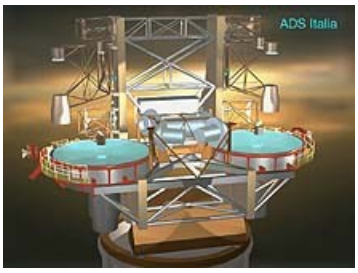

LBT PROJECT
2x8.4m TELESCOPE

Doc. No. : 481s013
Issue : v
Date : July 5, 2016
Issued by : Chris Biddick

LBT PROJECT

2 X 8.4m OPTICAL TELESCOPE

ICE Instrument Interface Control Document

	Signature	Date
Prepared	Jose Borelli	January 19, 2009
Reviewed	Chris Biddick	April 13, 2009
Approved		

1 Revision history

Issue	Date	Changes	Responsible
a	19-Jan-2009	First draft	Jose Borelli
b	13-Apr-2009	Converted from OpenOffice to MS Word. Edited for both form and content. Reorganized sections. Removed unnecessary material. Added additional material.	Chris Biddick
c	11-Dec-2009	Modified Rot commands; added AO commands	Chris Biddick
d	15-Feb-2010	Added SetPMTerm and guiding commands	Chris Biddick
e	13-Apr-2010	Added ClearHotspot, ClearOffset, PresetClear	Chris Biddick
f	07-Jul-2010	Rename PresetClear to BinocularControl, clarify side parameter, clarify default values, add Binocular operations section.	Chris Biddick
g	26-Oct-2010	Remove X and Y coordinates from RotateCommon rotation point. The angle is now in micro-radians. Update command status table. Update target coordinates and rotator coordinates descriptions. Fix syntax for OffsetPointing. Modified BinocularControl description.	Chris Biddick
h	26-May-2011	Correct filter and color in various examples.	Chris Biddick
i	23-Sep-2011	Correct BinocularControl description. Correct GetRotatorTrajectory description. Add non-sidereal target description. Add commands ClearNonSidereal, SetNonSidereal, and UpdateNonSiderealTarget. Modify RunAO description. Improve returned result description.	Chris Biddick
j	25-Oct-2011	Change JD to MJD in nonsidereal struct. Add complex adaptive telescope modes.	Chris Biddick
k	29-Feb-2012	Change ICE factory create function to redefine an existing proxy.	Chris Biddick
l	26-Sep-2012	Change remove command to Remove. Correct ClearNonSidereal description. Remove type and file name from UpdateNonSiderealTarget. Add GetKFPCoordinates and GetRotatorPolynomials commands.	Chris Biddick
m	06-Mar-2013	Allow using public names in SetParameter command. Correct Spelling of “AQUIRE”. Correct units description in SetOffset.	Chris Biddick
n	24-May-2013	Correct link to TCS status on the wiki. Update BinocularControl description. Add SetGuidingHotspot command. Note case insensitivity of parameters.	Chris Biddick

o	26-Sep-2013	Add filter and colorType definitions, update target coordinate systems.	Chris Biddick
p	12-Feb-2014	Add SetAGWFilter command.	Chris Biddick
q	14-Nov-2014	Clarify Offset command.	Chris Biddick
r	22-Jan-2015	Add newposition, SetStarsNew, GetKFPCoordinatesNew and SetReferenceNew.	Chris Biddick
s	01-May-2015	Clarify proper motion: coordinate 1 includes cos(DEC). Add SetOffsetNew. Update description for OffsetPointing.	Chris Biddick
t	04-Sep-2015	Rename the Offset “SKY” attribute “CS”. Remove “LGS” telescope modes.	Chris Biddick
u	23-Mar-2016	Add SetPMTerm2 command. Add OffsetPointing2 command. Rename ‘xxxNew’ commands to ‘xxx2’. Change position2 structure.	Chris Biddick
v	05-Jul-2016	Correct link in reference 13. Update pseudo-monocular and BinocularControl descriptions. Add SetGuidingBinning command.	Chris Biddick

2 Table of contents

3 List of abbreviations

Abbreviation	Description
AGW	Acquisition, Guiding, and Wavefront sensing
ALT	Altitude
AMI	Asynchronous Method Invocation
AO	Active Optic
AOS	Adaptive Optic Subsystem
AOS-Sup	Adaptive Optics Supervisor
AZ	Azimuth
CSQ	Command Sequencer
DEC	Declination
GCS	Guiding Control Subsystem
GUI	Graphic User Interface
ICE	Internet Communication Engine
ICS	Instrument Control Software
IIF	Instrument Interface
IRA	Initial Rotation Angle
LBT	Large Binocular Telescope
LBTI	Large Binocular Telescope Interferometer
LBTO	Large Binocular Telescope Observatory
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
TO	Telescope Operator
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 ICE Instrument-Telescope control software interface, the classes and structures involved, 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 ICE Instrument Interface library to communicate with the TCS, describing the set of commands that LBT will provide to all the instruments. The C and C++ interfaces are described in another document; see reference [1].

4.2 Notes

The document is 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 8 is in alphabetical order. At the moment, some of these commands are only proposed and they may change in the future. Appendix A and the following WIKI page show the most recent implementation notes and the operational status of the set of commands

http://wiki.lbto.org/bin/view/Software/IIFSupportStatus

5 Using the IIF

5.1 Introduction

The Ice Instrument Interface is distributed to the LBT instrument software teams and must be compiled using the ZeroC tools in order to generate the correspondent APIs. This allows the instruments to use heterogeneous environments: client and server can be written in different programming languages, can run on different operating systems and machine architectures, and can communicate using a variety of networking technologies.

The Ice interface supports synchronous and asynchronous calls. This means an instrument can send a command to the TCS and wait for the results, or it can use callbacks in order to do other tasks while the first one is in progress.

Multiple instruments can be connected at the same time, and in turn, each instrument instance supports multiple connections. This makes the interface extremely flexible.

The following section gives detailed information concerning the programmatic use of the Ice Instrument Interface, describing a walk-through of the coding steps that should be followed by the instrument teams when using the IIF libraries. See section 9 for more details.

5.2 Getting ICE

In order to communicate with the TCS Ice Instrument Interface, we need to install the Zeroc Ice run-time libraries and the developer kits. Both can be downloaded from the Zeroc website www.zeroc.com. Ice can be also installed using yum or up2date on CentOS, creating the following repository description and installing it in /etc/yum.repos.d:

```
[zeroc-ice]
name=Ice 3.2 for Red Hat Enterprise Linux $releasever - $basearch
baseurl=http://www.zeroc.com/download/Ice/3.2/rhel4/$basearch
enabled=1
gpgcheck=1
gpgkey=http://www.zeroc.com/download/RPM-GPG-KEY-zeroc-release
```

Then, just type:

```
# yum install ice.i386 ice-c++-devel.i386 ice-java-devel.i386
```

Once installed, you can view the available packages using the following command:

```
yum list ice* db45*
```

The LBTO configuration manager should be consulted before ICE is installed since he may have particular versions that are currently supported.

5.3 The ICE Architecture

5.3.1 Introduction

Ice is an object-oriented middleware platform. Fundamentally, this means that Ice provides tools, APIs, and library support for building object-oriented client–server applications. The source code for these applications is portable regardless of the deployment environment.

5.3.2 Terminology

Every computing technology creates its own vocabulary as it evolves. Ice is no exception. However, the amount of new jargon used by Ice is minimal. Rather than inventing new terms, we have used existing terminology as much as possible. If you have used another middleware technology, such as CORBA, in the past, you will be familiar with most of what follows.

5.3.3 Clients and Servers

The terms client and server are not firm designations for particular parts of an application; rather, they denote roles that are taken by parts of an application for the duration of a request:

- Clients are active entities. They issue requests for service to servers.
- Servers are passive entities. They provide services in response to client requests.

Frequently, servers are not "pure" servers, in the sense that they never issue requests and only respond to requests. Instead, servers often act as a server on behalf of some client but, in turn, act as a client to another server in order to satisfy their client's request.

Similarly, clients often are not "pure" clients, in the sense that they only request service from an object. Instead, clients are frequently client–server hybrids. For example, a client might start a long-running operation on a server; as part of starting the operation, the client can provide a callback object to the server that is used by the server to notify the client when the operation is complete. In that case, the client acts as a client when it starts the operation, and as a server when it is notified that the operation is complete.

Such role reversal is common in many systems, so, frequently, client–server systems could be more accurately described as peer-to-peer systems.

5.3.4 Ice Objects

An Ice object is a conceptual entity, or abstraction. An Ice object can be characterized by the following points:

- An Ice object is an entity in the local or a remote address space that can respond to client requests.
- A single Ice object can be instantiated in a single server or, redundantly, in multiple servers. If an object has multiple simultaneous instantiations, it is still a single Ice object.
- Each Ice object has one or more interfaces. An interface is a collection of named operations that are supported by an object. Clients issue requests by invoking operations.
- An operation has zero or more parameters as well as a return value. Parameters and return values

have a specific type. Parameters are named and have a direction: in-parameters are initialized by the client and passed to the server; out-parameters are initialized by the server and passed to the client. (The return value is simply a special out-parameter.)

- An Ice object has a distinguished interface, known as its main interface. In addition, an Ice object can provide zero or more alternate interfaces, known as facets. Clients can select among the facets of an object to choose the interface they want to work with.
- Each Ice object has a unique object identity. An object's identity is an identifying value that distinguishes the object from all other objects. The Ice object model assumes that object identities are globally unique, that is, no two objects within an Ice communication domain can have the same object identity.

In practice, you need not use object identities that are globally unique, such as UUIDs, only identities that do not clash with any other identity within your domain of interest.

5.3.5 Proxies

For a client to be able to contact an Ice object, the client must hold a proxy for the Ice object. A proxy is an artifact that is local to the client's address space; it represents the (possibly remote) Ice object for the client. A proxy acts as the local ambassador for an Ice object: when the client invokes an operation on the proxy, the Ice run time:

1. Locates the Ice object
2. Activates the Ice object's server if it is not running
3. Activates the Ice object within the server
4. Transmits any in-parameters to the Ice object
5. Waits for the operation to complete
6. Returns any out-parameters and the return value to the client (or throws an exception in case of an error)

A proxy encapsulates all the necessary information for this sequence of steps to take place. In particular, a proxy contains:

- Addressing information that allows the client-side run time to contact the correct server
- An object identity that identifies which particular object in the server is the target of a request
- An optional facet identifier that determines which particular facet of an object the proxy refers to

5.3.6 Stringified Proxies

The information in a proxy can be expressed as a string. For example, the string

SimplePrinter:default -p 10000

is a human-readable representation of a proxy. The Ice run time provides API calls that allow you to convert a proxy to its stringified form and vice versa. This is useful, for example, to store proxies in database tables or text files.

Provided that a client knows the identity of an Ice object and its addressing information, it can create a proxy "out of thin air" by supplying that information. In other words, no part of the information inside a proxy is considered opaque; a client needs to know only an object's identity, addressing information, and (to be able to invoke an operation) the object's type in order to contact the object.

5.3.7 Asynchronous Method Invocation

Ice also supports asynchronous method invocation (AMI): clients can invoke operations asynchronously, that is, the client uses a proxy as usual to invoke an operation but, in addition to passing the normal parameters, also passes a callback object and the client invocation returns immediately. Once the operation completes, the client-side run time invokes a method on the callback object passed initially, passing the results of the operation to the callback object (or, in case of failure, passing exception information).
The server cannot distinguish an asynchronous invocation from a synchronous one—either way, the server simply sees that a client has invoked an operation on an object.

5.4 The IIF Factory

The IIF Factory is the interface which controls the client connections, creating new IIF objects when needed, linking the existing ones with its correspondent client proxy object, and finally, releasing the client proxies when the connections are closed. Its main functions are:

```
IIFServer* create(string proxyName, string focalStation, string instrumentID);
```

Creates or links an ICE proxy with the client connection. The attributes are combined to generate a unique proxy id inside the factory. If proxyName already exists, but the proxy id is different, the old proxy is destroyed and a new one created with the new parameters. If proxyName already exists with the same proxy id, the proxy is linked to the connection. If the proxyName does not exist, a new proxy is created. Note that the focalStation and instrumentID strings are *not* case sensitive.

```
void destroy(IIFServer* proxy);
void destroyProxy(string proxyName);
```

Releases the proxy object and removes the entry from the factory. The first form is used to destroy a proxy created by the caller. The second form allows *any* proxy to be destroyed and should be used with great caution.

```
string getProxy(string proxyName);
```

Returns the factory internal proxy id. An empty string is returned if the proxy name does not exist.

```
string getProxyInstrument(string proxyName);
```

Returns the instrument ID associated with the proxy name. An empty string is returned if the proxy name does not exist.

```
string getProxyFocalStation(string proxyName);
```

Return the focal station name associated with the proxy name. An empty string is returned if the proxy name does not exist.

