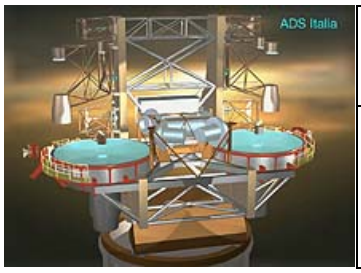


LBT PROJECT
2x8,4m TELESCOPE

Doc.No. : 678s001
Issue : f
Date : 20mar2007

LBT PROJECT

2 X 8,4m OPTICAL TELESCOPE

Preliminary Instrument Rotator Control Software Specification

	Signature	Date
Prepared	Tom Sargent, M. DeLa Pena, J. Kraus, D. Cox	Nov. 30, 2006
Reviewed	Joar Brynnel, Norm Cushing, M. De LaPena, R. Meeks, D. Ashby, D. Cox, J. Rosato, M. Gusick, J. Kraus, C. Biddick	Dec. 6, 2006
Approved		

1. Revision History

Issue	Date	Changes	Responsible
a	07-aug-06	First draft	Tom Sargent
b	18-aug-06	First revision after review	Tom Sargent
c	09-oct-06	Revision after further hardware definition	Tom Sargent
d	30-nov-06	Revision after further discussion & design	Tom Sargent
e	02-jan-07	Revision after further hardware design	Tom Sargent
f	20-mar-07	Revision for Rotator Design Review	Tom Sargent

2. Table Of Contents

3. List Of Abbreviations

DSP – Digital Signal Processor.

ERPLC – The Enclosure Rotation PLC. This controls movement of the building.

MCSPU – Mount Control System Processor Unit.

MJD – Modified Julian Date

OSS – Optical Support Subsystem

PCS – Pointing Control System.

RPC – Remote Procedure Call. The software protocol that is used in the LBT TCS to allow a program to invoke command functions in another program.

TCS – Telescope Control System.

TPLC – The telescope PLC. Controls many aspects of telescope operation, but not rotators or main axis drives.

4. About this document

The Instrument Rotators are a part of the LBT that will provide for de-rotation of various instruments attached to the telescope. The Rotators are developed as an internal LBTO project. The control software for the rotators runs in certain TCS subsystems and in the MCSPU computer. This document defines requirements for development of all of the LBT rotator software.

4.1. Purpose

This document was written to specify the rotator software control structure, its interfaces, and the command set.

4.2. Reference Documents

[RD1] CAN 483s199 Engineering Interface for the Mount Control System Control Program.

[RD2] CAN 481s001d LBT Software System Definition Revision 2.0

[RD3] CAN 670s007 Instrument Rotators and Cable Chain Detailed Description.

5. Introduction

The physical rotator hardware consists of one instrument rotator and an accompanying rotating cable chain for each of 4 different instrument focal stations on each side of the LBT. Additionally, the middle bent Gregorian rotators are dual units, having a second "inner" rotator mounted to the first, "outer" rotator. This gives a potential total of 10 instrument rotators which must be controlled. Their names are shown in the table below:

Description	Name	Description	Name
Left Direct Gregorian	LDG	Right Direct Gregorian	RDG
Left Front Bent Gregorian	LFBG	Right Front Bent Gregorian	RFBG
Left Center Bent Greg. Outer	LCBGO	Right Center Bent Greg. Outer	RCBGO
Left Center Bent Greg. Inner	LCBGI	Right Center Bent Greg. Inner	RCBGI
Left Rear Bent Gregorian	LRBG	Right Rear Bent Gregorian	RRBG

Each rotator, except for the dual units, also has an associated cable chain which handles the power and data cables, and cryogenic lines which connect to the instrument. The MCSPU computer turns the cable chains on and off and monitors their operation for error conditions, but does no actual motion control.

Normally, it is anticipated that one rotator (two in the case of the dual units) will be in use on each side of the telescope. At the same time other rotators may need to be moved for engineering or instrument maintenance purposes.

