The DISCOS project

Andrea Orlati[1], Marco Bartolini[1], Simona Righini[1], Marco Buttu[2], Giuseppe Carboni[2], Antonella Fara[2], Carlo Migoni[2] and Sergio Poppi[2]

[1]Institute of Radio Astronomy, INAF
[2]Observatory of Cagliari, INAF

Dec 14, 2018

DISCOS is a open source project by INAF - National Institute for Astrophysics, Italy. It is devoted to the production of the control software for INAF's radio telescopes (Medicina, Noto, Sardinia Radio Telescope).

CONTENTS

1 User Documentation **9**

1.1 User's guides: Observing with DISCOS . . . 9

1.1.1 Observing at Medicina with ESCS . . . 9

Release notes . . . 9

Overview . . . 10

What ESCS is . . . 10

Glossary: terms and abbreviations . . . 11

Troubleshooting and temporary procedures . . . 11

Setup . . . 12

Schedules . . . 12

Quick-look tools . . . 13

Checklist for Total Power schedule-based observations . . . 14

Checklist for spectral schedule-based observations . . . 16

ESCS startup . . . 18

Logistics . . . 18

Initial setup . . . 19

Antenna overall setup . . . 19

Logfile and project code . . . 20

Antenna operations . . . 20

Frontend operations . . . 22

General rules . . . 22

Backend operations . . . 22

Total Power - focus selector . . . 23

XARCOS . . . 24

Command-line measurements and acquisitions . . . 25

Raw counts readouts . . . 25

Tsys . . . 25

Weather parameters . . . 26

Manual acquisitions . . . 26

Caveat on offsets . . . 29

Command-line spectral observations . . . 30

Line rest frequency . . . 30

Target velocity parameters . . . 30

Commanding the doppler computation . . . 31

Acquiring data . . . 31

Generating and launching a schedule . . . 31

Data formats and online quick-look . . . 32

If writer is MANAGEMENT/FitsZilla . . . 32

If writer is MANAGEMENT/Point or MANAGEMENT/CalibrationTool . . . 34

Retrieving the data . . . 36
Appendix A - Monitor panels full description . . . 37
operatorInput . . . 37
AntennaBoss . . . 37
Observatory . . . 40
Mount . . . 40
GenericBackend . . . 42
ReceiversBoss . . . 43
Scheduler . . . 44
Logging Display . . . 45
Appendix B - Complete command list . . . 46
Appendix C - Output files . . . 50
Appendix D - Source catalogue . . . 50
Fault report . . . 51
1.1.2 Observing at Noto with ESCS . . . 52
Release notes . . . 52
Overview . . . 52
What ESCS is . . . 52
Glossary: terms and abbreviations . . . 52
Checklist for schedule-based observations . . . 54
ESCS startup . . . 56
Logistics . . . 56
On-site observations . . . 56
(Remotely controlled observations): under development . . . 58
Initial setup . . . 59
Antenna overall setup . . . 59
Logfile . . . 60
Antenna operations . . . 60
Frontend operations . . . 61
General rules . . . 62
Backend operations . . . 62
Total Power - focus selector . . . 63
Command-line measurements and acquisitions . . . 64
Raw counts readouts . . . 64
Tsys . . . 64
Weather parameters . . . 65
Manual acquisitions . . . 65
Caveat on offsets . . . 69
Generating and launching a schedule . . . 70
Data formats and online quick-look . . . 70
If writer is MANAGEMENT/FitsZilla . . . 71
If writer is MANAGEMENT/Point or MANAGEMENT/CalibrationTool . . . 72
Retrieving the data . . . 72
Appendix A - Monitor panels full description . . . 74
operatorInput . . . 74
AntennaBoss . . . 74
Observatory . . . 77
Mount . . . 77
GenericBackend . . . 79
ReceiversBoss . . . 80
Scheduler . . . 80
MinorServo . . . 81
Logging Display . . . 83
Appendix B - Complete command list . . . 83

Appendix C - Output files . . . 87
Appendix D - Source catalogue . . . 88
Troubleshooting and temporary procedures . . . 88
Setup . . . 89
Schedules . . . 89
(Fault report): under development . . . 90
1.1.3 Observing at the SRT with DISCOS . . . 91
Current release: what's new . . . 91
History . . . 92
Overview . . . 95
What DISCOS is . . . 95
Glossary: terms and abbreviations . . . 95
Checklist for schedule-based observations . . . 97
DISCOS startup . . . 99
Login . . . 99
What to use viewer01 for . . . 100
What to use viewer02 for . . . 100
Initial setup . . . 102
Antenna overall setup . . . 102
Active Surface setup . . . 102
Minor Servo configuration . . . 103
Logfile and project code . . . 104
Antenna operations . . . 104
Frontend operations . . . 106
General rules . . . 106
Specific notes for the L/P receiver . . . 107
Backend operations . . . 109
Total Power - focus selector . . . 110
XARCOS . . . 111
SARDARA . . . 112
Command-line measurements and acquisitions . . . 113
Raw-counts readouts . . . 113
Tsys . . . 114
Weather parameters . . . 114
Manual acquisitions . . . 116
Caveat on offsets . . . 119
Use of the MF derotator . . . 119
Rest position . . . 119
Configurations . . . 120
Rewinding . . . 122
Spectral line observations . . . 122
Line rest frequency . . . 122
Target velocity parameters . . . 123
Commanding the doppler computation . . . 123
Manually acquiring data . . . 124
Generating and launching a schedule . . . 124
Data formats and online quick-look . . . 125
DISCOS FITS output files . . . 125
Online data quick-look (FITS acquisitions) . . . 125
If writer is MANAGEMENT/Point or MANAGEMENT/CalibrationTool . . . 125
Retrieving the data . . . 126
Accessing your disk space . . . 126
Data structure . . . 126
Appendix A - Monitor panels full description . . . 127

operatorInput . . . 127
AntennaBoss . . . 128
Observatory . . . 130
Mount . . . 131
GenericBackend . . . 133
ReceiversBoss . . . 134
Scheduler . . . 136
MinorServo . . . 137
Logging Display . . . 138
Appendix B - Complete command list . . . 138
Appendix C - Output files . . . 143
Appendix D - Source catalogue . . . 143
Fault report and new feature requests . . . 144

2 Developer Documentation **145**
2.1 Development guide . . . 145
2.1.1 Releasing procedure . . . 145
Production-ready releases . . . 145
New major versions . . . 145
New minor versions . . . 145
New patch versions . . . 145
In-development versions . . . 145
Beta . . . 146
Release Candidate . . . 146
2.1.2 What's new in the next version . . . 146
Tracking of Solar System bodies . . . 146
Tracking of TLE based bodies . . . 146
Full support for long term archive . . . 146
2.1.3 Release Notes . . . 146
DISCOS 1.0 . . . 147
Remarks . . . 147
New features . . . 147
System changes . . . 149
Change log . . . 149
NURAGHE 0.6 . . . 149
New features . . . 149
System changes . . . 151
Change log . . . 152
NURAGHE 0.5 . . . 152
New features . . . 153
System changes . . . 153
Change log . . . 155
ESCS 0.4 . . . 155
New features . . . 155
System changes . . . 155
Change log . . . 156
ESCS 0.5 . . . 157
New features . . . 157
System changes . . . 158
Minor Changes . . . 159
Change log . . . 159
ESCS-Noto-0.2 . . . 159
New features . . . 159
System changes . . . 160

Change log . . . 160
2.1.4 Development workflow . . . 160
Write a hardware simulator . . . 160
Write a hardware communication library . . . 160
Write the component . . . 161
IDL interface . . . 161
Component directory . . . 162
Python components . . . 162
C++ components . . . 162
Integration tests . . . 163
Integrate the component in DISCOS . . . 163
Beta release . . . 163
Release candidate . . . 163
Final release . . . 163
How to deal with a bug . . . 163
2.1.5 DISCOS components . . . 163
DewarPositioner . . . 164
Guida per l'astronomo . . . 164
Descrizione delle API . . . 168
2.1.6 Protocols . . . 169
Backend protocol . . . 169
Introduction . . . 169
Handshake . . . 170
Messaging Protocol . . . 170
Data Types . . . 172
Request and Reply Messages . . . 172
Handling Errors . . . 177
Reference Implementation . . . 177
Considerations . . . 177
2.1.7 HOWTOs . . . 177
DISCOS deployment . . . 178
Dependencies . . . 178
Quick start . . . 179
Development . . . 182
Production . . . 184
Useful scripts . . . 185
Test Driven Development (TDD) in DISCOS . . . 188
Introduction . . . 188
Functional testing . . . 189
Unit testing . . . 193
How to integrate a device in DISCOS . . . 200
Github HOWTO . . . 200
Configuring github user and repository . . . 200
Contributing to DISCOS . . . 203
GitHub based workflow . . . 212
DISCOS customized logging system . . . 213
C++ . . . 214
Python . . . 215
Adopted formatting choices . . . 216
2.2 DISCOS control software team . . . 217
2.2.1 Core developers . . . 217
2.2.2 Astronomical referees . . . 217
2.2.3 Hardware and network support . . . 217
2.2.4 Contributors . . . 217

2.2.5 Responsibilities . . . 217
Project manager . . . 217
Support scientist . . . 218
Observation manager . . . 218
User's documentation . . . 218
Development . . . 218
Deploying and continuous integration . . . 218
Provisioning . . . 218
2.2.6 People involved in the project . . . 218
Marco Bartolini . . . 218
Marco Buttu . . . 218
Giuseppe Carboni . . . 219
Carlo Migoni . . . 219
Antonietta Fara . . . 219
Andrea Melis . . . 219
Andrea Orlati . . . 219
Sergio Poppi . . . 219
Simona Righini . . . 219

CHAPTER

ONE

USER DOCUMENTATION

1.1 User's guides: Observing with DISCOS

This section reports the user guides at the three INAF telescopes. Each guide, Medicina, Noto and SRT is tailored to telescope peculiarities and specific devices.

1.1.1 Observing at Medicina with ESCS

Release notes

Issue	Release Date	What's new
04	30/03/16	**Release for ESCS 0.4** Spectroscopy observations now available. FITS/schedule sections removed from this document and placed in separate manuals. Update on data quick-look tools and on troubleshooting/hints.
03	18/02/14	Addition of new details on BCK schedule files
02	13/12/13	Correction of errors. Addition of CalibrationTool as data writer
01	11/12/13	Issue 01

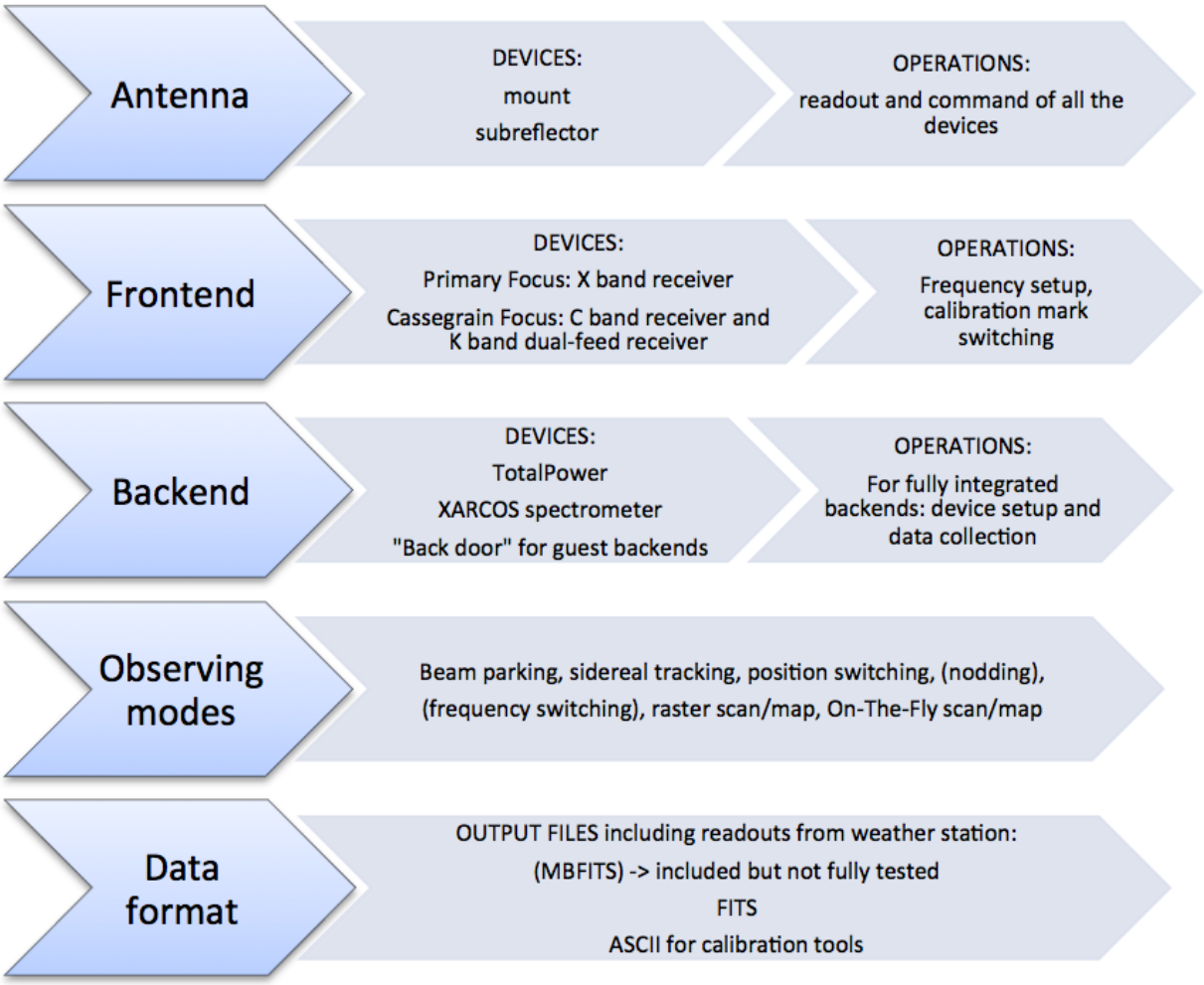
Overview

What ESCS is

ESCS is the DISCOS control software flavour produced for the Medicina and Noto radio telescopes. It is a distributed system based on ACS (ALMA Common Software), commanding all the devices of the telescope and allowing the user to perform single-dish observations in the most common modes. As of today, the code actually implemented for the telescopes (i.e. excluding the huge ACS framework) amounts to > 550000 active lines. Even VLBI (or guest-backend) observations partly rely on ESCS, as it must be used to perform the focus selection and the frontend setup.

This guide is meant to help the observer in the use of ESCS, without dealing with the "behind-the-curtains" complex details of the system. This release focuses on **single-dish continuum and spectroscopy observations**, as the fully integrated backends available on-site are the analog total power one and the XARCOS spectrometer. Since both the hardware and software implementations are continuously updated, this manual will forcibly undergo frequent revisions.

Here follows a simple schematization of the observing system, helpful to visualize all the main devices ESCS deals with and the most important operations it performs. *Notice: the features not yet available in this release are shown in brackets*.



Glossary: terms and abbreviations

> this symbol precedes commands to be given in the operatorInput panel

$ this symbol precedes commands to be given in a system shell

Beam-parking the antenna points to a fixed Azimuth-Elevation position

Bin frequency channel

DF Dual-Feed

FWHM Full-Width Half-Maximum

HPBW Half-Power BeamWidth

MS Minor Servo system, which includes the subreflector

Nodding (aka "beam switching") it involves two or more feeds. The source is alternatively observed with each of the feeds, so that there is always one feed "on source"

OTF On-The-Fly acquisition. The antenna moves according to user-defined parameters, scanning the sky at constant speed. Data acquisition is active during the scan

Position switching (aka "on-off") the beam is alternatively pointed to the source (on) and to a properly defined offset position (off)

Preset mount mode allowing only beam-parking observations, with no pointing model applied

ProgramTrack mount mode allowing tracking, OTF/raster scans, beam-parking

Raster acquisition is performed via discrete pointings (in tracking or beam-parking mode), planned to sample a certain path/area on the sky

Section acquisition stream (aka "logical channel"). E.g.: the 2 feeds of the K-band receiver used with the continuum backend produce 2 feeds x 2 polarisations = 4 sections

SF Single-Feed

Slewing motion of the antenna when it is going to the target position. Slewing is always performed at maximum speed, with no data acquisition taking place

Temporised commands user-input or schedule commands that are launched to a specific UT time by means of the "@" symbol

Time-tagged commands when commands are not temporised, the philosophy for their execution is asap (as soon as possible), but the system internally schedules them associating each operation to a properly computed time. This is particularly relevant for the positions commanded to the mount, so that the antenna motions take place along a fully controlled path (except in slewing)

TPB Total Power Backend

Tracking the antenna points to a target and, if it's a celestial target, follows its sidereal motion

Troubleshooting and temporary procedures

The following sections list the solutions to the most common problems encountered by users, including indications on how to deal with present bugs or inefficiencies in the system. The records are sorted according to the main operation areas in which the problems arise.

Note: Users are advised to read these "hints" at the beginning of the session and before attempting (possibly unnecessary) heavy-handed operations on the system.

Setup

PROBLEM:

- **The Scheduler panel shows a FAILURE flag.**

Due to a system bug, such flag **might** not indicate an actual problem. Usually, when the FAILURE status is fake, the launch of a new schedule resets it. If you are already running a schedule, stop it and re-launch it from the scan_subscan that was next due.

If you are performing the initial setup, get to the point of launching a schedule and check whether the flag resets to OK. Pay then attention to the fact that new acquisitions are correctly recorded (i.e. verify the presence of new FITS files, using the quick-look tools or checking the updates in the output data folder). If the FAILURE status persists and observations do not proceed correctly, call for assistance.

PROBLEM:

- **After a setup, the antenna slews to an unwanted position, even though I have never commanded any pointing**

This depends on the Antenna Control Unit (ACU) behavior: the ACU, after a setup, always commands the antenna to go to the last pointed position. This is very inconvenient, and ESCS cannot directly stop it at the moment. You may immediately stop the slewing by commanding, right after the setup, a properly nearby position using goTo=[Az'd',El'd'] so that, while you complete your setup procedures, the dish stops very near the present position. If the antenna is, for example, unstowing, the commanded position could be goTo=180d,80d as it is very near the 180d,90d stow position.

PROBLEM:

- **The initial setup fails, but it is difficult to assess what goes wrong.**

Instead of using the overall setup commands, give in the *operatorInput* the **individual commands** which deal with the different sub-systems, so that it is easier to identify the misbehaving element.

For example, the setupKKC command can be substituted by (the actual receiversMode code to be used depends on the desired setup):

```
> antennaSetup=KKC
> receiversSetup=KKC
> initialize=KKC
> device=0
> calOff
```

For other receivers, the codes of course vary.

Schedules

PROBLEM:

- **I've performed cross-scans in order to fine-tune the pointing, but the measured offsets are not actually applied to the following acquisitions**

System offsets, such as the ones measured with a Point acquisition, sum up to the ones indicated inside schedules ONLY if they are expressed in the same coordinate frame. This means that, if you perform observations using EQ offsets, also the fine-pointing cross-scans must be carried out in the EQ frame. The same holds for HOR scans. If there is a frame mismatch, the system offsets are automatically rejected (this bug is under fixing).

PROBLEM:

- **I've stopped an XARCOS schedule and no further operation is accepted by the system.**

Always stop your schedules with:

```
> haltSchedule
```

when you are using XARCOS; otherwise it will keep a busy status and the next operations will become impossible to perform.

PROBLEM:

- **I've launched a surely correct schedule, but parts of the system crashed.**

Pay attention to the syntax, you might have inserted unwanted characters like an extra "/": the *startSchedule* command must be either given as:

```
> startSchedule=[project]/[schedname].scd,[N]
```

or, if you have already given the project=code command:

```
> startSchedule=[schedname].scd,[N]
```

PROBLEM:

- **Dead time between consecutive subscans is much longer than expected**

Are your schedules old? With ESCS 0.4 it is necessary to delete from old schedules all the post-scan wait times, as the system now takes automatically care of computing and applying proper delays according to the deceleration ramps duration. If you specify a wait time in your post-scan procedures, it will add to the system-computed delays.

Quick-look tools

PROBLEM:

- **The calibrationtoolclient does not open**

In order for the client to start, the CalibrationTool writer must be active. So, you either wait for it to be active while the schedule runs, and then you timely start the client by giving the proper command in a terminal shell, or you

open it before launching any schedule and explicitly invoking the CalibrationTool using the command chooseRecorder=MANAGEMENT/CalibrationTool in the operatorInput panel, then starting the client in the terminal shell.

PROBLEM:

- **The IDL quick-look procedures abort, or they do not actually show the latest acquisitions**

These tools are all but refined. Ideally, they always show the last completed FITS subscan (fitslook.pro) or the last completed ON-OFF-CAL spectrum (onoff.pro). Yet, to do so while avoiding incomplete subscans or unsuitable files, they sometime fail. Folders containing one subscan only, as it is often the case with skydips, will never produce a display, because as soon as the FITS is complete a new folder is generated, and the procedure points to it. When UT midnight comes, the procedure might not automatically switch to the new YYYMYMMDD tree, so you might need to stop and restart it to go on displaying the incoming data. Please report any problem and your "wishlist" about these tools.

Checklist for Total Power schedule-based observations

Notice that actions take place in three different "locations":

- **(1)** = action to be performed in a terminal on escsRemote (.57)
- **(2)** = action to be performed in a terminal on escsConsole (.54)
- **(op)** = command to be given in the *operatorInput* panel of ESCS

Login on both (1) and (2) Connect via VNC to (1) as "observer", then use ssh to access (2).

Launch the monitors, if necessary (1):

```
$ escsConsole
```

Initial setup (op):

```
> setupCCC      (or other receiver code: XXP, KKC)
```

Tune the local oscillator, if any (op):

```
> setLO=[freq]
--> e.g. setLO=4900 - start frequency of the observed band will depend on the backend
```

Point the antenna to a reference position (op):

```
> goTo=[Az]d,[El]d
--> e.g. goTo=180d,45d
```

Always explicitly choose the Total Power backend (XARCOS might have been left active) (op):

```
> chooseBackend=TotalPower
```

Measure the cold sky signal level (op):

```
> getTpi
> setAttenuation=[sect],[att]
--> iteratively adjust attenuations until the level is about 800-1000 counts
```

Get a Tsys (op):

```
> tsys
```

Pointing optimisation (op):

```
> track=name                    (choose a proper calibrator from source catalogue)
> chooseRecorder=MANAGEMENT/CalibrationTool
--> the following command on (1):
        $ calibrationtoolclient  (to display the plots)
> crossScan=HOR,0.5d,00:00:20    (set proper parameters according to your beamsize)
> azelOffsets=0d,0d              (only if wanting to reject the measured offsets!)
```

WARNING:

- **System offsets, such as the ones measured with a Point acquisition, sum up to the ones indicated inside schedules ONLY if they are expressed in the same coordinate frame.**

This means that, if you perform observations using EQ offsets, also the fine-pointing cross-scans must be carried out in the EQ frame. The same holds for HOR scans. If there is a frame mismatch, the system offsets are automatically rejected (bug under fixing).

If needed, create a schedule (2):

```
Use schedulecreator (see its own guide):
$ basie -c [configfile] [out_directory]
```

Parse the schedule (2):

```
$ scheduleChecker [schedulename].scd
--> Move the schedule files to the observing machine
```

Launch the schedule (op):

```
> startSchedule=[project/][schedulename].scd,[N]
```

Data quick-look

- *Case A:* when using MANAGEMENT/Fitszilla, launch the raw-FITS quick-look (2):

  ```
  $ idl
  IDL> .r fitslook
  IDL> fitslook
  ```

- *Case B:* if also using MANAGEMENT/CalibrationTool, launch the quick-look (1):

  ```
  $ calibrationtoolclient MANAGEMENT/CalibrationTool
    (--> it opens only if the CalibratioTool writer is currently selected!)
  ```

Weather conditions and webcam (in a web browser)

Weather: www.med.ira.inaf.it/escs/meteo Webcam: www.med.ira.inaf.it/webcam.html

Stop the schedule (op):

```
> stopSchedule
```

Copy the data (2) —> Get the latest subfolders written in the main data folder

Stow the antenna (op):

```
> antennaPark
```

Close the monitors, if necessary (1):

```
$ escsConsole ---stop   (individual panels are closed typing "exit" in their command␣
↪lines)
```

Checklist for spectral schedule-based observations

Notice that actions take place in three different "locations":

- **(1)** = action to be performed in a terminal on escsRemote (.57)
- **(2)** = action to be performed in a terminal on escsConsole (.54)
- **(op)** = command to be given in the *operatorInput* panel of ESCS

Login on both (1) and (2) Connect via VNC to (1) as "observer", then use ssh to access (2).

Launch the monitors, if necessary (1):

```
$ escsConsole
```

Initial setup (op):

```
> setupCCC      (or other receiver code: XXP, KKC)
```

Tune the local oscillator, if any (op):

```
> setLO=[freq]
--> e.g. setLO=4900 - start frequency of the observed band will depend on the backend
```

Point the antenna to a reference position (op):

```
> goTo=[Az]d,[El]d
--> e.g. goTo=180d,45d
```

Always explicitly choose the Total Power backend (XARCOS might have been left active) (op):

```
> chooseBackend=TotalPower
```

Measure the cold sky signal level (op):

```
> getTpi
> setAttenuation=[sect],[att]  (values range from 0 dB to 15 dB, 1-dB step)
--> iteratively adjust attenuations until the level is about 700-900 counts
```

Get a Tsys (op):

```
> tsys
```

Pointing optimisation (op):

```
> track=name                    (choose a proper calibrator from source catalogue)
> chooseRecorder=MANAGEMENT/CalibrationTool
--> the following command on (1):
        $ calibrationtoolclient  (to display the plots)
```

(continues on next page)

(continued from previous page)

```
> crossScan=HOR,0.5d,00:00:20    (set proper parameters according to your beamsize)
> azelOffsets=0d,0d              (only if wanting to reject the measured offsets!)
```

WARNING:

- **System offsets, such as the ones measured with a Point acquisition, sum up to the ones indicated inside schedules ONLY if they are expressed in the same coordinate frame.**

This means that, if you perform observations using EQ offsets, also the fine-pointing cross-scans must be carried out in the EQ frame. The same holds for HOR scans. If there is a frame mismatch, the system offsets are automatically rejected (bug under fixing).

Choose and set the spectrometer (op):

```
> chooseBackend=XArcos
> initialize=[code]      (either XC00 or XK00)
```

If needed, set the spectrometer attenuators

> **setAttenuation=[sect],[att]** (values around 19-21 dB have proven to be OK to avoid saturation, but it depends on the signal level)

If needed, create a schedule (2):

```
Use schedulecreator (see its own guide):
$ basie -c [configfile] [out_directory]
```

Parse the schedule (2):

```
$ scheduleChecker [schedulename].scd
--> Move the schedule files to the observing machine
```

Launch the schedule (op):

```
> startSchedule=[project/][schedulename].scd,[N]
```

Data quick-look

- *Case A:* when using MANAGEMENT/Fitszilla, launch the raw-FITS quick-look (2):

```
$ idl
IDL> .r fitslook
IDL> fitslook

or (to inspect complete (ON-OFF)/OFF spectral acquisitions)

$ idl
IDL> .r onoff
IDL> onoff, dutyc='N_on:N_off:N_cal'
(where N_on, N_off, N_cal are integer numbers - may assume zero value)
```

- *Case B:* when using MANAGEMENT/CalibrationTool, launch the quick-look (1):

```
$ calibrationtoolclient MANAGEMENT/CalibrationTool
  (--> it opens only if the CalibratioTool writer is currently selected!)
```

Weather conditions and webcam (in a web browser)

Weather: www.med.ira.inaf.it/escs/meteo Webcam: www.med.ira.inaf.it/webcam.html

Stop the schedule (op):

```
> haltSchedule
```

Copy the data (2) —> Get the latest subfolders written in the main data folder

Stow the antenna (op):

```
> antennaPark
```

Close the monitors, if necessary (1):

```
$ escsConsole ---stop   (individual panels are closed typing "exit" in their command␣
↪lines)
```

ESCS startup

Logistics

Note: passwords are provided locally. Be sure to contact the single-dish operations staff before your session starts, in order to get the latest information.

Observations are always carried out through a VNC remote connection to a machine called escsRemote. Once connected, some of the operations must be carried out connecting via ssh to another machine called escsConsole.

escsRemote: ESCS panels

Open a VNC session and connect to:

```
192.167.189.57:2  (i.e. port 5902)
```

You will be asked to insert a password. A remote desktop will then appear. In it, the ESCS system should already be up and running, and you should see all the textual clients referring to the different containers of the system. If you don't see them, first check the bottom bar, as they might have been reduced to icons. If indeed they are all missing and you need to start the clients, open a terminal and use:

```
$ escsClients
```

This opens 9 panels at once:

- operatorInput - terminal for command line input
- antennaBoss
- observatory
- mount
- genericBackend (x2)
- receiversBoss
- scheduler

- logging

Rearrange the panels on the desktop.

In case any of them does not automatically start, you can manually open them by means of individual command lines, to be given in a terminal shell:

```
$ operatorInput
$ antennaBossTui
$ observatoryTui
$ mountTui
$ genericBackendTui BACKENDS/TotalPower
$ genericBackendTui BACKENDS/XArcos
$ receiversBossTui
$ schedulerTui
$ loggingDisplay
```

All the antenna/receiver/backend setup procedures are performed via the operatorInput window, which is also used to start/stop the schedules.

The other panels are monitors used to display a vast amount of information, see *Appendix A - Monitor panels full description* and *Appendix B - Complete command list* for a comprehensive description of their content and a list of all the commands available for the operatorInput (they can be inserted in schedules as well).

escsConsole: access to schedules, logs and data

Use escsConsole for the data quicklook and retrieval (see dedicated sections), for the generation of schedules using basie and for tools as DS9 or FV. Login credentials are specific to each project, you will connect via ssh as follows, using a terminal shell (you can take advantage of the different virtual desktops, in order not to crowd the system desktop):

```
$ ssh -X projectName@192.167.189.54
```

in your home you can find the following folders, whose names are self-explanatory:

```
~/data
~/schedules
~/logs
```

Note: users can generate subfolders according to their needs to store their schedules, but, in order to be executed, schedules must be places exactly in their ~/schedules folder.

Initial setup

Antenna overall setup

When opening an ESCS observing session, it is necessary to perform a setup which includes the antenna unstow, the mount configuration in tracking mode, the minor servo setup. This is done by means of a unique command, which is specific for the wanted receiver, to be written in the **operatorInput**. The currently available choices are:

```
> setupCCCL  for the C-band receiver (narrow bandwidth)
> setupCCC   for the C-band receiver (wide bandwidth)
```

(continues on next page)

(continued from previous page)

```
> setupXXP   for the X-band receiver
> setupKKC   for the K-band receiver
```

Note: Spaces within the command line content are **not** allowed!

The above setup command sets the antenna mount, the minor servos, the selected receiver and the default backend (TotalPower) according to **default parameters**. The antenna mode is set to **ProgramTrack** (allowing tracking and the execution of schedules), while the Local Oscillator frequency and the bandwidth are set as illustrated in the following table.

Receiver code	LO frequency (MHz)	Frontend IF band (MHz)	Backend IF band (MHz)	Observed bandwidth (MHz)	Observed band (MHz)
CCCL	4600	100-250	50-250	150	4700-4850
CCC	4600	100-900	50-780	680	4700-5380
XXP	8080	100-900	50-780	680	8180-8860
KKC	21964	100-2100	50-2400	2000	22064-24064

Notice that, depending on the devices in use, the sky frequency at the observed band starting point is given by the LO frequency plus an offset. For the present combinations of the frontends *with the total power backend*, which the above table refers to, this offset is 100 MHz. In general, the true observed band depends on the **intersection between the frontend IF band and the chosen backend filter**. The actual observed bandwidth and the band starting frequency are recorded in the output files (see *Appendix C - Output files*).

Logfile and project code

The default logfile is named **station.log**. If the user wants to change it:

```
> log=[logfilename]  (without extension)
```

Logfiles are stored in a dedicated folder (see Retrieving the data). **When schedules are run, a new logfile is automatically started**, and it is named after the schedule: [schedulename].log.

It is possible, and advisable, to insert the project code/name (a string assigned to the project by the TAC) using the command:

```
> project=[projectcode]   (e.g. project=maintenance)
```

This will make the user save time in later stages, as it will not be necessary to specify the project name in schedule-launching commands. The project code/name must correspond to an existing user, already known to the system. This means that, if its spelling does not match with the recorded name, an error rises.

Antenna operations

Besides the overall telescope setup previously described, individual commands are available to change the antenna mount status and manage its steering/pointing:

> antennaReset
: resets the antenna status after a failure, for example after the emergency stop button is released

> antennaUnstow
: it only performs the unstow procedure

> *antennaSetup=[code]*
code: CCC, XXP, KKC

it unstows the antenna (if it is stowed) then it sets the pointing model and the minor servo system according to the selected receiver. Mount is set to ProgramTrack mode

> *servoSetup=[code]*
code: CCC, XXP, KKC

it sets the minor servo system only, without affecting other systems

> *antennaTrack*
it sets the mount to ProgramTrack mode, allowing the execution of sidereal tracking, on-off, OTF scans, etc. Even beam-parking acquisitions (i.e. on a fixed Az-El position) can be now performed in ProgramTrack mode by means of the goTo command

> *goTo=[Az]d,[El]d*
it points the antenna to fixed Az-El positions, yet the mount remains in ProgramTrack mode

e.g. `> goTo=180d,45d`

> *track=[sourcename]*
if the antenna is in ProgramTrack mode and the sourcename is known within the station catalogue (which includes the most commonly observed calibrators), it directly points to the source and tracks it

e.g. `> track=3c286`

> *sidereal=[sourcename],[RA],[Dec],[epoch],[sector]*
if the antenna mode is ProgramTrack, it points to the supplied RA-Dec position and temporarily assigns the sourcename label to it. Epoch can be `1950`, `2000` or `-1`, the last one meaning that the provided coordinates are precessed to the observing epoch. The sector keyword forces the cable wrap sector, if needed: its value can be `CW`, `CCW` or `neutral`. The last option means the system will automatically choose the optimal alternative

e.g. `> sidereal=src12,319.256d,70.864d,2000,neutral`

Note: COORDINATE FORMATS Whenever celestial coordinates (Equatorial, Horizontal or Galactic) are specified, the allowed formats are:

- **decimal degrees**, using a `d` suffix, for any coordinate → e.g. `30.00d`
- **sexagesimal degrees**, with no suffix, for any coordinate → `30:00:00`
- **hh:mm:ss**, with a `h` suffix, for longitudes only → `02:00:00h` (not accepted for offsets)

Back to the commands:

> *goOff=[frame],[beams]*
it slews the antenna to an offset position, wrt a previously commanded target, along the longitude axis of the indicated coordinate frame (`EQ`, `HOR` or `GAL`). The user provides the offset value expressed in beamsizes. If the frame is HOR and target lies beyond the Elevation cutoff limits, the offset is applied in Elevation.

e.g. `> goOff=HOR,5`

> *azelOffsets=[azoff]d,[eloff]d*
it sets user-defined offsets in the Horizontal frame (degrees only). The following example sets an azimuth offset to 0.5 degrees and the elevation offset to 0.3 degrees

e.g. `> azelOffsets=0.5d,0.3d`

> *radecOffsets=[raoff]d,[decoff]d*
it sets user-defined offsets in the Equatorial frame (degrees only). The following example sets the right ascension offset to 0.3 degrees and the elevation offset to 0.0 degrees

e.g. > radecOffsets=0.3d,0.0d

> lonlatOffsets=[lonoff]d,[latoff]d
it sets user-defined offsets in the Galactic frame (degrees only). The following example sets the galactic longitude offset to 0.1 degrees and the galactic latitude offset to 0.5 degrees

e.g. `> lonlatOffsets=0.1d,0.5d`

Note: OFFSETS: the above user-defined offsets are the overall antenna offsets and they are mutually exclusive! If the user commands the offsets several times in a row (in one or different frames) only the last one will be effective. **Offsets specified within schedules, at subscan level, sum up to these user-defined offsets.**

On with the list:

> antennaStop
it stops the antenna motion, if any, and changes the mount mode to Stop

> antennaPark
it stows the antenna

Frontend operations

General rules

To change the frontend Local Oscillator frequency, use the following command:

```
> setLO=[freq1];[freq2];...;[freqN]
```

Notice the semicolon. Ideally, different values could be assigned to different IFs, thus tuning each section to a different sub-band. For the present hardware, though, this is not possible, so a single value must be specified:

e.g. `> setLO=4900`

Remember that the actually observed band begins at a frequency which is usually different from the LO one (see Initial setup)

The temperature of the calibration mark in use is recorded in the logfile whenever a Tsys is measured. It is also stored in the FITS/MBFITS output files. The calibration mark can be manually switched on and off respectively with:

```
> calOn
> calOff
```

If the user wants to perform the setup for the frontend only (without affecting the mount, the minor servo or the backend), the command is:

```
> receiversSetup=[code]   (CCC, XXP, KKC)
```

Backend operations

The backend to be used can be manually selected as follows:

```
> chooseBackend=BACKENDS/[bckname]
```

where *bckname* is the name of the backend. At present, the only available choices are:

- TotalPower

- XBackends (for the XARCOS spectrometer)

Total Power - focus selector

Warning: The TotalPower backend works as a **"focus selector"**, sending the signal from the wanted receiver to any other backend. For this reason, it **must** be set up even when acquisitions take place using another backend. This is accomplished simply using the overall setup commands (such as `setupCCC`, etc...).

Bandwidth and sampling rate can be changed using:

```
> setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]
```

where:

- *[sect]* is an integer specifying the section number
- *[startFreq]* is the initial frequency for the section (not applicable for TPB)
- *[bw]* is a double for the bandwidth (MHz)
- *[feed]* is the number of the feed connected to the section (not applicable for TPB)
- *[mode]* is the polarisation mode (not applicable for TPB)
- *[sampleRate]* is also given in MHz
- *[bins]* is the number of frequency bins for the given section (not applicable for TPB)

To leave a parameter at its previously set value, or equivalently skip it when it is not applicable, use an asterisk. For the TPB, in particular, always use:

```
> setSection=[sect],*,[bw],*,*,[sampleRate],*
```

where *bw* can be chosen from a restricted range of options (MHz):

- 300.0
- 730.0
- 1250.0
- 2000.0

These values do not correspond to the true observed bandwidth, for the reason discussed in Initial Setup. When accumulations of the data "dumps" are required before integrating them to the output file, it is possible to set the integration time as follows:

```
> integration=[N]
```

where *N* is given in milliseconds.

Each section is provided with an attenuator, to manage the signal intensity. To set the desired attenuation:

```
> setAttenuation=[sect],[att]
```

where *sect* is the section number and *att* can vary, for the TPB, form 0 dB to 15 dB, with a 1 dB step.

If a backend re-initialization is needed, use:

```
> initialize=[code]
```

(*code* can be *CCC*, *XXP*, etc)

This command resets the backend only, setting it to the default values foreseen for the specified focus/receiver. After the frontend/backend configuration is changed, it is necessary to update the value for the beamsize (HPBW), which is computed as a function of the actually observed band and is used – during the observations – to evaluate the pointing accuracy. This is accomplished using the command:

```
> device=[sect]
```

which uploads to the system the parameters relative to section number [sect] (you can generally use 0, which exists for all the receivers, unless you need to observe only with a different feed).

XARCOS

This spectrometer is fully integrated in ESCS, which means that ESCS can command the device, receive data from it and then write these data using a standard output (FITS format).

In order to configure the spectrometer, select it with the chooseBackend command and use one of the following commands in operatorInput:

> initialize=XK00
: This configuration is for the usage of the K-band central feed. It produces four full-Stokes sections respectively with bandwidths of 62.5 MHz, 8 MHz, 2 MHz and 0.5 MHz, each having 2048(x4) channels. Each digital sample has an 8-bit representation.

> initialize=XC00
: This configuration is C-band usage. It produces four full-Stokes sections respectively with bandwidths of 62.5 MHz, 8 MHz, 2 MHz and 0.5 MHz, each having 2048(x4) channels. Each digital sample has an 8-bit representation.

Ideally, configuration details can be changed using the setSection command:

```
> setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]
```

where:

- *[sect]* is an integer specifying the section number
- *[startFreq]* is the initial frequency for the section
- *[bw]* is a double for the bandwidth
- *[feed]* is the number of the feed connected to the section
- *[mode]* is the polarisation mode
- *[sampleRate]* is also given in MHz
- *[bins]* is the number of frequency bins for the given section

However, the present implementation allows the user to change only part of these parameters, in particular:

- *[startFreq]* – initial frequency, it must be in the 125-250 MHz range. This value, added to the LO frequency, gives the lowest sky frequency observed by the section. Different sections can have different start frequencies (contrarily to what happens for the TPB);
- *[bw]* – bandwidth must be chosen from the following values (all are expressed in MHz): 125.0, 62.5, 31.25, 15.625, 7.8125, 3.90625, 1.953125, 0.9765625, 0.48828125;

- *[sampleRate]* – its value (MHz) must be twice the bandwidth.

Users must not change the feed, mode and bins parameters, which are fixed as described in the configuration defaults.

A valid example of setup and setSection usage, then, is:

```
> chooseBackend=BACKENDS/XBackends
> initialize=XC00
> setSection=0,155.0,31.25,*,*,62.5,*
```

where asterisks indicate which parameters are to be set according to default values.

Warning: At present, integration time is equal to **10 seconds**. Shorter integrations might be available in the future. Data transfer requires about **2 seconds** for each integration, thus take this overhead into consideration when estimating how long your schedules are going to last.

Command-line measurements and acquisitions

Once the system and telescope setup had been completed, it is possible to manually perform measurements and observations **through the TPB**, which might as well pave the way – as preliminary checks – to longer lasting sessions carried out via schedules.

Raw counts readouts

The **raw counts readout** (called Tpi) of the signal can be obtained with:

```
> getTpi
```

The system reply consists in an array of values (one for each section). As concerns the TPB, it is important to ascertain that the Tpi lies within the 750-1000 counts range in order to keep your observations in the linear range of the backend and allow for fairly bright sources to stay within as well. If this requirement is not met, it is necessary to iteratively vary the attenuation for the needed sections and check the Tpi, until the signal intensity falls into the proper range.

As the signal level greatly varies with elevation, it is advisable to perform this operation in the elevation range that will be actually exploited during the observations or, as a general rule, at elevation=45°. Of course, the signal level is also greatly affected, especially at high frequencies, by weather conditions, therefore the attenuation tuning should be carried out again every time the conditions change.

When you are going to manually get the Tpi - in order to ascertain which attenuation values to use in your schedule - *remember to set the LO frequency and the bandwidth as they will be employed in the schedule*. After a setup command, in fact, they are set to defaults; instead, if a schedule has been previously run, these values remain set as indicated in the last schedule readout.

Note: **Tpi values include the Tp0 level** (the internal, "without sky" level, which is roughly 200 counts). Thus, when reading your output files, which contain the (Tpi-Tp0) signal, you will see a baseline 200 counts lower.

Tsys

To measure the Tsys value:

```
> tsys
```

the system replies with N values, where N is given by the total amount of (input_lines x sections). When using the TotalPower backend, N is the number of sections (2 for single-feed receivers and 14 for the MF).

Note: The last measured Tsys value will be stored in the system and used, *if the FITS format is selected for data storage*, to get a counts-to-Kelvin conversion factor, in turn applied to all the following acquisitions, until a new Tsys is measured. The FITS file will contain the raw data (in counts) and also a table with the data stream calibrated (in K) using this counts-to-Kelvin factor.

Weather parameters

The **weather station** measurements can be retrieved with:

```
> wx
```

the reply will list ground temperature (°C), relative humidity (%), atmospheric pressure (hPa), wind speed (dummy value, at present). Updated values are available every 10 seconds.

Opening a web browser, you can also access a web page where many weather parameters are displayed in real time. This is a beta-version monitor, please report any misfunctioning.

Manual acquisitions

When performing manually commanded acquisitions, it is necessary to **select the recording device**:

```
> chooseRecorder=[string]
```

where string can be:

- **MANAGEMENT/Point** (default) text output in the logfile, used for pointing calibration
- **MANAGEMENT/CalibrationTool** text output in .dat file, used for pointing calibration
- **MANAGEMENT/FitsZilla** if FITS output is desired
- (MANAGEMENT/MBFitsWriter) if MBFITS is preferred – not yet available

Note: When recording **manually-acquired** data in FITS format, the **output files** are stored in a peculiar path which is not accessible to users. This implies that they also cannot be shown by the quick-look procedure. This temporary situation, which requires the involvement of a system administrator in order to retrieve the data, will be corrected in the next release of ESCS.

Once the recorder is set, acquisitions on a target can be performed as follows. First, set the target:

```
> track=[sourcename]     (if the source is included in the system
                         catalogue)
```

For the available catalogue see *Appendix D - Source catalogue*. To set a generic target:

```
> sidereal=[sourcename],[RA],[Dec],[epoch],[sector]
```

(see *Antenna operations* for details)

Here follow the commands to be used to manually record your data (remember that the backend must have been properly set up and the target must have been specified as explained above):

```
> initRecording=[scn]
```

where [scn] in the scan number to be assigned to the acquisition. The `initRecording` command prepares the data recording. Then:

```
> startRecording=[sub_scn],[duration]
```

creates the output file and begins the data recording; [sub_scn] is the subscan number, [duration] is the acquisition duration, expressed as `hh:mm:ss`. Once the acquisition is completed, the user can launch another subscan and record the data with another `startRecording` command.

Finally, once the user wants to close the scan, the command to be used is:

```
> terminateScan
```

Output files will be found in the usual auxiliary folder where all the manual acquisitions are destined.

Example: acquisition of a sidereal scan on 3c123 composed by 2 subscans, each lasting 40 s, preceded by an off-source Tsys measurement:

```
> chooseRecorder=MANAGEMENT/FitsZilla
> track=3c123
> goOff=HOR,5
> waitOnSource
> tsys
> azelOffsets=0.0d,0.0d
> initRecording=1
> startRecording=1,00:00:40
> startRecording=2,00:00:40
> terminateScan
```

Pointing scans

Command cross-scans across a previously selected target (by means of the track or sidereal commands):

```
> crossScan=[subscanFrame],[span],[duration]
```

where subscanFrame is the coordinate frame along which the scan is performed (`eq`, `hor` or `gal`), span is the spatial length on sky of the individual subscan (i.e. one line of the cross) expressed in degrees, duration i s the time length espressed in hh:mm:ss,

e.g. `> crossScan=HOR,1.0d,00:00:30`

corresponds to one cross-scan carried out in Horizontal coordinates (one line along El, one line along Az), each line being 1° in span. Each subscan lasts 30 seconds, thus the resulting scan speed is 2°/min.

When the MANAGEMENT/Point writer is used, the cross-scan produces text output in the logfile only (no output file is recorded). This output text contains information obtained by the automatic processing of the subscans. In particular, a Gaussian fit is performed in order to measure the source position and estimate the **pointing** offsets. If the fitting procedure in successful and the achieved offsets are considered plausible, pointing **corrections are immediately applied**. This means that, if no user-defined offset is commanded (or cleared) afterwards, the measured offsets remain active and are applied to the following observations.

Here follows the function that is separately fitted to latitude and longitude subscans:

y(x)=A*e^W + ax +c

where:

- W = -2.7725887 * F^2
- F = (x-μ)/FWHM
- μ = abscissa of peak

The results are given in the logfile, in the following sequence of lines:

LATFIT latoff fwhm A a c i
LONFIT lonoff fwhm A a c i
OFFSET avlon avlat lonoff latoff lonflag latflag
XOFFSET avlon avlat lonoff*cos(lat) latoff lonoff_err *(cont.)*
latoff_err lonflag latflag
XGAIN target avlon avlat lonampl lonampl_err latampl *(cont.)*
latampl_err lonFWHM lonFWHM_err latFWHM latFWHM_err *(cont.)*
flux lonflag latflag

where (all angles in degrees):

latoff = latitude offset
lonoff = longitude offset
i = number of iterations performed by the fitting procedure
avlon = average longitude of peak (in same coordinate frame as the subscan execution)
avlat = average latitude of peak (in same coordinate frame as the subscan execution)
lonflag = fit result for longitude subscans
(1 = plausible fit, 0 = non plausible fit, -1 = fit did not converge)
latflag = fit result for latitude subscans
(1 = plausible fit, 0 = non plausible fit, -1 = fit did not converge)
lonoff_err = error on longitude offset
latoff_err = error on latitude offset
target = target name
lonampl = amplitude measured on longitude subscans (K)
lonampl_err = error on amplitude measured on longitude subscans (K)
latampl = amplitude measured on latitude subscans (K)
latampl_err = error on amplitude measured on latitude subscans (K)
lonFWHM = FWHM measured on longitude subscans
lonFWHM_err = error on FWHM measured on longitude subscans
latFWHM = FWHM measured on latitude subscans
latFWHM_err = error on FWHM measured on latitude subscans
flux = catalogue target flux (Jy), if available (otherwise it is put to 0.0).

.

Note: it is possible to **include such pointing scans using the MANAGEMENT/Point writer in schedules** as well. For example, an improved pointing can be achieved setting the first scan on a source as a /Point scan, then – in case the fit is successful – the following scans (e.g. producing FITS or MBFITS files) will hold the offsets optimising the pointing, given that no user-defined offset is updated by means of an explicit `radecOffsets`, `azelOffsets` or `lonlatOffsets` command.

Focus scans

Command a focus scan on a previously selected target (by means of the track or sidereal commands):

```
> focusScan=[span],[duration]
```

where span is the length run on the z-axis expressed in mm, duration is the time length expressed in `hh:mm:ss`

e.g. `> focusScan=60,00:01:00`

The `focusScan` command can be used inside schedules as well. See the separate guide to schedules for details.

Skydips

Skydip scans are indispensable in order to characterize the atmosphere. They consist in moving the telescope along a vast span in elevation (at fixed azimuth) while sampling with a backend. Their analysis allows the user to quantify the atmospheric opacity τ. There are different ways to perform this:

```
> skydip=[El1],[El2],[duration]
```

e.g. `skydip=20d,80d,00:05:00` performs a skydip between 80 and 20 degrees (at the current azimuth position), the scan will take 5 minutes (speed is thus 12 °/min). The arguments must be in the range 10-88.

The jolly character is supported for the elevation arguments. Example: `skydip=*,*,00:04:00` will perfom the skydip between the default values for elevation (15° and 90°). Please notice that pointing corrections are disabled.

Since no backend recording is automatically enabled by this command, remember to activate the FitsZilla recorder before launching the command, in order to save the data! This command can be used within schedules as well. See the separate guide to schedules for details.

Note: At present skydips are always performed **downwards**, i.e. starting from the highest elevation given in the command. **The greatest commandable elevation is 88°**, since the skydip, being an OTF subscan, will be additioned of an initial acceleration ramp – the length of which is proportional to the scanning speed.

Caveat on offsets

As seen in *Antenna operations*, there are commands used to set (or null) user-defined offsets. They are: `radeOffsets`, `azelOffsets` and `lonlatOffsets`. Such commands set **overall offsets** which **remain active** until they are explicitly changed/nulled by another call of one of the three commands.

Further offsets, having for example the purpose of pointing the antenna to an off-source position, are specified inside schedules, at the subscan level (see the separate guide to schedules). **The subscan-level offsets sum up to the overall offsets**, and they are zeroed by default every time a new subscan is commanded.

Command-line spectral observations

Spectral line observations can either take place exploiting proper schedules (see dedicated checklist, and the documents on how the schedules must be written or how to produce them with the *basie* tool) or by means of command-line operations. Here this command-line scenario is described.

Line rest frequency

The spectral line rest frequency must be provided to the system using the following command:

```
> restFrequency=[value]
```

[value] is given in MHz and is a multiple argument: it can list a different value for each section - as long as XARCOS is the backend in use (not all the backends allow this sub-tuning). Specifying a single value assigns the rest frequency to *all* the sections.

e.g. `> restFrequency=22235.8`

used with the 7-feed K-band receiver implies that all the 14 sections are meant to observe the same line. Instead, providing different values allows to tune the sections on different lines (notice the semicolon used as a separator):

e.g. `> restFrequency=22235.8;22235.8;22080.0,22080.0,...,`

notice that all the N values (these being 14 for the K-band MF) must be explicitly provided or errors will rise.

The specified values will hold until different ones are commanded, or until a new general *setup* command is entered.

Target velocity parameters

When spectral line observations are required, targets must be provided with proper indications on their radial velocity, i.e.:

- **Radial velocity**
- **Reference frame**
- **Definition**

in order for the system to compute, at runtime, the frequency drift due to the line of-sight doppler contribution.

Once a target has been defined, for example commanding a `sidereal` positioning, its velocity can be provided by means of a manual command in the operatorInput panel, i.e.:

```
> radialVelocity=[vrad],[vref],[vdef]
```

Where

- [vrad] (radial velocity) is in km/sec if vdef is *not* Z
- [vref] (reference frame) can be one of the following:
  - **BARY**: Solar System BARYCENTRE
  - **LSRK**: Kinematic Local Standard of Rest
  - **LSRD**: Dynamical Local Standard of Rest
  - **LGRP**: Local Group
  - **GALCEN**: Galactic Centre

  - **TOPOCEN**: TOPOCENTRIC (observer's frame)
- [vdef] (velocity definition) can either be:
  - **RD**: Radio Definition
  - **OP**: Optical Definition
  - **Z**: stands for Redshift

The wildcard code `*` can be provided for all the required arguments and will be considered as "keep the present values".

The specified velocity parameters are valid until a new target is commanded.

Note: The `radialVelocity` command overrides any other velocity value that might have been differently expressed (e.g. if the target was selected from the source catalogue with the `track` command).

Commanding the doppler computation

Once the above information have been provided to the system, the computation of the line observed frequency can take place. Use the command:

```
> fTrack=[dev]
```

It collects all the required data from the antenna, the back-end and the front-end, plus the information provided by the user (see the above `radialVelocity` and `restFrequency` commands), then it tunes the telescope devices in order to centre the line(s) in each section bandwidth. The command lets the user select which device [dev] is asked to perform the tuning:

- **LO**: only the front-end local oscillator is moved
- **ALL**: first of all the front-end local oscillator is tuned, then the back-end - if it allows such a sub-tuning - also performs a further frequency adjustment, in order to centre the line(s) in the various sections. This option is useful in case multiple rest frequencies are to be observed, yet a complete success cannot be guaranteed as the dopplered frequencies might turn out to be incompatible with the section bandwidths and the LO ranges. In case at least one line lies outside the RF band of the receiver or the back-end input bandwidth, an error rises.

Acquiring data

See *Command-line measurements and acquisitions*

Generating and launching a schedule

A schedule is a set of files where all the geometry/timing/frequency details of a sequence of data acquisitions are specified, according to a syntax that enables ESCS to read and execute them. The detailed structure of the several files composing a schedule is explained in this separate guide `pdf` Schedules for the most common continuum and spectroscopy observing modes (OTF cross-scans and maps, raster maps and ON-OFF) can be easily generated using a tool called **basie**. It can be found, together with its user manual, here: Basie repository

Once a schedule is ready, it must be copied to the folder reserved to the project. There, the schedule formal consistency can be tested using the scheduleChecker command given within a terminal (*not* in the operatorInput):

```
> scheduleChecker [schedname].scd
```

Only the syntax correctness will be verified. If errors are present, a reply will briefly address them indicating their position inside the files.

To launch a schedule, simply use:

```
> startSchedule=[project/][schedname].scd,[N]
```

project is the unique string assigned to every project. It can be omitted if the project command had been used. If both the choices are made, i.e. if the project name was set with project but a string is also inserted in the startSchedule command, the latter overrides the former. Then follows the schedule name, in particular the name of the SCD file. *N* is the identifier of the scan or subscan from which ESCS must start reading it – it is particularly useful in case of a sequential (i.e. not time-based) schedule. *N* can be the scan number, e.g. 2, or the scan_subscan specification, e.g. 2_5.

You can specify **when to start** the schedule (in UT):

```
> startSchedule=[project/][schedname].scd,[N]@[DOY-HH:MM:SS]
```

ESCS reads the configuration parameters from the schedule, which can be relative both to the receiver and the backend, and accordingly sets these devices. This might take several seconds, especially when using the MF receiver. While the setup takes place, several values change in the TPB monitor. The last operation is the upload of the first pointing/scan read from the schedule, whose parameters will show up in the bottom section of the AntennaBoss monitor. During the scans, all the three flags in the lower part of the AntennaBoss monitor must be a green "@". Pay attention to the "Tracking" one. It should turn to a red "o" only when the antenna is slewing between scans on the same source, or when slewing to/from a new source. **If tracking is not correct for some consecutive seconds during a scan, something is wrong.** Sequential schedules run ad libitum. To abruptly interrupt the running schedule, truncating the ongoing acquisition:

```
> stopSchedule
```

Instead, to stop the schedule allowing the completion of the present file:

```
> haltSchedule
```

Data formats and online quick-look

Details on the FITS file structure are given in a separate document. Download it here: *pdf*

Waiting for a comprehensive GUI which is going to include also a real-time preview of the data under acquisition, users are provided with two different tools in order to inspect the data produced by the TPB and XARCOS.

If writer is MANAGEMENT/FitsZilla

When acquiring FITS files through a schedule, there is an IDL tool available for the almost-realtime quick-look of the saved data.

Open a terminal on *escsConsole*. Launch IDL:

```
$ idl
```

At the IDL prompt, compile and run the program fitslook.pro:

```
IDL> .r fitslook
IDL> fitslook
```

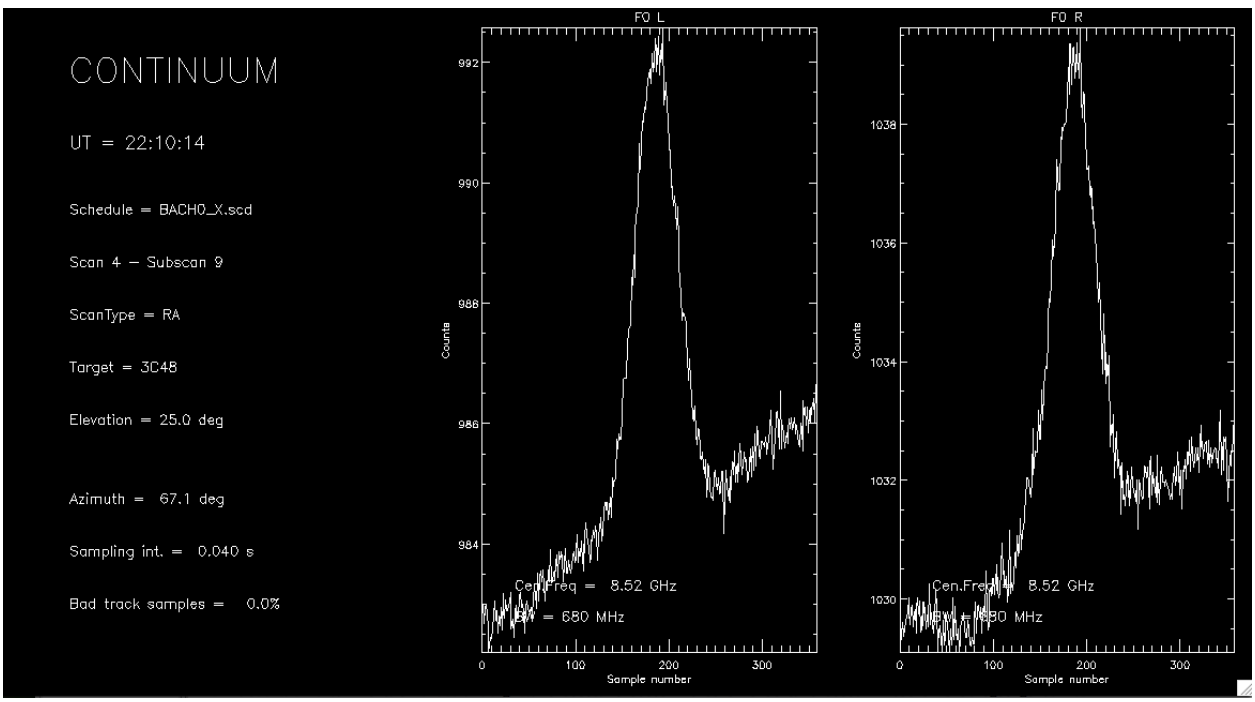
The last completed FITS file (i.e. the most recent subscan carried out, either being a TP or a spectral one) will be displayed.