Compile the interface using the Zeroc developer tools:

```
slice2cpp Factory.ice
```

This will generate two files, Factory.cpp and Factory.hpp. Include them in your code and follow the next steps. See also the full example in section 9

5.5 Example

Create the communicator (which contains the main handle to the Ice run time) and get a proxy object for your instrument:

```
Ice::CommunicatorPtr communicator;
try
{
      // Initialize the Ice communicator.
      communicator = Ice::initialize(argc, argv);
      // Create a connection to the IIF Factory. i.e. TCP Port 10000.
      // This should be in a configuration file. See the Ice Documentation [14].
      FactoryPrx factory =
      FactoryPrx::checkedCast(communicator->stringToProxy("Factory:tcp -p 10000"));
      if(!factory) throw "Invalid proxy: IIF Server not found.";
```

Use the factory to create a new IIF instance. See table 6.1 for valid focal station and instrument names.

```
      // Get a proxy object for this instrument.
      // If the proxy already exists with the same instrumentID and focalStation,
      // the arbitrator will link the proper IIF instance with it. If the proxy
      // already exists but not for the same instrumented/focalStation, the old
      // proxy is destroyed and a new one created.
      // Otherwise, it will create a new IIF instance for this client.
      iifs::IIFServerPrx iif = factory->create(CLIENT_PROXY_NAME, FOCAL_STATION,
      INSTRUMENT_ID);
      if ( !iif ) throw "Invalid proxy: Invalid instrument/focal station
      combination or invalid side.";
```

Check for authorization.

```
// Check if we are authorized
res = iif->Authorize();
//Here print out the error messages coming from IIF-TCS, then throw an
exception.
if (res.rescode != EXIT_SUCCESS) throw "Error: Instrument not authorized";
```

Use the command set

```
// TipTilt
res = iif->TipTilt(0.0009, 0.0001, "M1", “left”);
//Here you must analyze the TCS results. See next topic.
//MoveXY
res = iif->MoveXY(0.0009, 0.0001, "M1", “both”);
```

Analyze the command result

```
if (res.rescode == 0) cout << " command status: SUCCESS" << endl;
else if (res.rescode == 1) cerr << " command status: ERROR" << endl;
else cerr << " command status: WARNING" << endl;
for ( unsigned int i=0; i < res.resmsg.size(); i++)
cout << res.resmsg[i] << endl;
}
```

6 TCS fundamentals

All commands to the TCS have elements in common. The most obvious are the instrument name, focal station name, and the side. Only certain combinations are allowed; these are listed in table 6.1.

Parameters passed to the TCS through the ICE interface are not case sensitive, but this document often uses case for clarity. Inside the IIF instrument names and parameter strings are kept in upper case, while focal stations and sides are kept as lower case.

6.1 Valid instrument and focal station names

Instrument name	Focus	Side
LUCIFER	bentGregorianFront	left | right | both
LINC	bentGregorianBack	left | right | both
LBTI	bentGregorianCenter	left | right | both
PEPSIPOL	directGregorian	left | right | both
PEPSIPFU	bentGregorianRearFiberFeed	left | right | both
MODS	directGregorian	left | right | both
LBC	Prime	left | right | both
LAT	Prime	left
DIMM	Prime	right
IRTC	directGregorian or bentGregorianFront or bentGregorianBack or bentGregorianCenter	left | right | both

Table 6.1: list of valid instrument name/focal station combinations.

6.2 Instrument authorizing

Under normal operating conditions, only a single instrument can have full control of a telescope side; and this control is obtained through an IIF GUI controlled by the TO. 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. An IIF Authorize command exists which allows an instrument to check if it is currently authorized. In the command set descriptions in section 8 the requirement for authorization is listed in the “Preconditions” section.

6.3 Telescope status

The instruments are able to ask for telescope status. A status request could be on remaining ranges of optical elements (primary, secondary, etc.), actual coordinates the telescope is pointing to, enclosure (open or closed), ambient temperature, etc. All this information is in the TCS Data Dictionary, and is updated periodically by the TCS subsystems. It is only updated when the subsystems are running, so it

may not be current. The parameters an instrument is allowed to query are specified in CAN 481s015.

An instrument is also 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. Note that an instrument may only set parameters in the data dictionary that belong to it, and the parameters must all be predefined.

6.4 Coordinate systems

The telescope physical coordinate system is a right-handed system with Z toward the sky, X to the left, and Y toward the back of the telescope.

This defines the positive X, Y and Z directions for command that take such parameters (i.e., Move). Rotations about these axes are all right-handed.

6.4.1 Target coordinates

Target coordinates are used in the SetStars command for both the target and the guide stars. Note that the target and guide stars are not required to use the same system. The target coordinate systems are

- "RADEC" equatorial coordinates on the sky
- "AZALT" AZ/ALT coordinates
- "GALACTIC" galactic coordinates

6.4.2 Offset coordinates

Offset coordinates are used in the OffsetPointing command. The offset coordinate systems are

- "RADEC" RA/DEC coordinate system
- "AZALT" AZ/ALT coordinate system
- "DETXY" focal plane coordinate system
- "GALACTIC" galactic coordinate system

6.4.3 Rotator coordinates

The rotators have several modes. They are

- "POSITION" relative to the north
- "PARALLACTIC" vertical with respect to the horizon
- "IDLE" no rotator control

6.5 Operating modes

The TCS has the concept of "Mode" at several levels which govern many systems and behaviors.

6.5.1 Telescope modes

The telescope mode is set by the PresetTelescope command. The modes are

- "NONE" This indicates no active Preset, and cannot be set by a Preset.
- "STATIC" This moves the telescope to the given target, with no motion after that.
- "TRACK" This moves the telescope to the given target, and then starts sidereal tracking on the target.
- "ACQUIRE" This moves the telescope to the given target, starts tracking, and starts the GCS to take continuous guider acquisition images. This is normally used for GCS testing and engineering, but may also be used to adjust the pointing at the start of the night.
- "GUIDE" Moves the telescope to the given target, starts tracking, sends the provided guide star list to the GCS, and starts guiding.
- "ACTIVE" Moves the telescope to the given target, starts tracking, sends the provided guide star list to the GCS, starts guiding, and then starts active optics.
- "ADAPTIVETTM_TRACK"
  "ADAPTIVETTM_GUIDE"
  "ADAPTIVETTM_ACTIVE" Adds Tip/Tilt adaptive optics to TRACK, GUIDE or ACTIVE modes.
- "ADAPTIVEACE_TRACK"
  "ADAPTIVEACE_GUIDE"
  "ADAPTIVEACE_ACTIVE" Adds auto configured adaptive optics to TRACK, GUIDE, or ACTIVE modes.
- "ADAPTIVEICE_TRACK"
  "ADAPTIVEICE_GUIDE"
  "ADAPTIVEICE_ACTIVE" Adds interactive configured adaptive optics to TRACK, GUIDE, or ACTIVE modes.
- "INTERFEROMETRIC" Moves the telescope to the given target, starts tracking, sends the provided guide star list to the GCS, starts guiding, starts adaptive optics, and then performs operations needed by the interferometers. It is not currently supported, and may never be.

The modes are in order of increasing functionality (and complexity), and the first six modes include the functionality of the modes below it. The adaptive and interferometric modes build on simpler modes but not in a simple ordered way.

6.5.2 AO system operating modes

When the telescope mode is ADAPTIVE or higher, the AOS also has modes that govern how it behaves. These are

- "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 the 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 the full AO corrected mode where the observer is given the possibility to adjust AO loop parameters.

6.6 Binocular operations

When operating in full binocular mode, there is a restriction on how the two sides may point. The "co-pointing limit" is expected to be about ±20 arc-sec (but may change in the future) and is the maximum deviation from the "mount point". The "mount point" is the position of the mount, and is a compromise between the two sides.

6.6.1 Monocular operation

This is set by only authorizing an instrument on one side of the telescope (the other side is "None"). In this mode commands may only be issued for the correct side and all pointing changes are performed by the mount; the co-pointing limit does not apply.

6.6.2 Synchronous operation

In "synchronous" operation, the TCS will expect two BinocularControl commands, one from each side, followed by two PresetTelescope or OffsetPointing commands, and will not process the PresetTelescope or OffsetPointing commands until both have been received. Other commands will continue to be processed as they are received. The mount point will be computed by the weighted average of the two requests, (weighting is set by the PCS subsystem), and the destination position is not limited by the co-pointing limit. The optics may be adjusted to produce the requested pointing.

6.6.3 Asynchronous operation

In "asynchronous" operation, no BinocularControl commands are used (but see below), and only one PresetTelescope or OffsetPointing command is accepted at a time; others are held off until the telescope is not busy. Each command is subject to the co-pointing limit, and all motion is achieved by moving the optics on that side. This is to satisfy the requirement that an operation on one side must not disturb operations on the other side. As a special case, the BinocularControl CLEARPRESET command conditions the TCS to allow an asynchronous PresetTelescope to move the mount without regard to the co-pointing limit.

6.6.4 Pseudo-monocular operation

"Pseudo-monocular" operation is implemented for two reasons: One, to allow a binocular instrument observing operation to continue if one side of the telescope is not working, and two, to support "parasitic"

observing, where one instrument drives the telescope, and another (typically the LBC) takes data wherever the telescope is pointed. In "pseudo-monocular" operation, both sides are authorized and one side is "active" and one side is "passive". All SYNCPRESET and SYNCOFFSET BinocularControl commands are ignored (and return with a warning), all Preset and Offset commands from the "active" side are copied to the "passive" side as synchronous commands (so the telescope always moves and the co-pointing limit is never violated), and side "both" is converted to the "active" side. Preset and RunAO commands from the "passive" side are ignored (and return with a warning), while Offset commands from the "passive" side are optionally processed normally (for "parasitic" observing), otherwise they are also ignored (and return with a warning). This mode is normally selected on the IIFGUI. Turning off the mode reverts back to normal binocular operation.

6.6.5 Non-sidereal targets

Tracking mode and non-sidereal targets must be the same for both sides of the telescope. This means that when switching between sidereal and non-sidereal targets using asynchronous PresetTelescope commands, the first one *must* be preceded by a BinocularControl CLEARPRESET command. The second one must have the same non-sidereal target as the first. In monocular mode switching between sidereal and non-sidereal targets has no constraints.

6.6.6 BinocularControl flags

There are several internal TCS states that must be set correctly for proper binocular operation. These include allowing the telescope to move beyond the co-pointing limit, informing the TCS of synchronous commands, and various modifiers of these states Table 6.2 lists the BinocularControl command flags.

Flag	Description
CLEARPRESET	Tells the TCS that the next PresetTelescope command is allowed to move the telescope beyond the co-pointing limit in asynchronous mode. Only one CLEARPRESET command from any side need be issued. This does not affect the current telescope operation.
SYNCPRESET	Tells the TCS to expect two PresetTelescope commands, one for each side, and to allow the telescope to move beyond the co-pointing limit in synchronous mode. Two SYNCPRESET commands must be issued, one from each side. This does not affect the current telescope operation.
SYNCOFFSET	Tells the TCS to expect two OffsetPointing commands, one for each side, and to allow the telescope to move beyond the co-pointing limit in synchronous mode. Two SYNCOFFSET commands must be issued, one from each side. This does not affect the current telescope operation.
ADJUSTBALANCE	Tells the TCS to rebalance the mount point so the optics on both sides are positioned as near their centers of travel as possible. Only one ADJUSTBALANCE command from any side need be issued. This does affect the current telescope operation.
CANCELSYNCPRESET	Tells the TCS to cancel any current SYNCPRESET condition and puts the TCS into asynchronous Preset mode. Only one CANCELSYNCPRESET command from any side need be issued. This does affect the current telescope operation.
CANCELSYNCOFFSET	Tells the TCS to cancel any current SYNCOFFSET condition and puts the TCS into asynchronous Offset mode. Only one CANCELSYNCOFFSET command from any side need be issued. This does affect the current telescope operation.
RELEASESYNCPRESET	Tells the TCS to allow a pending synchronous Preset to be processed without waiting for the second Preset on the other side. The Preset will be processed as an asynchronous Preset and will be subject to the co-pointing limit. If a synchronous Preset is waiting for PCS target processing the PCS wait will be released and the target will be used for both sides. This does affect the current telescope operation.
RELEASESYNCOFFSET	Tells the TCS to allow a pending synchronous Offset to be processed without waiting for the second Offset on the other side. The Offset will be processed as an asynchronous Offset and will be subject to the co-pointing limit. If a synchronous Offset is waiting for PCS target processing the PCS wait will be released and the target will be used for both sides. This does affect the current telescope operation.
LEFTMONOCULAR	Tells the TCS to go into a special “pseudo-monocular” mode on the left side. In this mode all SYNCPRESET and SYNCOFFSET

Flag	Description
	BinocularControl commands will be ignored, all left Preset and Offset commands will be copied to the right side as synchronous commands, and side "both" will be converted to "left". All right Preset and RunAO commands will be ignored and return with a warning. Right Offset commands may optionally be processed normally (see RIGHTOFFSET below). When the command is issued, any pending left synchronous Preset or Offset will be released for processing. This does affect the current telescope operation.
RIGHTMONOCULAR	Tells the TCS to go into a special "pseudo-monocular" mode on the right side. In this mode all SYNCPRESET and SYNCOFFSET BinocularControl commands will be ignored, all right Preset and Offset commands will be copied to the left side as synchronous commands, and side "both" will be converted to "right". All left Preset and RunAO commands will be ignored and return with a warning. Left Offset commands may optionally be processed normally (See LEFTOFFSET below). When the command is issued, any pending right synchronous Preset or Offset will be released for processing. This does affect the current telescope operation.
LEFTOFFSET	Tells the TCS to allow normal processing of Offset commands from the pseudo-monocular passive side. LEFTOFFSET can only be on if RIGHTMONOCULAR is on, in which case it is the default. This supports "parasitic' observing. If this flag is off, Offset commands will be ignored and return with a warning. This does not affect the current telescope operation.
RIGHTOFFSET	Tells the TCS to allow normal processing of Offset commands from the pseudo-monocular passive side. RIGHTOFFSET can only be on if LEFTMONOCULAR is on, in which case it is the default. This supports "parasitic' observing. If this flag is off, Offset commands will be ignored and return with a warning. This does not affect the current telescope operation.

