Writing Observing Command Files

James Morgan

October 1995

1 Introduction

This document describes the syntax of the observing command files used to make BIMA observations. This format was written to make it easier for the observer to plan, to program, and to debug their observing scripts.

When considering how to write an observing script for a project, the user usually first thinks about how the source (be it the main source or one of the calibrators) will be observed. The user associates with the observation the frequency of the receivers, the correlator characteristics, the duration of the integration on the source, and under what conditions the source should or should not be observed. Rather than think about what commands are necessary to test these conditions and what steps are necessary to decide whether or not to observe the source, the observer simply wants to plan their observations around simple constructs that will do that work for them.

This observing command file style greatly reduces the need for the observer to use shell programming when writing their scripts. It is important to point out that the observe file is still a C-shell script. The difference is that commands have been provided to simplify the process for the user and make it more straightforward to create and debug observing scripts.

A key ingredient of this format is the setup. All observations in a project are setup driven. Commands have been written which take setups as arguments and these do the observations properly for the user.

Another important item is that the time format is in hour-minute (hhmm) format. The time format properly (and easily) handles relative times since most observations are planned that way.

Finally, almost all of the system management (directory structure and time management) is handled by the master project script and is not the responsibility of the observer. This frees the observer from such considerations that only detract from the actual observations.

The sections that follow will fully describe the syntax used in constructing observing scripts for a project (detailing some of these other concepts) and will also include actual working examples of usage. The reader may want to have these examples visible while reading the rest of the document as they may clarify the statements that follow.

2 Introduction to setups

The basic building block of all observations is the *setup*; all observations are driven by setups. A setup is a collection of keyword-value pairs (for example, `source=venus`). The concept of a setup means that everything pertinent to the observing of a particular source is kept in one concise location. The concept of setups is potentially very powerful. Because a setup contains all the information regarding a source, it is self contained, easier to maintain, to debug, and to control under what conditions observations will be permitted. Most importantly, it allows the user to create observing files based on the characteristics of the source rather than as a collection of various commands.

To create a setup, the user calls the `setup` command. The `setup` commands should be the first commands in the observing script because a setup *must* be defined before it is used. The `setup` command has the following syntax:

```
setup name=A key=value ...
```

where `A` is the name of the created setup and `key` is the name of a setup keyword and `value` is the value associated with that keyword. The name of the created setup should be unique and, for future ease in understanding the script, descriptive. Throughout this document, setups are given simple names like 'A'; these are poor examples of setup names. More useful names, for example, might be 'tuning', 'jupiter', and '3c273'.

Only one keyword name is reserved by the command `setup`. It is the keyword `name` and it is used to specify the name of the new setup. The other keyword names for use in the `setup` command and their meanings (as well as their likely defaults) are described below. Most of these keywords are derived from and are analogous to the keywords used by the `setfreq`, `setcorr`, and `mint` commands. For further descriptions of these keywords, the reader might want to look at the documentation for these tasks. Keywords followed by task name(s) in parenthesis identify keywords that are only used by these specific tasks and do not apply to any other task.

General

setup The value of this keyword is used to include other setups. Multiple setups may be included and may also be nested.

maxobs This value specifies the maximum number of times a setup can be observed during a project. If this is not set (or is set to 0), then there is no limit to the number of times this setup is run. This parameter affects an internal variable which is initialized only once at the start of a project. If the project script needs to be stopped early for some reason and then is restarted, the value of this internal parameter is still retained (not reset). Hence, judicial use of this keyword provides the user with the control necessary to limit the number of times a setup is run (no matter how many times a script is restarted).

An obvious example of usage is for passband calibration. This is often done at the beginning of observing a project. If a script were to be restarted after the passband observation had been observed, the user would not want the script to re-observe it; that would waste value source observing time. Properly setting the `maxobs` keyword would limit the passband observation so it is done only once.

source The source name of this setup. There is no default value for this keyword. If no source is specified, then the observation task will not be run (correlator and tuning modifications may, however, still happen). Multiple sources may be specified for many of the observing tasks. These sources should be separated by commas.

catalog The catalog for `source`. If this is not defined, then it defaults to `$HATCAT/j2000.cat`. Multiple catalogs may be specified whenever multiple sources may be provided. These catalogs should be separated by commas.

ants Antennas to be used in observing. A value of "`*`" means to use all available antennas; "`xyz`" means to use just antennas 'x', 'y', and 'z'; and "`*-xyz`" means to use every available antenna except 'x', 'y', and 'z'. If this is not present, it defaults to "`*`". This keyword should generally be omitted.

task This specifies which observing task will be executed. If this is not present, it defaults to `mint`. The acceptable choices for this keyword are `mint`, `int`, and `qint`. Any other choice results in an error message and no observing task is called.

corrtask This specifies which correlator setting task will be executed. If this is not present, it defaults to `setcorr`. Currently, the only acceptable choice for this keyword is `setcorr`.

tunetask This specifies which tuning task will be executed. If this is not present, it defaults to `setfreq`. Currently, the only acceptable choice for this keyword is `setfreq`.

checktask This specifies which tuning task will be executed. If this is not present, it defaults to `cksrc`. Currently, the only acceptable choice for this keyword is `cksrc`.

Source Checking

elevlim The lower and upper source elevation limit in degrees. This range is used to limit the valid source elevations over which a setup may be observed. If not defined, this defaults to 10 and 85 degrees. NOTE: the value of this keyword corresponds to the source elevation, *not* the antenna elevation.

lstrange The time ranges over which a setup is considered observable. If the current LST is not within the window specified by this keyword, then the source is not considered visible and

observing of this setup will not continue. Up to 4 time ranges can be entered as:
`lstrange=lstmin1,lstmax1,lstmin2,lstmax2,lstmin3,lstmax3,lstmin4,lstmax4`
The format for each of these values is in the `hhmm` time format described in Section 4. Only the time ranges specified will be tested. If the minimum value is later (greater) than the the maximum value, then the time range is considered to wrap around the 24 hour mark. The default time value for the minimum time range is one hour before the current LST and the default for the maximum time range is two hours after the corresponding minimum time range. If this keyword is not present, then no time range testing will be done.

Correlator

cormode This describes the correlator configuration and the number of correlator spectral windows. Each mode also determines how the correlator frequency (`corf`) will be arranged. This keyword defaults to 8 (*i.e.* 8 windows).

corf The correlator LO settings (4 numbers in the range: 90 – 900 MHz). The default is `corf=200,400,600,800`. Setting keywords `cormode=8` and `corbw=100` gives the maximum contiguous bandwidth from 100 to 900 MHz.