Note: The program automatically identifies the recorded sections and plots them all. If you are using the multi-feed receiver with the aim of actually exploiting only the central feed, adjust your schedule so that only sections 0 and 1 are enabled, otherwise all the sections will be recorded and displayed.

The plots show the proper file content, according to its continuum or spectral nature. Read all the program options using:

```
IDL> fitslook, /help
```

Note: By default, fitslook displays raw data (in arbitrary counts). Data streams in antenna temperature are available - and selectable - only for the TPB and under the condition that a Tsys has been correctly acquired prior to the execution of the scan.



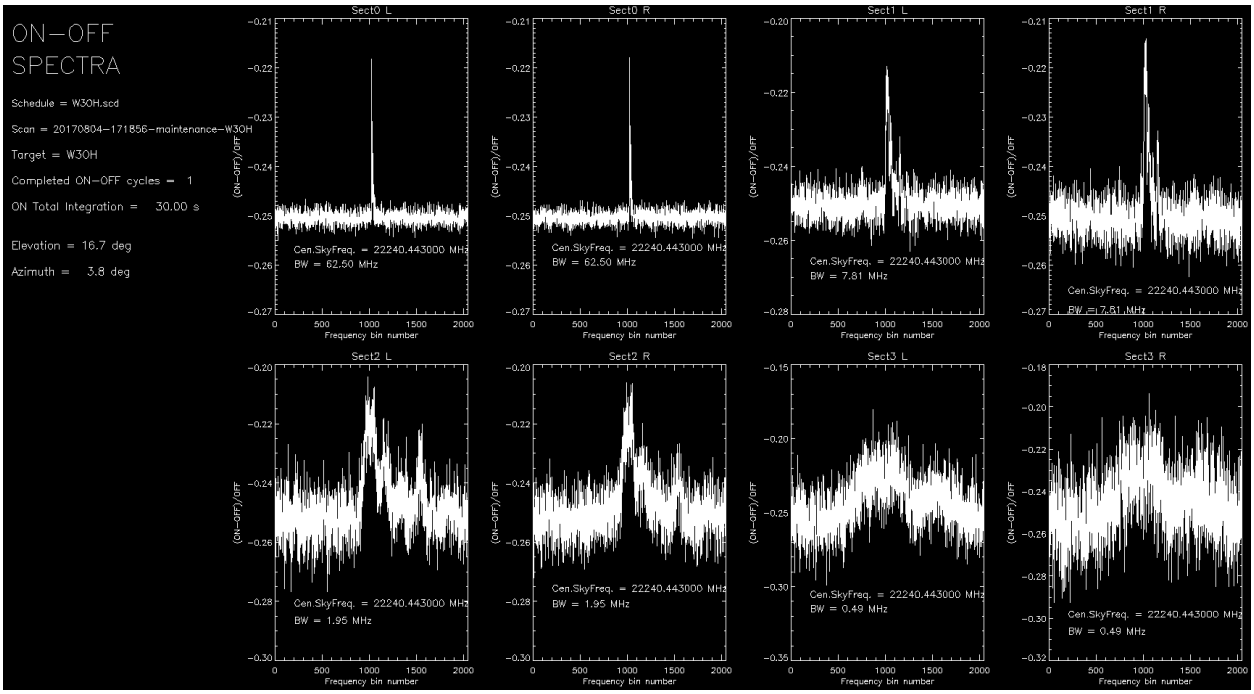
When performing ON-OFF spectral observations, users might also choose a different quick-look, displaying the (ON-OFF)/OFF result obtained by integrating all the subscans present in the last completed scan. Users must specify the duty cycle, e.g. '3:3:1' if each scan is composed by 3 on-scans, 3 off-scans and 1 cal-scan. Zero values are accepted. To invoke this program:

```
IDL> .r onoff
IDL> onoff, dutyc='[N_on]:[N_off]:[N_cal]'
```

If wanting to display the spectra vs the topocentric radial velocity, use:

```
IDL> onoff, dutyc='[N_on]:[N_off]:[N_cal]', xtype='vel'
```

Click on the above images to open the zoomable version.



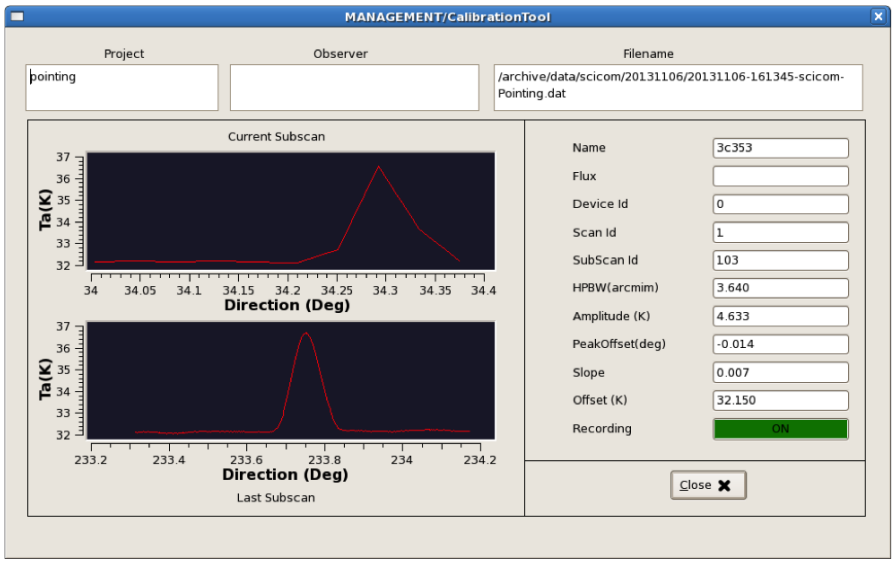
Please report any problem/request about these tools, which are constantly under development.

If writer is MANAGEMENT/Point or MANAGEMENT/CalibrationTool

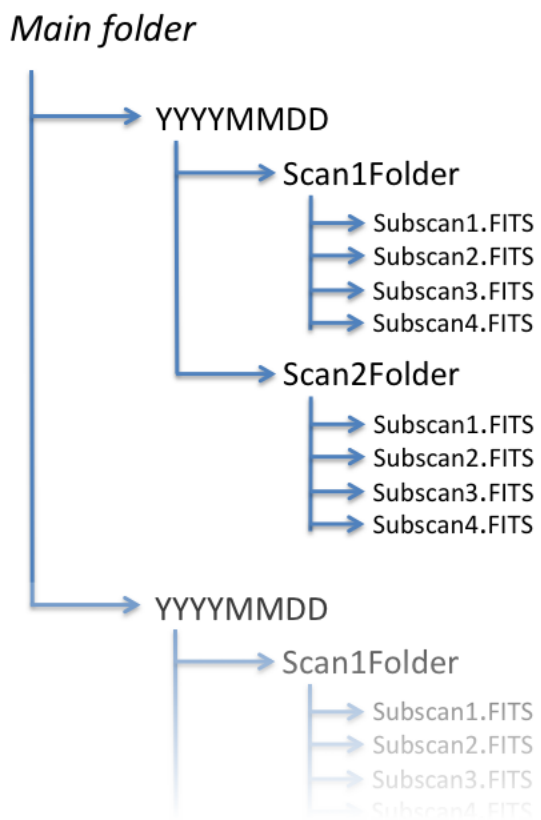
When data are acquired – both manually or through a schedule – using the Point or CalibrationTool writers, the quick-look must be performed using the CalibrationToolClient. Open a terminal on escsRemote and use the command:

```
$ calibrationtoolclient  [componentName]
```

where componentName is either MANAGEMENT/Point or MANAGEMENT/CalibrationTool. A graphic window will appear. Its content is given in the following figure.



Note: In this client, the subscan currently being acquired is shown *in real-time* (upper plot), even if in a low-res version. Under this display, the last completed subscan - in its full sampling - is shown.



Retrieving the data

Open a terminal on escsConsole. Your data folder is:

```
~/data
```

Inside that folder, subfolders named according to the date (YYYYMMDD) will be automatically created during acquisitions. Taking into account the choice of the FITS format, the only one so far fully tested, the date-dependent folder contains a subfolder for every scan, inside which there are the FITS files (one for each subscan).

Scan folder names are composed as:

YYYMMDD-HHMMSS-Project-Suffix

where

- **HHMMSS** is the UT time associated to the first sample of the acquisition.
- **Project** is the code/name specified using the code*project*= command,] or when starting a schedule
- **Suffix** is a user-defined string retrieved from the schedule files. Though no control can be applied on the choice/check of this string, the agreement is that it should coincide with the target name.

FITS files, each corresponding to a subscan, are composed as:

YYYYMMDD-HHMMSS-Project-Suffix_Scan#_Subscan#.fits

Data can be copied from *escsConsole* to your laptop using 'ssh' or 'rsync –e ssh' commands.

Note: When recording manually-acquired data in FITS format, the output files are stored in a peculiar path which is not accessible to users. This implies that they also cannot be shown by the quick-look procedure. This temporary situation, which requires the involvement of a system administrator in order to retrieve the data, will be corrected as soon as possible.

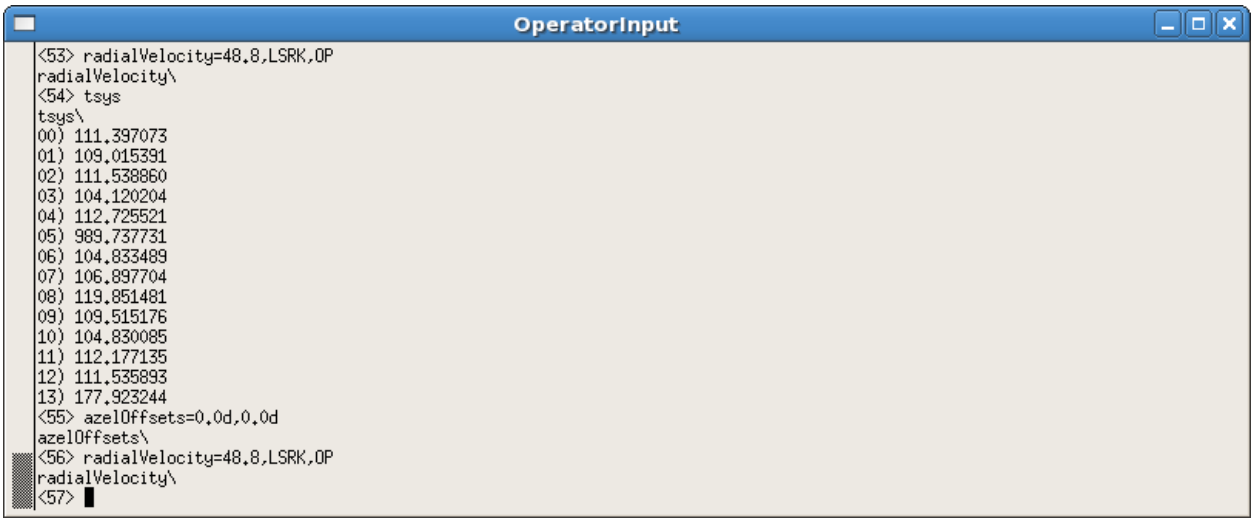
You can retrieve your logfiles here:

```
~/logs
```

Its reply will let you know where the logfiles are located. A subfolder named */WindLog* is devoted to the meteoClient output, if this client is used.

Appendix A - Monitor panels full description

operatorInput



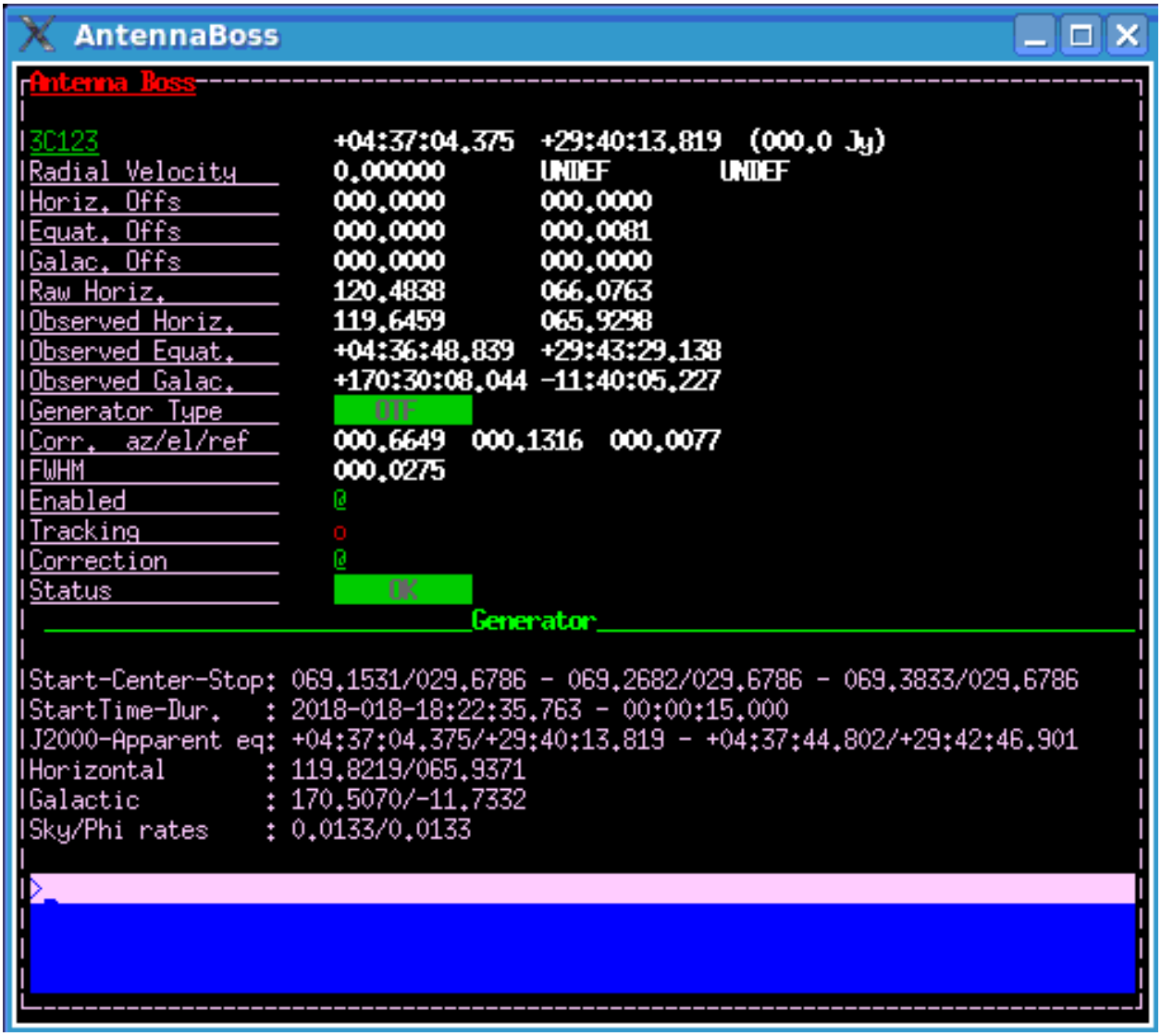
This is the **input console** where users write ESCS commands. The prompt is just a sequential number enclosed in <>. If a command is properly read, the system replies repeating the command itself, followed by the operation results (if they are foreseen). Otherwise, an error message appears. To close the window, type `exit`.

AntennaBoss

This monitor shows the target info, indicating the commanded and actual positions pointed by the antenna. It also gives a feedback on the pointing accuracy and on the overall antenna status.

Parameters:

- **Target**: label (as extracted from schedule or commanded), coordinates (RAJ2000.0, DecJ2000.0), estimated flux density (Jy, if available).
- **Radial Velocity**: (as extracted from schedule or commanded), value (km/s unless it is a redshift), velocity reference frame, velocity definition.
- **Horiz. Offs**: Horizontal (Az-El) offsets as read from schedule, degrees.
- **Equat. Offs**: Equatorial (RA-Dec) offsets as read from schedule, degrees.


AntennaBoss
Antenna Boss
3C123 +04:37:04.375 +29:40:13.819 (000.0 Jy)
Radial Velocity 0.000000 UNDEF UNDEF
Horiz. Offs 000.0000 000.0000
Equat. Offs 000.0000 000.0081
Galac. Offs 000.0000 000.0000
Raw Horiz. 120.4838 066.0763
Observed Horiz. 119.6459 065.9298
Observed Equat. +04:36:48.839 +29:43:29.138
Observed Galac. +170:30:08.044 -11:40:05.227
Generator Type OTF
Corr. az/el/ref 000.6649 000.1316 000.0077
FWHM 000.0275
Enabled @
Tracking o
Correction @
Status OK
Generator
Start-Center-Stop: 069.1531/029.6786 - 069.2682/029.6786 - 069.3833/029.6786
StartTime-Dur. : 2018-018-18:22:35.763 - 00:00:15.000
J2000-Apparent eq: +04:37:04.375/+29:40:13.819 - +04:37:44.802/+29:42:46.901
Horizontal : 119.8219/065.9371
Galactic : 170.5070/-11.7332
Sky/Phi rates : 0.0133/0.0133

- **Galac. Offs**: Galactic (l-b) offsets as read from schedule, degrees.
- **Raw Horiz.**: commanded Az-El, including pointing model, refraction, etc...
- **Observed Horiz.**: Az-El coordinates read from the mount encoders and cleaned from the pointing model and refraction contributions. Because of this, observed coordinates will differ from the raw ones.
- **Observed Equat.**: RA-Dec J2000.0 coordinates, converted from the observed horizontal.
- **Observed Galac.**: l-b Galactic coordinates, converted from the observed horizontal.
- **Generator Type**: which component is in charge of the generation of the coordinates. It can be `NONE` (which is the condition at startup), `OTF`, `MOON` or `SIDEREAL`.
- **Corr. az/el/ref**: azimuth and elevation corrections (degrees) applied by the pointing model, plus the refraction model contribution– which is an additional correction to elevation.
- **FWHM**: Full Width Half Maximum (corresponding to HPBW), degrees.
- **Enabled**: a green `@` indicates that the antenna is correctly receiving commands; a red `o` means the communication is disabled.
- **Tracking**: it indicates whether the pointing error exceeds 0.1*FWHM or not. A green `@` corresponds to error < 0.1*FWMH. It should turn to a red **o** only when the antenna is slewing between scans on the same source, or when slewing to/from a new source.
- **Correction**: application of the above horizontal coordinates corrections. If disabled (red circle), all corrections are zeroed.
- **Status**: `OK`, `WARNING` or `ERROR`. "Warning" needs investigation but usually does not stop the ongoing activity (it also appears at startup, before the setup commands), "Error" generally appears if something stops the observations.
- **Generator**: under this line all the subscan setup parameters appear when it is commanded. See the two cases here below.

In particular, for OTF observations:

- **Start-Center-Stop** – coordinates in the subscan frame of the start, center and stop positions (degrees)
- **StartTime-Dur.** – subscan start UT `yyyy-doy-hh:mm:ss.s` and duration `hh:mm:ss.s`
- **J2000-Apparent eq** – during the subscan, current RA-Dec position pointed by the antenna, both in the J2000 epoch and precessed to the date. Format is `hh:mm:ss.s-°:':".”`
- **Horizontal** - during the subscan, current Az-El position pointed by the antenna (degrees)
- **Galactic** - during the subscan, current l-b position pointed by the antenna (degrees)
- **Sky/Phi rate** – actual subscan speed on sky along scanning direction (°/s) . The 'Phi rate' value refers to great circle arcs, if it is the scanning geometry, otherwise it copies the first speed value.

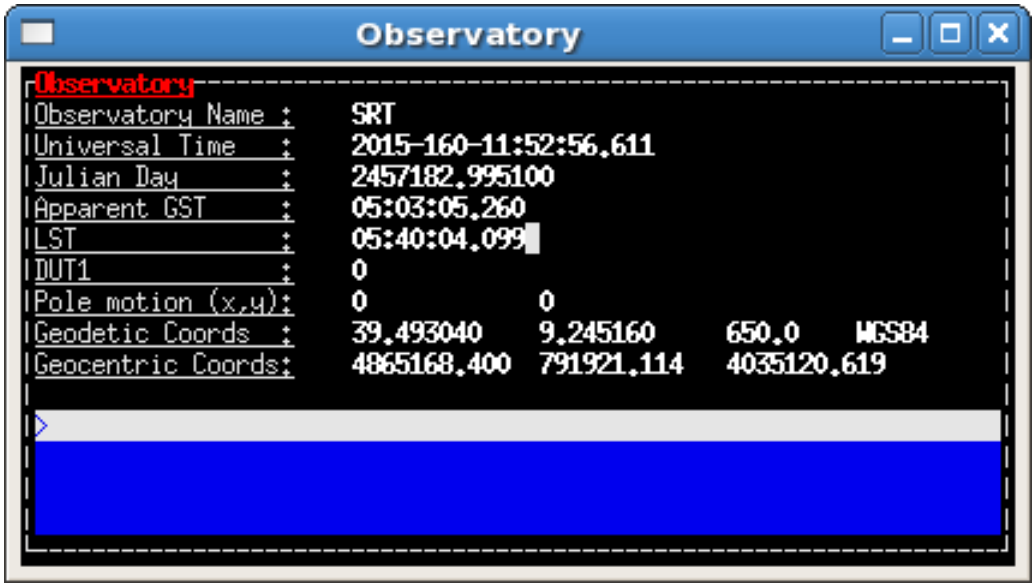
For **SIDEREAL** scans, instead, this is what appears:

- **Source name** – target name
- **Catalog Eq.** – target FK5 coordinates and info (if available): RA `hh:mm:ss.s`, Dec `°:':".”`, Epoch, RA proper motion (milliarcsec per julian year), Dec proper motion (milliarcsec per julian year), parallax (milliarcsec), radial velocity (km/s)
- **Catalog Gal.** – target Galactic longitude and latitude (degrees) converted from the above Equatorial or provided by user/catalogue
- **Apparent Eq.** – target current apparent Equatorial coordinates: RA `hh:mm:ss.s`, Dec `°:':".”`, Epoch
- **Galactic** – current l-b position pointed by the antenna (degrees)

- **Horizontal** –current Az-El position pointed by the antenna (degrees)

To close the monitor, type `exit` in its prompt (the grey line).

Observatory



It is devoted to the station coordinates and times:

- **Observatory Name**: SRT, Medicina or Noto
- **Universal Time**: *YYYY-DOY-HH:MM:SS.SSS*
- **Julian Day**: *d.ddd*
- **Apparent GST**: Greenwich Sidereal Time *HH:MM:SS.SSS*
- **LST**: Local Sidereal Time *HH:MM:SS.SSS*
- **DUT1**: difference between UT1 and UT (s), if applied.
- **Pole motion**: celestial pole offset w.r.t. a reference position (details are note provided here), measured in μarcsec on a tangential projection.
- **Geodetic Coords**: updated Latitude and Longitude (degrees) and Altitude (m) for the telescope, plus the geodetic model code.
- **Geocentric Coords**: geocentric cartesian coordinates (m) for the telescope.

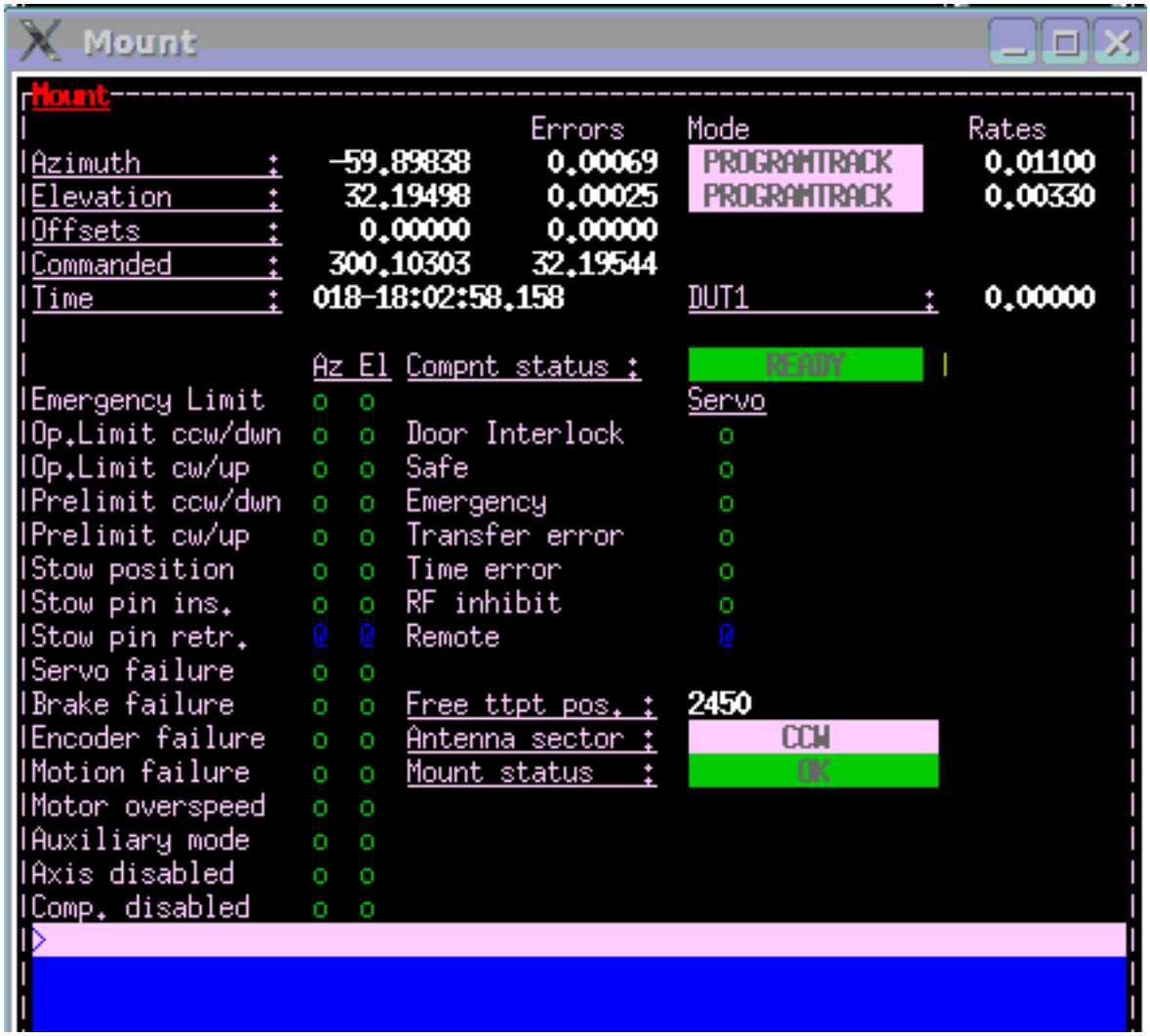
To close the monitor, type `exit` in its prompt (the grey line).

Mount

This is a quite complex frame, allowing the technical staff to monitor all the various parts of the antenna mount.

Observers must focus only on a subset of the displayed parameters and flags. We thus describe the main features only.

The top left section gives the online readouts for the Azimuth and Elevation axes, compared to the commanded positions. Line by line:


Mount
Mount
Errors
Mode
Rates
Azimuth : -59.89838 0.00069 PROGRAMTRACK 0.01100
Elevation : 32.19498 0.00025 PROGRAMTRACK 0.00330
Offsets : 0.00000 0.00000
Commanded : 300.10303 32.19544
Time : 018-18:02:58.158
DUT1 : 0.00000
Az El Compnt status : READY
Emergency Limit o o Servo
Op.Limit ccw/dwn o o Door Interlock o
Op.Limit cw/up o o Safe o
Prelimit ccw/dwn o o Emergency o
Prelimit cw/up o o Transfer error o
Stow position o o Time error o
Stow pin ins. o o RF inhibit o
Stow pin retr. @ @ Remote @
Servo failure o o
Brake failure o o Free ttpt pos. : 2450
Encoder failure o o Antenna sector : CCW
Motion failure o o Mount status : OK
Motor overspeed o o
Auxiliary mode o o
Axis disabled o o
Comp. disabled o o

- **Azimuth**: current azimuth position (in the -90°/450° wrapping range), rate (°/s) and commanded offset (degrees). Then the measured position error and tracking error (both in degrees) are given. Finally, the axis active mode is displayed: it can be *PRESET* (fixed Az-El pointing only), *PROGRAMTRACK* (for tracking, OTF and schedules), STOP (if axes brakes are on), *UNKNOWN* (indicating a non-configured status).
- **Cmd Az.**: commanded azimuth position (in the -90°/450° wrapping range), rate (°/s) and offset (°). This offset comes from metrology devices, it is not related to user-defined offsets or subscan-defined offsets.
- **Elevation**: current elevation position (degrees), rate (°/s) and commanded offset (degrees). Then the measured position error and tracking error (both in degrees) are given. Finally, the axis active mode is displayed: it can be *PRESET* (fixed Az-El pointing), *PROGRAMTRACK* (for OTF and schedules), *STOP* (if axes brakes are on), *UNKNOWN* (usually indicating an error condition).
- **Cmd El.**: commanded elevation position (degrees) and rate (°/s).
- **Wrap**: readouts from the cable wrap. The status flag on the right can be *CW''or ''CCW*.

The general condition of the mount is summed up by one keyword, in the central part:

- **Mount status**: it can be *OK*, *WARNING*, *FAILURE* or *BUSY*, the latter is associated to operations which take a lot of time to complete (as the stow-unstow procedure). When the mount is BUSY it will not accept further commands until the ongoing operation is completed.

The rest of the panel lists several **flags and status labels**. For average users, the only points to be taken into account are:

- in standard observing conditions, when a schedule runs, all flags should be green *o*;
- in case of warnings, flags turn to yellow *o*;
- errors correspond to red *o*;
- when a "failure" keyword turns steadily to a red *o*, or a permanent failure status appears, immediately call for technical assistance.

To close the monitor, type *exit* in its prompt (the grey line).

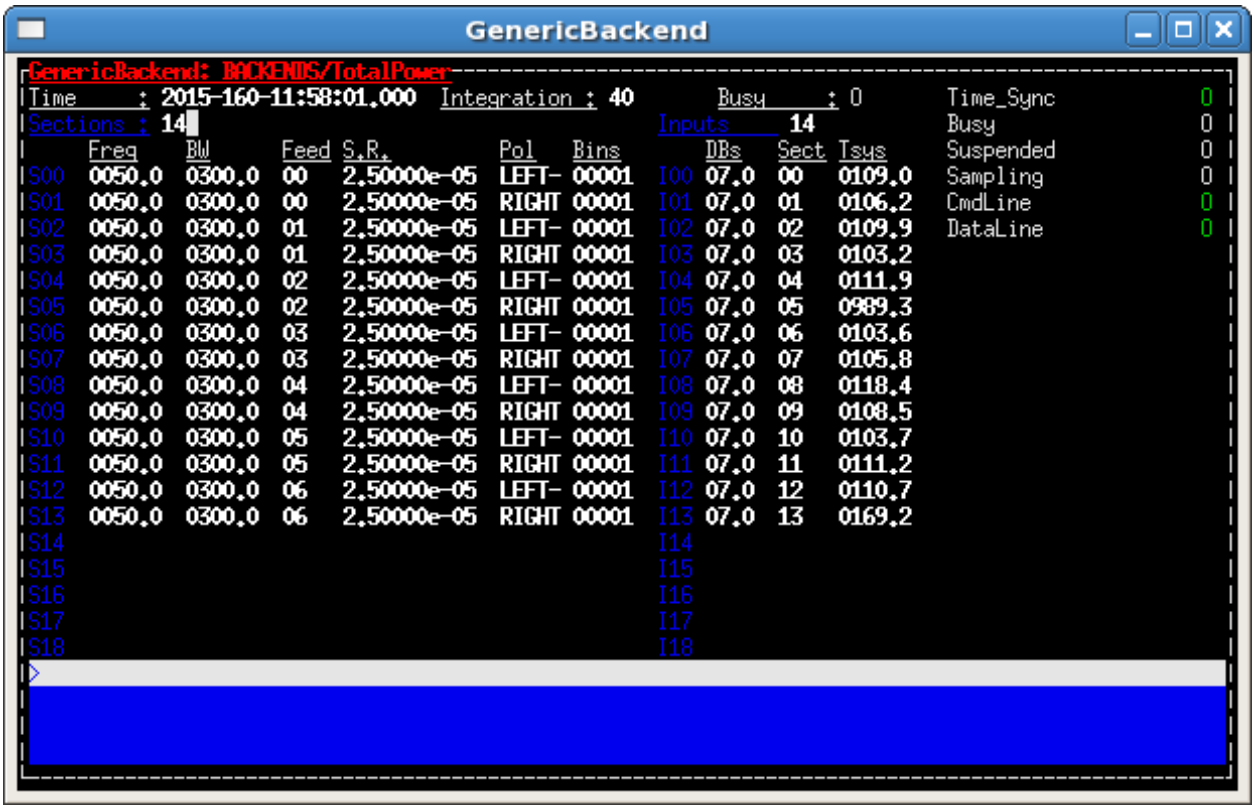
GenericBackend

The panel shows one row for every section.

- **Freq**: value (MHz) to be added to the LO frequency in order to obtain the observed frequency at the beginning of the band
- **BW**: bandwidth (MHz)
- **Feed**: number of the receiver feed connected to this section
- **S.R.**: sampling rate (MHz)
- **Pol**: polarisation (Left or Right)
- **Bins**: number of frequency bins (1 for total power)
- **DBs**: attenuation (dB) applied to the section
- **Sect**: section number
- **Tsys**: the last measured Tsys (K)

Also some backend-dependent status flags are present, in the top right area. When observing with the TPB, look for:

- **Time_Sync**: if it frequently or steadily turns red call for assistance (the backend time is not synchronized)
- **Busy**: when schedules are running, it must turn yellow. If it does not, the backend is not acquiring.



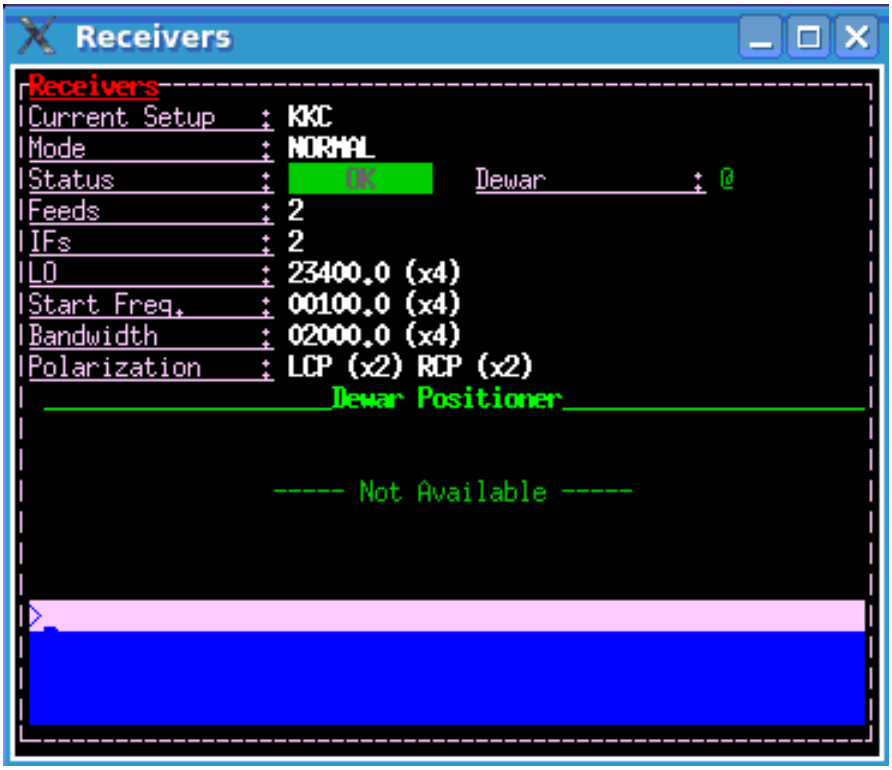
To close the monitor, type `exit` in its prompt (the grey line).

ReceiversBoss

This monitor summarizes the frontend setup. The bottom part is devoted to the derotator (dewar positioner), when available.

- **Current setup**: receiver code
- **Mode**: *NORMAL*, *SINGLEDISH*, *VLBI...* (other codes to come)
- **Status**: *OK*, *WARNING* or *FAILURE*
- **Feeds**: number of feeds (1 fore single-feed receivers, 2 for the K-band DF)
- **LO**: Local Oscillator frequency (MHz)
- **Start Freq.**: step (MHz) to be added to the LO frequency in order to obtain the actually observed frequency at the beginning of the band
- **Bandwidth**: actually observed bandwidth (MHz)
- **Polarization**: *LCP*, *RCP*

To close the monitor, type `exit` in its prompt (the grey line).

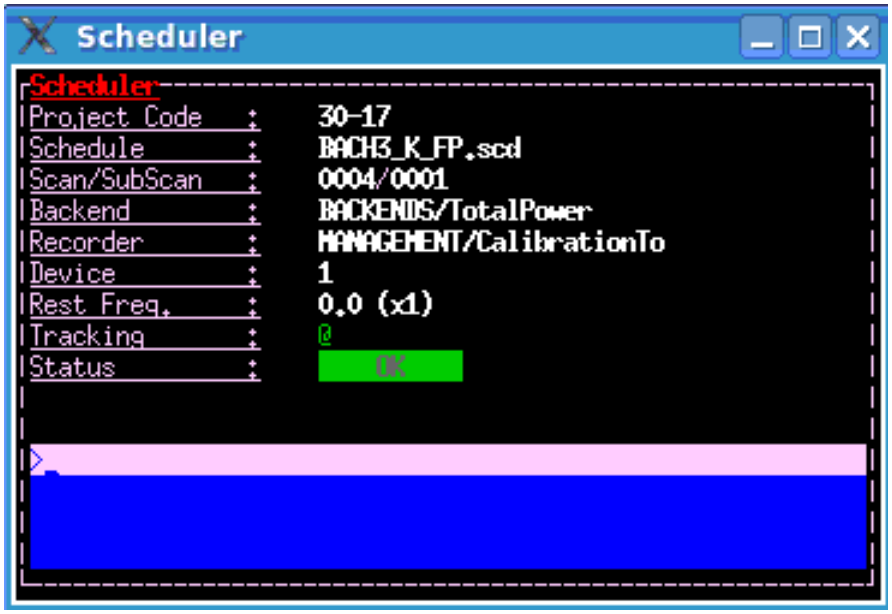


Scheduler

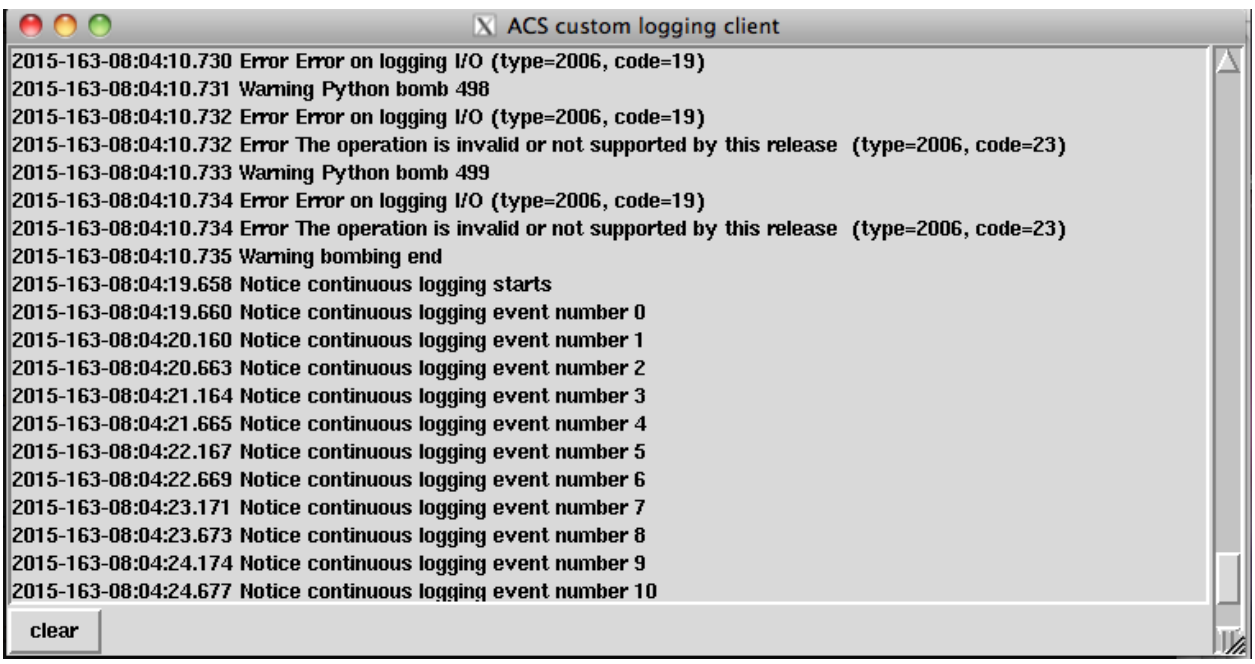
This monitor shows details on the selected data acquisition devices and on the running schedule, if any.

- **Project code**: project name, as specified at the launch of the schedule (or as input using the *project*= command)
- **Schedule**: name of the active schedule
- **Scan/Subscan**: scan number and subscan number, relative to the ongoing acquisition
- **Backend**: name of the selected backend, as listed in the schedule
- **Recorder**: name of the selected data recording device, as listed in the schedule
- **Device**: number of the currently selected device (see device= command)
- **Rest Freq.**: rest frequency of the line(s) being observed (MHz). The number in brackets indicates how many IFs the given rest frequency refers to.
- **Tracking**: tracking status of the telescope, including antenna, active surface, minor servo. A red *o* means the telescope is not yet in its tracking route (or that it is not tracking properly), a green light means the observation is ongoing and the tracking is accurate within 0.1*HPBW
- **Status**: flag summarising the telescope status. *OK* is self-explanatory, *WARNING* in principle indicates that the acquisition can go on even if a problem has been reported, *ERROR* signals that a major problem was detected and the observation cannot be performed.

To close the monitor, type *exit* in its prompt (the grey line).



Logging Display



Here all the log messages will appear. New messages are shown on top of the previous ones. To close the window, click on its dedicated icon.

Note: All the monitors can be closed at a time using the command (in a terminal on escsRemote):

```
$ escsConsole --stop
```

Appendix B - Complete command list

Note:

COMMAND TEMPORISATION All commands can be temporised (i.e. associated to a starting time) adding a proper **suffix**. There are two possibilities:

- **absolute temporisation** (the operation will be performed at the indicated time): `@DOY-HH:MM:SS`, where DOY is the Day-Of-Year (1-366) and HH:MM:SS is the UT time;
- **iterative temporisation** (the operation is performed now, then periodically according to the indicated time interval): `@!DAYS-HH:MM:SS`, where DAYS is the number of days and HH:MM:SS is hours, minutes, seconds.

Commands (temporised or not) can be used also in the init/pre-scan/post-scan procedures inside schedules. *Observers* are in charge of considering *if* and *when* the use of a certain command makes sense in their schedule, according to their specific needs and goals: this is something that no schedule parser can check!

Here follow all the commands exploitable in ESCS:

`> antennaPark`
sends the antenna to stow position

`> antennaReset`
resets the alarm conditions

`> antennaSetup=[code]`
(`code` can be `CCB`, `KKG`, ...) unstows the antenna, sets it to tracking mode and configures the pointing model according to the specified receiver. It does NOT perform the receiver and backend setup

`> antennaStop`
stops the antenna. Activities can start again only commanding a mode change as `antennaTrack` (which does not affect the overall setup) or a new setup

`> antennaTrack`
sets the antenna to PROGRAMTRACK mode. It does not change the pointing model or any receiver setup

`> azelOffsets=[double]d,[double]d`
sets the Az-El offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source Elevation.

Example: `> azelOffsets=-0.05d,0.05d`

`> calOn`
switches the calibration mark on

`> calOff`
switches the calibration mark off

`> chooseBackend=[string]`
selects the backend; *string* can be `BACKENDS/TotalPower` or `BACKENDS/XBackends`

`> chooseRecorder=[string]`
selects the backend; *string* can be `MANAGEMENT/FitsZilla`, `MANAGEMENT/CalibrationTool` or `MANAGEMENT/Point`

`> crossScan=[scanFrame],[span],[duration]`
performs a cross-scan on the previously selected target (indicated using the `track` or `sidereal` commands), along the *scanFrame* (`EQ`, `HOR` or `GAL`), spanning *span* degrees in *duration* seconds

> *device=[sect]*
computes the beamsize, taking into account the present receiver and backend configurations relative to section sect

> *flush=[N]*
deletes the N-th element in the queue of temporised commands

> *flushAll*
deletes all the queue of the temporised commands

> *fTrack=[dev]*
It collects all the required data from the antenna, the back-end and the front-end, plus the information provided by the user (see the `radialVelocity` and `restFrequency` commands), then it tunes the telescope devices in order to centre the line(s) in each section bandwidth. The command lets the user select which device [dev] is asked to perform the tuning:

- **LO**: only the front-end local oscillator is moved
- **ALL**: the back-end performa a sub-tuning in the various sections

> *getAttenuations*
reads the attenuation values (dB) currently configured for the active sections, and lists them according to increasing section number

> *getTpi*
reads the signal intensity (raw counts) for the active sections, and lists them according to increasing section number

> *goOff=[frame],[beams]*
slews the antenna to an offset position, wrt a previously commanded target, along the longitude axis of the indicated coordinate frame (`EQ`, `HOR` or `GAL`). The user provides the offset value expressed in beamsizes. If the frame is HOR and target lies beyond the Elevation cutoff limits, the offset is applied in Elevation.

> *goTo=[double]d,[double]d*
sends the antenna, while in TRACKING mode, to the specified Az-El position.

Example: `goTo=180d,45d`

Arguments are always rounded in the range 0-360 and 0-90 for azimuth and elevation respectively (in any case the ranges are limited to mechanical contraints). The jolly character is valid and is considered as: keep the present value. The differences from the `preset` command are:

- once the antenna reaches the destination, the system will acknowledge the "on source" status;
- the pointing corrections (pointing model and refraction) are applied. In case they are not required they must be turned off explicitly.

> *haltSchedule*
completes the current scan and then stops the schedule

> *initialize=[code]*
(`code` can be `CCC`, `XXP`, `KKC` , . . .) configures the backend using the default parameters relative to the selected receiver. It does *not* act on the receiver, pointing model or antenna mount mode.

> *integration=[double]*
sets the backend integration time (ms)

> *log=[filename]*
defines a custom name for the logfile (do not specify the extension)

> *lonlatOffsets=[double]d,[double]d*
sets the Galactic b-l offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source latitude.

Example: `> lonlatOffsets=2.0d,-1.0d`

`> moon`
points the antenna to the present coordinates of the center of the Moon

`> preset=[double]d,[double]d`
sends the antenna, if in PRESET mode, to the specified Az-El position, without applying any pointing correction. This is useful when needing to point to a position next to the zenith. Beware: the antenna will reach the destination but no "on source" flag will be raised.

Example: `> preset=180d,45d`

`> project=[code]`
lets the system know which project is observing (the code/name must correspond to the one provided by the TAC). This code/name is then considered as default when launching schedules: the system will search for them in a folder named "project/schedules". This code/name also forms part of the output FITS filename. Notice that the PROJECT keyword indicated inside the schedule, which is then written in the "Project Name" keyword in the FITS main header, is a free string and might differ from the project official name.

`> radecOffsets=[double]d,[double]d`
sets the RA-Dec offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source Declination.

Example: `> radecOffsets=1.0d,0.0d`

`> radialVelocity=[vrad],[vref],[vdef]`

- [vrad] (radial velocity) is in km/sec if vdef is *not* Z
- [vref] (reference frame) can be one of the following:
  - **BARY**: Solar System BARYCENTRE
  - **LSRK**: Kinematic Local Standard of Rest
  - **LSRD**: Dynamical Local Standard of Rest
  - **LGRP**: Local Group
  - **GALCEN**: Galactic Centre
  - **TOPOCEN**: TOPOCENTRIC (observer's frame)
- [vdef] (velocity definition) can either be:
  - **RD**: Radio Definition
  - **OP**: Optical Definition
  - **Z**: stands for Redshift

The specified velocity parameters are valid until a new target is commanded. The `radialVelocity` command overrides any other velocity value that might have been differently expressed

`> receiversMode=[code]`
configures the working mode of the receiver, according to its peculiar characteristics

`> receiversSetup=[code]`
(`code` can be `CCC`, `XXP`, ...) configures the receiver using the default parameters. It does *not* act on the backend, pointing model or antenna mount mode

`> restFrequency=[freq1];...;[freqN]`
[freq] is given in MHz and is a multiple argument: it can list a different value for each of the N sections - as long as XARCOS is the backend in use(not all the backends allow this sub-tuning). Specifying a single value

assigns the rest frequency to *all* the sections. The specified values will hold until different ones are commanded, or until a new general *setup* command is entered.

> *setAttenuation=[sect],[att]*
sets to *att* (dB) the attenuator of section *sect*

> *setLO=[freq]*
Local Oscillator frequency, in MHz (one per IF, separated by ";", usually the values are identical) This LO frequency corresponds to: SkyFreq(@band start) – 100 MHz when using the TPB

> *setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]*
configures the backend section sect.

> *setupCCC (setupCCCL, setupXXP, etc...)*
unstows the antenna, sets it to tracking mode, selects the pointing model, and configures the receiver and the backend using default parameters. In practice, it is a shortcut corresponding to this sequence:

```
> antennaSetup=[code]
> receiversSetup=[receiverCode]
> initialize=[receiverCode]
> device=0
> calOff
```

> *sidereal=[sourcename],[RA],[Dec],[epoch],[sector]*
points to the supplied RA-Dec position and temporarily assigns the sourcename label to it. Epoch can be `1950`, `2000` or `-1`, the last one meaning that the provided coordinates are precessed to the observing epoch. The sector keyword forces the cable wrap sector, if needed: its value can be `CW`, `CCW` or `NEUTRAL`. The last option means the system will automatically choose the optimal alternative.

Example: `> sidereal=src12,319.256d,70.864d,2000,neutral`

> *skydip=[El1]d,[El2]d,[duration]*
performs an OTF acquisition at the current azimuth position, spanning in elevation from *El1* to *El2* (both expressed in degrees, with 'd' suffix), in *duration* time expressed as hh:mm:ss. A recorder must have previously been enabled in order to save the data, and the antenna must have received a pointing command as `track` or `goTo`

> *startSchedule=[project/][schedulename].scd,[N]*
runs schedule *schedulename*.scd (project is the ID of the observing project, it is optional if it has already been input through the `project` command), reading it from line *N*

> *stopSchedule*
immediately stops the running schedule, truncating the acquisition

> *ti*
lists all the active temporised commands

> *track=[sourcename]*
points the antenna, in sidereal tracking, to the specified source, which must be present in the local catalogue

> *tsys*
measures the system temperature (K) in the position the antenna is pointing to. It returns a list of values, one for each section in use. Intermediate steps and calculations are stored in the active logfile

> *wait=[double]*
sets a delay (in seconds) which is applied before the system reads/executes the next command

> *winkingMark=[integer]*
activates the winking (i.e. fast switching) calibration mark, whose period in milliseconds is given as an integer parameter. To switch the mark off, simply command a zero-ms period. This must **always** be done when the winking mark is not needed anymore.

Example: `> winkingMark=40` activates the mark so that is alternately stays ON for 40 ms and OFF for 40 ms.

While: `> winkingMark=0` switches the mark off

`> wx`

returns the current weather parameters: ground temperature (°C), relative humidity (%), atmospheric pressure (hPa), wind speed (km/h).

Appendix C - Output files

The system at present allows the user to write the output data, coming from the integrated backends, in two different formats: FITS and MBFITS. The former is a substantially **standard FITS**, carrying the data and a load of ancillary information inside several extensions. A FITS is produced for every subscan composing the observation.

MBFITS is a more complex format, not yet fully implemented and whose writer component is still under testing. Each MBFITS contains the whole scan, and subscans are hierarchically organized inside its structure.

Details on both formats are provided in this separate manual `pdf`

Appendix D - Source catalogue

Here follow the sources which are listed in the system catalogue. Their names, as presented here, can be used in schedules (SIDEREAL subscans, inside the LIS file) or in the command:

```
> track=[sourcename]
```

Along with their J2000 Equatorial coordinates, the catalogue records usually contain further information such as apparent size, proper motion, radial velocity, coefficients of the *Ott et al.* polynomials describing the spectra, etc… ESCS uses these parameters to make computations during certain operations (e.g. when calibration tools are used – under development, they will be described in future releases of this manual):

- 3c48
- 3c84
- 3c123
- 3c147
- 3c161
- 3c218
- 3c227
- 3c249.1
- 3c286
- 3c295
- 3c309.1
- 3c348
- 3c353
- 3c380
- 3c391
- casa (i.e. Cas A)

- cyga (i.e. Cyg A)
- dr21
- ngc7027
- oria (i.e. Ori A)
- taua (i.e. Tau A)
- vira (i.e. Vir A)
- w3oh

Fault report

Users are kindly requested to report all faults and misbehaviors by writing to observing@ira.inaf.it

A ticketing system will be implemented as soon as possible.

1.1.2 Observing at Noto with ESCS

Release notes

Issue	Release Date	What's new
01	30/11/16	Issue 01 - first ESCS installation at Noto

Overview

What ESCS is

ESCS is the DISCOS control software flavour produced for the Medicina and Noto radio telescopes. It is a distributed system based on ACS (ALMA Common Software), commanding all the devices of the telescope and allowing the user to perform single-dish observations in the most common modes. As of today, the code actually implemented for the telescopes (i.e. excluding the huge ACS framework) amounts to > 550000 active lines.

This guide is meant to help the observer in the use of ESCS, without dealing with the "behind-the-curtains" complex details of the system. This release focuses on **single-dish continuum observations**, as the fully integrated backend available on-site is the analog total power one. Since both the hardware and software implementations are continuously updated, this manual will forcibly undergo frequent revisions.

Here follows a simple schematization of the observing system, helpful to visualize all the main devices ESCS deals with and the most important operations it performs. *Notice: the features not yet available in this release are shown in brackets*.

Glossary: terms and abbreviations

> this symbol precedes commands to be given in the operatorInput panel

$ this symbol precedes commands to be given in a system shell

Beam-parking the antenna points to a fixed Azimuth-Elevation position

Bin frequency channel

DF Dual-Feed

FWHM Full-Width Half-Maximum

HPBW Half-Power BeamWidth

MS Minor Servo system, which includes the subreflector

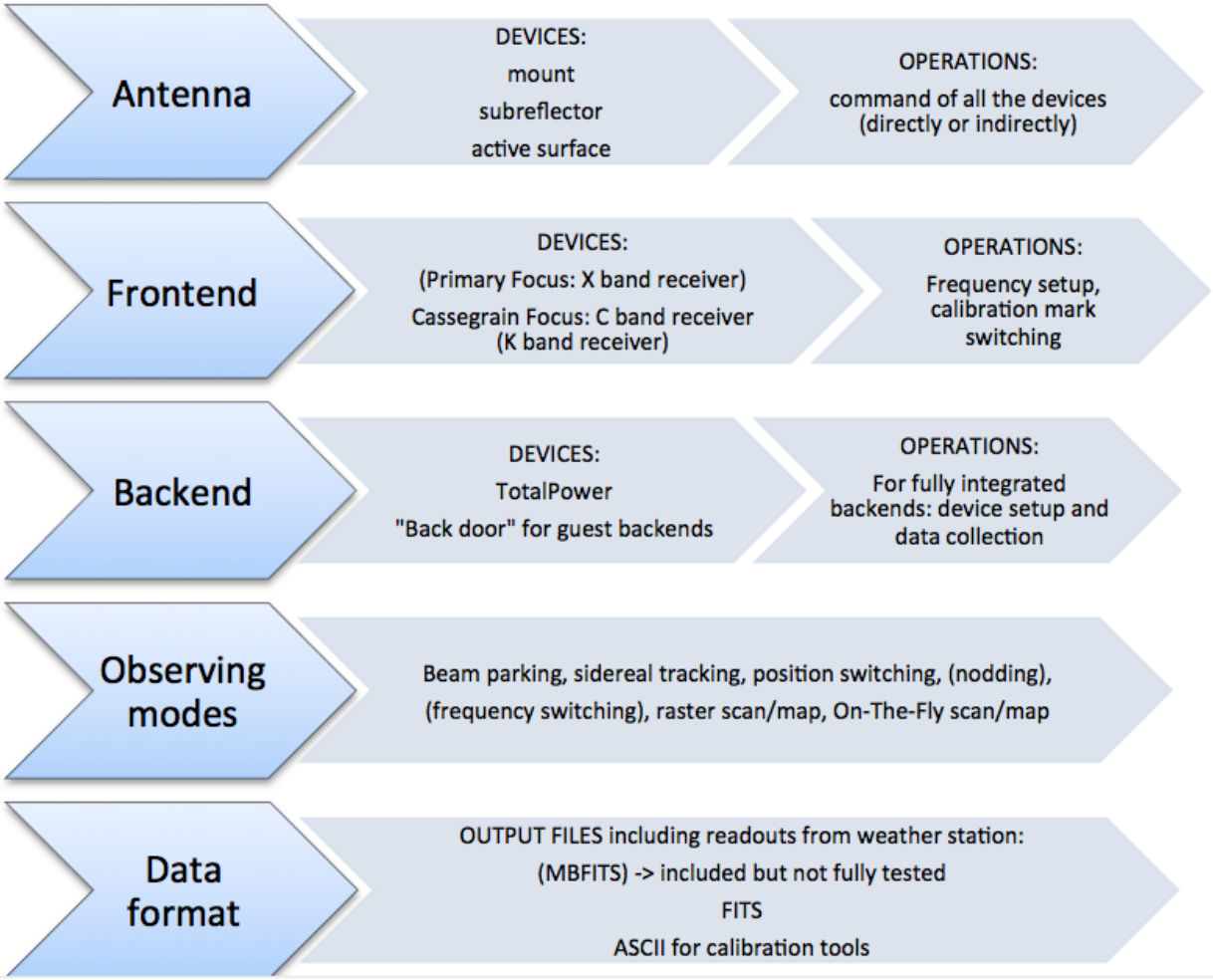
Nodding (aka "beam switching") it involves two or more feeds. The source is alternatively observed with each of the feeds, so that there is always one feed "on source"

OTF On-The-Fly acquisition. The antenna moves according to user-defined parameters, scanning the sky at constant speed. Data acquisition is active during the scan

Position switching (aka "on-off") the beam is alternatively pointed to the source (on) and to a properly defined offset position (off)

Preset mount mode allowing only beam-parking observations, with no pointing model applied

ProgramTrack mount mode allowing tracking, OTF/raster scans, beam-parking


Antenna
DEVICES:
mount
subreflector
active surface
OPERATIONS:
command of all the devices (directly or indirectly)
Frontend
DEVICES:
(Primary Focus: X band receiver)
Cassegrain Focus: C band receiver (K band receiver)
OPERATIONS:
Frequency setup, calibration mark switching
Backend
DEVICES:
TotalPower
"Back door" for guest backends
OPERATIONS:
For fully integrated backends: device setup and data collection
Observing modes
Beam parking, sidereal tracking, position switching, (nodding), (frequency switching), raster scan/map, On-The-Fly scan/map
Data format
OUTPUT FILES including readouts from weather station:
(MBFITS) -> included but not fully tested
FITS
ASCII for calibration tools

Raster acquisition is performed via discrete pointings (in tracking or beam-parking mode), planned to sample a certain path/area on the sky

Section acquisition stream (aka "logical channel"). E.g.: the 2 feeds of the K-band receiver used with the continuum backend produce 2 feeds x 2 polarisations = 4 sections

SF Single-Feed

Slewing motion of the antenna when it is going to the target position. Slewing is always performed at maximum speed, with no data acquisition taking place

Temporised commands user-input or schedule commands that are launched to a specific UT time by means of the "@" symbol

Time-tagged commands when commands are not temporised, the philosophy for their execution is asap (as soon as possible), but the system internally schedules them associating each operation to a properly computed time. This is particularly relevant for the positions commanded to the mount, so that the antenna motions take place along a fully controlled path (except in slewing)

TPB Total Power Backend

Tracking the antenna points to a target and, if it's a celestial target, follows its sidereal motion

Checklist for schedule-based observations

Notice that actions take place in three different "locations":

- **(1)** = action to be performed in a terminal on escs
- **(2)** = action to be performed in a terminal on euser
- **(op)** = command to be given in the *operatorInput* panel of ESCS

Login on both (1) and (2) *Locally*: login on (2) using the euser credentials, then Using the ESCS icon on the Desktop, connect to (1). *Remotely*: connect via VNC to (1) with the proper credentials, then use ssh to access (2).