Table 6.2: BinocularControl flags.

7 Structures

Prior to describing the different IIF commands we will define some important structures the instruments will need.

7.1 result

This structure contains the result information coming from the TCS. It has an integer result code which indicates if the command execution was successful, failed, or finished with a warning, and a vector of messages.

```
sequence<string> SeqRes; //In C++ slang, sequence is a vector.

struct result
{
      int rescode;
      SeqRes resmsg;
};
```

The possible rescode values are

- EXIT_SUCCESS (0) Command completed successfully
- EXIT_FAILURE (1) Command completed with error(s)
- EXIT_WARNING (2) Command completed with warning(s)

The first element of resmsg (resmsg[0])is always a string describing the command status. It contains the command name and the command status. For example the possible status messages for PresetTelescope are:

```
      PresetTelescope result status: OK
      PresetTelescope result status: Warning
      PresetTelescope result status: Error
```

If the command status is SUCCESS, there will be just the OK message. If the command status is either WARNING or FAILURE, addition messages may be present:

```
// check for errors
if (res.rescode == EXIT_SUCCESS) {
      // process any data
      string data = res.resmsg[1]; // get first result
      ...
} else if (res.rescode == EXIT_FAILURE) {
      // print all the error messages
      for(unsigned int i=0; i<res.resmsg.size(); i++)
            cout << res.resmsg[i] << endl; // get ith message
      ...
} else {
      // print all the warning messages
```

```
      for(unsigned int i=0; i<res.resmsg.size(); i++)
            cout << res.resmsg[i] << endl; // get ith message
      ...
}
```

Four commands have data included in the resmsg: GetParameter, GetRotatorTrajectory, GetRotatorPolynomials, and GetKFPCoordinates.

7.1.1 GetParameter data

For GetParameter the data is returned as the 1st through the Nth index of resmsg when the command status is SUCCESS. There is one value per element, and the caller must convert the string value to the desired type. For example

```
      double tempd = atof(res.resmsg[1].c_str());
      int tempi = atoi(res.resmsg[2] .c_str());
```

When the command status is FAILURE, messages will be returned instead of values; one message for each variable not allowed:

```
      if (res.rescode == EXIT_FAILURE) {
            // print all the error messages
            for(unsigned int i=1; i<res.resmsg.size(); i++)
            cout << res.resmsg[i] << endl; // get ith message
      }
```

The error messages are of the form

```
      Restricted : pcs.pointingStatus.target.target_RA.RAString
```

The text after the colon is the name of the failed variable.

7.1.2 GetRotatorTrajectory data

For GetRotatorTrajectory the data is returned as the 1st through the Nth index of resmsg when the command status is SUCCESS. Each element has two values: time and angle. For LBC, the time is in JD and the angle is in radians.

```
      if (res.rescode == EXIT_SUCCESS) {
            // get the data
            for(unsigned int i=1; i<res.resmsg.size(); i++) {
                  double time,angle;
                  istringstream in(res.resmsg[i]);
                  in >> time >> angle; // get ith data pair
            }
      }
```

When the command status is FAILURE, messages will be returned instead of values:

```
      if (res.rescode == EXIT_FAILURE) {
            // print all the error messages
            for(unsigned int i=1; i<res.resmsg.size(); i++)
            cout << res.resmsg[i] << endl; // get ith message
```

```
}
```

7.1.3 GetRotatorPolynomials data

For GetRotatorPolynomials the data is returned as the 1st through the Nth index of resmsg when the command status is SUCCESS. Each element has two, three, or four values: time, angle, velocity, and acceleration/2. The number of returned values depends on the ORDER input parameter. The time is MJD days UT, the angle is in radians, the velocity is in radians/sec, and the acceleration/2 is in radians/sec/sec.

```
if(res.rescode == EXIT_SUCCESS) {
      // get the data
      for(unsigned int i=1; i<res.resmsg.size(); i++) {
            double time,angle,v,aby2;
            istringstream in(res.resmsg[i]);
            in >> time >> angle;
            if(ORDER > 0) in >> v;
            if(ORDER > 1) in >> aby2;
      }
}
```

When the command status is FAILURE, messages will be returned instead of values:

```
if (res.rescode == EXIT_FAILURE) {
      // print all the error messages
      for(unsigned int i=1; i<res.resmsg.size(); i++)
      cout << res.resmsg[i] << endl; // get ith message
}
```

7.1.4 GetKFPCoordinates data

For GetKFPCoordinates (and GETKFPCoordinates2) the data is returned as the 1st through the Nth index of resmsg when the command status is SUCCESS. Each element has two values: x and y. The values are in Kernel Focal Plane coordinates in mm.

```
if (res.rescode == EXIT_SUCCESS) {
      // get the data
      for(unsigned int i=1; i<res.resmsg.size(); i++) {
            double x,y;
            istringstream in(res.resmsg[i]);
            in >> x >> y; // get ith data pair
      }
}
```

When the command status is FAILURE, messages will be returned instead of values:

```
if (res.rescode == EXIT_FAILURE) {
      // print all the error messages
      for(unsigned int i=1; i<res.resmsg.size(); i++)
      cout << res.resmsg[i] << endl; // get ith message
}
```

7.2 data dictionary

This is used to contain data dictionary data. See the link to the most recent list of data dictionary names at

http://wiki.lbto.org/bin/view/Software/TCSsoftware#IIF_Information

```
struct DDstruct
{
      string DDname;    // item name
      string DDkey;     // not used
      string dataType;  // not used
      string comment;   // not used
};
sequence<DDstruct> SeqDD;
```

SeqDD is a vector of DDstruct objects.

Example

```
DDstruct ddt;
SeqDD list;
ddt.DDname = "L_Instrument";
list.push_back(ddt);
ddt.DDname = "L_FocalStation";
list.push_back(ddt);
```

7.3 position

This structure contains the sidereal target and guide star information. It supports only the sidereal part of the IIF position object. Please read reference [1] to understand more about units and ranges of these attributes.

```
struct position
{
      double coord1;               // RA or AZ (radian)
      double coord2;               // DEC or ALT (radian)
      string system;               // "RADEC", "AZALT", (see 6.4.1)...
      double epoch;                // epoch (year)
      string equinox;              // "J2000"
      double pmcoord1;             // proper motion dRA/dt cos(DEC) (radian/yr)
      double pmcoord2;             // proper motion dDEC/dt (radian/yr)
      double apparentMagnitude;    // magnitude
      string filter;               // filter name (see 7.3.1)
      double color;                // color index
      string colorType;            // color type (see 7.3.2)
      float wavelength;            // wavelength (microns)
      string targname;             // not used
};
```

```
sequence<position> SeqPos;
```

SeqPos is a vector of position objects.

7.3.1 filter

This string defines filters:

“U”, “V”, “B”, “R”, “I”, “J”, “H”, “K”, “NONE”

where U, V, B, R, I, J, H, K are color filters and NONE is no filter. An empty string is acceptable for “NONE”.

7.3.2 colorType

This string defines color types:

“U_B”, “B_V”, “H_K”, “B_R”, “R_I”, “J_K”, “NONE”

where the elements represent differences between bands. An empty string is acceptable for “NONE”.

Example

```
position p;
SeqPos list;
p.coord1 = 1.234;
p.coord2 = 0.12345;
p.system = “RADEC”;
p.epoch = 2011.3
p.equinox = “J2000”;
p.pmcoord1 = 0.0;
p.pmcoord2 = 0.0;
p.apparentMagnitude = 12;
p.filter = “U”;
p.color = 0.0;
p.colorType = “U_B”;
p.wavelength = 1.2;
list.push_back(p);
```

7.4 position2

This structure contains the target and guide star information. It fully supports the IIF position object, which means it supports both sidereal and non-sidereal objects and is an extension of the position structure. Note that guide stars currently cannot be non-sidereal objects. Please read reference [1] to understand more about units and ranges of these attributes. If the type is “DIFFERENTIAL” then coord1 and coord2 are the same as value1 and value2 in the nonsidereal structure (section 7.5).

```
struct position2
```

```
{
      string type;                  // object type. See type in 7.5.
      double coord1;                // RA or AZ (radian)
      double coord2;                // DEC or ALT (radian)
      string system;                // "RADEC", "AZALT", (see 6.4.1)...
      double epoch;                 // epoch (year)
      string equinox;               // "J2000"
      double pmcoord1;              // proper motion dRA/dt cos(DEC) (radian/yr)
      double pmcoord2;              // proper motion dDEC/dt (radian/yr)
      double apparentMagnitude;     // magnitude
      string filter;                // filter name (see 7.3.1)
      double color;                 // color index
      string colorType;             // color type (see 7.3.2)
      float wavelength;             // wavelength (microns)
      string file;                  // non-sidereal ephemeris file name
      double time;                  // see value0 in 7.5.
      double RARate;                // see value3 in 7.5.
      double DECRate;               // see value4 in 7.5.
};

sequence<position2> SeqPos2;
```

SeqPos2 is a vector of position2 objects.

7.5 nonsidereal

This structure contains the non-sidereal target information. Please read reference [1] to understand more about units and ranges of these attributes.

```
struct nonsidereal
{
      string type;            // object type. Planet names, "FILE", "DIFFERENTIAL"
      float wavelength;       // effective wavelength (microns)
      string file;            // ephemeris file name for type FILE
      double value0;          // DIFFERENTIAL MJD (UT days)
      double value1;          // DIFFERENTIAL RA (radian)
      double value2;          // DIFFERENTIAL DEC (radian)
      double value3;          // DIFFERENTIAL dRA/dt * cos(DEC) (radian/day)
      double value4;          // DIFFERENTIAL dDEC/dt (radian/day)
};
```

Note the factor of cos(DEC) in the RA tracking rate. For type FILE the file is an ephemeris file from JPL Horizons and must be present in the required place in the TCS.

Examples

```
nonsidereal ns = {"",0.0, "",0.0,0.0,0.0,0.0,0.0};
ns.type = "SATURN";
ns.wavelength = 0.5;

nonsidereal ns = {"",0.0, "",0.0,0.0,0.0,0.0,0.0};
ns.type = "FILE";
```

```
ns.wavelength = 1.0;
ns.file = "1P/Halley8AUG2011";

nonsidereal ns = {"",0.0, "",0.0,0.0,0.0,0.0,0.0};
ns.type = "DIFFERENTIAL";
ns.wavelength = 1.5;
ns.file = "";
ns.value0 = 2455782.5
ns.value1 = 1.2345;
ns.value2 = 0.12345;
ns.value3 = 0.001;
ns.value4 = -0.0005;
```

7.6 wave front

This structure contains an array of Zernike polynomial coefficients to be sent to the active or adaptive optics control system.

```
sequence<double> SeqWF;       // Z1, Z2, Z3, ... (nanometers)
```

SeqWF is a vector of double.

Example

```
SeqWF wf;
wf.push_back(0.0);      // Z1
wf.push_back(100.0);    // Z2
wf.push_back(-200.0);   // Z3
wf.push_back(1200.0);   // Z4
```

8 IIF command set

8.1 AcquireRefAO

Description

This command is issued to acquire the AO reference star. It is normally used internally by the IIF in the RunAO command. More details in reference [3], section 12.4.

Syntax

```
result AcquireRefAO( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- A PresetAO must have been issued.

After execution

- The reference star is acquired.

Example

```
...
iifs::result res = iif->AcquireRefAO("left");
```

8.2 Authorize

Description

This command returns true if the IIF instance specified by the instrument is currently authorized to control one or both sides of the telescope. This authorization is done by the TO through the IIF GUI.

Syntax

```
result Authorize( )
```

Arguments

- None

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None

After execution

- Instruments know whether they are authorized by the TO.

Example

```
...
iifs::result res = iif->Authorize();
if(res.rescode == EXIT_SUCCESS)
{
// we are authorized
}
```

8.3 BinocularControl

Description

The BinocularControl command is issued to prepare the TCS to provide special binocular handling for PresetTelescope and OffsetPointing commands. See section 6.6 for a discussion of the command flags. The BinocularControl command is ignored if the telescope is in monocular mode; in this case the command has no effect at all, and returns with a warning. This command does not include a side; the side is determined from the sided focal station of the instrument. It must be issued twice (once for each instrument) for SYNCPRESET and SYNCOFFSET, but only once (from either instrument) for CLEARPRESET, CANCELSYNCPRESET, CANCELSYNCOFFSET, ADJUSTBALANCE, RELEASESYNCPRESET, RELEASESYNCOFFSET, LEFTMONOCULAR, RIGHTMONOCULAR, LEFTOFFSET, and RIGHTOFFSET. The last six flags are intended to be used only by the IIFGUI, and the last four are toggles (i.e., each time the command is issued, the state is reversed).

Syntax

```
result BinocularControl( string FLAG )
```

Arguments

- **string FLAG (in)**
  Description: the control flag.
  Unit: none
  Range or possible values: see 6.6.6.

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The TCS is prepared for the next binocular operation.

Example

```
...
iifs::result res = iif->BinocularControl("CLEARPRESET");
```

8.4 CheckRefAO

Description

This command is issued to adjust the telescope pointing to properly position the AO reference star. It is normally requested by the IIF through the AcquireRefAO command in the RunAO command. More details in reference [3], section 12.5.

Syntax

```
result CheckRefAO( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- A PresetAO must have been issued.

