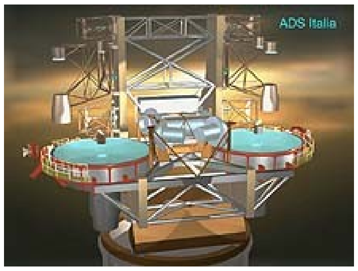


LBT PROJECT
2x8,4m TELESCOPE

Doc. No.	: 481s011
Issue	: c
Date	: March 15, 2007
Issued by	: Jose Luis Borelli

LBT PROJECT

2 X 8,4m OPTICAL TELESCOPE

Instrument Interface for the LBT Telescope Control System, C ++ Interface Control Document

	Signature	**Date**
Prepared	Jose Borelli	March 2, 2007
Reviewed	Martin Kuerster, Norm Cushing, Dave Thompson, Chris Biddick, Wolfgang Gaessler, Michele de la Peña	March 15, 2007 July 10, 2007
Approved		

1 Revision History

Issue	**Date**	**Changes**	**Responsible**
a	02-Mar-2007	First draft	Jose Borelli
b	15-Mar-2007	First revision.	Jose Borelli
c	15-Jul-2007	New classes added, Hotspot and Offset. New rotators commands added to the commands set. Strings variables changed to enumerated types.	Jose Borelli

2 Table of contents

3 List of abbreviations

Abbreviation	Description
AGW	Acquisition, Guiding, and Wavefront sensing
ALT	Altitude
AOS	Adaptive Optic Subsystem
AOS-Sup	AO Supervisor
AZ	Azimuth
CSQ	Instrument Interface Server
DEC	Declination
GCS	Guiding Control Subsystem
ICS	Instrument Control Software
IIF	Instrument InterFace
IRA	Initial Rotation Angle
LBT	Large Binocular Telescope
LBTI	Large Binocular Telescope Interferometer
LBTO	LBT Organization
LINC	LBT INterferometric Camera
LN	Linc Nirvana
LUCIFER	LBT NIR spectroscopic Utility with Camera and Integral-Field Unit for Extragalactic Research
M1	Primary Mirror
M2	Secondary Mirror
M3	Tertiary Mirror
MODS	Multiple Object Double Spectrograph
NIRVANA	Near-InfRared / Visible Adaptive iNterferometer for Astronomy
OPE	OPtical Element
PCS	Pointing Control Subsystem
PEPSI	Postdam Echelle Polarimetric and Spectroscopy Instrument
RA	Right Ascension
SFP	Standard Focal Plane
TBD	To Be Defined
TCS	Telescope Control System
WF	Wave Front

4 About this document

The Instrument Interface (IIF) is the software interface which allows the instrument software to communicate with the Telescope Control System (TCS), in order to issue commands which provide control, allow a transparent communication with the different subsystem and acquire status information. This document specifies the Instrument-Telescope control software, the different structures and the command set.

4.1 Purpose

The purpose of this document is to serve as a reference manual for software developers that want to use the C++[1] Instrument Interface library to communicate with the TCS, describing the set of commands that LBT will provide to all the instruments.

4.2 Notes

The document was divided into different sections. Each section has a flat structure and an unnumbered sequence of sub sections. The aim is to present each command or new topic in a confined space so that it can be quickly grasped. The list of commands presented in section 7 are in alphabetical order. At the moment, some of these commands are prototypes and they may change in the future.

1 The C interface is described in another document, 481g010d. See reference [1].

5 Introduction

The Instrument Interface is distributed to the LBT instrument software teams as a set of libraries. These libraries must be included into the instrument control software projects in order to issue commands.
The instrument is required to identify itself to the TCS. This is done by specifying a predefined, unique ID value during the instantiation of an IIF object. This globally unique identifier is composed of a instrument ID (1 -24 characters) plus a string, which determines the focal station and the telescope side to be controlled by the commanding instrument.

When an instrument calls an IIF command, this action triggers processing inside the TCS. The instrument then, has the option of halting the execution of its program thread until the requested TCS-internal processing has finished or, alternatively, the instruments' control code can proceed with its own program thread and decide to periodically query the status of the IIF command.

6 Functionality and Observation modes

All the astronomical instruments require similar basic functionalities from the TCS, which are shown in figure 6.1. Within this chapter we describe the functionalities in a general form.

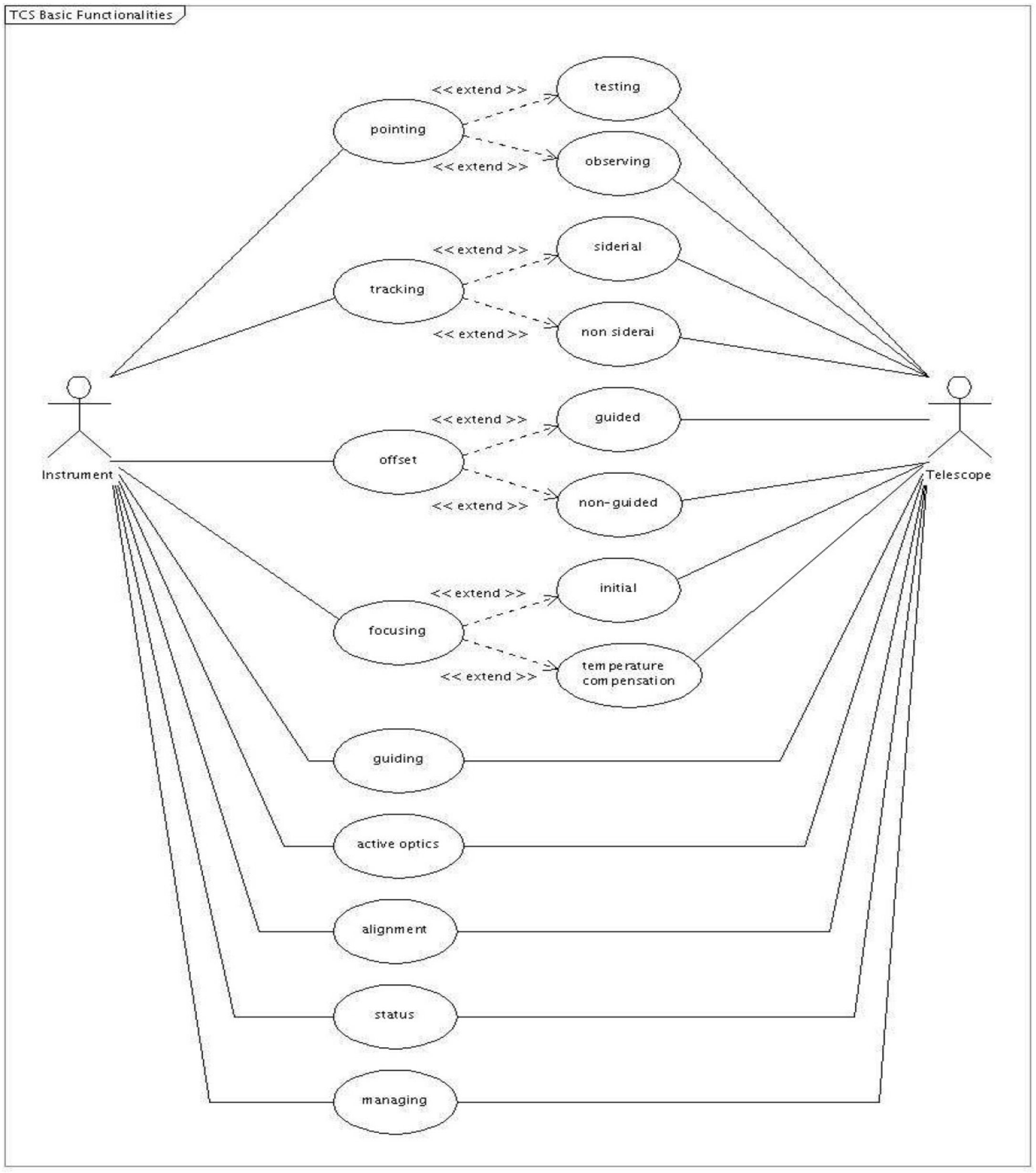


Figure 6.1:Use cases of the basic telescope functionalities.

6.1 Control Functionality

6.1.1 Authorizing with the TCS

Under normal operating conditions, only a single instrument can have full control of a telescope side; this control is obtained through the authorization command. When an instrument is authorized for a telescope side, it is the only instrument which may successfully issue a majority of the commands. Nevertheless, there are some functions the instruments can use without authorization, like requesting status or access to a limited set of rotator commands.

6.1.2 Giving up control

When the instrument is not using anymore one or both sides of the telescope, the `deauthorize` command must be issued, in order to yield all control of the corresponding side to the TCS.
Note, that multiple authorizations/deauthorizations from the same instrument are allowed by the CSQ, maintaining an internal count. A final deauthorize happens when the count goes to zero (more details in section 8.5 and table 8.3 as well). This is useful for instance, if the ICS is multi-threaded. Each thread that needs to interact with the TCS must authorize itself.
LBC is using this feature to authorize itself, and contemporaneously, the LBC active-optics program authorizes itself (a second authorization) to send the Zernickes it calculates from the pupil images.

6.1.3 Status

The instruments are able to ask for status of the TCS parameters and the telescope situation. A status request could be on remaining ranges of optical elements (primary, secondary, etc.), actual coordinates (the telescope is pointing on), enclosure (open or closed), ambient temperature, etc. All this information is not provided in real-time.
In the same way, the instrument is able to set a parameter or a group of parameters in the data dictionary, in order to give status information about the instrument to the TCS.

6.2 Basic Functionality

6.2.1 Pointing

Pointing defines where the telescope is looking on the sky in conjunction with where the astronomical object is imaged in the instrument focal plane. In order to point the telescope, the mount is moved to a position computed by the PCS which accommodates the observer request, accounts for a variety of observational corrections, and derives tip/tilt for the mirrors as

necessary.
In order to begin an observation cycle by slewing the telescope to the target position, IIF has the `presetTelescope()` command. This IIF command has a large number of arguments, which are described in detail in the section 7.28, PresetTelescope.

Pointing sequence

1. The instrument sends the telescope the preset information for each side (left, right or both) that is to be preset. Presetting both sides only makes sense for dual-side instruments being controlled through a single interface.
2. The telescope checks if the instrument is authorized.
3. The telescope checks the preset parameters for sanity.
4. The PCS applies proper motion and precession corrections, as necessary, to the input target and guide star objects.
5. The PCS computes tracking polynomials for the mount.
6. The Telescope performs the Guide Star Acquisition, picking a guide star from the set of passed guide stars in the order they were passed.
7. If steps 2 through 6 succeed, the telescope is on a new position, tracking, guiding and controlling active optic as well.
8. The Telescope informs the Instrument about success or failure of the preset operation.

6.2.2 Tracking

Tracking means to follow the science target in open loop, without guiding. It will be possible to follow a non-sidereal target (this functionality is not yet implemented as of July 2007), and in some occasions it is possible to switch-off the tracking.

6.2.3 Offset

During an observation it is often needed to offset the telescope; this means to move a small distance in RA and DEC. This might happen in two different ways, Non-guided offset or Guided-offset.
In the first one, the telescope moves a small delta RA and DEC but the

guiding loop opens because the distance is too large or the accuracy is not so relevant.
In the second case, the telescope moves a small delta RA and DEC, but the guiding loop is kept closed. This increases the accuracy. The guiding system has to take care of the jump the guide star will experience and has to reference the position of the guide star to its initial position.
IIF provides two commands for this purpose, OffsetGuiding and OffsetPointing (more details in sections 7.24 and 7.25):

6.2.4 Focusing

The optomechanics of telescope and instrument are normally not that perfect to be in focus without adjusting some parts within the optical path. Also, as the structure of the telescope and instrument changes with different temperatures, the focus has to be adjusted from time to time. Focusing needs information from the instrument. With LBT in adaptive mode this will be handled by the AOS.

Note, that for all Gregorian instruments, the Acquisition, Guiding, and Wavefront sensing units (AGw/W units) will accurately maintain the focus of the telescope relative to the AGw/W units doing the wavefront analysis. So, to effect an offset of the telescope with respect to the instrument, the w/W stage must be moved relative to the instrument. Offsetting the w/W stage in z will automatically drag the focus along with it because of the active/adaptive feedback. For seeing-limited instruments, it is likely sufficient to allow the TCS to offload the focus to whatever optic it thinks is best. The interferometers will likely want to specify that any offloading of the focus is done to specific optical elements (or groups of them) in order to preserve (or change) the plate scale at the focal plane.

<u>Positioning the Focus</u>
1.The instrument sends the telescope a request to move the respective OPE to a new absolute focus position, specified by:
   (a) the new position (in millimeters) of the optical element in the z-axis, and
   (b) the required OPE.
2.The telescope checks if the positioning information requested in step 1 is within range.
3.In case of success of step 2, the telescope positions the OPE to the focus desired position.
4.The telescope confirms success or failure of the operation to the Instrument.

Incrementing/Decrementing the Focus

1.The instrument sends to the telescope a request to change the respective OPE focus position relative to its current position, specified by:
   (a) the difference (in millimeters) to the current position of the optical element in z-axis, and
   (b) the required OPE.

2.The telescope checks if the positioning information requested in step 1 is within range.

3.In case of success of step 2, the telescope adapts the focus by the requested difference (Delta).

4.The telescope confirms success or failure of the operation to the instrument.

More information about these commands in section 7.21 and 7.45.

6.2.5 Guiding

The telescope provides a non-perfect open loop tracking of the mount. To optimize the tracking for errors due to friction, mechanical imperfection, or flexure, the telescope employs a guiding system.
In order for the guiding control subsystem (GCS) to perform its duties, delivering guiding and wavefront sensing information to the TCS, the GCS controls off-axis AGW units and their cameras for image acquisition.

6.3 Alignment Functionality

The optical axis of the instrument and the telescope have to be aligned, as well as the right and left sides with respect to each other. Such alignment finally happens in the combined focus with the help of stars.
The telescope-to-instrument alignment requires adjustments of the optical elements (OPE). It is possible to move the primary (M1), the secondary (M2) and tertiary (M3) mirrors in different degrees of freedom.
IIF provides several commands to perform these duties, which are described in the next sections.

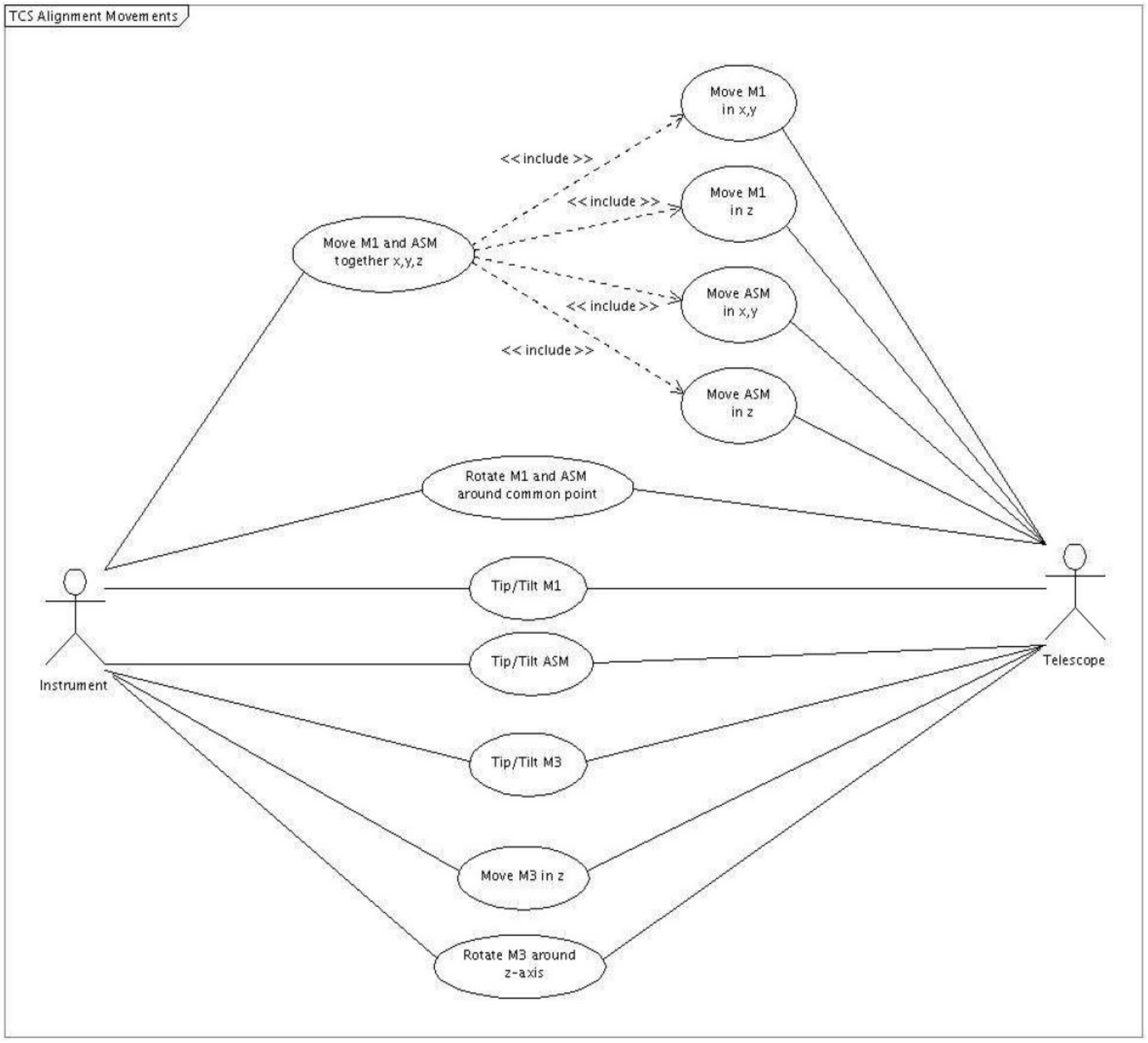


Figure 6.2:Use cases to move the telescope for alignment / adjustments purpose.