Missing values should be inserted but the values assigned do not matter. For example, with `cormode=3` one might assign `corf=200,400,100,600`. The 100 value is a bogus value but is necessary so that the value of 600 MHz gets assigned properly.

corbw The correlator filter widths. Up to 4 numbers, depending on the value of cormode, may be specified. Only the values 6.25, 12.5, 25, 50, and 100 MHz are considered valid. The default is `corbw=100.0,100.0,100.0,100.0`. The first and the third widths use the same shift rate (likewise for the second and the forth widths). Widths of 25, 50, and 100 may be paired, but the first and third (and second and forth) inputs should be the same for the smaller bandwidths. Missing values should be inserted. For example, if `cormode=6`, the user might specify `corbw=25,100,25`. The 100 MHz value is ignored but is necessary so the second 25 MHz value gets properly read. Note that the fourth value may be omitted if not used.

corspec Select which correlator spectral windows are written into uv-data. `corspec` has `nspect` values: 1 or 0, to write, or omit each spectral window respectively. For example, for `cormode=5` cross-correlation, `corspec=1,0,0,0,0,1,0,0,1,1` writes lsb window 1 and usb windows 1, 4, and 5. The default is to write all spectral windows. The wideband average of each window is always written (excluding flagged channels). The wideband currently written is the average lsb, average usb, followed by `nspect=2*cormode` averages for each spectral window.

corbits The number of bits used to digitize the signal. This value may only take on values of 1 or 2 and defaults to 2 if this keyword is not present.

maxcorr Two values used to specify the maximum correlation coefficient. The spectral window is flagged if the measured correlation is greater than these values. The first value applies to the whole correlation function. The second value applies outside the central 10correlation function. If this keyword is not used, then it defaults to `maxcorr=1,1`. The second value defaults to the first if it is not present. The appropriate values depend on the source flux density and system noise. For a source flux density, S Jy, the expected correlation is

$$S/(\mathrm{Jy/K} \times T_{\mathrm{SYS}}) = 2 \times 10^{-5} \times S,$$

for an antenna gain of 140 Jy/K and $T_{sys} = 300$ K. For Gaussian noise the rms correlation is

$$dT/T_{\mathrm{SYS}} = 1/\mathrm{sqrt}(2 \times \mathrm{bw} \times \mathrm{inttime}) = 10^{-5}$$

for bw=100 MHz and inttime=50 sec.

Set `maxcorr` greater than the expected correlation by at least 5-6 times the rms to avoid flagging random noise. For a continuum source, the correlation function falls rapidly, so a smaller value for `maxcorr` can be set outside the central 10%. For spectral line emission use the default that the second value equals the first. For example, for a 10 Jy quasar `maxcorr=2.0E-3` will flag data which is ten times the expected correlation, or 20 times the expected noise.

coptions Several processing options that may be given (separated by commas) to enhance how the correlator is set up. Possible options include:

auto Set the correlator up in auto-correlation mode. The default is cross-correlation mode.

hanning Hanning smoothing reduces the resolution and suppresses the ringing due to sharp edges. By default, there is no Hanning smoothing applied.

nocal No antenna temperature calibration. The default is to perform antenna temperature calibration.

nopass No on-line passband calibration. The default is to include passband calibration.

nocable No cable length calibration. The default is to include cable length calibration.

restfqs Specifies 1 to 16 rest frequencies used to calculate the velocities for each spectral window (in GHz). The spectral windows are defined by the correlator characteristics (described by the correlator keywords above). There should be `nspect` values in GHz, corresponding to the order defined by `cormode`, not just the ones selected for writing by `corspec`. The rest frequency is set equal to the value of `freq` in any windows for which `restfqs` = 0 or the input value lies outside the acceptable range (50 – 300 GHz for mm observing).

Rest frequencies are meant to be a convenience for data reduction when observing more than one molecule in different windows, and *do not affect the observations*. For convenience, one can set `restfqs` so that each window has a header parameter written with a rest frequency for that window. The default is to set `restfqs` equal to `freq`, which is just what is wanted if only one molecular line is being observed.

If there is more than one line in a window, or the correct rest frequencies were not specified using the `restfqs` keyword, then the Miriad task `puthd` will permit the user (during post-processing) to override *ALL* of the rest frequencies.

As an example, suppose the correlator setup was chosen to observe HCN in windows 5 and 6 (88.63 GHz rest frequency) and HCO+ in windows 7 and 8 (89.1 GHz rest frequency). Also suppose that the source has a VLSR of 1000 km/s. Now, if the setup used `freq=88.63`, then the velocity scale of the whole correlator would be set up so that the doppler shifted 88.63 GHz would correspond to a velocity of 1000 km/s. This means that the HCO+ line, which is at a frequency 500 MHz higher than the HCN line, would correspond to a velocity higher by about (500 MHz / 89 GHz)*c or a velocity of 1685 km/s. NOTE: the correlator *is* observing HCO+ at the 1000 km/s redshift; but because the rest frequency has been set to HCN, the reduction software assumes that 89.1 GHz corresponds to HCN at 1685 km/s, rather than HCO+ at 1000 km/s.

passfile Cache file for passband (created in the project directory). If this is named and contains data, it will be used for the passband. Use different names for different correlator setups. If it is missing, passbands will be measured for each source. NOTE: common variable `PRESET = Y` overrides this variable.

conffile Cache file for correlator setups (created in the project directory). If this is named and contains data, it will be used for the basic correlator settings. Use different names for the different correlator setups. If it is missing, new correlator settings will be loaded from common and detector tweaking will occur. NOTE: common variable `CRESET = Y` overrides this variable.

tsysfile Cache file for TSYS/Gains (created in the project directory). If this is named and contains data, it will be used for the Calibration values. Use different names for the different correlator setups. If it is missing, The TSYS measurement will be remeasured. NOTE: common variable `TRESET = Y` overrides this variable.