After execution

- The reference star is acquired and the telescope pointing adjusted to properly position the star on the pyramid sensor.
- This must be followed by an AcquireRefAO command

Example

```
...
iifs::result res = iif->CheckRefAO("left");
```

8.5 ClearHotspot

Description

ClearHotspot removes the hotspot object set by SetHotspot.

Syntax

```
result ClearHotspot( )
```

Arguments

- **None**

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The hotspot object is removed.

Example

```
...
iifs::result res = iif->ClearHotspot();
```

8.6 ClearNonSidereal

Description

ClearNonSidereal removes the non-sidereal object set by SetNonSidereal.

Syntax

```
result ClearNonSidereal( bool OVERRIDE )
```

Arguments

- **bool OVERRIDE (in)**
  Description: boolean flag to clear the IIF non-sidereal override flag
  This flag is reserved for use by the IRTC and should be false for all other instruments.
  Unit:
  Range or possible values: true | false

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The non-sidereal object is removed.
- If OVERRIDE is true, the non-sidereal override flag in the IIF is cleared.

Example

```
...
iifs::result res = iif->ClearNonSidereal(false);
```

8.7 ClearOffset

Description

ClearOffset removes the offset object set by SetOffset and SetOffset2.

Syntax

```
result ClearOffset( )
```

Arguments

- **None**

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The offset object is removed.

Example

```
...
iifs::result res = iif->ClearOffset();
```

8.8 ClearReference

Description

ClearReference removes the AO reference star set by SetReference and SetReference2.

Syntax

```
result ClearReference( )
```

Arguments

- **None**

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The AO reference star is removed.

Example

```
...
iifs::result res = iif->ClearReference();
```

8.9 ClearStars

Description

ClearStars removes the target and guide stars set by SetStars and SetStars2.

Syntax

```
result ClearStars( )
```

Arguments

- **None**

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The target and guide stars are removed.

Example

```
...
iifs::result res = iif->ClearStars();
```

8.10 CorrectModesAO

Description

This command is used in observation mode to send a vector of modal corrections to the adaptive mirror. More details in reference [3], section 12.11.

Syntax

```
result CorrectModesAO( SeqModes MODES, string SIDE )
```

Arguments

- **SeqModes MODES (in)**
  Description: SeqModes structure containing the correction vector.
  Unit: TBD
  Range or possible values: TBD
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The mirror has the corrections applied.

Example

```
...
SeqModes modes;
modes.push_back(aaaa);
modes.push_back(bbbb);
...
iifs::result res = iif->CorrectModesAO(modes, "left");
```

8.11 GetKFPCoordinates

Description

This command returns the positions in the focal plane for any number of stars. It is intended for instruments like LINC-NIRVANA who have many AO reference stars and need to know where they should be found. The input stars may be passed in through this routine, in which case the position of those stars will be returned. But if LIST is empty, the list of stars specified by SetReference will be used instead. This allows the instrument to avoid sending the same list many times.

Syntax

```
result GetKFPCoordinates      SeqPos LIST
                              string SIDE )
```

Arguments

- **SeqPos LIST (in)**
  Description: the list of position objects for the stars.
  Unit: none
  Range or possible values: [TBD]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The target must be known by the PCS.

After execution

- The command returns x and y positions for each star in the list. x and y are in Kernel Focal Plane coordinates (mm).

Example

```
...
SeqPos list;
list.push_back(position_object_1);
```

```
list.push_back(position_object_2);
list.push_back(position_object_n);
iifs::result res = anIIF->GetKFPCoordinates(list,"left");
if(res.rescode == EXIT_SUCCESS) {
      int n = res.resmsg.size()-1;
      double x[n],y[n];
      for(int i=0; i<n; i++) {
            istringstream in(res.resmsg[i+1]);
            in >> x[i] >> y[i];
      }
} else {
      for(int i=1; i<(int)res.resmsg.size(); i++) {
            cout << res.resmsg[i] << endl;
      }
}
```

8.12 GetKFPCoordinates2

Description

This command returns the positions in the focal plane for any number of stars using a SeqPos2 vector. It is intended for instruments like LINC-NIRVANA who have many AO reference stars and need to know where they should be found. The input stars may be passed in through this routine, in which case the position of those stars will be returned. But if LIST is empty, the list of stars specified by SetReference2 will be used instead. This allows the instrument to avoid sending the same list many times. This command is the same as GetKFPCoordinates but uses the position2 structure.

Syntax

```
result GetKFPCoordinates2     SeqPos2 LIST
                              string SIDE )
```

Arguments

- **SeqPos2 LIST (in)**
  Description: the list of position objects for the stars.
  Unit: none
  Range or possible values: [TBD]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The target must be known by the PCS.

After execution

- The command returns x and y positions for each star in the list. x and y are in Kernel Focal Plane coordinates (mm).

Example

```
...
SeqPos2 list;
```

```
list.push_back(position2_object_1);
list.push_back(position2_object_2);
list.push_back(position2_object_n);
iifs::result res = anIIF->GetKFPCoordinates2(list,"left");
if(res.rescode == EXIT_SUCCESS) {
      int n = res.resmsg.size()-1;
      double x[n],y[n];
      for(int i=0; i<n; i++) {
            istringstream in(res.resmsg[i+1]);
            in >> x[i] >> y[i];
      }
} else {
      for(int i=1; i<(int)res.resmsg.size(); i++) {
            cout << res.resmsg[i] << endl;
      }
}
```

8.13 GetParameter

Description

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

Syntax

```
result GetParameter ( SeqDD MULTIENTRIES )
```

Arguments

- **SeqDD MULTIENTRIES (in)**
  Description: the list of data dictionary entries.
  Unit: none
  Range or possible values: Valid Data Dictionary entries.

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- A SeqDD object must be populated with valid data dictionary entries as defined in the TCS public names list.

After execution

- The command returns a list with the values of the data dictionary entries from the TCS that is stored in the result messages.
- If the command fails, just the failing entries are returned along with the failure reason.

Example

```
...
DDStruct dd, dd1;
SeqDD DDEntries;
dd.DDname = "ChamberTemp";
dd1.DDname = "ChamberDewPoint";
DDEntries.push_back(dd);
DDEntries.push_back(dd1);
iifs::result res = iif->GetParameter(DDEntries);
if(res.rescode == EXIT_SUCCESS) {
      // get data
      double temp,dewPoint;
```

```
      temp = atof(res.resmsg[1].c_str());
      dewPoint = atof(res.resmsg[2].c_str());
} else {
      for(unsigned int i=1; i<res.resmsg.size(); i++) {
            cout << res.resmsg[i] << endl;
      }
}
```

8.14 GetRotatorPolynomials

Description

This command returns a rotator trajectory for the near future. The order of the returned polynomial may be selected as 0, 1, or 2. This is intended for instruments like LINC-NIRVANA which have internal rotators not controlled by the TCS.

Syntax

```
result GetRotatorPolynomials  double STARTTIME
                              int COUNT,
                              double INTERVAL,
                              int ORDER,
                              string SIDE )
```

Arguments

- **double STARTTIME (in)**
  Description: the start time.
  Unit: MJD days
  Range or possible values: [TBD]
- **int COUNT (in)**
  Description: the number of polynomials wanted.
  Unit: none
  Range or possible values: [TBD]
- **double INTERVAL (in)**
  Description: the time interval between polynomials.
  Unit: seconds
  Range or possible values: [TBD]
- **int ORDER (in)**
  Description: the polynomial order.
  Unit: none
  Range or possible values: [0,2]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The target must be known by the PCS.

After execution

- The command returns an ORDER+2 array of t0, p, [v, [a/2]] values for each polynomial requested in the result strings. v is returned if the order is greater than 0, and a/2 is returned if the order is greater than 1. t0 is in MJD days UT, p is in radians, v is in radians/sec, and a/2 is in radians/sec/sec. Thus the position of the rotator at time t is $P(t)=p + (t-t0)v+ (t-t0)^2a/2$. Where (t-t0) is in seconds. Note that a/2 (not a) is returned by the command.

Example

```
...
iifs::result res = iif->GetRotatorPolynomials(50.0,300,1.0,2,"left");
if(res.rescode == EXIT_SUCCESS) {
      int n = res.resmsg.size()-1;
      double time[n],angle[n],v[n],aby2[n];
      for(int i=0; i<n; i++) {
            istringstream in(res.resmsg[i+1]);
            in >> time[i] >> angle[i];
            if(ORDER > 0) in >> v[i];
            if(ORDER > 1) in >> aby2[i];
      }
} else {
      for(int i=1; i<(int)res.resmsg.size(); i++) {
            cout << res.resmsg[i] << endl;
      }
}
```

8.15 GetRotatorTrajectory

Description

This command returns the LBC rotator trajectory for the near future. This function has been designed specifically for the LBC and should not be used by any other instrument.

Syntax

```
result GetRotatorTrajectory( double NOOFSECS,
                             double INTERVAL,
                             double STARTTIME,
                             string SIDE )
```

Arguments

- **double NOOFSEC (in)**
  Description: the length of the trajectory.
  Unit: seconds
  Range or possible values: [TBD]
- **double INTERVAL (in)**
  Description: the time interval between points.
  Unit: seconds
  Range or possible values: [TBD]
- **double STARTTIME(in)**
  Description: the start time.
  Unit: MJD
  Range or possible values: [TBD]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The target must be known by the PCS.

After execution

- The command returns an array of (t, theta) pairs, representing time in days (JD, double) and

rotation angle (radians, double). The time and angle are blank separated, one pair per result string.

Example

```
...
iifs::result res = iif->GetRotatorTrajectory(50.0,1.0,0.0,"left");
if(res.rescode == EXIT_SUCCESS) {
      double time[res.resmsg.size()-1],angle[res.resmsg.size()-1];
      for(int i=1; i<res.resmsg.size(); i++) {
            istringstream in(res.resmsg[i]);
            in >> time[i-1] >> angle[i-1];
      }
} else {
      for(unsigned int i=1; i<res.resmsg.size(); i++) {
            cout << res.resmsg[i] << endl;
      }
}
```

8.16 LogEvent

Description

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

Syntax

```
result LogEvent( string eventName,
                 string eventDescription )
```

Arguments

- **string eventName (in)**
  Description: the event name to be logged in the TCS's logging system.
  Unit: none
  Range or possible values: Legal TCS event name.
- **string eventDescription (in)**
  Description: the descriptive text to be logged.
  Unit: none
  Range or possible values: Any string.

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The event must be previously defined in the data dictionary as "iif.<InstrumentID>.<eventName>".

After execution

- The telescope processes the logging request and the event description field is generated as "IIF <InstrumentID> <eventDescription>".
- If the event has not been previously defined, an event is logged with the specified name and description, and a priority of 3.

Example

```
...
iifs::result res = iif->LogEvent("testEvent","This is a test event");
```

8.17 MaximizeWrapTime

Description

This command is to provide some control over the use of the azimuth and rotator cable wraps. It sets two flags that condition the next PresetTelescope or OffsetPointing commands to either slew the fastest way, or the way that will maximize time on target. The rotator that will be affected is the rotator for the focal station issuing the command.

Syntax

```
result MaximizeWrapTime( bool AZFLAG, bool ROTFLAG,
                         string SIDE )
```

Arguments

- **bool** AZFLAG (in)
  Description: azimuth use maximum time flag.
  Unit: none
  Range or possible values: true | false
- **bool** ROTFLAG (in)
  Description: rotator use maximum time flag.
  Unit: none
  Range or possible values: true | false
- **string** SIDE (in)
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The flags will be set to control the next PresetTelescope command.

Example

...

```
iifs::result res = iif->MaximizeWrapTime(true, true, "left");
```

8.18 ModifyAO

Description

This command is used to modify various AO loop parameters before closing the AO loop. More details in reference [3], section 12.7.

Syntax

```
result ModifyAO(    int NMODES, double FREQ, int NBINS,
                    double TTMOD, string F1SPEC, string F2Spec,
                    string SIDE )
```

Arguments

- **int NMODES (in)**
  Description: number of modes to correct.
  Unit: none
  Range or possible values: TBD
- **double FREQ (in)**
  Description: CCD frequency.
  Unit: 1/second
  Range or possible values: TBD
- **int NBINS (in)**
  Description: CCD binning.
  Unit: none
  Range or possible values: TBD
- **double TTMOD (in)**
  Description: Tip-Tilt internal mirror modulation.
  Unit: none
  Range or possible values: TBD
- **string F1SPEC (in)**
  Description: position of filter wheel 1.
  Unit: none
  Range or possible values: TBD
- **string F2SPEC (in)**
  Description: position of filter wheel 2.
  Unit: none
  Range or possible values: TBD
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**

Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AOS must not be running.
- An AcquireRefAO must have been issued.