Note: Remember that spaces within the command line content are **not** allowed!

Launch the monitors, if necessary (1):

```
$ escsConsole
```

Initial setup (op):

```
> antennaReset  (if resuming after the emergency stop button is released)
> setupCCC      (or other setup code)
```

Tune the local oscillator, if any (op):

```
> setLO=[freq]
--> e.g. setLO=4900 - start frequency of the observed band will depend on the backend
```

Point the antenna to a reference position (op):

```
> goTo=[Az]d,[El]d
--> e.g. goTo=180d,45d
```

Measure the cold sky signal level (op):

```
> getTpi
> setAttenuation=[sect],[att]
--> iteratively adjust attenuations until the level is about 850 counts
```

Get a Tsys (op):

```
> tsys
```

Pointing and focusing optimisation (op):

```
> track=name                    (choose a proper calibrator from source catalogue)
> chooseRecorder=MANAGEMENT/Point
--> the following command on (1):
        $ calibrationtoolclient MANAGEMENT/Point            (to display the plots)
> crossScan=HOR,0.5d,00:00:20   (set proper parameters according to your beamsize)
> azelOffsets=0d,0d             (only if wanting to reject the measured offsets!)
> focusScan=60,00:01:00
> clearServoOffsets             (only if wanting to reject the updated focus position!
↪)
```

WARNING:

- **System offsets, such as the ones measured with a Point acquisition, sum up to the ones indicated inside schedules ONLY if they are expressed in the same coordinate frame.** This means that, if you perform observations using EQ offsets, also the fine-pointing cross-scans must be carried out in the EQ frame. The same holds for HOR scans. If there is a frame mismatch, the system offsets are automatically rejected (bug under fixing).

Create a schedule (2):

```
Use basie, the schedule creator (see its own guide):
$ basie -c [configfile] [out_directory]
--> Move the schedule files to the proper folder
```

Parse the schedule (2):

```
$ scheduleChecker [schedulename].scd
```

Launch the schedule (op):

```
> startSchedule=[schedulename].scd,[N]
```

Data quick-look

- *Case A:* when using MANAGEMENT/Fitszilla, launch the quick-look (2):

  ```
  $ idl
  IDL> .r fitslook
  IDL> fitslook
  ```

- *Case B:* when using MANAGEMENT/Point, launch the quick-look (2):

  ```
  $ calibrationtoolclient MANAGEMENT/Point
  ```

- *Case C:* when using MANAGEMENT/CalibrationTool, launch the quick-look (1):

```
$ calibrationtoolclient MANAGEMENT/CalibrationTool
```

Stop the schedule (op):

```
> haltSchedule
```

Copy the data (2) —> Get the latest subfolders written in the main data folder

Stow the antenna (op):

```
> antennaPark
```

Stow the and active surface and minor servo (op):

```
> asPark
> servoPark
```

Close the monitors, if necessary (1):

```
$ escsConsole ---stop   (individual panels are closed typing "exit" in their command␣
↪lines)
```

ESCS startup

Logistics

Note: passwords are provided locally. Be sure to contact the local personnel before your session starts, in order to get the latest information.

Observations involve the use of two machines, called escs and euser, located in the 32-m dish control room:

On-site observations

The involved hardware is composed by two PCs with displays on the table facing the window, just next to the AS control panel. When the observers perform their acquisitions on-site, they exploit the euser machine only (the one on the right).

escs: input terminal and system monitors

Login to euser using the provided credentials. Then, click on the "CONNECTION TO ESCS" icon located on the Desktop. It will connect you to escs via a remote desktop (vinagre). On the escs remote desktop, you should find the input terminal and all the monitors already running. If, instead, you need to start them, click on the icon named "escsConsole". This opens 8 panels at once:

- operatorInput - terminal for command line input
- antennaBoss
- observatory
- mount
- genericBackend

ESCS
EUSER

- receiversBoss
- scheduler
- logging

Rearrange the panels on the desktop.

In case any of them does not automatically start, you can manually open them by means of individual command lines, to be given in a terminal (open it inside the escs remote desktop):

```
> operatorInput
> antennaBossTui
> observatoryTui
> mountTui
> genericBackendTui BACKENDS/TotalPower
> receiversBossTui
> schedulerTui
> loggingDisplay
```

All the antenna/receiver/backend setup procedures are performed via the operatorInput window, which is also used to start/stop the schedules.

The other panels are monitors used to display a vast amount of information, see *Appendix A - Monitor panels full description* and *Appendix B - Complete command list* for a comprehensive description of their content and a list of all the commands available for the operatorInput (they can be inserted in schedules as well).

euser: access to schedules, logs and data

Directly use euser for the data quicklook and retrieval (see dedicated sections), for the generation of schedules using the "basie" tool and for tools as FV. Once logged in, in your home you can find the following folders, whose names are self-explanatory:

```
~/data
~/schedules
~/logs
```

Note: users can generate subfolders according to their needs to store their schedules, but, in order to be executed, schedules must be places *exactly* in their ~/schedules folder.

(Remotely controlled observations): under development

It is possible to remotely perform the observations, exploiting a VNC connection to escs. Open a VNC session and connect to:

```
192.167.187.XXX:1  (i.e. port 5901)
```

You will be asked to insert a password (again, passwords are provided by the local staff). If you need to start the clients, open a terminal and command:

```
> escsClients
```

and follow the same instructions provided for observations carried out on site.

To access your data, schedules and logs, simply open a terminal on your computer and use:

```
> ssh -X euser@192.167.187.16
```

Hence you can launch IDL, use basie, retrieve your data, etc…

Initial setup

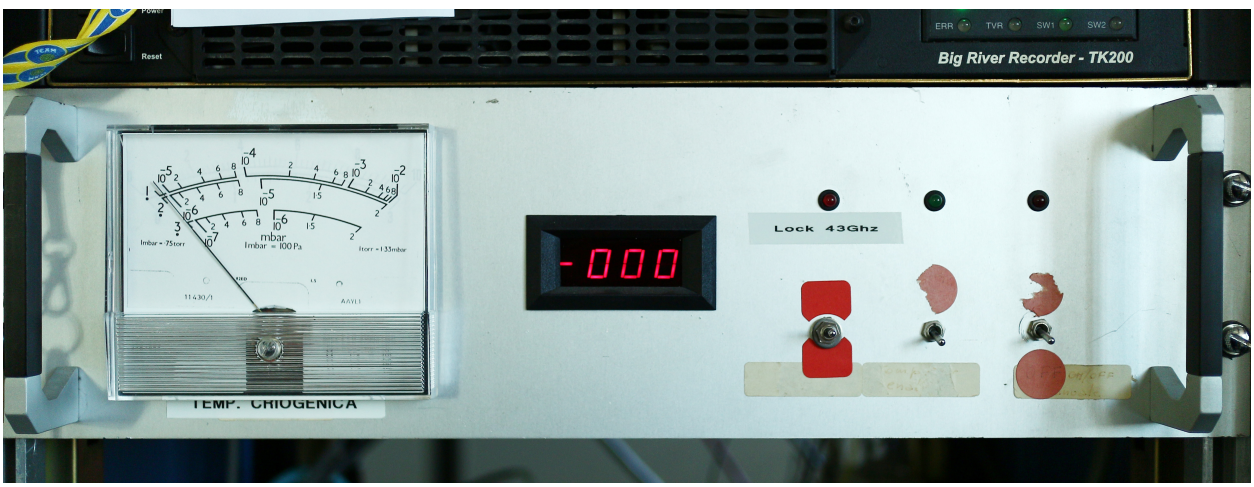
Antenna overall setup

When opening an ESCS observing session, it is necessary to perform a setup which includes the antenna unstow, the mount configuration in tracking mode, the minor servo setup. This is done by means of a unique command, which is specific for the wanted receiver, to be written in the **operatorInput**. The currently available choices are:

```
> setupSXP     for the SX-band receiver
> setupCCC     for the C-band receiver
> setupMMC     for the M-band receiver
> setupKKC     for the K-band receiver
> setupQQC     for the Q-band receiver
```

WARNING:

- It is necessary to manually select the observing focus using the hardware IF switch located in the control room. The figure below illustrates this rack.



The general setup command sets the antenna mount, the minor servos, the selected receiver and the default backend (TotalPower) according to **default parameters**. The antenna mode is set to **ProgramTrack** (allowing tracking and the execution of schedules), while the Local Oscillator frequency and the bandwidth are set as illustrated in the following table.

Receiver code	LO frequency (MHz)	Frontend IF band (MHz)	Backend IF band (MHz)	Observed bandwidth (MHz)	Observed band (MHz)
CCC	4600	4620 - 5020	50-730	400	4650-5020

Notice that the true observed band depends on the **intersection between the frontend IF band and the chosen backend filter**. The actual observed bandwidth and the band starting frequency are recorded in the output files (see *Appendix C - Output files*).

(1) For this setup, the FITS file will **NOT** report the actual values relative to frequency and bandwidth, as the present ESCS release does not take into account the different conversion taking place for this receiver.

Logfile

The default logfile is named **station.log**. If the user wants to change it:

```
> log=[logfilename]  (without extension)
```

Logfiles are stored in a dedicated folder (see Retrieving the data). **When schedules are run, a new logfile is automatically started**, and it is named after the schedule: [schedulename].log.

Antenna operations

Besides the overall telescope setup previously described, individual commands are available to change the antenna mount status and manage its steering/pointing:

> antennaReset
resets the antenna status after a failure, for example after the emergency stop button is released

> antennaUnstow
it only performs the unstow procedure

> antennaSetup=[code]
code: *PRIM* or *SEC*

it unstows the antenna (if it is stowed) then it sets the pointing model and the minor servo system according to the selected receiver. Mount is set to ProgramTrack mode

> servoSetup=[code]
code: CCC (MMC, KKC, QQC)

it sets the minor servo system only, without affecting other systems

> antennaTrack
it sets the mount to ProgramTrack mode, allowing the execution of sidereal tracking, on-off, OTF scans, etc. Even beam-parking acquisitions (i.e. on a fixed Az-El position) can be now performed in ProgramTrack mode by means of the goTo command

> goTo=[Az]d,[El]d
it points the antenna to fixed Az-El positions, yet the mount remains in ProgramTrack mode

e.g. *> goTo=180d,45d*

> track=[sourcename]
if the antenna is in ProgramTrack mode and the sourcename is known within the station catalogue (which includes the most commonly observed calibrators), it directly points to the source and tracks it

e.g. *> track=3c286*

> sidereal=[sourcename],[RA],[Dec],[epoch],[sector]
if the antenna mode is ProgramTrack, it points to the supplied RA-Dec position and temporarily assigns the sourcename label to it. Epoch can be *1950*, *2000* or *-1*, the last one meaning that the provided coordinates are precessed to the observing epoch. The sector keyword forces the cable wrap sector, if needed: its value can be *CW*, *CCW* or *neutral*. The last option means the system will automatically choose the optimal alternative

e.g. *> sidereal=src12,319.256d,70.864d,2000,neutral*

Note: COORDINATE FORMATS Whenever celestial coordinates (Equatorial, Horizontal or Galactic) are specified, the allowed formats are:

- **decimal degrees**, using a *d* suffix, for any coordinate → e.g. *30.00d*
- **sexagesimal degrees**, with no suffix, for any coordinate → *30:00:00*
- **hh:mm:ss**, with a *h* suffix, for longitudes only → *02:00:00h* (not accepted for offsets)

Back to the commands:

> goOff=[frame],[beams]
it slews the antenna to an offset position, wrt a previously commanded target, along the longitude axis of the indicated coordinate frame (*EQ*, *HOR* or *GAL*). The user provides the offset value expressed in beamsizes. If the frame is HOR and target lies beyond the Elevation cutoff limits, the offset is applied in Elevation.

e.g. *> goOff=HOR,5*

> azelOffsets=[azoff]d,[eloff]d
it sets user-defined offsets in the Horizontal frame (degrees only). The following example sets an azimuth offset to 0.5 degrees and the elevation offset to 0.3 degrees

e.g. *> azelOffsets=0.5d,0.3d*

> radecOffsets=[raoff]d,[decoff]d
it sets user-defined offsets in the Equatorial frame (degrees only). The following example sets the right ascension offset to 0.3 degrees and the elevation offset to 0.0 degrees

e.g. > radecOffsets=0.3d,0.0d

> lonlatOffsets=[lonoff]d,[latoff]d
it sets user-defined offsets in the Galactic frame (degrees only). The following example sets the galactic longitude offset to 0.1 degrees and the galactic latitude offset to 0.5 degrees

e.g. *> lonlatOffsets=0.1d,0.5d*

Note: OFFSETS: the above user-defined offsets are the overall antenna offsets and they are mutually exclusive! If the user commands the offsets several times in a row (in one or different frames) only the last one will be effective. **Offsets specified within schedules, at subscan level, sum up to these user-defined offsets if they match in reference frame. If there is a frame mismatch, the system offsets are automatically rejected (bug under fixing)**

On with the list:

> antennaStop
it stops the antenna motion, if any, and changes the mount mode to "Stop"

> antennaPark
it stows the antenna

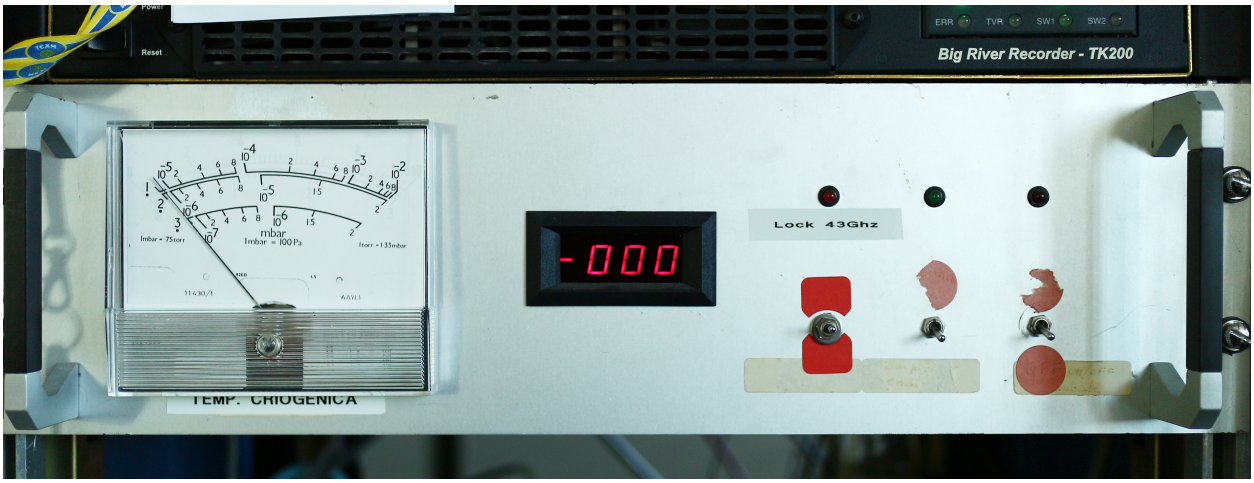
> asPark
it parks the active surface in the reference position

> servoPark
it stows the minor servo system (notice: after antennaPark, always give this command)

Frontend operations

WARNING:

- It is necessary to manually select the observing focus using the hardware IF switch located in the control room. The figure below illustrates this rack.



General rules

To change the frontend Local Oscillator frequency, use the following command:

```
> setLO=[freq1];[freq2];...;[freqN]
```

Notice the semicolon. Ideally, different values could be assigned to different IFs, thus tuning each section to a different sub-band. For the present hardware, though, this is not possible, so a single value must be specified:

e.g. `> setLO=4900`

Remember that the actually observed band begins at a frequency which is usually different from the LO one (see Initial setup)

The temperature of the calibration mark in use is recorded in the logfile whenever a Tsys is measured. It is also stored in the FITS/MBFITS output files. The calibration mark can be manually switched on and off respectively with:

```
> calOn
> calOff
```

If the user wants to perform the setup for the frontend only (without affecting the mount, the minor servo or the backend), the command is:

```
> receiversSetup=[code]    (CCC, MMC, KKC, QQC)
```

Backend operations

The backend to be used can be manually selected as follows:

```
> chooseBackend=BACKENDS/[bckname]
```

where *bckname* is the name of the backend. At present, the only available choice is:

- TotalPower

Total Power - focus selector

> **Warning:** The TotalPower backend works as a **"focus selector"**, sending the signal from the wanted receiver to any other backend. For this reason, it **must** be set up even when acquisitions take place using another backend. This is accomplished simply using the overall setup commands (such as `setupCCC`, etc...).

Bandwidth and sampling rate can be changed using:

```
> setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]
```

where:

- *[sect]* is an integer specifying the section number
- *[startFreq]* is the initial frequency for the section (not applicable for TPB)
- *[bw]* is a double for the bandwidth (MHz)
- *[feed]* is the number of the feed connected to the section (not applicable for TPB)
- *[mode]* is the polarisation mode (not applicable for TPB)
- *[sampleRate]* is also given in MHz
- *[bins]* is the number of frequency bins for the given section (not applicable for TPB)

To leave a parameter at its previously set value, or equivalently skip it when it is not applicable, use an asterisk. For the TPB, in particular, always use:

```
> setSection=[sect],*,[bw],*,*,[sampleRate],*
```

where *bw* can be chosen between two options (MHz):

- 300.0
- 730.0

Larger bandwidths (1250, 2000) are technically possible, but make no sense as long as the receivers' IF bandwidth is restricted to 500 MHz.

Consider that the above-listed bandwidths do not correspond to the true observed ones, for the reason discussed in Initial Setup. When accumulations of the data "dumps" are requiredbefore integrating them to the output file, it is possible to set the integration time as follows:

```
> integration=[N]
```

where *N* is given in milliseconds.

Each section is provided with an attenuator, to manage the signal intensity. To set the desired attenuation:

```
> setAttenuation=[sect],[att]
```

where *sect* is the section number and *att* can vary, for the TPB, form 0 dB to 15 dB, with a 1 dB step.

If a backend re-initialization is needed, use:

```
> initialize=[code]
```

(*code* can be *CCC*, *MMC*, *KKC*, *QQC*)

This command resets the backend only, setting it to the default values foreseen for the specified focus/receiver. After the frontend/backend configuration is changed, it is necessary to update the value for the beamsize (HPBW), which is computed as a function of the actually observed band and is used – during the observations – to evaluate the pointing accuracy. This is accomplished using the command:

```
> device=[sect]
```

which uploads to the system the parameters relative to section number [sect] (you can generally use 0, which exists for all the receivers, unless you need to observe only with a different feed).

Command-line measurements and acquisitions

Once the system and telescope setup had been completed, it is possible to manually perform measurements and observations **through the TPB**, which might as well pave the way – as preliminary checks – to longer lasting sessions carried out via schedules.

Raw counts readouts

The **raw counts readout** (called Tpi) of the signal can be obtained with:

```
> getTpi
```

The system reply consists in an array of values (one for each section). As concerns the TPB, it is important to ascertain that the Tpi lies within the 750-1000 counts range in order to keep your observations in the linear range of the backend and allow for fairly bright sources to stay within as well. If this requirement is not met, it is necessary to iteratively vary the attenuation for the needed sections and check the Tpi, until the signal intensity falls into the proper range.

As the signal level greatly varies with elevation, it is advisable to perform this operation in the elevation range that will be actually exploited during the observations or, as a general rule, at elevation=45°. Of course, the signal level is also greatly affected, especially at high frequencies, by weather conditions, therefore the attenuation tuning should be carried out again every time the conditions change.

When you are going to manually get the Tpi - in order to ascertain which attenuation values to use in your schedule - *remember to set the LO frequency and the bandwidth as they will be employed in the schedule*. After a setup command, in fact, they are set to defaults; instead, if a schedule has been previously run, these values remain set as indicated in the last schedule readout.

Note: Tpi values include the Tp0 level (the internal, "without sky" level, which is roughly 200 counts). Thus, when reading your output files, which contain the (Tpi-Tp0) signal, you will see a baseline 200 counts lower.

Tsys

To measure the Tsys value:

```
> tsys
```

the system replies with N values, where N is given by the total amount of (input_lines x sections). When using the TotalPower backend, N is the number of sections (2 for single-feed receivers and 14 for the MF).

Note: The last measured Tsys value will be stored in the system and used, *if the FITS format is selected for data storage*, to get a counts-to-Kelvin conversion factor, in turn applied to all the following acquisitions, until a new Tsys is measured. The FITS file will contain the raw data (in counts) and also a table with the data stream calibrated (in K) using this counts-to-Kelvin factor.

Weather parameters

The **weather station** measurements can be retrieved with:

```
> wx
```

the reply will list ground temperature (°C), relative humidity (%), atmospheric pressure (hPa), wind speed (km/h). Updated values are available every 10 seconds.

It is also possible to display the atmospheric temperature and the wind parameters (including wind direction) using a graphic interface. Activate the meteo client by using the following command in a terminal on nuraghe-obs1:

```
$ meteoClient
```

The following window will appear, it provides self-explanatory information.

Notice the top graph: the red line corresponds to wind speed = 61 km/h, at which the antenna must be stowed.

Considering that weather parameters are not necessarily written in the output files (in particular, wind speed and direction are not stored in the FITS files at present), it is possible to record a separate log containing these parameters plus information on the antenna pointing. Open a terminal on nuraghe-obs1 and execute the following command:

```
$ windLogger
```

The script, besides displaying the information on screen, will produce a file which contains:

- coordinates pointed by the antenna;
- atmospheric temperature;
- wind speed and direction.

These logfiles are stored in a dedicated folder on OBS1:

/archive/logs/WindLog (see also section "Retrieving the data").

Filenames are assigned according to date and time of the script execution. For example, if the script is launched on November 26th 2013 at 13:46:31 UT, the resulting file will be: azel_131126_134631.log During the acquisition, the shell must not be closed. To interrupt the acquisition, use CTRL+C from the keyboard.

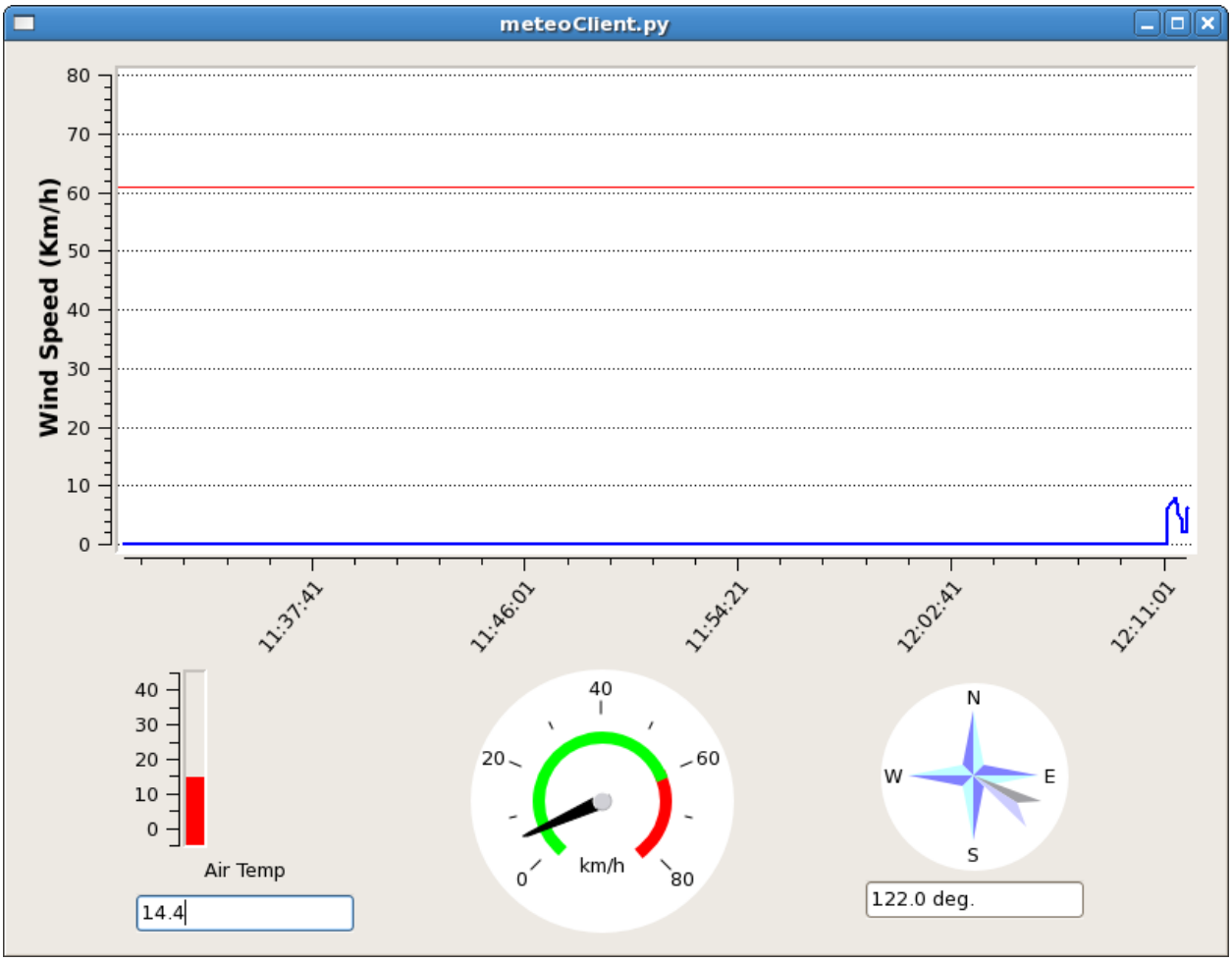
Manual acquisitions

When performing manually commanded acquisitions, it is necessary to **select the recording device**:

```
> chooseRecorder=[string]
```

where string can be:

- **MANAGEMENT/Point** (default) text output in the logfile, used for pointing calibration


meteoClient.py
Wind Speed (Km/h)
80
70
60
50
40
30
20
10
0
11:37:41
11:46:01
11:54:21
12:02:41
12:11:01
40
30
20
10
0
Air Temp
14.4
40
20
60
0
80
km/h
N
W
E
S
122.0 deg.

- **MANAGEMENT/CalibrationTool** text output in .dat file, used for pointing calibration
- **MANAGEMENT/FitsZilla** if FITS output is desired
- (MANAGEMENT/MBFitsWriter) if MBFITS is preferred – not yet available

Note: When recording **manually-acquired** data in FITS format, the **output files** are stored in a peculiar path: /archive/extraData. This implies that they also cannot be shown by the quick-look procedure.

Once the recorder is set, acquisitions on a target can be performed as follows. First, set the target:

```
> track=[sourcename]     (if the source is included in the system
                         catalogue)
```

For the available catalogue see *Appendix D - Source catalogue*. To set a generic target:

```
> sidereal=[sourcename],[RA],[Dec],[epoch],[sector]
```

(see *Antenna operations* for details)

Here follow the commands to be used to manually record your data (remember that the backend must have been properly set up and the target must have been specified as explained above):

```
> initRecording=[scn]
```

where [scn] in the scan number to be assigned to the acquisition. The `initRecording` command prepares the data recording. Then:

```
> startRecording=[sub_scn],[duration]
```

creates the output file and begins the data recording; [sub_scn] is the subscan number, [duration] is the acquisition duration, expressed as `hh:mm:ss`. Once the acquisition is completed, the user can launch another subscan and record the data with another `startRecording` command.

Finally, once the user wants to close the scan, the command to be used is:

```
> terminateScan
```

Output files will be found in the usual auxiliary folder where all the manual acquisitions are destined.

Example: acquisition of a sidereal scan on 3c123 composed by 2 subscans, each lasting 40 s, preceded by an off-source Tsys measurement:

```
> chooseRecorder=MANAGEMENT/FitsZilla
> track=3c123
> goOff=HOR,5
> waitOnSource
> tsys
> azelOffsets=0.0d,0.0d
> initRecording=1
> startRecording=1,00:00:40
> startRecording=2,00:00:40
> terminateScan
```

Pointing scans

Command cross-scans across a previously selected target (by means of the track or sidereal commands):

```
> crossScan=[subscanFrame],[span],[duration]
```

where subscanFrame is the coordinate frame along which the scan is performed (*eq*, *hor* or *gal*), span is the spatial length on sky of the individual subscan (i.e. one line of the cross) expressed in degrees, duration i s the time length espressed in hh:mm:ss,

e.g. *> crossScan=HOR,1.0d,00:00:30*

corresponds to one cross-scan carried out in Horizontal coordinates (one line along El, one line along Az), each line being 1° in span. Each subscan lasts 30 seconds, thus the resulting scan speed is 2°/min.

When the MANAGEMENT/Point writer is used, the cross-scan produces text output in the logfile only (no output file is recorded). This output text contains information obtained by the automatic processing of the subscans. In particular, a Gaussian fit is performed in order to measure the source position and estimate the **pointing** offsets. If the fitting procedure in successful and the achieved offsets are considered plausible, pointing **corrections are immediately applied**. This means that, if no user-defined offset is commanded (or cleared) afterwards, the measured offsets remain active and are applied to the following observations.

Here follows the function that is separately fitted to latitude and longitude subscans:

y(x)=A*e^W + ax +c

where:

- W = -2.7725887 * F^2
- F = (x-μ)/FWHM
- μ = abscissa of peak

The results are given in the logfile, in the following sequence of lines:

LATFIT latoff fwhm A a c i
LONFIT lonoff fwhm A a c i
OFFSET avlon avlat lonoff latoff lonflag latflag
XOFFSET avlon avlat lonoff*cos(lat) latoff lonoff_err *(cont.)*
latoff_err lonflag latflag
XGAIN target avlon avlat lonampl lonampl_err latampl *(cont.)*
latampl_err lonFWHM lonFWHM_err latFWHM latFWHM_err *(cont.)*
flux lonflag latflag

where (all angles in degrees):

latoff = latitude offset
lonoff = longitude offset
i = number of iterations performed by the fitting procedure
avlon = average longitude of peak (in same coordinate frame as the subscan execution)
avlat = average latitude of peak (in same coordinate frame as the subscan execution)
lonflag = fit result for longitude subscans
(1 = plausible fit, 0 = non plausible fit, -1 = fit did not converge)
latflag = fit result for latitude subscans
(1 = plausible fit, 0 = non plausible fit, -1 = fit did not converge)
lonoff_err = error on longitude offset

latoff_err = error on latitude offset
target = target name
lonampl = amplitude measured on longitude subscans (K)
lonampl_err = error on amplitude measured on longitude subscans (K)
latampl = amplitude measured on latitude subscans (K)
latampl_err = error on amplitude measured on latitude subscans (K)
lonFWHM = FWHM measured on longitude subscans
lonFWHM_err = error on FWHM measured on longitude subscans
latFWHM = FWHM measured on latitude subscans
latFWHM_err = error on FWHM measured on latitude subscans
flux = catalogue target flux (Jy), if available (otherwise it is put to 0.0).

.

Note: it is possible to **include such pointing scans using the MANAGEMENT/Point writer in schedules** as well. For example, an improved pointing can be achieved setting the first scan on a source as a /Point scan, then – in case the fit is successful – the following scans (e.g. producing FITS or MBFITS files) will hold the offsets optimising the pointing, given that no user-defined offset is updated by means of an explicit `radecOffsets`, `azelOffsets` or `lonlatOffsets` command.

Skydips

Skydip scans are indispensable in order to characterize the atmosphere. They consist in moving the telescope along a vast span in elevation (at fixed azimuth) while sampling with a backend. Their analysis allows the user to quantify the atmospheric opacity τ. There are different ways to perform this:

```
> skydip=[El1],[El2],[duration]
```

e.g. `skydip=20d,80d,00:05:00` performs a skydip between 80 and 20 degrees (at the current azimuth position), the scan will take 5 minutes (speed is thus 12 °/min). The arguments must be in the range 10-88.

The jolly character is supported for the elevation arguments. Example: `skydip=*,*,00:04:00` will perfom the skydip between the default values for elevation (15° and 90°). Please notice that pointing corrections are disabled.

Since no backend recording is automatically enabled by this command, remember to activate the FitsZilla recorder before launching the command, in order to save the data! This command can be used within schedules as well. See the separate guide to schedules for details.

Note: At present skydips are always performed **downwards**, i.e. starting from the highest elevation given in the command. **The greatest commandable elevation is 88°**, since the skydip, being an OTF subscan, will be additioned of an initial acceleration ramp – the length of which is proportional to the scanning speed.

Caveat on offsets

As seen in *Antenna operations*, there are commands used to set (or null) user-defined offsets. They are: `radeOffsets`, `azelOffsets` and `lonlatOffsets`. Such commands set **system offsets** which **remain active** until they are explicitly changed/nulled by another call of one of the three commands.

Further offsets, having for example the purpose of pointing the antenna to an off-source position, are specified inside schedules, at the subscan level (see the separate guide to schedules). **The subscan-level offsets sum up to the overall offsets if they are expressed in the same coordinate frame**, and they are zeroed by default every time a new subscan is commanded.

Generating and launching a schedule

A schedule is a set of files where all the geometry/timing/frequency details of a sequence of data acquisitions are specified, according to a syntax that enables ESCS to read and execute them. The detailed structure of the several files composing a schedule is explained in this separate guide *pdf* Schedules for the most common observing modes (OTF cross-scans and maps, raster maps and ON-OFF) can be easily generated using a tool called **basie**. It can be found, together with its user manual, here: Basie repository

Once a schedule is ready, it must be copied to the folder reserved to the schedules. There, the schedule formal consistency can be tested using the scheduleChecker command given within a terminal (*not* in the operatorInput):

```
> scheduleChecker [schedname].scd
```

Only the syntax correctness will be verified. If errors are present, a reply will briefly address them indicating their position inside the files.

To launch a schedule, simply use:

```
> startSchedule=[schedname].scd,[N]
```

The schedule name is the name of the SCD file. *N* is the identifier of the scan or subscan from which ESCS must start reading it – it is particularly useful in case of a sequential (i.e. not time-based) schedule. *N* can be the scan number, e.g. 2, or the scan_subscan specification, e.g. 2_5.

You can specify **when to start** the schedule (in UT):

```
> startSchedule=[schedname].scd,[N]@[DOY-HH:MM:SS]
```

ESCS reads the configuration parameters from the schedule, which can be relative both to the receiver and the backend, and accordingly sets these devices. This might take several seconds, especially when using the MF receiver. While the setup takes place, several values change in the TPB monitor. The last operation is the upload of the first pointing/scan read from the schedule, whose parameters will show up in the bottom section of the AntennaBoss monitor. During the scans, all the three flags in the lower part of the AntennaBoss monitor must be a green "@". Pay attention to the "Tracking" one. It should turn to a red "o" only when the antenna is slewing between scans on the same source, or when slewing to/from a new source. **If tracking is not correct for some consecutive seconds during a scan, something is wrong.** Sequential schedules run ad libitum. To abruptly interrupt the running schedule, truncating the ongoing acquisition:

```
> stopSchedule
```

Instead, to stop the schedule allowing the completion of the present file:

```
> haltSchedule
```

Data formats and online quick-look

Details on the FITS file structure are given in a separate document. Download it here: *pdf*

Waiting for a comprehensive GUI which is going to include also a real-time preview of the data under acquisition, users are provided with two different tools in order to inspect the data produced by the TPB.

If writer is MANAGEMENT/FitsZilla

When acquiring FITS files through a schedule, there is an IDL tool available for the almost-realtime quick-look of the saved data.

Open a terminal on *euser*. Launch IDL:

```
$ idl
```

At the IDL prompt, compile and run the program fitslook.pro:

```
IDL> .r fitslook
IDL> fitslook
```

The last completed FITS file (i.e. the most recent subscan carried out) will be displayed in terms of arbitrary counts vs sample number (i.e. raw intensity against time).

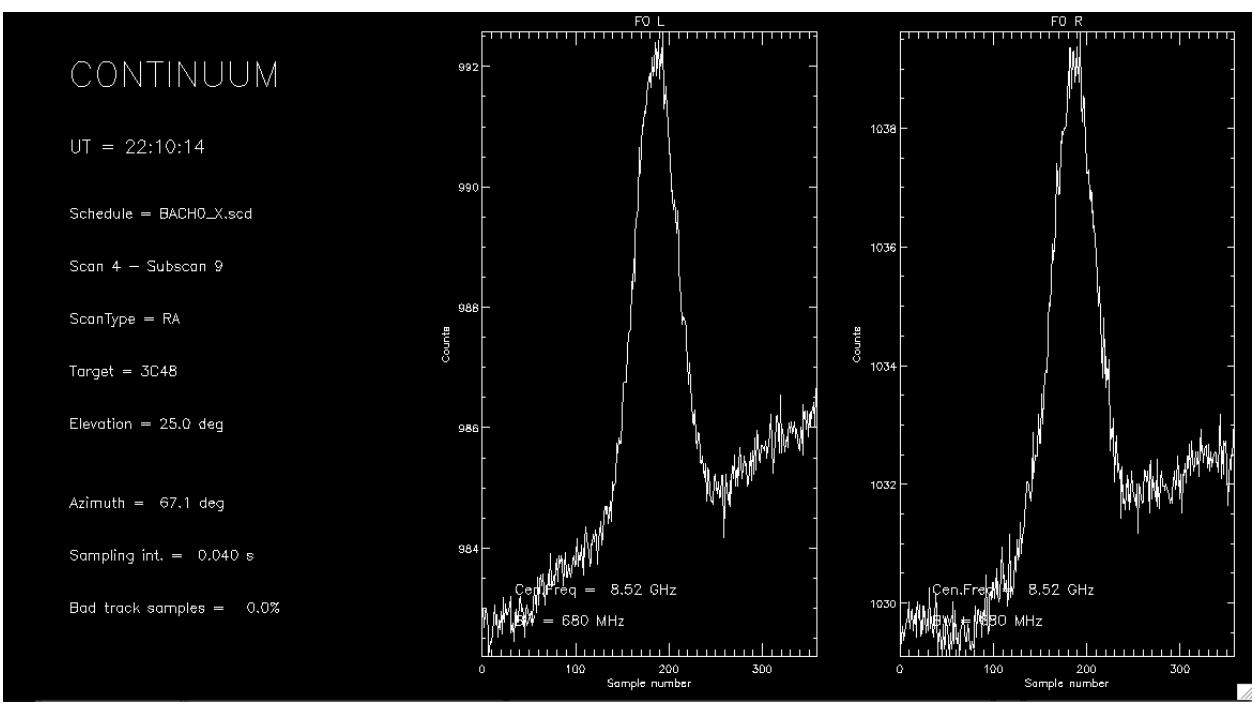
Note: The program automatically identifies the recorded sections and plots them all.

There are different options in order to plot the datai in terms of other quantities. Read all these options using:

```
IDL> fitslook, /help
```

Note: Data streams in antenna temperature are available - and selectable - only for the TPB and under the condition that a Tsys has been correctly acquired prior to the execution of the scan.

Please report any problem/request about this tool, as it is under development.



Note: If you are OK with the default settings, you can also launch the quicklook using the icon provided on the euser

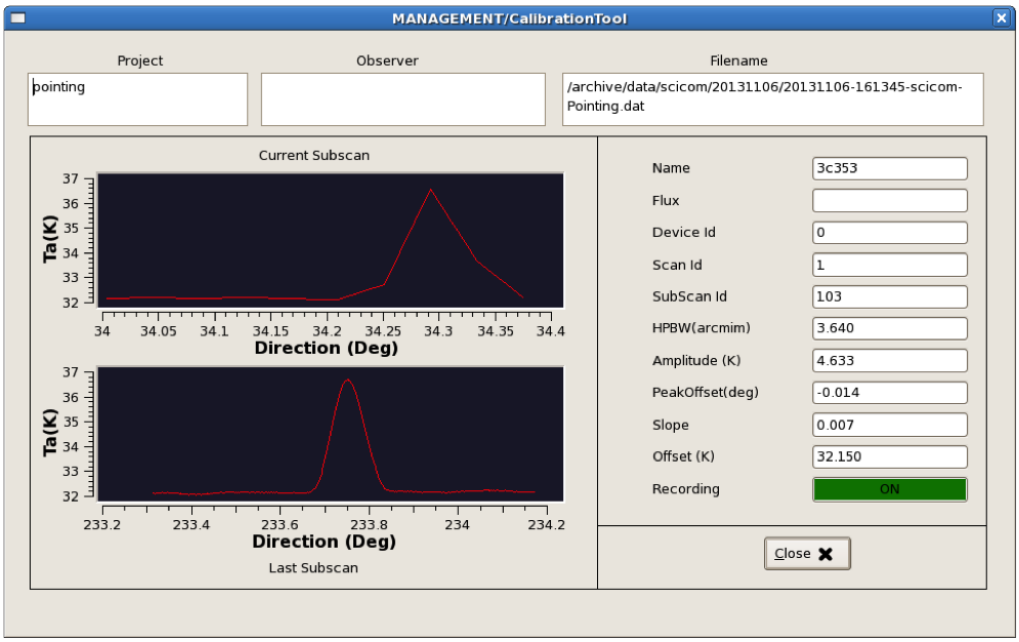
desktop, without having to start IDL, etc.

If writer is MANAGEMENT/Point or MANAGEMENT/CalibrationTool

When data are acquired – both manually or through a schedule – using the Point or CalibrationTool writers, the quick-look must be performed using the CalibrationToolClient. Open a terminal on escs and use the command:

```
$ calibrationtoolclient  [componentName]
```

where componentName is either MANAGEMENT/Point or MANAGEMENT/CalibrationTool. A graphic window will appear. Its content is given in the following figure.



In this client, the subscan currently being acquired is shown *in real-time* (upper plot), even if in a low-res version. Under this display, the last completed subscan - in its full sampling - is shown.

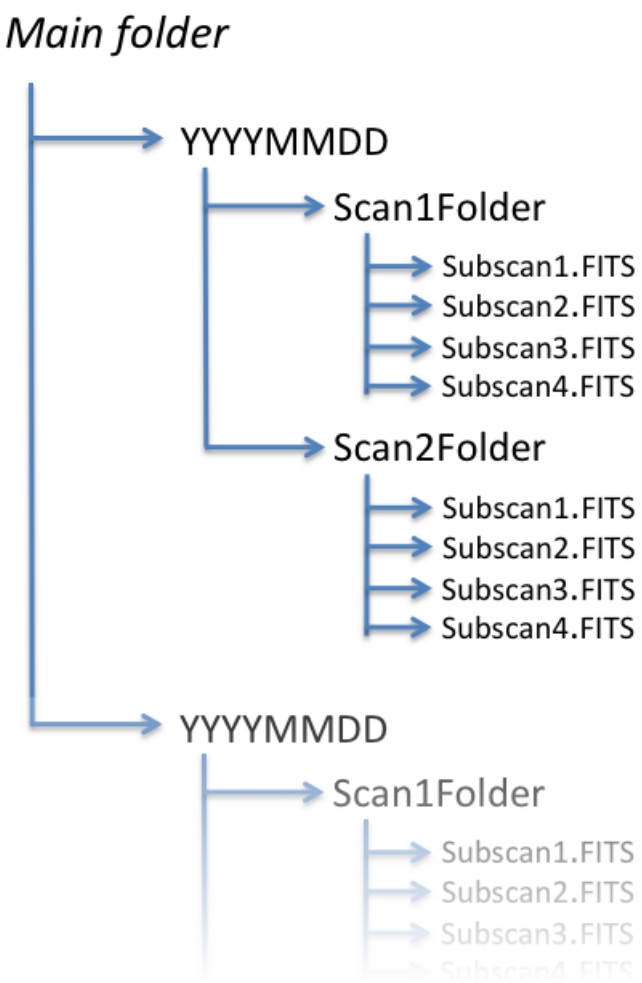
Note: You can also launch the calibrationtoolclient using the icon provided on the euser desktop, without having to open a terminal, etc.

Retrieving the data

Open a terminal on the euser machine. Your data folder is:

```
~/data
```

Inside that folder, subfolders named according to the date (YYYYMMDD) will be automatically created during acquisitions. Taking into account the choice of the FITS format, the only one so far fully tested, the date-dependent folder contains a subfolder for every scan, inside which there are the FITS files (one for each subscan).



Scan folder names are composed as:

YYYMMDD-HHMMSS-User-Suffix

where

- **HHMMSS** is the UT time associated to the first sample of the acquisition
- **User** is the user code (at the moment it can only be "euser")
- **Suffix is a user-defined string retrieved from the schedule files.** Though no control can be applied on the choice/check of this string, the agreement is that it should coincide with the target name.

FITS files, each corresponding to a subscan, are composed as:

YYYYMMDD-HHMMSS-User-Suffix_Scan#_Subscan#.fits

Data can be copied from *escsConsole* to your laptop using 'ssh' or 'rsync –e ssh' commands.

Note: When recording manually-acquired data in FITS format, the output files are still accessible from the euser machine, yet they are found in a peculiar path, different from the one employed when schedules run. You can find

these FITS files in /archive/extraData. This implies that they also cannot be automatically showed by the quick-look procedure.

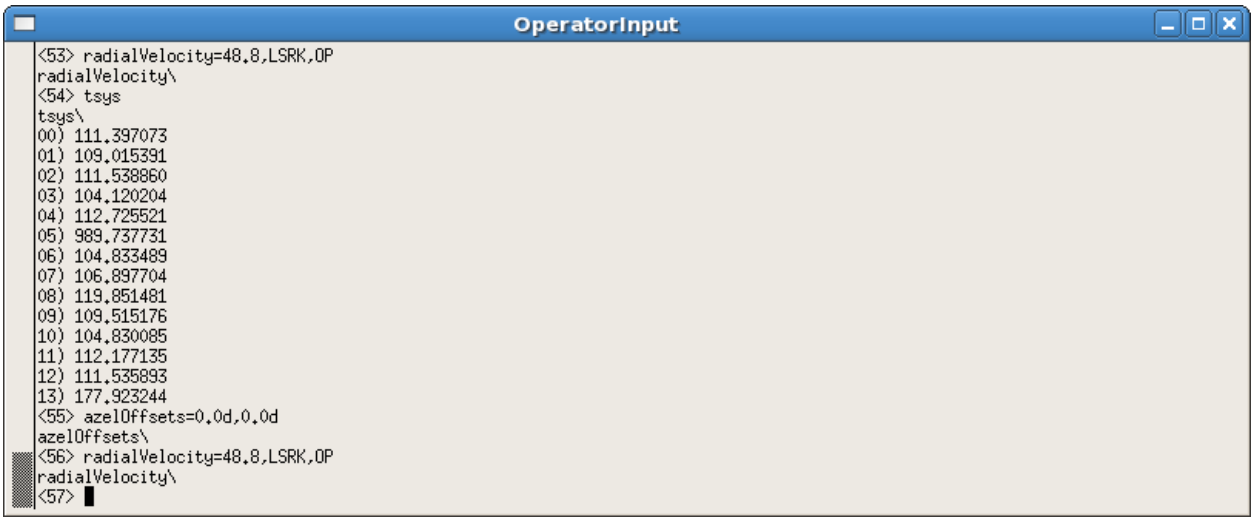
You can retrieve your logfiles here:

```
~/logs
```

Its reply will let you know where the logfiles are located. A subfolder named */WindLog* is devoted to the meteoClient output, if this client is used.

Appendix A - Monitor panels full description

operatorInput



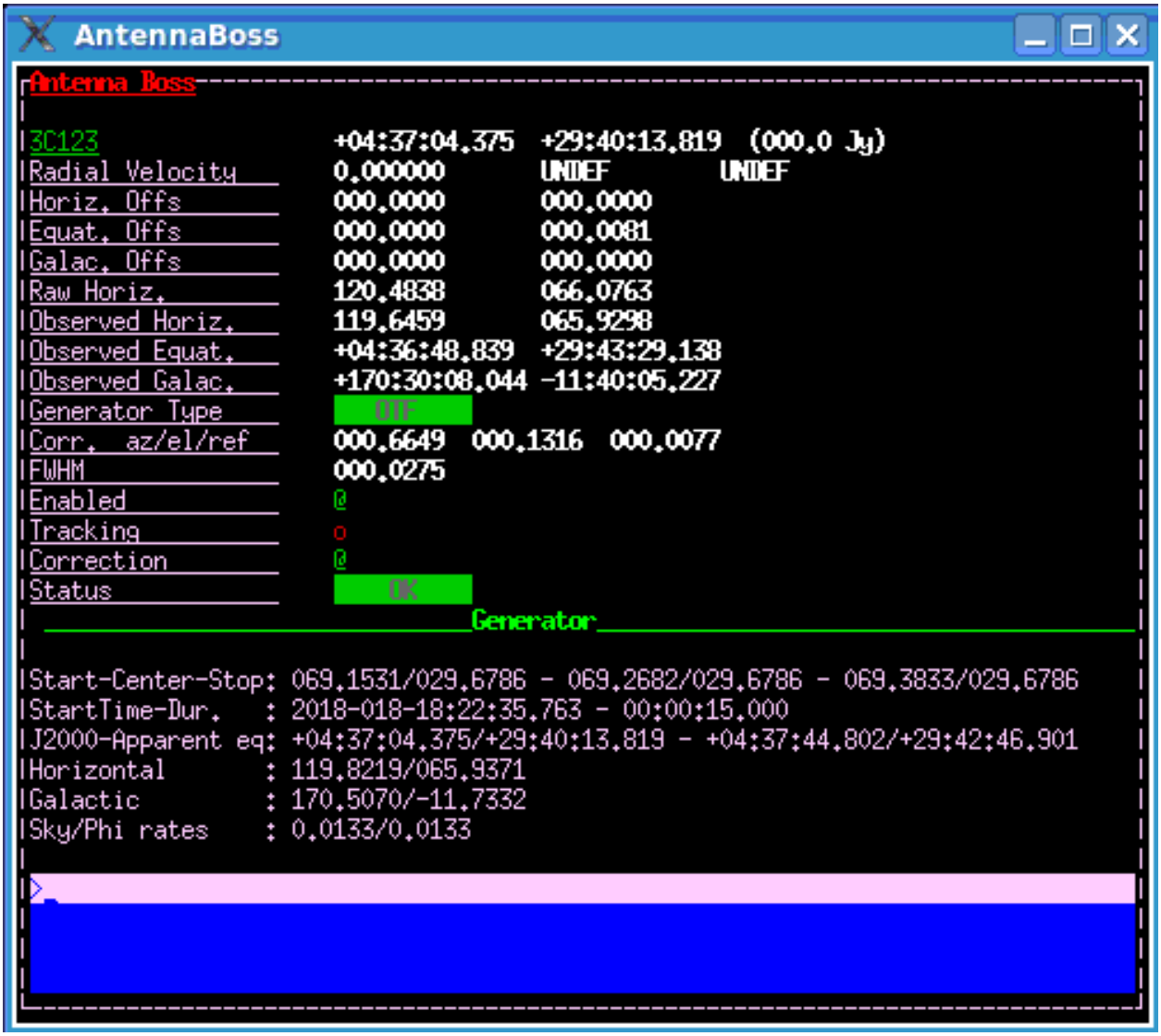
This is the **input console** where users write ESCS commands. The prompt is just a sequential number enclosed in <>. If a command is properly read, the system replies repeating the command itself, followed by the operation results (if they are foreseen). Otherwise, an error message appears. To close the window, type `exit`.

AntennaBoss

This monitor shows the target info, indicating the commanded and actual positions pointed by the antenna. It also gives a feedback on the pointing accuracy and on the overall antenna status.

Parameters:

- **Target**: label (as extracted from schedule or commanded), coordinates (RAJ2000.0, DecJ2000.0), estimated flux density (Jy, if available).
- **Radial Velocity**: (as extracted from schedule or commanded), value (km/s unless it is a redshift), velocity reference frame, velocity definition.
- **Horiz. Offs**: Horizontal (Az-El) offsets as read from schedule, degrees.
- **Equat. Offs**: Equatorial (RA-Dec) offsets as read from schedule, degrees.
- **Galac. Offs**: Galactic (l-b) offsets as read from schedule, degrees.
- **Raw Horiz.**: commanded Az-El, including pointing model, refraction, etc…


AntennaBoss
Antenna Boss
3C123 +04:37:04.375 +29:40:13.819 (000.0 Jy)
Radial Velocity 0.000000 UNDEF UNDEF
Horiz. Offs 000.0000 000.0000
Equat. Offs 000.0000 000.0081
Galac. Offs 000.0000 000.0000
Raw Horiz. 120.4838 066.0763
Observed Horiz. 119.6459 065.9298
Observed Equat. +04:36:48.839 +29:43:29.138
Observed Galac. +170:30:08.044 -11:40:05.227
Generator Type OTF
Corr. az/el/ref 000.6649 000.1316 000.0077
FWHM 000.0275
Enabled @
Tracking o
Correction @
Status OK
Generator
Start-Center-Stop: 069.1531/029.6786 - 069.2682/029.6786 - 069.3833/029.6786
StartTime-Dur. : 2018-018-18:22:35.763 - 00:00:15.000
J2000-Apparent eq: +04:37:04.375/+29:40:13.819 - +04:37:44.802/+29:42:46.901
Horizontal : 119.8219/065.9371
Galactic : 170.5070/-11.7332
Sky/Phi rates : 0.0133/0.0133

- **Observed Horiz.**: Az-El coordinates read from the mount encoders and cleaned from the pointing model and refraction contributions. Because of this, observed coordinates will differ from the raw ones.
- **Observed Equat.**: RA-Dec J2000.0 coordinates, converted from the observed horizontal.
- **Observed Galac.**: l-b Galactic coordinates, converted from the observed horizontal.
- **Generator Type**: which component is in charge of the generation of the coordinates. It can be `NONE` (which is the condition at startup), `OTF`, `MOON` or `SIDEREAL`.
- **Corr. az/el/ref**: azimuth and elevation corrections (degrees) applied by the pointing model, plus the refraction model contribution– which is an additional correction to elevation.
- **FWHM**: Full Width Half Maximum (corresponding to HPBW), degrees.
- **Enabled**: a green `@` indicates that the antenna is correctly receiving commands; a red `o` means the communication is disabled.
- **Tracking**: it indicates whether the pointing error exceeds 0.1*FWHM or not. A green `@` corresponds to error < 0.1*FWMH. It should turn to a red **o** only when the antenna is slewing between scans on the same source, or when slewing to/from a new source.
- **Correction**: application of the above horizontal coordinates corrections. If disabled (red circle), all corrections are zeroed.
- **Status**: `OK`, `WARNING` or `ERROR`. "Warning" needs investigation but usually does not stop the ongoing activity (it also appears at startup, before the setup commands), "Error" generally appears if something stops the observations.
- **Generator**: under this line all the subscan setup parameters appear when it is commanded. See the two cases here below.

In particular, for OTF observations:

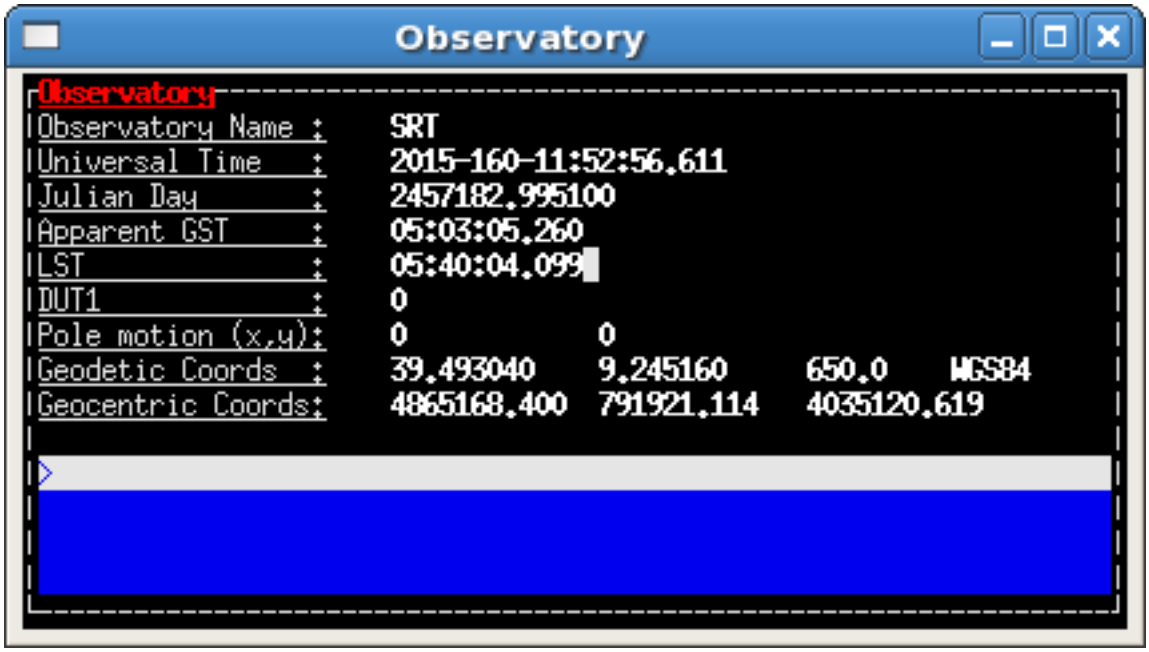
- **Start-Center-Stop** – coordinates in the subscan frame of the start, center and stop positions (degrees)
- **StartTime-Dur.** – subscan start UT `yyyy-doy-hh:mm:ss.s` and duration `hh:mm:ss.s`
- **J2000-Apparent eq** – during the subscan, current RA-Dec position pointed by the antenna, both in the J2000 epoch and precessed to the date. Format is `hh:mm:ss.s-°:′:″.″`
- **Horizontal** - during the subscan, current Az-El position pointed by the antenna (degrees)
- **Galactic** - during the subscan, current l-b position pointed by the antenna (degrees)
- **Sky/Phi rate** – actual subscan speed on sky along scanning direction (°/s) . The 'Phi rate' value refers to great circle arcs, if it is the scanning geometry, otherwise it copies the first speed value.

For **SIDEREAL** scans, instead, this is what appears:

- **Source name** – target name
- **Catalog Eq.** – target FK5 coordinates and info (if available): RA `hh:mm:ss.s`, Dec `°:′:″.″`, Epoch, RA proper motion (milliarcsec per julian year), Dec proper motion (milliarcsec per julian year), parallax (milliarcsec), radial velocity (km/s)
- **Catalog Gal.** – target Galactic longitude and latitude (degrees) converted from the above Equatorial or provided by user/catalogue
- **Apparent Eq.** – target current apparent Equatorial coordinates: RA `hh:mm:ss.s`, Dec `°:′:″.″`, Epoch
- **Galactic** – current l-b position pointed by the antenna (degrees)
- **Horizontal** –current Az-El position pointed by the antenna (degrees)

To close the monitor, type `exit` in its prompt (the grey line).

Observatory



It is devoted to the station coordinates and times:

- **Observatory Name**: SRT, Medicina or Noto
- **Universal Time**: `YYYY-DOY-HH:MM:SS.SSS`
- **Julian Day**: `d.ddd`
- **Apparent GST**: Greenwich Sidereal Time `HH:MM:SS.SSS`
- **LST**: Local Sidereal Time `HH:MM:SS.SSS`
- **DUT1**: difference between UT1 and UT (s), if applied.
- **Pole motion**: celestial pole offset w.r.t. a reference position (details are note provided here), measured in μarcsec on a tangential projection.
- **Geodetic Coords**: updated Latitude and Longitude (degrees) and Altitude (m) for the telescope, plus the geodetic model code.
- **Geocentric Coords**: geocentric cartesian coordinates (m) for the telescope.