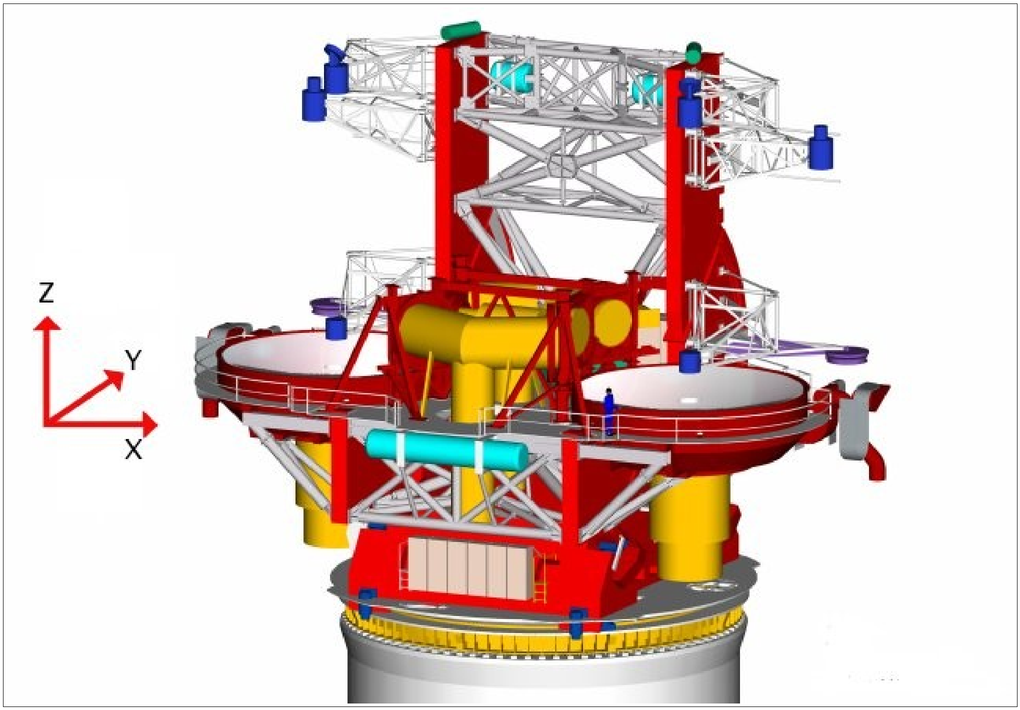


Figure 6.3: Coordinate system as used for all the alignment movements.

6.4 Adaptive Optic System

The Adaptive Optics Subsystem (AOS) is the subsystem of TCS providing all the functions needed for interaction between the LBT Adaptive Optics system and the rest of the telescope, including instruments.

AO System Operating Modes

When operating in support of an observation the AO System will provide four modes of operation:

- FIX-AO. Fixed Mode Operation. It is the seeing limited mode where the Adaptive Secondary mirror holds a fixed "flat" shape defined by a pre-calibrated vector of mirror commands. Depending on the particular kind of observation a specific "flat" vector may be selected.
- TTM-AO. Tip-Tilt Mode Operation. It is an AO mode with only tip-tilt correction performed by the secondary mirror.
- ACE-AO. Auto Configured Adaptive Optics Operation. It is the full AO corrected mode, with AO loop parameters automatically selected by the AO System based on reference source characteristics.

- ICE-AO. Interactively Configured Adaptive Optics Operation. It is a full AO corrected mode where the observer is given the possibility to adjust AO loop parameters.

TCS-IIF provides a set of commands, that correspond exactly to the AOS commands, in order to handle this subsystem. More details in sections 7.1 to 7.11 (AO prefix) and reference [3] as well.

7 Command set at the IIF-TCS

Prior to describing the different IIF commands we will define some important structures and classes the instruments will need[2]. These classes and structures are distributed in several files. Nevertheless, including `IIF.h` will be enough for the instruments to use the libraries.
In order to understand how to include and to use them, please see section 8.
The global definitions and constants showed in this document are defined in the file `IIFGlobals.h`. You will find the complete list in Appendix A : Global definitions .

The Coordinates System enumerated type

This enumerated type defines the different coordinates system[3].

```
coordType
{ COORD_RADEC_SKY, COORD_RADEC_FOCAL, COORD_ALTAZ,
  COORD_FOCAL_PIX, COORD_FOCAL_MM};
```

where `COORD_RADEC_SKY` represents the equatorial coordinates on the sky, `COORD_RADEC_FOCAL` represents the equatorial coordinates on the focal plane, `COORD_ALTAZ` represents the ALT/AZ coordinates, `COORD_FOCAL_PIX` represents the instrument focal plane coordinates in units of pixels and `COORD_FOCAL_MM` the instrument focal plane coordinates in units of millimeter and

The Optical Element enumerated type

The purpose of this enumerated type is to represent the respective optical element.

```
opeType
{ M1, M2, M3, M1M2, M1M3, M2M3, M1M2M3, MOUNT, HEXAPOD, DEFAULT};
```

where `M1` represents the primary mirror, `M2` the secondary mirror, `M3` the tertiary mirror, `M1M2`, `M1M3`, `M2M3` and `M1M2M3` represent the different combinations of all of them. `MOUNT` refers to the telescope mount, `HEXAPOD` represents the hexapod support structure for M2, and `DEFAULT` is defined according to the specific command.

2 As these classes/structures are used as arguments in many of the IIF commands.
3 In the near-term TCS will support COORD_RADEC_SKY, COORD_RADEC_FOCAL, and COORD_FOCAL_PIX.

The Movement enumerated type

This enumerated structure represents the type of movement to use.

```
moveType
{ MV_REL, MV_ABS};
```

where `MV_REL` represents a relative movement to the current position, and `MV_ABS` an absolute one, with respect to the coordinates in the last `PresetTelescope`.

The Rotator enumerated type

Enumerated type to represent the different instrument's rotator modes.

```
rotatorType
{ ROTATOR_PAR, ROTATOR_PSTN, ROTATOR_NATIVE, ROTATOR_GRAV,
  ROTATOR_IDLE };
```

where `ROTATOR_PAR` represents the parallactic angle, which is vertical with respect to the horizon. `ROTATOR_PSTN` relative to the north. `ROTATOR_NATIVE` means to use the rotator's native reference frame. `ROTATOR_GRAV` means to use the gravitational angle, and `ROTATOR_IDLE` for those instruments that will not use the rotator at all.

The Rotation Center enumerated type

Enumerated type to represent the rotation center. See section 7.48 for further details.

```
rotcenterType
{ M1, M2, M3, FS_PRIME, FS_DIRECTGREGORIAN, ROT_CENTER_POS };
```

where `M1` represents the primary mirror, `M2` the secondary mirror, `M3` the tertiary mirror, `FS_PRIME` represents the focal station prime focus, `FS_DIRECTGREGORIAN` represents the focal station direct Gregorian and `ROT_CENTER_POS` represents an user defined point of rotation.

The AO Mode enumerated type

The meaning of this enumerated type is to represent the different AO operational modes.

```
AOmodeType
{ FIX_AO, TTM_AO , ACE_AO , ICE_AO};
```

where FIX_AO represents the Fixed Mode Operation, TTM_AO represents Tip/Tilt operation, ACE_AO Auto Configured Adaptive Optics Operation, and ICE_AO Interactively Configured Adaptive Optics Operation. See section 6.4 for further details.

The Operational Mode enumerated type

Represents the operational modes of the telescope.

```
modeType
{ MODE_STATIC, MODE_TRACK, MODE_GUIDE, MODE_ACTIVE, MODE_ADAPTIVE,
  MODE_INTERFEROMETRIC };
```

MODE_STATIC, slews the telescope to the requested location on sky and stops.
MODE_TRACK sends the telescope to the coordinates and enters open-loop tracking. The guider stage is not moved, and the instrument does not need to supply guide stars.
MOD_GUIDE, the guided mode where both tracking and guiding are in operation. This requires a guide star from the instrument. The guider stage must move to the appropriate location, and the guide star is acquired on the AG, and XY guiding initiated at wherever the star is located on the AG chip.
MODE_ACTIVE, the active mode where tracking, guiding and active optics are engaged. In addition, once XY guiding has started, the AGw sends a pointing offset to the telescope to move the star to the w-unit's entrance aperture. The system goes through the startup sequence and ends closed-loop in both XY guiding and wavefront sensing.
MODE_ADAPTIVE, the adaptive mode where tracking, guiding, and adaptive optics are engaged. It is the big-W that must end up closed-loop at high speed for this mode.
MODE_INTERFEROMETRIC, this mode is ill-defined at this time. It could be used as a trigger for the TCS to set certain defaults in support of interferometric modes.

The Equinox enumerated type

```
equinoxType
{ J2000, ICRS };
```

where `J2000` represents the equatorial coordinate system based on the mean dynamical equator and equinox of the J2000 epoch, and `ICRS` represents the International Celestial Reference System.

The Filter enumerated type

Represents the different filters.

```
filterType
{ U, V, B , R, FILTER_NONE };
```

The Color enumerated type

Represents the different color types of the object.

```
colorType
{ U_B , B_V , H_K , COLORTYPE_NONE };
```

The Proper Motion structure

The proper motion of an astronomical target is the component of the space motion of the source perpendicular to the line of sight.

`ProperMotionType`

- **`double coord1`**
  Purpose: to define the first coordinate of the proper motion for RA.
  Description: double value.
  Unit: marcsec/year
  Range or possible values: [-11000.00, 11000.00]
  Default value: 0.0
- **`double coord2`**
  Purpose: to define the second coordinate of the proper motion for DEC.
  Description: double value that represents the second coordinate of the proper motion.
  Unit: marcsec/year
  Range or possible values: [-11000.00, 11000.00]
  Default value: 0.0

The Apparent Magnitude structure

The apparent magnitude (m) of a star, planet or other celestial body is a measure of its apparent brightness as seen by an observer on Earth.

`MagnitudeType`

- **`double apparentMagnitude`**
  Purpose: to define the apparent brightness of the star as seen by an observer on Earth.
  Description: double value representing the apparent magnitude.
  Unit: unitless
  Range or possible values: `[-27.0 , 40.0]`
  Default value: -
- **`filterType filter`**
  Purpose: to define the wavelength filter for the magnitude.
  Description: enumerated value representing the wavelength filter.
  Unit: unitless
  Range or possible values: `U | V | B | R | FILTER_NONE`
  Default value: -
- **`double color`**
  Purpose: to define the color index of the object.
  Description: double value representing the color index.
  Unit: none
  Range or possible values: `[-2.0, 2.0]` | `COLOR_NONE`
  Default value: -
- **`colorType colorType`**
  Purpose: to define the color type of the object.
  Description: enumerated value representing the color type.
  Unit: unitless
  Range or possible values: `U_B | B_V | H_K | COLORTYPE_NONE`
  Default value: -

The Offset class

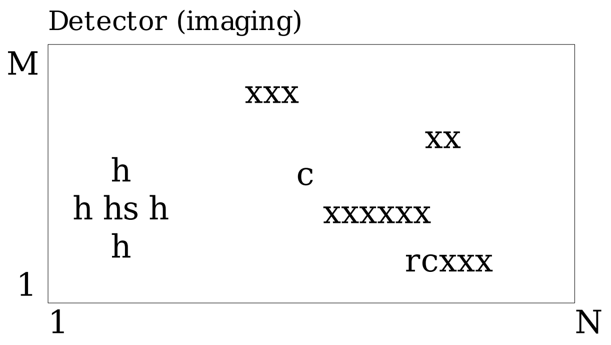
The term “offset” used in this section is really referring to two delta terms. These are relative deltas, referenced to the last coordinates provided. The coordinate system of the offset does not have to be the same as the target.

The constructor of the class has the following parameters:

- **`double coord1`**
  Purpose: to define the offset in RA, ALT or xi.
  Description: double value to represent the first coordinate of the offset.
  Unit: COORD_ RADEC _SKY = radians
  COORD_RADEC_FOCAL = radians
  COORD_ ALTAZ = radians
  COORD_FOCAL_PIX = pixels
  COORD_FOCAL_MM = millimeters
  Range or possible values: COORD_RADEC_SKY = `[-PI , PI]`
  COORD_RADEC_FOCAL = `[-PI , PI]`
  COORD_ALTAZ = `[-PI/2, PI/2]`
  COORD_FOCAL_PIX = `defined by the instrument.`
  COORD_FOCAL_MM = `defined by the instrument.`
  Default value: 0.0
- **`double coord2`**
  Purpose: to define the offset in DEC, AZ or eta.
  Description: double value to represent the second coordinate of the offset.
  Unit: COORD_ RADEC _SKY = radians
  COORD_RADEC_FOCAL = radians
  COORD_ ALTAZ = radians
  COORD_FOCAL_PIX = pixels
  COORD_FOCAL_MM = millimeters
  Range or possible values: COORD_RADEC_SKY = `[-PI , PI]`
  COORD_RADEC_FOCAL = `[-PI , PI]`
  COORD_ALTAZ = `[-PI, PI]`
  COORD_FOCAL_PIX = `defined by the instrument.`
  COORD_FOCAL_MM = `defined by the instrument.`
  Default value: 0.0
- **`coordType system`**
  Purpose: to define the coordinate system of reference for the coordinates.
  Description: enumerated value to represent the coordinate system of reference for the coordinates. See description above.
  Unit: unitless
  Range or possible values: `COORD_RADEC_SKY | COORD_RADEC_FOCAL | COORD_ALTAZ | COORD_FOCAL_MM | COORD_FOCAL_PIX`
  Default value: `COORD_RADEC_FOCAL`

The Hotspot class

The hotspot is given in instrument focal plane coordinates. These are absolute X and Y values specific to the detector. The units are pixels.



c = detector center
rc = rotation center
hs = hotspot(x=3,y=3)
h = dither positions around the hs
x = bad pixels

Note: In this example, the hotspot (hs) is the best position on the detector for the observation.

Figure 7.1: Detector during observation and the concept of Hotspot.

The constructor of the class has the following parameters:

- `double coord1`
  Purpose: to define the first coordinate of the hotspot.
  Description: double value to represent the X coordinate of the hotspot.
  Unit: pixels
  Range or possible values: `defined by the instrument.`
  Default value: -
- `double coord2`
  Purpose: to define the second coordinate of the hotspot.
  Description: double value to represent the Y coordinate of the hotspot.
  Unit: pixels
  Range or possible values: `defined by the instrument.`
  Default value: -

The Position class

Defining positions and all astronomically relevant information about the science target to be observed and the guide-stars, `Position` can exist in different coordinate systems, but all angles must be specified in radians.

The constructor of the class has the following parameters:

- **`double coord1`**
  Purpose: to define the first coordinate of the specified system in RA, ALT or xi.
  Description: double value to represent the first coordinate of position.
  Unit: COORD_ RADEC _SKY = radians
  COORD_ ALTAZ = radians
  Range or possible values: COORD_RADEC_SKY = `[0.0 , 2PI]`
  COORD_ALTAZ = `[0.0 , PI/2]`
  Default value: -
- **`double coord2`**
  Purpose: to define the second coordinate of the specified system in DEC, AZ or eta.
  Description: double value to represent the second coordinate of position.
  Unit: COORD_ RADEC _SKY = radians
  COORD_ ALTAZ = radians
  Range or possible values: COORD_RADEC_SKY = `[-1.0 , 2PI]`
  COORD_ALTAZ = `[-PI/2, 5PI/2]`
  Default value: -
- **`coordType system`**
  Purpose: to define the coordinate system of reference for the coordinates.
  Description: enumerated value to represent the coordinate system of reference for the coordinates. See `coordType` definition above, page 17.
  Unit: unitless
  Range or possible values: `COORD_RADEC_SKY | COORD_ALTAZ`
  Default value: `COORD_RADEC_SKY`
- **`equinoxType equinox`**
  Purpose: to define the value of the equinox for purpose of precession.
  Description: enumerated value representing the reference equinox or system of coordinates.
  Unit: none
  Range or possible values: J2000 | ICRS at this time[4].
  Default value: `J2000`
- **`double epoch`**
  Purpose: to define the value used in conjunction with proper motion values to reference the coordinates to the equinox.
  Description: double value representing the epoch of the given coordinates.
  Unit: year
  Range or possible values: Valid date.
  Default value: `2000.00`
- **`ProperMotionType *propmotion (optional)`**
  Purpose: to specify the potential proper motion of the celestial object described by the

4 Note: “B1950” will be supported in the future.

position.
Description: pointer to a `ProperMotionType struct` (see definitions above).
Unit: See `ProperMotionType` definition above.
Range or possible values: see definition above.
Default value: `NULL` (do not use proper motion).

- **`MagnitudeType *magnitude (optional)`**

  Purpose: to specify the apparent magnitude and filter of the celestial object.
  Description: pointer to a `MagnitudeType struct` (see definitions above).
  Unit: See description above.
  Range or possible values: See description above;
  Default value: `NULL` (do not use the magnitude).