After execution

- The AO parameters are adjusted.

Example

```
...
iifs::result res = iif-> ModifyAO(42, 2.0, 1, 0.0, "pppp", "qqqq", "both");
```

8.19 Move

Description

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

Syntax

```
result Move(   double X, double Y, double Z,
               double RX, double RY, double RZ,
               int D_FLAG,
               string MOVE_TYPE,
               string OPE,
               int TIME,
               string SIDE )
```

Arguments

- **double X,Y,Z (in)**
  Description: the telescope 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 is in Figure 6.1
  Unit: millimeters.
  Range or possible values: Depends on OPE and current position.
- **double RX,RY,RZ (in)**
  Description: the telescope focal plane rotations. MOVE_TYPE will determine if they are absolute or relative values. For OPE M3, RX is M3 Tip, RY is M3 Tilt.
  Unit: micro radians.
  Range or possible values: Depends on OPE and current position.
- **int D_FLAG (in)**
  Description: 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.
  Unit: none
  Range or possible values: [0x0, 0x3f]
- **string MOVE_TYPE (in)**
  Description: specifies if the movements are absolutes or relatives.
  Unit: none
  Range or possible values: "REL" | "ABS"
- **string OPE (in)**
  Description: specifies which optical element(s) to move.
  Unit: none
  Range or possible values: "DEFAULT" | "M1" | "M2" | "M3" | "M1M2" | "M1M3" | "M2M3" | "M1M2M3"
- **int TIME (in)**
  Description: the lookahead time for the collimation correction. This is

not currently supported.
Unit: seconds
Range or possible values: TBD

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE is in a new position.

Example

```
...
iifs::result res = iif->Move(1.42,1.03,1.0,1.0,0.4,0.4,255,
                             "REL","M1",0,"left");
```

8.20 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 6.1). If active or adaptive optics are running, the w/W stage is also moved.

Syntax

```
result MoveFocus( double ABSPOS,
                  string OPE,
                  string SIDE )
```

Arguments

- **double ABSPOS (in)**
  Description: the new absolute focus position.
  Unit: millimeter.
  Range or possible values: Depends on OPE.
- **string OPE (in)**
  Description: the optical element to move.
  Unit: none
  Range or possible values: "M1" | "M2" | "M3" | "M1M2"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE is in a new position.

Example

```
...
iifs::result res = iif->MoveFocus(1.42,"M1","left");
```

8.21 MoveXY

Description

The MoveXY command moves an OPE in X and Y direction, relative to the current position (more information about the coordinate system in figure 6.1). In closed-loop active or adaptive optics, the appropriate AGw/W stage is also moved.

Syntax

```
result MoveXY (     double XMOTION, double YMOTION,
                    string OPE,
                    string SIDE )
```

Arguments

- **double XMOTION, YMOTION (in)**
  Description: the distance to move in X and Y.
  Unit: millimeter.
  Range or possible values: Depends on OPE.
- **string OPE (in)**
  Description: the OPE to move.
  Unit: none
  Range or possible values: "M1" | "M2" | "M1M2"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE is in a new position.

Example

```
...
iifs::result res = iif->MoveXY(1.000,-0.810,"M1","left");
```

8.22 MoveXYZ

Description

The MoveXYZ command moves the primary and secondary together in X, Y and Z. The movement is relative and not synchronized between the OPE. (More information about the coordinate system in 6.4)

Syntax

```
result MoveXYZ(     double RELX, double RELY, double RELZ,
                    string SIDE )
```

Arguments

- **double RELX, RELY, RELZ (in)**
  Description: the X, Y, and Z motion.
  Unit: millimeters
  Range or possible values: Depends on OPE and current position.
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result** RESULT (returned)
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE are in a new position.

Example

```
...
iifs::result res = iif->MoveXYZ(1.000,-0.810,1.22,"left");
```

8.23 OffsetGuiding

Description

The OffsetGuiding command only supports the LBC and LBTI instruments. This command allows these instruments to inject guide centroid updates computed in the AZALT reference frame into the TCS. Guiding for other instruments is managed by the Guiding Control Subsystem (GCS).

Syntax

```
result OffsetGuiding(    double ANGLE,
                         double OFFX, double OFFY,
                         string COORDSYS,
                         string SIDE )
```

Arguments

- **double ANGLE (in)**
  Description: the rotator offset. This currently has no effect.
  Unit: radian.
  Range or possible values: [-2PI, 2PI]
- **double OFFX, OFFY (in)**
  Description: the guiding offsets in x and y. The values must be in AZALT coordinates.
  Unit: radians.
  Range or possible values: -
- **string COORDSYS (in)**
  Description: the offset coordinate system. This is ignored.
  Unit: none
  Range or possible values:
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The telescope has adjusted the pointing.

Example

```
...
iifs::result res = iif->OffsetGuiding(-1.1,0.4,0.9,"RADEC","left");
```

8.24 OffsetPointing

Description

The role of the OffsetPointing command is two-fold. The command allows the observer to either change the current coordinates on the sky to a new target position (RADEC), or change the position in the focal plane where the current target is imaged (DETXY). When changing the focal plane position, the coordinates on the sky are not modified. A preceding PresetTelescope command which provides the original reference conditions is required.

Syntax

```
result OffsetPointing(   double ANGLE,
                         double OFFX, double OFFY,
                         string COORDSYS,
                         string OPE (unused),
                         string NEW_POSITION (unused),
                         string MOVE_TYPE,
                         string SIDE )
```

Arguments

- **double ANGLE (in)**
  Description: the rotator offset (delta).
  Unit: radians.
  Range or possible values: [-2PI, 2PI]
- **double OFFX, OFFY (in)**
  Description: the offsets in x and y.
  Unit: radians or seconds for OFFX, radians for OFFY.
  Range or possible values: -
- **string COORDSYS (in)**
  Description: the offset coordinate system.
  Unit:
  Range or possible values: See 6.4.2.
- **string OPE (in)**
  Description: ignored.
  Unit: none
  Range or possible values: anything, the value is not checked
- **string NEW_POSITION (in)**
  Description: ignored.
  Unit: none
  Range or possible values: anything, the value is not checked
- **string MOVE_TYPE (in)**
  Description: applies to both the rotator angle and the supplied offsets, and specifies motion relative to the current position ("REL") or with respect to the original PresetTelescope position ("ABS"). "CS" may be appended to request the offset be considered a pure coordinate system offset

(the default is in the focal plane), and is only allowed for RADEC and GALACTIC. This means for RADEC the RA value is in seconds of time.
Unit: none
Range or possible values: "REL" | "RELCS" | "ABS" | "ABSCS"

- **string SIDE (in)**
Description: the command side.
Unit: none
Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The telescope is in a new position.

Example

```
...
iifs::result res = iif->OffsetPointing(28.45e-2,0.005,0.99,"M1","false",
                                       "REL","left");
```

8.25 OffsetPointing2

Description

The OffsetPointing2 command is the same as the OffsetPointing command except it does not have the OPE and NEW_POSITION arguments. The role of the OffsetPointing2 command is two-fold. The command allows the observer to either change the current coordinates on the sky to a new target position (RADEC), or change the position in the focal plane where the current target is imaged (DETXY). When changing the focal plane position, the coordinates on the sky are not modified. A preceding PresetTelescope command which provides the original reference conditions is required.

Syntax

```
result OffsetPointing2( double ANGLE,
                        double OFFX, double OFFY,
                        string COORDSYS,
                        string MOVE_TYPE,
                        string SIDE )
```

Arguments

- **double ANGLE (in)**
  Description: the rotator offset (delta).
  Unit: radians.
  Range or possible values: [-2PI, 2PI]
- **double OFFX, OFFY (in)**
  Description: the offsets in x and y.
  Unit: radians or seconds for OFFX, radians for OFFY.
  Range or possible values: -
- **string COORDSYS (in)**
  Description: the offset coordinate system.
  Unit:
  Range or possible values: See 6.4.2.
- **string MOVE_TYPE (in)**
  Description: applies to both the rotator angle and the supplied offsets, and specifies motion relative to the current position ("REL") or with respect to the original PresetTelescope position ("ABS"). "CS" may be appended to request the offset be considered a pure coordinate system offset (the default is in the focal plane), and is only allowed for RADEC and GALACTIC. This means for RADEC the RA value is in seconds of time.
  Unit: none
  Range or possible values: "REL" | "RELCS" | "ABS" | "ABSCS"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The telescope is in a new position.

Example

```
...
iifs::result res = iif->OffsetPointing2(28.45e-2,0.005,0.99,"REL","left");
```

8.26 OffsetXYAO

Description

This command is used to move the AO stage in the XY plane. More details in reference [3], section 12.9.

Syntax

```
result OffsetXYAO( double DELTAX, double DELTAY, string SIDE )
```

Arguments

- **double DELTAX (in)**
  Description: X position offset.
  Unit: mm
  Range or possible values: TBD
- **double DELTAY (in)**
  Description: Y position offset.
  Unit: mm
  Range or possible values: TBD
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- The AO loop may be running or not.

After execution

- The AO stage is moved.

Example

```
...
iifs::result res = iif->OffsetXYAO(2.0, 3.0, "left");
```

8.27 OffsetZAO

Description

This command is used to move the AO stage in the Z direction. More details in reference [3], section 12.10.

Syntax

```
result OffsetXYAO( double DELTAZ, string SIDE )
```

Arguments

- **double DELTAZ (in)**
  Description: Z position offset.
  Unit: mm
  Range or possible values: TBD
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- The AO loop may be running or not.

After execution

- The AO stage is moved.

Example

```
...
iifs::result res = iif->OffsetZAO(2.0, "left");
```

8.28 PauseAO

Description

This command is issued to temporarily suspend the current AO operation. More details in reference [3], section 12.13. The AGw guider is reactivated.

Syntax

```
result PauseAO( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- AOS is in pause mode.
- GCS is guiding and wave front sensing.

Example

```
...
iifs::result res = iif->PauseAO("both");
```

8.29 PauseGuiding

Description

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

Syntax

```
result PauseGuiding( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The GCS must be guiding.

After execution

- Guiding is in pause mode. If wave front sensing was running, it also is paused.

Example

```
...
iifs::result res = iif->PauseGuiding("both");
```

8.30 PresetAO

Description

The PresetAO command is issued in an AOS observation service status in order to prepare the AO system for an observation in adaptive mode. More details in reference [3], section 12.3. A SetReference command must have been issued to define the required AO reference star.

Syntax

```
result PresetAO( string AOMODE, string SIDE )
```

Arguments

- **string AOMODE (in)**
  Description: the AO mode.
  Unit: none
  Range or possible values: "AO_TTM", "AO_ACE", "AO_ICE"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- A SetReference must have been issued.

After execution

- AOS performs all set up operation needed, except acquisition of the reference object.

Example

```
...
iifs::result res = iif->PresetAO("AO_ACE", "left");
```

8.31 PresetFlatAO

Description

The PresetFlatAO command is issued in an AOS observation service status in order to request the AO system to prepare itself for the AO_FIX observation mode. More details in reference [3], section 12.2.

Syntax

```
result PresetFlatAO( string FLATSPEC, string SIDE )
```

Arguments

- **string FLATSPEC (in)**
  Description: desired flat name.
  Unit: none
  Range or possible values: TBD
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AOS must be in observation service status.
- The AO loop is not closed.

After execution

- The mirror is in the desired shape.

Example

```
...
iifs::result res = iif->PresetFlatAO(FLATSPEC, "left");
```

8.32 PresetTelescope

Description

The PresetTelescope command starts a new observing cycle, and allows for a full specification of observing conditions associated with a particular target or "target plus offset object" combination. The two mode arguments control how the telescope and rotator are moved. A SetStars (sidereal target), SetStars2 (either sidereal or non-sidereal target) or SetNonSidereal command must have been previously issued. If the SetHotspot, SetOffset, SetReference or SetReference2 commands have been issued, the corresponding object will be included in the command. Depending upon the observation mode, a guide star list must also be provided (using the SetStars or SetStars2 commands). All internal IIF settings from previous PresetTelescope and optional OffsetPointing commands (e.g., updated target hotspot/pointing origin etc.) are not retained, but SetStars, SetStars2, SetHotspot, SetOffset, SetNonSidereal, SetReference, and SetReference2 objects remain until overwritten or cleared. For non-sidereal targets only TRACK GUIDE, and ACTIVE modes are supported, and SetOffset is not supported.

Syntax

```
result PresetTelescope( double ANGLE,
                        string ROTATORMODE,
                        string MODE,
                        string SIDE )
```

Arguments

- **double ANGLE (in)**
  Description: the rotator position.
  Unit: radians
  Range or possible values: [-2PI, 2PI]
- **string ROTATORMODE (in)**
  Description: the rotator mode.
  Unit: none
  Range or possible values: See section 6.4.3.
- **string MODE (in)**
  Description: the operating mode.
  Unit: none
  Range or possible values: See section Telescope modes.
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- A sidereal target and any guide stars may be specified using the command SetStars.
- A non-sidereal target must be specified using the command SetNonSidereal.
- Optional: The hotspot, offset, and AO reference star may be specified using the commands SetHotspot, SetOffset, and SetReference.