The high level TCS system or an instrument can originate the control commands to the rotators. The commands are passed on to a middle layer of "soft real-time" control software - the MCSCP (Mount Control System Control Program) - which runs in the MCSPU computer. The MCSPU software, in turn, communicates with the lowest level, "hard real-time", control software running in dedicated DSP processors which actually control the rotator hardware. See figure 1 below.

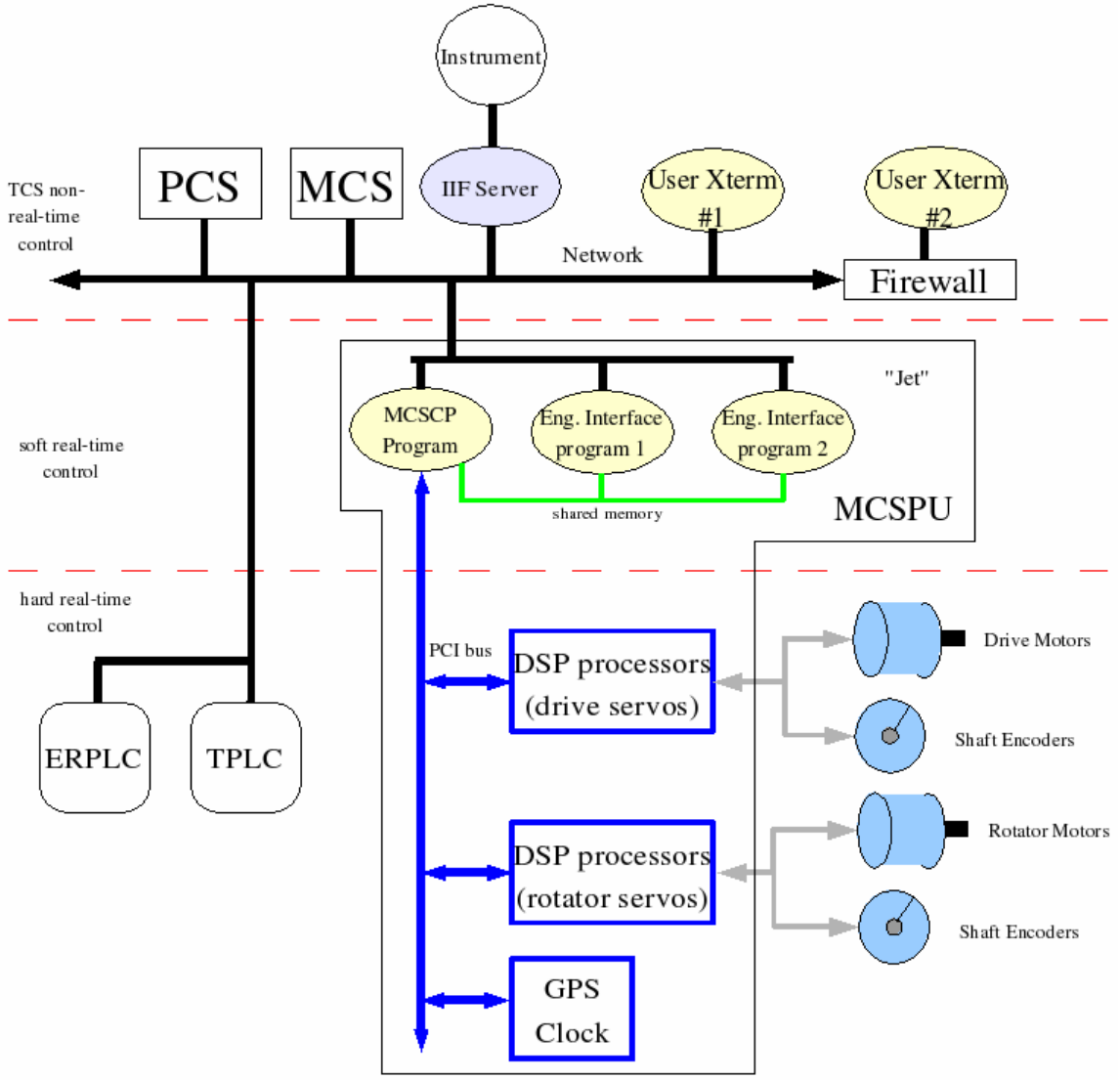


Figure 1 LBT Control Structure

The top level software runs on computers in the control room. The MCSPU, "jet", is mounted in the drive control rack in the auxiliary control room. The various DSP processors are in the same PCIbus card cage with the MCSPU processor, so they communicate with MCSPU over its PCIbus.