- **`unsigned int wavelength (optional)`**

  Purpose: to compute the object's atmospheric refraction correction.
  Description: integer value specifying the effective wavelength of the target.
  Unit: nanometer.
  Range or possible values: `[300 , 20000]`
  Default value: 500

Position public member functions

```
                  Position (  double coord1, double coord2,
                              coordType system=COORD_RADEC_SKY,
                              equinoxType equinox=J2000,
                              double epoch=2000.00,
                              ProperMotionType *propmotion=NULL,
                              MagnitudeType *mag=NULL,
                              unsigned int wavelength=500)
```
The class's parameterized Constructor.

```
virtual           ~Position ()
```
The class's Destructor.

```
                  Position (const Position &)
```
The class's Constructor.

```
Position          & operator= (const Position &)
bool              operator== (const Position &) const
bool              operator!= (const Position &) const
```
Operators =, ==, !=

```
double            Getcoord1 () const
```
Getter method for the coord1 property of this class.

```
double            Getcoord2 () const
```
Getter method for the coord2 property of this class.

```
coordType         Getcoordsystem () const
```
Getter method for the coordinate system property of this class.

`equinoxType`	`Getequinox () const` Getter method for the equinox property of this class.
`double`	`Getepoch () const` Getter method for the epoch property of this class.
`ProperMotionType`	`* Getpropermotion ()` Getter method for the propermotion property of this class.
`MagnitudeType`	`* Getmagnitude ()` Getter method for the magnitude property of this class.
`unsigned int`	`GetWavelength () const` Getter method for the wavelength property of this class.
`ProperMotionType`	`getProperMotion () const` Getter method for the propermotion property of this class.
`MagnitudeType`	`getMagnitude () const` Getter method for the magnitude property of this class.
`void`	`Setcoord1 (double newVal)` Setter method for the coord1 property of this class.
`void`	`Setcoord2 (double newVal)` Setter method for the coord2 property of this class.
`void`	`Setcoordsystem (coordType newVal)` Setter method for the coordsystem property of this class.
`void`	`Setequinox (equinoxType newVal)` Setter method for the equinox property of this class.
`void`	`Setepoch (double newVal)` Setter method for the epoch property of this class.
`void`	`Setpropermotion (ProperMotionType *newVal)` Setter method for the propermotion property of this class.
`void`	`Setmagnitude (MagnitudeType *newVal)` Setter method for the magnitude property of this class.
`void`	`Setwavelength (unsigned int newVal)` Setter method for the wavelength property of this class.
`bool`	`hasMagnitude ()` Return true if the magnitude was specified. Otherwise false.
`bool`	`hasProperMotion ()` Return true if the propermotion was specified. Otherwise false.
`bool`	`hasWavelength ()` Return true if the wavelength was specified.

The MultiDDEntry class

This class is used to create a list of data dictionary entries that will be requested from the TCS. Also, can be used to set a block of instrument specific entries into the data dictionary.
The instrument control software (ICS) needs a `MultiDDEntry` object in order to use the commands `GetMultiParameter` and `SetMultiParameter` as well (described in section 7.16 and 7.41).

```
                        MultiDDEntry();
                        MultiDDEntry(queue<string> entries);
                        MultiDDEntry(queue<string> entries, queue<string> values);
                        virtual ~MultiDDEntry();
                        MultiDDEntry& operator= (MultiDDEntry &);
```
copy-constructor and assignment-operator

```
void                    SetMultiEntries ( queue<string> entries);
void                    SetMultiValues  ( queue<string> values);
void                    SetMultiEntries ( queue<string> entries, queue<string>
                                          values);
```
Setter method for the Multiple values/entries property of the instance.

```
vector<string>          GetMultiEntries ();
vector<string>          GetMultiValues ();
```
Getter method for the Multiple values/entries property of the instance.

```
void                    PushEntry(string entry);
void                    PushEntry(string entry, string value);
void                    PushValue(string value);
```
Push methods, to push a new value or a new entry into the queue

```
void                    PopEntry();
void                    PopValue();
```
Pop methods, to remove an entry or value from the queue

```
string                  FrontEntry() const;
string                  FrontValue() const;
```
Front methods, return the front entry or value of the queue

```
bool                    empty() const;
bool                    emptyEntry() const;
bool                    emptyValue() const;
```
empty methods, return true if the queue is empty, otherwise false.

```
unsigned int            size() const;
unsigned int            sizeEntry() const;
unsigned int            sizeValue() const;
```
size methods, return the number of entries in the queue.

`void`	`Clear();` clear method, cleans the queues. The number of entries in the queues after Clear() is 0.

Wavefront class

This class contains information about the wave front correction to account for the optical aberration. The ICS uses this class in order to issue the `SendWavefront()` command. See more details in section 7.40

Public Member Functions

	`WaveFront (double coeffs[POLY_ORDER])` The class's parametized Constructor.
	`WaveFront ()` The class's Constructor.
`virtual`	`~WaveFront ()` The class's Destructor.
	`WaveFront (const WaveFront &)`
`WaveFront &`	`operator= (const WaveFront &)`
`bool`	`operator== (const WaveFront &) const`
`bool`	`operator!= (const WaveFront &) const`
`double *`	`GetCoefficients (double dest[POLY_ORDER]) const` Getter method for the Coefficients property of the instance.
`double`	`GetCoefficient (unsigned int order)` Returning a single coefficient of the instance.
`void`	`SetCoefficients (double coeffs[POLY_ORDER])` Setter method for the Coefficients property of the instance.
`void`	`SetCoefficient (double coeff, int order)` Setting a single coefficient of the instance to a new value.
`vector<double>`	`getCoefficients ()`
`double`	`getCoefficient (unsigned int order)`
`void`	`setCoefficients (vector< double > coeffs)`
`void`	`setCoefficient (double coeff, int order)`

The Result class

This class defines the results that are returned to the Instrument when commands are sent to the telescope (to the CSQ).
Results are immediately returned by the respective IIF methods. This means that the operation inside the TCS initiated by the IIF command may not have finished by the time the IIF method call returns.
The status of an initiated operation can be queried using the command handle inside the `Result` instance, as we describe in section 8.4.
However it is possible to halt the program thread's execution until the TCS has finished the command processing ("sleep wait", the CPU is not occupied by the wait). To achieve this, the IIF user simply calls the block() method on the command handle that is associated with the requested command.

Result attributes

- **`CSQHandle* commandHandle`**
  Purpose: to define the command handle associated with the `Result` instance.
  Description: reference to a `CSQHandle` object.
  Unit: unitless.
  Range or possible values: -
  Default value: `NULL`.
- **`int  resCode`**
  Purpose: to define the result code, to indicate the success of sending the command to the TCS.
  Description: integer value representing the result code.
  Unit: unitless.
  Range or possible values: `RESCODE_OK | RESCODE_FAIL`
  Default value: `RESCODE_FAIL`
- **`string resString`**
  Purpose: to define the result string to indicate the success of sending the command to the TCS.
  Description: string variable defining the result string.
  Unit: unitless.
  Range or possible values: `RESSTRING_OK | RESSTRING_FAIL`
  Default value: `RESSTRING_FAIL`.
- **`double ttc`**
  Purpose: to define the expected time to complete the operation in the TCS.
  Description: double value representing the "time to complete" attribute.
  Unit: [TBD]
  Range or possible values: `[TBD]`
  Default value: `-1.0 | [TBD]`

Result public member functions

	`Result (CSQHandle *handle=NULL, int resCode=RESCODE_FAIL, string resString=RESSTRING_FAIL, double ttc=-1.0)` The class's parametized Constructor.
`CSQHandle`	`*GetcommandHandle ()` Getter method for the commandHandle property of this class.
`int`	`GetresultCode ()` Getter method for the resultCode property of this class.
`string`	`GetresultString ()` Getter method for the resultString property of this class.
`double`	`GettimeToComplete ()` Getter method for the ttc (time to complete) property of this class.

The Status class

Defines the requested status information for the processing request (specified by a handle), and (in case of completion of the command) the structure returned by the processing request.

StatusInfo public member functions

`string`	`GetCommandResult();` Retrieves the (string) result of the CSQ command associated with the StatusInfo instance, which is `STATE_NORESULT`, in case the command handle is not assigned.
`string`	`GetCommandStatus();` Retrieves the status string of the CSQ command associated with the StatusInfo instance, or a string indicating that there is no associated command.
`double`	`GetEstimateTTC();` Retrieves the latest estimate for the time to completion of the CSQ command associated with the StatusInfo instance, or -1.0 if there is no estimate for the time to completion (most TCS commands do not support this feature).
`void`	`Block();` Method responsible for blocking the command till its execution is completed by the TCS.

StatusInfo definitions

`STATE_RUNNING "RUNNING"`
Definition of the string indicating that a command is still being executed by the CSQ.

`STATE_CANCELED "CANCELED"`
Definition of the string indicating that a CSQ command has been previously canceled.

`STATE_SUCCESS "SUCCESS"`
Definition of the string indicating that the CSQ has successfully finished processing the associated command.

`STATE_FAILURE "FAILED"`
Definition of the string indicating that an error has occurred during execution of the associated command by the CSQ.

`STATE_WRONGHANDLE "NO ASSOCIATED COMMAND HANDLE"`
Definition of the string indicating there is no associated command for the specified command handle.

`STATE_NORESULT "NO COMMAND RESULT"`
Definition of the string indicating there is no valid result string that can be returned as the command's result.

7.1 AOPreset

Description

The `AOPreset` command is issued in a AOS observation service status in order to prepare the AO system for an observation in adaptive mode. More details in reference [3], section 2.1.3

Syntax

```
Result * AOPreset(  AOmodeType AOMODE,
                    wfsType WFS,
                    float SOCOORD1, float SOCOORD2,
                    float ROCOORD1, float ROCOORD2,
                    float ROTANGLE, float MAG,
                    float COLOR, float WANGLE,
                    const char * SIDE )
```

Attributes

- **`AOmodeType AOMODE`(in)**
  Purpose: to specify the AO mode, either TTM-AO, ACE-AO or ICE-AO
  Description: enumerated variable to represent the AO mode.
  Unit: unitless
  Range or possible values: `FIX_AO | TTM_AO | ACE_AO | ICE_AO`
  Default value: -
- **`wfsType WFS`(in)**
  Purpose: to specify the source of WFS data.
  Description: enumerated variable. TBD at the moment.
  Unit: unitless
  Range or possible values: `TBD`
  Default value: -
- **`float SOCOORD1, SOCOORD2` (in)**
  Purpose: to specify the position of the scientific object in focal plane coordinates.
  Description: double value.
  Unit: mm
  Range or possible values: `[TBD.00, TBD.00]`
  Default value: -
- **`float ROCOORD1, ROCOORD2` (in)**
  Purpose: to specify the position of the reference object in focal plane coordinates.
  Description: double value
  Unit: mm
  Range or possible values: `[-TBD.00 , TBD.00]`
  Default value: -
- **`float ROTANGLE` (in)**

Purpose: to specify the angular position of the rotator.
Description: double value
Unit: radians
Range or possible values: `[-TBD.00 , TBD.00]`
Default value: -

- **`float MAG(in)`**

  Purpose: to specify the magnitude of the reference star.
  Description: double value
  Unit: unitless
  Range or possible values: `[-TBD , TBD]`
  Default value: -

- **`float WANGLE(in)`**

  Purpose: to specify the wind direction.
  Description: double value
  Unit: radians
  Range or possible values: `[-TBD , TBD]`
  Default value: -

- **`const char * SIDE (in)`**

  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**

  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RRESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.

After execution

- AOS will wait for a change of the corresponding AO variable (TBD at the moment) reflected also in the data dictionary.
- AOS perform all set up operation needed, except acquisition of the reference object, which is performed with the subsequent `AOAcquireRef` command.

- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->AOPreset(    ACE_AO, 1.0, 1.2, 2.0, 1.67,
                              3.15, -3, 20, SIDE_LEFT );
...
```

7.2 AOAcquireRef

Description

`AOAcquireRef`, issued after a `AOPreset`, requests the AOS to proceed into the reference object acquisition, in order to find the reference star within the field of view of the technical viewer.
More details in reference [3], section 2.1.4

Syntax

```
Result * AOAcquire( const char * SIDE )
```

Attributes

- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- The instrument must check that the previous `PresetAO` command has been successfully completed, the telescope has reached the pointing position and the guiding system is operating.

After execution

- AOS will issue either a `AOStart` or a `AORefine` command.
- The command returns the computed AO loop parameters, that can be

retrieved via `CommandReturn`'s `getResultCount()` and `GetResultDescription(int n)` methods.
This parameters are TBD. More information in reference [3], section 2.1.4.

Examples

```
....
aResult = anIIF->AOAcquireRef(SIDE_RIGHT);
....
```

7.3 AORefine

Description

The `AORefine` command is used to support the ICE-AO operating mode. It maybe used to request the AOS to modify the value of some loop parameter before closing the loop.
More details in reference [3], section 2.1.5

Syntax

```
Result * AORefine(  int NMODES, float ITIME,
                    int NBINS, float TTMOD,
                    const char * F1SPEC,
                    const char * F2SPEC,
                    const char * SIDE)
```

Attributes

- **`int NMODES(in)`**
  Purpose: to specify the number of corrected modes.
  Description: integer variable
  Unit: unitless
  Range or possible values: `TBD`
  Default value: TBD
- **`float ITIME (in)`**
  Purpose: to specify the CCD integration time.
  Description: float value.
  Unit: s
  Range or possible values: `[TBD, TBD]`
  Default value: 0
- **`int NBINS(in)`**
  Purpose: to specify the CCD binning
  Description: integer value
  Unit:
  Range or possible values: `[-TBD , TBD]`
  Default value: 0
- **`float TTMOD (in)`**
  Purpose: to specify the Tip-Tilt internal mirror modulation.
  Description: float value
  Unit: TBD
  Range or possible values: `[-TBD, TBD]`
  Default value: 0.00
- **`const char * F1SPEC (in)`**
  Purpose: to specify the selected position of filter wheel number 1.
  Description: string value.

Unit: unitless
Range or possible values: `TBD`
Default value: `TBD`

- **`const char * F2SPEC` (in)**
  Purpose: to specify the selected position of filter wheel number 2.
  Description: string value.
  Unit: unitless
  Range or possible values: `TBD`
  Default value: `TBD`
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, "left", "right" or "both".
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- `AOPreset` must be done, `AOAcquireRef` must be done.

After execution

- AOS will wait for a change of the corresponding AO variable, reflected also in the data dictionary.
- AOS perform all set up operation needed, except acquisition of the reference object, which is performed with the subsequent `AOAcquireRef` command.
- The command returns the computed AO loop parameters, that can be retrieved via `CommandReturn`'s `getResultCount()` and `GetResultDescription(int n)` methods.
  This parameters are TBD. More information in reference [3], section 2.1.5.

Examples

```
....
aResult = anIIF->AORefine(1, 1.123, 1, .20, TBD, TBD, SIDE_LEFT );
....
```

7.4 AOStart

Description

This command is used to request the closing of the AO loop.
More details in reference [3], section 2.1.6

Syntax

```
Result * AOStart(const char * SIDE )
```

Attributes

- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- The instrument must check that the previous `AOPreset` command has been successfully completed, the telescope has reached the pointing position and the guiding system is operating.
- AOS must be fully ready to close the loop.

After execution

- Loop closed.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was

successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOStart(SIDE_LEFT);
....
```

7.5 AOOffsetXY

Description

`AOOffsetXY` is issued in AOS observation service status in order to offset the pointing of the AOS. This command is meaningful only in closed loop mode. More details in reference [3], section 2.1.7

Syntax

```
Result * AOOffsetXY(     float XOFF, float YOFF,
                         const char * SIDE )
```

Attributes

- **`float XOFF` (in)**
  Purpose: to specify the requested position offset in X.
  Description: float value.
  Unit: mm
  Range or possible values: `[TBD, TBD]`
  Default value: 0
- **`float YOFF` (in)**
  Purpose: to specify the requested position offset in Y.
  Description: float value
  Unit: TBD
  Range or possible values: `[-TBD, TBD]`
  Default value: 0.00
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- AOS is operating in closed loop mode.

After execution

- Secondary mirror follows the offset, so this operation results in an offset of the field on the scientific camera.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOOffsetXY(1.255, -0.815, SIDE_BOTH);
....
```

7.6 AOOffsetZ

Description

This command is issued in AOS observation service status in order to offset the focus of the AOS. It is meaningful only in closed loop mode. More details in reference [3], section 2.1.8

Syntax

```
Result * AOOffsetZ( float ZOFF, const char * SIDE )
```

Attributes

- **`float ZOFF` (in)**
  Purpose: to specify the requested focus offset.
  Description: float value.
  Unit: mm
  Range or possible values: `[TBD, TBD]`
  Default value: 0
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- AOS is operating in closed loop mode.

After execution

- Secondary mirror follows the offset, so this operation results in a change of focus on the scientific camera.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOOffsetZ(1.955, SIDE_LEFT);
...
```

7.7 AOCorrectModes

Description

This command is used in AOS observation service status to apply a modal correction on the mirror shape. More details in reference [3], section 2.1.9

Syntax

```
Result * AOCorrectModes(     float DELTAM[762],
                             const char * SIDE )
```

Attributes

- **float DELTAM[762] (in)**
  Purpose: to specify the modes correction vector.
  Description: float vector with 762 elements.
  Unit: TBD
  Range or possible values: `[TBD, TBD]`
  Default value: 0
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.