To close the monitor, type `exit` in its prompt (the grey line).

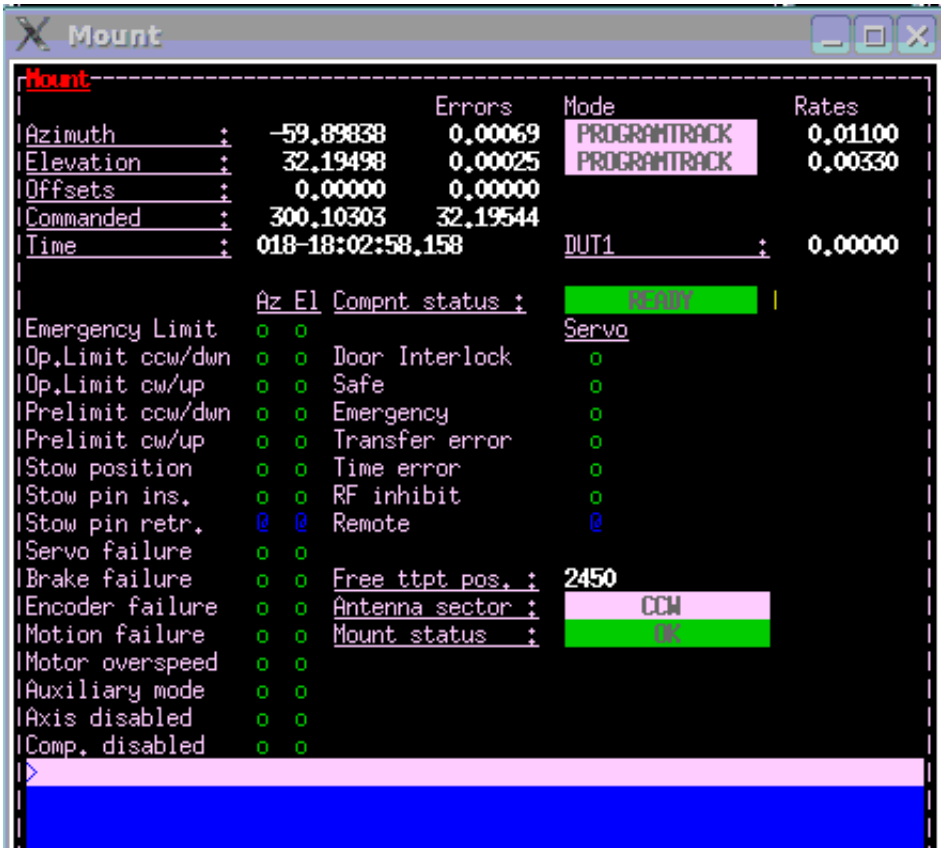
Mount

This is a quite complex frame, allowing the technical staff to monitor all the various parts of the antenna mount.

Observers must focus only on a subset of the displayed parameters and flags. We thus describe the main features only.

The top left section gives the online readouts for the Azimuth and Elevation axes, compared to the commanded positions. Line by line:

- **Azimuth**: current azimuth position (in the -90°/450° wrapping range), rate (°/s) and commanded offset (degrees). Then the measured position error and tracking error (both in degrees) are given. Finally, the axis active



mode is displayed: it can be *PRESET* (fixed Az-El pointing only), *PROGRAMTRACK* (for tracking, OTF and schedules), STOP (if axes brakes are on), *UNKNOWN* (indicating a non-configured status).

- **Cmd Az.**: commanded azimuth position (in the -90°/450° wrapping range), rate (°/s) and offset (°). This offset comes from metrology devices, it is not related to user-defined offsets or subscan-defined offsets.
- **Elevation**: current elevation position (degrees), rate (°/s) and commanded offset (degrees). Then the measured position error and tracking error (both in degrees) are given. Finally, the axis active mode is displayed: it can be *PRESET* (fixed Az-El pointing), *PROGRAMTRACK* (for OTF and schedules), *STOP* (if axes brakes are on), *UNKNOWN* (usually indicating an error condition).
- **Cmd El.**: commanded elevation position (degrees) and rate (°/s).
- **Wrap**: readouts from the cable wrap. The status flag on the right can be *CW* ''*or* ''*CCW*.

The general condition of the mount is summed up by one keyword, in the central part:

- **Mount status**: it can be *OK*, *WARNING*, *FAILURE* or *BUSY*, the latter is associated to operations which take a lot of time to complete (as the stow-unstow procedure). When the mount is BUSY it will not accept further commands until the ongoing operation is completed.

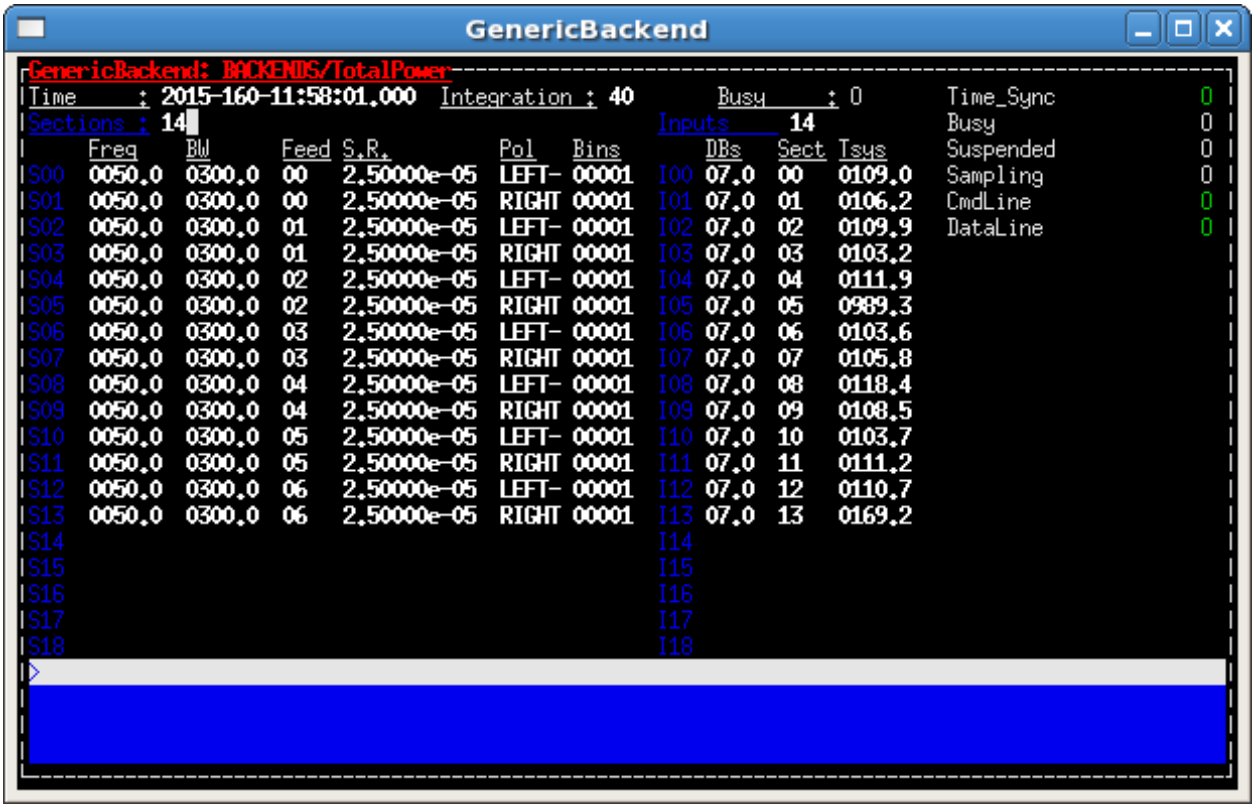
The rest of the panel lists several **flags and status labels**. For average users, the only points to be taken into account are:

- in standard observing conditions, when a schedule runs, all flags should be green *o*;
- in case of warnings, flags turn to yellow *o*;
- errors correspond to red *o*;

- when a "failure" keyword turns steadily to a red `o`, or a permanent failure status appears, immediately call for technical assistance.

To close the monitor, type `exit` in its prompt (the grey line).

GenericBackend



The panel shows one row for every section.

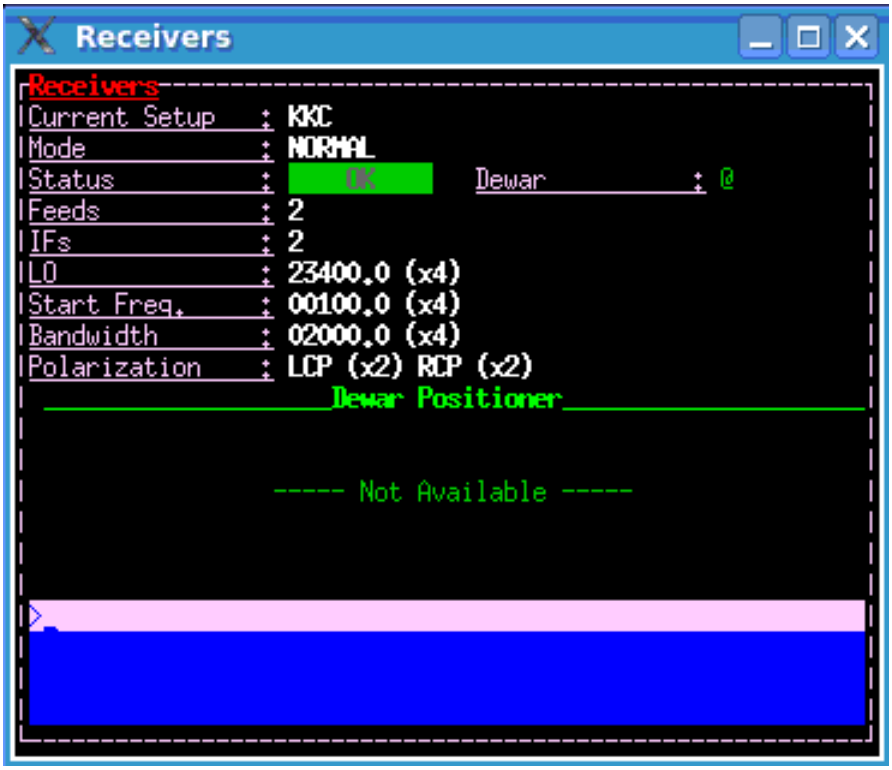
- **Freq**: value (MHz) to be added to the LO frequency in order to obtain the observed frequency at the beginning of the band
- **BW**: bandwidth (MHz)
- **Feed**: number of the receiver feed connected to this section
- **S.R.**: sampling rate (MHz)
- **Pol**: polarisation (Left or Right)
- **Bins**: number of frequency bins (1 for total power)
- **DBs**: attenuation (dB) applied to the section
- **Sect**: section number
- **Tsys**: the last measured Tsys (K)

Also some backend-dependent status flags are present, in the top right area. When observing with the TPB, look for:

- **Time_Sync**: if it frequently or steadily turns red call for assistance (the backend time is not synchronized)
- **Busy**: when schedules are running, it must turn yellow. If it does not, the backend is not acquiring.

To close the monitor, type `exit` in its prompt (the grey line).

ReceiversBoss



This monitor summarizes the frontend setup. The bottom part is devoted to the derotator (dewar positioner), when available.

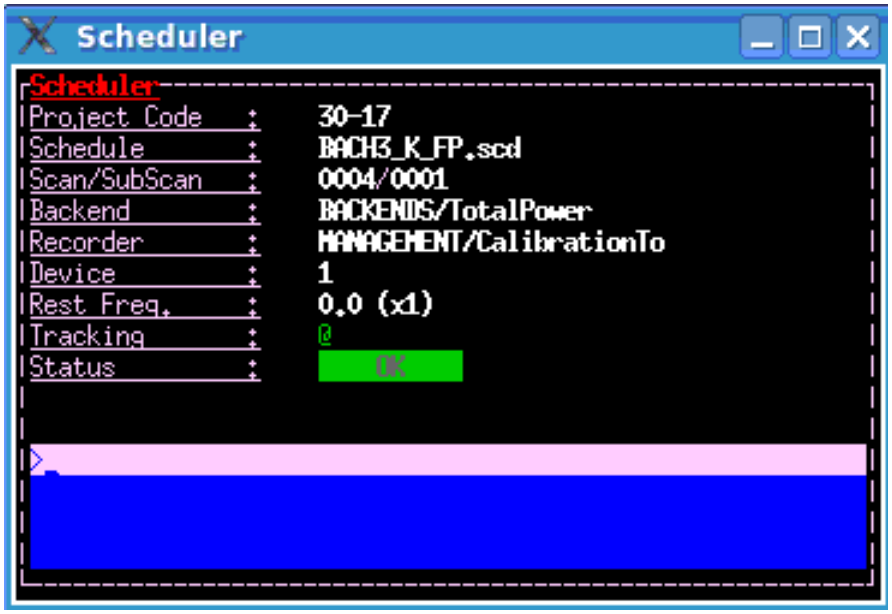
- **Current setup**: receiver code
- **Mode**: `NORMAL`, `SINGLEDISH`, `VLBI...` (other codes to come)
- **Status**: `OK`, `WARNING` or `FAILURE`
- **Feeds**: number of feeds (1 fore single-feed receivers, 2 for the K-band DF)
- **LO**: Local Oscillator frequency (MHz)
- **Start Freq.**: step (MHz) to be added to the LO frequency in order to obtain the actually observed frequency at the beginning of the band
- **Bandwidth**: actually observed bandwidth (MHz)
- **Polarization**: `LCP`, `RCP`

To close the monitor, type `exit` in its prompt (the grey line).

Scheduler

This monitor shows details on the selected data acquisition devices and on the running schedule, if any.

- **Project code**: project name, as specified at the launch of the schedule (or as input using the *project*= command)



- **Schedule**: name of the active schedule
- **Scan/Subscan**: scan number and subscan number, relative to the ongoing acquisition
- **Backend**: name of the selected backend, as listed in the schedule
- **Recorder**: name of the selected data recording device, as listed in the schedule
- **Device**: number of the currently selected device (see device= command)
- **Rest Freq.**: rest frequency of the line(s) being observed (MHz). The number in brackets indicates how many IFs the given rest frequency refers to.
- **Tracking**: tracking status of the telescope, including antenna, active surface, minor servo. A red *o* means the telescope is not yet in its tracking route (or that it is not tracking properly), a green light means the observation is ongoing and the tracking is accurate within 0.1*HPBW
- **Status**: flag summarising the telescope status. *OK* is self-explanatory, *WARNING* in principle indicates that the acquisition can go on even if a problem has been reported, *ERROR* signals that a major problem was detected and the observation cannot be performed.

To close the monitor, type *exit* in its prompt (the grey line).

MinorServo

This monitor shows details on the Minor Servo systems:

- **Current Setup**: current setup code (**CCC**, **XXP**, **KKC**, . . .)
- **Ready**: when green, the receiver is in its working position
- **Starting** : when green, the minor servo are moving in order to put the selected receiver in its working position
- **AS Conf**: when green, the subreflector position is computed by assuming the primary mirror surface is shaped and active (it is changing with the antenna elavation)
- **Elevation Track**: when green, the subreflector is changing its position with the elevation

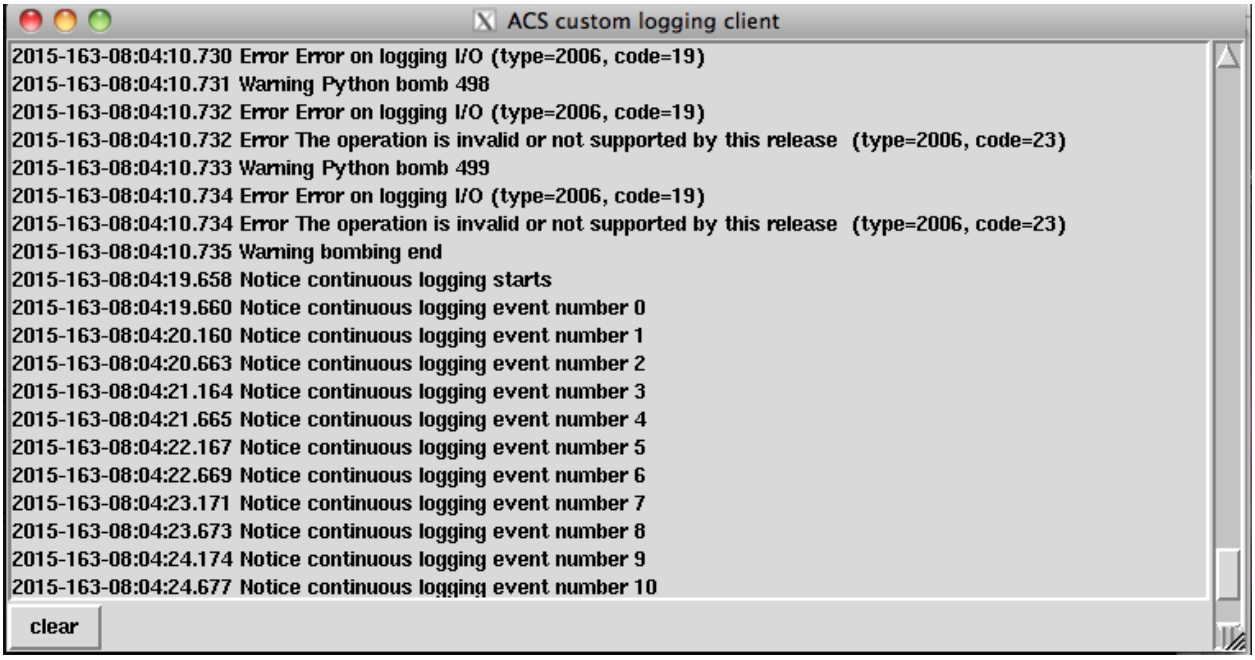
- **Scan active**: when green, a scan is active. That means either the system is performing a scan or a scan is terminated and so the system is waiting for a stopScan command or for a new scan
- **Scanning**: when green, the system is performing a scan

To close the monitor, type *exit* in its prompt (the grey line).

Note: All the monitors can be closed at a time using the command (in a terminal on escsRemote):

```
$ escsConsole --stop
```

Logging Display



This panel is not part of the general Console, so it must be individually opened/closed. Here all the log messages will appear. New messages are shown on top of the previous ones. To close the window, click on its dedicated icon.

Appendix B - Complete command list

Note:

COMMAND TEMPORISATION All commands can be temporised (i.e. associated to a starting time) adding a proper **suffix**. There are two possibilities:

- **absolute temporisation** (the operation will be performed at the indicated time): `@DOY-HH:MM:SS`, where DOY is the Day-Of-Year (1-366) and HH:MM:SS is the UT time;
- **iterative temporisation** (the operation is performed now, then periodically according to the indicated time interval): `@!DAYS-HH:MM:SS`, where DAYS is the number of days and HH:MM:SS is hours, minutes, seconds.

Commands (temporised or not) can be used also in the init/pre-scan/post-scan procedures inside schedules. *Observers* are in charge of considering *if* and *when* the use of a certain command makes sense in their schedule, according to their specific needs and goals: this is something that no schedule parser can check!

Here follow all the commands exploitable in ESCS:

Note: Spaces within the command line content are **not** allowed!

> antennaPark
sends the antenna to stow position

> antennaReset
resets the alarm conditions

> *antennaSetup=[code]*
(*code* can be *PRIM* or *SEC*) unstows the antenna, sets it to tracking mode and configures the pointing model according to the specified focus (primary or secondary). It does NOT perform the receiver and backend setup

> *antennaStop*
stops the antenna. Activities can start again only commanding a mode change as *antennaTrack* (which does not affect the overall setup) or a new setup

> *antennaTrack*
sets the antenna to PROGRAMTRACK mode. It does not change the pointing model or any receiver setup

> *asOff*
disables the AS tracking: actuators go to the fixed position set for 45° of Elevation

> *asOn*
enables the AS tracking: actuators move according to the pointed Elevation (with a 0.5° step)

> *asPark*
sends all the actuators to their lowest position

> *azelOffsets=[double]d,[double]d*
sets the Az-El offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source Elevation.

Example: *> azelOffsets=-0.05d,0.05d*

> *calOn*
switches the calibration mark on

> *calOff*
switches the calibration mark off

> *chooseBackend=[string]*
selects the backend; *string* can be *BACKENDS/TotalPower*

> *chooseRecorder=[string]*
selects the backend; *string* can be *MANAGEMENT/FitsZilla*, *MANAGEMENT/CalibrationTool* or *MANAGEMENT/Point*

> *crossScan=[scanFrame],[span],[duration]*
performs a cross-scan on the previously selected target (indicated using the *track* or *sidereal* commands), along the *scanFrame* (*EQ*, *HOR* or *GAL*), spanning *span* degrees in *duration* seconds

> *device=[sect]*
computes the beamsize, taking into account the present receiver and backend configurations relative to section sect

> *flush=[N]*
deletes the N-th element in the queue of temporised commands

> *flushAll*
deletes all the queue of the temporised commands

> *fTrack=[dev]*
It collects all the required data from the antenna, the back-end and the front-end, plus the information provided by the user (see the *radialVelocity* and *restFrequency* commands), then it tunes the telescope devices in order to centre the line(s) in each section bandwidth. The command lets the user select which device [dev] is asked to perform the tuning:

- **LO**: only the front-end local oscillator is moved
- **ALL**: the back-end performs a sub-tuning in the various sections

> *getAttenuations*
reads the attenuation values (dB) currently configured for the active sections, and lists them according to increasing section number

> *getTpi*
reads the signal intensity (raw counts) for the active sections, and lists them according to increasing section number

> *goOff=[frame],[beams]*
slews the antenna to an offset position, wrt a previously commanded target, along the longitude axis of the indicated coordinate frame (`EQ`, `HOR` or `GAL`). The user provides the offset value expressed in beamsizes. If the frame is HOR and target lies beyond the Elevation cutoff limits, the offset is applied in Elevation.

> *goTo=[double]d,[double]d*
sends the antenna, while in TRACKING mode, to the specified Az-El position.

Example: `goTo=180d,45d`

Arguments are always rounded in the range 0-360 and 0-90 for azimuth and elevation respectively (in any case the ranges are limited to mechanical contraints). The jolly character is valid and is considered as: keep the present value. The differences from the `preset` command are:

- once the antenna reaches the destination, the system will acknowledge the "on source" status;
- the pointing corrections (pointing model and refraction) are applied. In case they are not required they must be turned off explicitly.

> *haltSchedule*
completes the current scan and then stops the schedule

> *initialize=[code]*
(`code` can be `CCC`, `MMC`, `KKC`, `QQC`, ...) configures the backend using the default parameters relative to the selected receiver. It does *not* act on the receiver, pointing model or antenna mount mode.

> *integration=[double]*
sets the backend integration time (ms)

> *log=[filename]*
defines a custom name for the logfile (do not specify the extension)

> *lonlatOffsets=[double]d,[double]d*
sets the Galactic b-l offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source latitude.

Example: `> lonlatOffsets=2.0d,-1.0d`

> *moon*
points the antenna to the present coordinates of the center of the Moon

> *preset=[double]d,[double]d*
sends the antenna, if in PRESET mode, to the specified Az-El position, without applying any pointing correction. This is useful when needing to point to a position next to the zenith. Beware: the antenna will reach the destination but no "on source" flag will be raised.

Example: `> preset=180d,45d`

> *project=[code]*
lets the system know which project is observing (the code/name must correspond to the one provided by the TAC). This code/name is then considered as default when launching schedules: the system will search for them in a folder named "project/schedules". This code/name also forms part of the output FITS filename. Notice that the PROJECT keyword indicated inside the schedule, which is then written in the "Project Name" keyword in the FITS main header, is a free string and might differ from the project official name.

> radecOffsets=[double]d,[double]d
sets the RA-Dec offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source Declination.

Example: `> radecOffsets=1.0d,0.0d`

> radialVelocity=[vrad],[vref],[vdef]

- [vrad] (radial velocity) is in km/sec if vdef is *not* Z
- [vref] (reference frame) can be one of the following:
  - **BARY**: Solar System BARYCENTRE
  - **LSRK**: Kinematic Local Standard of Rest
  - **LSRD**: Dynamical Local Standard of Rest
  - **LGRP**: Local Group
  - **GALCEN**: Galactic Centre
  - **TOPOCEN**: TOPOCENTRIC (observer's frame)
- [vdef] (velocity definition) can either be:
  - **RD**: Radio Definition
  - **OP**: Optical Definition
  - **Z**: stands for Redshift

The specified velocity parameters are valid until a new target is commanded. The `radialVelocity` command overrides any other velocity value that might have been differently expressed

> receiversMode=[code]
configures the working mode of the receiver, according to its peculiar characteristics

> receiversSetup=[code]
(`code` can be `CCC`, `MMC`, `KKC`, `QQC`, ...) configures the receiver using the default parameters. It does *not* act on the backend, pointing model or antenna mount mode

> restFrequency=[freq1];...;[freqN]
[freq] is given in MHz and is a multiple argument: it can list a different value for each of the N sections - as long as XARCOS is the backend in use(not all the backends allow this sub-tuning). Specifying a single value assigns the rest frequency to *all* the sections. The specified values will hold until different ones are commanded, or until a new general *setup* command is entered.

> setAttenuation=[sect],[att]
sets to *att* (dB) the attenuator of section *sect*

> setLO=[freq]
Local Oscillator frequency, in MHz (one per IF, separated by ";", usually the values are identical) This LO frequency corresponds to: SkyFreq(@band start) – 100 MHz when using the TPB

> setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]
configures the backend section sect.

> setupCCC (setupKKC, etc...)
unstows the antenna, sets it to tracking mode, selects the pointing model, and configures the receiver and the backend using default parameters. In practice, it is a shortcut corresponding to this sequence:

```
> antennaSetup=[code]
> receiversSetup=[receiverCode]
> initialize=[receiverCode]
> device=0
> calOff
```

> *sidereal=[sourcename],[RA],[Dec],[epoch],[sector]*
points to the supplied RA-Dec position and temporarily assigns the sourcename label to it. Epoch can be *1950*, *2000* or *-1*, the last one meaning that the provided coordinates are precessed to the observing epoch. The sector keyword forces the cable wrap sector, if needed: its value can be *CW*, *CCW* or *NEUTRAL*. The last option means the system will automatically choose the optimal alternative.

Example: *> sidereal=src12,319.256d,70.864d,2000,neutral*

> *skydip=[El1]d,[El2]d,[duration]*
performs an OTF acquisition at the current azimuth position, spanning in elevation from *El1* to *El2* (both expressed in degrees, with 'd' suffix), in *duration* time, expressed in hh:mm:ss. A recorder must have previously been enabled in order to save the data, and a reference position must have been selected via a *sidereal*, *track* or *goTo* command.

> *startSchedule=[project/][schedulename].scd,[N]*
runs schedule *schedulename*.scd (project is the ID of the observing project, it is optional if it has already been input through the *projectCode* command), reading it from line *N*

> *stopSchedule*
immediately stops the running schedule, truncating the acquisition

> *ti*
lists all the active temporised commands

> *track=[sourcename]*
points the antenna, in sidereal tracking, to the specified source, which must be present in the local catalogue

> *tsys*
measures the system temperature (K) in the position the antenna is pointing to. It returns a list of values, one for each section in use. Intermediate steps and calculations are stored in the active logfile

> *wait=[double]*
sets a delay (in seconds) which is applied before the system reads/executes the next command

> *wx*
returns the current weather parameters: ground temperature (°C), relative humidity (%), atmospheric pressure (hPa), wind speed (km/h).

Appendix C - Output files

The system at present allows the user to write the output data, coming from the integrated backends, in two different formats: FITS and MBFITS. The former is a substantially **standard FITS**, carrying the data and a load of ancillary information inside several extensions. A FITS is produced for every subscan composing the observation.

MBFITS is a more complex format, not yet fully implemented and whose writer component is still under testing. Each MBFITS contains the whole scan, and subscans are hierarchically organized inside its structure.

Details on both formats are provided in this separate manual *pdf*

Appendix D - Source catalogue

Here follow the sources which are listed in the system catalogue. Their names, as presented here, can be used in schedules (SIDEREAL subscans, inside the LIS file) or in the command:

```
> track=[sourcename]
```

Along with their J2000 Equatorial coordinates, the catalogue records usually contain further information such as apparent size, proper motion, radial velocity, coefficients of the *Ott et al.* polynomials describing the spectra, etc… ESCS uses these parameters to make computations during certain operations (e.g. when calibration tools are used – under development, they will be described in future releases of this manual):

- 3c48
- 3c84
- 3c123
- 3c147
- 3c161
- 3c218
- 3c227
- 3c249.1
- 3c286
- 3c295
- 3c309.1
- 3c348
- 3c353
- 3c380
- 3c391
- casa (i.e. Cas A)
- cyga (i.e. Cyg A)
- dr21
- ngc7027
- oria (i.e. Ori A)
- taua (i.e. Tau A)
- vira (i.e. Vir A)
- w3oh

Troubleshooting and temporary procedures

The following sections list the solutions to the most common problems encountered by users, including indications on how to deal with present bugs or inefficiencies in the code. The records are sorted according to the main operation areas in which the problems arise.

Note: Users are advised to read these "hints" at the beginning of the session and before attempting (possibly unnecessary) heavy-handed operations on the system.

Setup

PROBLEM:

- **The initial setup fails, but it is difficult to assess what goes wrong.**

Instead of using the overall setup commands, give in the *operatorInput* the **individual commands** which deal with the different sub-systems, so that it is easier to identify the misbehaving element.

For example, the setupCCC command can be substituted by (the actual receiversMode code to be used depends on the desired setup):

```
> antennaReset
> antennaSetup=SEC
> servoSetup=CCC
> receiversSetup=CCC
> initialize=CCC
> device=0
> calOff
```

For other receivers, the codes of course vary.

Schedules

PROBLEM:

- **I've performed cross-scans in order to fine-tune the pointing, but the measured offsets are not actually applied to the following acquisitions**

System offsets, such as the ones measured with a Point acquisition, sum up to the ones indicated inside schedules ONLY if they are expressed in the same coordinate frame. This means that, if you perform observations using EQ offsets, also the fine-pointing cross-scans must be carried out in the EQ frame. The same holds for HOR scans. If there is a frame mismatch, the system offsets are automatically rejected (bug under fixing).

PROBLEM:

- **I've launched a surely correct schedule, but parts of the system crashed.**

Pay attention to the syntax, you might have inserted unwanted characters like an extra "/": the *startSchedule* command must be either given as:

```
> startSchedule=euser/[schedname].scd,[N]
```

or, as long as euser is the only available user, as:

```
> startSchedule=[schedname].scd,[N]
```

(Fault report): under development

Users are required to report all faults and misbehaviors by means of the ticketing system:

```
XXXXXXXXXXXX
```

1.1.3 Observing at the SRT with DISCOS

Current release: what's new

DISCOS 1.0 is the present release. Here are the new elements it introduces:

New logistics

SARDARA configurations and commands

New data quick-look tool

FITS update: new columns in the SECTION TABLE, new values for the SIGNAL keyword in main header

summary.fits update

History

Issue	Release Date	What's new
13	15/11/18	**Release of DISCOS 1.0** New logistics (observing machines and relative login). SARDARA is now fully integrated; addition of new configurations and capabilities. New online data quick-look, replacing the previous procedures. Updates in the FITS files content (inside the SECTION TABLE) and in the summary.fits content (several keywords are not set to NULL anymore) This release is to be paired to the following documents * DISCOS schedule structure (v. 1) * FITS and MBFITS output formats in DISCOS (v. 1) * basie User Manual (v. 1)
12	29/01/16	**Release of NURAGHE 0.6** Addition of the SARDARA back-end. New instructions for LLP LO setup. Updates in the Startup, Data retrieval and Quick-look sections. This release is to be paired to the following documents * Nuraghe schedule structure (v. 3) * FITS and MBFITS output formats in Nuraghe (v. 3) * basie User Manual (v. 1)

11	25/05/15	**Release of NURAGHE 0.5** Addition of Spectroscopy and Derotator sections. Changes in goOff command. New commands for manual data acquisitions. Changes in XARCOS configuration codes. This release is to be paired to the following documents * Nuraghe schedule structure (v. 2) * FITS and MBFITS output formats in Nuraghe (v. 2)
10	02/02/15	Checklist revision. Added Troubleshooting and FaultReport sections. Removal of previous Appendix C (Schedule structure), moved to a separate guide. “antennaReset” removed from *setupXXX* command internal list of actions.
09	11/11/14	**Release of NURAGHE 0.4** Addition of XARCOS commands. Revision of AS and general commands. Improved codification of commands. Addition of TAB-completion of commands.

08	04/04/14	**Release of NURAGHE 0.3** Added commands for AS, MS, setup/park. Addedd focusScan command. Updated info on Meteo client. Added info on BCK schedule file. Deleted sections on data formats and on “Nuraghe from Scratch”, which become independent documents. Checklist moved to document beginning. Updates on the operating machines.
07	03/12/13	Addition of info on observing machines. Updates on receviersMode command. Addition of minor servo commands. Addition of “Start Nuraghe from Scratch” appendix. Addition of clients: “Meteo”, “Scheduler”, “CalibrationTool”.
06	10/06/13	Corrections on receiversMode description.
05	31/05/13	Added details on the L/P receiver configurations.
04	23/05/13	Details on login and data access temporarily removed Correction to the setSection command description. Correction and more info on data quick-look

03	20/05/13	Added info on user login. Added info on quick-look. Added info on data-retrieval. Description of OTFC scanning option. Addition of system calibrator catalogue.
02	30/04/13	Updated info on commands. New formatting.
01	20/02/13	Issue 01

Overview

What DISCOS is

DISCOS is the control software produced for the Italian radio telescopes. It is a distributed system based on ACS (ALMA Common Software), commanding all the devices of the telescope and allowing the user to perform single-dish observations in the most common modes. As of today, the code specifically implemented for the telescopes (i.e. excluding the huge ACS framework) amounts to about 650000 lines. Even VLBI (or guest-backend) observations partly rely on DISCOS, as it must be used to perform the focus selection and the frontend setup.

This guide is meant to help the observer in the use of DISCOS, without dealing with the "behind-the-curtains" complex details of the system. This release focuses on **single-dish continuum and spectroscopy observations**; the fully integrated backends available on-site are the analog total power one, the XARCOS and SARDARA spectrometers. Since both the hardware and software implementations are still going on at SRT, this manual will forcibly undergo continuous revisions.

Here follows a simple schematization of the observing system, helpful to visualize all the main devices DISCOS deals with and the most important operations it performs.

Note: (1) Frequency switching is possible only manually editing the schedules, as the schedule-creating tool does not foresee this specific mode, yet.

Glossary: terms and abbreviations

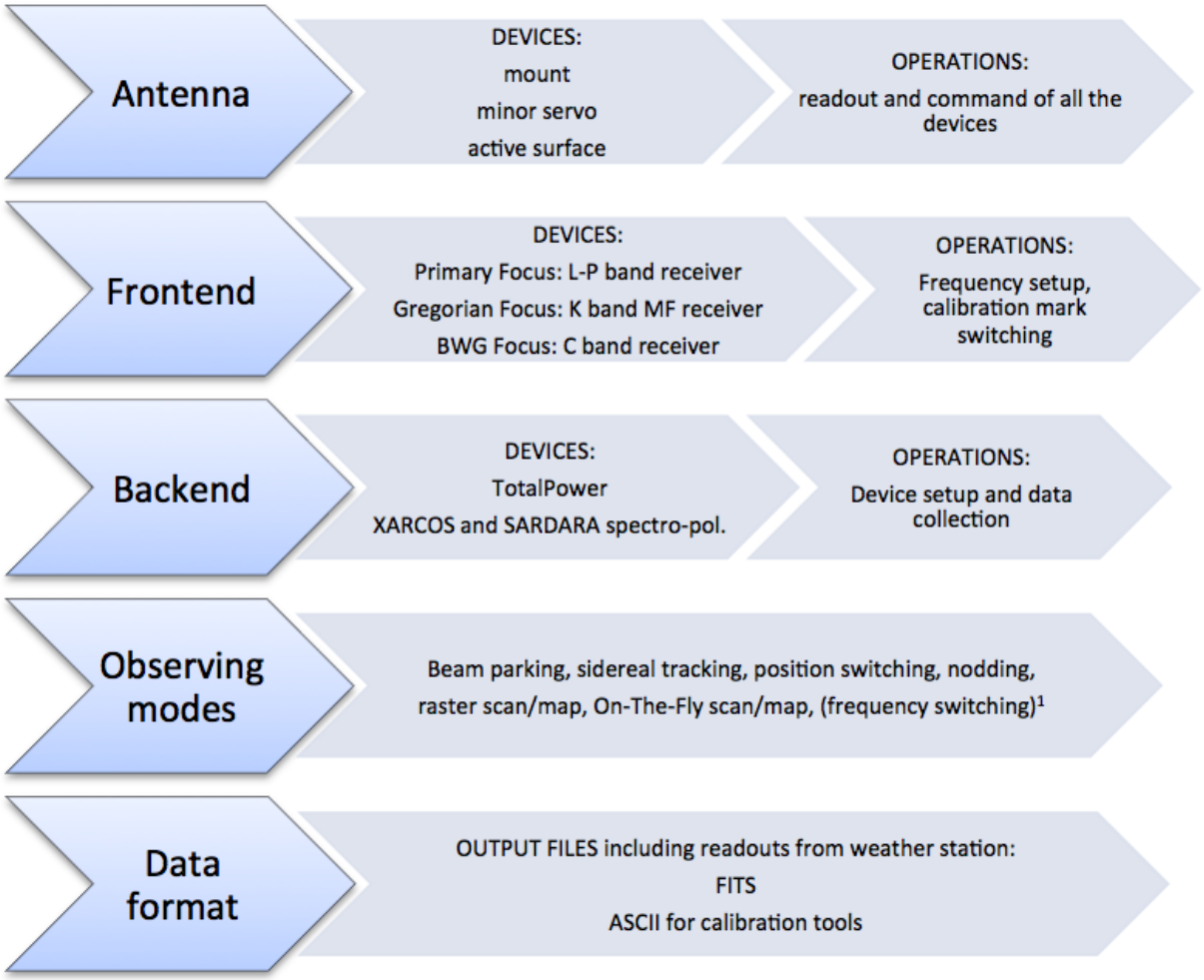
> this symbol precedes commands to be given in the operatorInput panel

$ this symbol precedes commands to be given in a system shell

AS active surface

Beam-parking the antenna points to a fixed Azimuth-Elevation position

Bin frequency channel


Antenna
DEVICES:
mount
minor servo
active surface
OPERATIONS:
readout and command of all the devices
Frontend
DEVICES:
Primary Focus: L-P band receiver
Gregorian Focus: K band MF receiver
BWG Focus: C band receiver
OPERATIONS:
Frequency setup, calibration mark switching
Backend
DEVICES:
TotalPower
XARCOS and SARDARA spectro-pol.
OPERATIONS:
Device setup and data collection
Observing modes
Beam parking, sidereal tracking, position switching, nodding, raster scan/map, On-The-Fly scan/map, (frequency switching)1
Data format
OUTPUT FILES including readouts from weather station:
FITS
ASCII for calibration tools

FWHM Full-Width Half-Maximum

HPBW Half-Power BeamWidth

MF Multi-Feed

MS Minor Servo system, which includes the subreflector and the PFP

Nodding (aka "beam switching") it involves two or more feeds. The source is alternatively observed with each of the feeds, so that there is always one feed "on source"

OTF On-The-Fly acquisition. The antenna moves according to user-defined parameters, scanning the sky at constant speed. Data acquisition is active during the scan

PFP Primary Focus Positioner, the servo which allows to put the primary focus receivers in place

Position switching (aka "on-off") the beam is alternatively pointed to the source (on) and to a properly defined offset position (off)

Preset mount mode allowing only beam-parking observations, with no pointing model applied

ProgramTrack mount mode allowing tracking, OTF/raster scans, beam-parking

Raster acquisition is performed via discrete pointings (in tracking or beam-parking mode), planned to sample a certain path/area on the sky

Section acquisition stream (aka "logical channel"). E.g.: the 7-feeds of the K-band receiver used with the continuum backend produce 7 feeds x 2 polarisations = 14 sections

SF Single-Feed

Slewing motion of the antenna when it is going to the target position. Slewing is always performed at maximum speed, with no data acquisition taking place

Temporised commands user-input or schedule commands that are launched to a specific UT time by means of the "@" symbol

Time-tagged commands when commands are not temporised, the philosophy for their execution is asap (as soon as possible), but the system internally schedules them associating each operation to a properly computed time. This is particularly relevant for the positions commanded to the mount, so that the antenna motions take place along a fully controlled path (except in slewing)

TPB Total Power Backend

Tracking the antenna points to a target and, if it is a celestial target, follows its sidereal motion

Checklist for schedule-based observations

Notice that actions take place in three different "locations":

- **(dc)** = action to be performed in a terminal on the observing machine (viewer02), within the VNC connection to *discos-console*
- **(op)** = command to be given in the *operatorInput* panel of DISCOS

Login on (dc)

Launch the monitors, if necessary (dc):

```
$ discosConsole
```

Initial setup (op):

```
> antennaReset  (if resuming after the emergency stop button is released)

> setupCCB      (or other receiver code)

> asSetup=S     (or other AS code)
```

Project declaration (op):

```
> project=[projectID]    (to correctly store all your data and more easily retrieve␣
↪schedules)
```

Tune the local oscillator, if any (op):

```
> setLO=[freq]
--> e.g. setLO=6600 - start frequency of the observed band will depend on the backend
```

Point the antenna to a reference position (op):

```
> goTo=[Az]d,[El]d
--> e.g. goTo=180d,45d
```

Always explicitly choose the Total Power backend (a spectrometer might have been left active) (op):

```
> chooseBackend=BACKENDS/TotalPower
```

Measure the cold sky signal level (op):

```
> getTpi
> setAttenuation=[sect],[att]
--> iteratively adjust attenuations until the level is in the 800-1000 count range
```

Get a Tsys (op):

```
> tsys
```

Pointing and focusing optimisation (op):

```
> track=name                    (choose a proper calibrator from source catalogue)
> chooseRecorder=MANAGEMENT/Point
--> the following command on (1):
        $ calibrationtoolclient MANAGEMENT/Point            (to display the plots)
> crossScan=HOR,0.5d,00:00:20   (set proper parameters according to your beamsize)
> azelOffsets=0d,0d             (if wanting to reject the measured offsets)
> focusScan=60,00:01:00
> clearServoOffsets             (if wanting to reject the updated focus position)
```

WARNING:

- **System offsets, such as the ones measured with a Point acquisition, sum up to the ones indicated inside schedules ONLY if they are expressed in the same coordinate frame.**

This means that, if you perform observations using EQ offsets, also the fine-pointing cross-scans must be carried out in the EQ frame. The same holds for HOR scans. If there is a frame mismatch, the system offsets are automatically rejected (bug under fixing).

If needed, choose and set the spectrometer (op):

```
> chooseBackend=XARCOS  (or SARDARA)
> initialize=[code]
```

Create a schedule (dc):

```
Use the *basie* schedule creator (read its guide for details):
$ basie -c [configfile] [out_directory]
```

Copy all the schedule files (dc):

```
Copy the schedule to the destination folder on discos-console.
```

Parse the schedule (dc):

```
$ scheduleChecker [schedulename].scd
```

Launch the schedule (op):

```
> startSchedule=[project/][schedulename].scd,[N]
```

Data quick-look (dc):

```
Launch the real-time quick-look of the data under acquisition
by double-clicking on the quicklook.html icon on the desktop
```

Stop the schedule (op):

```
> stopSchedule
```

Copy the data (dc):

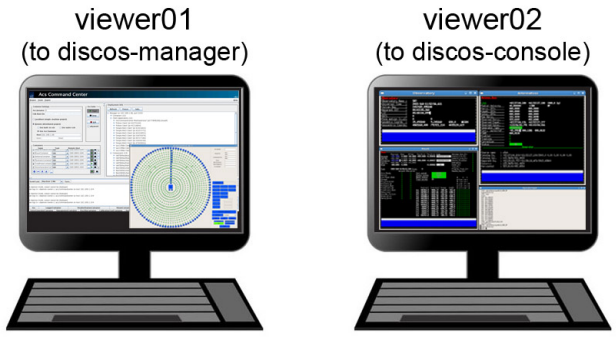
```
Get your data, see :ref:`Retrieving-the-data`
```

Stow the antenna (op):

```
> telescopePark
```

DISCOS startup

Login



Note: **passwords are provided by the project friend**. As the control room logistics might be updated, be sure to contact the project friend before your session starts, in order to get the latest information.

What to use viewer01 for

This machine is accessed to manage the ACS Command Center and the Active Surface GUI. Under ordinary conditions it is used to monitor the AS status, plus it is to be accessed to reboot discos/ACS whenever needed. You can login with:

```
username: discos       pwd:  **********
```

If VNC is not active, open a terminal and use the vncStart command, which is automatically going to connect to the **discos-manager** machine:

```
login: discos     pwd: **********
```

What to use viewer02 for

Observations take place using the **viewer02** machine, in turn exploiting a VNC connection to the **discos-console** machine.

You can login with:

```
username: observer       pwd:  **********
```

If VNC is not active, open a terminal and use the vncStart command, which is going to connect to the **discos-console** machine:

```
login: observer     pwd: **********
```

The VNC desktop might be larger than the monitor; you can scroll horizontally using the bottom scrollbar. Beware that, when launching programs, they might open windows in the invisible desktop area!

Once logged in on discos-console, *you should find the input terminal and all the monitors already running*. If this is not the case, open a new terminal, then:

```
$  discosConsole
```

This command opens 9 panels at once:

- **operatorInput** - terminal for command line input
- **AntennaBoss**
- **GenericBackend**
- **Mount**
- **Observatory**
- **Receivers**
- **Scheduler**
- **MinorServo**

- **ACS Custom Logger**

Rearrange the panels on the desktop according to your preferences. In case any of them does not automatically start, or if you close any of them by accident, you can manually open them by means of individual command lines, to be given in the open terminal:

```
$ operatorInput
$ antennaBossTui
$ genericBackendTui BACKENDS/TotalPower  (or XARCOS, or SARDARA)
$ mountTui
$ observatoryTui
$ receiversBossTui
$ schedulerTui
$ minorservoBossTui
$ loggingDisplay
```

All the antenna/receiver/backend setup procedures are performed via the operatorInput window, which is also used to start/stop the schedules.

The other panels are monitors used to display a vast amount of information, see *Appendix A - Monitor panels full description* and *Appendix B - Complete command list* for a comprehensive description of their content and a list of all the commands available for the operatorInput (they can be inserted in schedules as well).

The operatorInput panel supports "TAB-completion". For example (op):

```
> f<TAB>
flush      flushAll     focusScan

> fo<TAB>
> focusScan
```

Online help is available:

```
> help(focusScan)

   focusScan=span,duration
   commands a focus scan on a previously selected target
   (by means of the track or sidereal commands).
   Parameters:

   * span: the length run on the z-axis expressed in mm,
   * duration: the time length espressed in hh:mm:ss,

   e.g. > focusScan=60,00:01:00
```

TAB-completion is active for the online help as well:

```
> help(fo<TAB>
> help(focusScan
```

The help() command shows the complete command list:

```
> help()

   antennaPark
   antennaReset
   antennaSetup
   ...
```

(continues on next page)

(continued from previous page)

```
tsys
wait
wx
```

Note: **discos-console** is also the location where users need to copy their schedules to, and where they can retrieve their data from. See section *Retrieving the data* for details.

Initial setup

Antenna overall setup

When opening a DISCOS observing session, it is necessary to perform a setup which includes the antenna unstow, the mount configuration in tracking mode, the minor servo setup. This is done by means of a unique command, which is specific for the wanted receiver, to be written in the **operatorInput**. The currently available choices are:

```
> setupLLP  for the L band receiver (Primary focus)
> setupPPP  for the P band receiver (Primary focus)
> setupCCB  for the C band receiver (BWG focus)
> setupKKG  for the K band receiver (Gregorian focus)
```

Note: Spaces within the command line content are **not** allowed!

The above setup command sets the antenna mount, the minor servos, the selected receiver and the default backend (TotalPower) according to **default parameters**. The antenna mode is set to **ProgramTrack** (allowing tracking and the execution of schedules), while the Local Oscillator frequency and the bandwidth are set as illustrated in the following table.

Receiver code	LO frequency (MHz)	Frontend IF band (MHz)	Backend IF band (MHz)	Observed bandwidth (MHz)	Observed band (MHz)
LLP	(0)	1300-1800	(50-2350)	500	(1300-1800)[1]
PPP	(0)	305-390	(50-730)	85	(305-390)
CCB	5600	100-2100	50-2350	2000	5700-7700
KKG	21964	100-2100	50-2350	2000	22064-24064

[1]

Notice that, depending on the devices in use, the sky frequency at the observed band starting point is given by the LO frequency plus an offset. For the present combinations of the frontends *with the total power backend*, which the above table refers to, this offset is 100 MHz. In general, the true observed band depends on the **intersection between the frontend IF band and the chosen backend filter**. The actual observed bandwidth and the band starting frequency are recorded in the output files (see *Appendix C - Output files*).

Active Surface setup

After the initial setup, to configure and enable the active surface (AS) give the following commands in the operatorInput panel:

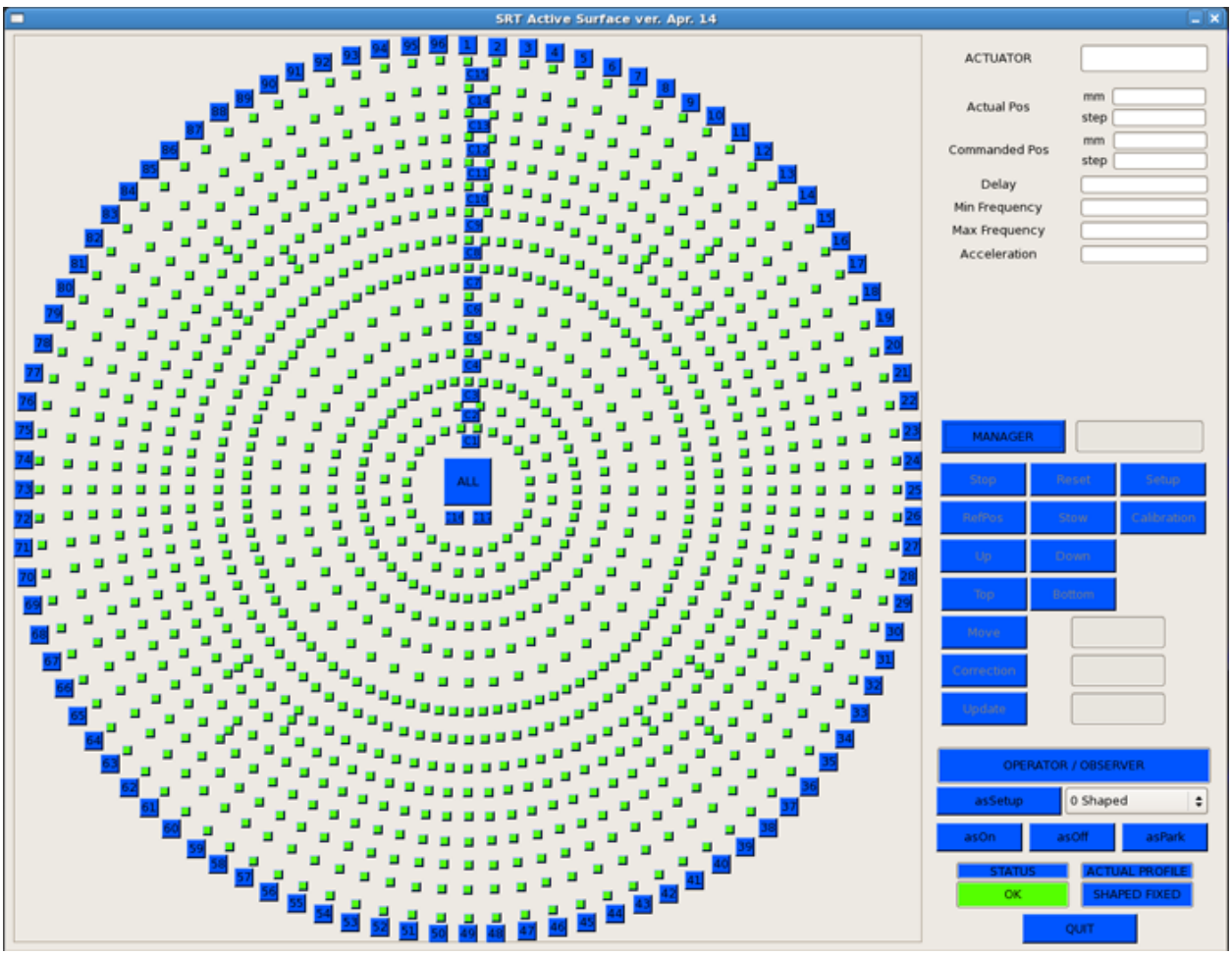
[1] Still subject to variations

```
> asSetup=[code]
```

where [code] is:

- **SF** shaped configuration in fixed position (optimised for El=45°)
- **S** shaped configuration in tracking – i.e. it adjusts according to the observed elevation position
- **PF** parabolic configuration in fixed position (optimised for El=45°)
- **P** parabolic configuration in tracking – i.e. it adjusts according to the observed elevation position

Warning: Check the AS status! In case the system - or just the AS - was rebooted, pay attention to give the `asSetup` command only once all the AS actuators are online and ready. You can check this condition in the GUI shown below. Red pixels in the graphical representation of the AS indicate off-line actuators, while green pixels identify the on-line actuators.



Minor Servo configuration

When using the Gregorian or BWG focus only, the *Antenna overall setup* automatically configures the minor servo system in order for the elevation to be tracked, and assumes the active surface is enabled. That means the minor servo

setup automatically issues the following commands:

```
> setServoElevationTracking=ON
> setServoASConfiguration=ON
```

The user is asked to issue the correct configuration in case other behaviours are required.

Logfile and project code

The default logfile is named **station.log**. If the user wants to change it:

```
> log=[logfilename]  (without extension)
```

Logfiles are stored in a dedicated folder (see *Retrieving the data*). **When schedules are run, a new logfile is automatically started**, and it is named after the schedule: [schedulename].log.

It is possible, and advisable, to insert the project code/name (a string assigned to the project by the TAC) using the command:

```
> project=[projectcode]   (e.g. project=scicom)
```

This will make the user save time in later stages, as it will not be necessary to specify the project name in schedule-launching commands. The project code/name must correspond to an existing user, already known to the system. This means that, if its spelling does not match with the recorded name, an error rises.

Antenna operations

Besides the overall telescope setup previously described, individual commands are available to change the antenna mount status and manage its steering/pointing:

> antennaReset
: resets the antenna status after a failure, for example after the emergency stop button is released

> antennaUnstow
: it only performs the unstow procedure

> antennaSetup=[code]
: *code*: LP - for both L and P bands -, CCB, KKG

  it unstows the antenna (if it is stowed) then it sets the pointing model and the minor servo system according to the selected receiver. Mount is set to ProgramTrack mode

> servoSetup=[code]
: *code*: LLP, PLP, PPP, CCB, KKG

  it sets the minor servo system only, without affecting other systems

> antennaTrack
: it sets the mount to ProgramTrack mode, allowing the execution of sidereal tracking, on-off, OTF scans, etc. Even beam-parking acquisitions (i.e. on a fixed Az-El position) can be now performed in ProgramTrack mode by means of the goTo command

> goTo=[Az]d,[El]d
: it points the antenna to fixed Az-El positions, yet the mount remains in ProgramTrack mode

  e.g. *> goTo=180d,45d*

> track=[sourcename]

if the antenna is in ProgramTrack mode and the sourcename is known within the station catalogue (which includes the most commonly observed calibrators), it directly points to the source and tracks it

e.g. `> track=3c286`

> sidereal=[sourcename],[RA],[Dec],[epoch],[sector]

if the antenna mode is ProgramTrack, it points to the supplied RA-Dec position and temporarily assigns the sourcename label to it. Epoch can be `1950`, `2000` or `-1`, the last one meaning that the provided coordinates are precessed to the observing epoch. The sector keyword forces the cable wrap sector, if needed: its value can be `CW`, `CCW` or `neutral`. The last option means the system will automatically choose the optimal alternative

e.g. `> sidereal=src12,319.256d,70.864d,2000,neutral`

Note: COORDINATE FORMATS Whenever celestial coordinates (Equatorial, Horizontal or Galactic) are specified, the allowed formats are:

- **decimal degrees**, using a `d` suffix, for any coordinate → e.g. `30.00d`
- **sexagesimal degrees**, with no suffix, for any coordinate → `30:00:00`
- **hh:mm:ss**, with a `h` suffix, for longitudes only → `02:00:00h` (not accepted for offsets)

Back to the commands:

> goOff=[frame],[beams]

it slews the antenna to an offset position, wrt a previously commanded target, along the longitude axis of the indicated coordinate frame (`EQ`, `HOR` or `GAL`). The user provides the offset value expressed in beamsizes. If the frame is HOR and target lies beyond the Elevation cutoff limits, the offset is applied in Elevation.

e.g. `> goOff=HOR,5`

> azelOffsets=[azoff]d,[eloff]d

it sets user-defined offsets in the Horizontal frame (degrees only). The following example sets an azimuth offset to 0.5 degrees and the elevation offset to 0.3 degrees

e.g. `> azelOffsets=0.5d,0.3d`

> radecOffsets=[raoff]d,[decoff]d

it sets user-defined offsets in the Equatorial frame (degrees only). The following example sets the right ascension offset to 0.3 degrees and the elevation offset to 0.0 degrees

e.g. > radecOffsets=0.3d,0.0d

> lonlatOffsets=[lonoff]d,[latoff]d

it sets user-defined offsets in the Galactic frame (degrees only). The following example sets the galactic longitude offset to 0.1 degrees and the galactic latitude offset to 0.5 degrees

e.g. `> lonlatOffsets=0.1d,0.5d`

Note: OFFSETS: the above user-defined offsets are the overall antenna offsets and they are mutually exclusive! If the user commands the offsets several times in a row (in one or different frames) only the last one will be effective. **Offsets specified within schedules, at subscan level, sum up to these user-defined offsets, but only if they are expressed in the same frame.** Otherwise, the overall (or "system") offsets are zeroed.

On with the list:

> setServoOffset=[axis_code],[value] (→ for technical

activities)

The [axis_code] argument must be one of the following codes:

- SRP_TX # SRP translation along the X axis (mm)
- SRP_TY # SRP translation along the Y axis (mm)
- SRP_TZ # SRP translation along the Z axis (mm)
- SRP_RX # SRP rotation around the X axis (arcsec)
- SRP_RY # SRP rotation around the Y axis (arcsec)
- SRP_RZ # SRP rotation around the Z axis (arcsec)
- PFP_RY # PFP rotation around the Y axis (arcsec)
- PFP_TX # PFP translation along the X axis (mm)
- PFP_TZ # PFP translation along the Z axis (mm)
- GFR_RZ # GFR rotation (mm)
- M3R_RZ # M3R rotation (degrees)

The [value] argument is a mm value that is assigned to the offset. For instance, in order to set a 5mm offset to the subreflector Z axis:

```
> setServoOffset=SRP_TZ,5
```

> antennaStop

it stops the antenna motion, if any, and changes the mount mode to Stop

> antennaPark

it stows the antenna

> asPark

it parks the active surface in the reference position for El=45°

> servoPark

it stows the minor servo system (notice: after antennaPark, always give this command)

> telescopePark

it parks all the elements: mount (sending the antenna to stow position), minor servo and active surface

Frontend operations

General rules

To change the frontend Local Oscillator frequency, use the following command:

```
> setLO=[freq1];[freq2];...;[freqN]
```

Notice the semicolon. Ideally, different values could be assigned to different IFs, thus tuning each section to a different sub-band. For the TPB, though, this is not possible, so a single value must be specified:

```
e.g. ``> setLO=5600``
```

Remember that the actually observed band begins at a frequency which is usually different from the LO one (see *Antenna overall setup*)

For the L-P band receiver, filters are available in order to set the observed sub-band and select the polarisation (circular or linear), specifying a code with the command:

```
> receiversMode=[code]    (see the next section for the details)
```

The temperature of the calibration mark in use is recorded in the logfile whenever a Tsys is measured. It is also stored in the FITS/MBFITS output files. The calibration mark can be manually switched on and off respectively with:

```
> calOn
> calOff
```

If users want to perform the setup for the frontend only (without affecting the mount, the minor servo or the backend), the command is:

```
> receiversSetup=[code]   (PPP, LLP, PLP, CCB, KKG)
```

Specific notes for the L/P receiver

The L/P receiver is actually seen by DISCOS as a group of three different receivers, each having its own setup code:

- **PPP** for the usage of the P band only
- **LLP** for the usage of the L band only
- **PLP** for the dual-feed usage

These are the codes to be used in the *setupXXX* command. Then the configuration of the band filter and of the polarisation mode must be performed, by means of the *receiversMode* command. Here follows the list of options.