6. Telescope Control System

The LBT Telescope Control System (TCS) is a high-level application comprised of several major software components: subsystems (software which controls hardware through an interface chain), controllers (compute engines or mediators), and their associated GUIs. These components are layered on a services infrastructure which

provides fundamental functionality needed by all of the high-level tasks. The subsystems are the software representations of major telescope hardware (e.g., primary mirror cell subsystem PMC, enclosure control subsystem ECS, guiding control subsystem GCS). The controllers (e.g., pointing control subsystem PCS, point spread function PSF, CSQ) either compute data needed by other subsystems and/or act as the logical routers of commands destined for particular subsystems. If appropriate, the subsystems and controllers have associated GUIs which allow for operator control of much of the telescope functionality.

6.1. Instrument Interface and the CSQ

The Instrument Interface (IIF) is the software interface which allows the instrument software to communicate with the TCS, through its CSQ component (CSQ = IIF Server), in order to issue commands which provide control and acquire status information. During actual operations only a single instrument can have full control of a telescope side; this control is obtained through the authorization command. When an instrument is authorized for a telescope side, it is the only instrument which may successfully issue a majority of the commands. However, there are a limited number of commands which can be safely issued by a non-authorized instrument.

Under normal operating conditions, it is envisioned that only one instrument rotator per telescope side would be enabled (i.e., running) at a time. However, for testing or calibration purposes, there is no reason to inhibit (from a software perspective) the number of rotators which can be running on a telescope side at once. As such, a new IIF command needs to be defined which can be used primarily by instruments which are NOT authorized for any telescope side in order to allow them to control their rotators for calibration or other purposes. While the authorized instrument could use this new command, under nominal operating conditions, it is envisioned that the authorized instrument would enable its rotator through the use of the "preset" command, and disable its rotator via the deauthorization command which yields the instrument authorization. It should be noted the telescope operator (TO) also has access to these same commands via the operator's GUI, allowing the TO to enable/disable any instrument rotator at any time it is safe to do so, regardless of the state of any instrument.

To clarify, when an instrument requests authorization to control a telescope side, the instrument would not automatically have its corresponding rotator enabled just based upon the authorization request. Rather the instrument software would then issue the "preset" command, as is done now, to set up an observation. Part of the observation set up includes the specification of the initial rotator angle and a rotator tracking mode (e.g., keep angle relative to North, keep angle relative to the parallactic angle, and no rotator use). The underlying code in the CSQ which supports the "preset" command would enable the appropriate rotator for the authorized instrument. Contemporaneously, a non-authorized instrument could use the new IIF rotator command to indicate to the TCS that the specified rotator should be readied for use.

The new IIF command has a function prototype of the form:

```
Result *result =
setRotator (string focalStation, string side, bool enable,
            [optional argument], [optional argument])
```

The *focalStation* argument is a string from the list currently specified in the IIF: 'directGregorian', 'bentGregorianFront', 'bentGregorianBack', or 'bentGregorianCenterOuter'. The *side* is a string from the list currently specified in the IIF: 'left', 'right', or 'both'. *Enable* is either true or false and determines whether to activate or deactivate the rotator associated with the specified focal station. The optional arguments are contingent upon the instrument requesting the TCS to enable the rotator. These optional arguments could include a *mode* (e.g., 'position': angle relative to North, 'vertical': angle relative to the parallactic angle, 'gravity': angle relative to the gravity vector) and associated values needed in support of the *mode*.

Returns: Result **result* is a structure containing an integer result value, text describing the result code, and a request handle which can be used to query for further information.

Lacking a complete set of requirements, only a few other new IIF commands can be anticipated at this time. These commands would include the set described below where *result*, *focalStation*, and *side* have been described for the setRotator command above.

```
Result *result =
forceRotWrap (string focalStation, string side, integer force)
```

The *force* argument has possible values of: -1 = negative (counter-clock wise) wrap, 0 = no preference (use the default which is to follow the shortest path), and 1 = positive (clock wise) wrap.