<u>After execution</u>

- AOS sends the related request message to AO-Sup. The mirror change the shape.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

<u>Examples</u>

```
....
float DeltaM[762];
....
aResult = anIIF->AOCorrectModes(DeltaM, SIDE_LEFT);
....
```

7.8 AOStop

Description

This command is used to stop the current operation. After this command any setting defined by a previous `AOPreset` is canceled. More details in reference [3], section 2.1.10

Syntax

```
Result * AOStop(    const char * MSG,
                    const char * SIDE )
```

Attributes

- **`const char * MSG` (in)**
  Purpose: to specify the reason for stopping.
  Description: string value to represent the message for stopping.
  Unit: unitless
  Range or possible values: -
  Default value: -
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- AOS is in closed loop mode.

After execution

- AOS send the request message to AO-Sup, which properly updates the related variable in the data dictionary.
- AOS is in open loop mode.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOStop(“I'm tired”, SIDE_LEFT);
...
```

7.9 AOPause

Description

This command is issued to temporarily suspend the current AO operation. More details in reference [3], section 2.1.11

Syntax

```
Result * AOPause( const char * SIDE )
```

Attributes

- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- AOS is in closed loop mode.

After execution

- AOS sends the request message to AO-Sup, which properly updates the related status in the data dictionary.
- AOS is in pause mode.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOPause(SIDE_BOTH);
....
```

7.10 AOResume

Description

This command resumes suspended operation after a AOPause.
More details in reference [3], section 2.1.12

Syntax

```
Result * AOResume(  const char * SIDE )
```

Attributes

- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: Result structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = RESCODE_FAIL: an error occurred.
  RESCODE_OK: Command accepted.

Preconditions

- The instrument must be authorized.
- A previous AOPause must be issued.
- AOS is in pause mode.

After execution

- AOS will send the request message to AO-Sup. The it will properly update the related variable in the data dictionary.
- The loop is closed.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOResume(SIDE_LEFT);
....
```

7.11 AOUserPanic

Description

This command is issued whenever some TCS subsystem, including an instrument, detects any dangerous condition, and decides to perform a fast shutdown. More details in reference [3], section 2.1.14

Syntax

```
Result * AOUserPanic(    const char * MSG
                         const char * SIDE )
```

Attributes

- **const char * MSG (in)**
  Purpose: to specify the reason for panic.
  Description: string value to represent the message for panic.
  Unit: unitless
  Range or possible values: -
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.

After execution

- AOS sends the corresponding request message to AO-Sup.
- The connection is immediately closed.
- AOS shutdown ASAP and puts the secondary mirror in a safe mode[5] .
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->AOUserPanic(“Ahhhh What I did !!!!”, SIDE_LEFT);
....
```

5 After the acknowledge no other interaction will be possible between AOS and AO-Sup.

7.12 Authorize

Description

This command allows the instrument to authorize itself to control one or both sides of the telescope. For multi-threaded ICS, TCS allows multiple authorizations, maintaining an internal count for each side.
For further information, please see section 6.1.1, 6.1.2, and section 8.2 as well.

Syntax

```
bool Authorize()
```

Return values

- **bool (returned)**
  Purpose: to evaluate if the instrument was authorized to control the specified side. See section 8.2 for further details.
  Description: bool value.
  Unit: unitless
  Range or possible values: `true` authorization was granted by the TCS for the specified instrument and side on the IIF instance.
  `false` The authorization failed, most likely another instrument already controls one or all of the telescope sides specified in the failing IIF instance's constructor.

Preconditions

- The instruments must create a new instance of the IIF class, identifying themselves by two pieces of information, their unique name (i.e. “MODS1” or “LUCIFER2”) and their focal station. (See section 8.2 for further details).

After execution

- Instruments are ready to initiate command processing inside the TCS via the IIF.

Examples

```
....
IIF * anIIF;
```

```
try
{
   anIIF = new IIF("prime left", "LBCBLUE"); //pretend to the left-side,
                                             //prime focus blue-channel LBC
}
catch (int e)
{
  cout << "The IIF failed to initialize";
  exit(-1);
}

//Permission for commanding the respective telescope sides is requested, and
//-upon failure to achieve authorization- the program is exited
if (!anIIF->Authorize())
{
  cout << "Authorize() was not successful ... terminating test program.";
  exit(-1);
}
cout << "Authorize() successful" << endl;
```

7.13 CancelCommand

Description

`CancelCommand` is used to cancel an already issued command[6].

Syntax

```
Result * CancelCommand( CSQHandle * HANDLE )
```

Attributes

- **`CSQHandle * HANDLE` (in)**
  Purpose: to specify the `CSQHandle` pointer of the IIF command to be canceled.
  Description: Unique `CSQHandle` object.
  Unit: unitless
  Range or possible values: -
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error or `RESCODE_OK` The command was accepted.

Preconditions

- The instruments must be previously authorized.
- An IIF command must have been issued and accepted by the CSQ.

After execution

- The execution of the command specified by the CSQHandle is canceled.

Examples

```
...
aResult = anIIF->CancelCommand(aResStandby->GetcommandHandle());
```

6 At the moment, most commands do NOT support cancel in the TCS.

7.14 Deauthorize

Description

The `Deauthorize` command is issued to finish the observation. It un-does one 'authorize' request. The TCS maintains a count of 'authorize' requests for each side, and this command decrements the count. When the count reaches zero the instrument is no longer authorized on that side.

Syntax

```
Result * Deauthorize(const char * SIDE)
```

Attributes

- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error
  `RESCODE_OK` The command was accepted.

Preconditions

- None

After execution

- The CSQ count for the specified side is decremented. When the count is zero, this side transits into a “idle” state and another instrument can take control of it.
- The command returns a string “`DEREGISTERED`” in case that the count is zero, Otherwise returns “`AUTHORIZED`”. The result can be retrieved via `StatusInfo`'s `getCommandResult.` Please find more details in

section 8.4, in particular table 8.3.

Examples

```
...
aResult = anIIF->Deauthorize(SIDE_RIGHT);
...
```

7.15 GetCommandStatus

Description

`GetCommandStatus` is used to inquire the status of a previously issued IIF command.

Syntax

```
StatusInfo * GetCommandStatus(CSQHandle * HANDLE)
```

Attributes

- **`CSQHandle* HANDLE` (in)**
  Purpose: to specify the processing request whose status is being inquired.
  Description: Unique `CSQHandle` object.
  Unit: unitless
  Range or possible values: -
  Default value: -

Return values

- **`StatusInfo * RESULT` (returned)**
  Purpose: to specify the requested status information for the processing request specified by HANDLE.
  Description: `StatusInfo` structure received from the TCS with the status of the command. See `StatusInfo` class above.
  Unit: unitless
  Range or possible values: See `StatusInfo` above.

Preconditions

- An IIF command must have been issued and accepted by the CSQ.

After execution

- The instrument receives a structure containing the requested status information for the processing request specified by HANDLE, and (in case of completion) the structure returned by the processing request; By checking the returned STATUSINFO's command status (see section 8.4 for further information), the Instrument can determine if a command has finished processing inside the TCS.

Examples

```
...
// Issue the Standby command
aResult = anIIF->Standby(3, SIDE_LEFT);

if (aResult->GetresultCode() != RESCODE_FAIL) {
     aResult->GetcommandHandle()->block(); /* wait for TCS */

     StatusInfo* statusInf =
          anIIF->GetCommandStatus(aResult->GetcommandHandle());
...
```

7.16 GetMultiParameter

Description

This command is used to read a block of entries from the data dictionary in one shot.

Syntax

```
Result * GetMultiParameter(MultiDDEntry MULTIENTRIES)
```

Attributes

- **`MultiDDEntry MULTIENTRIES` (in)**
  Purpose: to define the list of parameters to be read by the command from the data dictionary.
  Description: `MultiDDEntry` structure to represent the list of data dictionary entries. See `MultiDDEntry` class above.
  Unit: unitless
  Range or possible values: `Valid Data Dictionary entries.`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error or `RESCODE_OK` The command was accepted.

Preconditions

- A `MultiDDEntry` object must be populated with valid data dictionary entries.

After execution

- The command returns a list with the values of the data dictionary entries on the TCS side, that can be retrieved via `CommandReturn`'s `getResultCount()` and `GetResultDescription(int n)` methods. More details in section 8.4
- If the command fails, just the failing entries are returned along with

the failure reason.

Examples

```
...
MultiDDEntry DDEntries;
DDEntries.PushEntry(“pmc.side[0].elevationAngle”);
DDEntries.PushEntry(“pmc.side[1].elevationAngle”);
aResult = anIIF->GetMultiParameter(DDEntries);
...
```

7.17 GetParameter

Description

`GetParameter` requests telescope status by querying a valid data dictionary entry.

Syntax

```
Result * GetParameter( const char * DDENTRY )
```

Attributes

- **`const char * DDENTRY` (in)**
  Purpose: to specify the data dictionary entry to be requested.
  Description: string value.
  Unit: unitless
  Range or possible values: `Valid Data Dictionary entry.`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error or
  `RESCODE_OK` The command was accepted.

Preconditions

- The data dictionary entry must be a valid one.

After execution

- The command returns the value of the data dictionary entry on the TCS side, that can be retrieved via `CommandReturn`'s `getResultCount()` and `GetResultDescription(int n)` methods.
- If the command fails, returns the failure reason.

Examples

```
...
aResult = anIIF->GetParameter("pmc.side[0].elevationAngle");
...
```

7.18 GetRotatorTrajectory

Description

This command inquires the rotator trajectory for the near future. This function has been designed specifically for LBC use in that it accommodates constraints in their instrument. It is not intended for use by any other instrument.

Syntax

```
Result * GetRotatorTrajectory(     double NOOFSECS,
                                   double INTERVAL,
                                   double STARTTIME,
                                   const char * SIDE )
```

Attributes

- **double NOOFSEC (in)**
  Purpose: to specify the number of seconds for the desired 'look-ahead' for the trajectory.
  Description: double value.
  Unit: seconds
  Range or possible values: [TBD]
  Default value: -
- **double INTERVAL (in)**
  Purpose: to specify the number of seconds for the desired time interval between returned trajectory points.
  Description: double value.
  Unit: seconds
  Range or possible values: [TBD]
  Default value: -
- **double STARTTIME(in)**
  Purpose: to specify the desired start time for prediction as a Modified Julian Date.
  Description: double value.
  Unit: days
  Range or possible values: [TBD]
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**

  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error or `RESCODE_OK` The command was accepted.

Preconditions

- The instrument must be authorized.
- The telescope is observing.

After execution

- The command returns an array of (t, theta) pairs, representing time in days in days (JD, double - please note the time is intentionally returned as JD) and rotation angle (radians, double) that can be retrieved via `CommandReturn`'s `getResultCount()` and `GetResultDescription(int n)` methods. More details in section 8.4.

Examples

```
...
aResult = anIIF->GetRotatorTrajectory(50.0,1.0,0.0, SIDE_LEFT);
...
```

7.19 LogEvent

Description

This command is issued to log a message inside the TCS's logging system.

Syntax

```
Result * LogEvent(  const char * EVENTNAME,
                    const char * EVENTDESCRIPTION )
```

Attributes

- **const char * EVENTNAME (in)**
  Purpose: to specify an event name to be logged in the TCS's logging system.
  Description: string to define the event name.
  Unit: unitless
  Range or possible values: -
  Default value: -
- **const char * EVENTDESCRIPTION (in)**
  Purpose: to specify a descriptive text that the instrument desires to be logged.
  Description: string to define the event description.
  Unit: unitless
  Range or possible values: -
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The event must be previously defined in the data dictionary.

After execution

- The telescope processes the logging request. The event name is recorded in the format “`CSQ.<InstrumentID>.<EventName>` and the event description in the format `“CSQ “ + <InstrumentID> + “ “ +`

`<EventDescription>`.

- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->LogEvent("My_First_log”, ”Hello World!");
...
```

7.20 Move

Description

This command has the same functionality that `MoveXY`, `TipTilt`, `StepFocus` and `MoveFocus`. See the respective section for further details.

Syntax

```
Result * Move (double X, double Y, double Z,
              double RX, double RY, double RZ,
              int D_FLAG, moveType MOVE_TYPE,
              opeType OPE, double TIME,
              const char * SIDE)
```

Attributes

- **`double X,Y,Z` (in)**
  Purpose: to specify the naked focal plane movements. MOVE_TYPE will determine if they are absolute or relative values. For OPE M3, X and Y are ignored, and Z is M3 piston. More information about the coordinate system in Figure 6.3
  Description: double value.
  Unit: millimeters
  Range or possible values: `Depends on OPE and current position.`
  Default value: -
- **`double RX,RY,RZ` (in)**
  Purpose: to specify the the naked focal plane rotation. MOVE_TYPE will determine if they are absolute or relative values. For OPE M3, RX is M3 Tip, RY is M3 Tilt.
  Description: double value .
  Unit: micro radians
  Range or possible values: `Depends on OPE and current position.`
  Default value: -
- **`int D_FLAG` (in)**
  Purpose: 6 bits with a bit for each of the preceding 6 variables. Bit 0 enables X, bit 1 enables Y, Bit 2 enables Z, and so on.
  Description: integer to represent the mask or flag, needed for absolute movements.
  Unit: unitless
  Range or possible values: [00000000, 00111111] bin.
  Default value:
- **`moveType MOVE_TYPE` (in)**
  Purpose: to determine if the movements are absolutes or relatives.
  Description: value to represent the movement type, `MV_REL` or `MV_ABS`. See `moveType` definition in section 7.
  Unit: unitless
  Range or possible values: `MV_REL | MV_ABS`

Default value: -

- **`opeType` `OPE` (in)**
  Purpose: to determine which optical element(s) to move.
  Description: value to represent the PSF mode, `DEFAULT`: TCS decides the OPE to move, `M1` the primary, `M2` the secondary and so on. See `opeType` definition above, page 17.
  Unit: unitless
  Range or possible values: `DEFAULT | M1 | M2 | M3 | M1M2 | M1M3 | M2M3 | M1M2M3`
  Default value: `DEFAULT`
- **`int` `TIME` (in)**
  Purpose: to specify the lookahead time in seconds for the collimation correction.
  Description: integer value.
  Unit: seconds
  Range or possible values: TBD
  Default value: -
- **`const char *` `SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result *` `RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The OPE can be moved.

After execution

- OPE is on a new position.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->Move(  1.42, 1.03, 1.0, 1.0,0.4, 0.4, 255,
                        MV_REL, M1, 0, SIDE_LEFT );
...
```

7.21 MoveFocus

Description

`MoveFocus` moves an optical element to a new absolute position z to adjust or to define a new focus position (more information about the coordinate system in figure 6.3). Any focus move in closed-loop mode must be accompanied by the corresponding offset of the w/W stage along the focus direction.

Syntax

```
Result * MoveFocus( double ABSPOS, opeType OPE,
                    const char * SIDE )
```

Attributes

- **`double ABSPOS` (in)**
  Purpose: to specify the new absolute position Z of the respective optical element.
  Description: double value. For OPE M3 this is M3 piston.
  Unit: millimeter
  Range or possible values: `Depends on OPE.`
  Default value: -
- **`opeType OPE` (in)**
  Purpose: to specify the optical element which should move in Z direction. The OPE could be the primary, the secondary or the tertiary mirror.
  Description: enumerated variable to define the optical element (OPE). See description above.
  Unit: unitless
  Range or possible values: `M1 | M2 | M3 | M1M2`
  Default value: -
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless

Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
`RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The OPE can be moved.
- The telescope is tracking and/or guiding and/or controlling AO.

After execution

- OPE is on a new position.
- Telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->MoveFocus(1.42, M1, SIDE_LEFT);
...
```

7.22 MoveXY

Description

The `MoveXY` command moves an OPE in X or Y direction, relative to the current position (more information about the coordinate system in figure 6.3). In closed-loop mode with w/W, this may require offsetting the relevant stage/s as well to maintain lock on the specified reference star.

Syntax

```
Result * MoveXY(    double XMOTION, double YMOTION,
                    opeType OPE, const char * SIDE )
```

Attributes

- **double XMOTION (in)**
  Purpose: to specify the motion along the x axis, relative to the current position with a precision of micro meters.
  Description: double value to represent the motion along x axis.
  Unit: millimeter
  Range or possible values: `Depends on OPE.`
  Default value: 0
- **double YMOTION (in)**
  Purpose: to specify the motion along the y axis, relative to the current position with a precision of micro meters.
  Description: double value to represent the motion along y axis.
  Unit: millimeter
  Range or possible values: `Depends on OPE.`
  Default value: 0
- **opeType OPE (in)**
  Purpose:to define the optical element this command applies to. The OPE could be the primary or the secondary mirror.
  Description: enumerated variable to define the optical element (OPE). See description above.
  Unit: unitless
  Range or possible values: `M1 | M2`
  Default value: `M1`
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The OPE can be moved.
- The telescope is tracking and/or guiding and/or controlling AO.

After execution

- OPE is on a new position.
- Telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command's result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->MoveXY(1.000, -0.810 , M1, SIDE_LEFT);
...
```

7.23 MoveXYZ

Description

The `MoveXYZ` command moves the primary and secondary together in X, Y and Z direction. The movement is relative and not synchronized between the OPE. (More information about the coordinate system in figure 6.3)

Syntax

```
Result * MoveXYZ(   double RELX, double RELY, double RELZ,
                    const char * SIDE )
```

Attributes

- **`double RELX` (in)**
  Purpose: to specify the relative movement in X direction.
  Description: double value.
  Unit: millimeters
  Range or possible values: `Depends on OPE and current position.`
  Default value: 0
- **`double RELY` (in)**
  Purpose: to specify the relative movement in Y direction.
  Description: double value.
  Unit: millimeters
  Range or possible values: `Depends on OPE and current position.`
  Default value: 0
- **`double RELZ` (in)**
  Purpose: to specify the relative movement in Z direction.
  Description: double value.
  Unit: millimeters
  Range or possible values: `Depends on OPE and current position.`
  Default value: 0
- **`const char * SIDE` (in)**
  Purpose: to define the side this command applies to.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to

query for further information.