After execution

- The telescope is at a new target operating according to the requested settings.

Example

```
...
iifs::position target =
{0.009,0.11,"RADEC",2000.0,"J2000",0.0,0.0,0.0,"U",0.0,"U_B",0.8};
iifs::position guidestar =
{0.001,0.21,"RADEC",2000.0,"J2000",0.0,0.0,0.0,"U",0.0,"U_B",0.4};
iifs::SeqPos stars;
stars.push_back(target);
stars.push_back(guidestar);
iif->SetStars(stars);
//optional
iif->SetOffset(0.099, 1.100, "RADEC");
iif->SetHotspot(0.013, 0.199);
iifs::position reference =
{0.009,0.11,"RADEC",2000.0,"J2000",0.0,0.0,0.0,"U",0.0,"U_B",0.8};
iifs::SeqPos refs;
refs.push_back(reference)
iif->SetReference(refs);
//Call presetTelescope
iifs::result res =
iif->PresetTelescope(1.0,"POSITION","ADAPTIVEACE_ACTIVE","left");
```

8.33 RefineAO

Description

This command is issued to optimize the AO loop parameters. More details in reference [3], section 12.6.

Syntax

```
result RefineAO( string METHOD, string SIDE )
```

Arguments

- **string METHOD (in)**
  Description: the name of the optimization method.
  Unit: none
  Range or possible values: TBD
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- An AcquireRefAO must have been issued.

After execution

- The AO loop is optimized.

Example

```
...
iifs::result res = iif->RefineAO("OPT1", "left");
```

8.34 Remove

Description

The Remove command removes the target, guidestars, offset, hotspot, nonSidereal and AO reference objects.

Syntax

```
void Remove( )
```

Arguments

- None

Return value

- None

Preconditions

- None.

After execution

- The target, guidestars, offset, hotspot, nonSidereal and reference objects are deleted.

Example

```
...
iif->remove()
```

8.35 ResumeAO

Description

This command resumes suspended operation after a PauseAO. More details in reference [3], section 12.14.

Syntax

```
result ResumeAO( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- A PauseAO must have been issued.
- AOS is in pause mode.

After execution

- The loop is closed.
- AGw guiding is de-activated.

Example

```
...
iifs::result res = iif->ResumeAO("left");
```

8.36 ResumeGuiding

Description

This command resumes operation after a PauseGuiding.

Syntax

```
result ResumeGuiding( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- A PauseGuiding must have been issued.
- The AGw guiding must be paused.

After execution

- Guiding is running. If wave front sensing was paused, it also is running.

Example

```
...
iifs::result res = iif->ResumeGuiding("both");
```

8.37 RotateCommon

Description

The RotateCommon command rotates the primary and the secondary mirror around a common point. The movement is relative and not synchronized. DIRECTION is with respect to the primary mirror: a DIRECTION of zero means the primary mirror will move in the +X direction and the secondary will move in the opposite direction.

Syntax

```
result RotateCommon(     double DISTANCE,
                         double ANGLE,
                         double DIRECTION,
                         string SIDE)
```

Arguments

- **double DISTANCE (in)**
  Description: the distance above the mirror about which to rotate.
  Unit: millimeters.
  Range or possible values: TBD
- **double ANGLE (in)**
  Description: the rotation angle.
  Unit: micro-radians
  Range or possible values: [-9999.999, 9999.999]
- **double DIRECTION (in)**
  Description: the direction of rotation.
  Unit: radians
  Range or possible values: [0.0, 2PI]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result** RESULT (returned)
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE are in a new position.

Example

```
...
iifs::result res = iif->RotateCommon(2.00,10,3.15,"left")
```

8.38 RotatePrimary

Description

RotatePrimary rotates the primary mirror about a point on axis above the mirror. The direction is zero along the X axis and increases counter-clockwise. A DIRECTION of zero means the primary mirror will move in the +X direction.

Syntax

```
result RotatePrimary( double DISTANCE,
                      double ANGLE,
                      double DIRECTION,
                      string SIDE )
```

Arguments

- **double DISTANCE (in)**
  Description: the distance above the mirror about which to rotate.
  Unit: mm
  Range or possible values: [999.000, 99999.99]
- **double ANGLE (in)**
  Description: the rotation angle.
  Unit: micro-radian
  Range or possible values: [0, 999.99]
- **double DIRECTION (in)**
  Description: the rotation direction.
  Unit: radian
  Range or possible values: [0, 2Pi]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The primary mirror is in a new position.

Example

```
...
iifs::result res = iif->RotatePrimary(200.0,25.0,1.57,"left");
```

8.39 RotateZ

Description

RotateZ rotates the selector axis of the tertiary mirror to adjust the incoming beam angle for the instrument. A relative rotation is from the current position, and an absolute rotation is with respect to the focal station position maintained by the OSS.

Syntax

```
result RotateZ( double ANGLE,
                string MOVE_TYPE,
                string SIDE )
```

Arguments

- **double ANGLE (in)**
  Description: the rotation angle.
  Unit: micro radians
  Range or possible values: TBD
- **string MOVE_TYPE (in)**
  Description: specifies relative or absolute movement.
  Unit: none
  Range or possible values: "REL" | "ABS"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The tertiary mirror is in a new position.

Example

```
...
iifs::result res = iif->RotateZ(1.35,"REL","left");
```

8.40 RotHold

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. The rotator that will be affected is the rotator for the focal station issuing the command. See reference [4] for further details.

Syntax

```
result RotHold( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

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.

Example

```
...
iifs::result res = iif->RotHold("left");
```

8.41 RotReady

Description

This command is issued to enable or disable a rotator. "Enable" means to turn the cable chain and 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. The rotator that will be affected is the rotator for the focal station issuing the command.

Syntax

```
result RotReady( bool ENABLE, string SIDE )
```

Arguments

- **bool** ENABLE (in)
  Description: determines whether to enable or disable the rotator.
  Unit: none
  Range or possible values: true | false
- **string** SIDE (in)
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

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.

Example

```
...
iifs::result res = iif->RotReady(true, "left");
```

8.42 RotServicePosition

Description

Moves the rotator to the specified angle in the native coordinate frame and holds at that position. This is intended for instrument use to position their rotator for maintenance operations. The rotator that will be affected is the rotator for the focal station issuing the command.

Syntax

```
result RotServicePosition( double ANGLE,
                           string SIDE )
```

Arguments

- **double ANGLE (in)**
  Description: the rotator position.
  Unit: degrees.
  Range or possible values: [-90, 460]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The rotator will be at the specified position and holding.

Example

```
...
iifs::result res = iif->RotServicePosition(90.0, "left");
```

8.43 RotTrack

Description

Makes a rotator begin tracking according to the polynomial stream it is currently receiving from the PCS. The rotator that will be affected is the rotator for the focal station issuing the command. See reference [4], section 7.1 for further details.

Syntax

```
result RotTrack( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The rotator is in "hold" mode.

After execution

- The rotator will be tracking as described by the polynomials being generated by the PCS.

Example

```
...
iifs::result res = iif->RotTrack("left");
```

8.44 RunAO

Description

This command is issued following a PresetTelescope to acquire the AO reference star and start the AO loop. It optionally executes AcquireRefAO and StartAO, and insures the GCS is operating correctly. The plethora of types are to support AO engineering; they are:

TYPE	Description
NORMAL	Issues an AcquireRefAO without rePoint followed by a StartAO.
REPOINT	Issues an AcquireRefAO with rePoint followed by a StartAO.
SKIPREF	Skips the AcquireRefAO with assumed success followed by a StartAO.
SKIPSTART	Issues an AcquireRefAO without rePoint and skips the StartAO with assumed success.
SKIPSTARTREPOINT	Issues an AcquireRefAO with rePoint and skips the StartAO with assumed success.
SKIPALL	Skips both the AcquireRefAO and StartAO with assumed success.

Whenever the AcquireRefAO is issued the rePoint condition can be overridden on the IIFGUI.

Syntax

```
result RunAO( string TYPE, string SIDE )
```

Arguments

- **string TYPE (in)**
  Description: execution control flag.
  Unit: none
  Range or possible values: "NORMAL" | "REPOINT" | "SKIPREF" | "SKIPSTART" | "SKIPSTARTREPOINT" | "SKIPALL"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- A PresetTelescope for an adaptive mode must have been issued.

After execution

- Internal flags are set indicating the AO reference star is acquired and the AO loop is closed. Whether this is true depends on the command type.
- The telescope pointing may be changed if rePoint is requested.

Example

```
...
iifs::result res = iif->RunAO("NORMAL", "left");
```

8.45 SendWavefront

Description

SendWavefront sends an array of Zernike coefficients (see reference [11]) to be applied to either the primary or secondary mirror.

Syntax

```
result SendWavefront( SeqWF POLYNOM,
                      string OPE,
                      string SIDE )
```

Arguments

- **SeqWF POLYNOM (in)**
  Description: SeqWF structure containing no more than 28 Zernike coefficients. Non specified coefficients at the end of the list will be 0.0.
  Unit: nanometers.
  Range or possible values: TBD
- **string OPE (in)**
  Description: the optical element.
  Unit: none
  Range or possible values: "M1" | "M2" | "DEFAULT"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- Mirror in new shape.

Example

```
...
```

```
iifs::SeqWF wfs;
wfs.push_back(0.001);
wfs.push_back(0.014);
wfs.push_back(0.012);
wfs.push_back(0.969);
wfs.push_back(0.101);
iifs:result res = iif->SendWavefront(wfs,"M1","left");
```

8.46 SetAGWFilter

Description

SetAGWFilter sets the guider filter.

Syntax

```
result SetAGWFilter( int FILTERNUM,
                     string SIDE)
```

Arguments

- **int FILTERNUM (in)**
  Description: the number of the guider filter desired.
  Unit: none
  Range or possible values: [1,5]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The guider filter has been changed.

Example

```
...
iifs::result res = iif->SetAGWFilter(3,"left");
```

8.47 SetGuidingBinning

Description

SetGuidingBinning sets the guider binning in pixels. The value affects both X and Y.

Syntax

```
result SetGuidingBinning( int FACTOR, string SIDE)
```

Arguments

- **int FACTOR (in)**
  Description: the binning factor.
  Unit: pixel
  Range or possible values: [1, n]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The guider binning has been changed.

Example

```
...
iifs::result res = iif->SetGuidingBinning(4,"left");
```

8.48 SetGuidingHotspot

Description

SetGuidingHotspot sets the guider hotspot in pixels. The values can be either relative or absolute. This command is intended for use by PEPSIPFU since its guider does not have a movable probe.

Syntax

```
result SetGuidingHotspot( double COORDX, double COORDY,
                          string MOVE, string SIDE)
```

Arguments

- **double COORDX, COORDY (in)**
  Description: the X and Y coordinates of the hotspot.
  Unit: pixel
  Range or possible values:
- **string MOVE (in)**
  Description: the move type.
  Unit: none
  Range or possible values: "REL" | "ABS"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The guider hotspot has been changed.

Example

```
...
iifs::result res = iif->SetGuidingHotspot(0.98,3.76,"REL","left");
```

8.49 SetHotspot

Description

SetHotspot defines a hotspot to be used by the PresetTelescope command. This is an optional step that may be done before PresetTelescope is called. If a hotspot is defined, it persists until overwritten or cleared.

Syntax

```
result SetHotspot( double COORDX, double COORDY )
```

Arguments

- **double COORDX, COORDY (in)**
  Description: the X and Y coordinates of the hotspot.
  Unit: mm
  Range or possible values:

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The hotspot is defined.

Example

```
...
iifs::result res = iif->SetHotspot(0.98,3.76);
```

8.50 SetNonSidereal

Description

SetNonSidereal defines a non-sidereal target to be used by the PresetTelescope command. This is an optional step that may be done before PresetTelescope is called. If a non-sidereal target is defined, it persists until overwritten or cleared, and takes precedence over a sidereal target defined with the SetStars command.

Syntax

```
result SetNonSidereal( nonsidereal TARGET, bool OVERRIDE )
```

Arguments

- **nonsidereal TARGET (in)**
  target structure.
  Unit:
  Range or possible values:
- **bool OVERRIDE (in)**
  Description: boolean flag to set the IIF non-sidereal override flag This flag is reserved for use by the IRTC and should be false for all other instruments.
  Unit:
  Range or possible values: true | false

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The non-sidereal object is defined.
- If OVERRIDE is true, the non-sidereal override flag in the IIF is set.

Example

```
...
nonsidereal ns = {"DIFFERENTIAL",0.55f,"",255567.345,0.01,0.5,0.01,-0.02};
iifs::result res = iif->SetNonSidereal(ns, false);
```

8.51 SetOffset

Description

SetOffset defines offset values to be used by the PresetTelescope command. The coordinate system of the offset does not have to be the same as the target. This is an optional step that may be done before PresetTelescope is called. If an offset is defined, it persists until overwritten or cleared.

Syntax

```
result SetOffset( double OFFX, double OFFY,
                  string COORDSYS )
```

Arguments

- **double OFFX, OFFY (in)**
  Description: the X and Y offset.
  Unit: radians.
  Range or possible values:
- **string COORDSYS (in)**
  Description: the offset coordinate system.
  Unit: none
  Range or possible values: See 6.4.2.

