array, we would have to decide what sort of antenna is implied or provide a function for each type.)

Correcting spectral line observations for the changing velocity of the earth can be done in many different ways, including different splits of the effort between the Observing Tool, Scheduling Tool, and the on-line system. As a first suggestion, I suggest having the Scheduling Tool do nothing, the Observing Tool handle the conversion to a frequency in the solar system barycenter, and providing a function to the Control Script that does the rest:

```
dopset(Double time, Source source, LoIfSetup setup, integer lo1,
       integer lo2, integer lo3, integer lo4)
```

This function would calculate the velocity of the VLA relative to the solar system barycenter at the given time and find the component in the direction of the given source. It would then examine the four codes (lo1, 2, 3, 4 above), and apply a doppler correction to each of the four L302 oscillators as requested:

0 = do not change

1 = set the L302 oscillator appropriately

2 = set both the frequency and the rate of change of frequency.

(A function equivalent to dopset for Pulsar objects could also be provided.)

EXAMPLES

The examples below are written as a person would write them directly. One suspects that the Observing Tool will choose a more verbose format, probably eschewing the use of 'while'.

EXAMPLE 1: Phase referencing on a weak source with a quick trip to 3C 48.

```
mysub = Subarray()
for (x in antennas)      # antennas is predefined list of antennas
  mysub.addAntenna(x)
myband = LoIfSetup('K') # default continuum setup for K band
mysub.setLoIfSetup(myband)
execfile("correlator/8GHzCont.py") # creates a correlatorSetup
mysub.correlatorSetup = correlator
hh7 = Source("03h28m28.20", "31d05'44.0")
cal = Source("03h48m46.9045", "33d53'14.965")
c3c48 = Source("01h37m41.2995", "33d09'35.134")

mysub.setSource(cal)
mysub.execute(time())

while time() < mjd("03h")
  cycle = time()
  # set 10s dwell time on cal
  mysub.setSource(hh7)
  mysub.execute(moveTime(cycle, hh7, cal)+toTurn("10s"))
  # spend rest of a 45s cycle on the target
  mysub.setSource(cal)
  mysub.execute(cycle + toTurn("45s"))

mysub.setSource(c3c48)
mysub.execute(moveTime(time(), hh7, cal)+toTurn("10s"))
mysub.setSource(cal)
# dwell on 3C 48 1 minute
```

```

mysub.execute(moveTime(time(), cal, c3c48)+toTurn("60s"))

while time() < mjd("05h")
  cycle = time()
  # set 10s dwell time on cal
  mysub.setSource(hh7)
  mysub.execute(moveTime(cycle, hh7, cal)+toTurn("10s"))
  # spend rest of a 45s cycle on the target
  mysub.setSource(cal)
  mysub.execute(cycle + toTurn("45s"))
pass

```

EXAMPLE 2: Long integration in HI, with once per hour calib/bandpass on 3C286

```

mysub = Subarray()
for (x in antennas)      # antennas is predefined list of antennas
  mysub.addAntenna(x)
myband = LoIfSetup('L', 1415.686, 1415.686)
execfile("correlator/HIline.py")      # creates a correlator setup
n4258 = Source("12h18m55", "47d27'30")
cal = Source("13h31m08.2881", "30d30'32.960")
mysub.correlatorSetup = correlator
mysub.setLoIfSetup(myband)
mysub.setSource(cal)
mysub.execute(time())
while time() < mjd("18h")
  cycle = time()
  dband = myband.clone()
  dopset(time(), n4258, dband, 1, 0, 0, 0)
  mysub.setLoIfSetup(dband)
  mysub.setSource(n4258)
  execute(cycle + toTurn("5m"))
  mysub.setSource(cal)
  execute(cycle + toTurn("55m"))
pass

```

EXAMPLE 3: Two subarrays observing a radio star simultaneously in K and X bands

```

sub1 = Subarray()
sub2 = Subarray()
ix = 0
for (x in antennas)
  if(ix % 2)
    sub2.addAntenna(x)
  else
    sub1.addAntenna(x)
  ix += 1
kband = LoIfSetup('K')
xband = LoIfSetup('X')
sub1.setIfLoSetup(kband)
sub2.setIfLoSetup(xband)
execfile("correlator/8GHzCont.py")      # creates a correlatorSetup
sub1.correlatorSetup = correlator
execfile("correlator/4GHzCont.py")      # creates a correlatorSetup
sub2.correlatorSetup = correlator
rscvn = Source("13h10m47.7", "36d14'40")

```

```
cal = Source("13h31m08.2881", "30d30'32.960")
sub1.setSource(cal)
sub2.setSource(cal)
sub1.execute(time())
sub2.execute(time())
while time() < mjd("19h")
  cycle = time()
  sub1.setSource(rscvn)
  sub1.execute(cycle + toTurn("3m"))
  sub2.setSource(cal)
  sub2.execute(time())          # sub2 cal starts after sub1 cal end
  sub2.setSource(rscvn)
  sub2.execute(cycle + toTurn("6m"))
  sub1.setSource(cal)
  sub1.execute(cycle+toTurn("54m"))
pass
```