Unit: unitless

Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
`RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The OPE can be moved.
- The telescope is tracking and/or guiding and/or controlling AO.

After execution

- OPE is on a new position.
- Telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->MoveXYZ(1.000, -0.810 , 1.22, M1, SIDE_LEFT);
...
```

7.24 OffsetGuiding

Description

`OffsetGuiding` moves the telescope a small distance without changing the value of the pointing coordinates. Note that this command does not result in the movement of the w/W stage. At the moment, this command is only needed by LBC.

Syntax

```
Result * OffsetGuiding(  double ROTANGLE, Offset * OFFSET,
                         const char * SIDE )
```

Attributes

- **`double ROTANGLE` (in)**
  Purpose: to specify the value in radians of the optional relative rotation angle.
  Description: double value.
  Unit: radian
  Range or possible values: `[-2PI, 2PI]`
  Default value: `0`
- **`Offset * OFFSET` (in)**
  Purpose: to specify the desired guiding offset from the current position to the science object.
  Description: `Offset` object to define the guiding offset.
  Unit: see `Offset` class above.
  Range or possible values: see `Offset` class in section 7.
  Default value: -
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The telescope can be moved.
- The telescope is tracking and/or guiding and/or controlling AO.

After execution

- The telescope is on a new position, tracking, guiding and controlling the AOS.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
...
offset = new Offset(2.354, 1.1224, COORD_FOCAL_PIX );
//call OffsetGuiding()
aResult = anIIF->OffsetGuiding(-1.1, offset, SIDE_LEFT);
...
```

7.25 OffsetPointing

Description

`OffsetPointing` moves the telescope a small distance, setting the value of the telescope pointing coordinates to the new position. This command uses the existing target information and the setup declared in the last `PresetTelescope` command provided for the given telescope side.

Syntax

```
Result * OffsetPointing( double ROTANGLE,
                         Offset * OFFSET,
                         moveType MOVE_TYPE,
                         opeType OPE,
                         bool NEW_POSITION,
                         const char * SIDE )
```

Attributes

- **`double` `ROTANGLE` (in)**
  Purpose: to specify the value in radians of the rotation angle.
  Description: double value.
  Unit: radian
  Range or possible values: `[-2PI, 2PI]`
  Default value: -
- **`Offset *` `OFFSET` (in)**
  Purpose: to specify the desired offset from the position specified by the MOVE_TYPE parameter.
  Description: `Offset` object.
  Unit: see `Offset` class in section 7.
  Range or possible values: see `Offset` class above.
  Default value: -
- **`moveType` `MOVE_TYPE` (in)**
  Purpose: to determine if the movements are absolutes or relatives.
  Description: value to represent the movement type, `MV_REL` or `MV_ABS`. See `moveType` definition in section 7.
  Unit: unitless
  Range or possible values: `MV_REL | MV_ABS`
  Default value: `MV_REL`
- **`opeType` `OPE` (in)**
  Purpose: to define the element this command applies to.
  Description: enumerated variable to define the OPE, `MOUNT`, `M1`, `M2`, `M3`, `HEXAPOD` or `DEFAULT` (see description in section 7). For this command the `DEFAULT` option refers to the pointing kernel. The `DEFAULT` selection allows the

pointing kernel to choose the action which should be taken based upon its internal logic.
Unit: unitless
Range or possible values: `MOUNT | M1 | M2 | M3 | HEXAPOD | DEFAULT`
Default value: `DEFAULT`

- **`bool NEW_POSITION` (in)**
  Purpose: to determine if the target position should be changed or not.
  Description: bool variable. `true` = move the guide stage but does not change target RA and DEC. `false` = Update RA and DEC (dither).
  Unit: unitless
  Range or possible values: `true | false`
  Default value: `false`
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The telescope can be moved.
- The telescope is tracking and/or guiding and/or controlling AO.

After execution

- The telescope is on a new position, tracking, guiding and controlling the AOS.
- The pointing coordinates in the data dictionary do not change.
- New pointing coordinates.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
...
offset = {1.15, 1.35, COORD_RADEC_FOCAL}

//call OffsetPointing()
aResult = anIIF->OffsetPointing( 28.4567891e-2, offset, MV_REL, M1,
                                 false, SIDE_LEFT );
...
```

7.26 PauseGuiding

Description

This command is issued to temporarily suspend the current guiding operation.

Syntax

```
Result * PauseGuiding()
```

Attributes

- None

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope is observing.
- The telescope must be guiding.

After execution

- Guiding system is in pause mode.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->PauseGuiding( );
....
```

7.27 PresetGuiding

Description

This command is issued to start the guiding.

Syntax

```
Result * PresetGuiding ( Position ** guideStars,
                         const char * SIDE )
```

Attributes

- **Position ** guideStars (in)**
  Purpose: to specify the list of guide stars. **Note** that the guide stars in the list will be tried in the order as they were provided. The first one that is usable will be the one the GCS will use for the guiding and possible WF sensing.
  Description: `Position` structure to define the guide star list.
  Unit: unitless
  Range or possible values: -
  Default value: -
- **const char * SIDE (in)**
  Purpose: to define the side this command applies to.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing and tracking.
- The guide unit must be ready to be closed.

After execution

- The first suitable guide star from the list will have been acquired.
- The telescope is observing and guiding.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
....
Position *star1 = new Position(     2.354, .1234, COORD_RADEC_SKY, J2000,
                                    2000.0,NULL, &magnitude, 3421 );

Position *star2 = new Position(     3.5, .6, COORD_ALTAZ, J2000,
                                    &propm, &magnitude, 0 );

Position *star3 = new Position(     1.5, 1.6, COORD_RADEC_SKY, J2000,2007.0,
                                    &propm, &magnitude1, 2020 );

guidestars = (Position **) malloc(4 * sizeof(Position *));
guidestars[0] = star1;
guidestars[1] = star2;
guidestars[2] = star3;
guidestars[3] = (Position *) NULL;

aResult = anIIF->PresetGuding( guidestars, SIDE_LEFT );
....
```

7.28 PresetTelescope

Description

Slew the telescope into position in order to begin an observation cycle.

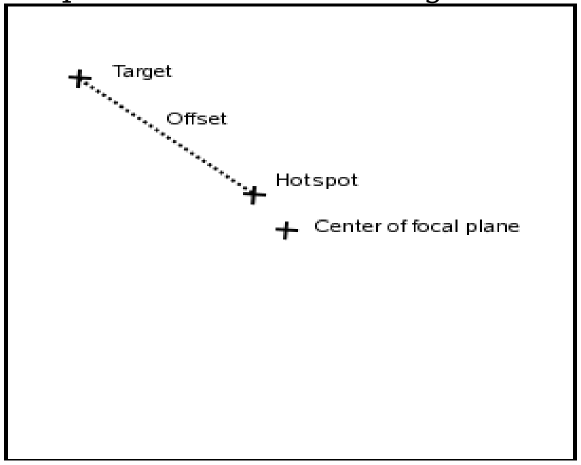


Figure 7.2: Positions in the focal plane for `PresetTelescope`.

Syntax

```
Result * PresetTelescope(double ROTANGLE,
                         rotatorType ROTATORMODE,
                         Position* TARGET,
                         [Position** GUIDESTARS],
                         modeType MODE,
                         const char* SIDE,
                         [Offset* OFFSET][,Hotspot* HOTSPOT],
                         [bool WRAPFLAG])
```

Attributes

- **double ROTANGLE (in)**
  Purpose: to specify the value in radians for the initial rotator angle.
  Description: double value to represent the IRA.
  Unit: radian
  Range or possible values: `[-2PI, 2PI]`
  Default value: `0`
- **rotatorType ROTATORMODE (in)**
  Purpose: to specify the rotator mode of the instrument.
  Description: enumerated value that represents the rotator mode. See description in section 7.
  Unit: -

Range or possible values: `ROTATOR_PAR | ROTATOR_PSTN | ROTATOR_NATIVE | ROTATOR_GRAV | ROTATOR_IDLE`
Default value: -

- **`Position * TARGET` (in)**
  Purpose: To specify all characteristics of the target and its location according to the coordinate system chosen.
  Description: `Position` object to define the target.
  Unit: See `Position` class in section 7.
  Range or possible values: See `Position` class in section 7.
  Default value: -
- **`Position ** GUIDESTARS` (in, optional)**
  Purpose: To specify all characteristics of the guide stars and their locations according to the coordinate system chosen.
  Description: list of positions to define the set of guide stars.
  Unit: See `Position` class above.
  Range or possible values: See `Position` class in section 7.
  Default value: `NULL`
- **`modeType MODE` (in)**
  Purpose: to specify the operating mode of the telescope.
  Description: value representing the telescope mode operation. See description in section 7.
  Unit: unitless
  Range or possible values: `MODE_STATIC | MODE_TRACK | MODE_GUIDE | MODE_ACTIVE | MODE_ADAPTIVE| MODE_INTERFEROMETRIC`
  Default value: -
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -
- **`Offset * OFFSET` (in, optional)**
  Purpose: to specify the offset for the target in RA and DEC, ALT and AZ, or SFP coordinates.
  Description: `Offset` object to define the pointing offset.
  Unit: see `Offset` class in section 7.
  Range or possible values: see `Offset` class 7.
  Default value: `NULL`
- **`Hotspot * HOTSPOT` (in, optional)**
  Purpose: to specify the reference position in the focal plane, by default, the center of the focal plane. See figure 7.1 and 7.2 as well.
  Description: `Hotspot` object.
  Unit: see `Hotspot` class in section 7.
  Range or possible values: see `Hotspot` class in section 7.
  Default value: `NULL`
- **`bool WRAPERFLAG` (in, optional)**
  Purpose: This is a "maximize-time-on-target" flag.
  Description: `bool` variable where `true` means to choose the path that selects the cable wrap that will provide the longest possible observing time on the

object. `false` means to move from the present position to a new object by the shortest path possible.
Unit: unitless
Range or possible values: `true | false`
Default value: `false`

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be ready to move, to track and control the active optics / adaptive optic system (AOS).

After execution

- The telescope is on a new position and tracking.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
....
MagnitudeType magnitude = {-27.0, V, 1.0, U_B};

ProperMotionType propm =  {0.123, 0.234};

Offset *offset = new Offset(  2.354, .1234, COORD_RADEC_FOCAL );

Hotspot *hotspot = new Hotspot(3.5, .6);

Position *target = new Position(    3.5, .6, COORD_RADEC_SKY, J2000,
                                    2007.00, &propm, &magnitude1, 2020);

Position GuideStar = new Position(  2.354, .1234, COORD_RADEC_SKY, J2000,
                                    2007.00, NULL, &magnitude, 3421);

guidestars = (Position **) malloc(3 * sizeof(Position *));
```

```
guidestars[0] = target;
guidestars[1] = GuideStar;
guidestars[2] = (Position *) NULL;

aResult = anIIF->PresetTelescope(  1.0, ROTATOR_PSTN, target, guidestars,
                                   MODE_ACTIVE, SIDE_LEFT, false);
....
```

7.29 ResumeGuiding

Description

This command resumes suspended operation after a `PauseGuiding`. See section 7.26

Syntax

```
Result * ResumeGuiding()
```

Attributes

- None

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- A previous `PauseGuiding` must be issued.
- Guiding system is in pause mode.

After execution

- Guiding system is running.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the AOS.

Examples

```
....
aResult = anIIF->ResumeGuiding();
....
```

7.30 RotateCommon

Description

The `RotateCommon` command rotates the primary and the secondary mirror around a common point. The movement is relative and not synchronized. Initial positions for the mirror are depending on the focal station and must be defined in the collimation model.

Syntax

```
Result * RotateCommon (  double X,
                         double Y,
                         double Z,
                         double ANGLE,
                         double DIRECTION,
                         const char * SIDE)
```

Attributes

- **`double X, Y, Z` (in)**
  Purpose: to specify the rotation point position in XYZ.
  Description: double values representing the position of the point the mirrors rotate around. The coordinate zero is TBD.
  Unit: millimeters
  Range or possible values: TBD
  Default value: -
- **`double ANGLE` (in)**
  Purpose: to specify the rotation angle.
  Description: double value representing the angle to be rotated.
  Unit: radians
  Range or possible values: `[-9999.999, 9999.999]`
  Default value: -
- **`double DIRECTION` (in)**
  Purpose: to specify the direction of the rotation.
  Description: double value to represent the direction (between 0 and 2pi).
  Unit: radians
  Range or possible values: `[0.0 , 2PI]`
  Default value: -
- **`const char * SIDE` (in)**
  Purpose: to define the side this command applies to.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope is observing.
- The OPE can be moved.
- Optional: Telescope is tracking, guiding, controlling active optics or in adaptive mode.

After execution

- The optical elements are in a new position.
- The telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
....
     aResult = anIIF->RotateCommon(1.35, 1.00, 2.00, 10, 3.15, SIDE_LEFT);
....
```

7.31 RotatePrimary

Description

The `RotatePrimary` command rotates the primary mirror around a fixed reference point on the optical axis.

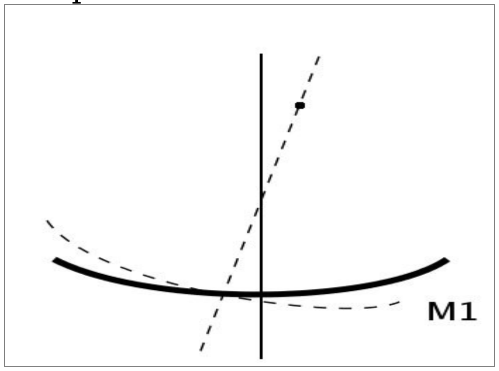


Figure 7.2: Primary mirror rotation diagram.

Syntax

```
Result * RotatePrimary(  double DISTANCE, double ANGLE,
                         double DIRECTION, const char * SIDE)
```

Attributes

- **`double DISTANCE` (in)**
  Purpose: to specify the rotation reference point's distance above the mirror in Z direction. (See figure 6.3 for more details)
  Description: double value
  Unit: millimeter
  Range or possible values: `[999.999 , 99999.999]`
  Default value: -
- **`double ANGLE` (in)**
  Purpose: to specify the value of the rotation angle with a precision of micro radians.
  Description: double value representing the angle.
  Unit: radians
  Range or possible values: `[-0.000 , 999.999]`
  Default value: -
- **`double DIRECTION` (in)**
  Purpose: to specify the direction of the rotation. Note that it is zero along the X axis and moves counterclockwise.
  Description: double value to represent the direction (between 0 and 2pi).
  Unit: radians
  Range or possible values: `[0.0, 2PI]`
  Default value: -

- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The primary mirror can be moved.

After execution

- Primary mirror is on a new position.
- Telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->RotatePrimary(1200.567, 110.499, 3.141592654, SIDE_LEFT);
...
```

7.32 RotateZ

Description

Using this command, the tertiary mirror is rotated to adjust the incoming beam angle for the instrument.

Syntax

```
Result * RotateZ (  double ANGLE,
                    moveType MOVE_TYPE,
                    const char * SIDE)
```

Attributes

- **double ANGLE (in)**
  Purpose: to specify the relative value of the angle to be rotated.
  Description: double value. The sign is such that a positive rotation will move the light beam toward the front of the telescope, regardless of side.
  Unit: micro radians
  Range or possible values: TBD
  Default value: -
- **moveType MOVE_TYPE (in)**
  Purpose: to determine if the movements are absolutes or relatives.
  Description: value to represent the movement type, MV_REL or MV_ABS. A relative rotation is incremental, that is, it adds to the current position. The absolute rotation is with respect to the focal station position maintained by the OSS. So an absolute rotation of zero will go back to the default focal station position.
  Unit: unitless
  Range or possible values: MV_REL | MV_ABS
  Default value: -
- **const char * SIDE (in)**
  Purpose: to define the side this command applies to.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: Result structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless

Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
`RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The OPE can be moved.

After execution

- Tertiary mirror (M3) is on a new position.
- We don't expect the TCS to send back any parameter. The evaluation of the command's result will let us know if the command was successfully executed by the TCS.

Examples

```
....
     aResult = anIIF->RotateZ(1.35, MV_REL, SIDE_LEFT);
....
```

7.33 RotAdjustPosition (Prototype)

Description

This command allows the instrument to specify (typically) fine adjustments to the current rotator angle (in order to position an image on a detector or a focal plane mask, for instance), although large angles can be specified also. This command would normally be issued while the rotator is tracking. In that case it makes an offset adjustment to the polynomials being generated by PCS. If this command is issued when the rotator is HOLDING, it will move the rotator by the specified amount and then resume holding at that new position. More information in reference [4], section 7.1.