PPP

Only the P band can be used. The possible configurations are:

```
> receiversMode=C1XX
> receiversMode=C2XX
> receiversMode=C3XX   (default)
> receiversMode=L1XX
> receiversMode=L2XX
> receiversMode=L3XX
```

The first two characters in the code identify the polarisation mode C = Circular, L = linear) and the band filter index, where:

1. all band, 305-410 MHz (no filter)
2. 310-350 MHz
3. 305-410 MHz (band-pass filter)

The **XX chars are placeholders** for the L-band configuration, which is not allowed in PPP mode.

LLP

Only the L band can be used. The possible configurations are:

```
> receiversMode=XXC1
> receiversMode=XXC2
> receiversMode=XXC3
> receiversMode=XXC4   (default)
> receiversMode=XXC5
> receiversMode=XXL1
> receiversMode=XXL2
> receiversMode=XXL3
> receiversMode=XXL4
> receiversMode=XXL5
```

The last two characters in the code identify the polarisation mode (C = Circular, L = linear) and the band filter index, where:

1. all band, 1300-1800 MHz (no filter)
2. 1320-1780 MHz
3. 1350-1450 MHz (VLBI)
4. 1300-1800 MHz (band-pass)
5. 1625-1715 MHz (VLBI)

The **XX chars are placeholders** for the P-band configuration, which is not allowed in LLL mode.

Local Oscillator for L-band observations

Starting from the release of DISCOS 0.6, it is possible to use the `setLO` command for this receiver, in the operator input console:

```
> receiversSetup=LLP
> setLO=2300
```

This new feature is only available using the L-band feed (it is not possible to set the LO in the `PPP` configuration).

It is not allowed to set the LO inside the observed sky band. For instance:

```
> receiversMode=XXL5    (Bandwidth 1625:1715)
> setLO=1650
Error - Value out of legal ranges
```

It is also not possible to set a receiver mode in case the current LO value is inside the mode band:

```
> receiversMode=XXL5    (Bandwidth 1625:1715)
> setLO=1500            (Ok, it is outside the RF band)
> receiversMode=XXL4    (Bandwidth 1300:1800)
Error - Value out of legal ranges
```

Notice that, for this receiver, the LO frequency is higher than the observed frequency. For this reason the bandwidth value displayed in the `Receivers` monitor is positive while the IF is negative, indicating that the IF band is inverted. For instance, in the following case:

```
> receiversMode=XXL5    (Bandwidth 1625:1715)
> setLO=2300
```

the receiver panel will show `startFreq=-675` and `bandwidth=90`. The band initial frequency can be obtained as LO+startFreq.

A 1000 MHz low-pass filter is applied to the IFs, so it is not possible to set a LO value that brings the IFs outside the range of the low-pass filter:

```
> receiversMode=XXL4    (Band 1300:1800)
> setLO=200             (IF band: 1100:1600)
Error - Value out of legal ranges
```

IF distributor

The SRT has a new device (IFDistributor) capable to perform down conversion and IF filtering when the L-band receiver is configured. The control of the local oscillator and the down conversion is transparent and can be done as already described. In particular, all the caveats regarding the limits still hold. In case a filtering of the IF bandwidth is required (e.g. when observing with the DFB3) the command `ifd` must be used:

> ifd=[filterid]

FILTER ID	Central Freq (MHz)	Bandwidth (MHz)
BW-NARROW	576	115
BW-MEDIUM	640	230
BW-WIDE	768	460
BW-UNFILTERED	see L-Band	

PLP (dual band)

In this setup, both bands are used and can be configured, according to the following scheme, based on the permutation of all the previously described codes:

```
> receiversMode=C1C1
> receiversMode=C2C1
> receiversMode=C3C1
> receiversMode=C1C2
...
> receiversMode=C3C4  (default)
> receiversMode=C3C5
...
> receiversMode=L1C1
...
> receiversMode=C1L1
...
> receiversMode=L1L1
...
```

Backend operations

The backend to be used can be manually selected as follows:

```
> chooseBackend=BACKENDS/[bckname]
```

where *bckname* is the name of the backend. The currently available choices are:

- TotalPower
- Xarcos (for the XARCOS spectrometer)

- Sardara (for SARDARA, the ROACH-based spectrometer)

Total Power - focus selector

Warning: The TotalPower backend works as a **"focus selector"**, sending the signal from the wanted receiver to any other backend. For this reason, it **must** be set up even when acquisitions take place using another backend. This is accomplished simply using the overall setup commands (such as `setupCCB`, etc...).

Bandwidth and sampling rate can be changed using:

```
> setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]
```

where:

- *[sect]* is an integer specifying the section number
- *[startFreq]* is the initial frequency for the section (not applicable for TPB)
- *[bw]* is a double for the bandwidth (MHz)
- *[feed]* is the number of the feed connected to the section (not applicable for TPB)
- *[mode]* is the polarisation mode (not applicable for TPB)
- *[sampleRate]* is also given in MHz
- *[bins]* is the number of frequency bins for the given section (not applicable for TPB)

To leave a parameter at its previously set value, or equivalently skip it when it is not applicable, use an asterisk. For the TPB, in particular, always use:

```
> setSection=[sect],*,[bw],*,*,[sampleRate],*
```

where *bw* can be chosen from a restricted range of options (MHz):

- 300.0
- 730.0
- 1250.0
- 2000.0

These values do not correspond to the true observed bandwidth, for the reason discussed in Initial Setup. When accumulations of the data "dumps" are required before integrating them to the output file, it is possible to set the integration time as follows:

```
> integration=[N]
```

where *N* is given in milliseconds.

Each section is provided with an attenuator, to manage the signal intensity. To set the desired attenuation:

```
> setAttenuation=[sect],[att]
```

where *sect* is the section number and *att* can vary, for the TPB, form 0 dB to 15 dB, with a 1 dB step.

If a backend re-initialization is needed, use:

```
> initialize=[code]
```

(*code* can be *CCB*, *KKG*, etc)

This command resets the backend only, setting it to the default values foreseen for the specified focus/receiver. After the frontend/backend configuration is changed, it is necessary to update the value for the beamsize (HPBW), which is computed as a function of the actually observed band and is used – during the observations – to evaluate the pointing accuracy. This is accomplished using the command:

```
> device=[sect]
```

which uploads to the system the parameters relative to section number [sect] (you can generally use 0, which exists for all the receivers, unless you need to observe only with a different feed).

XARCOS

This spectrometer is fully integrated in DISCOS, which means that DISCOS can command the device, receive data from it and then write these data using a standard output (FITS format).

In order to configure the spectrometer, select it with the *chooseBackend* command and use one of the following commands in the operatorInput:

> initialize=XK77
: This is to use the full K-band MF; each digital sample has a 6-bit representation. Full-Stokes sections are recorded, each having a 62.5 MHz bandwidth and 2048(x4) channels

> initialize=XK03
: It exploits the K-band feeds 0 and 3. Each feed produces two full-Stokes sections respectively having bandwidths of 62.5 MHz and 4 MHz and 2048(x4) channels. Each digital sample has an 8-bit representation.

> initialize=XK06
: It enables the K-band feeds 0 and 6. Each feed produces two full-Stokes sections respectively having bandwidths of 62.5 MHz and 4 MHz and 2048(x4) channels. Each digital sample has an 8-bit representation.

> initialize=XK00
: This configuration is for the usage of the K-band central feed. It produces four full-Stokes sections respectively with bandwidths of 62.5 MHz, 8 MHz, 2 MHz and 0.5 MHz, each having 2048(x4) channels. Each digital sample has an 8-bit representation.

> initialize=XC00
: This configuration is C-band usage. It produces four full-Stokes sections respectively with bandwidths of 62.5 MHz, 8 MHz, 2 MHz and 0.5 MHz, each having 2048(x4) channels. Each digital sample has an 8-bit representation.

Ideally, configuration details can be changed using the setSection command:

```
> setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]
```

where:

- *[sect]* is an integer specifying the section number
- *[startFreq]* is the initial frequency for the section
- *[bw]* is a double for the bandwidth
- *[feed]* is the number of the feed connected to the section
- *[mode]* is the polarisation mode

- *[sampleRate]* is also given in MHz
- *[bins]* is the number of frequency bins for the given section

However, the present implementation allows the user to change only part of these parameters, in particular:

- *[startFreq]* – initial frequency, it must be in the 125-250 MHz range. This value, added to the LO frequency, gives the lowest sky frequency observed by the section. Different sections can have different start frequencies (contrarily to what happens for the TPB);
- *[bw]* – bandwidth must be chosen from the following values (all are expressed in MHz): 125.0, 62.5, 31.25, 15.625, 7.8125, 3.90625, 1.953125, 0.9765625, 0.48828125;
- *[sampleRate]* – its value (MHz) must be twice the bandwidth.

Users must not change the feed, mode and bins parameters, which are fixed as described in the configuration defaults.

A valid example of setup and setSection usage, then, is:

```
> chooseBackend=BACKENDS/Xarcos
> initialize=XC00
> setSection=0,155.0,31.25,*,*,62.5,*
```

where asterisks indicate which parameters are to be set according to default values.

Warning: At present, integration time is equal to **10 seconds**. Shorter integrations might be available in the future. Data transfer requires about **2 seconds** for each integration; take this overhead into consideration when estimating how long your schedules are going to last.

SARDARA

This is a ROACH-based spectrometer, now fully integrated in DISCOS. In order to configure it, select it with the chooseBackend command as described above and use one of the following commands in the operatorInput:

> initialize=SP00
This is to use P-band receiver in L/R mode

> initialize=SP00S
This is to use P-band receiver in full Stokes mode

> initialize=SL00
This is to use L-band receiver in L/R mode

> initialize=SL00S
This is to use L-band receiver in full Stokes mode

> initialize=SC00
This is to use C-band receiver in L/R

> initialize=SC00S
This is to use C-band receiver in full Stokes mode

> initialize=SK00
This is to use K-band receiver (central feed only) in L/R

> initialize=SK00S
This is to use K-band receiver (central feed only) in full Stokes mode

> initialize=SK03
This is to use K-band receiver (feeds number 0 and 3) in L/R

> initialize=SK03S
This is to use K-band receiver (feeds number 0 and 3) in full Stokes mode

> initialize=SK06
This is to use K-band receiver (feeds number 0 and 6) in L/R

> initialize=SK06S
This is to use K-band receiver (feeds number 0 and 6) in full Stokes mode

> initialize=SK77
This is to use K-band receiver (all the 7 feeds) in L/R

> initialize=SK77S
This is to use K-band receiver (all the 7 feeds) in full Stokes mode

By default, the spectral bin number is set to 1024 and the filter bandwidth to 1500 MHz (except for the RP00 configuration, which uses a 500 MHz bandwidth). The actually observed bandwidth and frequency range are reported in the FITS files (see the dedicated documentation).

Users can change the frequency bin number to 16384, using:

```
> setSection=[sect],*,*,*,*,*,16384
```

Where *sect* is the section number. Users can also change the integration time:

```
> integration=[N]
```

where *N* is given in milliseconds. Attenuations can be handled with the command:

```
> setAttenuation=[sect],[att]
```

where *att* can vary form 0 dB to 15 dB, with a 1 dB step. This attenuation is actually applied at the focus selector level.

Command-line measurements and acquisitions

Once the system and telescope setup had been completed, it is possible to manually perform measurements and observations **through the TPB**, which might as well pave the way – as preliminary checks – to longer lasting sessions carried out via schedules.

Raw-counts readouts

The **raw counts readout** (called Tpi) of the signal can be obtained with:

```
> getTpi
```

The system reply consists in an array of values (one for each section). As concerns the TPB, it is important to ascertain that the Tpi lies within the 750-1000 counts range in order to keep your observations in the linear range of the backend and allow for fairly bright sources to stay within as well. If this requirement is not met, it is necessary to iteratively vary the attenuation for the needed sections and check the Tpi, until the signal intensity falls into the proper range.

As the signal level greatly varies with elevation, it is advisable to perform this operation in the elevation range that will be actually exploited during the observations or, as a general rule, at elevation=45°. Of course, the signal level is also greatly affected, especially at high frequencies, by weather conditions, therefore the attenuation tuning should be carried out again every time the conditions change.

When you are going to manually get the Tpi - in order to ascertain which attenuation values to use in your schedule - *remember to set the LO frequency and the bandwidth as they will be employed in the schedule*. After a setup command,

in fact, they are set to defaults; instead, if a schedule has been previously run, these values remain set as indicated in the last schedule readout.

Note: Tpi values include the Tp0 level (the internal, "without sky" level, which is roughly 200 counts). Thus, when reading your output files, which contain the (Tpi-Tp0) signal, you will see a baseline 200 counts lower.

Tsys

To measure the Tsys value:

```
> tsys
```

the system replies with N values, where N is given by the total amount of (input_lines x sections). When using the TotalPower backend, N is the number of sections (2 for single-feed receivers and 14 for the MF).

Note: The last measured Tsys value will be stored in the system and used, *if the FITS format is selected for data storage*, to get a counts-to-Kelvin conversion factor, in turn applied to all the following acquisitions, until a new Tsys is measured. The FITS file will contain the raw data (in counts) and also a table with the data stream calibrated (in K) using this counts-to-Kelvin factor.

Weather parameters

The **weather station** measurements can be retrieved with:

```
> wx
```

the reply will list ground temperature (°C), relative humidity (%), atmospheric pressure (hPa), wind speed (km/h). Updated values are available every 10 seconds.

It is also possible to display the atmospheric temperature and the wind parameters (including wind direction) using a graphic interface. Activate the meteo client by using the following command in a terminal on DISCOS-obs1:

```
$ meteoClient
```

The following window will appear, it provides self-explanatory information.

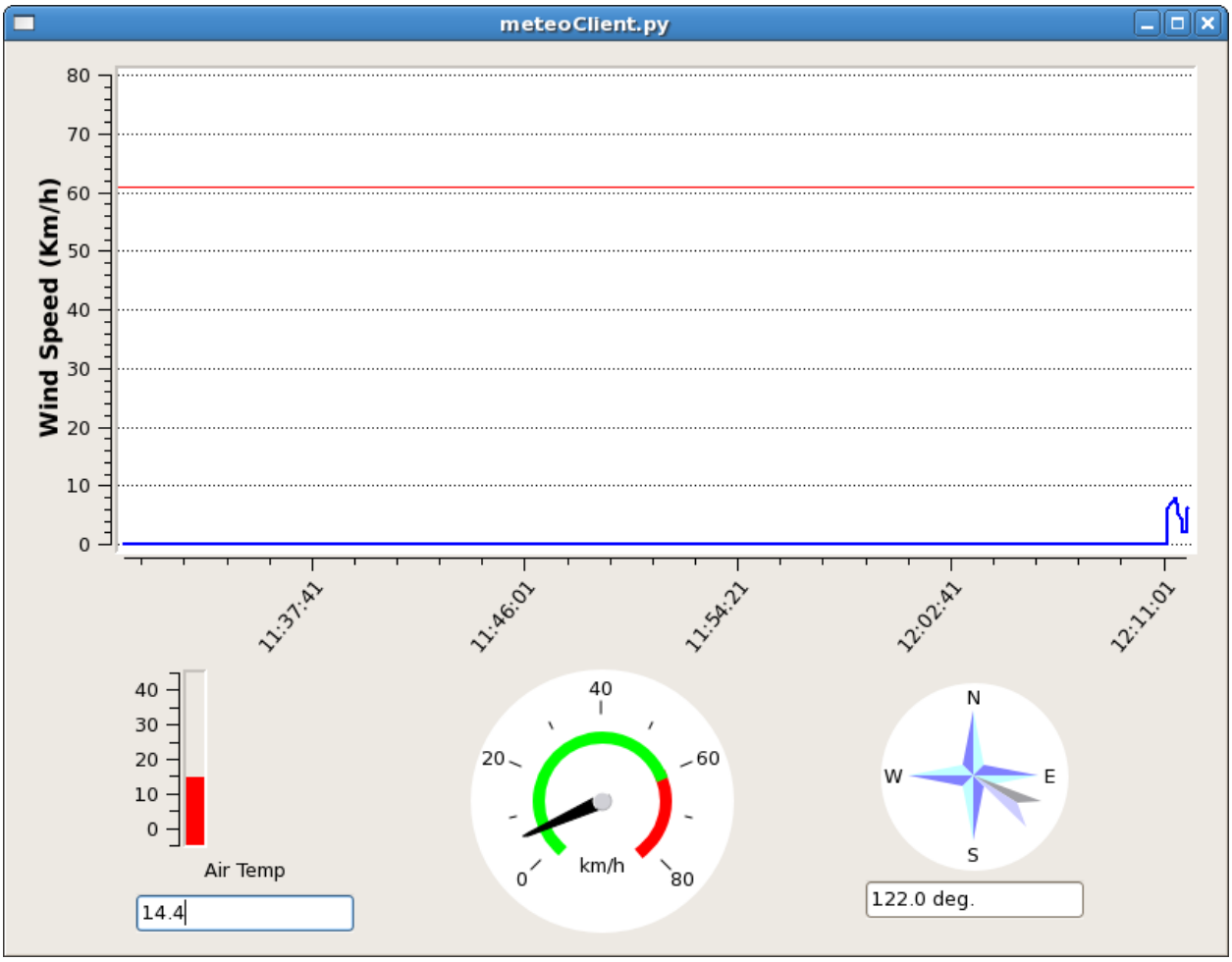
Notice the top graph: the red line corresponds to wind speed = 61 km/h, at which the antenna must be stowed.

Considering that weather parameters are not necessarily written in the output files (in particular, wind speed and direction are not stored in the FITS files at present), it is possible to record a separate log containing these parameters plus information on the antenna pointing. Open a terminal on DISCOS-obs1 and execute the following command:

```
$ windLogger
```

The script, besides displaying the information on screen, will produce a file which contains:

- coordinates pointed by the antenna;
- atmospheric temperature;
- wind speed and direction.


meteoClient.py
Wind Speed (Km/h)
80
70
60
50
40
30
20
10
0
11:37:41
11:46:01
11:54:21
12:02:41
12:11:01
40
30
20
10
0
Air Temp
14.4
40
20
60
0
80
km/h
N
W
E
S
122.0 deg.

These logfiles are stored in a dedicated folder on OBS1. See section *Retrieving the data*.

Filenames are assigned according to date and time of the script execution. For example, if the script is launched on November 26th 2013 at 13:46:31 UT, the resulting file will be: azel_131126_134631.log During the acquisition, the shell must not be closed. To interrupt the acquisition, use CTRL+C from the keyboard.

Wind park

For safety reasons, in case of excessively strong winds the system interrupts the activities and auto-stows the antenna ("wind park"). Do **not** unstow the antenna unless you have been given explicit permission by the support staff.

Manual acquisitions

When performing manually commanded acquisitions, it is necessary to **select the recording device**:

```
> chooseRecorder=[string]
```

where string can be:

- **MANAGEMENT/Point** (default) text output in the logfile, used for pointing calibration
- **MANAGEMENT/CalibrationTool** text output in .dat file, used for pointing calibration
- **MANAGEMENT/FitsZilla** if FITS output is desired
- (MANAGEMENT/MBFitsWriter) if MBFITS is preferred – not yet available

Once the recorder is set, acquisitions on a target can be performed as follows. First, set the target:

```
> track=[sourcename]     (if the source is included in the system
                         catalogue)
```

For the available catalogue see *Appendix D - Source catalogue*. To set a generic target:

```
> sidereal=[sourcename],[RA],[Dec],[epoch],[sector]
```

(see *Antenna operations* for details)

Here follow the commands to be used to manually record your data (remember that the backend must have been properly set up and the target must have been specified as explained above):

```
> initRecording=[scn]
```

where [scn] in the scan number to be assigned to the acquisition. The `initRecording` command prepares the data recording. Then:

```
> startRecording=[sub_scn],[duration]
```

creates the output file and begins the data recording; [sub_scn] is the subscan number, [duration] is the acquisition duration, expressed as `hh:mm:ss`. Once the acquisition is completed, the user can launch another subscan and record the data with another `startRecording` command.

Finally, once the user wants to close the scan, the command to be used is:

```
> terminateScan
```

Output files will be found in the usual auxiliary folder where all the manual acquisitions are destined.

Example: acquisition of a sidereal scan on 3c123 composed by 2 subscans, each lasting 40 s, preceded by an off-source Tsys measurement:

```
> chooseRecorder=MANAGEMENT/FitsZilla
> track=3c123
> goOff=HOR,5
> waitOnSource
> tsys
> azelOffsets=0.0d,0.0d
> initRecording=1
> startRecording=1,00:00:40
> startRecording=2,00:00:40
> terminateScan
```

Pointing scans

Command cross-scans across a previously selected target (by means of the track or sidereal commands):

```
> crossScan=[subscanFrame],[span],[duration]
```

where subscanFrame is the coordinate frame along which the scan is performed (*eq*, *hor* or *gal*), span is the spatial length on sky of the individual subscan (i.e. one line of the cross) expressed in degrees, duration i s the time length espressed in hh:mm:ss,

e.g. *> crossScan=HOR,1.0d,00:00:30*

corresponds to one cross-scan carried out in Horizontal coordinates (one line along El, one line along Az), each line being 1° in span. Each subscan lasts 30 seconds, thus the resulting scan speed is 2°/min.

When the MANAGEMENT/Point writer is used, the cross-scan produces text output in the logfile only (no output file is recorded). This output text contains information obtained by the automatic processing of the subscans. In particular, a Gaussian fit is performed in order to measure the source position and estimate the **pointing** offsets. If the fitting procedure in successful and the achieved offsets are considered plausible, pointing **corrections are immediately applied**. This means that, if no user-defined offset is commanded (or cleared) afterwards, the measured offsets remain active and are applied to the following observations.

Here follows the function that is separately fitted to latitude and longitude subscans:

y(x)=A*e^W + ax +c

where:

- W = -2.7725887 * F^2
- F = (x-μ)/FWHM
- μ = abscissa of peak

The results are given in the logfile, in the following sequence of lines:

LATFIT latoff fwhm A a c i
LONFIT lonoff fwhm A a c i
OFFSET avlon avlat lonoff latoff lonflag latflag
XOFFSET avlon avlat lonoff*cos(lat) latoff lonoff_err *(cont.)*
latoff_err lonflag latflag

XGAIN target avlon avlat lonampl lonampl_err latampl *(cont.)*
latampl_err lonFWHM lonFWHM_err latFWHM latFWHM_err *(cont.)*
flux lonflag latflag

where (all angles in degrees):

latoff = latitude offset
lonoff = longitude offset
i = number of iterations performed by the fitting procedure
avlon = average longitude of peak (in same coordinate frame as the subscan execution)
avlat = average latitude of peak (in same coordinate frame as the subscan execution)
lonflag = fit result for longitude subscans
(1 = plausible fit, 0 = non plausible fit, -1 = fit did not converge)
latflag = fit result for latitude subscans
(1 = plausible fit, 0 = non plausible fit, -1 = fit did not converge)
lonoff_err = error on longitude offset
latoff_err = error on latitude offset
target = target name
lonampl = amplitude measured on longitude subscans (K)
lonampl_err = error on amplitude measured on longitude subscans (K)
latampl = amplitude measured on latitude subscans (K)
latampl_err = error on amplitude measured on latitude subscans (K)
lonFWHM = FWHM measured on longitude subscans
lonFWHM_err = error on FWHM measured on longitude subscans
latFWHM = FWHM measured on latitude subscans
latFWHM_err = error on FWHM measured on latitude subscans
flux = catalogue target flux (Jy), if available (otherwise it is put to 0.0).

.

Note: it is possible to **include such pointing scans using the MANAGEMENT/Point writer in schedules** as well. For example, an improved pointing can be achieved setting the first scan on a source as a /Point scan, then – in case the fit is successful – the following scans (e.g. producing FITS or MBFITS files) will hold the offsets optimising the pointing, given that no user-defined offset is updated by means of an explicit `radecOffsets`, `azelOffsets` or `lonlatOffsets` command.

Focus scans

Command a focus scan on a previously selected target (by means of the track or sidereal commands):

```
> focusScan=[span],[duration]
```

where span is the length run on the z-axis expressed in mm, duration is the time length expressed in `hh:mm:ss`

e.g. `> focusScan=60,00:01:00`

The *focusScan* command can be used inside schedules as well. See the separate guide to schedules for details.

Skydips

Skydip scans are indispensable in order to characterize the atmosphere. They consist in moving the telescope along a vast span in elevation (at fixed azimuth) while sampling with a backend. Their analysis allows the user to quantify the atmospheric opacity τ. There are different ways to perform this:

```
> skydip=[El1],[El2],[duration]
```

e.g. *skydip=20d,80d,00:05:00* performs a skydip between 80 and 20 degrees (at the current azimuth position), the scan will take 5 minutes (speed is thus 12 °/min). The arguments must be in the range 10-88.

The jolly character is supported for the elevation arguments. Example: *skydip=*,*,00:04:00* will perfom the skydip between the default values for elevation (15° and 90°). Please notice that pointing corrections are disabled.

Since no backend recording is automatically enabled by this command, remember to activate the FitsZilla recorder before launching the command, in order to save the data! This command can be used within schedules as well. See the separate guide to schedules for details.

Note: At present skydips are always performed **downwards**, i.e. starting from the highest elevation given in the command. **The greatest commandable elevation is 88°**, since the skydip, being an OTF subscan, will be additioned of an initial acceleration ramp – the length of which is proportional to the scanning speed.

Caveat on offsets

As seen in *Antenna operations*, there are commands used to set (or null) user-defined offsets. They are: *radecOffsets*, *azelOffsets* and *lonlatOffsets*. Such commands set **overall offsets** which **remain active** until they are explicitly changed/nulled by another call of one of the three commands.

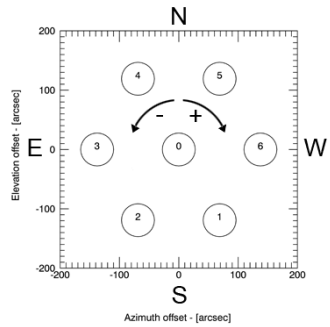
Further offsets, having for example the purpose of pointing the antenna to an off-source position, are specified inside schedules, at subscan level (see the separate guide to schedules). **Subscan-level offsets sum up to the system offsets ONLY if their coordinate frames coincide**, otherwise the system offsets are zeroed. In practice, when performing observations that imply the setting of offsets in a certain frame (EQ, HOR, GAL), the system offsets must have been set/achieved accordingly (e.g. performing Point scans in the same frame), or will be lost. Subscan-based offsets - the ones commanded via the schedule - are zeroed by default every time a new subscan is commanded.

Use of the MF derotator

The K-band 7-feed receiver is provided with a rotating dewar, in order to prevent field rotation during long acquisitions (while performing OTF/raster mapping or simple sidereal tracking). When the receiver is commanded a setup, this derotator is automatically set up in the default FIXED configuration - i.e. it will be placed in its rest position and it will not update its position along time.

Rest position

When the dewar angle is set to 0°,the MF receiver is positioned as displayed in the following figure. Feeds 3-0-6 are aligned along a line that is parallel to the horizon, with feed number 3 being on the right side of the central feed. Note that the cardinal directions refer to southern azimuth quadrants.



The static x-y offsets corresponding to this rest position, also provided in the output FITS files (see dedicated documentation), are:

Feed	x-offset (arcsec)	y-offset (arcsec)
1	68.800	-119.165
2	-68.800	-119.165
3	-137.600	0.000
4	-68.800	119.165
5	68.800	119.165
6	137.600	0.000

In order to be transformed in Az-El offsets, a cos(El) factor must be taken into account for the Azimuth (x) offsets.

Convention on angles

The dewar rotation is considered positive when moving clockwise.

Configurations

To select the derotator configuration, use the following command in the operatorInput panel:

```
> derotatorSetConfiguration=[conf]
```

where [conf] can have one of the following values: *FIXED*, *BSC*, *CUSTOM*. Here follows a brief description of these different options.

FIXED

The dewar keeps a fixed position wrt the horizon, i.e. no derotation is applied. It is possible to command a specific static angle:

```
> derotatorSetPosition=[ang]d
```

e.g. *> derotatorSetPosition=10.0d* rotates the dewar 10° clockwise

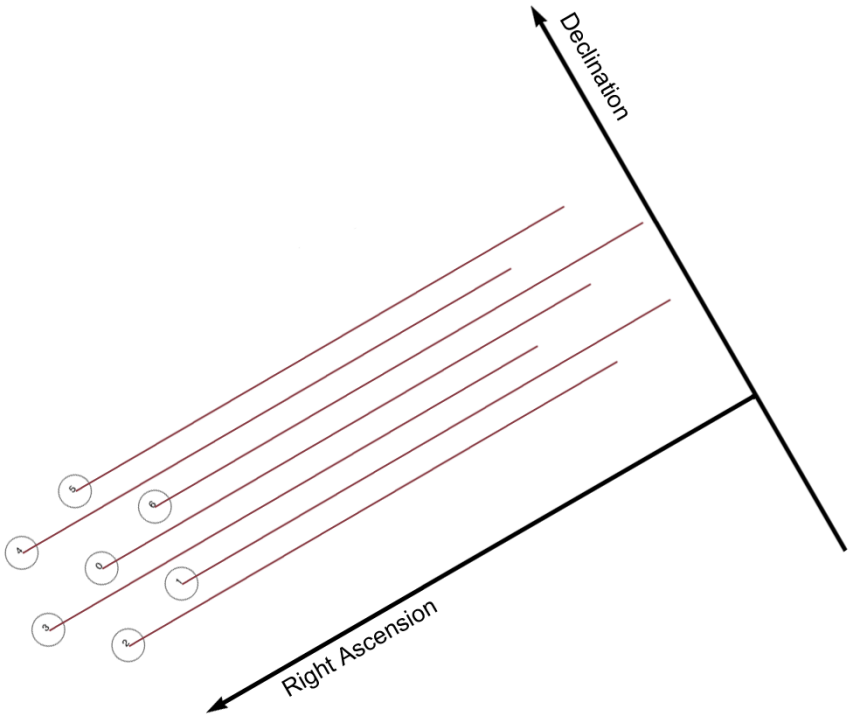
To read back the dewar position, use:

```
> derotatorGetPosition
```

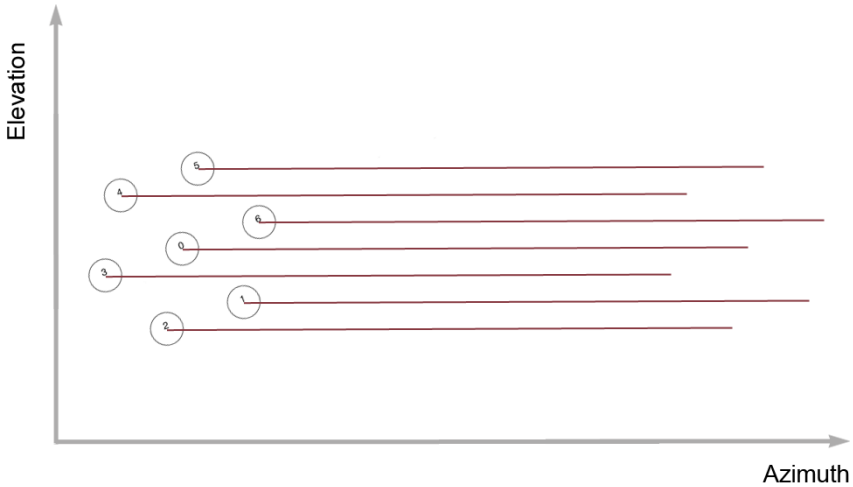
BSC

The *Best Space Coverage* configuration is useful for scans. It automatically rotates the dewar in order to best cover the scanned area, according to system-defined parameters which are specific to the receiver in use and taking into account the reference frame of the observation (Equatorial, Galactic and Horizontal).

If scans (OTF or raster) are being held in a celestial frame (i.e. EQ or GAL), this means that the dewar will be positioned so that the feed tracks on the sky are monospaced along the axis which is orthogonal to the scanning axis, and this geometry will be kept along the whole acquisition - i.e. the derotation is active.



For the Horizontal frame (scans taking place along Azimuth or Elevation) the derotation is not enabled: the dewar is in a fixed position, where the specific angle is chosen by the system in order to obtain the equally-spaced scans.



As in BSC mode the needed angles are pre-defined, any use of the `derotatorSetPosition` command will raise an error.

CUSTOM

In this configuration it is the user to choose which angle the dewar axis must form with the y-axis of the scanning frame. Once this angle is set by means of the `derotatorSetPosition` command, and the scans are commanded, this angle is kept for the whole duration of the acquisition - i.e. the derotation is active. As above, when acquisitions are performed in the Horizontal frame, the derotation is inhibited and the initial angle is kept fixed wrt the horizon.

Rewinding

The dewar can rotate within a limited range due to physical constraints. For the K-band 7-feed receiver this range, wrt the rest position, goes from -85.77° to +125.23°. This means that it is necessary to periodically rewind it in order to achieve the desired feed positions. It can happen both when a new scan is commanded and during long acquisitions, when the parallactic angle variation overcomes the available rotation range.

The system is in charge of automatically commanding a dewar rewind whenever it is required. This means that pauses might take place between consecutive subscans, and that a certain sky position will be observed by two different feeds before and after the rewind: a rewind implies the rotation by N*60° (where N can be 1, 2 or 3). The system will always choose the maximum allowed rewind, in order to delay the following rewind as much as possible.

When a rewind is necessary and during its execution, the antenna status will be labeled as "off-source" (this involving the tracking flag associated to any data sample which might still be under acquisition). The rewinding operations will not stop an ongoing subscan/acquisition, but no further acquisitions will be commanded until the antenna is back on-source.

Spectral line observations

Spectral line observations can either take place exploiting proper schedules (see dedicated documents on how the schedules must be written or how to produce them with the *schedulecreator* tool) or by means of command-line operations. Here this command-line scenario is described.

Line rest frequency

The spectral line rest frequency must be provided to the system using the following command:

```
> restFrequency=[value]
```

[value] is given in MHz and is a multiple argument: it can list a different value for each section - as long as XARCOS is the backend in use (not all the backends allow this sub-tuning). Specifying a single value assigns the rest frequency to *all* the sections.

e.g. `> restFrequency=22235.8`

used with the 7-feed K-band receiver implies that all the 14 sections are meant to observe the same line. Instead, providing different values allows to tune the sections on different lines (notice the semicolon used as a separator):

e.g. `> restFrequency=22235.8;22235.8;22080.0,22080.0,...,`

notice that all the N values (these being 14 for the K-band MF) must be explicitly provided or errors will rise.

The specified values will hold until different ones are commanded, or until a new general *setup* command is entered.

Target velocity parameters

When spectral line observations are required, targets must be provided with proper indications on their radial velocity, i.e.:

- **Radial velocity**
- **Reference frame**
- **Definition**

in order for the system to compute, at runtime, the frequency drift due to the line of-sight doppler contribution.

Once a target has been defined, for example commanding a `sidereal` positioning, its velocity can be provided by means of a manual command in the operatorInput panel, i.e.:

```
> radialVelocity=[vrad],[vref],[vdef]
```

Where

- [vrad] (radial velocity) is in km/sec if vdef is *not* Z
- [vref] (reference frame) can be one of the following:
  - **BARY**: Solar System BARYCENTRE
  - **LSRK**: Kinematic Local Standard of Rest
  - **LSRD**: Dynamical Local Standard of Rest
  - **LGRP**: Local Group
  - **GALCEN**: Galactic Centre
  - **TOPOCEN**: TOPOCENTRIC (observer's frame)
- [vdef] (velocity definition) can either be:
  - **RD**: Radio Definition
  - **OP**: Optical Definition
  - **Z**: stands for Redshift

The wildcard code `*` can be provided for all the required arguments and will be considered as "keep the present values".

The specified velocity parameters are valid until a new target is commanded.

Note: The `radialVelocity` command overrides any other velocity value that might have been differently expressed (e.g. if the target was selected from the source catalogue with the `track` command).

Commanding the doppler computation

Once the above information have been provided to the system, the computation of the line observed frequency can take place. Use the command:

```
> fTrack=[dev]
```

It collects all the required data from the antenna, the back-end and the front-end, plus the information provided by the user (see the above `radialVelocity` and `restFrequency` commands), then it tunes the telescope devices in order to centre the line(s) in each section bandwidth. The command lets the user select which device [dev] is asked to perform the tuning:

- **LO**: only the front-end local oscillator is moved
- **ALL**: first of all the front-end local oscillator is tuned, then the back-end - if it allows such a sub-tuning - also performs a further frequency adjustment, in order to centre the line(s) in the various sections. This option is useful in case multiple rest frequencies are to be observed, yet a complete success cannot be guaranteed as the dopplered frequencies might turn out to be incompatible with the section bandwidths and the LO ranges. In case at least one line lies outside the RF band of the receiver or the back-end input bandwidth, an error rises.

Manually acquiring data

See *Command-line measurements and acquisitions*

Generating and launching a schedule

A schedule is a set of files where all the geometry/timing/frequency details of a sequence of data acquisitions are specified, according to a syntax that enables DISCOS to read and execute them. The detailed structure of the several files composing a schedule is explained in this separate guide `pdf` Schedules for the most common continuum and spectroscopy observing modes (OTF cross-scans and maps, raster maps and ON-OFF) can be easily generated using a tool called **basie**. It can be found, together with its user manual, here: Basie repository

Once a schedule is ready, it must be copied to the folder reserved to the project. There, the schedule formal consistency can be tested using the scheduleChecker command given within a terminal (*not* in the operatorInput):

```
> scheduleChecker [schedname].scd
```

Only the syntax correctness will be verified. If errors are present, a reply will briefly address them indicating their position inside the files.

To launch a schedule, simply use:

```
> startSchedule=[project/][schedname].scd,[N]
```

project is the unique string assigned to every project. It can be omitted if the project command had been used. If both the choices are made, i.e. if the project name was set with project but a string is also inserted in the startSchedule command, the latter overrides the former. Then follows the schedule name, in particular the name of the SCD file. *N* is the identifier of the scan or subscan from which DISCOS must start reading it – it is particularly useful in case of a sequential (i.e. not time-based) schedule. *N* can be the scan number, e.g. 2, or the scan_subscan specification, e.g. 2_5.

You can specify **when to start** the schedule (in UT):

```
> startSchedule=[project/][schedname].scd,[N]@[DOY-HH:MM:SS]
```

DISCOS reads the configuration parameters from the schedule, which can be relative both to the receiver and the backend, and accordingly sets these devices. This might take several seconds, especially when using the MF receiver. While the setup takes place, several values change in the TPB monitor. The last operation is the upload of the first pointing/scan read from the schedule, whose parameters will show up in the bottom section of the AntennaBoss monitor. During the scans, all the three flags in the lower part of the AntennaBoss monitor must be a green "@". Pay attention to the "Tracking" one. It should turn to a red "o" only when the antenna is slewing between scans on the same source, or when slewing to/from a new source. **If tracking is not correct for some consecutive seconds**

during a scan, something is wrong. Sequential schedules run ad libitum. To abruptly interrupt the running schedule, truncating the ongoing acquisition:

```
> stopSchedule
```

Instead, to stop the schedule allowing the completion of the present file:

```
> haltSchedule
```

Data formats and online quick-look

DISCOS FITS output files

Details on the FITS file structure are given in a separate document. Download it here: *pdf*

Online data quick-look (FITS acquisitions)

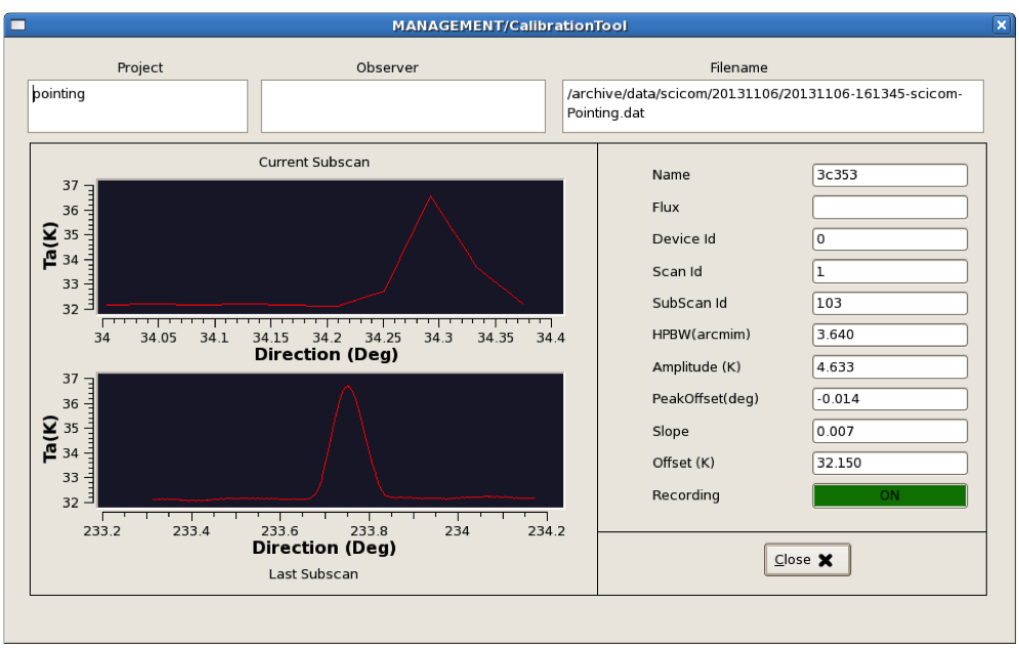
During acquisitions, you can inspect the recorded data in real time, by using a dedicated web tool. Launch it by double-clicking on the quicklook.html icon located on the *discos-console* desktop.

If writer is MANAGEMENT/Point or MANAGEMENT/CalibrationTool

When data are acquired – both manually or through a schedule – using the Point or CalibrationTool writers, the quick-look can be performed using the CalibrationToolClient. Open a terminal on *discos-console* and use the command:

```
$ calibrationtoolclient  [componentName]
```

where componentName is either MANAGEMENT/Point or MANAGEMENT/CalibrationTool. A graphic window will appear. Its content is given in the following figure.



Note: In this client, the subscan currently being acquired is shown *in real-time* (upper plot), even if in a low-res version. Under this display, the last completed subscan - in its full sampling - is shown.

Retrieving the data

Accessing your disk space

Each observing project owns disk space on *discos-console*. Open a terminal on such machine and use:

```
$ su - [project-code]
```

The necessary password is provided by your project friend. Once logged in, you will find specific folders under your home folder, in particular

- **data** Folder containing the data acquired by running schedules when observing with XARCOS or the TPB
- **sardaraData** Folder containing the data acquired by running schedules when observing with SARDARA
- **extraData** Folder containing the data acquired by using command lines
- **logs** Folder where logfiles are stored
- **schedules** Folder where your schedules must be uploaded to

Your disk space can also be remotely accessed, as long as you connect to it from the SRT campus network. Use:

```
$ ssh [project-code]@discos-console
```

To retrieve your data, open a terminal on your machine and use:

```
$ scp -r [project-code]@discos-console:./data/[YYYYMMDD] .
```

For SARDARA data, this becomes:

```
$ scp -r [project-code]@discos-console:./sardaraData/[YYYYMMDD] .
```

When you need to upload your schedules to the system, instead, open a terminal on your machine and write:

```
$ scp [schedulename].* [project-code]@discos-console:./schedules
```

Data structure

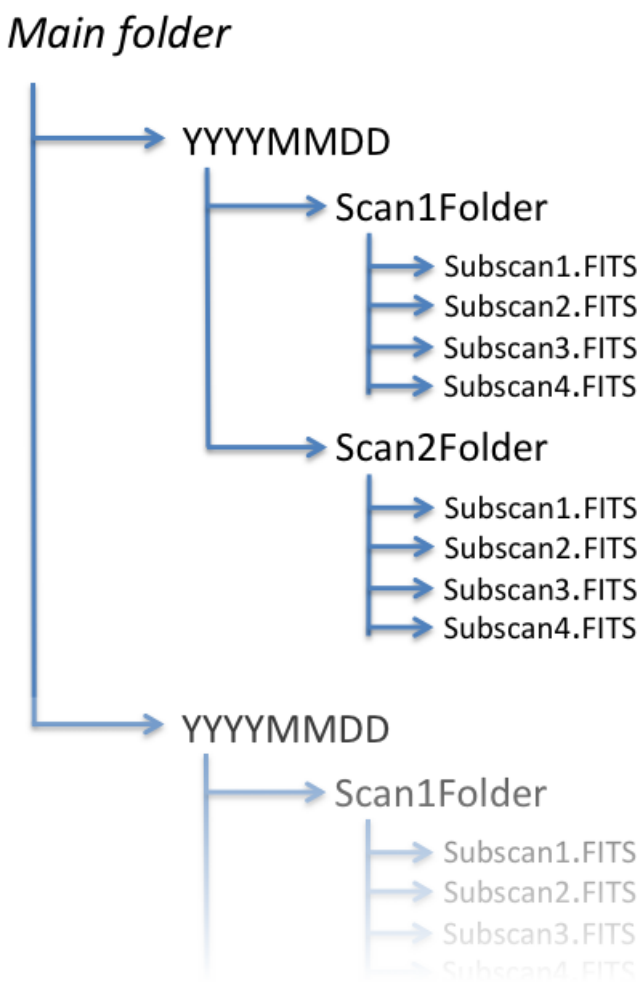
Inside the data folders, subfolders named according to the date (YYYYMMDD) will be automatically created during acquisitions. Taking into account the choice of the FITS format, the date-dependent folder contains a subfolder for every scan, inside which there are the FITS files (one for each subscan).

Scan folder names are composed as:

YYYMMDD-HHMMSS-Project-Suffix

where

- **HHMMSS** is the UT time associated to the first sample of the acquisition
- **Project is the code/name specified using the *`project=`* command,** or when starting a schedule with *`startSchedule=[project/][schedulename].scd,[N]`*



- **Suffix is a user-defined string retrieved from the schedule files.** Though no control can be applied on the choice/check of this string, the agreement is that it should coincide with the target name.

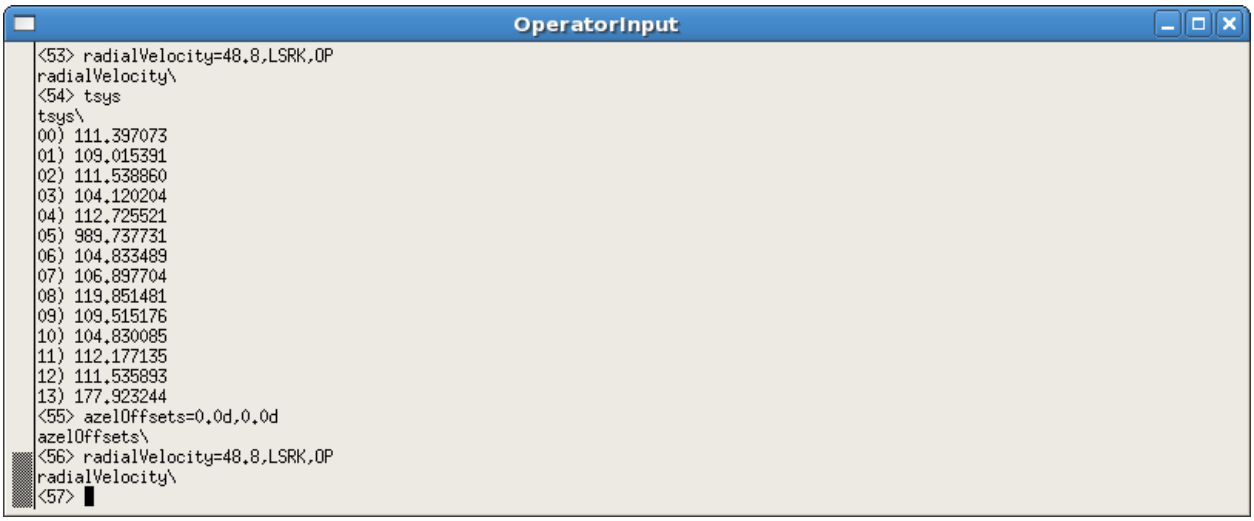
FITS files, each corresponding to a subscan, are composed as:

YYYYMMDD-HHMMSS-Project-Suffix_Scan#_Subscan#.fits

Appendix A - Monitor panels full description

operatorInput

This is the **input console** where users write DISCOS commands. The prompt is just a sequential number enclosed in <>. If a command is properly read, the system replies repeating the command itself, followed by the operation results (if they are foreseen). Otherwise, an error message appears. To close the window, type `exit`.

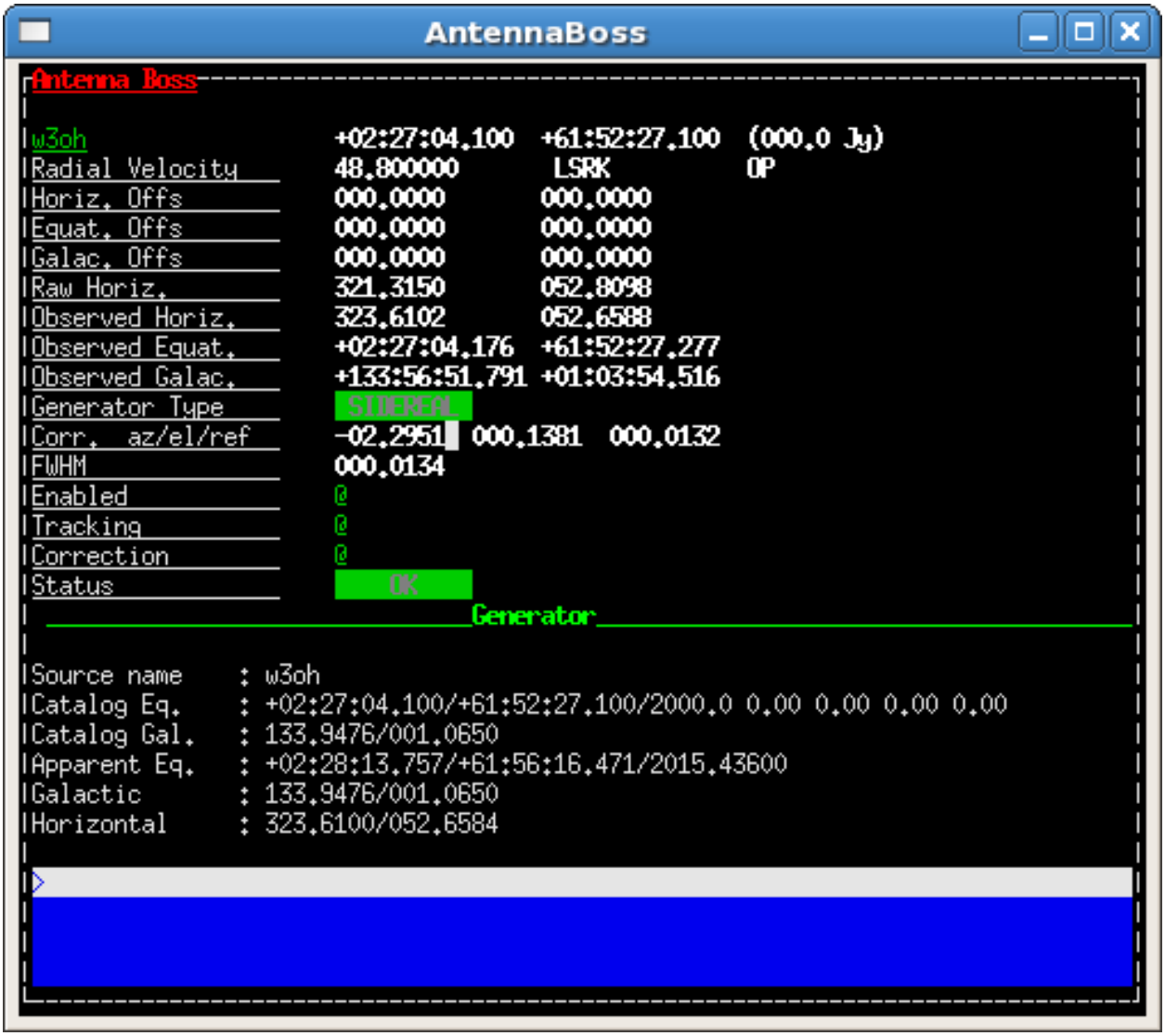


AntennaBoss

This monitor shows the target info, indicating the commanded and actual positions pointed by the antenna. It also gives a feedback on the pointing accuracy and on the overall antenna status.

Parameters:

- **Target**: label (as extracted from schedule or commanded), coordinates (RAJ2000.0, DecJ2000.0), estimated flux density (Jy, if available).
- **Radial Velocity**: (as extracted from schedule or commanded), value (km/s unless it is a redshift), velocity reference frame, velocity definition.
- **Horiz. Offs**: Horizontal (Az-El) offsets as read from schedule, degrees.
- **Equat. Offs**: Equatorial (RA-Dec) offsets as read from schedule, degrees.
- **Galac. Offs**: Galactic (l-b) offsets as read from schedule, degrees.
- **Raw Horiz.**: commanded Az-El, including pointing model, refraction, etc...
- **Observed Horiz.**: Az-El coordinates read from the mount encoders and cleaned from the pointing model and refraction contributions. Because of this, observed coordinates will differ from the raw ones.
- **Observed Equat.**: RA-Dec J2000.0 coordinates, converted from the observed horizontal.
- **Observed Galac.**: l-b Galactic coordinates, converted from the observed horizontal.
- **Generator Type**: which component is in charge of the generation of the coordinates. It can be `NONE` (which is the condition at startup), `OTF`, `MOON` or `SIDEREAL`.
- **Corr. az/el/ref**: azimuth and elevation corrections (degrees) applied by the pointing model, plus the refraction model contribution– which is an additional correction to elevation.
- **FWHM**: Full Width Half Maximum (corresponding to HPBW), degrees.
- **Enabled**: a green `@` indicates that the antenna is correctly receiving commands; a red `o` means the communication is disabled.
- **Tracking**: it indicates whether the pointing error exceeds 0.1*FWHM or not. A green `@` corresponds to error < 0.1*FWMH. It should turn to a red **o** only when the antenna is slewing between scans on the same source, or when slewing to/from a new source.


AntennaBoss
Antenna Boss
w3oh +02:27:04.100 +61:52:27.100 (000.0 Jy)
Radial Velocity 48.800000 LSRK OP
Horiz. Offs 000.0000 000.0000
Equat. Offs 000.0000 000.0000
Galac. Offs 000.0000 000.0000
Raw Horiz. 321.3150 052.8098
Observed Horiz. 323.6102 052.6588
Observed Equat. +02:27:04.176 +61:52:27.277
Observed Galac. +133:56:51.791 +01:03:54.516
Generator Type SIDEREAL
Corr. az/el/ref -02.2951 000.1381 000.0132
FWHM 000.0134
Enabled @
Tracking @
Correction @
Status OK
Generator
Source name : w3oh
Catalog Eq. : +02:27:04.100/+61:52:27.100/2000.0 0.00 0.00 0.00 0.00
Catalog Gal. : 133.9476/001.0650
Apparent Eq. : +02:28:13.757/+61:56:16.471/2015.43600
Galactic : 133.9476/001.0650
Horizontal : 323.6100/052.6584

- **Correction**: application of the above horizontal coordinates corrections. If disabled (red circle), all corrections are zeroed.
- **Status**: *OK*, *WARNING* or *ERROR*. "Warning" needs investigation but usually does not stop the ongoing activity (it also appears at startup, before the setup commands), "Error" generally appears if something stops the observations.
- **Generator**: under this line all the subscan setup parameters appear when it is commanded. See the two cases here below.

In particular, for OTF observations:

- **Start-Center-Stop** – coordinates in the subscan frame of the start, center and stop positions (degrees)
- **StartTime-Dur.** – subscan start UT *yyyy-doy-hh:mm:ss.s* and duration *hh:mm:ss.s*
- **J2000-Apparent eq** – during the subscan, current RA-Dec position pointed by the antenna, both in the J2000 epoch and precessed to the date. Format is *hh:mm:ss.s-°:′:″.″*
- **Horizontal** - during the subscan, current Az-El position pointed by the antenna (degrees)
- **Galactic** - during the subscan, current l-b position pointed by the antenna (degrees)
- **Sky/Phi rate** – actual subscan speed on sky along scanning direction (°/s) . The 'Phi rate' value refers to great circle arcs, if it is the scanning geometry, otherwise it copies the first speed value.

For **SIDEREAL** scans, instead, this is what appears:

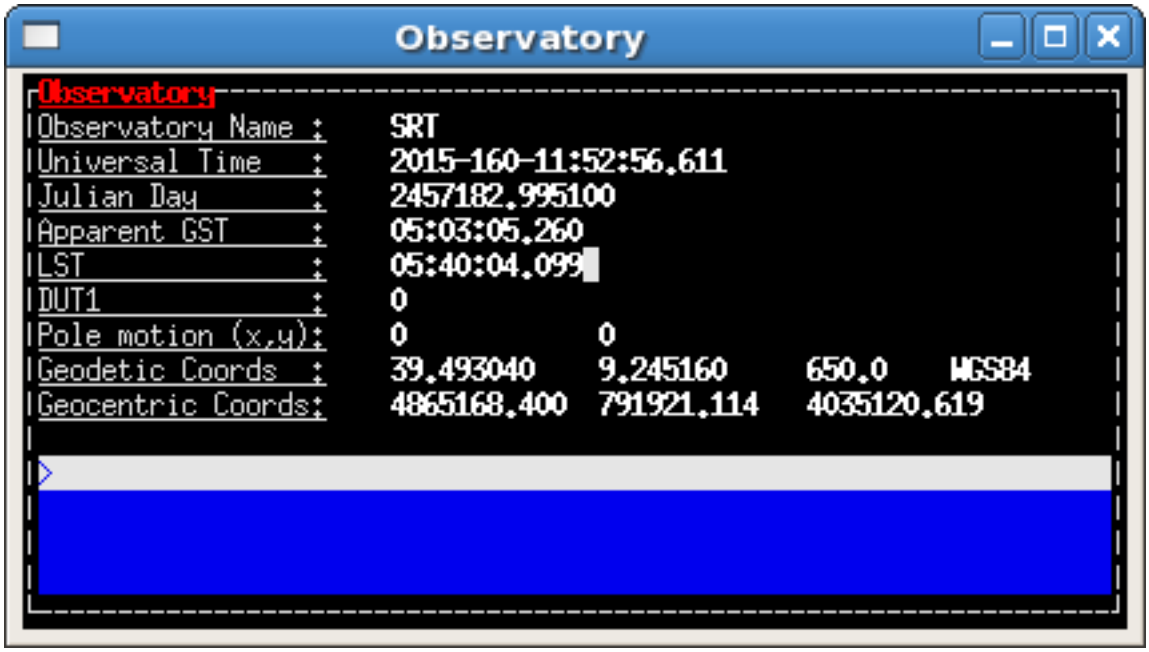
- **Source name** – target name
- **Catalog Eq.** – target FK5 coordinates and info (if available): RA *hh:mm:ss.s*, Dec *°:′:″.″*, Epoch, RA proper motion (milliarcsec per julian year), Dec proper motion (milliarcsec per julian year), parallax (milliarcsec), radial velocity (km/s)
- **Catalog Gal.** – target Galactic longitude and latitude (degrees) converted from the above Equatorial or provided by user/catalogue
- **Apparent Eq.** – target current apparent Equatorial coordinates: RA *hh:mm:ss.s*, Dec *°:′:″.″*, Epoch
- **Galactic** – current l-b position pointed by the antenna (degrees)
- **Horizontal** –current Az-El position pointed by the antenna (degrees)

To close the monitor, type *exit* in its prompt (the grey line).