Tuning

freq The observing frequency (in GHz). The nominal range for this value is between 82 and 116 GHz (but might be different depending on the doppler velocity of the source). Normally this will be the rest frequency of a spectral line. If the `freq` and `iffreq` keywords are not both present in the setup, then no frequency adjustment or tuning will take place.

iffreq The second IF frequency (in MHz). The nominal range is $\pm(90 - 900)$ MHz. If `iffreq` is negative, then the spectral line will appear in the lower sideband of the first local oscillator (LO1). This sets nominal frequency at which the spectral line appears at the correlator input (the actual frequency differs slightly because the source's doppler shift is not fully removed by correcting LO1. A value of zero for this keyword sets LO1 = freq $\times(1 - V_{\mathrm{source}}/c)$. If the `freq` and `iffreq` keywords are not both present in the setup, then no frequency adjustment or tuning will take place.

dopsrc The name of the source used for the doppler correction. If this keyword is not present, then, when tuning is done, the doppler correction applied will be with respect to a VLSR of 0.

dopcat The catalog for `dopsrc`. If this is not defined, then it defaults to `$HATCAT/j2000.cat`.

harm An optional override for the harmonic number. By definition, `harm = (LO1 - 0.07) / Xfreq`, where `Xfreq` must lie between 8.0 and 12.5 GHz. The default (0) is to use the lowest odd harmonic. NOTE: This is useful only if the mm-oscillator on an antenna fails to lock to the default harmonic.

obsline A descriptive name of the observed line (there is a maximum of 24 characters for this value).

options Several processing options that may be given (separated by commas) to describe what should be done after the frequency is set. Possible options include:

nohcheck does not perform a harmonic check. By default, a harmonic check is performed after tuning (*i.e.* confirms that the mm-oscillator is correctly phase locked to frequency LO1).

allants tune rcvr even if antenna is "off".

lockonly phaselock the mm osc, then exit.

musttune performs all receiver tuning steps whether needed or not.

If none of these tuning options are present, then the default action is to only perform tuning as necessary.

start The starting time to begin this particular observation of this source. *It is typical to omit this keyword.* If it is not present, the default implies to begin observing "now". This keyword differs from the `lstrange` keyword in that this time represents the earliest an observation can begin on this source and if the current LST has not yet past this time, the `scan` command will wait until it is reached. The value for this keyword is specified in the `hhmm` time format described in Section 4.

stop Specifies the length of a single task observation. Typical values are generally on the order of a few minutes or up to 30 minutes or so. If this keyword is not present, it defaults to one hour after the start of the observation. The value for this keyword is specified in the `hhmm` time format described in Section 4.

vis The name of the output Miriad visibility dataset. If this keyword is not present, then no observing will be done for this source.

itime The integration time per grid field in seconds. The default value is 50 seconds.

grid The grid selector. This applies to all observing tasks except the `int` task. This can be a file name from the directory `$UGRID`, a file name from the directory `$HATGRID`, or it can be a direct field descriptor of the form:
`grid="DRA(min,...),DDEC(min,...),NS(1,2,...),DATE(yyMONdd:hh:mm:ss.s)"`
where `DRA` and `DDEC` are right ascension and declination offsets measured in arcminutes on the sky. They are added onto the position of the source specified for the grid point. The index in the NS array identifies which entry in the source and catalog array will be observed. Use 1 to identify the first source in the list, 2 the second, and so on. If the number used is negative, then doppler tracking is turned off for that source; otherwise doppler tracking is turned on. Each grid point is observed after each `nreps` integrations and then back to the first point after all have been observed and continues until the stop time is reached. The `DATE` descriptor is used for solar offsets only and gives the epoch of the `DRA` and `DDEC` offsets. If a file name is given, it will point to a file containing a descriptor similar to the one above. Newlines are permitted in the grid file. The (0,0) offset must be included in the grid if observing of the source is wanted. By default (if this keyword is omitted), only the main source position is observed (*i.e.* there is no grid).

NOTE: Newlines are allowed in the grid file, but a comma must be included at the end of each line.

scale A scale factor which is multiplied into each of the offsets defined by the `grid` keyword. This is most commonly used with the generic grid files found in `$HATGRID`. The default value is 1 which means the values in the `DRA` and `DDEC` arrays are actual arcminute offsets. This keyword applies to all observing tasks except the `int` task.

polar Polarization patterns; 1-20 strings separated by commas consisting of one of the following characters for each antenna: R(right circular polarization); L(left circular polarization); or Y (or -) standard polarization. The pattern is changed each `polrep` integrations and the sequence is then repeated. If any characters other than `RLYrly-` are present, the field is treated as the name of a file (which is in either the current directory, `$UGRID`, or `$HATGRID`) and contains the Polarization patterns. The default is: `YYYYYYYYYY`.

polrep The number of times to cycle through the polarization pattern. For example, to cycle through 7 pointing positions for each polarization pattern, set `polrep=7`. The default is 1.

nreps Up to two numeric arguments may be provided for this keyword. The first argument specifies the number of integrations performed at each grid position defined by the `grid` keyword. The second specifies the number of times to go through the grand grid/polarization loop. WARNING: THIS CAN SUPERSEDE THE STOP TIME IF THE GRAND CYCLE TAKES LESS TIME! For example, to cycle through 16 polarization patterns at each grid point, and stop after 2 complete loops of pointing and polarization, set `nreps=16,2`. The default is 1 for the first value and 0 (ignored) for the second. This keyword applies to all observing tasks except the `int` task.

nchan A flag to identify the number of channels. If `nchan` is 0, it means only wideband data should be generated; +1 means both spectral and wideband are written; and -1 means both spectral and wideband data are gathered but are not doppler tracked. The actual number of channels is determined by the correlator keywords. If not present, this keyword defaults to +1.

trakopt Tracking option specifier. This defaults to `CH` for the `int` task and `CH+` for the other tasks.

cabint Interval of time (in seconds) between cable phase (or line-length) measurements. The measurement will take place at the first opportunity (break between integrations) `cabint` seconds after the previous measurement. The default value is 100 seconds for the `mint` task; 90 seconds for the `qint` task; and does not apply to the `int` task.

dbon (qint) A flag that permits observation of sources with 10 dB in the second IF turned on (this is mandatory for the Sun). Use a value of 1 to turn on the attenuation; use 0 to turn it off. The default is off; but defaults to on for the sources `SUN` and `TEST`.

3 The scan and loop Commands

To handle setups, observing commands have been written which package together the usual collection of observing commands into objects that the user can easily chain together (along with their setups) to create their own unique observing scripts. The two fundamental observing commands that deal with setups are `scan` and `loop`.

The `scan` command groups together the standard observing programs and source checking into one concise setup driven package. The syntax of the `scan` command is:

```
scan setup='setupList' [start=value] [stop=value]
```

where the `start` and `stop` keywords are optional (see below). The value assigned to the `setup` keyword may be one setup or a group of several setups. The next two sections will discuss how the `scan` command behaves for a single setup entry and then how `scan` handles multiple (or groups of) setups.