Syntax

```
Result * RotAdjustPosition ( double DELTA_ROTANGLE,
                             rotatorType ROTATORMODE,
                             const char * SIDE )
```

Attributes

- **double DELTA_ROTANGLE (in)**
  Purpose: This signed angle increment is added as a cumulative offset to the current rotator angle. The angle is interpreted in the reference frame that is specified by the 'ROTATORMODE' parameter.
  Description: double value.
  Unit: Radians
  Range or possible values: `[ -Pi/2 , Pi/2 ]`
  Default value: -
- **rotatorType ROTATORMODE (in)**
  Purpose: This argument indicates in what coordinate system the angle should be interpreted. See section 7 for a full description of these options.
  Description: enumerated value that represents the rotator mode.
  Unit: -
  Range or possible values: `ROTATOR_PAR | ROTATOR_PSTN | ROTATOR_NATIVE |ROTATOR_GRAV | ROTATOR_IDLE`
  Default value: -
- **const char * SIDE (in)**
  Purpose: to define the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- Rotator should be on and enabled and either tracking or holding.

After execution

- The rotator position will be permanently shifted by the specified amount.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->RotAdjustPosition( 1.0252, ROTATOR_PAR, SIDE_RIGHT );
....
```

7.34 RotHold (Prototype)

Description

If the rotator is tracking or slewing, this command makes it stop moving and hold position at the point it was at when it received the hold command. If the rotator is already holding position, this command has no effect. See reference [4] for further details.

Syntax

```
Result * RotHold (  const char * SIDE )
```

Attributes

- **const char * SIDE (in)**
  Purpose: to define the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The rotator must be on and in the “ready” state.

After execution

- The rotator will be held motionless at the position it had when the command was issued.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

<u>Examples</u>

```
....
aResult = anIIF->RotHold( SIDE_LEFT );
....
```

7.35 RotMaximizeTime (Prototype)

Description

This command is to provide some control over the use of the rotator's cable wrap. When this command is executed, if the rotator and AZ axis are already in that part of their cable wraps that maximizes the time left to observe this object, this command does nothing. If they are not in the wrap which maximizes observing time on the object, one or both will do a "slew-to-track" to acquire the same object in the other end of their cable wrap. So, this command will either do nothing or it will slew the rotator and/or AZ axis 360 degrees.

Syntax

```
Result * RotMaximizeTime ( const char * SIDE )
```

Attributes

- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string defining the side, "left", "right" or "both".
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The rotator should be tracking an object.

After execution

- The rotator will be positioned tracking the object with the rotator in that part of the cable wrap which maximizes the amount of time available to observe an object.

- The rotator and AZ axis will continue tracking the object that was previously being tracked.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->RotMaximizeTime( SIDE_LEFT );
....
```

7.36 RotServicePosition (Prototype)

Description

Makes the rotator move to the specified angle in the specified coordinate frame and hold at that position.

Syntax

```
Result * RotServicePosition ( double ANGLE,
                              rotatorType ROTATORMODE,
                              const char * SIDE )
```

Attributes

- **double ANGLE (in)**
  Purpose: to specify the angle to which the rotator should be moved. It is taken to be expressed in the reference frame described by TRACKMODE
  Description: double value to represent the angle.
  Unit: radians
  Range or possible values: [-Pi/2, 5Pi/2] in the native coordinate frame
  [-3Pi/2, 3Pi/2] in the parallactic ref. frame
  [-3Pi/2, 3Pi/2] in gravitational ref. Frame
  Maximum range in the "position angle" reference frame is a function of where the telescope is pointing.
  Default value: 0
- **rotatorType ROTATORMODE (in)**
  Purpose: to specify the rotator mode of the instrument.
  Description: enumerated value that represents the rotator mode. See section 7 for a full description of these options.
  Unit: -
  Range or possible values: ROTATOR_PAR | ROTATOR_PSTN | ROTATOR_GRAV
  ROTATOR_NATIVE | ROTATOR_IDLE
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string defining the side, "left", "right" or "both".
  Unit: unitless
  Range or possible values: SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.

Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
Unit: unitless
Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
`RESCODE_OK`: Command accepted.

Preconditions

- The rotator must be on and in the "ready" state.

After execution

- The rotator will be at the specified position and holding position, ready to start tracking or to slew to another position.
- We don't expect the TCS to send back any parameter. The evaluation of the command's result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->RotServicePosition( 0.1415, TRACK_GRAV, SIDE_LEFT );
....
```

7.37 RotSetRotator (Prototype)

Description

This command is issued to enable or disable a rotator. “Enable” means to turn the rotator on and make it ready to respond to commands. This command is specifically designed to be used by an instrument that is not the “authorized” instrument. However, an authorized instrument can invoke this command as necessary.

Syntax

```
Result * RotSetRotator (  bool ENABLE ,
                          const char * SIDE )
```

Attributes

- **bool ENABLE (in)**
  Purpose: to determine whether to activate or deactivate the rotator associated with the specified focal station.
  Description: bool variable.
  Unit: unitless
  Range or possible values: `true | false`
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- None

After execution

- The rotator will turn on and become ready to respond within 10 or 20 seconds. It will be holding its present position and ready to slew or track.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->RotSetRotator(true, SIDE_LEFT);
....
```

7.38 RotTrack (Prototype)

Description

Makes rotator begin tracking according to the polynomial stream it is currently receiving from the PCS. It will in general, need to do a slew to the target position and then start tracking. See reference [4], section 7.1 for further details.

Syntax

```
Result * RotTrack ( const char * SIDE )
```

Attributes

- **const char * SIDE (in)**
  Purpose: to define the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The system is ready to track, so the rotator is in "Holding" mode.
- The command can also be issued with the rotator in “Tracking” mode, in which case this command would have no effect.

After execution

- The rotator will be tracking as described by the polynomials being generated by the PCS.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was

successfully executed by the TCS.

Examples

```
....
aResult = anIIF->RotTrack( SIDE_LEFT );
....
```

7.39 RotNextPosition (Prototype)

Description

The PCS maintains in reflective memory a set of variables which describe the time-to-limit for this “next position”. These variables include the time remaining for an object at the specified R.A. and Dec. position to reach the specified elevation limit with EL **decreasing** and the time to reach it with EL **increasing**. This RA and Dec may or may not be the position of an object that will ever be observed. This command has no affect on telescope or rotator motion. Also maintained in reflective memory is the time remaining for this object until the AZ wrap will hit a limit and the time remaining until each rotator wrap will hit a limit. This set of time-to-limit variables is separate from the set which relate to the position which the PCS is presently tracking. This set is for an observer/instrument to use to “ask” how long a particular object could be observed if observation of it was to begin now. More information in reference [4], section 7.1.

Syntax

```
Result * RotNextPosition( double RA,
                          double DEC,
                          double LIMIT,
                          const char * SIDE )
```

Attributes

- **double RA (in)**
  Purpose: Specifies the R.A. of the object of interest.
  Description: double value.
  Unit: radians
  Range or possible values: [0.0 , 2Pi]
  Default value: -
- **double DEC (in)**
  Purpose: Specifies the declination of the object of interest.
  Description: double value.
  Unit: radians
  Range or possible values: [-Pi/2 , Pi/2]
  Default value: -
- **double LIMIT (in)**
  Purpose: Specifies the EL limit value that is of interest.
  Description: double value.
  Unit: radians
  Range or possible values: [0.0 , Pi/2]
  Default value: -

- **`const char *` `SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value:

Return values

- **`Result *` `RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- None

After execution

- The “time-to-limit” values for the “next position” in reflective memory will be computed using these parameters. It also returns the current AZ and EL for the specified object.

Examples

```
....
aResult = anIIF->RotNextPosition( 2.245, 1.231, 2.000, SIDE_LEFT );
....
```

7.40 SendWavefront

Description

`SendWavefront` sends an array of Zernike coefficients (see reference [11]) to be compensated by the actuators of the primary or secondary mirror. Reducing atmospheric and optical aberrations, this command should improve the image quality.

Syntax

```
Result * SendWavefront(  Wavefront * POLYNOM,
                         opeType OPE,
                         const char * SIDE )
```

Attributes

- **`Wavefront * POLYNOM` (in)**
  Purpose: to define the 28 Zernike polynomial coefficients with a precision of 0.01 nm for wavefront correction.
  Description: `Wavefront` object to define the Zernike coefficients. See `Wavefront` class in section 7.
  Unit: nanometers
  Range or possible values: `[-99999.99 , 99999.99]`
  Default value: `0` (all coefficients of the WF instance are pre-populated with 0.0)
- **`opeType OPE` (in)**
  Purpose:to define the optical element this command applies to. The OPE could be the primary or the secondary mirror.
  Description: enumerated variable to define the optical element (OPE). See description in section 7.
  Unit: unitless
  Range or possible values: `M1 | M2`
  Default value: `M1 (For Gregorian instrument M2).`
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to

query for further information.

Unit: unitless

Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
`RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The telescope can be moved.
- The telescope is tracking and/or guiding.

After execution

- Mirror in new shape, means better image quality.
- Telescope observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command's result will let us know if the command was successfully executed by the TCS.

Examples

```
...
WaveFront *wfs = new WaveFront(); //Dummy instance. All coefficients 0.0
wfs->setCoefficient(812.0 , 1); /* set one coefficient (#1) non-zero */
wfs->setCoefficient(999.4 , 2); /* set one coefficient (#2) non-zero */
wfs->setCoefficient(1120.0 , 3);  /* set one coefficient (#3) non-zero */
wfs->setCoefficient(1102.8 , 4); /* set one coefficient (#4) non-zero */

// Call SendWavefront()
aResult = anIIF->SendWavefront(wfs, M1, SIDE_LEFT);
...
```

7.41 SetMultiParameter

Description

`SetMultiParameter` sets the values of the specified data dictionary entries on the TCS in one shot. Note, that the instrument only has permission to modify its own predefined entries.

Syntax

```
Result * SetMultiParameter( MultiDDEntry MULTIENTRIES )
```

Attributes

- **`MultiDDEntry MULTIENTRIES` (in)**
  Purpose: to define the structure specifying the list of parameters and values to set in the data dictionary.
  Description: `MultiDDEntry` structure to represent the list of data dictionary entries and their respective values.
  Unit: unitless
  Range or possible values: `Valid Data Dictionary entries / values.`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The MultiDDEntry object is populated with string pairs: the first string is the local data dictionary name, and the second is the value. The CSQ subsystem will generate the fully qualified data dictionary name as "csq.<InstrumentID>.entry_name". An instrument is not allowed to set variables it does not own.

After execution

- New values for the specified data dictionary entries.

- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
...
MultiDDEntry DDEntries;
DDEntries.PushEntry(“side[0].cooler” , “ON”);
DDEntries.PushEntry(“side[1].cooler” , “OFF”);
aResult = anIIF->SetMultiParameter(DDEntries);
...
```

7.42 SetParameter

Description

Allows the instruments to set a new value of a predefined data dictionary entry on the TCS.

Syntax

```
Result * SetParameter(   const char * DDENTRY,
                         const char * DDENTRYVALUE )
```

Attributes

- **`const char * DDENTRY` (in)**
  Purpose: to specify the name of the data dictionary entry to set with a new value.
  Description: string to define the data dictionary entry.
  Unit: unitless
  Range or possible values: `valid data dictionary entry.`
  Default value: -
- **`const char * DDENTRYVALUE` (in)**
  Purpose: to specify the new value of the data dictionary entry.
  Description: string variable to define the new value of the data dictionary entry.
  Unit: unitless
  Range or possible values: `Valid Value.`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The data dictionary entry must be predefined in the TCS (the fully qualified data dictionary name will be generated by the CSQ as "csq.<InstrumentID>.<DDENTRY>").

After execution

- The command sets the new value of the data dictionary entry on the TCS side.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->SetParameter("side[0].coolers",“OFF”);
...
```

7.43 Standby

Description

Standby command tells the TCS that currently the instrument is not using one or both of the telescope sides, but the side(s) will not given over to another instrument for control. This command is useful to avoid that other instruments change the settings that the authorized instrument have already done before the observations

Syntax

```
Result * Standby(int LEVEL, const char * SIDE)
```

Attributes

- **int LEVEL (in)**
  Purpose: to specify the new value of the standby level. The meaning of the standby levels is TBD, and will probably be instrument specific.
  Description: integer value to represent the standby level.
  Unit: unitless
  Range or possible values: `TBD`
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -
-

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instruments must be previously authorized.

After execution

- The specified side is in an “standby” state. No other instrument can take control of it.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->Standby(4, SIDE_RIGHT);
...
```

7.44 StartGuiding

Description

This command is used to start again the guiding loop, using the existing information and the setup declared in the last `PresetGuiding` command provided for the given telescope side.

Syntax

```
Result * StartGuiding()
```

Attributes

- Nones.

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- PresetTelescope and PreseGuiding must be done.
- Telescope is tracking.

After execution

- Telescope is tracking and guiding.
- We don't expect the TCS to send back any parameter. The evaluation of the command's result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->StartGuiding( );
....
```

7.45 StepFocus

Description

StepFocus moves the respective focus position, by moving the OPE a given distance in the direction of of the telescope's Z axis[6].

Syntax

```
Result * StepFocus( double RELPOS, opeType OPE,
                    const char * SIDE )
```

Attributes

- **double RELPOS (in)**
  Purpose: to specify the new relative position Z of the respective optical element.
  Description: double value to represent the delta to the current position.
  Unit: millimeter
  Range or possible values: `Depends on OPE and current position.`
  Default value: -
- **opeType OPE (in)**
  Purpose: to specify the optical element which should move in Z.
  Description: enumerated variable to define the optical element (OPE)[7]. See description in section 7.
  Unit: unitless
  Range or possible values: `M1 | M2 | M3 | M1M2 (scale-preserving focus)`
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

7 Note that M3 does not move in z in the traditional sense. See figure 6.3 for more details.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The OPE can be moved.
- The telescope might be tracking, guiding or controlling AOS.

After execution

- OPE is on a new position.
- Telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command's result will let us know if the command was successfully executed by the TCS.

Examples

```
...
aResult = anIIF->StepFocus(-1.42, M1, SIDE_LEFT);
...
```

7.46 StopGuiding

Description

This command is issued to stop the current guiding operation.

Syntax

```
Result * StopGuiding()
```

Attributes

- None

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing and guiding, otherwise this command has no effect.

After execution

- The telescope is observing, tracking and not guiding.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the GCS or AOS.

Examples

```
....
aResult = anIIF->StopGuiding();
....
```

7.47 TelescopeMove (Prototype)

Description

This command moves the entire optics on a single side as a rigid body.

Syntax

```
Result * TelescopeMove(  float TRANSX,
                         float TRANSY,
                         float TRANSZ,
                         moveType MOVE_TYPE,
                         const char * SIDE )
```

Attributes

- **float TRANSX,TRANSY,TRANSZ (in)**
  Purpose: to specify the translation value in XYZ.
  Description: float value to define the translation value.
  Unit: TBD
  Range or possible values: TBD
  Default value: -
- **moveType MOVE_TYPE (in)**
  Purpose: to specify the movement type, relative or absolute.
  Description: enumerated variable to define the movement type, “relative” or “absolute”.
  Unit: unitless
  Range or possible values: `MV_REL | MV_ABS`
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error
  `RESCODE_OK` The command was accepted.

Preconditions

- The instruments must be previously authorized.

After execution

- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

7.48 TelescopeRotate (Prototype)

Description

This command rotates the entire optics on a single side as a rigid body.

Syntax

```
Result * TelescopeRotate( rotcenterType ROTATIONCENTER,
                          float ANGLEX,
                          float ANGLEY,
                          float ANGLEZ,
                          moveType MOVE_TYPE,
                          const char * SIDE )
```

Attributes

- **rotcenterType ROTATIONCENTER (in)**
  Purpose:
  Description: enumerated variable to define the rotation center.
  Unit: unitless
  Range or possible values: `M1 | M2 | M3 | FS_PRIME | FS_DIRECTGREGORIAN | ROT_CENTER_POS`
  Default value: -
- **float ANGLEX,ANGLEY,ANGLEZ (in)**
  Purpose: to specify the rotation angle in X, Y, and Z.
  Description: float value to define the rotation angle.
  Unit: TBD
  Range or possible values: TBD
  Default value: -
- **moveType MOVE_TYPE (in)**
  Purpose: to specify the coordinate type, relative or absolute.
  Description: enumerated variable to define the movement type, relative or absolute.
  Unit: unitless
  Range or possible values: `MV_REL | MV_ABS`
  Default value: -
- **const char * SIDE (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**

  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error
  `RESCODE_OK` The command was accepted.

Preconditions

- The instruments must be previously authorized.
- The OPE can be moved.

After execution

- The entire optics are in a new position.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

7.49 TelescopeScale (Prototype)

Description

It adjusts the overall plate scale of the telescope side as delivered to the instrument. The plate scale will be changed and the side will remain in focus.

Syntax

```
Result * TelescopeScale( float SCALE, const char * SIDE )
```

Attributes

- **`float SCALE` (in)**
  Purpose:
  Description: float value to define the scale.
  Unit: unitless
  Range or possible values: `[ -2.5E-4 , -2.5E-4 ]`
  Default value: -
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL` indicating an error
  `RESCODE_OK` The command was accepted.

Preconditions

- The instruments must be previously authorized.
- The OPE can be moved.

After execution

- We don't expect the TCS to send back any parameter. The evaluation of the command's result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

7.50 TipTilt

Description

The `TipTilt` command moves an OPE in tip or tilt direction, relative to the current position.
Please note that for OPE M3, `XROTATION` is Tip, and `YROTATION` is Tilt, which are defined local to the M3 mirror. Positive tip will move the beam up, and positive tilt will move the beam toward the front of the telescope, regardless of side.