Observatory

It is devoted to the station coordinates and times:

- **Observatory Name**: SRT, Medicina or Noto
- **Universal Time**: *YYYY-DOY-HH:MM:SS.SSS*
- **Julian Day**: *d.ddd*
- **Apparent GST**: Greenwich Sidereal Time *HH:MM:SS.SSS*
- **LST**: Local Sidereal Time *HH:MM:SS.SSS*
- **DUT1**: difference between UT1 and UT (s), if applied.
- **Pole motion**: celestial pole offset w.r.t. a reference position (details are note provided here), measured in μarcsec on a tangential projection.



- **Geodetic Coords**: updated Latitude and Longitude (degrees) and Altitude (m) for the telescope, plus the geodetic model code.
- **Geocentric Coords**: geocentric cartesian coordinates (m) for the telescope.

To close the monitor, type `exit` in its prompt (the grey line).

Mount

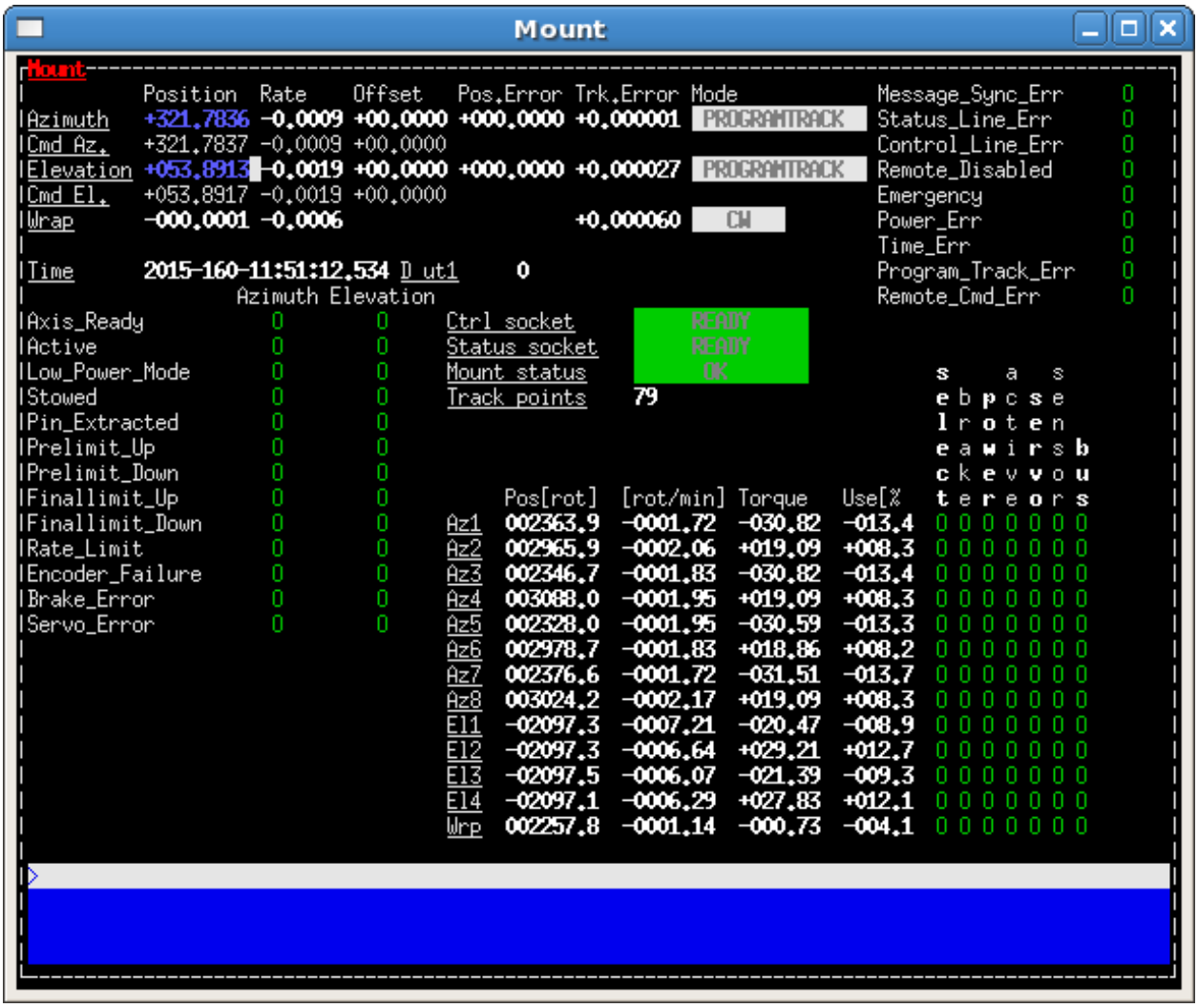
This is a quite complex frame, allowing the technical staff to monitor all the various parts of the antenna mount.

Observers must focus only on a subset of the displayed parameters and flags. We thus describe the main features only.

The top left section gives the online readouts for the Azimuth and Elevation axes, compared to the commanded positions. Line by line:

- **Azimuth**: current azimuth position (in the -90°/450° wrapping range), rate (°/s) and commanded offset (degrees). Then the measured position error and tracking error (both in degrees) are given. Finally, the axis active mode is displayed: it can be *PRESET* (fixed Az-El pointing only), *PROGRAMTRACK* (for tracking, OTF and schedules), STOP (if axes brakes are on), *UNKNOWN* (indicating a non-configured status).
- **Cmd Az.**: commanded azimuth position (in the -90°/450° wrapping range), rate (°/s) and offset (°). This offset comes from metrology devices, it is not related to user-defined offsets or subscan-defined offsets.
- **Elevation**: current elevation position (degrees), rate (°/s) and commanded offset (degrees). Then the measured position error and tracking error (both in degrees) are given. Finally, the axis active mode is displayed: it can be *PRESET* (fixed Az-El pointing), *PROGRAMTRACK* (for OTF and schedules), *STOP* (if axes brakes are on), *UNKNOWN* (usually indicating an error condition).
- **Cmd El.**: commanded elevation position (degrees) and rate (°/s).
- **Wrap**: readouts from the cable wrap. The status flag on the right can be *CW*''*or* ''*CCW*.

The general condition of the mount is summed up by one keyword, in the central part:


Mount
Mount
Position Rate Offset Pos.Error Trk.Error Mode
Azimuth +321.7836 -0.0009 +00.0000 +000.0000 +0.000001 PROGRAMTRACK
Cmd Az. +321.7837 -0.0009 +00.0000
Elevation +053.8913 -0.0019 +00.0000 +000.0000 +0.000027 PROGRAMTRACK
Cmd El. +053.8917 -0.0019 +00.0000
Wrap -000.0001 -0.0006 +0.000060 CW
Time 2015-160-11:51:12.534 D ut1 0
Azimuth Elevation
Axis_Ready 0 0
Active 0 0
Low_Power_Mode 0 0
Stowed 0 0
Pin_Extracted 0 0
Prelimit_Up 0 0
Prelimit_Down 0 0
Finallimit_Up 0 0
Finallimit_Down 0 0
Rate_Limit 0 0
Encoder_Failure 0 0
Brake_Error 0 0
Servo_Error 0 0
Ctrl socket READY
Status socket READY
Mount status OK
Track points 79
Pos[rot] [rot/min] Torque Use[%
Az1 002363.9 -0001.72 -030.82 -013.4
Az2 002965.9 -0002.06 +019.09 +008.3
Az3 002346.7 -0001.83 -030.82 -013.4
Az4 003088.0 -0001.95 +019.09 +008.3
Az5 002328.0 -0001.95 -030.59 -013.3
Az6 002978.7 -0001.83 +018.86 +008.2
Az7 002376.6 -0001.72 -031.51 -013.7
Az8 003024.2 -0002.17 +019.09 +008.3
El1 -02097.3 -0007.21 -020.47 -008.9
El2 -02097.3 -0006.64 +029.21 +012.7
El3 -02097.5 -0006.07 -021.39 -009.3
El4 -02097.1 -0006.29 +027.83 +012.1
Wrp 002257.8 -0001.14 -000.73 -004.1
Message_Sync_Err 0
Status_Line_Err 0
Control_Line_Err 0
Remote_Disabled 0
Emergency 0
Power_Err 0
Time_Err 0
Program_Track_Err 0
Remote_Cmd_Err 0

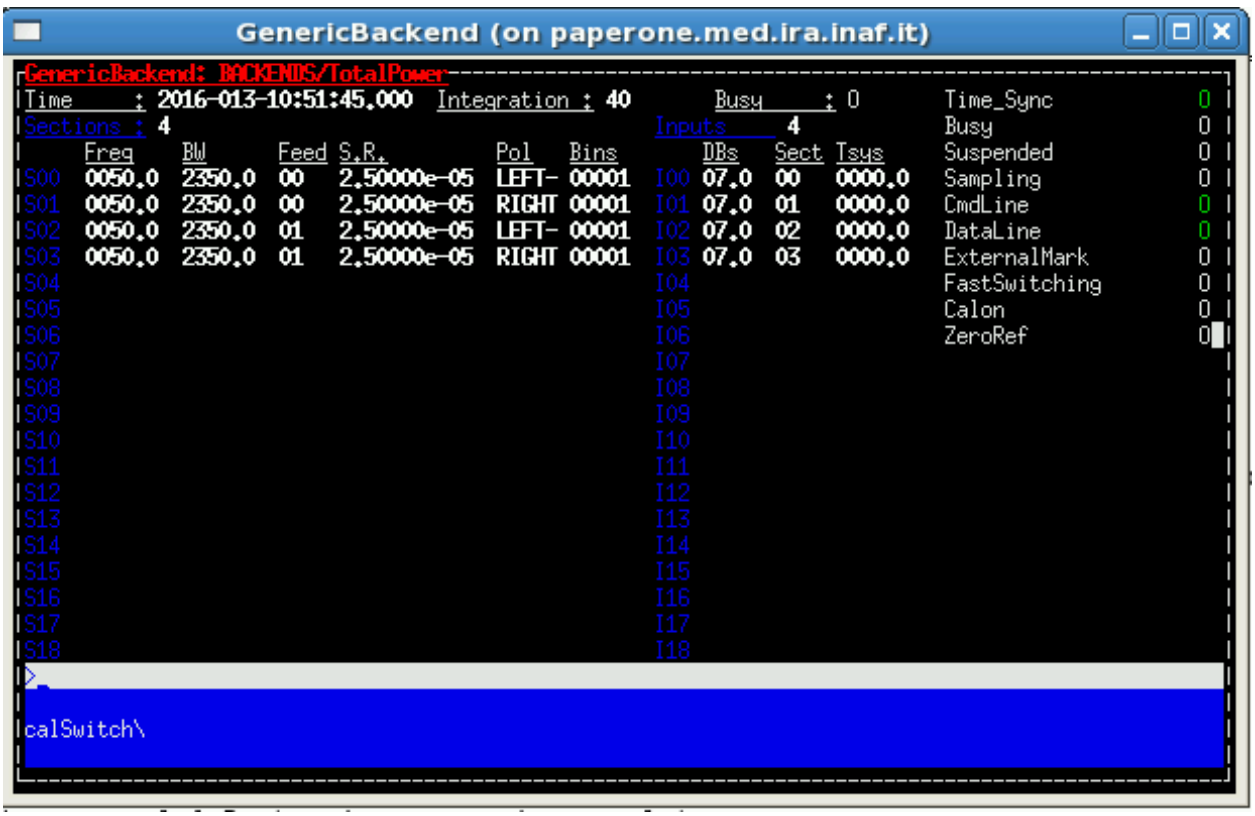
- **Mount status**: it can be `OK`, `WARNING`, `FAILURE` or `BUSY`, the latter is associated to operations which take a lot of time to complete (as the stow-unstow procedure). When the mount is BUSY it will not accept further commands until the ongoing operation is completed.

The rest of the panel lists several **flags and status labels**. For average users, the only points to be taken into account are:

- in standard observing conditions, when a schedule runs, all flags should be green `o`;
- in case of warnings, flags turn to yellow `o`;
- errors correspond to red `o`;
- when a "failure" keyword turns steadily to a red `o`, or a permanent failure status appears, immediately call for technical assistance.

To close the monitor, type `exit` in its prompt (the grey line).

GenericBackend



The panel shows one row for every section.

- **Freq**: value (MHz) to be added to the LO frequency in order to obtain the observed frequency at the beginning of the band
- **BW**: bandwidth (MHz)
- **Feed**: number of the receiver feed connected to this section
- **S.R.**: sampling rate (MHz)
- **Pol**: polarisation (Left or Right)

- **Bins**: number of frequency bins (1 for total power)
- **DBs**: attenuation (dB) applied to the section
- **Sect**: section number
- **Tsys**: the last measured Tsys (K)

Also some backend-dependent status flags are present, in the top right area. When observing with the TPB, look for:

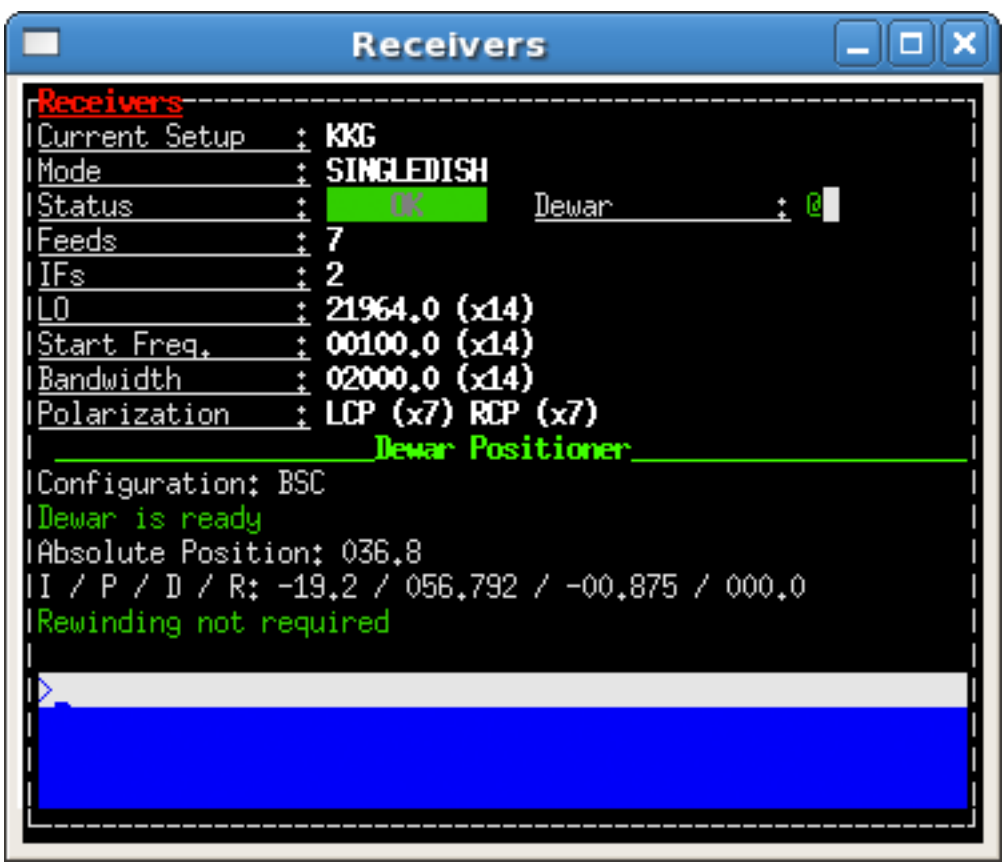
- **Time_Sync**: if it frequently or steadily turns red call for assistance (the backend time is not synchronized)
- **Busy**: when schedules are running, it must turn yellow. If it does not, the backend is not acquiring.

With DISCOS 0.6, an additional set of keywords was inserted:

- **ExternalMark**: it turns yellow when the external mark control is enabled
- **FastSwitching**: it turns yellow when the total power is sampling with the mark swicthing under its direct control. Note: only when the sampling is active.
- **Calon**: it turns yellow when the noise mark is on (non-fast mode) under the control of the Backend. This is not implemented yet.
- **ZeroRef**: it turns yellow when the backend inputs are connected to the 500hm. It may be yellow during a system temperature measurement.

To close the monitor, type `exit` in its prompt (the grey line).

ReceiversBoss



This monitor summarizes the frontend setup. The bottom part is devoted to the derotator (dewar positioner), when available.

- **Current setup**: receiver code
- **Mode**: *NORMAL*, *SINGLEDISH*, *VLBI...* (other codes to come)
- **Status**: *OK*, *WARNING* or *FAILURE*
- **Feeds**: number of feeds (1 fore single-feed receivers, 7 for the K-band MF)
- **LO**: Local Oscillator frequency (MHz)
- **Start Freq.**: step (MHz) to be added to the LO frequency in order to obtain the actually observed frequency at the beginning of the band
- **Bandwidth**: actually observed bandwidth (MHz)
- **Polarization**: *LCP*, *RCP*, *HLP* or *VLP*
- **Dewar positioner**: under this line all the derotator details are given.

In particular:

- **Configuration**: shows the current configuration (FIXED, BSC, CUSTOM)
- **Dewar is ready**: is a green-colored string appearing when the dewar positioner is ok. It changes to a red "Dewar in not ready" otherwise.
- **Absolute Position**: shows the present position (encoder readout) wrt the mechanic zero (degrees). A positive angle means CW.
- **I / P / D / R**: notable angles involved in the current derotator usage:

  I = Initial offset angle set for the current configuration (degrees); for FIXED or CUSTOM configuration it is the user-set angle, for BSC is a receiver-dependent angle, -19.2° for the K-band MF)

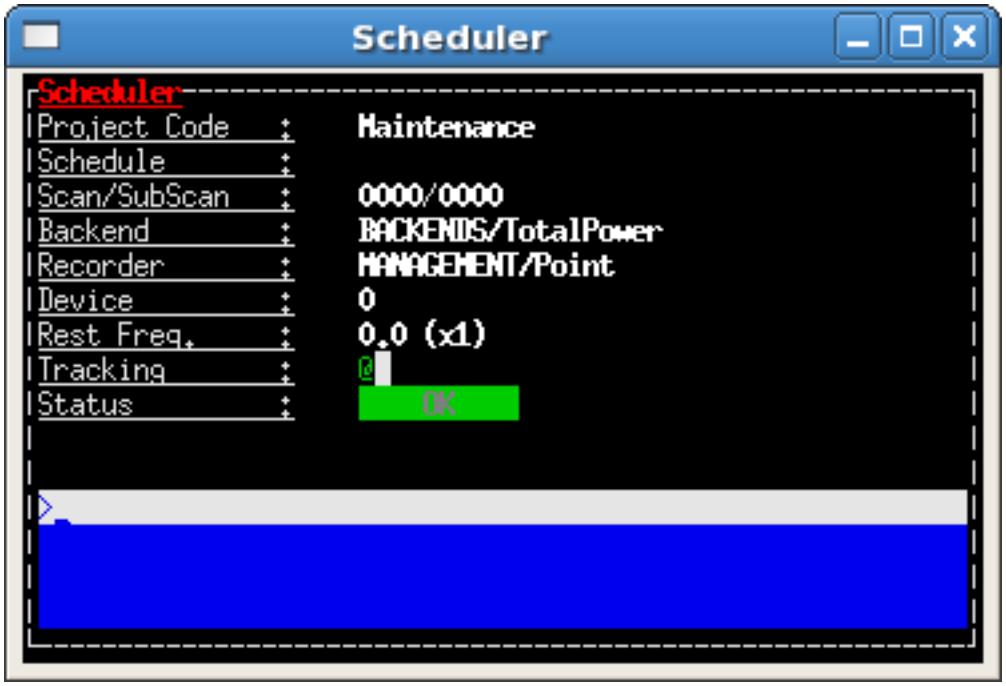
  P = Parallactic angle of the pointed position (degrees)

  D = variation (Delta) of the parallactic angle since the beginning of the acquisition (degrees)

  R = reports the angle (degrees) applied if a derotator rewind has occurred.
- **Rewinding not required**: is a green-colored string appearing when the dewar rewind is not necessary. It changes to a red "Rewinding is required" when the derotator reaches its range limits and to a yellow "Rewinding required (rewinding...)" while the rewind is taking place. When red or yellow, it implies that the general tracking flag in the Antenna TUI will turn red and the antenna status will be labeled as "off-source" (this involving the tracking flag associated to any data sample which might still be under acquisition). The rewinding operations will not stop an ongoing subscan/acquisition, but no further acquisitions will be commanded until the antenna is back on-source.

To close the monitor, type *exit* in its prompt (the grey line).

Scheduler



This monitor shows details on the selected data acquisition devices and on the running schedule, if any.

- **Project code**: project name, as specified at the launch of the schedule (or as input using the *project*= command)
- **Schedule**: name of the active schedule
- **Scan/Subscan**: scan number and subscan number, relative to the ongoing acquisition
- **Backend**: name of the selected backend, as listed in the schedule
- **Recorder**: name of the selected data recording device, as listed in the schedule
- **Device**: number of the currently selected device (see device= command)
- **Rest Freq.**: rest frequency of the line(s) being observed (MHz). The number in brackets indicates how many IFs the given rest frequency refers to.
- **Tracking**: tracking status of the telescope, including antenna, active surface, minor servo. A red `o` means the telescope is not yet in its tracking route (or that it is not tracking properly), a green light means the observation is ongoing and the tracking is accurate within 0.1*HPBW
- **Status**: flag summarising the telescope status. `OK` is self-explanatory, `WARNING` in principle indicates that the acquisition can go on even if a problem has been reported, `ERROR` signals that a major problem was detected and the observation cannot be performed.

To close the monitor, type `exit` in its prompt (the grey line).

MinorServo

This monitor shows details on the Minor Servo systems:

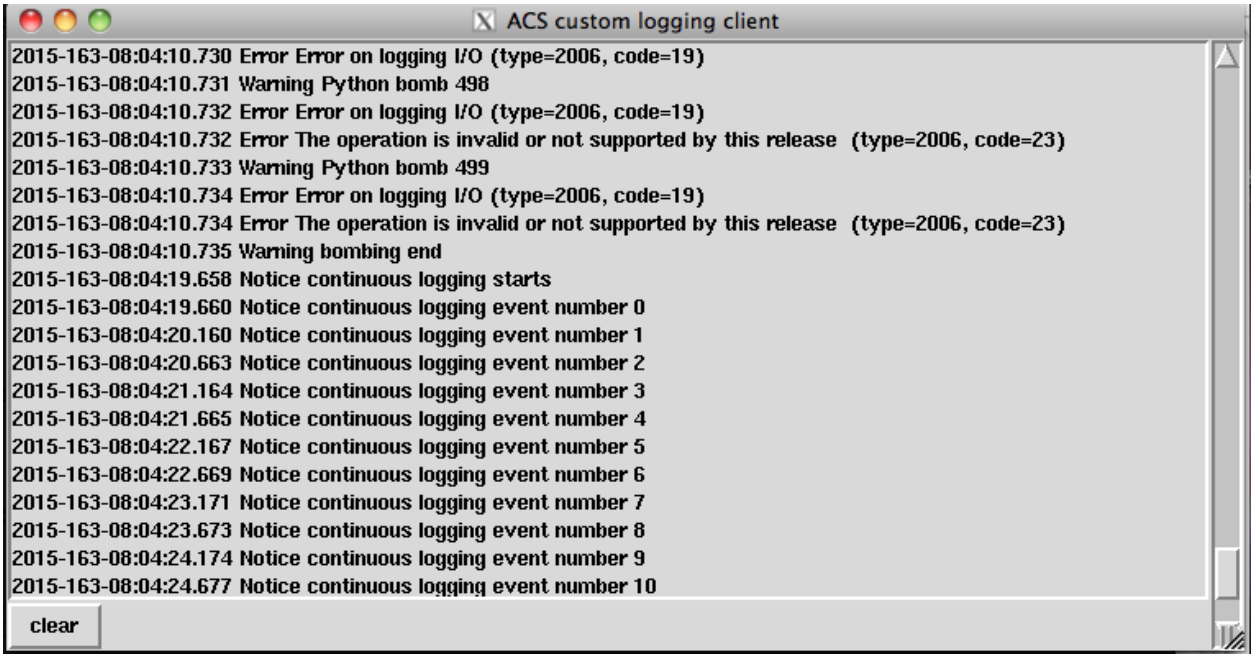
- **Current Setup**: current setup code (**CCB**, **KKG**, **KKG_ASACTIVE**, . . .)
- **Tracking**: a red `o` means the minor servo system is not tracking properly, and a green `@` light indicates the minor servo actual position is as expected.
- **Status**: `OK`, `WARNING` or `FAILURE`. A *warning* needs investigation but usually does not stop the ongoing activity (it also appears at startup, before the setup commands), and a *failure* appears if there is a fault in the system. In case of failure the tracking field can be green, and that means the observation can continue (for instance, it is possible to have a PFP fault when the required receiver is on focus).
- **Motion Info**: it describes the minor servo movement. It can be: `Starting`, `Parking`, `Elevation Track Mode`, `No Elevation Track Mode`, and `Not Ready to Move` (in that case a setup is required).

To close the monitor, type `exit` in its prompt (the grey line).

Note: All the monitors can be closed at a time using the command (in a terminal on OBS1):

```
$ DISCOSConsole --stop
```

Logging Display



This panel is not part of the general Console, so it must be individually opened/closed. Here all the log messages will appear. New messages are shown on top of the previous ones. To close the window, click on its dedicated icon.

Appendix B - Complete command list

Note:

COMMAND TEMPORISATION All commands can be temporised (i.e. associated to a starting time) adding a proper **suffix**. There are two possibilities:

- **absolute temporisation** (the operation will be performed at the indicated time): `@DOY-HH:MM:SS`, where DOY is the Day-Of-Year (1-366) and HH:MM:SS is the UT time;
- **iterative temporisation** (the operation is performed now, then periodically according to the indicated time interval): `@!DAYS-HH:MM:SS`, where DAYS is the number of days and HH:MM:SS is hours, minutes, seconds.

Commands (temporised or not) can be used also in the init/pre-scan/post-scan procedures inside schedules. *Observers* are in charge of considering *if* and *when* the use of a certain command makes sense in their schedule, according to their specific needs and goals: this is something that no schedule parser can check!

Here follow all the commands exploitable in DISCOS:

`> antennaPark`
sends the antenna to stow position

`> antennaReset`
resets the alarm conditions

`> antennaSetup=[code]`
(`code` can be `CCB`, `KKG`, ...) unstows the antenna, sets it to tracking mode and configures the pointing model according to the specified receiver. It does NOT perform the receiver and backend setup

> *antennaStop*
stops the antenna. Activities can start again only commanding a mode change as `antennaTrack` (which does not affect the overall setup) or a new setup

> *antennaTrack*
sets the antenna to PROGRAMTRACK mode. It does not change the pointing model or any receiver setup

> *asSetup=[code]*
(`code` can be `S`, `SF`, `P`, `PF`) performs the setup of the active surface

> *azelOffsets=[double]d,[double]d*
sets the Az-El offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source Elevation.

Example: `> azelOffsets=-0.05d,0.05d`

> *calOn*
switches the calibration mark on

> *calOff*
switches the calibration mark off

> *chooseBackend=[string]*
selects the backend; *string* can be `BACKENDS/TotalPower`, `BACKENDS/XBackends`, `BACKENDS/Roach`

> *chooseRecorder=[string]*
selects the backend; *string* can be `MANAGEMENT/FitsZilla`, `MANAGEMENT/CalibrationTool` or `MANAGEMENT/Point`

> *clearServoOffsets*
zeroes the subreflector system-defined offsets (e.g. the ones resulting from a focus scan)

> *crossScan=[scanFrame],[span],[duration]*
performs a cross-scan on the previously selected target (indicated using the `track` or `sidereal` commands), along the *scanFrame* (`EQ`, `HOR` or `GAL`), spanning *span* degrees in *duration* seconds

> *derotatorGetPosition*
reads out the present dewar position angle (degrees)

> *derotatorSetConfiguration=[conf]*
[conf] can have one of the following values: `FIXED`, `BSC`, `CUSTOM`

> *derotatorSetPosition=[ang]d*
sets the dewar angle (degrees). Positive angles = CW. According to the derotator configuration in use, the command has different meanings. It is illegal for the `BSC` configuration

> *device=[sect]*
computes the beamsize, taking into account the present receiver and backend configurations relative to section sect

> *flush=[N]*
deletes the N-th element in the queue of temporised commands

> *flushAll*
deletes all the queue of the temporised commands

> *focusScan=[span],[duration]*
performs a focus scan over the tracked source, span is in mm along the z-axis, duration is expressed in `hh:mm:ss`

Example: `> focusScan=60,00:01:00`

> fTrack=[dev]
It collects all the required data from the antenna, the back-end and the front-end, plus the information provided by the user (see the `radialVelocity` and `restFrequency` commands), then it tunes the telescope devices in order to centre the line(s) in each section bandwidth. The command lets the user select which device [dev] is asked to perform the tuning:

- **LO**: only the front-end local oscillator is moved
- **ALL**: the back-end performa a sub-tuning in the various sections

> getAttenuations
reads the attenuation values (dB) currently configured for the active sections, and lists them according to increasing section number

> getTpi
reads the signal intensity (raw counts) for the active sections, and lists them according to increasing section number

> goOff=[frame],[beams]
slews the antenna to an offset position, wrt a previously commanded target, along the longitude axis of the indicated coordinate frame (`EQ`, `HOR` or `GAL`). The user provides the offset value expressed in beamsizes. If the frame is HOR and target lies beyond the Elevation cutoff limits, the offset is applied in Elevation.

> goTo=[double]d,[double]d
sends the antenna, while in TRACKING mode, to the specified Az-El position.

Example: `goTo=180d,45d`

Arguments are always rounded in the range 0-360 and 0-90 for azimuth and elevation respectively (in any case the ranges are limited to mechanical contraints). The jolly character is valid and is considered as: keep the present value. The differences from the `preset` command are:

- once the antenna reaches the destination, the system will acknowledge the "on source" status;
- the pointing corrections (pointing model and refraction) are applied. In case they are not required they must be turned off explicitly.

> haltSchedule
completes the current scan and then stops the schedule

> initialize=[code]
(`code` can be `CCB`, `KKG`, ..., `XC00`, `XK77`, ...) configures the backend using the default parameters relative to the selected receiver. It does *not* act on the receiver, pointing model or antenna mount mode.

> integration=[double]
sets the backend integration time (ms)

> log=[filename]
defines a custom name for the logfile (do not specify the extension)

> lonlatOffsets=[double]d,[double]d
sets the Galactic b-l offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source latitude.

Example: `> lonlatOffsets=2.0d,-1.0d`

> moon
points the antenna to the present coordinates of the center of the Moon

> preset=[double]d,[double]d
sends the antenna, if in PRESET mode, to the specified Az-El position, without applying any pointing correction. This is useful when needing to point to a position next to the zenith. Beware: the antenna will reach the destination but no "on source" flag will be raised.

Example: `> preset=180d,45d`

`> project=[code]`
lets the system know which project is observing (the code/name must correspond to the one provided by the TAC). This code/name is then considered as default when launching schedules: the system will search for them in a folder named "project/schedules". This code/name also forms part of the output FITS filename. Notice that the PROJECT keyword indicated inside the schedule, which is then written in the "Project Name" keyword in the FITS main header, is a free string and might differ from the project official name.

`> radecOffsets=[double]d,[double]d`
sets the RA-Dec offsets (degrees). They are intended "on sky", i.e. it is the actual offset run on the sky at the source Declination.

Example: `> radecOffsets=1.0d,0.0d`

`> radialVelocity=[vrad],[vref],[vdef]`

- [vrad] (radial velocity) is in km/sec if vdef is *not* Z
- [vref] (reference frame) can be one of the following:
  - **BARY**: Solar System BARYCENTRE
  - **LSRK**: Kinematic Local Standard of Rest
  - **LSRD**: Dynamical Local Standard of Rest
  - **LGRP**: Local Group
  - **GALCEN**: Galactic Centre
  - **TOPOCEN**: TOPOCENTRIC (observer's frame)
- [vdef] (velocity definition) can either be:
  - **RD**: Radio Definition
  - **OP**: Optical Definition
  - **Z**: stands for Redshift

The specified velocity parameters are valid until a new target is commanded. The `radialVelocity` command overrides any other velocity value that might have been differently expressed

`> receiversMode=[code]`
configures the working mode of the receiver, according to its peculiar characteristics

`> receiversSetup=[code] (CCB, KKG, etc...)`
(`code` can be `CCB`, `KKG`, ...) configures the receiver using the default parameters. It does *not* act on the backend, pointing model or antenna mount mode

`> restFrequency=[freq1];...;[freqN]`
[freq] is given in MHz and is a multiple argument: it can list a different value for each of the N sections - as long as XARCOS is the backend in use (not all the backends allow this sub-tuning). Specifying a single value assigns the rest frequency to *all* the sections. The specified values will hold until different ones are commanded, or until a new general *setup* command is entered.

`> servoPark`
stows the minor servo system

`> servoSetup=[code]`
(`code` can be `CCB`, `KKG`, ...) configures the minor servo system only

`> setAttenuation=[sect],[att]`
sets to *att* (dB) the attenuator of section *sect*

> *setLO=[freq]*
Local Oscillator frequency, in MHz (one per IF, separated by ";", usually the values are identical) This LO frequency corresponds to: SkyFreq(@band start) – 100 MHz when using the TPB

> *setSection=[sect],[startFreq],[bw],[feed],[mode],[sampleRate],[bins]*
configures the backend section sect.

> *setServoASConfiguration=ON (or OFF)*
chooses the subreflector pointing model according to the AS being enabled (ON) or disabled (OFF)

> *setServoElevationTracking=ON*
sets the subreflector to tracking mode, with OFF tracking is disabled

> *setServoOffset=[axis_code],[value]*
(see details in *Antenna operations*)

> *setupCCB (setupLLP, setupKKG, etc...)*
unstows the antenna, sets it to tracking mode, selects the pointing model, and configures the receiver and the backend using default parameters. In practice, it is a shortcut corresponding to this sequence:

```
> antennaSetup=[code]
> receiversSetup=[receiverCode]
> initialize=[receiverCode]
> device=0
> calOff
```

> *sidereal=[sourcename],[RA],[Dec],[epoch],[sector]*
points to the supplied RA-Dec position and temporarily assigns the sourcename label to it. Epoch can be `1950`, `2000` or `-1`, the last one meaning that the provided coordinates are precessed to the observing epoch. The sector keyword forces the cable wrap sector, if needed: its value can be `CW`, `CCW` or `NEUTRAL`. The last option means the system will automatically choose the optimal alternative.

Example: `> sidereal=src12,319.256d,70.864d,2000,neutral`

> *skydip=[El1]d,[El2]d,[duration]*
performs an OTF acquisition at the current azimuth position, spanning in elevation from *El1* to *El2* (both expressed in degrees, with 'd' suffix), in *duration* time, expressed as hh:mm:ss. A recorder must have previously been enabled in order to save the data, and a reference sky position must have been specified via a `sidereal`, `track` or `goTo` command.

> *startSchedule=[project/][schedulename].scd,[N]*
runs schedule *schedulename*.scd (project is the ID of the observing project, it is optional if it has already been input through the `projectCode` command), reading it from line *N*

> *stopSchedule*
immediately stops the running schedule, truncating the acquisition

> *telescopePark*
stows the antenna and parks both the AS and the MS

> *ti*
lists all the active temporised commands

> *track=[sourcename]*
points the antenna, in sidereal tracking, to the specified source, which must be present in the local catalogue

> *tsys*
measures the system temperature (K) in the position the antenna is pointing to. It returns a list of values, one for each section in use. Intermediate steps and calculations are stored in the active logfile

> *wait=[double]*
sets a delay (in seconds) which is applied before the system reads/executes the next command

> winkingMark=[integer]

activates the winking (i.e. fast switching) calibration mark **for the TPB**. The duty cycle is specified by the integer parameter, which indicates the number of off-cal samples after which an on-cal sample is acquired. To switch the mark off, simply command a zero-sample period. This must **always** be done when the winking mark is not needed anymore.

Example: `> winkingMark=3` activates the mark so that it is ON for one sample every 3 samples OFF.

While: `> winkingMark=0` switches the mark off

> wx

returns the current weather parameters: ground temperature (°C), relative humidity (%), atmospheric pressure (hPa), wind speed (km/h).

Appendix C - Output files

The system at present allows the user to write the output data, coming from the integrated backends, in a substantially standard FITS format, carrying the data and a load of ancillary information inside several extensions. A FITS is produced for every subscan composing the observation.

Details on this format are provided in this separate manual `pdf`

Appendix D - Source catalogue

Here follow the sources which are listed in the system catalogue. Their names, as presented here, can be used in schedules (SIDEREAL subscans, inside the LIS file) or in the command:

```
> track=[sourcename]
```

Along with their J2000 Equatorial coordinates, the catalogue records usually contain further information such as apparent size, proper motion, radial velocity, coefficients of the *Ott et al.* polynomials describing the spectra, etc… DISCOS uses these parameters to make computations during certain operations (e.g. when calibration tools are used – under development, they will be described in future releases of this manual):

- 3c48
- 3c84
- 3c123
- 3c147
- 3c161
- 3c218
- 3c227
- 3c249.1
- 3c286
- 3c295
- 3c309.1
- 3c348
- 3c353
- 3c380

- 3c391
- casa (i.e. Cas A)
- cyga (i.e. Cyg A)
- dr21
- ngc7027
- oria (i.e. Ori A)
- taua (i.e. Tau A)
- vira (i.e. Vir A)
- w3oh

Fault report and new feature requests

Users are required to report all faults and misbehaviors by means of the GitHub tracking system. A newbie guide on GitHub and DISCOS GitHub workflow is here *Github HOWTO*

Fault reporting:

When filing an issue, make sure to answer the following questions:

- What version of DISCOS are you referring? Which telescope are you at?
- What did you do?
- What did you expect to see?
- What did you see instead?

Feature requests:

If you find yourself wishing for a feature or enhancement that doesn't exist, you are probably not alone. Open an issue which describes the feature you would like to see, why you need it, and how it should work.

CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 Development guide

2.1.1 Releasing procedure

DISCOS uses a *major.minor.patch* nomenclature, so Nuraghe-1.5.2 has a *major version* of 1, a minor version of 5, and a *patch version* of 2. The *major version* zero (0.y.z) is for initial development, when anything may change at any time, so the public API, the *operator input* commands and the schedule *grammar* should not be considered stable.

Production-ready releases

This section describes the semantic used to assign the version number to the production-ready releases, that is the releases the astronomers use during their normal observations.

New major versions

New *major versions* are exceptional, because they only come when strongly incompatible changes are deemed necessary, and are planned very long in advance.

That means if an astronomer *schedule* can be executed using *DISCOS-1.x.y*, then it will also be executed using any of the next releases of DISCOS with major version of 1.

New minor versions

New *minor versions* are feature releases, so they add functionality in a backwards-compatible manner.

New patch versions

New *patch versions* are bugfix releases that make backwards-compatible bug fixes

In-development versions

We also have *non*-production ready versions which get an additional qualifier: *beta* and *release candidate* (RC). The beta versions are aimed at testing by advanced users, not production use, while the RC is aimed at testing by a group of astronomers, as described in section *Release Candidate*.

Beta

In this stage no more features are accepted. Only bug fixes can now be committed. This is when core developers should concentrate on the task of fixing regressions and other new issues. The new user's manual must be released before the end of this stage.

Release Candidate

A *release candidate* (RC) must be tested by a group of astronomers, called RC testers. They should interact whit the core developers in order to report the malfunctions.

This release can only have bugfixes applied that have been reviewed by other core developers, and can become a final-release only after the RC testers give their approval.

After the final-release is published, the full *development cycle* starts again for the next minor version. Only the *Project manager* can approve new changes to the final releases.

2.1.2 What's new in the next version

Abstract

This section shows the new features we are planning to develop and what's cooking for next releases of DISCOS.

Tracking of Solar System bodies

We plan to add the capability to track any objects (planets, satellites, probes . . .) of the Solar System. The astrometric computation will be based on the SPICE toolkit provided by NASA.

Tracking of TLE based bodies

Two-Line elements is a data format for orbital objects like artificial satellites. With pyephem it is possible to obtain proper ephemerides.

Full support for long term archive

The discos will support the interaction with the long term data archiving. In particular the data format must be completed according to specification. Moreover the control software is responsible to synchronize the data transfer to the archive ingestion area.

2.1.3 Release Notes

Abstract

This section shows up the release notes of all already released version of DISCOS flavours. Starting from discos1.0 the control software will be released for all the telescopes.

DISCOS 1.0

Discos-1.0 (Release Candidate) was released 2018 September the 12th

Remarks

This release is the result of a huge process that involved a large part of our code base in order to migrate the control software from a 32 bits to 64 bits platform. We now support CENTOS 6.8 64 bits and ACS-FEB1027. For a mere matter of manpower, we won't be able to support older version any more.

We also migrate our repository from svn to github (DISCOS organization).

In parallel, in order to improve quality and reliability we introduced new concepts. Our development and bug fixing workflow is now TDD (Test Development Driven), also we developed a number of hardware simulator that, eventually, will drive us to a Continuous Integration approach. Moreover, in order to improve deployment we developed am automatic provisioning system based on ansible.

New features

SARDARA backend

The SARDARA backend integration has been further improved. A better error handling and configuration management has been put in place.The data marging is now completely automatic and quasi-realtime. Support for multi-feed, full stokes observations is now available (SRT only).

With respect to the previous release the configurations changed. This configurations have been tested only at the SRT and can be summarized as follows:

SK00, SK03, SK06, SK77, SC00, SL00, SP00 (L/R) SK00S, SK03S, SK06S, SK77S, SC00S, SL00S, SP00S (Full Stokes)

More details on the various configuration can be found in the user guide. The *setSection* command can be used to tune the configuration of the backend.

A system temperature (*tsys* command) is also available.

A new command *getRms* has been included in order to allows a better tuning of the input levels. More details on the guide.

The user interface for the SARDARA component is now available.

Typically the configuration of the roach backend is done by the schedules. Nevertheless it is possible to manually configure the backend with a sequence of commands:

```
> chooseBackend=SARDARA
> initialize=SK00
> integration=100
> setSection=
```

The setSection syntax follows the general rules:

```
> setSection=input,freq,bw,feed,pol,sr,bins
```

where:

- input: it could only be 0
- freq: start frequency (MHz), placeholder * allowed

- bw: band width (MHz), placeholder * allowed
- feed: feed identifier, it cannot be changed, placeholder * allowed
- pol: polarization, it cannot be changed, placeholder * allowed
- sr: sample rate, it cannot be changed, placeholder * allowed
- bins: number of frequency channels, it could be 1024 or 16384, placeholder * allowed

IF Distributor for the SRT

The SRT has a new IFdistributor capable to perform down conversion and IF filtering. The control of the local oscillator and the down conversion will be transparent with respect to the previous release as the `LPBandReceiver` component can set it up as the other receivers. It is possible to use the *setLO* command from the operator input console:

```
> receiversSetup=LLP
> setLO=2300
```

This new feature is only available using the L band feed (it is not possible to set the local oscillator in the `PPP` configuration).

It is not allowed to set the LO inside the observed sky band. For instance:

```
> receiversMode=XXL5 # Bandwidth 1625:1715
> setLO=1650
Error - Value not allowed. It is within the band...
```

It is also not possible to set a receiver mode in case the current LO value is inside the mode band:

```
> receiversMode=XXL5 # Bandwidth 1625:1715
> setLO=1500 # Ok, it is outside the RF band
> receiversMode=XXL4 # Bandwidth 1300:1800
Error - Mode not allowed. The LO value is within the band...
```

The bandwith value is always positive, while the IFs could be negative, indicating that the IF band is inverted. For instance, in the following case:

```
> receiversMode=XXL5 # Bandwidth 1625:1715
> setLO=2300
```

the receiver panel will show `startFreq=-675` and `bandwidth=90`.

In order to control the IF filtering the command *ifd* (SRT only) has been added. The available filters are:

FILTER ID	Central Freq (MHz)	Bandwidth (MHz)
BW-NARROW	576	115
BW-MEDIUM	640	230
BW-WIDE	768	460
BW-UNFILTERED	see L-Band	

```
> ifd=filter
```

where:

- filter: it could be BW-NARROW,BW-MEDIUM,BW-WIDE,BW-UNFILTERED

System changes

FitsZilla fits file

The ''SECTION TABLE" a been extended by adding two additional columns ''frequency" and ''bandwidth''. This data reports (in MHz) the effective start frequency and bandwidth of the bands sampled by the currently configured backend. Some keyword in summary.fits have now meaningful values (EXPTIME, LOGFILE ecc. for example). A more detailed list of this keywords can found in user documentation.

Bootup procedure

The DISCOS bootup has been reviewed. The frontend is still acscommandcenter based but the ssh protocol is used instead of the acsdeamons. This should both improve the ease of use and speed up the completion of the process.

Change log

A change log of this release may be retrieved from this `document`

NURAGHE 0.6

Nuraghe-0.6 will be released on Friday 29 January 2016.

New features

SARDARAS backend integrated

The Roach component was integrated, even if some caveat must be taken into account.

The Roach backend is connected directly to the focus selector (total power backend) so bandwidth and signal level fed into the Roach are controlled through the total power component. For example the attenuation (command *setAttenuation*) of the Raoch component is binded to the TP.

There are 4 available configuration. Each configuration is specific for a single installed receiver. The following table shows the deafulat value for each setup:

Configuration	FrontEnd	Bandwidth	TP bandwidth	integration	bins
RK00	KKG	1500MHz	1250Mhz	40 ms	1024
RC00	CCB	1500MHz	1250MHz	40 ms	1024
RL00	LLP	1500MHz	2000MHz	40 ms	1024
RP00	PPP	1500MHz	2000MHz	40 ms	1024

A system temperature is also available. The measurment is relative to the bandwith portion selected by the *setSection* command.

The merging phase (the raw data coming from the roach are merged with the meta data coming from the telescope control system) is, temporarly, perfomed offline. At the end of the schedule a command *_send_backend_command* must be issued from a shell of the user workstation.

Typically the configuration of the roach backend is done by the schedules. Also it is possible to manullay configure the backenf to a sequence of commands:

```
> chooseBackend=Roach
> initialize=RK00
> integration=100
> setSection=
```

The setSection syntax follows the general rules:

```
> setSection=input,freq,bw,feed,pol,sr,bins
```

where:

- input: it could only be 0
- freq: start frequency (MHz), placeholder * allowed
- bw: band width (MHz), placeholder * allowed
- feed: feed identifier, it cannot be changed, placeholder * allowed
- pol: polarization, it cannot be changed, placeholder * allowed
- sr: sample rate, it cannot be changed, placeholder * allowed
- bins: number of frequency channels, it could be 1024 or 16384, placeholder * allowed

At present, the Roach component does not provide a user interface.

Acknowledgement to SARDARAS development team: Andrea Melis, Raimondo Concu, Alessio Trois.

Local Oscillator for the LP-band receiver

Now the `LPBandReceiver` component can set the local oscillator, so the component behaves as the other receivers, and it is possible to use the `setLO` command from the operator input console:

```
> receiversSetup=LLP
> setLO=2300
```

This new feature is only available using the L band feed (it is not possible to set the local oscillator in the `PPP` configuration).

It is not allowed to set the LO inside the observed sky band. For instance:

```
> receiversMode=XXL5 # Bandwidth 1625:1715
> setLO=1650
Error - Value not allowed. It is within the band...
```

It is also not possible to set a receiver mode in case the current LO value is inside the mode band:

```
> receiversMode=XXL5 # Bandwidth 1625:1715
> setLO=1500 # Ok, it is outside the RF band
> receiversMode=XXL4 # Bandwidth 1300:1800
Error - Mode not allowed. The LO value is within the band...
```

The bandwith value is always positive, while the IFs could be negative, indicating that the IF band is inverted. For instance, in the following case:

```
> receiversMode=XXL5 # Bandwidth 1625:1715
> setLO=2300
```

the receiver panel will show *startFreq=-675* and *bandwidth=90*.

A 1000MHz low pass filter is applyed to the IFs, so it is not possible to set a LO value that puts the IFs over the low pass filter:

```
> receiversMode=XXL4 # Band 1300:1800
> setLO=200 # IF band: 1100:1600
Error - Value not allowed. The IF bandwidth is outside the low pass filter.
```

System changes

FitsZilla fits file

The "signal" keyword was added to the main header. Presently, three possible values are available:

```
* SIGNAL #indicating that the subscan was performed on-source
* REFERENCE # indicating the telescope was off-source during the subscan
* NONE # meaning the the keyword is unknown or the keyword is not pertinet
        to the current subscan
```

Alias name for fully integrated backends

The command *chooseBackend* now allows the use of aliases for backends. Standard names continue to work regularly. The mnemonic names or aliases are the following:

Standard name	Alias
BACKENDS/TotalPower	TotalPower
BACKENDS/XBackends	XArcos
BACKENDS/Roach	Roach

Continuos calibration

In order to control the calibration mark from an external source (i.e. backends like the DFB or DBBC) the Total Power backend (focus selector) should be configured accordingly. The frontend also requires an identical configuration, this part has been addressed in previous release (see command *externalCalOn*). The command *calSwitch*, when the Total Power is the current backend, has been slightly modified. It now allows for the following sintax :

```
> calSwitch=N # where N is an integer value. One sample every N is acquired
  with the noise cal turned on. This is the old behaviour of the Total
  Power. It also disables the configuration EXT
> calSwitch=OFF # switches off all the previous setup and disables the
  calibration mark control through the Total Power backend
> calSwitch=EXT # the mark will be controlled from external sources
  (DBBC, DFB .....)
```

User Interface

The backend panel associated to the Total Power backend shows an extended status word. Four bits were added:

```
* ExternalMark # it turns yellow when the external mark control is enabled
* FastSwitching # it turns yellow when the total power is sampling with the
  mark swicthing under its direct control. Note: only when the sampling
  is active.
* Calon # it turns yellow when the noise mark is on (non-fast mode)
  under the control of the Backend. This is not implemented yet.
* ZeroRef # it turns yellow when the backend inputs are connected to the
  500hm. It may be yellow during a system temperature measurement.
```

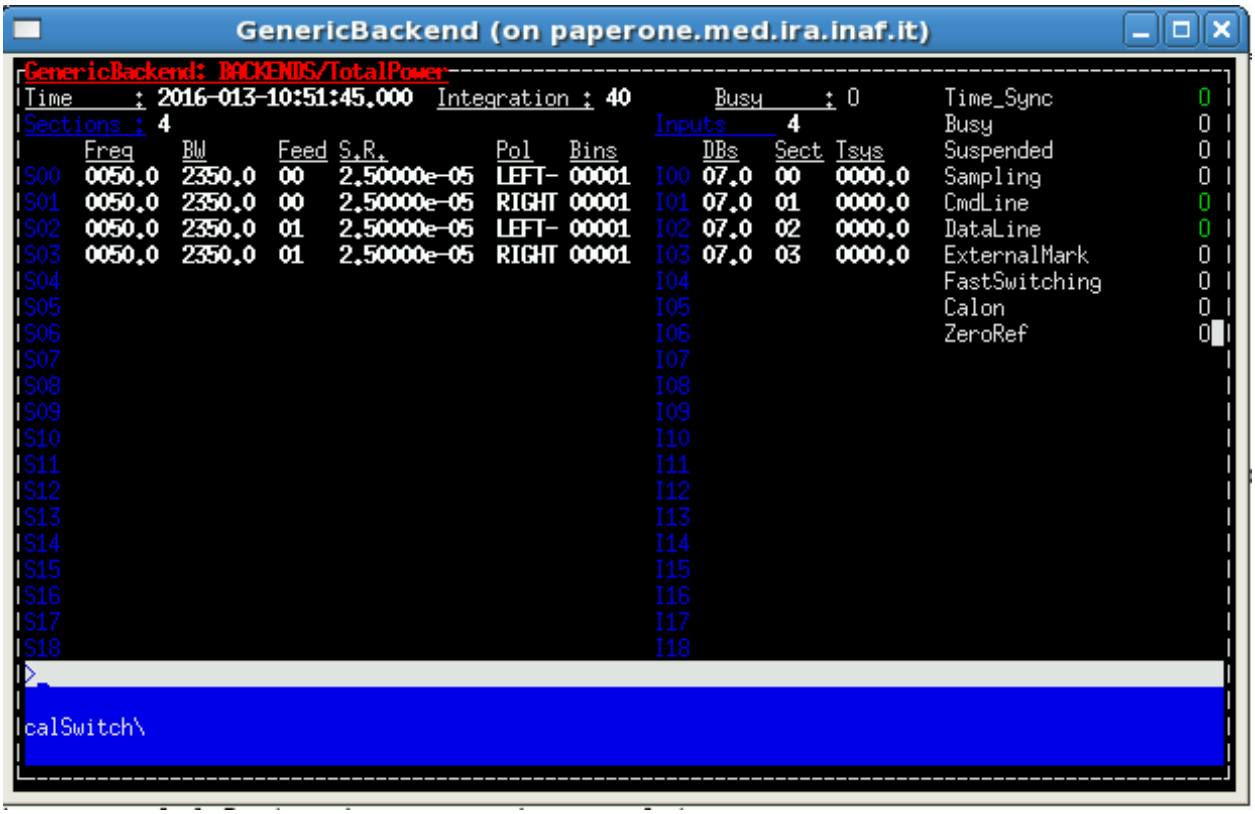


Fig. 1: Figura 1: The Total Power console of Nuraghe 0.6

Auto stow for wind

The telescope will be parked (and the running schedule stopped) if an excessive wind is detected.

Change log

A complete and browsable change log of this release may be consulted from our tracking system.

NURAGHE 0.5

Nuraghe-0.5-RC (Release Candidate) will be released on Monday 22 June 2015. The release date of the stable version depends on the astronomer feedback, as explained in *Release Candidate*, so it is not yet decided.

New features

Dewar positioning

The *DewarPositioner* component is in charge of compensating the field rotation when dealing with multi-pixel receivers (if equipped with a derotation system). Details on supported modes, configurations and more precise information required when dealing with the derotation system are available *here*.

K band derotator control

The derotation system installed with the K band receiever is now completely under control.

Frequency tracking

In order to fulfill the requirements for spectroscopy observations, the system can now keep a requested rest frequency in centre of the band of the backend (section). Both frontend and backend local oscillator could be tuned during this operation. This capability is now supported from both command line and schedule.

System changes

Continuous calibration

Some backends can quickly switch the calibration diode on and off (Total Power, DFB). Up to now this capability had to be enabled in the frontend control boards. From now on the operation is fully automated and does not require the human intervention any more.

FitsZilla fits file

Some keywords added (mainly belonging to spectroscopy). In the output scan folder, when the scan is closed a *summary.fits* is created. The file contains just the main header with some keywords describing the current scan.

Minor servo system

Gregorian and BWG default configuration

When using the Gregorian or BWG focus only, the *Antenna overall setup* automatically configures the subreflector in order for the elevation to be tracked, and assumes the active surface is enabled. That means the minor servo setup automatically issues the following commands:

```
> setServoElevationTracking=ON
> setServoASConfiguration=ON
```

so they are not required any more. The user is asked to issue the correct configuration in case other behaviours are required.

LP band receiver setup

From now on, the LP setup procedure turns off the encoder of the PFP theta axis. That allows to eliminate the encoder RFIs when observing in L and P bands.

User interface

A few changes to the existing panels. Some parameters from the telescope, mainly from the new features are now monitored, and the minor servo console reports only four fields: `Current Setup`, `Tracking`, `Status`, and `Motion Info`, as shown in Figure 1.

Fig. 2: Figura 1: The new minor servo console of Nuraghe 0.5

Now the red color is only used in case of improper functioning, either when the `Status` field indicates `FAILURE` or the minor servo system is not `Tracking` the commanded position.

Logging

The user commands from operator console are now logged. Logging system is now more robust. The management of the system log file is now more crafty, avoiding the creation of bigger log files. System log files are daily rotated and stored for 1 year.

Xarcos

Fully supported and integrated in the system. Integration time is now in milliseconds.

Schedules

The schedule now allows to call parametric configuration procedures.

Testing Integration

The development framework now integrates tools for automatic tests development and execution.

Build automation

The build process has been reorganized in order to have nightly builds and build status reports

Change log

A complete and browsable change log of this release may be consulted from our tracking system.

ESCS 0.4

ESCS 0.4 was released March the 24th, 2016. The previuos version was released on October 2012 so this release contains lots of new updates and improvements coming from 3 years and a half of developments mainly driven by SRT requirements.

New features

Frequency tracking

In order to fulfill the requirements for spectroscopy observations, the system can now keep a requested rest frequency in centre of the band of the backend (section). Both frontend and backend local oscillator could be tuned during this operation. This capability is now supported from both command line and schedule.

Continuous calibration

TotalPower backend can quickly switch the calibration diode on and off (Total Power). This capabilities can also be exposed to other backends (DBBC) after a proper configuration. THis is now fully under software control.

Xarcos

Fully supported and integrated in the system. Integration time is now in milliseconds.

System changes

FitsZilla fits file

Some keywords added (mainly belonging to spectroscopy). In the output scan folder, when the scan is closed a *summary.fits* is created. The file contains just the main header with some keywords describing the current scan. The "signal" keyword was added to the main header. Presently, three possible values are available:

```
* SIGNAL #indicating that the subscan was performed on-source
* REFERENCE # indicating the telescope was off-source during the subscan
* NONE # meaning the the keyword is unknown or the keyword is not pertinet
        to the current subscan
```

Alias name for fully integrated backends

The command *chooseBackend* now allows the use of aliases for backends. Standard names continue to work regularly. The mnemonic names or aliases are the following:

Standard name	Alias
BACKENDS/TotalPower	TotalPower
BACKENDS/XBackends	XArcos

User interface

The console will now present an additional panel (genericBackend) that shows the current configuration and status of XArcos.

Manual Recording

The user now could manage a recording from the currently selected backend to the defualt data writer(see commands *chooseBackend* and *chooseRecorder*). Three new commands are now availble from the operatorInput:

```
> initRecording : prepares the data recording
> startRecording: starts the recording and file creation
> terminateScan: closes the scan
```

Logging

The user commands from operator console are now logged. Logging system is now more robust. The management of the system log file is now more crafty, avoiding the creation of bigger log files. System log files are daily rotated and stored for 1 year.

Schedules

The schedule now allows to call parametric configuration procedures.

Testing Integration

The development framework now integrates tools for automatic tests development and execution.

Build automation

The build process has been reorganized in order to have nightly builds and build status reports

Change log

A complete and browsable change log of this release may be consulted from our tracking system.

ESCS 0.5

ESCS 0.5 was released by March, 2017. SVN release 1692. Some features are not fully completed. Their inclusion in the relase is meant to allow further testing and technical commissioning in sight of the final deployment of the capability.

New features

SARDARAS backend

The Roach component, able to control the SARADARAS backend, has been pre-released. Some caveats must be taken into account.

This version is directly derived from the SRT installation. Even though at the SRT is fully functional, at Medicina some more testings are required.

The Roach backend is connected directly to the focus selector (total power backend) so bandwidth and signal level fed into the Roach are controlled through the total power component. For example the attenuation (command *setAttenuation*) of the Raoch component is binded to the TP.

There are some available configurations. Each configuration is specific for a single installed receiver. More confugrations will be avilable when the commisisoning stage is over. The following table shows the deafulat value for each setup:

Configuration	FrontEnd	Bandwidth	TP bandwidth	integration	bins
RK00	KKC	1500MHz	1250Mhz	40 ms	1024
RC00	CCC	1500MHz	1250MHz	40 ms	1024

A system temperature is also available. The measurment is relative to the bandwith portion selected by the *setSection* command.

The merging phase (the raw data coming from the roach are merged with the meta data coming from the telescope control system) is perfomed almost real time so the user can check his data.