3.1 The steps scan takes for each setup

The actual logic of what steps the `scan` command takes is directly related to the contents of the setup. The order in which each step is taken and the required keyword(s) to proceed is shown in the list that follows.

- The keyword-value pairs for this setup are loaded internally.
- If the number of times this setup has been run is greater than or equal to the maximum number of times permitted by the `maxobs` keyword, then processing of this setup terminates.
- If the `source` keyword is present, then the LST time range and the source elevation are checked. If the current LST is not within the specified LST time ranges or if the source is not currently within the proper source elevation limits, then processing of this setup terminates.
- If the tuning keywords `freq` and `iffreq` are both present in the setup, then the task associated with the keyword `tunetask` is executed. If the keyword `tunetask` is not present, then the command `setfreq` is run causing the frequency to be adjusted and the receivers tuned.
- If any of the correlator keywords (`cormode`, `corf`, `corbw`, or `corbits`) are present in the setup, then the task associated with the keyword `corrtask` is executed. If the keyword `corrtask` is not present, then the command `setcorr` is run.
- If the keywords `source` and `vis` are present in the setup, then the task associated with the keyword `task` is executed. In other words, the source is observed. If the keyword `task` is not present, then the `mint` command is run.
- If the observation of the source in the setup is successful, then the number of times this setup has been successfully run is incremented.

The contents of the setup control which of these steps will be performed but, in principle, each might be done. For example, to set up the correlator, fix the frequency, and tune the receivers, a setup could be constructed which has the necessary correlator and frequency keywords but lacks the `source` keyword. The task of observing would be omitted in this case. Alternatively, by omitting the `freq` and `iffreq` keywords from a series of setups, a user can skip the process of resetting the frequency and retuning before each and every observation.

3.2 How scan handles multiple setups

When `scan` is called, the one required argument is the value associated with the `setup` keyword. One of the powerful qualities of the `scan` command is that more than one setup can be passed as its argument. The value associated with this keyword is a (single) quoted string which may be either the name of a single setup or a group of setups which are separated by operators (and perhaps grouped). The operators and the grouping characters define which setups in the list will be executed and in which order. The logical operators are the two characters '`|`' and '`,`' and correspond to the operations *OR* and *AND*, respectively. Groups are multiple setups treated by `scan` as a single object and are identified by enclosing the desired setups between a matching pair of parenthesis.

If two or more objects (setups or groups) are separated by the *OR* character (`|`), then the first object is run through `scan`. If that object is successful, then none of the remaining *OR*'d objects will be run. If the first object fails, then the test if applied to the next object. This process continues until either an object is successful; the setup list is completely exhausted; or the *AND* character (`,`) is encountered.

When objects are separated by the *AND* character (`,`), every object in the *AND* list is run through `scan`. The precedence of the operators is always left to right so the logic of the user's operation may need to be explicitly identified by the use of parentheses. When an object is a grouping of setups, the success or failure of the object is based on the resulting logic of the entire grouping; not necessarily on any one setup.

To illustrate the use of these logical operators and how they apply to individual setups and groups, several examples follow. In each of these examples, it is assumed that the setups A, B, C, and D have already been defined.

setup=	Result
'A'	Setup A is executed.
'A|B'	Setup A or setup B is executed; but not both. If the setup A could not be observed properly, then the setup B would be executed.
'A|B,C'	The same as the last case except that setup C is also executed.
'A,B'	Both setups A and B will be executed.
'A,B|C'	Setup A will be executed. After that, scan will attempt to execute setup B. If that succeeds, then setup C will be ignored; otherwise setup C will be executed.
'A,B,C,D'	All four setups are executed.
'(A,B)|C,D'	Both A and B will be run. If either of these setups is successful, then C will be skipped; otherwise, it will be executed. Setup D is always run.
'(A|B)|C|D'	Only one of the four setups would be executed. In this example, the parentheses have no impact.
'A|(B,C,D)'	Setup A is executed. If that succeeds, then the other three are ignored. If setup A does not succeed, then all three setups are run.

3.3 How scan knows when to start and stop

The `scan` command has two optional keyword arguments called `start` and `stop`. Both of these keywords take values with the `hhmm` time format detailed in Section 4. The `start` keyword indicates the earliest time that the `scan` command can execute. **Beware of dead time!** If the time identified by the `start` keyword is later than the current LST, then scan will *wait* until the `start` time is reached. This may be useful to insure that an observation of a source does not begin so early that the source has not yet risen. If the `start` keyword is not present, then the default is to begin processing the setup keyword right away. (Because a setup may also have the `start` keyword which might also delay processing of the setup, it is generally suggested that the `start` keyword be omitted from setups.)

The other optional keyword to `scan` is `stop`. This keyword value specifies how long the `scan` command has to execute all of the setups present as the value of the `setup` keyword. This stop time may be referred to as the scan length. Note that this stop time is different from the setup keyword `stop` in that the setup keyword defines the duration of one setup's observation (or task observation) while `scan`'s keyword identifies how long `scan` has to finish working through the list of input setups. This optional keyword to `scan` will, in general, not need to be set by the user since it is set to the project stopping time if it is not present.

Since the scan steps through the setup keyword in a left-to-right fashion, this stop time represents how much time should be needed to execute (or observe) each of the setups. In the most simple case (for example: `scan setup='A'`), the stop argument would need to just be equivalent to the time needed to observe the one source. However, since the setup can actually request more than one observation (for example: `scan setup='A,B,C'`), the user needs to be careful when assigning this stop time. It should be reiterated that this time has nothing to do with the individual setup stop times; it is only used to specify how long the `scan` command has to work through the setup keyword list.

Hence, there are three conditions which determine when `scan` should stop and the *earliest* of these always dominates:

- `scan` has processed the entire `setup` keyword;
- The time specified by the optional keyword `stop` has been reached;
- The project stop time has been exceeded.

3.4 The loop command

The `loop` task combines multiple calls of `scan` to handle the common observing practice of alternating between sources and calibrators. The syntax for the `loop` command is:

```
loop srcsetup='sourceList' calsetup='calList' [start=value] \
     [stop=value] [thresh=mins]
```

where the `start`, `stop`, and `thresh` keywords are optional. As with the `scan` command, if the `start` argument is not present, it defaults to now. Likewise, if the `stop` argument is not present, it defaults to the project stop time. The keyword `stop` can be used to terminate the `loop` command so that enough time remains for other `scan` or `loop` commands.