Syntax

```
Result * TipTilt(   double XROTATION, double YROTATION,
                    opeType OPE, const char * SIDE )
```

Attributes

- **`double XROTATION` (in)**
  Purpose: to specify the angle to rotate around x axis, relative to the current position with a precision of 0.001 micro radians.
  Description: double value to represent the angle to rotate around x axis.
  Unit: micro radians
  Range or possible values: `Depends on OPE.`
  Default value: -
- **`double YROTATION` (in)**
  Purpose: to specify the angle to rotate around y axis, relative to the current position with a precision of 0.001 micro radians.
  Description: double value to represent the angle to rotate around y axis.
  Unit: micro radians
  Range or possible values: `Depends on OPE.`
  Default value: -
- **`opeType OPE` (in)**
  Purpose: to specify the optical element this command applies to. The OPE could be the primary, the secondary or the tertiary mirror.
  Description: enumerated variable to define the optical element (OPE). See description in section 7.
  Unit: unitless
  Range or possible values: `M1 | M2 | M3`
  Default value: `M1`
- **`const char * SIDE` (in)**
  Purpose: to specify the side of the telescope.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: `SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH`
  Default value: -

Return values

- **`Result * RESULT` (returned)**
  Purpose: to evaluate if the command was accepted and successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL:` an error occurred.
  `RESCODE_OK:` Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be observing.
- The OPE can be moved
- The telescope might be guiding or controlling AO.

After execution

- OPE is on a new position.
- Telescope is observing.
- We don't expect the TCS to send back any parameter. The evaluation of the command’s result will let us know if the command was successfully executed by the TCS.

Examples

```
....
aResult = anIIF->TipTilt(1.255, -0.815, M1, SIDE_LEFT);
...
```

7.51 UpdateGuidestar

Description

This command is used to update the list of guide stars for the specified side.

Syntax

```
Result * UpdateGuidestar (    Position ** GUIDESTARS,
                              const char * SIDE )
```

Attributes

- **Position ** GUIDESTARS (in)**
  Purpose: to specify the list of guide stars.
  Description: `Position` structure to define the guide star list. See definition in section 7.
  Unit: unitless
  Range or possible values: -
  Default value: -
- **const char * SIDE (in)**
  Purpose: to define the side this command applies to.
  Description: string to define the side, “left”, “right” or “both”.
  Unit: unitless
  Range or possible values: SIDE_LEFT | SIDE_RIGHT | SIDE_BOTH
  Default value: -

Return values

- **Result * RESULT (returned)**
  Purpose: to evaluate if the command was successfully executed.
  Description: `Result` structure received from the TCS with the result code, a text describing the result code and a request handle, which can be used to query for further information.
  Unit: unitless
  Range or possible values: Result code = `RESCODE_FAIL`: an error occurred.
  `RESCODE_OK`: Command accepted.

Preconditions

- The instrument must be authorized.
- The telescope must be ready to move, to track and control the active optic system.
- In the current version of GCS it is required to issue this command before the start of the acquisition in order to have an effect. This

restriction might change or become obsolete with future versions of GCS

After execution

- We don't expect the TCS to send back any parameter. The evaluation of the command’s result (described in detail on section 8.4) will let us know if the command was successfully executed by the TCS.

Examples

```
....
Position *star1 = new Position(     2.354, .1234, COORD_RADEC_SKY, J2000,
                                    2000.0,NULL, &magnitude, 3421);

Position *star2 = new Position(     3.5, .6, COORD_ALTAZ, J2000,
                                    &propm, &magnitude, 0);

Position *star3 = new Position(     1.5, 1.6, COORD_RADEC_SKY, J2000, 2007.0,
                                    &propm, &magnitude1, 2020);

guidestars = (Position **) malloc(4 * sizeof(Position *));
guidestars[0] = star1;
guidestars[1] = star2;
guidestars[2] = star3;
guidestars[3] = (Position *) NULL;

aResult = anIIF->UpdateGuidstar( guidestars , SIDE_LEFT );
....
```

8 Process flow. Usage

The following section gives detailed information concerning the programmatic use of the IIF classes and data types, describing a walk-through of the coding steps that should be followed by the instrument teams when use the IIF libraries to implement the processes for an entire observation cycle.

8.1 Including IIF into the project

To access the IIF's classes through object-oriented C++ code, the instrument code developer includes the header file `IIF.h`. The include directive for the C++ source file equivalent to the example in the above section read as follows:

```
#include <IIF/Instrument/IIF.h>
```

To link the IIF libraries into a C++ project, one can use the same dynamic or static library files, `libLBTIIFlib.so` or `libLBTIIFlib.a`, respectively. An example for a C++ compiler and linker call to link in the IIF libraries follows:

```
g++ -I/usr/local/include/LBT –Wall iiftest_c++.cpp \
–o iiftest_c++ -lLBTIIFlib
```

For further details about the procedure to compile and install the IIF libraries into the instrument control computer, see reference [2].

8.2 Creating IIF and Authorizing with TCS

The first step in using the IIF libraries is to create a new instance of the `IIF` class. When creating an `IIF` instance, instruments identify themselves by two pieces of information:

1. their unique name (i.e., 'LBC', or 'LUCIFER2'),
2. their focal station, which consists of
    a. the location of the instrument (i.e. 'prime', or 'directGregorian'), plus
    b. the telescope side over which they intend to control (i.e., 'left', 'right', or 'both').

The instrument control software now instantiates an IIF, in which instrument

ID and focal-station/side string must be correctly specified. Otherwise, an exception will be thrown.

Instrument name	Focus and side
LUCIFER	bentGregorianFront both
LUCIFER1	bentGregorianFront left
LUCIFER2	bentGregorianFront right
LINC	bentGregorianBack left, bentGregorianBack right, bentGregorianBack both
LBTI	bentGregorianCenter left, bentGregorianCenter right, bentGregorianCenter both
PEPSI	directGregorian both
PEPSI1	directGregorian left
PEPSI2	directGregorian right
PEPSIFIBER	bentGregorianRearFiberFeed both
PEPSIFIBER1	bentGregorianRearFiberFeed left
PEPSIFIBER2	bentGregorianRearFiberFeed right
MODS	directGregorian both
MODS1	directGregorian left
MODS2	directGregorian right
LBC	prime both
LBCRED	prime right
LBCBLUE	prime left
MAT	prime both

Table 8.1: list of valid instrument “name – focal station” combinations.

An example of how to code this in C++ code:

```
...
//Before any communication over the IIF, it needs to be initialized with
//IDs determining the instruments focal station and its name.

IIF * anIIF;
try
{
      //pretend to be the left-side, prime focus blue-channel LBC.

      anIIF = new IIF("prime left", "LBCBLUE");
}
catch (int e)
{
      cout  << "The IIF failed to initialize" << endl;
      exit(-1);
}
...
```

The next step is to authorize the instrument specified in the instantiation of the IIF object, in order to request any command that potentially will change the state of the telescope. To request such authorization, the ICS has to call the IIF `authorize()` method.
If the authorization is granted by the TCS, this function returns `true`, allowing the instrument to successfully issue new commands.
In case that another instrument already controls the specified side (or both sides) of the telescope, `authorize()` returns `false`.
Please note that `Authorize()` does not return a `CommandReturn` object. See its description for more details in section 7.12.

Coding in C++:

```
...
if (!anIIF->Authorize())
{
      cout << "Authorize() was not successful..." << endl;
      exit(-1);
}

cout << "Authorize() successful" << endl;
...
```

For multi-threaded ICS, TCS allows multiple authorizations, maintaining an internal count for each side. Please find more details in sections 6.1.1, 6.1.2, 7.12, and section 8.5 as well.

8.3 Command request process

In order to initiate command processing inside the TCS via the IIF, an application needs to use the `IIF`, `Result` and `StatusInfo` classes provided by the IIF libraries. These classes are described in detail in section 7.

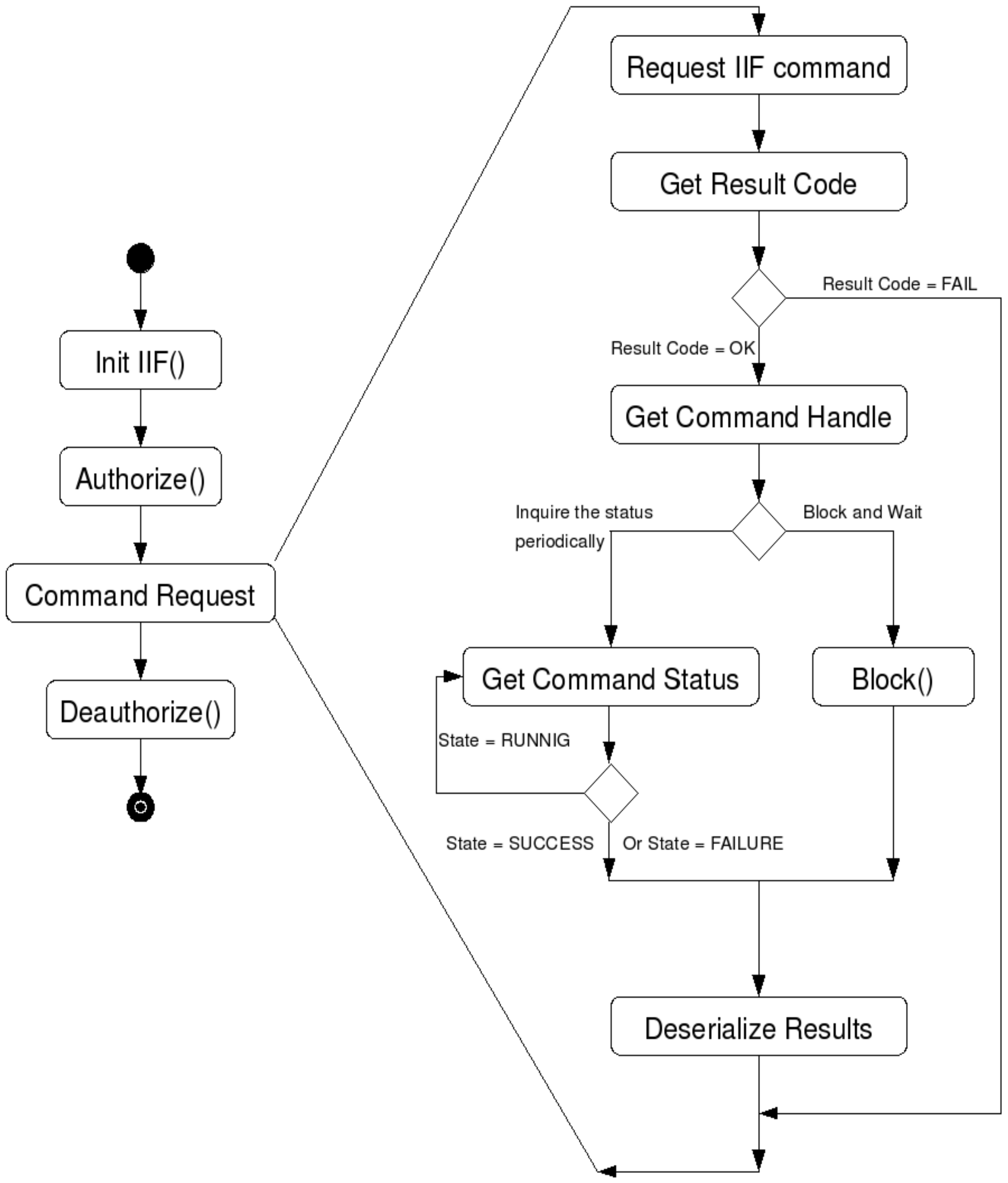
After the creation and authorization of its own `IIF` instance (see details in section 8.2), each IIF command immediately returns a `Result` entity, while command processing might still be in process inside the TCS (i.e., the telescope is still slewing, while the IIF's `PresetTelescope()` command call has already returned a `Result` instance).

Each `Result` instance, then contains a result code, which indicates if requesting the specified action(s) from the TCS was successfully sent to the TCS. If the result code indicates that there was a problem (i.e., it evaluates to `RESCODE_FAIL` as defined by the IIF library's header files), then the programmer can find a textual indication of the reason for the failure in the `Result`'s reply code.

Otherwise, the `Result` instance's result code evaluates to `RESCODE_OK`. In this successful case, another component of the result becomes meaningful, namely the command handle, which is a pointer that identifies the associated command processing inside the TCS. In the failure case above, this pointer is simply `NULL`, but if the command was successfully sent to the TCS, then the command handle has a value different from `NULL`.

Using a (non-null) command handle, the IIF's `GetCommandStatus()` method can be used to inquire about the status of the respective command's processing inside the TCS. This method returns a StatusInfo entity for a given command handle, which can be used to query the command's processing status and to get the XML-based string representation of the command's result once processing has finished inside the TCS. More details about this process can be found in section 8.4.

The UML activity diagram 8.1 shows the command request process.



UML activity diagram 8.1: Command request process

8.4 Command result evaluation

Once command processing has finished on the TCS' side, the `StatusInfo` object that is associated with this command exhibits a status code that differs from `STATE_RUNNING` (See UML activity diagram 8.1); i.e., the command status property of the respective `StatusInfo` object is either `STATE_SUCCESS`, `STATE_CANCELED`, `STATE_FAILURE`, or `STATE_WRONGHANDLE` (as defined in file `StatusInfo.h`).

The programmer has two options in waiting for such finishing of the command processing: “asynchronous” or “synchronous” IIF command processing. In “asynchronous” mode, the programmer will periodically execute the IIF’s `GetCommandStatus()` method, in order to receive a new `StatusInfo` object, with an updated command status property. The “synchronous” mode means that the programmer can choose to halt the program thread’s execution until the TCS has finished the command processing (“sleep wait”, no CPU activity is consumed). To achieve this, the IIF user simply calls the `Block()` method on the `StatusInfo` object that is associated with the requested command.

Table below briefly describes the meaning of the status code values mentioned above.

Status code	Description
`STATE_RUNNING`	The command is still being executed by the TCS
`STATE_SUCCESS`	The TCS has successfully finished processing the associated command; note that the actual actions requested to be performed by the telescope can still have failed.
`STATE_CANCELED`	The TCS command’s processing has been previously cancelled.
`STATE_FAILURE`	An error has occurred during execution of the associated command by the TCS.
`STATE_WRONGHANDLE`	There is no valid TCS-side command associated with the `StatusInfo` object’s command handle.

Table 8.2: Status codes in `StatusInfo`.

If the status code is either `STATE_SUCCESS`, `STATE_CANCELED`, or `STATE_FAILURE`, then the IIF user should analyze the command’s result that was produced and returned by the TCS. In order to process the returned result, the programmer has to access the `StatusInfo` object’s command result property[8]. The string inside the command result is an XML-string encoded `CommandReturn` object, as provided by the TCS’ Common Software library. This section will explain how to access these objects’ data as it will be most commonly accessed by the IIF user. For further details on the Common

8 NOTE: The command result will be `STATE_NORESULT`, if the user tries to retrieve the command result on a `StatusInfo` object whose command handle is invalid, or `NULL`.

Software library and the `CommandReturn` class, please consult the respective sections of the LBT TCS Common Software documentation (See reference [12]).

First, the IIF user instantiates a `CommandReturn` object. On this object, it calls the `deseralize()` method with the above mentioned `StatusInfo` object's command result. As a consequence, the `CommandReturn` object is ready to access the associated TCS command's results in a more convenient way. In order to decide if the actual actions inside the telescope control system, which were triggered by firing off the associated TCS command, have been performed without any problems, one has to check the `CommandReturn`'s `isError()` method. If it returns `false`, then the processes inside the TCS showed no glitches, otherwise, there were problems or failures.

The actual results (in the success case), or respectively indications about errors (in the error case), are generally retrieved via `CommandReturn`'s `getResultCount()` and `getResultDescription(int n)` methods. The first method returns the number of results that can be queried via the second method, while the second method returns an STL string representation of the TCS command's result, or error message respectively.

Let us see an example. We want to tip-tilt the primary mirror on the left side (assuming the instrument is already authorized):

```
Result* aResult = anIIF->TipTilt(1.255, -0.815, M1, SIDE_LEFT);

if (aResult->GetresultCode() != RESCODE_FAIL)
{
  //Once TCS has finished, we want to know the status of the command;
  //In order to do it, we instantiate a StatusInfo object.
  StatusInfo* statusInf = anIIF->GetCommandStatus(aResult->GetcommandHandle());
  statusInf->Block(); // wait for TCS

  cout << "TIPTILT command status: "<< statusInf->GetCommandStatus() << endl;

  //get the information from the XML-string encoded CommandReturn object
  CommandReturn cmdRet;
  cmdRet.deserialize(statusInf->GetCommandResult());

  //Check if the deserialization fails
  if(cmdRet.isError()) cout << "TIPTILT command result status: Error" << endl;
  else cout << "TIPTILT command result status: OK" << endl;

  cout << "TIPTILT command result: ";

  // get the result(s) of the command
  for(int i=1; i<=cmdRet.getResultCount(); i++)
      cout << cmdRet.getResultDescription(i) << endl;
  delete statusInf;
}
else
{
  cout << "TipTilt execution failed: " << aResult->GetresultString() << endl;
}
  delete aResult; // free memory for CSQHandle
```

An exception to the rule that all results are accessed via `getResultDescription()` is made for the LBC IIF's `GetRotatorTrajectory()`and `GetMultiParamer()` (only for the C wrapper). Here, if the commands have finished successfully, the results are accessed via `StatusInfo_CommandResult_getTrajectory()` method and `StatusInfo_CommandResult_getMultiParameter()`respectively. Error indications (i.e., if `isError()` returned `true`) can still be retrieved through `getResultDescription()`.

Note also that commands `Authorize`, `Deauthorize` and `GetCommandStatus` do not return `CommandReturn` objects. See their descriptions for details.