`Result *result =` trackRot (string focalStation, string side)

This command activates the tracking action for the specified rotator.

```
Result *result = holdRot (string focalStation, string side)
```

This command puts the specified rotator in a hold state. Its drive motors stop moving. Tracking polynomials are ignored.

```
Result *result = adjustRotPosition (double deltaRA, double deltaDec,
double deltaRotAngle)
```

This command allows the instrument to input fine adjustments to the current RA, Dec, and rotator angle values (in order to position an image on a detector, for instance).

Finally, a means to determine the amount of time left on the instrument and azimuth cable wraps, as well as a time to interception with the horizon has been requested. This information could be available to an instrument via an IIF command or

as a standalone observing tool. The command could have a function prototype as follows:

```
Result *result = timeToIntercept (string focalStation, string side,
double RA, double Dec, float exposureLength)
```

The *result*, *focalStation*, and *side* have been described for the setRotator command above. The input *RA*, *Dec*, and *exposureLength* would be used in conjunction with the currently known telescope state parameters to compute estimates for the time remaining on the instrument and azimuth cable wraps before an unwrap procedure must be done, as well as an estimate of the amount of time remaining before the specified target dips below the horizon.

The time to intercept for the object currently under observation will also be placed in reflective memory by the PCS so it will be available to instruments and GUIs.

6.2. Pointing Control Subsystem (PCS)

The primary role of the PCS in the context of the instrument rotators is to generate the needed polynomials to allow an instrument at a specified focal station either to track on a requested astronomical target or to control its rotator for calibration purposes (for example). Under normal observation conditions, it is envisioned that only the rotator(s) at one focal station will actually be enabled on a side; the focal station that is associated with the authorized instrument. However, in the extreme case, all 5 rotators on a side could be active simultaneously. It should be noted that the prime focus focal stations are outside the scope of this discussion.

6.2.1. PCS Communication with MCSPU

The specific design and implementation of the software to support the rotator control, in particular the communication between the PCS and the MCSPU control process, is contingent upon the evaluation of the performance of two possible scenarios. Further, the discriminators for the scenarios are the specific requirements imposed by the instruments (e.g., how often do the instrument rotators need to be updated, how often will multiple rotators on a side be requested).

The preferred scenario is for the PCS to issue a single remote procedure call (RPC) to the MCSPU control process no faster that once every 50 ms. This single RPC would contain the polynomials for all ten rotators on the telescope, in a specified order. Enabled rotators would be distinguished by non-zero timestamps and corresponding coefficients. In contrast, the disabled rotators would have timestamps and coefficients equal to zero. The advantage of this scenario is that it minimizes the number of RPCs which need to be issued and interpreted by the MCSPU control process and the network bandwidth. The disadvantage is that the MCSPU control process must parse all of the timestamps being sent to determine the enabled rotators. However, this is minor compared to RPC overhead. Also, the PCS may be computing up to ten polynomials in a single thread, requiring the computations to be consecutive.

The alternative scenario for PCS is to run independent threads for all enabled rotators and issue separate RPCs comprised of polynomials only for each rotator which

must be updated. The advantage of this scenario is that the PCS threads can be executing simultaneously, and hence, the polynomials are computed simultaneously. From the PCS side, the implementation is straightforward in that each thread would be running a different instance of a generalized rotator class. The disadvantages are the additional thread overhead, the increased number of RPCs which could be issued and processed by the MCSPU control process in the worst case scenario and the impact on the network bandwidth. Of course, both of these scenarios have implementation variations in the details which may be explored as necessary.

The concerns for the proposed scenarios are the number and frequency of the RPC calls which must be issued. In order to determine the most appropriate method of communication between PCS and the MCSPU control process, it is intended the scenarios be implemented as streamlined prototypes.

In order to support the PCS functionality, new PCS data dictionary (aka reflective memory) variables need to be created which indicate which rotators are enabled. Since there are a limited number of rotators, the nomenclature used for the variables will correspond to the focal station designation for explicit identification, rather than the use of an array structure.