The optional keyword `thresh` specifies the *minimum* number of minutes that is considered acceptable to observe any object in the source setup list. There is only one value associated with the `thresh` keyword and it is entered in minutes. If this optional keyword is not present, the threshold will be set internally to the average time spent on the calibrator list (typically between 6 and 10 minutes).

The observing logic of the `loop` command is that it will call `scan` using the `calsetup` value (as `scan`'s `setup` argument) followed by a call to `scan` using the `srcsetup` value (again, as the `setup` argument). These two calls will alternate finishing up with a final call to `scan` with the `calsetup` value.

The `loop` command will stop when the time specified by the optional keyword `stop` has been reached or the project stop time has been exceeded; whichever is earlier. The internal structure of `loop` will attempt to insure that a final call to the `calsetup` is the last `scan` call performed before the stop conditional is reached. This may cause the final `scan` call of the `srcsetup` to be truncated. The details of how the `srcsetup` time may be adjusted are explained below.

What happens at the end of the `loop` command is a function of how much time remains. Remember that `loop` always tries to save enough time to insure that the calibrator can be done at the end. This means that the last source observation will either be truncated or lengthened to account for the scrap time. Whether the source time is made shorter or longer depends on the value of the `thresh` parameter. The details are outlined in the following paragraph.

If the amount of time that remains prior to the invocation of a source observation greater than the sum of the average source integration time `(Tsrc)`, twice the average calibrator integration time `(Tcal)`, and the threshold time `(Tthresh)`, then the source setup will be observed as normal. If the time remaining is less than `(Tsrc + (2 * Tcal) + Tthresh)` but greater than `(Tsrc + Tcal)`, then the source setup will be run but will have its integration time increased so that it ends at the `loop` stop time minus the calibrator time `(stop - Tcal)`. However,, if the time remaining is less than `(Tsrc + Tcal)` but greater than `(Tthresh + Tcal)`, then the source setup will be executed but the integration time will be shorten to insure that it ends at `(stop - Tcal)`. Finally, if the time remaining is less than `(Tthresh + Tcal)`, then the final calibrator setup is executed and `loop` exits with about `Tthresh` minutes remaining until the scheduled stop of the `loop` command. NOTE: This last (pathetic) case will only be reached if the `loop` command is *started* with less than `(Tthresh + Tcal)` minutes remaining.

4 The HHMM Time Format

Observers, in general, only need to specify times relative either to the current LST or to the project's stopping time. All observing programs identify time inputs by the two keywords `start` and `stop`. Keywords which deal with time permit one of the following forms:

```
hhmm[+hhmm|-hhmm ...]
stop[+hhmm|-hhmm ...]
lst[+hhmm|-hhmm ...]
+hhmm[+hhmm|-hhmm ...]
-hhmm[+hhmm|-hhmm ...]
```

where additional (optional) offset terms are shown in brackets (see the examples below). Wherever the expression `hhmm` is used, it symbolically represents the actual time input by the user. The expression `lst` means that the user would literally type in the characters `l`, `s`, and `t`; likewise for `stop`. In other words, the strings `lst` and `stop` are translated internally (and dynamically) and, this is important, do not represent shell variables accessible to the user.

The first form (`hhmm`) indicates that the input represents an absolute time; the second (`stop`) is translated as the project stop time; and the third (`lst`), as the current LST. The fourth form is shorthand for `lst+hhmm` and the fifth is shorthand for `stop-hhmm`.

NOTE: The project stop time (`stop`) and the current LST (`lst`) may only be specified as the first member of a time string.

NOTE: The project stop time (`stop`) may never be exceeded.

Additionally (only for VLBI), seconds of time may be included by using the modified `hhmm` form:

```
hhmm:ss[.s]
```

in place of the basic `hhmm` syntax. The colon character is required and fractional seconds may also be included. Note that because the return value is single precision, an accuracy greater than about 0.05 seconds should not be expected.

Examples of this new time syntax include:

Keyword Expression	Sets the keyword `time` to ...
`time=1350`	Exactly 13h 50m.
`time=+100`	One hour from now.
`time=lst+100`	One hour from now.
`time=-0200`	Two hours before the project stop time.
`time=stop-0200`	Two hours before the project stop time.
`time=stop-30-100`	One hour and 30 minutes before the project stop.

4.1 A Breakdown of Stop Times

Here is a summary of the various stopping times possibly encountered in a project:

Project stop time This is established by the observing schedule and can not be changed by the user at any time. No observation in the observing script is allowed to continue beyond this time. The use of the `scan` and `loop` commands insure that this limit is observed.

Loop stop time This is an optional argument to the loop command and specifies when the loop should terminate. In other words, the stop time for this command is used to specify how much time should be reserved for looping between the calibrator and source setups. If it is not given, it defaults to the project stop time. Internally, an adequate amount of time will be reserved to insure that the final calibration setup will be executed. (Where this will be not true is when the time remaining is not enough to get a proper scan. For example, assume the calibrator setup needs 10 minutes and the loop command is executed as `loop stop=+6 ...`, it will fail to observe the calibrator setup. Another example of how the calibrator setup might not be observed would be if a loop command with no stop keyword was followed by another loop command in the observer's script. Both of these are user errors and should be identified by the checker program.)

Scan stop time This determines the amount of time to allow for the successful execution the setup keyword. In general, the user should rarely have a need to specify a stop time for this command. Without the keyword, the command defaults the scan stop time to the project stop time.

Setup stop time This determines how long one observation of a setup should last. By default, if the source can be observed, this will last for one hour. Most of the observing scripts, when they set this value, set it to something between a few minutes to around 30 or 40 minutes.

Note that, moving upwards, each of the above listed times is wrapped by the time directly above it. In other words, the scan command calls the observing setup and the loop command calls scan. Also, note that the highest time in the list enforces its stop time on the stopping time of the lower items.

The only thing that the user really needs to know about the project stop time is that they can specify other times relative to it. Also, it should be emphasized that the setup stop time is specified in each source's setup as a keyword=value pair. The loop and scan stop times are specified as command line arguments and are used to control the flow through the observing script. These later two stop times are, in general, always specified *relative* to the project's stop time.