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The initial offset is defined.

Example

```
...
iifs::result res = iif->SetOffset(0.98,3.76,"RADEC");
```

8.52 SetOffset2

Description

SetOffset2 defines offset values to be used by the PresetTelescope command. This command may be used instead of SetOffset and must be issued before the PresetTelescope command. (A new command is needed because ICE does not support C++ default arguments or overloading.) The coordinate system of the offset does not have to be the same as the target. This is an optional step that may be done before PresetTelescope is called. If an offset is defined, it persists until overwritten or cleared.

Syntax

```
result SetOffset2( double OFFX, double OFFY,
                   string COORDSYS, string CS )
```

Arguments

- **double OFFX, OFFY (in)**
  Description: the X and Y offset.
  Unit: radians.
  Range or possible values:
- **string COORDSYS (in)**
  Description: the offset coordinate system.
  Unit: none
  Range or possible values: See 6.4.2.
- **string CS (in)**
  Description: the offset cs attribute. "CS" is only allowed for RADEC and GALACTIC coordinate systems. See reference [1].
  Unit: none
  Range or possible values: "CS" | "".

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The initial offset is defined.

Example

```
...
iifs::result res = iif->SetOffset2(23.45,.005,"RADEC","CS");
```

8.53 SetParameter

Description

SetParameter sets multiple values into the Data Dictionary. Note that the instrument only has permission to modify its own predefined entries. If one or more names are invalid, or cannot be written for some reason, the command will fail and return a list of all the failing entries. All the successful entries will be written.

Syntax

```
result SetParameter( SeqDD MULTIENTRIES )
```

Arguments

- **SeqDD MULTIENTRIES (in)**
  Description: SeqDD structure containing the data dictionary names and values.
  Unit: none
  Range or possible values: Valid data dictionary entries.

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The SeqDD object is populated with string pairs: the first string is the data dictionary name, and the second is the value. If the name does *not* contain a period, the IIF will look up the name in the TCS public name list. If that fails, or the name *does* contain a period, the IIF will generate the fully qualified data dictionary name as "iif.<InstrumentID>.<DDname>". An instrument is not allowed to set variables it does not own.

After execution

- The Data Dictionary has new values.

Example

```
...
SeqDD DDEntries;
DDstruct dd;
dd.DDname = "side[0].cooler";
dd.DDkey = "ON";
```

```
DDEntries.push_back(dd);
dd.DDname = "L_MODSBlueCCDTemp";
dd.DDkey = "40";
DDentries.push_back(dd);
iifs::result res = iif->SetParameter(DDEntries);
```

8.54 SetPMTerm

Description

SetPMTerm temporarily changes any term in the currently loaded pointing model. Note that it does not change the pointing model file on disk, so any changes are lost when a new pointing model is loaded.

Syntax

```
result SetPMTerm( string NAME, double VALUE, string SIDE)
```

Arguments

- **string NAME (in)**
  Description: the pointing model parameter name.
  Unit: none
  Range or possible values: TBD
- **double VALUE (in)**
  Description: the pointing model parameter value.
  Unit: radian
  Range or possible values:
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument is authorized.

After execution

- The parameter is changed.

Example

```
...
iifs::result res = iif->SetPMTerm("IA", 0.002, "left");
```

8.55 SetPMTerm2

Description

SetPMTerm2 temporarily changes any term in the currently loaded pointing model. Note that it does not change the pointing model file on disk, so any changes are lost when a new pointing model is loaded. This command supports incremental changes (MOVE_TYPE "REL"). (A new command is needed because ICE does not support C++ default arguments or overloading.)

Syntax

```
result SetPMTerm( string NAME, double VALUE, string MOVE_TYPE
string SIDE)
```

Arguments

- **string NAME (in)**
  Description: the pointing model parameter name.
  Unit: none
  Range or possible values: TBD
- **double VALUE (in)**
  Description: the pointing model parameter value.
  Unit: radian
  Range or possible values:
- **string MOVE_TYPE (in)**
  Description: specifies relative or absolute changes.
  Unit: none
  Range or possible values: "REL" | "ABS"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument is authorized.

After execution

- The parameter is changed.

Example

```
...
iifs::result res = iif->SetPMTerm2("IA", 0.002, "REL", "left");
```

8.56 SetReference

Description

SetReference sets the AO reference star positions for the PresetTelescope command. It is optional, but must be issued before the PresetTelescope command if adaptive optics is desired. If AO reference stars are defined, they persist until overwritten or cleared. The PresetTelescope command only uses the first star in the list. Additional stars in the list may be used by the GetKFPCoordinates command.

Syntax

```
result SetReference( SeqPos STARS )
```

Arguments

- **SeqPos STARS (in)**
  Description: SeqPos structure containing the AO reference star.
  Unit: none
  Range or possible values:

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The SeqPos object is populated with valid data.

After execution

- The AO reference star set.

Example

```
...
SeqPos stars;
position reference =
      {2.399,1.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.8};
stars.push_back(reference);
iifs::result res = iif->SetReference(stars);
```

8.57 SetReference2

Description

SetReference2 sets the AO reference star positions for the PresetTelescope command using a SeqPos2 vector. (Note that the AO reference star currently cannot be a non-sidereal object.) It is optional, but must be issued before the PresetTelescope command if adaptive optics is desired. If AO reference stars are defined, they persist until overwritten or cleared. The PresetTelescope command only uses the first star in the list. Additional stars in the list may be used by the GetKFPCoordinates2 command. This command is the same as SetReference but uses the position2 structure.

Syntax

```
result SetReference2( SeqPos2 STARS )
```

Arguments

- **SeqPos2 STARS (in)**
  Description: SeqPos2 structure containing the AO reference star.
  Unit: none
  Range or possible values:

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The SeqPos2 object is populated with valid data.

After execution

- The AO reference star set.

Example

```
      ...
      SeqPos2 stars;
      position2 reference =

      {"sidereal",2.399,1.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.8,"",0,0,
0};
      stars.push_back(reference);
      iifs::result res = iif->SetReference2(stars);
```

8.58 SetStars

Description

SetStars sets the sidereal target position and guide star positions for the PresetTelescope command. It must be issued before the PresetTelescope command unless the SetTarget command is used. Once a sidereal target is defined, it persists until overwritten or cleared. But note that a non-sidereal target takes precedence over a sidereal target. See section 7.3 for a description of the position structure.

Syntax

```
result SetStars( SeqPos STARS )
```

Arguments

- **SeqPos STARS (in)**
  Description: SeqPos structure containing the target and guide stars.
  Unit: none
  Range or possible values:

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The SeqPos object is populated with valid data.

After execution

- The target and guide stars are set.

Example

```
...
SeqPos stars;
position target =
      {2.399,1.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.8,"JB10_092"};
position guidestar =
      {1.399,2.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.4,"UT10_012"};
stars.push_back(target);
stars.push_back(guidestar);
iifs::result res = iif->SetStars(stars);
```

8.59 SetStars2

Description

SetStars2 sets the target position (either sidereal or non-sidereal) and guide star positions for the PresetTelescope command. This command may be used instead of SetStars and must be issued before the PresetTelescope command. (A new command is needed because ICE does not support C++ default arguments or overloading.) Once a target is defined, it persists until overwritten or cleared. This command uses the SeqPos2 structure which supports both sidereal and non-sidereal targets. (See section 7.4 for a description of the position2 structure.)

Syntax

```
result SetStars2( SeqPos2 STARS )
```

Arguments

- **SeqPos2 STARS (in)**
  Description: SeqPos2 structure containing the target and guide stars.
  Unit: none
  Range or possible values:

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The SeqPos2 object is populated with valid data.

After execution

- The target and guide stars are set.

Example

```
      ...
      SeqPos2 stars;
      position2 target =

      {"SIDEREAL",2.399,1.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.8,"",0,0,
0};
      position2 guidestar =

      {"SIDEREAL",1.399,2.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.4,"",0,0,
```

```
0};
      stars.push_back(target);
      stars.push_back(guidestar);
      iifs::result res = iif->SetStars2(stars);
      ...
      SeqPos2 stars;
      position2 target =

      {"FILE",2.399,1.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.8,"ephemeris_
file",0,0,0};
      position2 guidestar =

      {"SIDEREAL",1.399,2.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.4,"",0,0,
0};
      stars.push_back(target);
      stars.push_back(guidestar);
      iifs::result res = iif->SetStars2(stars);
      ...
      SeqPos2 stars;
      position2 target =

      {"DIFFERENTIAL",2.399,1.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.8,"",
12345678.123,10.0,20.0};
      Position2 guidestar =

      {"SIDEREAL",1.399,2.0009,"RADEC",2000.0,"J2000",0,0,0,"U",0,"U_B",0.4,"",0,0,
0};
      stars.push_back(target);
      stars.push_back(guidestar);
      iifs::result res = iif->SetStars2(stars);
```

8.60 SetTarget

Description

This deprecated command defines target parameters to be used by the PresetTelescope command. If the SetStars or SetStars2 commands are used, this command need not be called.

Syntax

```
result SetTarget( double COORD1, double COORD2,
                  string COORDSYS,
                  double EPOCH,
                  float WAVELENGTH )
```

Arguments

- **double COORD1, COORD2 (in)**
  Description: the X and Y coordinates.
  Unit: radians or mm.
  Range or possible values:
- **string COORDSYS (in)**
  Description: the target coordinate system.
  Unit: none
  Range or possible values: See 6.4.1.
- **double EPOCH (in)**
  Description: the epoch.
  Unit: year
  Range or possible values: valid year
- **float WAVELENGTH (in)**
  Description: the target effective wavelength.
  Unit: microns
  Range or possible values: [0.3, 15]

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- None.

After execution

- The target is defined.

Example

```
...
iifs::result res = iif->SetTarget(0.98,3.76,"RADEC",2009.2,0.5);
```

8.61 Standby

Description

Standby sets the standby level for the telescope side. This is not currently used by the TCS, but is set to zero whenever an instrument is authorized by the TO.

Syntax

```
result Standby( int LEVEL, string SIDE )
```

Arguments

- **int LEVEL (in)**
  Description: the standby level.
  Unit: none
  Range or possible values: [0. 5].
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result** RESULT (returned)
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The standby level is changed.

Example

```
...
iifs::result res = iif->Standby(2,"left");
```

8.62 StartAO

Description

This command is issued to close the AO loop. It is issued by the IIF in the RunAO command. More details in reference [3], section 12.8.

Syntax

```
result StartAO( string SIDE )
```

Arguments

- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AO reference star must have been acquired.

After execution

- AOS will send the request message to AO-Sup which properly updates the related variable in the data dictionary.
- AOS is in closed loop mode.

Example

```
...
iifs::result res = iif->StartAO("left");
```

8.63 StepFocus

Description

StepFocus changes the telescope focus a delta amount by moving the specified OPE.

Syntax

```
result StepFocus( double RELPOS,
                  string OPE,
                  string SIDE )
```

Arguments

- **double RELPOS (in)**
  Description: the amount to change the focus.
  Unit: millimeter
  Range or possible values: Depends on OPE and current position.
- **string OPE (in)**
  Description: the OPE to move.
  Unit: none
  Range or possible values: "M1" | "M2" | "M3" | "M1M2" (scale-preserving focus)
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result** RESULT (returned)
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE is in a new position.

Example

```
...
iifs::result res = iif->StepFocus(-1.42,"M1","left");
```

8.64 StopAO

Description

This command is issued to open the AO loop. More details in reference [3], section 12.12.

Syntax

```
result StopAO( string REASON, string SIDE )
```

Arguments

- **string REASON (in)**
  Description: the reason for the request.
  Unit: none
  Range or possible values:
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- The AO must be in closed loop mode.

After execution

- AOS will send the request message to AO-Sup which properly updates the related variable in the data dictionary.
- AOS loop is stopped.