Typically the configuration of the roach backend is done by the schedules. Also it is possible to manullay configure the backenf to a sequence of commands:

```
> chooseBackend=Roach
> initialize=RK00
> integration=100
> setSection=
```

The setSection syntax follows the general rules:

```
> setSection=input,freq,bw,feed,pol,sr,bins
```

where:

- input: it could only be 0
- freq: start frequency (MHz), placeholder * allowed
- bw: band width (MHz), placeholder * allowed
- feed: feed identifier, it cannot be changed, placeholder * allowed
- pol: polarization, it cannot be changed, placeholder * allowed

- sr: sample rate, it cannot be changed, placeholder * allowed
- bins: number of frequency channels, it could be 1024 or 16384, placeholder * allowed

At present, the Roach component does not provide a user interface.

Acknowledgement to SARDARAS development team: Andrea Melis, Raimondo Concu, Alessio Trois.

7 GHz receiver

Observations with 7GHz receiver are now available. Information on the receiver can be found here.

The configuration can be enabled by typing the proper command:

```
> setupCHC
```

Other useful information follows:

Available bands: RF 5900-6300 MHz LO=5900, IF 100-500 MHz RF 6500-6900 MHz LO=6400, IF 100-500 MHz

The default configuration is the 6.7GHz one. Default bandwidth for the totalpower is 300 MHz, corresponding to this command:

In order to setup backend up: .. code-block:: discos

> initialize=CHC

System changes

injectCommand

It is now available a program that allows to issue commands [RAL command] to the control system without directly accessing the operator input. Any authorized user (member of the system manager group) can use this command from its shell. The command accept just one argument, i.e. the RAL command to issue. It should be noted that a complex command should be included between “”.

```
> injectCommand [RALCommand]
```

Support for long-term archive

The long-term archive requires an external tool to allow data and meta data collection (fits file, schedules and logs generated during the observation). The tool is called finalizer and is available under the DISCOS git oranization. The finalizer will run in the background and every duty cycle and it will:

- check the if the lock file exists. if it exists exit, otherwise continue to following step
- check if the are schedule repot files.
- based on the content of each report file pack all required data in a tar file
- copy the tarball in the sharable (with long-term archive) area

Generally the tarball produced by the finalizer should contains:

- the schedule files
- the corresponding log file
- all the scans recorded during the schedule execution.

The information stored in the report are:

- the location of the backup schedule files
- the location of the corresponding log file
- all the folder (scans) created during on execution of the schedule

Minor Changes

- **XARCOS**: added a new XK01 backend mode, solved a bug taht causes the system to crach on random basis
- **TotalPower**: added *calOn* and *calOff* commands. At the moment these are only used in Noto installation
- **FTrack**: fixed some bugs.
- **ReceiversBoss**: new configuration takes into account the new control PC at Medicina station.
- **Scheduler**: inserted support for integration of the *finalizer*.

Change log

A complete and browsable change log of this release may be consulted from our tracking system.

ESCS-Noto-0.2

ESCS-Noto, version 0.2 was released December the 2nd, 2016. This release was the evolution of the prototype installaed Febraury 2015 and contains lots of new updates and improvements. in order to have an overall view of capabilities and features, take a look at the release notes of the other production lines and respective user manuals.

New features

Weather station

The component to read weather parameters is now completed and stable.

Calibration signal

The Phase calibration signal is now under control.

Field System

Now the Field System for VLBI observation is a client application of DISCOS. Almost all required features should now be implemented by forwarding all Fully supported and integrated in the system. Some further tests should have been carried on.

Active surface

The AS can now be direclty controlled with DISCOS. The low-level M&C is still delegated to the ASCU (ACtive Surface Control Unit). The available commands follow below:

command	Result
asOn	Activate the AS tracking
asPark	park the acrive surface (all actuators are brought to hardware limit)
asOff	Bring the AS to 45° degrees of elevation

Subreflector

The subrelftor can now be direclty controlled with DISCOS. The low-level M&C is still delegated to the SCU (Subreflector Control Unit).

command	Result
TDB	TDB

System changes

Change log

Not yet available

2.1.4 Development workflow

Abstract

This section illustrates how the development of new features and the bugs fixing is pursued in our team. This workflow is highly TDD as detailed in *Test Driven Development (TDD) in DISCOS*.

Write a hardware simulator

Hardware Simulator and related tests. The same tests must pass both for the hardware and the simulator. An important goal is to test the hardware interface robustness, in order to simplify the diagnosis of component malfunctions and the DISCOS maintenance. In that way, if we have to upgrade an hardware firmware or an interface, we just have to run the tests to be sure the upgrade does not break the system.

Write a hardware communication library

We have to write a hight level library in order to communicate to the hardware from the component, in order to simplify the component code, its maintenance and its readability, and also the tests (we have to mock the library, not the protocol).

Write the component

In this section we will see how to write a component, adopting the TDD in each and every step. We will write a receiver component, both in Python and in C++.

IDL interface

We would like the example to be as clear and simple as possible, so the receiver will have just a few methods, one property, one notification channel and an external component reference. The interface (located in *Common/Interfaces/DummyInterface*) is the following:

```
#ifndef __DUMMY_RECEIVER__IDL__
#define __DUMMY_RECEIVER__IDL__

#include <baci.idl>
#include <maciErrType.idl>
#include <ComponentErrors.idl>
#include <ManagmentDefinitions.idl>

#pragma prefix "alma"


module DummyReceiver {

    struct DummyReceiverDataBlock {
        boolean ready;
    };

    const string DUMMY_RECEIVER_DATA_CHANNEL = "DummyReceiverData";

    interface MyReceiver:
        ACS::CharacteristicComponent,
        Management::CommandInterpreter
    {
        /** Take a configuration code (VLBI or SD) and set the receiver */
        void setConfiguation(in string code) raises␣
↪(ComponentErrors::ComponentErrorsEx);


        /** Return the actual configuration (VLBI, SD) */
        string getConfiguration();


        /** Set the local oscillator value */
        void setLO(in ACS::doubleSeq value);


        /** Get the local oscillator value */
        ACS::doubleSeq getLO();


        readonly attribute ACS::RWdouble temperature;
    };
};

#endif
```

The default configuration is described below:

```
>>> from Acspy.Clients.SimpleClient import PySimpleClient
>>> client = PySimpleClient()
>>> receiver = client.getComponent('RECEIVERS/MyReceiver')
>>> receiver.getConfiguration() # Default configuration
'SD'
>>> receiver.getLO() # Default LO value
[1500, 1500]
```

We can change the configuration at any time:

```
>>> receiver.setLO([2000, 2000])
>>> receiver.getLO()
[2000, 2000]
```

Eventually, the receiver has to monitor the `LO` component status, by checking the `LO.isReady()` method. When that method returns `true`, `DummyReceiverDataBlock.ready` field must be `true`, otherwise it must be `false`.

As a first step we will write the functional tests.

Component directory

We create the component directory:

```
$ mkdir Common/Servers/DummyReceiver
$ cd Common/Servers/DummyReceiver/
```

and we use the `getTemplateForTest` command in order to create the *test* directory:

```
$ getTemplateForTest
$ tree
.
`-- test
     |-- Makefile
     |-- external
     |   `-- __init__.py
     |-- functional
     |   `-- __init__.py
     |-- pyunit
     |   `-- __init__.py
     |-- results
     `-- unittest.cpp

5 directories, 5 files
```

We are ready to implement the component.

Python components

I create the directory and the schema for the component. So start from here.

C++ components

Integration tests

Integrate the component in DISCOS

Write the integration tests (subsytems communication), ...

Beta release

Test the *Beta* in the dev-environment and then in production.

Release candidate

Give the *Release Candidate* to the RC testers (in procution)

Final release

See *Production-ready releases*.

How to deal with a bug

Once we have confirmed the existence of a bug, we do not never start fixing it until we have reproduced that bug in a unit test. So, if we want to fix a bug, we have to:

- work in the discos-x-dev environment
- write a small, simple and fast test that reproduces the bug
- ensure the test breaks
- fix the code
- run the test to confirm it now passes
- run all the regression tests to confirm that the changes does not brake something else
- leave the test in the *test* directory (forever) in order to ensure that the bag does not reoccur
- commit the changes
- update the changes in production
- run all the tests in production

This is the only way to ensure the DISCOS stability and robustness, and you will soon realize the TDD is not at all a lose of time.

2.1.5 DISCOS components

Abstract

DISCOS is designed following a software architectural pattern called *component-container model*. During the normal execution, the ACS framework runs 10xx DISCOS components, each one has in charge a specific hardware management (control and monitoring). The goal of this document is to comprehensively explain the APIs

of some DISCOS components (CLARIFY: ExternalClient because currently is the only one that provides an ACS-independent DISCOS interface, but also other component that we want to explain the API since thier non-trivial configuration).

DewarPositioner

Abstract

Questo documento è il manuale di utilizzo del component di DISCOS che gestisce i derotatori, chiamato `DewarPositioner`. Il documento è suddiviso in due parti: la sezione *Guida per l'astronomo* spiega come interagire con il derotatore tramite la console dell'*operator input*, mentre la sezione *Descrizione delle API* descrive come interagire con il component in modo programmatico, per cui è la parte di riferimento per chi intende scrivere dei programmi che utilizzano il `DewarPotitioner`.

Sia la parte per gli astronomi che quella per i programmatori sono suddivise in cinque sezioni, che riguardano rispettivamente il *setup*, le *configurazioni*, il *parcheggio*, il *riavvolgimento* e una lista dei comandi o metodi.

Guida per l'astronomo

In questa sezione viene descritto come interagire con il `DewarPositioner` tramite la sua console di *operator input*. Per avviare tale console, si esegua il comando `operatorIpunt` specificando come target `RECEIVERS/DewarPositioner`:

```
$ operatorInput RECEIVERS/DewarPositioner
```

Esecuzione del setup

Il `DewarPositioner` gestisce tutti i derotatori alla stessa maniera, per cui in fase di *setup* è necessario indicare il codice del derotatore che si intende utilizzare. Il setup viene eseguito con il comando `derotatorSetup`:

```
> derotatorSetup=CODE
```

I codici sono i medesimi che vengono utilizzati per effettuare il setup dell'antenna. Ad esempio, se si vuole utilizzare il derotatore del ricevitore in banda K di SRT:

```
> derotatorSetup=KKG
```

Al termine del *setup* il derotatore sarà pronto per essere utilizzato. Il comando `derotatorGetActualSetup` restituisce il setup attuale, mentre il comando `derotatorIsReady` ci dice se il derotatore è pronto per essere movimentato:

```
> derotatorGetActualSetup
KKG
> derotatorIsReady
True
```

Durante il *setup* al derotatore viene comandata la posizione `0`, che è quella scelta per l'allineamento iniziale. Ad esempio, per il il derotatore del ricevitore in banda K di SRT, la posizione iniziale è quella in cui i tre feed 3, 0, 6 sono paralleli all'orizzonte, con il feed 3 a est.

Possiamo verificare la posizione con il comando `derotatorGetPosition`:

```
> derotatorGetPosition
0.0084d
```

La `d` sta ad indicare che il valore numerico rappresenta un angolo espresso in gradi decimali.

Note: Il valore restituito potrebbe essere diverso da zero, per qualche cifra decimale, come nell'esempio appena mostrato. Questo è normale, ed è dovuto al fatto che il movimento del motore avviene a step, per cui i valori di posizionamento sono discreti.

Il *setup* infine imposta i valori di default per la configurazione e la modalità di riavvolgimento:

```
> derotatorGetConfiguration
FIXED
> derotatorGetRewindingMode
AUTO
```

Parleremo delle configurazioni e del riavvolgimento nelle prossime sezioni.

Configurazione

Il `DewarPositioner` ha sette configurazioni: *fixed*, *best space coverage*, *bsc optimized*, *custom*, *custom optimized*, *aligned* e *aligned optimized*. Il comando `derotatorSetConfiguration` consente all'utente di impostare la configurazione desiderata, mentre il comando `derotatorGetConfiguration` di leggerla:

```
> derotatorSetConfiguration=FIXED
> derotatorGetConfiguration
FIXED
> derotatorSetConfiguration=CUSTOM
> derotatorGetConfiguration
CUSTOM
```

Le configurazioni *aligned* e *aligned_opt* non sono al momento disponibili:

```
> derotatorSetConfiguration=ALIGNED
Error - configuration ALIGNED not available
```

Vediamo ora nel dettaglio le varie configurazioni, suddividendole in *statiche* e *dinamiche*.

Configurazioni *statiche*

Nelle configurazioni statiche la posizione del derotatore non cambia al variare della posizione dell'antenna o dell'asse di scansione.

Configurazione *fixed*

In questa configurazione, che è quella che viene impostata per default dal *setup*, la posizione del derotatore viene mantenuta fissa al variare della posizione dell'antenna, e questo è il motivo per cui le è stato assegnato il codice identificativo `FIXED`.

Nella configurazione `FIXED` è possibile impostare la posizione del derotatore utilizzando il comando `derotatorSetPosition`:

```
> derotatorSetConfiguration=FIXED
> derotatorSetPosition=30d
> derotatorGetPosition
30d
```

Se il derotatore si trova in una certa posizione P_x e viene impostata la modalità `FIXED`, viene tenuta la posizione P_x. Il derotatore quindi non viene riportato in posizione di zero:

```
> derotatorGetPosition
50d
> derotatorSetConfiguration=FIXED
> derotatorGetConfiguration
FIXED
> derotatorGetPosition
50d
> derotatorSetPosition=10d
> derotatorGetPosition
10d
```

Configurazioni dinamiche

Nelle configurazioni *dinamiche*, a differenza di quelle statiche, il `DewarPositioner` aggiorna la posizione del derotatore in funzione della posizione dell'antenna, al fine di compensare l'angolo parallatico (più un eventuale contributo del *galactic parallactic angle*, a seconda dell'asse di scansione).

Nelle configurazioni dinamiche la posizione del derotatore è data dalla seguente equazione:

$$P(az, el) = P_{is} + P_{ip}(az_0, el_0) + P_{dp}(az, el)$$

dove `az` ed `el` sono rispettivamente l'azimuth e l'elevazione dell'antenna, mentre:

- P_{is} è la *initial static position*, ovvero una posizione (letta dal Configuration Data Base, CDB) che non dipende dall'azimuth ed elevazione dell'antenna ma solamente dall'asse di scansione
- P_{ip} è la *initial parallactic position*, ovvero il valore dell'angolo parallatico ad inizio scansione: questo dipende sia dall'asse di scansione (vale 0 per gli assi `HOR_LON` e `HOR_LAT`) sia dal dal puntamento (azimuth, elevazione e settore)
- P_{dp} è il delta di angolo parallatico rispetto a P_{ip}

Nelle configurazioni ottimizzate (`BSC_OPT` e `CUSTOM_OPT`) si ha $P_{ip} = 0$. Queste configurazioni sono utili quando si utilizza un derotatore con un limitato range di escursione (ad esempio, quello del ricevitore S-Band di SRT).

Oltre al fatto che sia ottimizzata o meno, ciò che differenzia una configurazione dinamica dall'altra è il valore della posizione iniziale P_{ip}, perchè la funzione di compensazione dell'angolo parallatico non cambia, e vale 0 quando l'asse di scansione è `HOR_LAT` o `HOR_LON`, è il risultato della funzione `getParallacticAngle()` quando l'asse di scansione è `TRACK`, `EQ_LON`, `EQ_LAT` o `GCIRCLE`:

```
def getParallacticAngle(latitude, az, el):
    p = atan2(-sin(az), tan(latitude)*cos(el) - sin(el)*cos(az))
    return degrees(p)
```

mentre è dato dalla funzione `getGalacticParallacticAngle()` quando l'asse è `GAL_LON` o `GAL_LAT`:

```
def getGalacticParallacticAngle(latitude, az, el, ra, dec):
    p = PosGenerator.getParallacticAngle(latitude, az, el)
    g = PosGenerator.getGalacticAngle(ra, dec)
```

(continues on next page)

(continued from previous page)

```
    return p + g

def getGalacticAngle(ra, dec):
    # North celestial pole coordinates - (j2000)
    # ncp = ('12 51 26.28', '27 07 41.7')
    ra0 = 3.3660332687500043
    dec0 = 0.47347728280415174
    g = atan2(sin(ra-ra0), cos(dec)*tan(dec0) - sin(dec)*cos(ra-ra0))
    return degrees(g)
```

Quando viene impostata una configurazione, la posizione del derotatore non viene aggiornata, visto che non è ancora noto l'asse di scansione. L'aggiornamento viene comandato nel momento in cui DISCOS avvia lo scan.

Configurazione *best space coverage*

Il codice associatò a questa configurazione è `BSC`:

```
> derotatorSetConfiguration=BSC
> derotatorGetConfiguration
BSC
```

In questa configurazione il valore della posizione iniziale P_{ip} viene letto da un database di configurazione ed è tale da garantire che i feed vengano disposti in modo da avere la miglior copertura spaziale della sorgente durante la scansione.

Note: Tipicamente la miglior copertura viene ottenuta equispaziando, quando possibile, i beam nella direzione ortogonale a quella di scansione (se si sta facendo una scansione in azimuth i feed vengono equispaziati in elevazione, in modo da ottimizzare la scansione dell'area osservata).

Quando è impostata la modalità `BSC`, all'utente non è consentito il posizionamento del derotatore:

```
> derotatorSetConfiguration=BSC
> derotatorSetPosition=50d
Error - setPosition() not allowed in BSC configuration
```

In questa modalità l'insieme dei feed posizionati in modo da garantire la massima copertura spaziale sono stabiliti a priori (ad esempio per il banda K sono i feed 3, 0 e 6, con il 3 a est).

Configurazione BSC *optimized*

Questa configurazione è analoga alla *best space coverage* ma a differenza di quest'ultima, come abbiamo detto nella sezione *Configurazioni dinamiche*, il valore dell'angolo parallatico iniziale P_{ip} non viene preso in considerazione. Questa configurazione è identificata dal codice `BSC_OPT`:

```
> derotatorSetConfiguration=BSC_OPT
> derotatorGetConfiguration
BSC_OPT
```

Analogamente alla `BSC`, all'utente non è consentito il posizionamento del derotatore:

```
> derotatorSetConfiguration=BSC_OPT
> derotatorSetPosition=50d
Error - setPosition() not allowed in BSC_OPT configuration
```

Configurazione *custom*

In questa configurazione la posizione iniziale statica P_{is} viene impostata dall'utente, e per tale motivo a questa configurazione è stato assegnato il codice identificativo `CUSTOM`.

Configurazione *custom* optimized

Analogamente a quanto abbiamo visto per la configurazione `BSC_OPT`, anche in questo caso ciò che differezia la `CUSTOM_OPT` dalla `CUSTOM` è la mancanza del contributo dell'angolo parallattico iniziale P_{ip}.

Interrompere l'aggiornamento

Se si vuole interrompere l'aggiornamento della posizione, si deve impostare la configurazione *fixed*. In questo caso il derotatore si fermerà all'ultima posizione comandata.

Riavvolgimento

Il derotatore ha una corsa limitata, per cui la sua posizione ha un limite massimo e uno minimo. Ad esempio, per il derotatore del ricevitore in banda K di SRT la massima posizione raggiungibile è 125.23 gradi, mentre la minima è di -85.77 gradi:

```
> derotatorSetup=KKG
> derotatorGetMaxLimit
125.2300d
> derotatorGetMinLimit
-85.7700d
```

Quando il derotatore sta aggiornado la sua posizione per tener conto dell'angolo parallattico, è quindi possibile che si arrivi a fine corsa. In questo caso, per default il derotatore viene riavvolto in modo automatico, e l'effetto del riavvolgimento è che il feed più vicino al fine corsa viene rimpiazzato da un altro, in modo da garantire che il derotatore abbia (durante il proseguo dello scan) la massima corsa.

Durante il riavvolgimento, la console dei ricevitori indicherà che il derotatore è in fase di riavvolgimento, e il campo *rewindingOffset* della medesima console riporterà l'offset che avrà la posizione al termine del riavvolgimento.

Descrizione delle API

Di seguito l'elenco completo dei metodi:

```
setup(CODE) # CODE: KKG, ...
getActualSetup()
getCommandedSetup()
isReady()
isConfigured()
park()
setPosition(POSITION)
getPosition()
setConfiguration(CODE) # CODE: FIXED, BSC, ...
clearConfiguration()
getConfiguration()
startUpdating(AXIS, AZ_SECTION)
```

(continues on next page)

(continued from previous page)

```
stopUpdating()
isTracking()
isUpdating()
isSlewing()
setRewindingMode(MODE) # MODE: MANUAL or AUTO (default)
clearRewindingMode()
setAutoRewindingFeeds(NUMBER_OF_FEEDS)
getRewindingMode()
rewind(NUMBER_OF_FEEDS)
isRewindingRequired()
isRewinding()
getStatus() # Restituisce lo status pubblicato dal NC
... # TODO
```

2.1.6 Protocols

Abstract

This section lists some of the protocols we use to communicate to the hardware. If you want to integrate a device in DISCOS (a backend, a receiver, etc.), you should read this document in order to find for for a protocol to choice. If either your device is not listen in this document or for technical reasons your hardware does not suit the protocols listed here, then please do not go ahead on your own initiative but please, contact the *Project manager* or the *Core developers*.

Backend protocol

This section describes the protocol used by DISCOS to communicate with external backends. Here you can find the protocol definition, the grammar definition and a simple example implementation of the protocol server as a twisted protocol. This definition is liberally inspired by the one used by KAT telescope for device communication ([KAT]) as we found that work of really good quality.

Protocol Version	1.2
Last revision	25/01/2016
Status	PRODUCTION

Introduction

DISCOS BACKEND PROTOCOL consits of newline separated text messages sent synchronously over a TCP/IP connection. Messages can be of two kinds: request and reply. Requests are sent from a client to a server, while replies are sent from the server in response to a client's request. Each request must receive one and only one reply message back and in this kind of topology the client will be tipically represented by a DISCOS component, while the server is represented by the backend controller. This version of the protocol does not impose any constraint to the number of clients connecting to a server but leaves to the clients the responsibility of orchestrating their requests in a consistent way. Implementations consisting of a single client will be the first and foremost ones.

Here you can find a quick list of the requests defined by the protocol, which will be better described in the next sections:

Request	Description
status	get the backend status code
version	get the server protocol version
get-configuration	get the backend actual configuration
set-configuration	set a new backend configuration
get-integration	get the backend integration time
set-integration	set a new integration time
get-tpi	ask the backend a total power level read
get-tp0	ask the backend a zero level read
time	get the backend time
start [timestamp]	start the acquisition [at a given time]
stop [timestamp]	stop the acquisition [at a given time]
set-section	configure a section of the backend
cal-on [interleave=0]	(de)activate the calibration mark
set-filename	tell the backend the output file name
convert-data	instructs the backend to convert file

Handshake

The protocol is a connected protocol. This means that a client must establish a connection with the server and keep it open for message exchange. In this preliminary version a very simple handshake is required: on every new connection the server will write to the client a REPLY message containing the version of the protocol that the server is implementing. Further decisions are left to the client.

Example:

```
client.open_connection( server_ip, server_port )

server -> client
reply: "!version,1.0.1\r\n"
```

client now can continue communication or close it because it detects an incomptible server version

Note that the message is exactly the same one published in reply to the *version* command.

Messaging Protocol

Communication consists of a number of messages, each message consisting of a line of text. The protocol supports requests and replies messages, identified by the first character of the message: requests are indicated by '**?**', replies by '**!**'. Each message is made up of a **type code** followed by the message **name**, and optionally by one or many message **arguments**; name and arguments are separated one each other using the character '**,**' .

A reply is necessary for every request, however the nature of the reply may change depending on the request:

- The reply message should have the same name as the request message.
- The first argument of a reply message should always be a return code.
- A return code of **ok** indicates successful processing of the request, while anything else indicates failure.
- The recommended failure strings are **invalid** (for malformed requests) and **fail** (for valid requests which could not be processed).
- On success, further reply arguments are specific to the type of request.

- In the case of failure a second argument should describe the failure in more detail and in human-readable form.

Return Code	Description
ok	Request successfully processed. Further arguments are request specific.
invalid	Request malformed. Second argument is a human-readable description of the error.
fail	Valid request that could not be processed. Second argument is a human-readable description of the error.

Line Separation

Each message is terminated by the sequence **CR LF** (carriage-return + line feed) as specified by the TELNET standad ([TELNET]) . This will make the protocol easily usable also for debug purposes using simple telnet clients.

Message Grammar

The message grammar is described in extended BNF [EBNF] where:

- Optional items are enclosed in square brackets
- Items repeating 0 or more times are suffixed with a *
- Items repeating 1 or more times are suffixed with a +
- Alternative choices in a production are separated by the '|' symbol
- Set difference is indicated by the '/' symbol

The grammar is defined as:

```
  <message> ::= <type> <name> <arguments> <eol>
     <type> ::= "?" | "!"
     <name> ::= alpha (alpha | digit | "-")*
<arguments> ::= ("" | <separator> <argument>) [<arguments>]
      <eol> ::= newline
<separator> ::= ","
 <argument> ::= (<plain> | <escape>)+
    <plain> ::= character / <special>
  <special> ::= backslash | null | newline | carriage-return | escape | tab
   <escape> ::= "\" <escapecode>
<escapecode>::= "\" | "t" | <separator>
```

Note that arguments can contain spaces and tabs and are limited only by commas and newlines at the end of the message.

Data Types

Being the protocol string based, whenever we need to transmit other data types they must be encoded into strings in an unequivocable manner. This is defined as per the table below:

- **integer** as formatted by printf("%d",i). i.e. 10 -15
- **float** as formatted by printf("%f",f) i.e. -1209087123.234234 1.0
- **boolean** True as 1 and False as 0 i.e. 1, 0
- **timestamp** is the number of centinanoseconds since epoch. It is the format used by the ACS framework and it can be obtained by multiplying the unix timestamp in floating point format by 10000000. All times are intended to be **UT** i.e. 143092278297088300

Request and Reply Messages

For each command we give a brief description of how the command can be used and the description of the reply to the command. We then provide a simple example.

status

Asks the status of the backend. The request message has no arguments. The Reply message has 3 arguments:

- **timestamp** the timestamp of the answer message according to the backend clock
- **status code** in normal working condition should be **ok**, any other value should be used for representing any possible failure state
- **acquiring** is a boolean value indicating if the backend is performing an acquisition, can be 0 for *false* or 1 for *true*

Example communication:

```
request: "?status\r\n"
  reply: "!status,ok,1430922782.97088300,ok,0\r\n"

request: "?status\r\n"
  reply: "!status,ok,1430922782.97088300,clock error,0\r\n"
```

version

Asks the backend server what version of the protocol it is implementing. The Request message has no argument. The Reply message has 1 argument:

- **version id** a string representing the protocol version

Example communication:

```
request: "?version\r\n"
  reply: "!version,1.0.1\r\n"
```

get-configuration

Asks the backend server what configuration is loaded at the moment. Request message has no argument. The Reply message has 1 argument:

- **configuration id** a string representing the loaded configuration

If the backend has not yet been configured a special value of **unconfigured** is returned as reply argument.

Example communication:

```
request: "?get-configuration\r\n"
  reply: "!get-configuration,K2000\r\n"

request: "?get-configuration\r\n"
  reply: "!get-configuration,unconfigured\r\n"
```

set-configuration

Instruct the backend to configure itself according to the specified configuration code given as argument. Reply message has no argument. Request message has one argument:

- **configuration id** a string identifying the configuration to be loaded

Example communication:

```
request: "?set-configuration,K2000\r\n"
  reply: "!set-configuration,ok\r\n"

request: "?set-configuration,nonexistent\r\n"
  reply: "!set-configuration,fail,cannot find configuration 'nonexistent'\r\n"
```

get-integration

Asks the backend server what integration time is configured. Request message has no argument. The Reply message has 1 argument:

- **integration time** an integer representing the integration time configured in ms.

If the backend has not yet been configured a special value of **0** is returned as reply argument.

Example communication:

```
request: "?get-integration\r\n"
  reply: "!get-integration,20\r\n"

request: "?get-integration\r\n"
  reply: "!get-configuration,0\r\n"
```

set-integration

Instruct the backend to configure itself with the specified integration time given as argument. Reply message has no argument. Request message has one argument:

- **integration time** an integer representing the integration time to be set in ms.

Example communication:

```
request: "?set-integration,20\r\n"
  reply: "!set-integration,ok\r\n"

request: "?set-integration,wrong\r\n"
  reply: "!set-integration,fail,integration time must be an integer number"
```

get-tpi

Ask the backend a total power level read. Request message has no argument. Reply message has as many arguments as the number of sections configured in the backend:

- **TPI_sec0, TPI_sec1, ...** a sequence of floating point numbers representing the total power intensity level of each section of the backend

Example communication:

```
request: "?get-tpi\r\n"
  reply: "!get-tpi,ok,900.00,1240.00\r\n"
```

get-tp0

Ask the backend a total power read with zero level input. Request message has no argument. Reply message has as many arguments as the number of sections configured in the backend:

- **TP0_sec0, TP0_sec1, ...** a sequence of floating point numbers representing the total power intensity level of each section of the backend with terminated input.

Example communication:

```
request: "?get-tp0\r\n"
  reply: "!get-tp0,ok,00.00,00.00\r\n"
```

time

Asks the backend to return its own timestamp, this command should be used to verify that the backend has an acceptable clock working before issuing time tagged acquisition commands. Request has no argument. The reply has one only argument:

- **timestamp** the timestamp of the answer message according to the backend clock

Example communication:

```
request: "?time\r\n"
  reply: "!time,ok,1430922782.97088300\r\n"
```

start [timestamp]

Tell the backend to start the acquisition. The reply has no parameter. The reqeust has one optional parameter:

- **timestamp** the exact time at which the acquisition should start

If given with a timestamp the backend should continue to accept commands while waiting for the start time. A stop command will cancel any further pending acquisition. If a new start command is issued while waiting for a start time, the most recent start command will overwrite the pending one.

Example communication:

```
request: "?start\r\n"
  reply: "!start,ok\r\n"

request: "?start,1430922782.97088300\r\n"
  reply: "!start,ok\r\n"

request: "?start,1430922782.97088300\r\n"
  reply: "!start,fail,cannot start at given time\r\n"
```

stop [timestamp]

Tell the backend to stop the acquisition. The reply has no parameter. The reqeust has one optional parameter:

- **timestamp** the exact time at which the acquisition should stop

If given with a timestamp the backend should continue to accept commands while waiting for the stop time. If a new stop command is issued while waiting for a stop time, the most recent stop command will overwrite the pending one.

Example communication:

```
request: "?stop\r\n"
  reply: "!stop,ok\r\n"

request: "?stop,1430922782.97088300\r\n"
  reply: "!stop,ok\r\n"

request: "?stop,1430922782.97088300\r\n"
  reply: "!stop,fail,cannot stop at given time\r\n"
```

Note: In general we note that the correct behaviour of time tagged commands is left as a responsibility to the backend itself and not to the protocol. It will be duty of the particoular implementation to keep track of pending start and stop timestamps during the acquisition process. For example it is possible to have both a start timestamp and a stop timestamp issued in the future, and these should work as expected.

set-section

Configure a section of the backend, just as explained in *Total Power - focus selector*. The command expects 7 parameters, all of which are mandatory, but can be substituted by an asterisk '*' meaning that the corresponding parameter should be left unchanged. The synthax is:

```
set-section=sect,startFreq,bw,feed,mode,sampleRate,bins
```

The server should respond **ok** if the operation succeds, and **fail** if the server could not perform the operation or the backend does not support this kind of operation.

Example communication:

```
request: "?set-section,1,50.0,200.0,1,CP,10,2048\r\n"
  reply: "!set-section,ok\r\n"

request: "?set-section,1,*,*,*,*,*,*\r\n"
  reply: "!set-section,ok\r\n"

request: "?set-section,1,*\r\n"
  reply: "!set-section,fail,set-section needs 7 arguments\r\n"

request: "?set-section,1,badparam,200.0,1,CP,10,2048\r\n"
  reply: "!set-section,fail,wrong parameter format\r\n"
```

cal-on [interleave=0]

Activate the noise diode for calibration purposes using a cal switching acquisition mode. The command takes in input one mandatory parameter:

- **interleave** the number of backend samples to be acquired with the mark on and off alternatively. If **0** is given (default) as parameter the mark is always off, this can be used to turn the fast mark off.

Example communication:

```
request: "?cal-on\r\n"
  reply: "!cal-on,ok\r\n"

request: "?cal-on,10\r\n"
  reply: "!cal-on,ok\r\n"

request: "?cal-on,-10\r\n"
  reply: "!cal-on,fail,interleave samples must be a positive int\r\n"
```

set-filename

Tell the backend where to find the telescope file with metadata to be merged with the backend data.

- **filename** absolute path of the file to be merged

Example communication:

```
request: "?set-filename,/hi/im/a/file.fits\r\n"
  reply: "!set-filename,ok\r\n"
```

convert-data

New in version 1.2.

Instructs the backend to convert data. This means that the control system has successfully closed the auxiliary metadata file which can now be completed with collected data from the backend.

Example communication:

```
request: "?convert-data\r\n"
  reply: "!convert-data,ok\r\n"
```

Handling Errors

As specified above, the protocol permits to distinguish between two kinds of errors, both of which are identified in the response messages:

- **protocol errors** are identified by the response argument **invalid**
- **application errors** are identified by the response argument **fail**

Both responses permit a second argument to specify a description of the error.

Example communication:

```
request: "?nonexistentcommand\r\n"
  reply: "!nonexistentcommand,invalid,cannot find command\r\n"

request: "?--asdf\r\n"
  reply: "!--asdf,invalid,invalid characters in command name\r\n"

request: "ciao\r\n"
  reply: "!ciao,invalid,requests must start with '?'\r\n"

request: "?start,0\r\n"
  reply: "!start,fail,invalid timestamp\r\n"

request: "?start,0\r\n"
  reply: "!start,fail,invalid timestamp\r\n"
```

Reference Implementation

You can find a reference implementation of the protocol at https://github.com/discos/discos-backend . This package implements all the logics related to the protocol, including parsing and serialization of messages, transmission, checks for correctness and error management. The package also defines a server implementation which enables a pluggable protocol to be used. The developer can just look at the tests (as described in the package docs) in order to define its own protocol implementation.

Considerations

The proposed protocol is intentionally very simple and little powerful; this choice is derived from the specs given in the meeting held the 13 / 04 / 2015 at OaC. More complex interactions would require a different protocol resulting in a more complex definition and different technologies involved. In particular the actual definition does **not** :

- Version the protocol in the protocol itself
- Permit to send asynchronous messages
- Permit biderectional requests
- Permit to send the same message to multiple recipients
- Enable any security mechanism

2.1.7 HOWTOs

DISCOS deployment

Abstract

This howto describes how to deploy DISCOS. Even though the whole process is automatic and hides the underline technologies, before starting you have to spend 15 minutes to install some dependencies: Python 2.7, Git, Ansible, Vagrant and VirtualBox. Look at the section *Dependencies* to see how to install them, then go ahead and read the *Quick start* section.

Dependencies

To automatically deploy DISCOS, you need to install some dependencies: Python 2.7, Git, Ansible, Vagrant and VirtualBox. It usually takes about 15 minutes.

Install Python 2.7

Verify if Python 2.7 is already installed on your system:

```
$ python --version
Python 2.7.15rc1
```

If your system comes bundled with a different Python version, you may want to consider installing a Python 2.7 environment such as Anaconda, Miniconda or Pyenv and its Virtualenv plugin.

Install Git

Before installing Git, verify if it is already installed:

```
$ git --version
git version 2.17.1
```

You do not need a particular version of Git, but if the command `git --version` fails, than you have to install Git by following the instructions at the official Git website.

Install Ansible

Verify if Ansible is already installed:

```
ansible --version
ansible 2.7.2
```

If the command `ansible --version` fails, than you have to install Ansible.

The suggested way to do this is via *pip*:

```
$ sudo pip install ansible
```

Note: If you are using a Python virtual environment, you do not need administration permissions to install Ansible.

In case the command fails, or if you want to install Ansible in a different way, check out the official ansible website.

Install VirtualBox and Vagrant

You need to install VirtualBox and Vagrant only if you want to deploy DISCOS on virtual machines, as in the case of a development environment. As a first step, check if VirtualBox is already installed:

```
$ which virtualbox
/usr/bin/virtualbox
```

In case it is not, download the binary file from the official website and install it.

Note: The suggested VirtualBox version to install is the 5.1. As of today, the latest release of VirtualBox, version 5.2, seems to introduce some lag in SSH sessions to the deployed virtual machines. Version 5.1 also matches the VirtualBox guest additions already installed in the virtual machines.

Now verify if Vagrant is installed:

```
$ vagrant --version
Vagrant 2.2.1
```

If it is not, download the binary file from the vagrant official website and install it.

At this point you are ready to deploy DISCOS. The *Quick start* section is a good starting point, because it covers a typical scenario.

Quick start

If you have installed the dependencies, as explained in section *Dependencies*, then you are ready to download the DISCOS deployment repository and install it:

```
$ git clone https://github.com/discos/deployment.git
Cloning into 'deployment'...
$ cd deployment
$ python setup.py
Would you like to automatically install the requirements? [Y/n]:
Installing 'pexpect' package...done.
Would you like to add the scripts to your ~/.bashrc file? [Y/n]:
The file that sources the scripts has been added to your ~/.bashrc file.
They will be available as soon as you open a new terminal. Have fun!
```

As you may have noticed from the above command output, the `setup.py` script can handle the python required packages setup procedure automatically. It can also add some useful scripts by sourcing them via your .bashrc file. You can read additional information about these scripts in the section *Useful scripts*.

Note: You can decide not to add the provided scripts to your .bashrc file, but you can still execute them directly from the `scripts` directory inside the repository directory.

Provision the system

The provisioning procedure is entirely handled by the *discos-deploy* script, that we installed in the previous step. Below you can find an explanation of how the script works, along with some command line examples that show you how to deploy the environment you want. For instance, the following command will create the *manager* virtual machine, and will install all dependencies and configuration files on it:

```
$ discos-deploy manager:development
```

In this case we gave the *discos-deploy* script the *manager:development* argument. It means we want to deploy only the *manager* machine, in the *development* environment. We will see more about the environments in sections *Development* and *Production*.

The deployment of a single machine can take between 30 minutes and one hour, depending of your machine, because it has to download and install the operating system, all the dependencies (ACS, DISCOS libraries, etc.), create users and groups, configure the network, and create a complete working environment, as we will see in a few moments.

Note: You can run the deployment more than once, because the process is idempotent, that is, the result of performing the deployment once is exactly the same as the result of performing it repeatedly without any intervening actions. In fact, if you run the deployment process once again, it will take just a few minutes.

When the deployment procedure is complete, we will have the *manager* virtual machine deployed with everything we need.

Get a DISCOS branch

Login via SSH to *manager* machine (or use the *discos-login* script described in the *Useful scripts* section):

```
$ ssh discos@<manager IP address>
(branch?) discos@manager ~ $
```

Currently we have no active branch, that is why there is the *(branch?)* text at the beginning of the prompt. To get a DISCOS branch and activate it, we have to use the *discos-get* command. In the following case we get the *master* branch for the *SRT* station:

```
(branch?) discos@manager ~ $ discos-get --branch master --station srt
Cloning into 'master-srt'...
remote: Enumerating objects: 1383, done.
remote: Counting objects: 100% (1383/1383), done.
remote: Compressing objects: 100% (238/238), done.
remote: Total 38919 (delta 1195), reused 1177 (delta 1127), pack-reused 37536
Receiving objects: 100% (38919/38919), 25.27 MiB | 3.80 MiB/s, done.
Resolving deltas: 100% (22848/22848), done.
Repository cloned into /discos-sw/discos/master-srt
Introot created into /discos-sw/introots/master-srt
(master-srt:telescope) discos@manager ~ $ ls
master-srt
```

Note that this is not a *code-block*, but a screenshot of the shell. As you can see the environment shows the prompt using a syntax highlight. The *branch?* text has been replaced by *master-srt:telescope*, because we are working on the branch *master*, using its *telescope* CDB, which is the actual production CDB, hosted into *SRT/Configuration*. The *INTROOT* has been created outside the *discos* user home directory, in order to

enable access to it to a generic *observer* user, and it shares the same name as the downloaded branch. Also, you may have noticed that the *discos-get* command accepts a *--station* argument. This argument is required on development machines, since they are generic and do not completely reflect the behavior of a production machine. In case of production machines, the *--station* argument is disabled since the station is already specified by the system itself. This was made in order to avoid using a different station configuration database. Here is the current environment:

```
(master-srt:telescope) discos@manager ~ $ echo $INTROOT
/discos-sw/introots/master-srt
(master-srt:telescope) discos@manager ~ $ echo $ACS_CDB
/discos-sw/discos/master-srt/SRT/Configuration
(master-srt:telescope) discos@manager ~ $ echo $STATION
SRT
```

Change the CDB

The *discos-set* command allows us to specifying the CDB. We can choose either *--cdb telescope* or *--cdb test*:

```
(master-srt:telescope) discos@manager ~ $ discos-set master-srt --cdb test
(master-srt:test) discos@manager ~ $ echo $ACS_CDB
/discos-sw/discos/master-srt/SRT
```

Change the active branch

Sometime we want to have more than one branch and switch between them. For instance, let's get the *stable* branch for *Medicina* station:

```
(master-srt:telescope) discos@manager ~ $ discos-get --branch stable --station medicina
Cloning into 'stable-medicina'...
remote: Enumerating objects: 1286, done.
remote: Counting objects: 100% (1286/1286), done.
remote: Compressing objects: 100% (159/159), done.
remote: Total 38822 (delta 1172), reused 1139 (delta 1121), pack-reused 37536
Receiving objects: 100% (38822/38822), 25.22 MiB | 2.90 MiB/s, done.
Resolving deltas: 100% (22825/22825), done.
Repository cloned into /discos-sw/discos/stable-medicina
Introot created into /discos-sw/introots/stable-medicina
(stable-medicina:telescope) discos@manager ~ $ echo $INTROOT
/discos-sw/introots/stable-medicina
(stable-medicina:telescope) discos@manager ~ $ echo $ACS_CDB
/discos-sw/discos/stable-medicina/Medicina/Configuration
(stable-medicina:telescope) discos@manager ~ $ echo $STATION
Medicina
(stable-medicina:telescope) discos@manager ~ $ ls
master-srt  stable-medicina
```

Now we have two branches, *master-srt* and *stable-medicina*, and the latter is the active one. We can switch back to the *master* branch using the *discos-set* command:

```
(stable-medicina:telescope) discos@manager ~ $ discos-set master-srt
(master-srt:telescope) discos@manager ~ $ echo $INTROOT
/discos-sw/introots/master-srt
(master-srt:telescope) discos@manager ~ $ echo $ACS_CDB
/discos-sw/discos/master-srt/SRT/Configuration
(master-srt:telescope) discos@manager ~ $ echo $STATION
SRT
```

Remove a branch

To remove a branch, just remove the directory:

```
(master-srt:telescope) discos@manager ~ $ ls
master-srt  stable-medicina
(master-srt:telescope) discos@manager ~ $ rm -rf stable-medicina/
(master-srt:telescope) discos@manager ~ $ rm -rf master-srt/
(branch?) discos@manager ~ $
```

The related `INTROOT` will be deleted automatically right after the branch directory is removed.

Install DISCOS

To install the DISCOS control system, execute the commands shown in the image below:

```
(master-srt:telescope) discos@manager ~ $ cd master-srt/
(master-srt:telescope) discos@manager ~/master-srt $ ls
CHANGELOG.md  CONTRIBUTING.md  Noto       SRT          SystemMake
Common        Medicina         README.md  SystemBuild  UML-Project
(master-srt:telescope) discos@manager ~/master-srt $ cd SystemMake/
(master-srt:telescope) discos@manager ~/master-srt/SystemMake $ make
(master-srt:telescope) discos@manager ~/master-srt/SystemMake $ make install
```

Note: The output of the `make` and `make install` commands shown above was suppressed. The whole building and installation process can take up to one hour or more, depending on your system.

What we have seen so far is enough for deploying a small development environment, but there is more to know. Maybe you want to deploy the whole system, composed of several VMs, or maybe you want to deploy the system on production machines. For more information about these topics have a look at the *Development* and *Production* sections. You will realize that everything is as easy as we saw here, and for deploying in production is even easier.

Development

Machines deployment

In the *Quick start* section we have seen how to deploy a `manager` virtual machine. You can also deploy the whole control system. For instance, to deploy a development system composed of three virtual machines (`manager`, `console` and `storage`), pass the `discos` *cluster*, followed by the `development` system, to the `discos-deploy` script:

```
$ discos-deploy discos:development
```

This command will connect via SSH to all development machines and provision the whole system (create users, configure networking, install yum packages, configure NIS, Lustre, ACS and its dependencies, install some utilities, and finally the DISCOS dependencies).

Working without Vagrant

If you wish to deploy a development environment on physical machines, or you simply want to install the virtual machines by yourself using VirtualBox or another virtual machine provider, you can still use the *discos-deploy* script to perform the automatic deployment procedure. You first have to setup the machines' network interfaces, in order to enable Ansible access to them via the correct IP address, and then start the deployment procedure by passing the *--no-vagrant*, *-n* flag to the *discos-deploy* script:

```
$ discos-deploy discos:development --no-vagrant
```

This will let the script know that you do not want to create any (other) virtual machine on your system, and that you want to apply the whole configuration onto an existing, physical or virtual, set of machines.

DISCOS setup

Manual setup

To download the DISCOS control software, you can use the *discos-get* command, covered in the *Get a DISCOS branch* section.

```
(branch?) discos@manager ~ $ discos-get --branch master --station srt
Cloning into 'master-srt'...
remote: Enumerating objects: 1383, done.
remote: Counting objects: 100% (1383/1383), done.
remote: Compressing objects: 100% (238/238), done.
remote: Total 38919 (delta 1195), reused 1177 (delta 1127), pack-reused 37536
Receiving objects: 100% (38919/38919), 25.27 MiB | 3.80 MiB/s, done.
Resolving deltas: 100% (22848/22848), done.
Repository cloned into /discos-sw/discos/master-srt
Introot created into /discos-sw/introots/master-srt
(master-srt:telescope) discos@manager ~ $ ls
master-srt
```

Once the download of the repository is completed, you have to perform the setup procedure like we have seen in the *Install DISCOS* section.

```
(master-srt:telescope) discos@manager ~ $ cd master-srt/
(master-srt:telescope) discos@manager ~/master-srt $ ls
CHANGELOG.md  CONTRIBUTING.md  Noto       SRT          SystemMake
Common        Medicina         README.md  SystemBuild  UML-Project
(master-srt:telescope) discos@manager ~/master-srt $ cd SystemMake/
(master-srt:telescope) discos@manager ~/master-srt/SystemMake $ make
(master-srt:telescope) discos@manager ~/master-srt/SystemMake $ make install
```

Automatic setup

You can let the deployment procedure perform the DISCOS control system setup for you by specifying the `--branch` argument to the `discos-deploy` script, followed by the branch name you want to deploy and install. Since the procedure will internally call the `discos-get` script for you, if you are deploying in a development environment, you also have to provide to the script the `--station` argument, followed by the name of the desired station.

```
$ discos-deploy discos:development --tag discos1.0-rc02 --station noto
```

You can choose a station among `medicina`, `noto` and `srt`. The `discos-get` command gets executed on all machines of the system at the same time.

Production

Machines deployment

To deploy the system in production, you have to specify a *cluster* of machines, followed by the name of the station, you can choose among `medicina`, `noto` or `srt`:

```
$ discos-deploy discos:srt
```

As mentioned in the *Development* section, the `discos` *cluster* makes the automatic procedure provision the whole DISCOS system, composed by all the required station machines. If you only want to deploy a single machine, change the cluster from `discos` to the related machine name. For instance, in the following case we are deploying only the `console` machine for the `srt` station:

```
$ discos-deploy console:srt
```

DISCOS setup

Manual setup

To install the DISCOS control software, you can use the `discos-get` command and then build and install the system by yourself, as we already saw in the *Get a DISCOS branch* paragraph. Since this time we are deploying in a production environment, you may want to deploy a DISCOS tag. The `discos-get` script can handle this case just like it does for normal branches. All you have to do is pass to the scripts command line the `--tag` argument instead of the `--branch` one. Of course, you also have to specify the desired DISCOS tag afterwards:

The downloaded tag will be handled by `discos-get` and `discos-set` just like a normal branch.

Note: As you may have noticed from the last image, the downloaded repository will be left in a `detached HEAD' state`. This means that any modification you make to the repository will not be tracked by any remote branch. If you want to edit some files in order to push a hotfix you should download and work on the `stable` branch.

You can now build and install the DISCOS control system as we already saw in the *Install DISCOS* paragraph.

Automatic setup

The `discos-deploy` script can automatically handle the DISCOS setup procedure even for tags. In order for it to do this, you have to pass the `--tag` argument to the `discos-deploy` command, followed by the DISCOS tag you

```
(branch?) discos@manager ~ $ discos-get --tag discos1.0-rc02
Cloning into 'discos1.0-rc02'...
remote: Enumerating objects: 1351, done.
remote: Counting objects: 100% (1351/1351), done.
remote: Compressing objects: 100% (141/141), done.
remote: Total 38761 (delta 1262), reused 1211 (delta 1210), pack-reused 37410
Receiving objects: 100% (38761/38761), 25.20 MiB | 3.97 MiB/s, done.
Resolving deltas: 100% (22802/22802), done.
Note: checking out '97a81c4b5d463f1e9ccc13600bbb01456f342af9'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

Repository cloned into /discos-sw/discos/discos1.0-rc02
Introot created into /discos-sw/introots/discos1.0-rc02
```

want to install on the machines:

```
$ discos-deploy discos:srt --tag discos1.0-rc02
```

Note: Since you are performing the deployment procedure on station machines, the station name is already specified inside the machines themselves as an environment variable, so you can omit the `--station` argument from both the `discos-deploy` and `discos-get` scripts. If you pass the `--station` argument anyway, if the given argument does not match the correct station you will receive an error and the procedure will stop.

Useful scripts

If you executed the `setup.py` script as mentioned in the *Quick start* section, you will have access via terminal to the following useful scripts, along with the `discos_deploy` one (described in the *Quick start* section):

- `discos-vms`: a script used to manage DISCOS virtual machines
- `discos-login`: a script that performs SSH login to the specified virtual machine
- `discos-vnc`: a script that opens a VNC viewer to the specified virtual machine

Start and stop DISCOS virtual machines, *discos-vms*

To start or stop a DISCOS virtual machine, you can run the `discos-vms` script specifying an action and a machine. The action is one among *–start*, *–stop*, *–status* and *–restart*, the machine argument is the name of an already deployed virtual machine. For instance, if you want to start the `manager` virtual machine, all you have to do is execute the following command:

```
$ discos-vms --start manager
Starting machine manager..............done.
```

As you may have noticed from the previous examples, the *discos-vms --start <machine>* command will block and wait until the machine is booted up and ready. Executing this command when the selected machine is already powered on will just print on screen:

```
$ discos-vms --start manager
Machine manager is already running.
```

The script can also handle multiple machines:

```
$ discos-vms --start manager console
Starting machine manager..............done.
Starting machine console..............done.
```

If you omit the machine, the script will try to execute the same action on all available machines:

```
$ discos-vms --start
Starting machine storage..............done.
Starting machine manager..............done.
Starting machine console..............done.
```

Note: Right after the deployment procedure is completed, the deployed virtual machines will be running already, so starting them with the above command will just output *Machine <machine> is already running*.

If you want to stop a running machine you just need to specify the *--stop* action:

```
$ discos-vms --stop manager
Powering off machine manager......done.
```

Just like the *--start* command, the *--stop* command will block and wait until the selected machine has been completely powered off. Trying to stop a powered off machine will print a message just like the *--start* command:

```
$ discos-vms --stop manager
Machine manager is not running.
```

If you use the *--restart* action, machines will of course be restarted.

In case you specify the *--status* action, the script will display the virtual machine status:

```
$ discos-vms --status manager console
Machine manager status: running.
Machine console status: powered off.
```

Login into a DISCOS virtual machine, *discos-login*

The script *discos-login* acts as a wrapper to *ssh*, and is useful to easily perform login on a deployed virtual machine. To login into a DISCOS virtual machine with this script, you can simply execute the following code:

```
$ discos-login manager
discos@manager's password:
(branch?) discos@manager ~ $
```

The *discos-login* command handles the login procedure by internally executing the following command:

```
$ ssh -CX discos@manager
```

You can specify the user with which you want to login to the virtual machine, by appending the `-u`, or `--user`, argument, followed by the desired user name, to the *discos-login* script, just as follows:

```
$ discos-login -u observer console
observer@console's password:
(branch?) observer@console ~ $
```

Note: Currently the `discos-login` command only handles logins to virtual machines. It does not rely on host names present in the `/etc/hosts` file, it reads host names and their IP addresses from the Ansible inventory directory. Changing any development machine's IP address in the Ansible inventory after the deployment procedure is completed could result in a login failure using this script. This behavior could change in the future in order to enable the login to any machine (even actual ones).

Graphical login into a DISCOS virtual machine (using VNC), *discos-vnc*

The script `discos-vnc` acts as a wrapper to `vncviewer`, and is useful to perform a graphical login on a deployed virtual machine.

In order to be able to use it you should install `vncviewer`, we suggest the `tigervnc` one, that can be installed via `yum` or `apt`. On red-hat based linux distributions you can install it by typing:

```
$ sudo yum install tigervnc
```

Whether on debian-based linux distributions you can install it by typing:

```
$ sudo apt install tigervnc-viewer
```

If you fail to install the `vncviewer` using the previous commands, or if you are running a different operating system than the previously mentioned ones, check out the official tigervnc website.

Once you installed the `vncviewer`, you can correctly execute the `discos-vnc` command. In order to login into a DISCOS virtual machine by the means of it, you can simply execute the following code and insert type desired user's login password:

```
$ discos-vnc manager
discos@manager's password:
```

At this point the VNC session will be opened on your display. The `discos-vnc` command handles the graphical login procedure by establishing a ssh tunnel to the desired machine and launching the `vncviewer` in order to display the specified user's desktop. Only the `manager` and `console` machines hosts some VNC servers. The `manager` machine hosts the VNC server for the `discos` user, the `console` machine hosts both the VNC servers for the `discos` user and the `observer` user.

Note: Like the *discos-login* script, even *discos-vnc* relies on IP addresses read from the Ansible inventory directory. This behavior could change in the future in order to enable the graphical login to any machine (even actual ones).

Test Driven Development (TDD) in DISCOS

Abstract

Here the abstract…

Introduction

DISCOS TDD cycle

From functionals to unit (summary of *Write the component*).

Where to put the tests

Tests must be defined in the *tests* directory within the module you are writing:

```
$ tree tests/

tests
|-- Makefile
|-- .discos
|-- external
|   `-- __init__.py
|-- functional
|   `-- __init__.py
|-- procedure.cfg
|-- pyunit
|   `-- __init__.py
|-- results
`-- unittest.cpp
```

This is the default structure for tests definitions but more complex structures are also possible, depending on your needs. Unit tests are defined in *tests/unittest.cpp* while the *tests/functional* directory and its sub-directories contain the *functional tests* and, as you probably figured out, the *api* sub-directory hosts the API while the *commands* one hosts the command tests.

Note: Do not worry if you do not know this terminology, in the next sections we will give a definition and an explanation of every of these words.

You can create that directory using the `getTemplateForTest` command. For instance, if you want to create that directory for the *DewarPositioner* component:

```
$ cd $NURAGHE_ROOT/Common/Servers/DewarPositionier
$ ls
doc  srt
$ getTemplateForTest
$ ls
doc srt tests
```

Functional testing

A test is called *functional* when it aims to verify the component behavior from the user point of view. The components usually have two kind of users:

1. **operators** (human beings) who interact with the component through the *operator input* console
2. **programs** (clients or other components) that interact with the component by using its API (calling its methods)

In the first case, we should verify the behavior of the commands the operator sends to our component. In other words, we we have to test the component `command()` method. We call this kind of functional tests by the name of *command tests*.

In the second case, we should verify all the other methods defined in the component IDL interface. We should prove they give us the expected results, and when required they raise the expected exceptions. We give this functional tests the name of *API tests*.

Writing functional tests in Python is really straightforward, and of course it is easier than C++, so we will use it in this section. The Python standard library provides a unit test framework, called unittest. Unfortunatly the Python version we can use with ACS-8.2 is the old Python 2.5, and in its unittest module there is a lack of useful assertions. The solution is to install a third part module, unittest2, in order to use it in place of the standard library unittest module:

```
$ easy_install unittest2 # Install the unittest2 library
```

Let's have a real example: the *DewarPositioner* component. We will see both how to write a command and an API test.

Command tests

In this section we will test the `derotatorSetup` command. As we said before, we have to think from the user (the astronomer in that case) point of view. So, let's see what the user wants to obtain after executing the command from the *operatorInput* console:

```
> derotatorSetup=KKG
>
```

Well, we are ready to write our functional test. The file and the methods must start with *test*, so we can create a file called *test_derotatorSetup.py* inside the *functional/commands* directory:

```
# File: test/functional/commands/test_derotatorSetup.py
import unittest2
from Acspy.Clients.SimpleClient import PySimpleClient
from DewarPositioner.DewarPositionerImpl import DewarPositionerImpl


class DerotatorSetupCmdTest(unittest2.TestCase):
    """Test the derotatorSetup and derotatorGetActualSetup commands"""

    def test_proper_setup(self):
        # Get the component
        client = PySimpleClient()
        dp = client.getComponent('RECEIVERS/DewarPositioner')

        # Send the component a command
        success, answer = dp.command('derotatorSetup=KKG')
        # Verify the command is executed as expected
        self.assertEqual(answer, '')
```

(continues on next page)

(continued from previous page)

```
        # Release the component and disconnect the client
        client.releaseComponent('RECEIVERS/DewarPositioner')
        client.disconnect()

if __name__ == '__main__':
    unittest2.main()
```

Before implementing the code, we write just the component interface without writing the command execution method. We do not write the method implementation because we want to get an expected failure. In that case, we expect the test fails saying the command does not exist.

Important: One of the most important things about testing is to be sure the tests can fail. It is better do not have a test than to have one that does not fails when it should do.