5 Setting and Retrieving Variable Values

One of the tools added to the list of commands that access setups is `svalue`. This command can be used to both set and retrieve the value of a variable for a particular setup or a list of setups. The syntax for `svalue` is

```
svalue [setup=setupList] [name=variableName] [value=integer]
```

The `name` keyword identifies the variable that should be acted on. If it is not present, then it defaults to the variable `obscnt`. The `value` keyword provides a new value to be assigned to the variable. If the `value` keyword is not present, then `svalue` simply returns the value of variable. For example, calling `svalue` without any arguments is the same as asking for the current value of the variable `obscnt`. The `value` keyword must be an integer but may be either an absolute value or a relative value (so that the current value may be incremented or decremented relative to the current value). For example, if `value=+1`, then the variable identified by the `name` keyword (or `obscnt`) will be incremented by 1. If `value=1`, the variable is set to 1.

In order to keep variables distinct for each setup, the optional keyword `setup` may be used. This keyword, if present, is used to prepend the setup name in front of the variable. This keyword can actually be a list of setups. No logic is applied to the list of setups. If more than one setup is present, the value that is written is the sum of the returned values for each setup in the list. For example, the command

```
svalue setup='A,B' value=+1
```

would increment the value of the variable `obscnt` by 1 for both setups `A` and `B` and will echo the sum of the new value of `obscnt` for each setup. Hence, if the value of `obscnt` was 2 for setup `A` and 4 for setup `B` before `svalue` was called in the above example, then, after the call, the new value of `obscnt` would be 3 for setup `A` and 5 for `B`. Also, `svalue` would echo the value of 8 (the sum of 3 and 5 from the two setups).

The `svalue` command provides a handy way to determine if something has been observed previously and how often. An example of how to use this command in an observing script is presented in the example in Section 8.5).

6 Questions about Observing Scripts

This section describes some of the most common difficulties with the observing format. The questions and answers found here are likely to change as new difficulties arise and are addressed.

What is the proper layout order of commands in an observing script?

For most standard scripts, the user should first define all the setups. The setup definition section should be followed by the individual scan and/or loop commands.

How does specifying times relative to the project stop time work in practice?

When chosing a stop time, remember that the stop time that is referenced will determine what action is requested. Section 4 provides a breakdown of the various stop times and how each one is applied. In practice, however, the user will probably just need to specify the stop times for the individual setups (*i.e.* how long to observe a particular source) and the stop time for any loop commands (to insure that time remains for any following observations). The other stop times would only be needed for more complicated observing scripts.

How do I know the time range I have for my project?

Every project can be thought to run from the project start time until the project stop time. Once the project stop time has been reached, the project will end; there are no exceptions! Since the current time will always be greater than or equal to the project starting time when the project begins, all relative times can be addressed in terms of either the current LST or the project stop time. The range of valid times for any project can always be viewed in terms of these two times. Therefore, since there is no way to know the exact time your observing file will begin (or what time will be assigned as the project stop time), it is best to arrange your observations based on the `hhmm` timing method described in Section 4.

How do I plan my observations based on this relative scheme?

A general rule of thumb is to specify stop times inside the individual setups relative to the current LST (for example, `stop=+24`) and optional stop times for the scan and loop commands as times relative to the project stop time (for example, `stop="stop-30"`).

Should I set `maxobs` for each setup?

In general, the `maxobs` keyword can be omitted from most setups. The only time it needs to be included is when the number of times a setup should be executed during one run of the project needs to be limited. This can be used to help control multiple observations of a source in the case when a project script is terminated and then restarted or when the number of times it should be observed normally needs to be monitored(see the example in Section 8.5). For example, if the user only wants to observe a passband source once at the beginning of the project, that setup should have `maxobs` set to 1. Then, if the script is terminated early (but after the passband has been observed) and then restarted, the passband observation will not be observed again (and waste valuable source observing time!).

Can the frequency and/or correlator be set just once for the entire project?

Some projects will only want to set the frequency and tune once or may only want to adjust the correlator once. Is there a simple way to set the frequency and/or correlator only once for a project? Yes.

Remember that if you include the keywords `freq` and `iffreq` in a setup, then the tuning program will be called every time you use this setup. This will be done regardless of whether tuning is necessary! Some observations do not call for tuning more than once for a project, so tuning each time the setup is used may not be desirable. To solve this, create a setup with just the tuning information (see the tuning keywords in Section 2). Also, add a call to `scan` at the beginning of your observations which uses this tuning setup as the setup argument. For example:

```
# Setups follow:
setup name=tune freq=88.6318 iffreq=-770 obsline=hcn dopsrc=orimsr
### define other setups...
# Observing follows:
scan setup='tune'
### other observing commands....
```

Likewise, if any of the correlator keywords (`cor*`) are present in a setup, then the correlator program will be called every time this setup is used. This will be done regardless of whether or not the correlator characteristics have changed! Including the correlator keywords would be appropriate when the source and calibrator use different correlator modes. However, for those observations where the same correlator setup is used for all observations, this may not be desirable. In this case, like with the tuning, the correlator should be set once at the beginning of the observation and then not again. As in the example above for tuning, a single setup can also be created which defines the correlator setup (see the correlator keywords in Section 2) and is then executed with a single call to `scan` at the beginning of the project observation.

NOTE: For more information regarding how to pick your correlator settings, refer to the Miriad tasks xcorf and corset.

Furthermore, if the tuning and the correlator are only set once per project, the setup in the above example can be modified to include both the tuning and correlator information and then set by running `scan`.

If a keyword appears in more than one nested setup, which one is used?

Nested setups are definitions that include other setups by use of the `setup` keyword. There is no restriction on the number of times a keyword can appear in a setup definition. The trick is in determining which one will actually be used when a command that uses that setup is run. A few example setups will serve to clarify this better than any other description. If a user defines their setups as:

```
setup name=A elevlim=30
setup name=B setup=A elevlim=20
setup name=C elevlim=20 setup=A
setup name=D elevlim=20 elevlim=30
```

then only setup 'B' will use an elevation limit of 20 degrees; setups 'A', 'C', and 'D' will each use an elevation limit of 30 degrees. In other words, the "rightmost" keyword will *ALWAYS* take precedence.

How is my observing script executed?

When your script is tested within the `checker` environment (and also when it is executed at the observatory), it is passed as an argument to the `project` command. The `project` command controls the starting and stopping time of every project, determines the name of the archive directory, and deals with several other system management items. Your observing script should *never* be executed directly because many of the items necessary to run your tasks successfully will not be established. See the discussion of the `project` command in the `checker` guide for more details.