Example

```
...
iifs::result res = iif->StopAO("Done for the night", "left");
```

8.65 TipTilt

Description

The TipTilt command moves an OPE in tip and tilt direction, relative to the current position. 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,
                string OPE,
                string SIDE )
```

Arguments

- **double XROTATION, YROTATION (in)**
  Description: the X and Y rotation angles.
  Unit: micro radians.
  Range or possible values: Depends on OPE.
- **string OPE (in)**
  Description: the OPE to move.
  Unit: none
  Range or possible values: "M1" | "M2" | "M3"
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

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

After execution

- The OPE is in a new position.

Example

```
...
iifs::result res = iif->TipTilt(1.255,-0.815,"M1","left");
```

8.66 UpdateNonSiderealTarget

Description

The UpdateNonSiderealTarget command modifies the current non-sidereal target data in the TCS. Only the tracking rates may be updated. This command is not sided because both sides of the telescope must have the same non-sidereal target.

Syntax

```
result UpdateNonSiderealTarget( double RARATE, double DECRATE )
```

Arguments

- **double RARATE (in)**
  Description: the new RA differential tracking rate.
  Unit: radian/day
  Range or possible values:
- **double DECRATE (in)**
  Description: the new DEC differential tracking rate.
  Unit: radian/day
  Range or possible values:

Return value

- **result** RESULT (returned)
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.
- Presets must be active.
- The existing targets must be non-sidereal.

After execution

- The non-sidereal target data is updated.

Example

```
...
iifs::result res = iif->UpdateNonSiderealTarget(12.123, -10.002);
```

8.67 UpdatePointingReference

Description

The UpdatePointingReference command saves changes generated by the OffsetPointing command. If the OffsetPointing command has been used to modify the pointing origin (aka hotspot) and/or the angle associated with the rotator mode, UpdatePointingReference can then be invoked in order to save the current coordinates of these modified values so they may be referenced by successive OffsetPointing commands issued against the same PresetTelescope target. The saved values are reset when a new PresetTelescope command is issued.

Syntax

```
result UpdatePointingReference( string OFFSETTYPE,
                                string SIDE )
```

Arguments

- **string OFFSETTYPE (in)**
  Description: the offset type.
  Unit: none
  Range or possible values: See 6.4.2
  Notes: RADEC, AZALT, and GALACTIC update the sky coordinate reference of RA/Dec, Az/Alt, or L/B respectively. DETXY updates the pointing origin coordinate reference of X/Y.
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result** RESULT (returned)
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- Saves the current coordinates of modified values so they may be referenced by successive OffsetPointing commands.

Example

```
...
iifs::result res = iif->UpdatePointingReference("RADEC","left");
```

8.68 UpdateTargetWavelength

Description

This command updates the effective wavelength of the target without issuing a new preset.

Syntax

```
result UpdateTargetWavelength( float WAVELENGTH,
                               string SIDE )
```

Arguments

- **float WAVELENGTH (in)**
  Description: the new effective wavelength of the target.
  Unit: microns.
  Range or possible values: [0.3, 15.0]
- **string SIDE (in)**
  Description: the command side.
  Unit: none
  Range or possible values: "left" | "right" | "both"

Return value

- **result RESULT (returned)**
  Description: result structure received from the TCS. See section 7.1.

Preconditions

- The instrument must be authorized.

After execution

- The target wavelength kept by the PCS is updated.

Example

```
...
iifs::result res = iif->UpdateTargetWavelength(0.8,"left");
```

9 Full examples

The following code shows the entire process used to communicate with the TCS: initializing the IIF, sending commands to the 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 (iif/instrument/examples).

9.1 Synchronous example

```
#include <Ice/Ice.h>
#include <iif/ice/Factory.hpp>

#define INSTR_ID "LBC"
#define FOC_STATION "prime left"
#define COMMAND_SIDE "left"

// Just to identify the clients.
#define CLIENT_PROXY_NAME "LBC left"


// commands
#define TIPTILT
#define UPDATEWL


using namespace std;
using namespace iifs;

// to print out the result messages coming from TCS
void showResults(const iifs::result _res, const string _cmd);

int
main(int argc, char* argv[])
{
    int status = EXIT_SUCCESS;

    /// Ice communicator
    Ice::CommunicatorPtr communicator;

    /// result object
    iifs::result res;

    try
    {
        communicator = Ice::initialize(argc, argv);
        FactoryPrx factory =
           FactoryPrx::checkedCast(communicator->stringToProxy("Factory:tcp -p
10000"));
        if(!factory) throw "Invalid proxy: IIF Server not found.";
```

```
      /// Get a proxy object for this instrument.
      /// If the proxy already exist, the arbitrator will link the proper IIF
instance
      /// with it.
      /// Otherwise, it will create a new IIF instance for this client.
      iifs::IIFServerPrx iif = factory->create ( CLIENT_PROXY_NAME, FOC_STATION,
INSTR_ID );
      if ( !iif ) throw "Invalid proxy: Invalid instrument/focal station
combination or invalid side.";

      ///Check for authorized
      res = iif->Authorize();
      showResults(res,"Authorize");
      if (res.rescode != EXIT_SUCCESS) throw "Error: Instrument not authorized.";

      #ifdef TIPTILT
      ///TipTilt
      res = iif->TipTilt(0.0009, 0.0001, "M1", COMMAND_SIDE);
      showResults(res,"TipTilt");
      #endif

      #ifdef UPDATEWL
      ///UpdateTargetWavelength
      res = iif->UpdateTargetWavelength(0.69, COMMAND_SIDE);
      showResults(res,"UpdateTargetWavelength");
      #endif

      /// Destroy the IIF instance from the factory when finish the
      /// client execution (optional)
      factory->destroy(iif);

      /// close the communicator
      if(communicator) communicator->destroy();
    }
    catch(const Ice::Exception& ex)
    {
        cerr << ex << endl;
        status = EXIT_FAILURE;
    }

    catch (const char* msg)
    {
          cerr << msg << endl;
          status = EXIT_FAILURE;
    }

    catch (string msg)
    {
          cerr << msg << endl;
          status = EXIT_FAILURE;
    }
```

```
    return status;
}

void showResults(const iifs::result _res, const string _cmd)
{
      if (_res.rescode == 0) cout << _cmd << " command status: SUCCESS" << endl;
      for ( unsigned int i=0; i< _res.resmsg.size(); i++)
            cout << _res.resmsg[i] << endl;
}
```

9.2 Asynchronous example

The following code shows the entire process in order to communicate with the TCS: initializing IIF, sending **asynchronous** commands over the instrument interface and printing the results and error messages to the standard output (stdout). Please see reference [14], AMI calls to clarify the process.

```
#include <Ice/Ice.h>
#include <iif/ice/Factory.hpp>

#define INSTR_ID "LBC"
#define FOC_STATION "prime left"
#define COMMAND_SIDE "left"

//!Just to identify the clients.
#define CLIENT_PROXY_NAME "LBC left"


//!commands
#define TIPTILT
#define UPDATEWL


using namespace std;
using namespace iifs;

// Class to call an asynchronous tiptilt.
// We can create a template-based class here and use it for every command.
class AMI_IIFServer_TipTiltI : public iifs::AMI_IIFServer_TipTilt
{
  public:
        virtual void ice_response(const iifs::result & res)
        {
              if (res.rescode == 0) cout << "TipTilt command status: SUCCESS" <<
endl;
              else cout << " TipTilt command status: FAILED" << endl;
              for ( unsigned int i=0; i< res.resmsg.size(); i++)
                    cout << res.resmsg[i] << endl;
        }

        virtual void ice_exception(const Ice::Exception & ex)
        {
        }
```

```
};

int
main(int argc, char* argv[])
{
    int status = EXIT_SUCCESS;

    /// Ice communicator
    Ice::CommunicatorPtr communicator;

    /// result object
    iifs::result res;

    try
    {
        communicator = Ice::initialize(argc, argv);
        FactoryPrx factory =
           FactoryPrx::checkedCast(communicator->stringToProxy("Factory:tcp -p
10000"));
        if(!factory) throw "Invalid proxy: IIF Server not found.";

      // Get a proxy object for this instrument.
      // If the proxy already exist, the arbitrator will link the
      // proper IIF instance with it.
      // Otherwise, it will create a new IIF instance for this client.
      iifs::IIFServerPrx iif = factory->create ( CLIENT_PROXY_NAME, FOC_STATION,
INSTR_ID );
      if ( !iif ) throw "Invalid proxy: Invalid instrument/focal station
combination or invalid side.";

      // Check for authorization
      res = iif->Authorize();

      #ifdef TIPTILT

      // Asynchronous TipTilt, create a callback.
      AMI_IIFServer_TipTiltPtr cb = new AMI_IIFServer_TipTiltI;
      iif->TipTilt_async(cb, 0.0009, 0.0001, "M1", COMMAND_SIDE);

      #endif

      #ifdef UPDATEWL

      ///UpdateTargetWavelength
      res = iif->UpdateTargetWavelength(0.69, COMMAND_SIDE);

      #endif

      ///close the communicator
      if(communicator) communicator->destroy();
    }
    catch(const Ice::Exception& ex)
    {
```

```
        cerr << ex << endl;
        status = EXIT_FAILURE;
    }

    catch (const char* msg)
    {
          cerr << msg << endl;
          status = EXIT_FAILURE;
    }

    catch (string msg)
    {
          cerr << msg << endl;
          status = EXIT_FAILURE;
    }

    return status;
}
```

10 References

[1] J. Borelli, “C++ Instrument Interface Control Document”, LBT document CAN 481s011.

[2] C. Biddick, A. Lovell-Troy, “Instrument Team IIF Testing Instructions”, LBT document CAN 481s265 (Draft: July, 2006)

[3] L. Fini, A. Puglisi, L. Busoni, “AOS The Complete Guide”, LBT document CAN 481f341 (July, 2009)

[4] T. Sargent, “Instrument Rotator Control SW Specification”, LBT document CAN 678s001 (December, 2006)

[5] J. Borelli, T. Schmelmer, “Instrument Interface User Guide”, LBT document CAN 481s261 (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 481s501 (July, 2006)

[13] Wiki page describing the operational status of the set of commands
http://wiki.lbto.arizona.edu/twiki/bin/view/TCS/IIFSupportStatus

[14] Michi Henning, Mark Spruiell, “Distributed Programming with Ice”, Zeroc.com. Version 3.2.1

11 Appendix A: IIF commands status

IIF Command	Section	Status	Comments
AcquireRefAO	8.1	Operational	
Authorize	8.2	Operational	
BinocularControl	8.3	Operational	ADJUSTBALANCE not supported
CheckRefAO	8.4	Operational	
ClearHotspot	8.5	Operational	
ClearNonSidereal	8.6	Operational	
ClearOffset	8.7	Operational	
ClearReference	8.8	Operational	
ClearStars	8.9	Operational	
CorrectModesAO	8.10	Operational	
GetKFPCoordinates	8.11	Operational	
GetKFPCoordinates2	8.12	Operational	
GetParameter	8.13	Operational	
GetRotatorPolynomials	8.14	Operational	
GetRotatorTrajectory	8.15	Operational	LBC only
LogEvent	8.16	Operational	
MaximizeWrapTime	8.17	Under development	
ModifyAO	8.18	Operational	
Move	8.19	Operational	Does not support AGW guiding or AO
MoveFocus	8.20	Operational	This command does not work as desired since there is no concept of ‘absolute focus’
MoveXY	8.21	Operational	Does not support AGW guiding or AO
MoveXYZ	8.22	Operational	Does not support AGW guiding or AO
OffsetGuiding	8.23	Operational	Only for LBC and LBTI
OffsetPointing	8.24	Operational	Does not support OPE (always mount), and new position flag.
OffsetPointing2	8.25	Operational	Does not support OPE (always mount), and new position flag.

OffsetXYAO	8.26	Operational	
OffsetZAO	8.27	Operational	
PauseAO	8.28	Operational	
PauseGuiding	8.29	Operational	
PresetAO	8.30	Operational	
PresetFlatAO	8.31	Operational	
PresetTelescope	8.32	Partly Operational	Does not support multiple guidestars, and gravity or native rotator modes. Does not support coordinate mode COORD_FOCAL_MM. Does not support equinox ICRS. Only TRACK, GUIDE and ACTIVE modes supported for non-sidereal target.
RefineAO	8.33	Operational	
Remove	8.34	Operational	
ResumeAO	8.35	Operational	
ResumeGuiding	8.36	Operational	
RotateCommon	8.37	Operational	Does not support AGW guiding or AO
RotatePrimary	8.38	Operational	Does not support AGW guiding or AO
RotateZ	8.39	Operational	Does not support AGW guiding or AO
RotHold	8.40	Operational	
RotReady	8.41	Operational	
RotServicePosition	8.42	Operational	
RotTrack	8.43	Operational	
RunAO	8.44	Operational	
SendWavefront	8.45	Operational	
SetAGWFilter	8.46	Operational	
SetGuidingBinningt	8.47	Operational	
SetGuidingHotspot	8.48	Operational	
SetHotspot	8.49	Operational	
SetNonSiderealt	8.50	Operational	

SetOffset	8.51	Operational	
SetOffset2	8.52	Operational	
SetParameter	8.53	Operational	
SetPMTerm	8.54	Operational	
SetPMTerm2	8.55	Operational	
SetReference	8.56	Operational	
SetReference2	8.57	Operational	
SetStars	8.58	Operational	
SetStars2	8.59	Operational	
SetTarget	8.60	Operational	Deprecated in favor of SetStars
Standby	8.61	Operational	No current TCS use
StartAO	8.62	Operational	
StepFocus	8.63	Operational	
StopAO	8.64	Operational	
TipTilt	8.65	Operational	Does not support AGW guiding or AO
UpdateNonSiderealTarget	8.66	Operational	Only supports differential tracking
UpdatePointingReference	8.67	Operational	
UpdateTargetWavelength	8.68	Operational	

Doc_info_start
Title: The ICE Instrument Interface Control Document
Document Type: Technical Manual
Source: Steward Observatory
Issued by: Jose *L. Borelli*
Date_of_Issue: *19-Jan-2009*
Revised by: *Chris Biddick*
Date_of_Revision: *05-Jul-2016*
Checked by:
Date_of_Check:
Accepted by:
Date_of_Acceptance:
Released by:
Date_of_Release:
File Type: *MS Word*
Local Name: *481s013u.docx*
Category: *400*
Sub-Category: *480*
Assembly: *481*
Sub-Assembly:
Part Name:
CAN Designation: *481s013*
Revision: *v*
Doc_info_end