Let's start the test:

```
$ python test_derotatorSetup.py

======================================================================
FAIL: test_proper_setup (__main__.DerotatorSetupCmdTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_derotatorSetup.py", line 19, in test_proper_setup
    self.assertEqual(answer, '')
AssertionError: 'Error - command derotatorSetup does not exist' != ''

----------------------------------------------------------------------
Ran 1 test in 1.388s

FAILED (failures=1)
```

As we can see from the output message, only one test has been run, and it failed as expected, with a straighforward error message. Now we are ready to write the *derotatorSetup* command code. Let's write it, and than we run again the test, of course offline. To spin up the test offline we need to simulate the external resources. The *DewarPositioner* gets a reference to a derotator component, and this one communicates to the hardware. The best approach is to simulate the external resource API, but in our case, because the derotator protocol is a bit complex, we choose to implement a simulator of the ACS derotator component. To get this component ready, we just have to point to the testing CDB:

```
$ export ACS_CDB=~/Nuraghe/ACS/trunk/SRT/
```

We can now start ACS and the required containers:

```
$ acsStartContainer -py DerotatorsContainer
$ acsStartContainer -py DewarPositionerContainer
```

Let's spin up the test:

```
$ python test_derotatorSetup.py
.
----------------------------------------------------------------------
Ran 1 test in 2.825s

OK
```

If we write a wrong setup code we want the component to behave this way:

```
> derotatorSetup=GIGIRIVA
Error - setup GIGIRIVA not available"
```

So, we write an additional test that verifies this case:

```
# File: test/functional/commands/test_derotatorSetup.py
import unittest2
from Acspy.Clients.SimpleClient import PySimpleClient
from DewarPositioner.DewarPositionerImpl import DewarPositionerImpl


class DerotatorSetupCmdTest(unittest2.TestCase):
    """Test the derotatorSetup and derotatorGetActualSetup commands"""

    def test_proper_setup(self):
        # Get the component
        client = PySimpleClient()
        dp = client.getComponent('RECEIVERS/DewarPositioner')

        # Send the component a command
        success, answer = dp.command('derotatorSetup=KKG')
        # Verify the command is executed as expected
        self.assertEqual(answer, '')

        # Release the component and disconnect the client
        client.releaseComponent('RECEIVERS/DewarPositioner')
        client.disconnect()

    def test_wrong_setup(self):
        # Get the component
        client = PySimpleClient()
        dp = client.getComponent('RECEIVERS/DewarPositioner')

        # Send the component a command
        success, answer = dp.command('derotatorSetup=GIGIRIVA')
        # Verify the answer starts with 'Error'
        self.assertTrue(answer.startswith('Error'))

        # Release the component and disconnect the client
        client.releaseComponent('RECEIVERS/DewarPositioner')
        client.disconnect()

if __name__ == '__main__':
    unittest2.main()
```

As we can see, we added a new test, called `test_wrong_setup()`. In that test we command a `derotatorSetup` with a wrong error code, and we assert that the answer starts with `'Error'`. We also notice that our code smells, because there is a lot of duplication among the two tests. The `unittest` framework provides special methods, called `setUp()` and `tearDown()`, that it calls respectively before and after each test. So, we can refactor our test case moving the common code inside this two methods:

```
# File: test/functional/commands/test_derotatorSetup.py
import unittest2
from Acspy.Clients.SimpleClient import PySimpleClient
from DewarPositioner.DewarPositionerImpl import DewarPositionerImpl
```

(continues on next page)

(continued from previous page)

```
class DerotatorSetupCmdTest(unittest2.TestCase):
    """Test the derotatorSetup and derotatorGetActualSetup commands"""

    def setUp(self):
        self.client = PySimpleClient()
        self.dp = self.client.getComponent('RECEIVERS/DewarPositioner')

    def tearDown(self):
        self.client.releaseComponent('RECEIVERS/DewarPositioner')
        self.client.disconnect()

    def test_proper_setup(self):
        success, answer = self.dp.command('derotatorSetup=KKG')
        self.assertTrue(success)
        self.assertEqual(answer, '')

    def test_wrong_setup(self):
        success, answer = self.dp.command('derotatorSetup=GIGIRIVA')
        self.assertFalse(success)
        self.assertTrue(answer.startswith('Error'))

if __name__ == '__main__':
    unittest2.main()
```

API tests

In this section we will see how to write an API test. To follow the previous example, we will test the *DewarPositioner.setup()* method. Let's start peeking at the IDL interface:

```
/* Take a configuration code and configure the component
 *
 * This method takes a configuration code, gets the corresponding
 * derotator component reference and initializes the DewarPositioner.
 * For instance, by giving the code KKG, the DewarPositioner gets the
 * KBandDerotator reference and performs its setup. It also sets the
 * rewinding mode and configuration default values as:
 *
 *     setConfiguration('FIXED')
 *     setRewindingMode('AUTO')
 *
 * @param code the setup mode (for instance: LLP, KKG, CCB, ecc.)
 * @throw ComponentErrors::ComponentErrorsEx in case of wrong
 * configuration code or derotator component not available
 */
void setup(in string code) raises (ComponentErrors::ComponentErrorsEx);
```

We should write a test similar to the previous one. In particular, we want the the *setup()* to raise a *ComponentErrorsEx* in case of wrong code. Our test could be the following one:

```
# File: test/functional/test_setup.py
from __future__ import with_statement
import unittest2
import time
```

(continues on next page)

(continued from previous page)

```
from ComponentErrors import ComponentErrorsEx
from Acspy.Clients.SimpleClient import PySimpleClient


class SetupTest(unittest2.TestCase):
    """Test the DewarPositioner.setup() method"""

    def setUp(self):
        self.client = PySimpleClient()
        self.dp = self.client.getComponent('RECEIVERS/DewarPositioner')

    def tearDown(self):
        self.client.releaseComponent('RECEIVERS/DewarPositioner')
        self.client.disconnect()

    def test_proper_setup(self):
        self.dp.setup('KKG')
        self.assertEqual(self.dp.getActualSetup(), 'KKG')

    def test_wrong_setup(self):
        with self.assertRaises(ComponentErrorsEx):
            self.dp.setup('GIGIRIVA')


if __name__ == '__main__':
    unittest2.main()
```

Note: The import *from __future__ import with_statement* is required in Python 2.5 when, as in this case, we use the *with* statement.

Unit testing

Unit tests are just what the name suggests: portion of code which ensure that each code unit is performing as expected. These are especially useful for documenting the interface of a class by providing real usage examples, and whenever someone makes a change to a code unit in order to verify that the change did not break the behavior of the unit itself.

Principles

Contrary to what may seem obvious, unit testing involves much more the design of your code rather then the testing of already written code. That's why we talk about *Test Driven Development*, because if you develop with testing in your mind, and also writing tests first, your code will result better conceived, better designed, more maintanable and easily documented and usable by other memebers of your team.

Unit testing and TDD enforce the adoption of SOLID principles when writing your code, and this is expecially true for C++ code: if your code module is well concieved in terms of object oriented design, it should contain a lot of little classes each of which encapsulates a well determined behavior and is responsible for a well defined set of actions. That's to say that if you adhere to the single responsibility principle your code will naturally be divided into code units. Also, dependency management between code modules will be better managed if you adhere to dipendency inversion principles. We will see how both of these requirements will be necessary for writing effective tests and will lead to better code design and implementation.

Note: Unit tests can be effectively developed also against already written code to ensure its stability and safe refactoring. Even if it is not proper TDD it will provide good value to your code.

DISCOS Integration

Within **DISCOS** we have developed some automation which eases the development and the execution of unit tests, and this can be applied to every module you write.

We will demonstrate how to write unit tests and how to integrate the tests development into your workflow by developing a simple library, the same approach can be used for every code unit defined within discos.

The library we will develop is a unit of code we can use to connect to an external backend according to the protocol specified in *Backend protocol*.

GOOGLE Tests

Within **DISCOS** we are using the third party libraries **google test** and **google mock** to ease the writing and execution of tests. Documentation for these libraries can be found online at:

- Google test introduction is a must read for everyone using this library for the first time and contains very introductory material. You can keep this document at your disposal while writing your first tests or reading this guide.
- Google test advanced guide contains more advanced documentation about more complex assert statements, exceptions ecc…
- Google mock for dummies is the introductory material for the google mock library. Mocking is a slightly advanced topic in developing unit tests, expecially when it comes to C++ implementations. you should read this guide if you are developing tests which contain dependencies from external objects.

Install

Google mock and google test frameworks are automatically installed by the provisioning scripts defined in azdora projects. If you are running on a different machine you can still use azdora gmock script as a reference for installation. It is important that you do not change the paths where libraries are installed if you want to rely on all the automations defined within the **DISCOS** framework.

C++ implementation

We start by writing our first test in our **tests** directory generated with **getTemplateForTest** script. We thus edit *tests/unittest.cpp* to define a simple behavior for our **Request** class: this will encapsulate a request message we'd like to send to our backend server:

```
#include "gtest/gtest.h"

#include "grammar.h"

using namespace backend;

TEST(MessageTest, MessageRequestConstruction){
```

(continues on next page)

(continued from previous page)

```
8       Request request("myrequest");
9       EXPECT_EQ("?myrequest", request.toString());
10  }
```

Note how we already define how we'd like the class to behave.

- Line 1 includes the necessary google tests headers.
- Line 3 This file is still not defined but it will contain the necessary definitions
- Line 7 defines a new unit test using the **TEST** macro defined in google library. The two parameters define a test-case-name and a test-name which will be used in the final report to identify and group unit test results.
- Line 8 defines the behaior of our Request class
- Line 9 is the actual test we are performing. We are using the **EXPECT_EQ** macro to check that our class is behaving as expected.

Note: We are leaving out some boilerplate code from this online guide for better readability. You can download *the complete archive* containing the full working code.

Next step will be to define the code which will make this test pass. So we define a simple *ExternalBackendLibrary/include/grammar.h*:

```
1   #include <string>
2   #include <vector>
3
4   #include <boost/algorithm/string.hpp>
5
6   #define BACKEND_REQUEST '?'
7   #define BACKEND_REPLY '!'
8   #define BACKEND_REPLY_OK "ok"
9   #define BACKEND_REPLY_INVALID "invalid"
10  #define BACKEND_REPLY_FAIL "fail"
11  #define BACKEND_SEPARATOR ","
12
13  using namespace std;
14
15  namespace backend{
16
17  class Message
18  {
19      public:
20          Message(const char message_type,
21                  const char* name,
22                  vector<string> arguments = vector<string>()) :
23              m_type(message_type),
24              m_name(name),
25              m_arguments(arguments){};
26          virtual ~Message();
27          virtual string toString() = 0;
28      protected:
29          const char m_type;
30          string m_name;
31          vector<string> m_arguments;
32  }; //class Message
33
```

(continues on next page)

(continued from previous page)

```
class Request : public Message
{
    public:
        Request(const char* name,
                vector<string> arguments = vector<string>()) :
                Message(BACKEND_REQUEST,
                        name,
                        arguments){};
        virtual string toString();
}; //class Request

}; //namespace backend
```

And the corresponding *ExternalBackendLibrary/src/grammar.cpp* implementation:

```
#include "grammar.h"

using namespace backend;

string
Request::toString()
{
    ostringstream output;
    output << BACKEND_REQUEST << m_name;
    return output.str();
}
```

The Makefile for this library will be composed of:

```
INCLUDES        = grammar.h
LIBRARIES       = ExternalBackend
ExternalBackend_OBJECTS   = grammar
ExternalBackend_LDFLAGS   = -lstdc++
```

We have to compile and install the library in order to run our test. Once compiled we need to adjust the test Makefile in *ExternalBackendLibrary/tests/Makefile* as:

```
1 EXECUTABLES_L = unittest
2 unittest_OBJECTS = unittest
3 unittest_LIBS = $(GTEST_LIBS) ExternalBackend
4 unittest_LDFLAGS = -lstdc++ -lpthread
```

Note that in line 3 we are adding the link to the library we want to test, this could also be a component library or any other unit of code.

Running the tests

Now we can try to compile our test:

```
developer@15:34:02:ExternalBackendLibrary $ cd tests
developer@15:34:05:tests $ make
== Creating Missing directories
/alma/ACS-8.2/ACSSW/include/acsMakefile.all:382: ../object/unittest.d: No such
file or directory
/alma/ACS-8.2/ACSSW/include/acsMakefile.all:388: ../object/unittest.dx: No such
```

(continues on next page)

(continued from previous page)

```
file or directory
== Dependencies: ../object/unittest.dx
== Dependencies: ../object/unittest.d


== C++ Compiling: unittest.cpp
== Building executable: ../bin/unittest


 . . . 'all' done
```

And run it with **make unit**:

```
developer@15:34:12:tests $ make unit
== Creating Missing directories


 . . . 'all' done
 running cpp unit tests
 ../bin/unittest --gtest_output=xml:results/cppunittest.xml
 Running main() from gtest_main.cc
 [==========] Running 1 test from 1 test case.
 [----------] Global test environment set-up.
 [----------] 1 test from MessageTest
 [ RUN      ] MessageTest.MessageRequestConstruction
 [       OK ] MessageTest.MessageRequestConstruction (0 ms)
 [----------] 1 test from MessageTest (2 ms total)

 [----------] Global test environment tear-down
 [==========] 1 test from 1 test case ran. (6 ms total)
 [  PASSED  ] 1 test.
  . . . 'unit' done
```

From the output it should be clear that the *MessageTest.MessageRequestConstruction* unit test has been executed successfully.

Test Failures

What if our implementation does not conform to the expected unit of code defined in our test? Well, obviously the test will fail. Suppose for example that the string representation of our request should contain a # character instead of an ?, the result of our test will look like:

```
 [----------] Global test environment set-up.
 [----------] 1 test from MessageTest
 [ RUN      ] MessageTest.MessageRequestConstruction
 unittest.cpp:24: Failure
 Value of: request.toString()
   Actual: "?myrequest"
 Expected: "#myrequest"
 [  FAILED  ] MessageTest.MessageRequestConstruction (1 ms)
 [----------] 1 test from MessageTest (1 ms total)

 [----------] Global test environment tear-down
 [==========] 1 test from 1 test case ran. (6 ms total)
 [  FAILED  ] 1 test, listed below:
```

(continues on next page)

(continued from previous page)

```
 [  FAILED  ] MessageTest.MessageRequestConstruction

  1 FAILED TEST
```

You can see how the test is executed and the result clearly explains what did not work and also why by providing insights about what was expected and what has been found at runtime.

TDD is essentially this, writing a test and produce the necessary code until the test passes stepping through successive failures.

Testing Exceptions

Now we want to add to our library a function which will parse a line of text and turn it into a reply message. We can proceed like in the previous request case by defining first the expected behavior in a unit test:

```
TEST(MessageTest, ParseGoodReply){
    string message("!prova,ok,1,2,3");
    Reply msg = parseReply(message.c_str());
    EXPECT_EQ(msg.toString(), message);
}
```

And we define the necessary Reply class in our library module header:

```
class Reply : public Message
{
    public:
        Reply(const char* name,
              const char* code,
              vector<string> arguments = vector<string>()) :
              Message(BACKEND_REPLY,
                      name,
                      arguments),
              m_code(code){};
        virtual string toString();
    private:
        string m_code;
}; //class Reply

Reply parseReply(const char*);
```

And in the grammar.cpp implementation:

```
 Reply
 backend::parseReply(const char* msg)
 {
     string msg_string(msg);
     // first character must be BACKEND_REPLY
     if(!(msg_string[0] == BACKEND_REPLY))
         throw GrammarError("not a valid reply");
     // type + name + separator + reply_code
     if(!(msg_string.length() >= 4))
         throw GrammarError("reply must contain at least 4 characters");
     vector<string> split_msg;
     boost::split(split_msg, msg_string, boost::is_any_of(BACKEND_SEPARATOR));
     if(split_msg.size() < 2)
```

(continues on next page)

(continued from previous page)

```
14          throw GrammarError("reply must contain at least a name and a reply code");
15      string msg_name = split_msg[0].substr(1, string::npos);
16      string msg_code = split_msg[1];
17      if((!(msg_code == BACKEND_REPLY_OK)) &&
18         (!(msg_code == BACKEND_REPLY_FAIL)) &&
19         (!(msg_code == BACKEND_REPLY_INVALID)))
20          throw GrammarError("not a valid reply code");
21      vector<string> msg_arguments(split_msg.begin() + 2, split_msg.end());
22      return Reply(msg_name.c_str(), msg_code.c_str(), msg_arguments);
23  }
```

As you can see in lines 7, 10, 14 and 20, the parsing function raises an exception whenever the reply string does not conform to the protocol. We want to add a check to our test to ensure that the exception does not get risen if the string is correct, and obviously we want to make sure that the exception is risen when appropriate. Our test will become:

```
TEST(MessageTest, ParseGoodReply){
    string message("!prova,ok,1,2,3");
    Reply msg;
    ASSERT_NO_THROW({
        msg = parseReply(message.c_str());
    });
    EXPECT_EQ(msg.toString(), message);
}

TEST(MessageTest, ParseBadReply){
    string bad_type_reply("#prova,ok,1,2,3");
    string bad_code_reply("!prova,badcode,1,2,3");
    EXPECT_THROW(parseReply(bad_type_reply.c_str()), GrammarError);
    EXPECT_THROW(parseReply(bad_code_reply.c_str()), GrammarError);
}
```

We introduced the **ASSERT_NO_THROW** and **EXPECT_THROW** macros, defined in google test framework. Every macro in the framework appears with both the *ASSERT_* and the *EXPECT_* prefixes, if the assert fails the unit test is interrupted while if the expect fails the unit test is not interrupted and successive instructions within the unittest are also executed.

Test Fixtures

As you can see we are using some variables within our unit tests. Whenever there's some code that can be shared between different unit tests we can incapsulate those definitions in a reusable class that we call a test fixture. Now each test case exploiting the test fixture will have access to the members of that class and will execute some default methods upon initialization and destrucion of the unit test. In our example we want to share the message strings between all our message tests so that we will have those defined in one common, convenient, place. Our test will become:

```
1  class Messages : public ::testing::Test {
2      public:
3          static const char *good_request, *bad_request, *good_reply_ok,
4                            *good_reply_fail, *good_reply_invalid, *bad_reply_type,
5                            *bad_reply_code; //*
6  };
7
8  const char * Messages::good_request = "?prova,1,2,3";
9  const char * Messages::bad_request = "#prova,1,2,3";
```

(continues on next page)

(continued from previous page)

```
10  const char * Messages::good_reply_ok = "!prova,ok,1,2,3";
11  const char * Messages::good_reply_fail = "!prova,fail,1,2,3";
12  const char * Messages::good_reply_invalid = "!prova,invalid,1,2,3";
13  const char * Messages::bad_reply_type = "#prova,invalid,1,2,3";
14  const char * Messages::bad_reply_code = "!prova,badcode,1,2,3";
15
16  TEST_F(Messages, ParseBadReply){
17      EXPECT_THROW(parseReply(bad_reply_type), GrammarError);
18      EXPECT_THROW(parseReply(bad_reply_code), GrammarError);
19  }
```

- In line 1 we defined a new class inheriting from **::testing::Test** , this is the base class for our unit tests.
- In line 16 we use the **TEST_F** macro which tells the framework that this unit test has fixtures and must inherit from the type specified as the first argument of the macro.
- Within the unit test with fixtures now we have access to all the variables and methods defined in the fixture class.

Test Fixtures can be much more complex than this simple example, in particular they can define special methods such as **setUp** and **tearDown** which get executed for each unit test. We will see how to use those by procceding in our library example.

Mocking

Python implementation

How to integrate a device in DISCOS

The goal of this document is to explain the steps you have to follow if you want to integrate a device (a backend, a receiver, etc.) in DISCOS. It is really easy, just follow this two steps:

- contact the *project manager* in order to plan the component development
- from your side, implement the *protocol* related to your device

That's all.

Github HOWTO

Abstract

This section is a quick newbie guide that shows how to configure a new user to interact with the DISCOS software repository hosted on github.com. We will also show how to clone the repository and making changes in a new branch and merge it into the master development branch via a pull request. Finally we describe in detail the GitHub based workflow we decided to adopt during development and maintenance.

> **IMPORTANT NOTE:** Our complete step-by-step development flow for both new features and bug fixing is reported in *Development workflow*

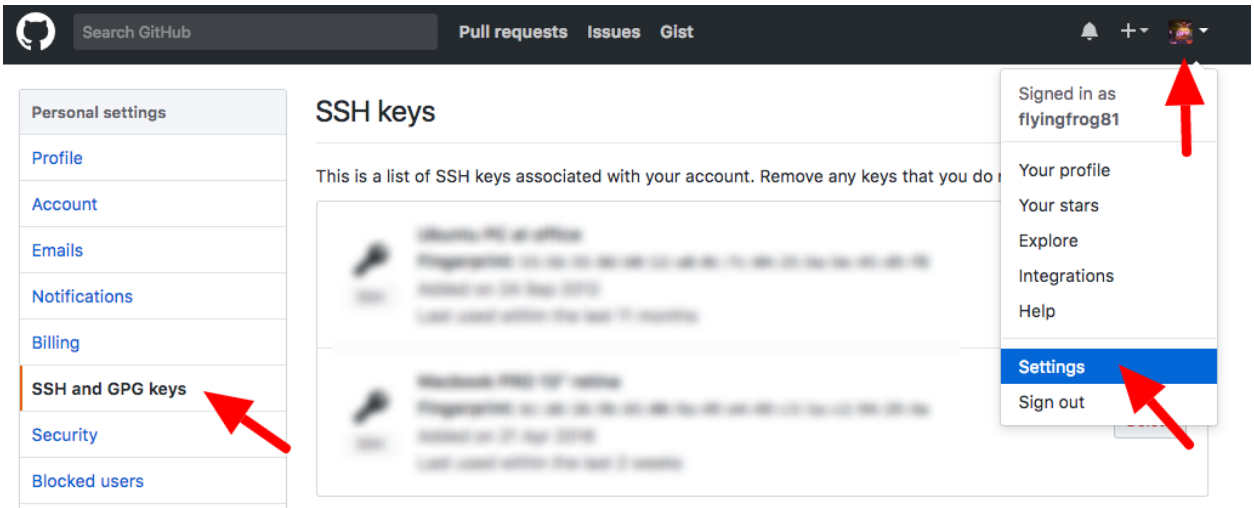
Configuring github user and repository

Access DISCOS organization

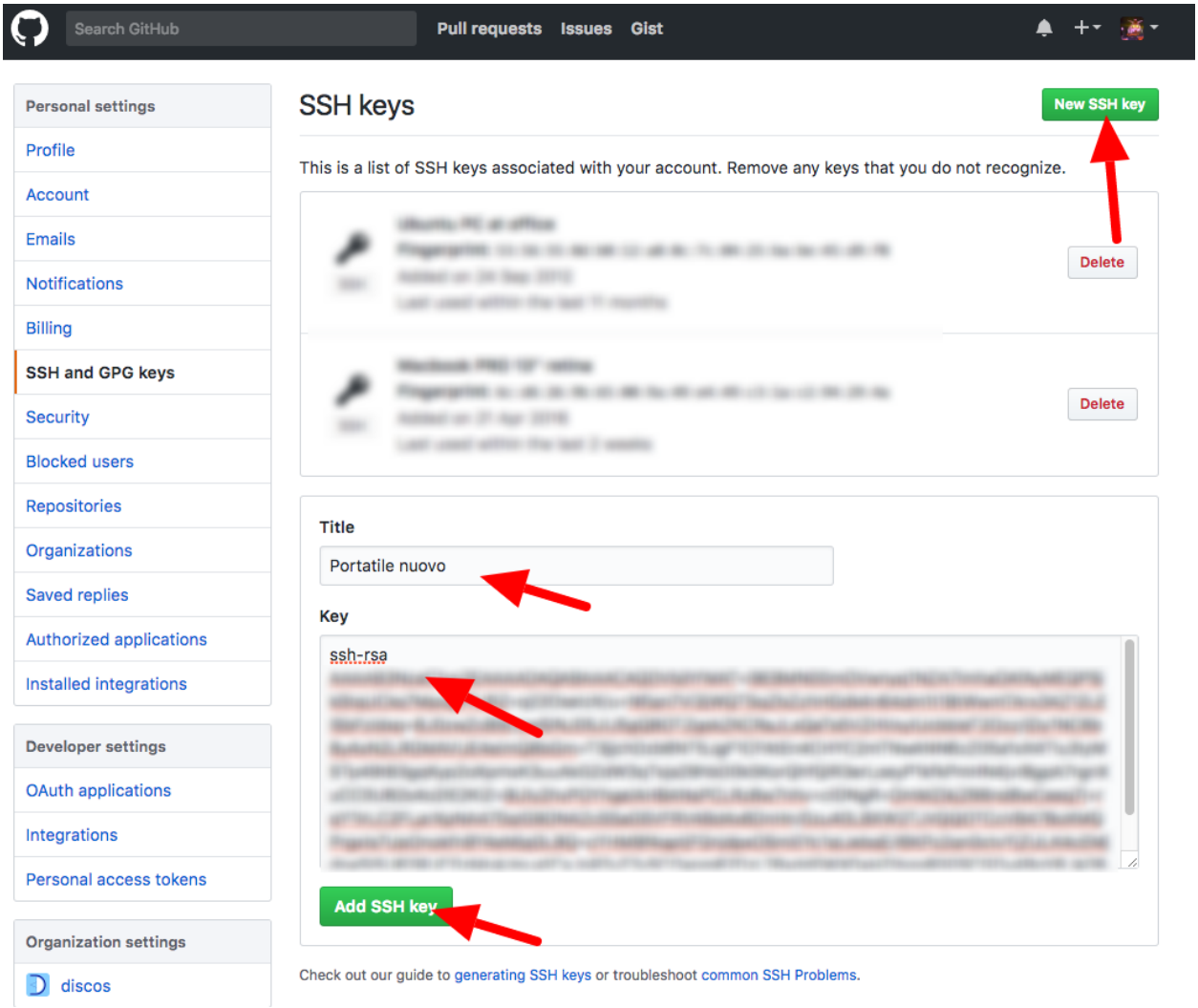
In order to configure your access to github.com/discos/discos repository you need to be part of the DISCOS organization. You can verify if you are part of the organization, visiting the settings page of your profile on github.com and verifying that DISCOS is listed in the organizations you belong to. As of 2017 it appears as an icon under your profile picture

Adding an SSH key to github user

You than need to make sure to have a proper SSH key associated with your github account. Navigate your profile settings to the **SSH and GPG keys** section



Then click on the **New SSH Key** button and insert details of your ssh key in the tab that opens up.



Clone the discos repository

You can now clone the discos repository into a new *discos* folder:

```
$ git clone git@github.com:discos/discos.git discos
Cloning into 'discos'...
remote: Counting objects: 35171, done.
remote: Compressing objects: 100% (7346/7346), done.
remote: Total 35171 (delta 20599), reused 35007 (delta 20442), pack-reused 0
Receiving objects: 100% (35171/35171), 24.43 MiB | 6.00 MiB/s, done.
Resolving deltas: 100% (20599/20599), done.
```

This will create a new directory called *discos* which will automatically track the remote **master** branch. You can see all the branches hosted on the remote server using the **git branch -a** command.

```
$ cd discos
$ ls
Common                Noto            SRT             SystemMake
Medicina      README.md       SystemBuild     UML-Project
$ git branch -a
```

(continues on next page)

(continued from previous page)

```
* master
remotes/origin/DerotatorAndMinorServo
remotes/origin/HEAD -> origin/master
remotes/origin/OffsetReview
remotes/origin/latest64
remotes/origin/master
remotes/origin/medicina-0.1
remotes/origin/noto-0.1
remotes/origin/srt-0.1
```

Adding Git aliases

We will use an alias command **gittree** in the documentation. The command is defined as:

```
alias gittree='git log --oneline --graph --decorate --color --all'
```

Feel free to define it into you *.bashrc .bash_profile* ecc.. The command is used to display an ASCII tree of the repository in a pretty-printed format with colors and essential informations about branches and tags.

```
* 04fc562 (HEAD -> master, origin/srt-0.1, origin/noto-0.1, origin/medicina-0.1,␣
↪origin/master, origin/HEAD) added gitignore and readme
* 50233f3 backport of bugfix in calOn calOff commands from escs-0.5

* 4768d39 fixed bug in buffer length too short for filenames
* a7a69bb some tuning for the Roach2 backend integration in Medicina
* 2327af6 added bash script to inject a command into the system: inject command
* b8639ef small tuning for finalizer, reviewd configuration for CHC revc
* e52308c Receievers CHC included in Medicina system
...
```

Contributing to DISCOS

Contents

- *Contributing to DISCOS*
  - *Making a change on the master branch*
    - *Making a commit in your local repository*
    - *Pushing into the remote repository*
  - *Developing on a feature branch (RECOMMENDED)*
    - *Creating a new branch*
    - *Making changes and moving between branches*
    - *Generating a pull request*
    - *Accepting a pull request*
    - *Updating your local copy*

Making a change on the master branch

The **master** branch is our main development branch, we do our best to keep it stable and clean but we should never assume that master is a production-stable branch.

Working on the master branch is simple, the only difference with svn is that now you work on a local copy of the project, and commits are done on your workstation rather than on the server. Once you feel confident of your changes you can push them on the remote repository stored on github.com so that others can access your changes.

Making a commit in your local repository

First we verify that our repository is up-to-date with the remote repository, then we edit a file and note how git is aware of the new change.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
$ git branch
* master
$ echo "DISCOS is in use at three Italian radio telescopes" >> README.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

The git status command shows and tracks all the changes made. There are cases this is not the best option. If you want to track changes only on files already under the control of the repository you should add an option.

```
$git status -uno
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit (use -u to show untracked files)
```

Just as SVN, git offers a **diff** command that shows you what changes:

```
$ git diff
diff --git a/README.md b/README.md
index 041992e..f1d6f53 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,4 @@
 # DISCOS

  Discos is the Development of Italian Single-dish COntrol System
  +DISCOS is in use at three italian radio telescopes
```

We now add the changes to the next commit operation and create a new commit object with an associated message. If the commit deals with an open issue the test should reference the issue:

```
$ git add README.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

     modified:   README.md

$ git commit -m "fix issue #xxx: updated README.md for the doc example"
[master 4ba4caa] fix issue #xxx: updated README.md for the doc example
 1 file changed, 1 insertion(+)
```

We can see how git has just created a new *commit* on our local repository. The commit is uniquely identified by the hashtag starting with *4ba4caa* and is on the branch *master*. The commit also has associated a messaged that we specified on the command line. We could achieve the same result in one single step adding every changeset to the commit automatically using the *-a* option, without explicitly adding the files:

```
$ git commit -am "fix issue #xxx: updated README.md for the doc example"
```

If we look now at the status and the commit tree of our local repo we can see:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
$ gittree
* 4ba4caa (HEAD -> master) updated README.md for the doc example
* 04fc562 (origin/srt-0.1, origin/noto-0.1, origin/medicina-0.1, origin/master,␣
↪origin/HEAD) added gitignore and readme
```

We can see how our newly created commit object is automatically labelled **HEAD** and is positioned one step forward with respect to the remote repository identified by **origin/HEAD**

Pushing into the remote repository

If we want the remote repo to do this same step we need to *push* the commit object into the remote repository:

```
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 385 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:discos/discos.git
   04fc562..4ba4caa  master -> master
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
$ gittree
* 4ba4caa (HEAD -> master, origin/master, origin/HEAD) updated README.md for the doc␣
↪example
...
```

We can see how **HEAD** and **origin/HEAD** are now positioned on the same commit object. If we look at the repository stored on github.com we would see this same commit has appeared.

Developing on a feature branch (RECOMMENDED)

Using a new branch for the development of a new feature is a common practice among git users. This comes with some advantages:

- git branches are lightweight
- git merge is really effective, no need to be scared
- merging two branches can be realized using a **pull request** mechanism which enables review of the proposed change
- Pull requests can be tested in isolation before being accepted

For any non trivial change **we strongly recommend** to adopt this approach .

Creating a new branch

A new branch can be created on your local copy of the repository using the **checkout** command with the **-b** option:

```
$ git checkout -b feature-branch-example
Switched to a new branch 'feature-branch-example'
$ git branch -a
* feature-branch-example
  master
```

We are automatically positioned on the new branch and we can start working on this.

Making changes and moving between branches

We will add another line to the README file and generate the commit on the newly created branch:

```
echo "A different product line is maintained for each radio telescope" >> README.md
$ git commit -am "README modified on feature branch"
[feature-branch-example 6dada91] README modified on feature branch
 1 file changed, 1 insertion(+)
$ gittree
* 6dada91 (HEAD -> feature-branch-example) README modified on feature branch
* 4ba4caa (origin/master, origin/HEAD, master) updated README.md for the doc example
```

We can see how our commit is positioned on the HEAD of the new branch, that is one step forward with respect to the master branch. In order to make our example a litlle more realistic and demonstrate a more concrete pull request, we will add a commit on the master branch. This will also show how to move between branches:

```
$ git branch
* feature-branch-example
  master
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git branch
feature-branch-example
```

(continues on next page)

(continued from previous page)

```
* master
$ vim .gitignore
$ git commit -am "minor change in gitignore"
[master 248e9c8] minor change in gitignore
 1 file changed, 1 insertion(+)
$ gittree
* 248e9c8 (HEAD -> master) minor change in gitignore
| * 6dada91 (feature-branch-example) README modified on feature branch
|/
* 4ba4caa (origin/master, origin/HEAD) updated README.md for the doc example
```

We can see that the branch **master** has now diverted from **feature-branch-example**. To better understand how git works we can now switch between branches and see how commits are applied to the files:

```
$ git branch
feature-branch-example
* master
$ less .gitignore
*~
.svn
*swp
...
$ less README.md
# DISCOS

Discos is the Development of Italian Single-dish COntrol System
DISCOS is in use at three italian radio telescopes

$ git checkout feature-branch-example
$ git branch
* feature-branch-example
  master
$ less .gitignore
.svn
*swp
...
$ less README.md
# DISCOS

Discos is the Development of Italian Single-dish COntrol System
DISCOS is in use at three italian radio telescopes
A different product line is maintained for each radio telescope
```

We can see how the commits are incorporated into the repository depending on the branch.

Generating a pull request

Now we want to generate a **pull request** so that the changes made in our feature branch can be seen by everyone, reviewed, tested, and eventually merged into the master branch or declined.

We start by pushing the local branch into the remote repository:

```
$ git push -u origin feature-branch-example
Counting objects: 3, done.
Delta compression using up to 4 threads.
```

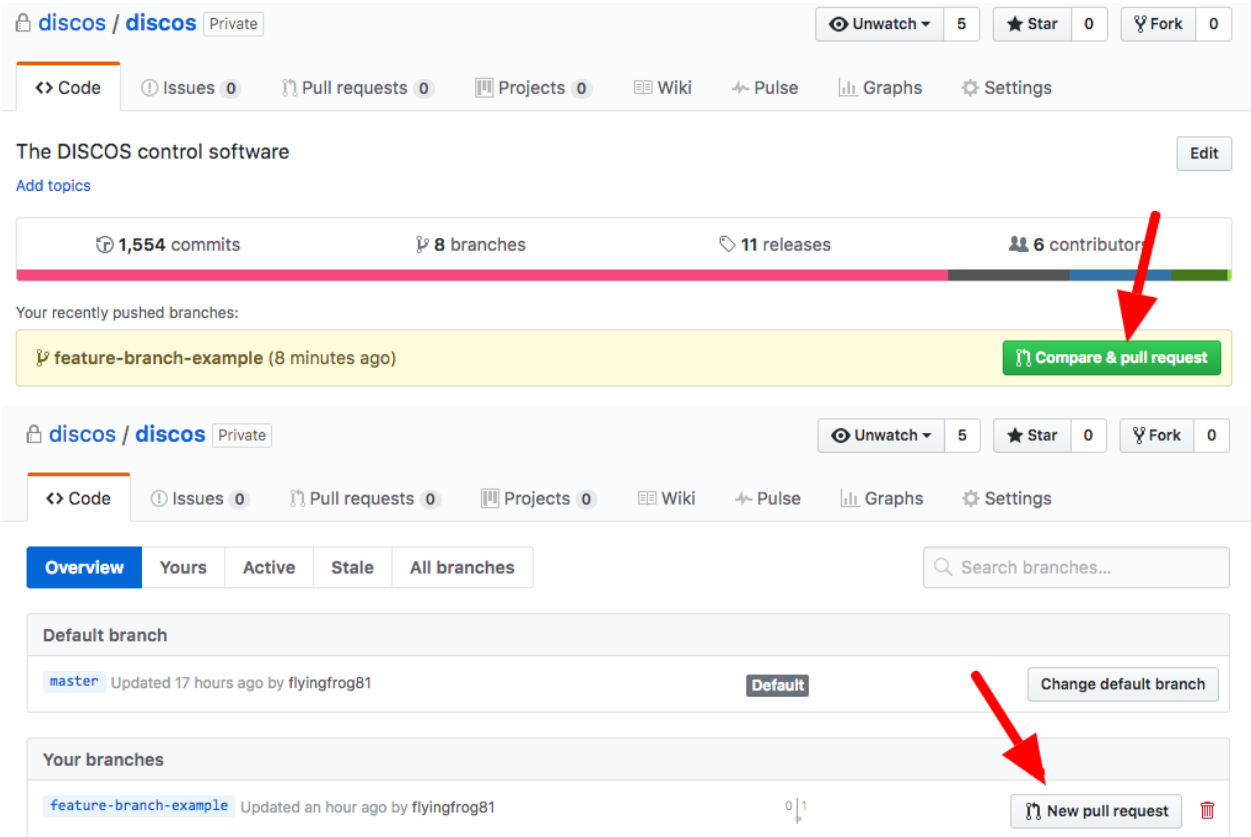
(continues on next page)

(continued from previous page)

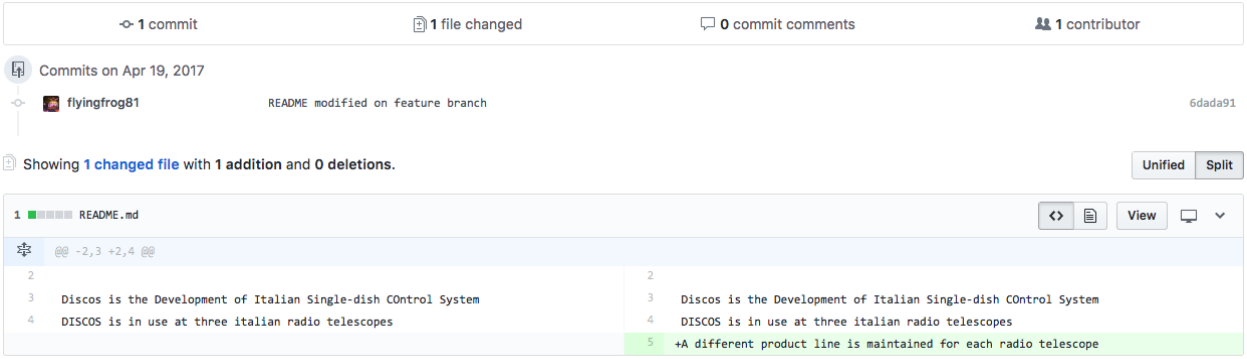
```
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 360 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:discos/discos.git
 * [new branch]      feature-branch-example -> feature-branch-example
Branch feature-branch-example set up to track remote branch feature-branch-example␣
↪from origin.
```

The **push** command asks git to copy the local branch called *feature-branch-example* into the remote copy at *origin*, that in our case points to the github repo. The **-u** option sets this remote branch as *upstream* for this local branch, this means that further changes on this branch will be pushed to the remote branch just configured. We now move to the github.com site and manage the pull request from there.

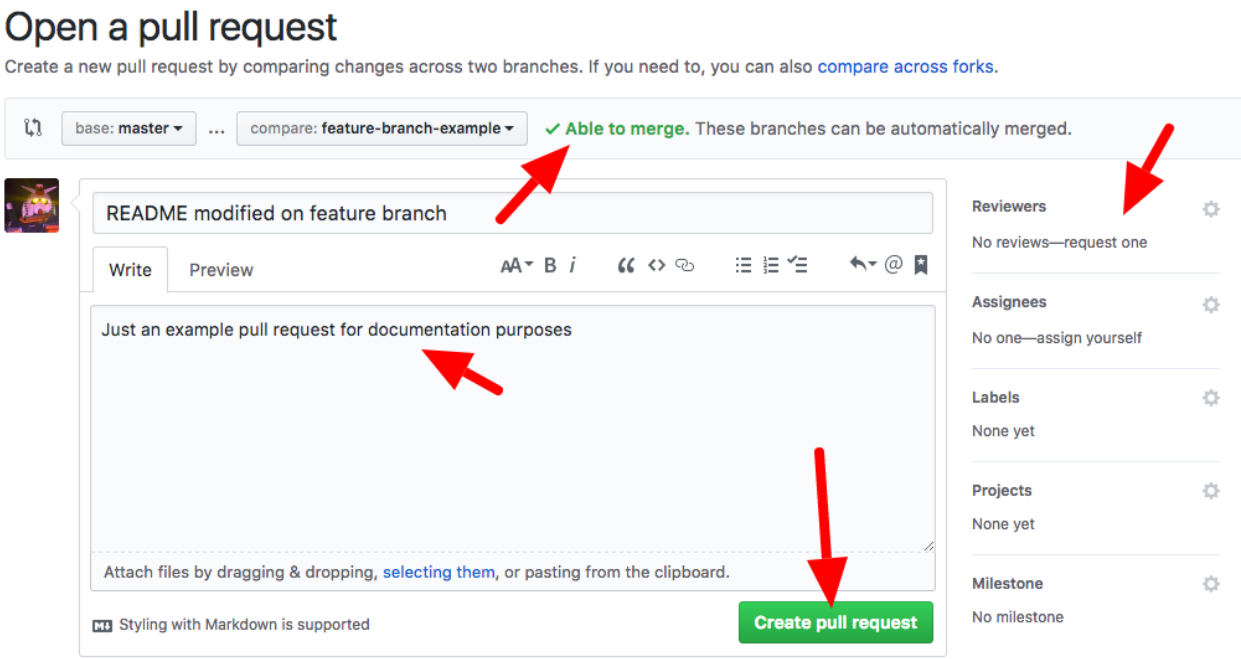
We create the pull request from the new branch directly from the github homepage or navigating to the **branches** tab of the github repo.



We are now prompted with a page summarizing facts about our pull request. At first, if we scroll down the page, we can see a summary of changes introduced by this requet:

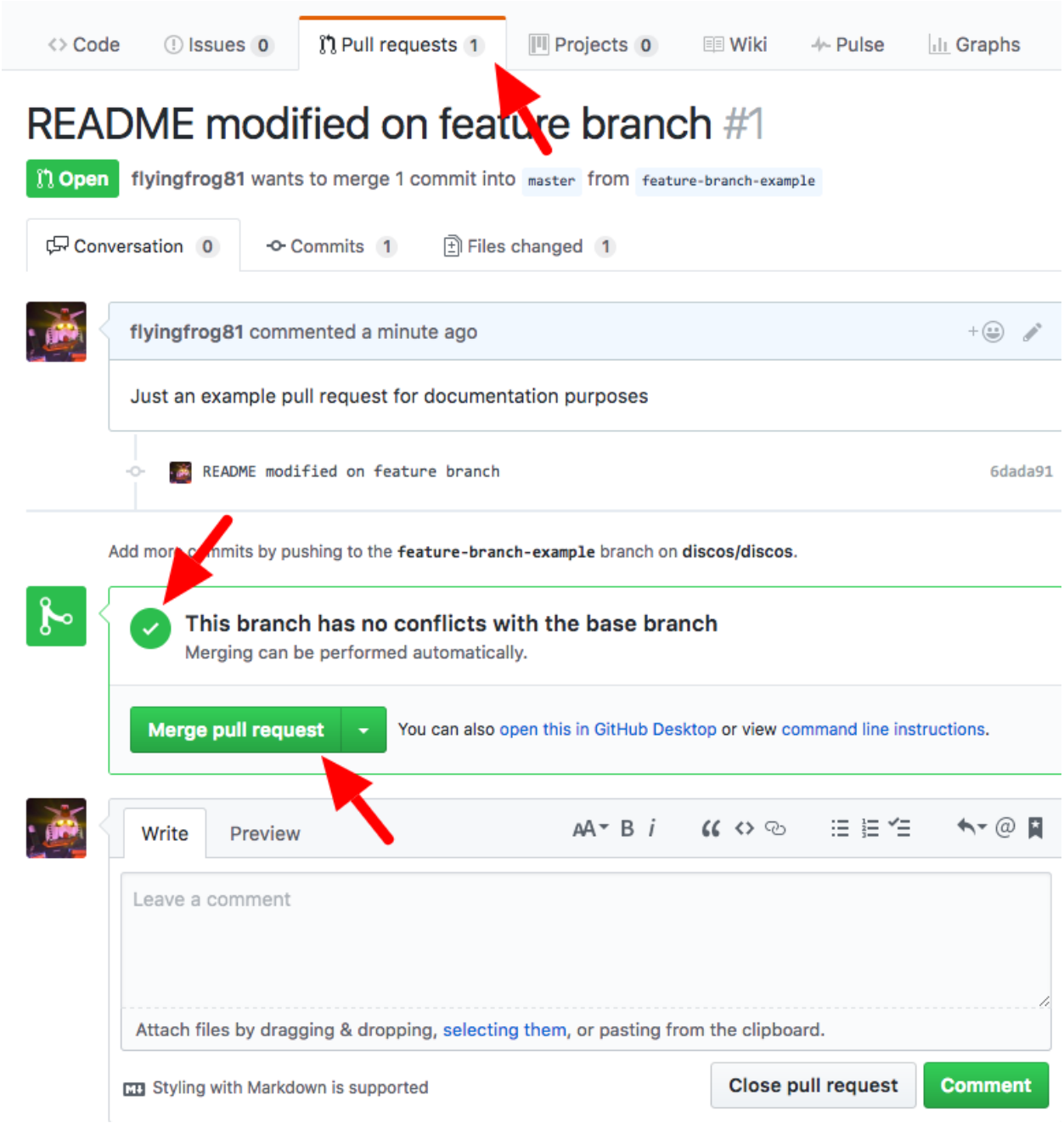


On the top of the page we can actually generate the request. You can see that git is already telling us that the request can be merged without conflicts, you can add comments, ask for a review of someone in particular, assign labels, milestones etc.. finally we generate the pull request:

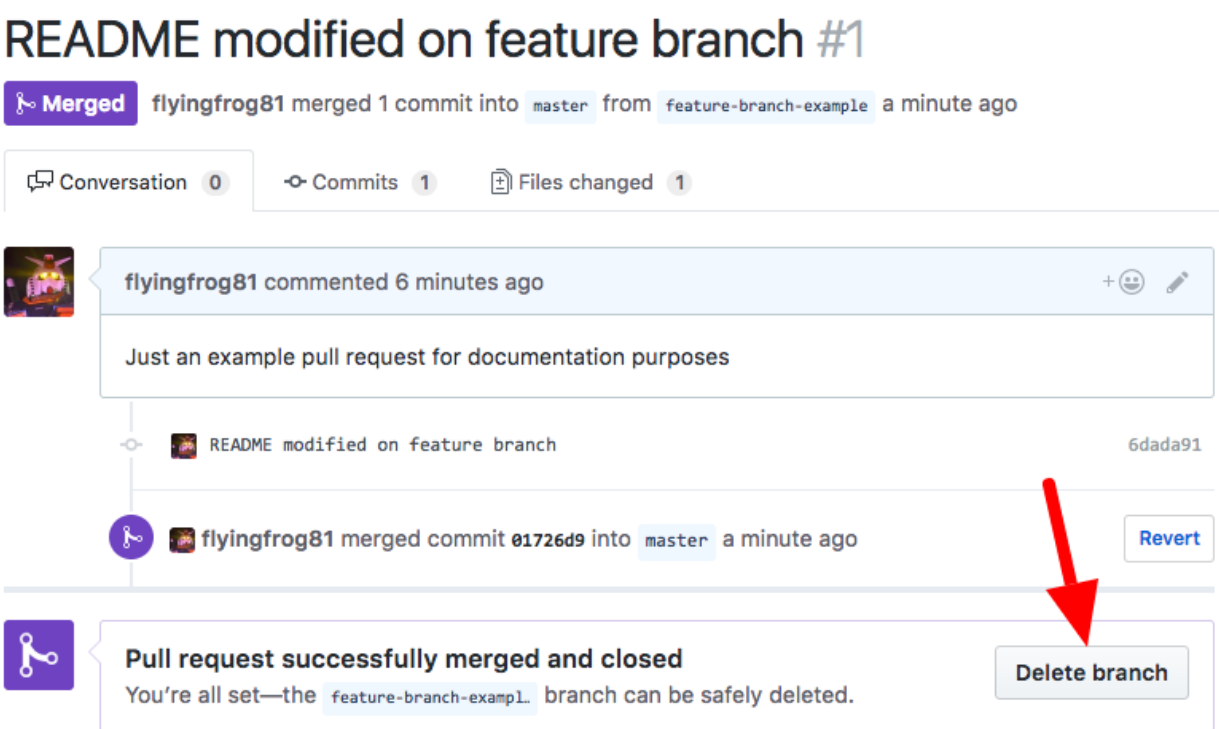


Accepting a pull request

A new page is generated for this pull request. From this page, everybody can review the proposed changes, make comments, and eventually accept the request for merging it into the master branch:



The request is merged, and we have the possibility to revert it or eventually to delete the feature branch from the remote repo. We will delete this as keeping it will only pollute our environment, now that changes have been accepted into master.



Updating your local copy

We can now opsition ourselves on our local master branch and update it to reflect changes in the remote repo:

```
$ git checkout master
$ git pull
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From github.com:discos/discos
  4ba4caa..01726d9  master     -> origin/master
Merge made by the 'recursive' strategy.
  README.md | 1 +
  1 file changed, 1 insertion(+)
$ gittree
*   3f9086f (HEAD -> master) Merge branch 'master' of github.com:discos/discos
|\
| *   01726d9 (origin/master, origin/HEAD) Merge pull request #1 from discos/feature-
↪branch-example
| |\
| | * 6dada91 (origin/feature-branch-example, feature-branch-example) README modified␣
↪on feature branch
| |/
* | 248e9c8 minor change in gitignore
|/
* 4ba4caa updated README.md for the doc example
* 04fc562 (origin/srt-0.1, origin/noto-0.1, origin/medicina-0.1) added gitignore and␣
↪readme
```

A new commit is automatically created for merging our local changes with changes in the remote branch. We can now push our local changes to the remote branch.

```
$ git push
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 567 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 2 local objects.
To github.com:discos/discos.git
   01726d9..3f9086f  master -> master
```

And see how everything is now aligned:

```
$ gittree
*   3f9086f (HEAD -> master, origin/master, origin/HEAD) Merge branch 'master' of␣
↪github.com:discos/discos
|\
| *   01726d9 Merge pull request #1 from discos/feature-branch-example
| |\
| | * 6dada91 (origin/feature-branch-example, feature-branch-example) README modified␣
↪on feature branch
| |/
* | 248e9c8 minor change in gitignore
|/
* 4ba4caa updated README.md for the doc example
```

A number of commits have been created for the purpose of merging branches, these could be avoided using different merge strategies.

GitHub based workflow

Contents

- *GitHub based workflow*
  - *Remarks on working with DISCOS GitHub repository*
  - *Hot-Fixes workflow*
  - *Feature implementation workflow*

Remarks on working with DISCOS GitHub repository

All the submissions, including submissions by project members, require a review by a member of the core team. We use GitHub pull requests for this purpose. Any feature, enhancement or bug must be submitted to GitHub tracking system before starting any further elaboration. All subsequent changes are done on dedicated branches that clearly reference the issue number. DISCOS settle on two persistent branches, the *master* branch dedicated to development and the *stable* branch in which the releases are maintained.

Hot-Fixes workflow

- Check out into master branch: *git checkout master*

- Fetch all remote updates: *git remote update*
- Update local master branch with remote copy: *git pull origin master*
- Check out into stable branch: *git checkout stable*
- Update local stable branch with remote copy: *git pull origin stable*
- Create a hotfix branch related to the issue xxx: *git checkout -b fix-issue-xxx*
- Do some fixes and commit to them
- Push the hotfix branch to remote repository: *git push origin fix-issue-xxx*
- Open a "Pull request" in GitHub, in order to merge the hotfix branch onto the stable branch, and reqeust a review from the team
- Once the team approves the pull request, the branch can be merged using the "Squash and merge" strategy, be sure to not delete the hotfix branch
- Open a second "Pull request" in GitHub, in order to merge the hotfix branch onto the master branch
- Once again, once the team approves the pull request, the branch can be merged using the "Squash and merge" strategy, this time the branch can be deleted
- If some conflict between the master and the hotfix branch arises, don't worry, they can be fixed right away when merging online. Resolve the conflict by porting the new lines from the hotfix branch into the master branch version of the file.
- Finally, delete the hotfix branch from the local repository: *git branch -D fix-issue-xxx*
- Now that the hotfix is on the online repository, a new tag can be created onto the stable branch. In order to do this, go to the releases page of the repository, and select "Draft a new release". Fill the "Tag version" and "Release title" fields according to the previous releases. Also, be sure to select the stable branch as target and to check "This is a pre-release" if you are not sure if your release is ready to be deployed on production line.

Feature implementation workflow

- Check out into master branch: *git checkout master*
- Fetch all remote updates: *git remote update*
- Update local master branch with remote copy: *git pull origin master*
- Create a branch related to the issue xxx: *git checkout -b fix-issue-xxx*
- Do some coding and commit to it: *git commit -m "Fix #xxx:…"*
- Push feature branch to remote repository: *git push origin fix-issue-xxx*
- Open a "Pull request" in GitHub for team to verify the feature
- Once the team approves the pull request, the branch can be merged using the "Squash and merge" strategy, and it can be deleted from the online repository
- Finally, delete the feature branch from the local repository: *git branch -D fix-issue-xxx*

DISCOS customized logging system

Abstract

This howto describes how to custom logging API are used inside DISCOS. The system is very simple and requires the use of very few APIs or macros. It supports (with small differencies both Python and C++). Python implementation suffers from several bugs of ACSPy libraries so we put in place some workarounds. The basic reference manual of DISCOS custom logging is given in this `document` from *Marco Bartolini*. The rest of this section covers some items not described or updated in the above document. The general idea behind this implementation is that we'd like to avoid to flood the user with the hundreds of log messages generated inside the system. The decision about which message has to be presented to the user is left to the developer. The general rule that an exception should be logged only by the initiator of the call sequence (no further throws), remains valid. In summary the following items are covered:

- System loggings (messages are not shown to the user, they are stored in system log file)
- Custom loggings (messages are reported to the user, the same message is stored into both system and user logfile)
- System exceptions (The user is not directly notified of the exception and the event is logged only into system log file)
- Custom exceptions (The user is notified of the exception and the event is logged into both system and user log file)
- Interaction with operator input (This is a special case allowing the developer to present a specific message to the user when an exception is risen inside an operation triggered by the user. The message is shown in the operator input)

C++

System loggings

This are normal ACS logging macros.

```
ACS_LOG(LM_FULL_INFO,"MyModule::MyClass::MyMethod",(LM_INFO,"My system log message"));
```

Custom loggings

```
#include <IRA>
CUSTOM_LOG(LM_FULL_INFO, "MyModule::MyClass::MyMethod", (LM_ALERT, "My user log␣
↪message"));
```

System exceptions

```
_EXCPT(ErrorsNamespace::MyErrroEventExImpl,dummy,"MyModule::MyClass::MyMethod");
dummy.log(LM_DEBUG); //
throw dummy; // if required
```

Custom exceptions

```
#include <IRA>
_EXCPT(ErrorsNamespace::MyErrroEventExImpl,dummy,"MyModule::MyClass::MyMethod");
CUSTOM_EXCPT_LOG(dummy,LM_ERROR);
throw dummy; // if required
```

Interaction with operator input

This macro is for components development. The user message is encapsulated with the exception and presented to the user if the error is directly generated by a user command.

```
#include <ParserConnector.h>
_EXCPT(ErrorsNamespace::MyErrroEventExImpl,dummy,"MyModule::MyClass::MyMethod");
_ADD_USER_MESSAGE(dummy,"My special message to the user"); // Show in the operator␣
↪input if the error was triggered by an user command
throw dummy; // this will be logged by the caller
```

This macro is for stand alone programs that are issued as system calls by the command parser. The user message is handled like the other case and it's also written into standard err and intercepted the the parser to be presented to the user.

```
#include <ParserConnector.h>
_EXCPT(ErrorsNamespace::MyErrroEventExImpl,dummy,"MyModule::MyClass::MyMethod");
_ADD_USER_MESSAGE_DIRECT(dummy,"My special message to the user"); // Show in the␣
↪operator input
throw dummy; // this will be logged by the caller
```

Python

System loggings

```
from IRAPy import logger
     logger.logNotice("My system log message")
```

Custom loggings

```
from IRAPy import userLogger
logger.logNotice("My user log message")
```

System exceptions

```
from IRAPy import logger
ex=MyErrroExImpl()
logger.logException(ex)
```

Custom exceptions

```
from IRAPy import userLogger
ex=MyErrroExImpl()
userLogger.logException(ex)
```

Interaction with operator input

This function is for components development. The user message is encapsulated with the exception and presented to the user if the error is directly generated by a user command.

```
from IRAPy import logger
from SimpleParserPy import add_user_message
ex=MyErrroExImpl()
add_user_message(ex,"My special message to the user",False)
raise ex
```

This macro is for stand alone programs that are issued as system calls by the command parser. The user message is handled like the other case and it's also written into standard err and intercepted by the parser to be shown to the user.

```
from IRAPy import logger
from SimpleParserPy import add_user_message
newEx = ClientErrorsImpl.CouldntPerformActionExImpl()
add_user_message(newEx,"My special message to the user")
userLogger.logException(newEx)
sys.exit(1)
```

Adopted formatting choices

The *user's guides* source files were written according to the following formatting conventions:

>>>********** >>>Page Titles >>>**********

>>>| ******* >>>| Page Titles >>>| *******

>>>Section Titles >>>==========

>>>| Section Titles >>>| ==========

>>>Subsection Titles >>>————————

>>>| Subsection Titles >>>| ————————

>>>.. describe:: Lists of commands

>>>| .. describe:: Lists of commands

```
>>> Tables
```

>>>======== ========== ========== >>>Col1 Col2 Col3 >>>——— ————- ————- >>>unit unit unit >>>======== ========== ========== >>>Text1 1 1 >>>Text2 2 2 >>>======== ========== ==========

>>>| ======== ========== ========== >>>| Col1 Col2 Col3 >>>| ——— ————- ————- >>>| unit unit >>>| ======== ========== ========== >>>| Text1 1 1 >>>| Text2 2 2 >>>| ======== ========== ==========

2.2 DISCOS control software team

2.2.1 Core developers

Leading DISCOS is a small team of core developers, that is people who have made and continue to make a significant contribution to the project, and have a good understanding not only of the code, but also the longer-term aims and directions of the project:

- *Marco Bartolini*
- *Marco Buttu*
- *Carlo Migoni*
- *Andrea Orlati*
- *Sergio Poppi*
- *Giuseppe Carboni*

2.2.2 Astronomical referees

Astronomers are involved in the various phases of the software design, implementation and in-antenna testing, in order to exploit their specific knowledge and ensure that our system answers to the needs of the astronomical community:

- *Sergio Poppi*
- *Simona Righini*

2.2.3 Hardware and network support

- *Antonietta Fara*

2.2.4 Contributors

Anyone can be a contributor to DISCOS project. Being a contributor simply means that you take an interest in the project and contribute in some way, ranging from asking sensible questions (which documents the project and provides feedback to developers) through to providing new features as components, patches or user's specifications.

If you become a valuable contributor to the project, you may well be invited by the *core developer* to become member of the team.

Here is the list of DISCOS contributors:

- *Andrea Melis*

2.2.5 Responsibilities

Project manager

The *project manager* heads the whole DISCOS project and has in charge the software releasing. This responsibility is currently assigned to *Andrea Orlati*.

Support scientist

The *support scientist* is responsible for every communication related to DISCOS changes that affect the users. This responsibility is currently assigned to:

- Medicina: *Simona Righini*
- Noto: ?
- Sardinia Radio Telescope: *Sergio Poppi*

Observation manager

The *observation manager* is responsible for the observations with DISCOS:

- Medicina: *Simona Righini*
- Sardinia Radio Telescope: *Marco Buttu*
- Noto: ?

User's documentation

The *user's documentation manager* is responsible for the *User's guide: Observing with DISCOS* contents. This responsibility is currently assigned to *Simona Righini*.

Development

The *development manager* is responsible for the *Development guide* contents, for the software (and documentation) repository and for the whole development process. This responsibility is currently assigned to *Marco Buttu*.

Deploying and continuous integration

This responsibility is currently assigned to *Giuseppe Carboni*.

Provisioning

This responsibility is currently assigned to *Giuseppe Carboni*.

2.2.6 People involved in the project

Marco Bartolini

Marco…

Marco Buttu

Marco Buttu works for the Osservatorio Astronomico di Cagliari, and he is a member of the Sardinia Radio Telescope staff.

Giuseppe Carboni

Giuseppe works at the Osservatorio Astronomico di Cagliari, and he is a member of the Sardinia Radio Telescope staff.

Carlo Migoni

Carlo works at the Osservatorio Astronomico di Cagliari, and he is a member of the Sardinia Radio Telescope staff.

Antonietta Fara

Antonietta…

Andrea Melis

Andrea Melis…

Andrea Orlati

Andrea works at the Medicina Radiotelescope of the INAF - Institute of Radioastronomy.

via Fiorentina, 3513
40059 - Medicina (BO) - Italy
Phone: +39 051 696 5831
Fax: +39 051 696 5810

Sergio Poppi

Sergio works for the Osservatorio Astronomico di Cagliari, and he is a member of the Sardinia Radio Telescope staff.

Simona Righini

Simona: user manual, software specification definition, …

BIBLIOGRAPHY

[EBNF] http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[KAT] https://casper.berkeley.edu/wiki/images/1/11/NRF-KAT7-6.0-IFCE-002-Rev4.pdf

[TELNET] http://www.freesoft.org/CIE/RFC/1123/31.htm