7 Summary

This document has shown that observing files generated as a collection of setups frees the observer to think of the observations themselves rather than the mechanics of observing. The few commands necessary to handle setups free up the observer to think of the observations needed for a source rather than the individual commands necessary to achieve them.

At the simplist level, the user plans their observing file in the following way:

1. Define the characteristics of *one* observation of a source and put these qualities into a `setup` command.

2. Define the setups for the other sources. The setup definitions should, in general, be the first commands in a user's script.

3. Call the `scan` and `loop` commands with the defined setups as arguments. Perhaps, as needed, multiple setups can be logically combined.

4. Add relative stop times, as needed, to the `loop` and `scan` commands to insure that *all* observations will finish properly by the project stopping time. Remember, no script will be allowed to continue beyond the project stop time. The best way to insure this is to specify the `stop` keywords of the `loop` and `scan` commands as a time *relative* to the project stop time.

5. While in `checker`, run the `project` command with this observing file as its argument to insure that the script runs as expected.

Another point to consider is that all observers plan their observations based on a desired time range. Invariably, the *actual* time range assigned by the observing schedule will be different. The observing script that the user writes should be flexible enough to handle this situation. The use of relative times provides the user with a powerful and easy way to control the flow of their script without necessarily knowing the absolute stopping time of their project. Where the user should take additional care in preparing their scripts is to insure that observations do not begin too early. This is done both by specifying (in the comment section at the top of the script) the acceptable LST time ranges of the project and by specifying an absolute start time (the `start` keyword) to the first `scan` or `loop` commands. This is illustrated in each of the examples that follow.

The only drawback to using the `start` keyword is that the script will just wait around for the start time to be reached if the command is executed too early. Perhaps a better approach might be to observe calibrator setups until the specified start time is reached. For example, if the user did not want to begin the main loop until at least 15:00 LST and had a single calibrator with the setup name `3c273`, the command

```
loop calsetup=3c273 stop=1500
```

placed before the main loop would insure that the main loop would not be started until at least 15:00 LST *and* would observe (repetitively) the calibrator until that time. If this example command was called and the time was beyond 15:00 LST, it would be ignored.

8 Examples

This section provides examples of observing files. The first example illustrates how to observe a passband source followed by a single source and a calibrator. The next example extends the first example by illustrating how to use multiple calibrators during the main observation loop. Finally, the calibration step is extended to using a wide, narrow, and wide passband observation on the calibrator source. Each example is explained in the text that follows.

8.1 A Basic Example

The first example illustrates the most basic observing script. Here, a passband source is observed at the start of the project. After that, a calibrator and a source are alternately observed until the end of the project.

8.2 A Simple Script

```
#! /bin/csh -f
#
#    Observing Script for ORIMSR
#
#    PI:  Jim Morgan (UMD)
#    Email: morgan@astro.umd.edu
#    Telephone: (301) 405-6853
#    Correlator set to observe CO
#    LST Range of main source: 01:00 - 10:00.
#
# Setups follow     ==========================================
#
setup name=quality s=orimsr c=0541-056 f=0541-056 p=3C454.3
#
setup name=tuning cormode=4 corbw=50,50 corf=525,475,525,475 \
      restfqs=115.2701 obsline=co \
      dopsrc=orimsr dopcat=j2000.cat freq=115.2712 iffreq=500
#
setup name=all itime=30 elevlim=12 nchan=-1 stop=+24
#
```

```
setup name=passband setup=all source=3C454.3 vis=3C454.3 maxobs=1
#
setup name=calib setup=all source=0541-056 vis=0541-056 stop=+6
#
setup name=orimsr setup=all source=orimsr vis=orimsr \
      catalog=j2000.cat nchan=1
#
# Observing follows ==========================================
#
#  Since the tuning and correlator setup only needs to be
#  done once for the entire project, it is done as following:
#
scan setup='tuning'
#
#  Observe the passband at the beginning.  This will only
#  be done once regardless of how many times the script
#  is stopped and restarted.
#
scan setup='passband'
#
#  Start the main loop and run it until the project ends.
#
loop srcsetup='orimsr' calsetup='calib'
#
# End of script    ===========================================
#
exit 0
```

Note that long lines have been broken up by putting a slash (\) at the end of a line (*make sure there are no spaces after the slash!*) and the remainder of the command is continued on the next line(s). Also, it is assumed that the catalog called `j2000.cat` exists and contains the proper information for the sources observed.

Since, in this example, the information about the correlator and receiver tuning is identical for all observed sources, it is easier to define a generic setup (called `tuning`) that is called at the very beginning of the script. This insures that the correlator is set properly and the receivers are tuned prior to any observations.

Also, in this example, there are characteristics that are common to all of the source setups. These common features are grouped together in a setup (called `all`) and it is included by each source in its definition.

The setup for the source `3C454.3` has included the `maxobs` keyword to insure that, if the script is stopped before the end of the project and then restarted, the passband would not be observed more than once.

If, in this example, the passband was to be observed at the end rather than at the beginning of the program, the user would only need to switch the order of the `scan` and `loop` commands; add the `stop` keyword to the `loop` command to insure that it finishes with enough time to observe the passband (changing `loop` to `loop stop=stop-30` should suffice); and remove the `stop` keyword from the passband `scan` command.

It is worth noting that this example consists of only 8 command lines! The rest of the example is made up of comments describing the observations and messages designed to help the scheduler. Writing this identical observation as a shell script would take well over 80 lines of C-shell programming!

8.3 Multiple Calibrators

The next example illustrates how to do the same observation as shown in the first example but using two calibrators. If only one of the two calibrators is desired to be observed at a time, then the previous example only needs to be modified to include the `setup` command for the additional calibrator and the `calsetup` keyword in the `loop` would just list both calibrators with the *OR* operator between them.

However, to make this a more interesting example, assume that the first calibrator (`0541-056`) will be used at the start of the loop command and will continue to be used until ninety minutes after the second calibrator (`0739+016`) has begun to be used. This use of calibrator overlap might be more helpful in the calibration step of the data reduction process than if one source took over only after the first one finished.