8.5 De-authorizing the instruments. Destroying IIF instance

In order to let the TCS know that an instrument wants to surrender control of one or more telescope sides (end of the observation), the instrument control code must execute the IIF's `Deauthorize()` command. The IIF user has to specify the telescope side(s) that it wants to release, i.e., `SIDE_LEFT`, `SIDE_RIGHT`, or `SIDE_BOTH`.
The `Deauthorize` command can return the following:

Side	String result	Meaning
`SIDE_LEFT`	`AUTHORIZED`	left side is still authorized
`SIDE_LEFT`	`DEREGISTERED`	left side is not authorized
`SIDE_RIGHT`	`AUTHORIZED`	right side is still authorized
`SIDE_RIGHT`	`DEREGISTERED`	right side is not authorized
`SIDE_BOTH`	`AUTHORIZED`	both sides are still authorized
`SIDE_BOTH`	`DEREGISTERED`	both sides are not authorized
`SIDE_BOTH`	`DEREGISTERED left`	left side is not authorized, right side is still authorized
`SIDE_BOTH`	`DEREGISTERED right`	right side is not authorized, left side is still authorized

Table 8.3: Possible results the `Deauthorize` command can return.

The returned string will be `DEREGISTERED` if the authorized count is now zero. If there are still authorize requests remaining, the string will be `AUTHORIZED`. Thus every successful `Authorize` command must be paired with a `Deauthorize` command in order to fully release that side of the telescope.

8.6 Full example

The following code shows the entire process in order to communicate with the TCS, initializing IIF, sending commands over the TCS<->Instrument interface and printing the results and error messages to the standard output (stdout). You will find more examples ready to compile and use in the source code of the IIF (`LBTO/TCS/IIF/Instrument/Examples`).

```
#include <IIF/Instrument/IIF.h>

int main(int argc, char *argv[])
{

       // Before any communication over the IIF, it needs to be initialized with
       // IDs determining the instruments focal station and its name.

       IIF * anIIF;
       try
       {
              //pretend to be the left-side, prime focus blue-channel LBC
              anIIF = new IIF("prime left", "LBCBLUE");
       }
       catch (int e)
       {
              cout << "The IIF failed to initialize" << endl;
              exit(-1);
       }

       // Permission for commanding the respective telescope sides is requested,
       // and -upon failure to achieve authorization- the program is exited
       if (!anIIF->Authorize())
       {
              cout << "Authorize() was not successful ... terminating test program.";
              exit(-1);
       }
       cout << "Authorize() successful" << endl;

       // Declare the result object reference needed as a container for the
       // information returned via the IIF.

       Result* aResult;

       // Initialize some objects to use as example

       MagnitudeType magnitude1 = {-27.0, V, 1.5, B_V};
       MagnitudeType magnitude2 = {20.0, R ,2.0, U_V};
       ProperMotionType propm1= {0.123, 0.234};
       ProperMotionType propm2= {0.567, 0.789};
       Position *target =
       new Position(3.5,1.6,COORD_RADEC_SKY, J2000, 2007, &propm1, &magnitude1, 2000);
       Position *guide1 =
       new Position(1.5234,1.6123,COORD_RADEC_SKY, J2000, 2007, &propm1, &magnitude1,
                     2020);
       Position *guide2 =
       new Position(3.5234,2.23,COORD_RADEC_SKY, J2000, 2007, &propm1, &magnitude1,
                     2100);
```

```
Offset *offset;
Hotspot *hotspot = new Hotspot(3.5, 1.6);
Position **guidestars;

// Get some Parameters from the data dictionary
cout << "GetMultiParameter" << endl;

MultiDDEntry multiEntries;
multiEntries.PushEntry("csq.authorizedInstrument.left");
multiEntries.PushEntry("csq.authorizedCount.left");
multiEntries.PushEntry("csq.authorizedFocalStation.left");
multiEntries.PushEntry("csq.standbyLevel.left");
multiEntries.PushEntry("pmc.side[1].temperature");

aResult = anIIF->GetMultiParameter(multiEntries);

if (aResult->GetresultCode() != RESCODE_FAIL)
{
      StatusInfo* statusInf =
            anIIF->GetCommandStatus(aResult->GetcommandHandle());
      statusInf->Block(); /* wait for TCS */

      cout << "GETMULTIPARAMETER command status: " <<
            statusInf->GetCommandStatus() << endl;

      CommandReturn cmdRet;

      cmdRet.deserialize(statusInf->GetCommandResult());

      if(cmdRet.isError())
            cout << "GETMULTIPARAMETER command result status: Error" << endl;
      else
            cout << "GETMULTIPARAMETER command result status: OK" << endl;

      cout << "GETMULTIPARAMETER command result: ";
      if(cmdRet.isError()) cout << "FAILED" << endl;
      else cout << "SUCCESS" << endl;
      for(int i=1; i<=cmdRet.getResultCount(); i++)
            cout << cmdRet.getResultDescription(i) << endl;
      delete statusInf;
      } else {
      cout << "GetMultiParameter execution failed: " <<
                  aResult->GetresultString() << endl;
      }

      delete aResult; /* free memory for CSQHandle */

cout << "Preset" << endl;

offset = new Offset(2.34, .124, COORD_RADEC_FOCAL);

guidestars = (Position **) malloc(4 * sizeof(Position *));
guidestars[0] = target;
guidestars[1] = guide1;
guidestars[2] = guide2;
guidestars[3] = (Position *) NULL;

// Call PresetTelescope()
aResult =
        anIIF->PresetTelescope(1.0, ROTATOR_PSTN, target, guidestars, MODE_ACTIVE,
                               SIDE_LEFT, true);
    if (aResult->GetresultCode() != RESCODE_FAIL)
    {
          StatusInfo* statusInf =
                anIIF->GetCommandStatus(aResult->GetcommandHandle());
          statusInf->Block(); /* wait for TCS */

          cout << "PRESET_TELESCOPE command status: " <<
                statusInf->GetCommandStatus() << endl;

          CommandReturn cmdRet;

          cmdRet.deserialize(statusInf->GetCommandResult());

          if(cmdRet.isError())
                cout << "PRESET_TELESCOPE command result status: Error" << endl;
          else
                cout << "PRESET_TELESCOPE command result status: OK" << endl;

          cout << "PRESET_TELESCOPE command result: ";
          for(int i=1; i<=cmdRet.getResultCount(); i++)
                cout << cmdRet.getResultDescription(i) << endl;

          delete statusInf;
    }
    else
    {
          cout << "PresetTelescope execution failed: " <<
                  aResult->GetresultString() << endl;
    }

    delete aResult; /* free memory for CSQHandle */
    delete offset;
    delete guidestars;

    //------------- Tell telescope that the instrument wants to idle -------------
    cout << "Deauthorize" << endl;

   // Call Deauthorize(), this indicating that the observation has finished.

    aResult = anIIF->Deauthorize(SIDE_LEFT);

    if (aResult->GetresultCode() != RESCODE_FAIL)
    {
          StatusInfo* statusInf =
                      anIIF->GetCommandStatus(aResult->GetcommandHandle());
          statusInf->Block(); /* wait for TCS */

          cout << "DEAUTHORIZE command status: "
                << statusInf->GetCommandStatus() << endl;

          cout << "DEAUTHORIZE command result: "
                << statusInf->GetCommandResult() << endl;

          delete statusInf;

          }
          else
```

```
                {
                        cout << "Deauthorize execution failed: "
                                << aResult->GetresultString() << endl;
        }

        delete aResult; /* free memory for CSQHandle */

//---------------------------------- Lead out ----------------------------------
// Clean up after yourself when done

        delete anIIF;

// Return an exit code indicating that everything went OK.

        return 0;
}
```

9 References

[1] J. Borelli, T. Schmelmer, T. Axelrod, M. Wagner, “Command Set at the Instrument -Telescope Interface for the LBT”, LBT document CAN 481g010d (Draft: August, 2006)

[2] C. Biddick, A. Lovell-Troy, “Instrument Team IIF Testing Instructions”, LBT document CAN 481s265a (Draft: July, 2006)

[3] L. Fini, L. Busoni, A. Puglisi, “AOS Functional Description”, LBT document CAN 486f006b (December, 2006)

[4] T. Sargent, “Instrument Rotator Control SW Specification”, LBT document CAN 678s001f (December, 2006)

[5] J. Borelli, T. Schmelmer, “Instrument Interface User Guide”, LBT document CAN 481s261b (August, 2006)

[6] W.Gaessler, M. Kuerster, “Linc Nirvana-Telescope Control Software”, LN-MPIA-SWDR-ICS-010 (December, 2006)

[7] R. Pogge, “MODS and LBT TCS Use Case description”, OSU-MODS-2006-002 (January, 2007)

[8] M. Juette, V. Knierim, “LUCIFER Use cases Description” (January, 2007)

[9] V. Vaitheeswaran, “LBTI and LBT use case document”, LBTI-SW-100 (February, 2007)

[10] I. Ilyin, M. Andersen, “Use case for PEPSI at the LBT” (March, 2007)

[11] R. Noll, "Zernike Polynomials and Atmospheric Turbulence", J. Opt. Soc. Am., Vol 66, No. 3 (March 1976).

[12] G. Gibson, C. Biddick, “Common Software Description” LBT document CAN 481s501a (July, 2006)

Appendix A : Global definitions

File: `LBT/IIFGlobal.h`

Definition of the focal station ID for the prime focus
```
#define FS_PRIME "prime"
```

Definition of the focal station ID for the direct Gregorian focus
```
#define FS_DIRECTGREGORIAN "directGregorian"
```

Definition of the focal station ID for the bent Gregorian focus (back)
```
#define FS_BENTGREGBACK "bentGregorianBack"
```

Definition of the focal station ID for the bent Gregorian focus (front)
```
define FS_BENTGREGFRONT "bentGregorianFront"
```

Definition of the focal station ID for the bent Gregorian focus (center)
```
#define FS_BENTGREGCENTER "bentGregorianCenter"
```

Definition of the focal station ID for the fiber feed in the bent Gregorian focus
```
#define FS_BENTGREGREARFIBER "bentGregorianRearFiberFeed"
```

Definition of the instrument ID for the first LUCIFER
```
#define IS_LUCIFER1 "LUCIFER1"
```

Definition of the instrument ID for the second LUCIFER
```
#define IS_LUCIFER2 "LUCIFER2"
```

Definition of the instrument ID for a virtual LUCIFER instrument, controlling both sides
```
#define IS_LUCIFER "LUCIFER"
```

Definition of the instrument ID for LINC/NIRVANA
```
#define IS_LINC "LINC"
```

Definition of the instrument ID for LBTI
```
#define IS_LBTI "LBTI"
```

Definition of the instrument ID for the first PEPSI in direct Gregorian
```
#define IS_PEPSI1 "PEPSI1"
```

Definition of the instrument ID for the second PEPSI in direct Gregorian
```
#define IS_PEPSI2 "PEPSI2"
```

Definition of the instrument ID for a virtual PEPSI instrument, controlling both sides
```
#define IS_PEPSI "PEPSI"
```

Definition of the instrument ID for the first PEPSI in the bent Gregorian
```
#define IS_PEPSIFIBER1 "PEPSIFIBER1"
```

Definition of the instrument ID for the second PEPSI in the bent Gregorian
```
#define IS_PEPSIFIBER2 "PEPSIFIBER2"
```

Definition of the instrument ID for a virtual PEPSIFIBER instrument, controlling both sides
```
#define IS_PEPSIFIBER "PEPSIFIBER"
```

Definition of the instrument ID for the first MODS
```
#define IS_MODS1 "MODS1"
```

Definition of the instrument ID for a combined MODS
```
#define IS_MODS2 "MODS2"
```

Definition of the instrument ID for a virtual MODS instrument, controlling both sides
```
#define IS_MODS "MODS"
```

Definition of the instrument ID for red-channel LBC
```
#define IS_LBCRED "LBCRED"
```

Definition of the instrument ID for blue-channel LBC
```
#define IS_LBCBLUE "LBCBLUE"
```

Definition of the instrument ID for a virtual LBC instrument, controlling both sides
```
#define IS_LBC "LBC"
```

Definition of the instrument ID for MAT
```
#define IS_MAT "MAT"
```

Definition of the strings specifying the left telescope side
```
#define SIDE_LEFT "left"
```

Definition of the strings specifying the right telescope side
```
#define SIDE_RIGHT "right"
```

Definition of the strings specifying both telescope sides
```
#define SIDE_BOTH "both"
```

String constant that is returned by a call in IIF::Authorize, in case the IIF instance has successfully registered with the CSQ
```
#define INSTANCE_AUTHORIZED "AUTHORIZED"
```

/** Enum type representing the optical elements (OPE)*/
```
enum opeType
{
      M1, M2, M3, M1M2, M1M3, M2M3, M1M2M3, MOUNT, HEXAPOD, DEFAULT
};
```

/** Enum type representing the type of movement: relative or absolute */
```
enum moveType
{
      MV_REL, MV_ABS
};
```

/** Enum type representing the operational mode of the telescope*/
```
enum modeType
{
       MODE_STATIC,           /** Static mode */
       MODE_TRACK,             /** Passive Mode */
       MODE_GUIDE,             /** Guided Mode */
       MODE_ACTIVE,            /** Active Mode */
       MODE_ADAPTIVE,          /** Adaptive Mode */
       MODE_INTERFEROMETRIC    /** Interferometric Mode */
};
```

/** Enum type representing the AO mode*/

```
enum AOmodeType
{
       FIX_AO,
       TTM_AO,
       ACE_AO,
       ICE_A
};
```

/** Enum type representing the Coordinate system */

```
enum coordType
{
     COORD_RADEC_SKY,      /** Right Ascension, Declination, on the sky*/
     COORD_RADEC_FOCAL,    /** Right Ascension, Declination, on the focal plane */
     COORD_ALTAZ,          /** Altitude, Azimuth */
     COORD_FOCAL_PIX,      /** Focal Plane in Pixels*/
     COORD_FOCAL_MM        /** Focal Plane in millimeters*/
};
```

/** Enum type representing the instrument's rotator mode */

```
enum rotatorType
{
      ROTATOR_PSTN,     /** Instrument rotator's in position mode */
      ROTATOR_PAR,      /** Instrument rotator's vertical mode, parallactic ang. */
      ROTATOR_NATIVE,   /** To use the rotator's native reference frame.*/
      ROTATOR_GRAV,     /** To use the gravitational angle. */
      ROTATOR_IDLE      /** Instrument rotator's idle mode */
};
```

Appendix B : TCS-IIF commands status

IIF Command	Section	Status	Comments
AOPreset	7.1	Pending input	
AOAcquireRef	7.2	Pending input	
AORefine	7.3	Pending input	
AOStart	7.4	Pending input	
AOOffsetXY	7.5	Pending input	
AOOffsetZ	7.6	Pending input	
AOCorrectModes	7.7	Pending input	
AOStop	7.8	Under development	
AOPause	7.9	Under development	
AOResume	7.10	Under development	
AOUserPanic	7.11	Under development	
Authorize	7.12	Operational	
CancelCommand	7.13	Operational	Most TCS commands do not support cancel
Deauthorize	7.14	Operational	
GetCommandStatus	7.15	Operational	
GetMultiParameter	7.16	Operational	
GetParameter	7.17	Operational	
GetRotatorTrajectory	7.18	Operational	At the moment, it is only used by LBC
LogEvent	7.19	Operational	
Move	7.20	Under development	
MoveFocus	7.21	Operational	Only supports M1
MoveXY	7.22	Operational	Only supports M1
MoveXYZ	7.23	Under development	
OffsetGuiding	7.24	Operational	
OffsetPointing	7.25	Pending input	There is an operational version for LBC, but it has to be modified in order to support other instruments
PauseGuiding	7.26	Under development	
PresetGuiding	7.27	Pending input	
PresetTelescope	7.28	Partly Operational	
ResumeGuiding	7.29	Under development	

RotateCommon	7.30	Under development	
RotatePrimary	7.31	Operational	
RotateZ	7.32	Under development	
RotAdjustPosition	7.33	Under development	
RotHold	7.34	Under development	
RotMaximizeTime	7.35	Under development	
RotServicePosition	7.36	Under development	
RotSetRotator	7.37	Under development	
RotTrack	7.38	Under development	
RotNextPosition	7.39	Under development	
SendWavefront	7.40	Operational	Only supports M1 active optics
SetMultiParameter	7.41	Operational	
SetParameter	7.42	Operational	
Standby	7.43	Under development	
StartGuiding	7.44	Undefined	
StepFocus	7.45	Operational	Only supports M1
StopGuiding	7.46	Under development	
TelescopeMove	7.47	Under development	
TelescopeRotate	7.48	Under development	
TelescopeScale	7.49	Under development	
TipTilt	7.50	Operational	Only supports M1
UpdateGuidestar	7.51	Undefined	

--oOo --

Doc_info_start

Title: *Instrument-Telescope Control Document*

Document Type: *Technical Manual*

Source: *Steward Observatory*

Issued by: *Jose L. Borelli*

Date_of_Issue: *15-Dec-2007*

Revised by:

Date_of_Revision: *20-Mar-2007*

Checked by: *Norm Cushing*

Date_of_Check:

Accepted by:

Date_of_Acceptance:

Released by:

Date_of_Release:

File Type: *Open Office Writer*

Local Name: *Instrument-Telescope Control Document*

Category: *400*

Sub-Category: *480*

Assembly: *481*

Sub-Assembly:

Part Name:

CAN Designation: *481s011*

Revision: *c*

Doc_info_end