8.4 An Example of How to Overlap Calibrators

```
#! /bin/csh -f
#
#    Observing Script for ORIMSR
#
#    PI:  Jim Morgan (UMD)
#    Email: morgan@astro.umd.edu
#    Telephone: (301) 405-6853
#    Correlator set to observe CO
#    LST Range of main source: 01:00 - 10:00.
#
# Setups follow     ==========================================
#
setup name=quality s=orimsr c=0541-056,0739+016 f=0541-056 p=3C454.3
#
setup name=tuning cormode=4 corbw=50,50 corf=525,475,525,475 \
      restfqs=115.2701 obsline=co \
```

```
      dopsrc=orimsr dopcat=j2000.cat freq=115.2712 iffreq=500
#
setup name=all itime=30 elevlim=12 nchan=-1 stop=+24
#
setup name=passband setup=all source=3C454.3 vis=3C454.3 maxobs=1
#
setup name=calib1 setup=all source=0541-056 vis=0541-056 \
      stop=+6 lstrange=0100,0700
#
setup name=calib2 setup=all source=0739+016 vis=0739+016 \
      stop=+6 lstrange=0530,stop
#
setup name=orimsr setup=all source=orimsr vis=orimsr \
      catalog=j2000.cat nchan=1
#
# Observing follows ==========================================
#
#  Since the tuning and correlator setup only needs to be
#  done once for the entire project, it is done as following:
#
scan setup='tuning'
#
#  Observe the passband at the beginning.  This will only
#  be done once regardless of how many times the script
#  is stopped and restarted.
#
scan setup='passband'
#
#  Start the main loop and run it until the project ends.
#
loop srcsetup='orimsr' calsetup='calib1,calib2'
#
# End of script    ===========================================
#
exit 0
```

Note that the `calsetup` logic was defined as an *AND* between the setups `calib1` and `calib2`. The overlap is handled completely by the `lstrange` keyword in the setups. When the LST is between 1 and 7 hours, the setup `calib2` is not considered valid and is skipped. Likewise, when the LST is beyond 7 hours, the setup `calib1` will no longer be considered valid and will be skipped. However, between 5:30 and 7 hours LST, both sources are valid and are both observed.

If the logic had been defined as an *OR* instead of an *AND*, then only the `calib1` setup would be observed while it is valid (between 01:00 and 07:00 LST) and `calib2` would only be observed after `calib1` is no longer valid (LST beyond 07:00). Also, there would be no calibrator observations made that would overlap.

8.5 Multiple Passband Observations

The next example illustrates a more realistic (and more complex) observation script. In this example, the passband observation is only attempted if the script begins at the best starting times (*i.e.* it does not begin in the middle of the requested LST time range). When observed, the passband consists of a wide, narrow, and wide band observation on the calibrator source. If the first passband source is not visible, then the second group of wide, narrow, wide observations is attempted.

Some C-shell programming has been done to determine how long the `loop` command should run. Remember that the `svalue` command returns the number of times a setup has been observed (when the `maxobs` keyword is set in a setup). Because the setups listed are separated by commas, the return value will be the sum of the number of observations for each of the four setups. Since observations of 3c454.3 will only be attempted if 3c273 is not visible, the return value of `svalue` will be 0 if neither source was visible or it will be 3 (one for each of the wide, narrow, wide observations) if one of the passband sources was able to be observed. Hence, if the passband combination has not been observed already, then the `loop` command should terminate early enough (before the project stop time) so that enough time will remain to observe the passband at the end of the project.

Notice also that the passband sources have been restricted to an LST range of (by using the setup keyword) `lstrange=0730,1730` and `lstrange=1700,0200`. Incidentally, this last one could have been equivalently specified as `lstrange=0,0200,1700,2400`.

8.6 An Example of How to Observe Multiple Passbands

```
#! /bin/csh -f
#
#    Observing Script for ORIMSR
#
#    PI:  Jim Morgan (UMD)
#    Email: morgan@astro.umd.edu
#    Telephone: (301) 405-6853
#    Correlator set to observe CO
#    LST Range of main source: 01:00 - 10:00.
#    Main source = ORIMSR
#
# Setups follow     ===========================================
#
setup name=quality s=orimsr c=0541-056 f=0541-056 \
```

```
  p=3C273.w,3c273.n,3c454.3.w,3c454.3.n
#
setup name=tuning freq=115.271201 iffreq=450 obsline=co \
  dopsrc=orimsr dopcat=j2000.cat
#
setup name=corr restfqs=115.271201,115.271201 elevlim=11 itime=50 nchan=-1
#
setup name=wide setup=corr cormode=8 corbw=100,100 corf=175,375,575,775
#
setup name=narrow setup=corr cormode=2 corbw=50,50 corf=485,415
#
setup name=orimsr setup=narrow source=orimsr vis=orimsr catalog=j2000.cat \
  nchan=1 stop=+24
#
setup name=3c273.w setup=wide source=3c273 vis=3c273.w \
  stop=+9 maxobs=2 lstrange=0730,1730
#
setup name=3c273.n setup=narrow source=3c273 vis=3c273.n \
  stop=+30 maxobs=1 lstrange=0730,1730
#
setup name=3c454.3.w setup=wide source=3c454.3 vis=3c454.3.w \
  stop=+9 maxobs=2 lstrange=1700,0200
#
setup name=3c454.3.n setup=narrow source=3c454.3 vis=3c454.3.n \
  stop=+30 maxobs=1 lstrange=1700,0200
#
setup name=0541-056 setup=wide source=0541-056 vis=0541-056 stop=+9
#
# Observing follows ========================================
#
#  Since the tuning and correlator setup only needs to be
#  done once for the entire project, it is done as following:
#
scan setup='tuning'
#
#  The passband sources will only be observed if the current
#  time is not right in the midst of the best time to observe
#  the main source (see the keywords lstrange for these setups).
#  Note, that the maxobs keyword will insure these only get done
#  once even if the script restarts.
#
scan setup='(3c273.w,3c273.n,3c273.w)|(3c454.3.w,3c454.3.n,3c454.3.w)'
#
#  Start the main loop.  The loop stop time is set according
```

```
#  to whether or not the passband observation has been done yet.
#  The amount of time reserved has been determined from the sum
#  of the stop values in the setups defined above.
#
set numobs = `svalue setup=3c273.w,3c273.n,3c454.3.w,3c454.3.n`
set quittime="stop-50"
if ($numobs > 2) set quittime="stop"
loop srcsetup='orimsr' calsetup='0541-056' stop="$quittime"
#
#  If the passband observations were missed at the beginning,
#  then get them now.  If they have already been observed,
#  then this next command will just be ignored.
#
scan setup='(3c273.w,3c273.n,3c273.w)|(3c454.3.w,3c454.3.n,3c454.3.w)'
#
# End of script    ===========================================
#
exit 0
```