6.2.2. PCS Polynomial Format

A polynomial as computed by the pointing kernel portion of the PCS consists of an epoch (a starting time) called T0, and the 3 coefficients which define the second order polynomial. The T0 is a time expressed as a modified Julian Date (MJD) using the International Atomic timescale. The T0 is converted by PCS from MJD days to TAI seconds for the convenience of the low-level MCSPU code. The units of the three coefficients are radians, radians/sec and radians/sec/sec. The mount servo computes the instantaneous position demand by subtracting T0 from the current time (which it reads from the GPS interface) and evaluates the polynomial using that time difference. The rotator position in radians as a function of time is:

```
angle(t) =  a0 + a1*(t-T0) + a2*(t-T0)**2
```

where a0, a1, and a2 are the coefficients of the polynomial. So, a polynomial is described by a list of 4 double precision floating point numbers. The polynomials are normally issued by the PCS at a regular rate which will not exceed 20/sec. If the rotator software goes for more than 1 second without receiving a new polynomial, it considers this is an error condition and automatically switches the rotator to hold mode. To resume tracking in this case, the operator has to make PCS resume issuing polynomials and then command the rotator to start tracking again.

.All polynomials are computed by the PCS in double precision (IEEE 64 bit) floating point arithmetic. However, it is clear that single precision floating point numbers (IEEE 32-bit) are adequate for the servo to use for moving the rotators. The DSP processors perform double precision arithmetic much more slowly than single precision. So, it is required that the rotator polynomials (unlike the main drive polynomials) be converted to single precision before being sent to the DSP servo software

7. MCS Rotator Software

The MCS is the TCS system's interface to the mount and rotator control software (as well as swing arms, dynamic balance, stow pins, etc.). In reality the MCS subsystem does little but serve as a TCS proxy for the MCSPU (see Figure 1). It queries the MCSPU for status information several times per second which it records in reflective memory, passes commands from TCS down to MCSPU, and interfaces with the system's event logging facilities. The MCSPU has responsibilities which are more real-time than those of the TCS, so, since MCS and MCSPU execute on 2 different computers, MCSPU is able to offload most of the TCS overhead activity onto the MCS computer. It is really the MCSPU that is doing the control work. For this reason this document refers generally to the actions of the MCSPU rather than the MCS.

The MCSPU instantiates one Rotator class object for each rotator on the telescope. There are 5 rotators on each side of the LBT for a total of 10 such objects. (It is conceivable that 2 additional prime focus rotators might be added to the system at some future time. From a software standpoint, expanding the number of rotators at a later time would be relatively easy. However, no such prime focus rotators are anticipated at this time.)

The Rotator class has a state machine that handles the sequencing of rotator start-up and shut-down procedures, watches for various error conditions and reports status to the TCS.

Each Rotator object also instantiates one Tracker class object and one CableChain object. The Tracker is a state machine which handles all rotator movement. It receives and responds to the tracking polynomials that are generated by the PCS and handles slewing, and holding, and all communication with the servo hardware that drives the rotator. The CableChain object has a state machine that starts and stops the cable chain drive and monitors its performance. The DSP runs servo software that controls the cable chain motion, making it follow the rotator. So, the MCSPU level software does not command any cable chain motion. It just starts the cable chain servo and expects the servo to make the cable chain always follow the rotator.

7.1. Rotator States

The Rotator state machine is used to turn the rotator and its cable chain on or off and to monitor them during normal operation. Turning the whole rotator system on or off is a multi-step process. The major states (there will be many other states which are very transient) of this state machine are:

WAIT_OPR: Power supplies and power amplifiers are off. Brakes are on. State machine is waiting for a command from the operator to turn on the rotator and make it ready to run. Tracker is held in HOLD state at this time. The operator can command it to go from WAIT_OPR to IDLE or READY state.

IDLE: The power supplies are on and power amplifiers are disabled. Brakes are on. Operator may command it to become READY or go to WAIT_OPR.

READY: The rotator is in a condition ready to operate. It may be holding, slewing or tracking depending on the state of the Tracker state machine described below. While in this state, the state machine monitors the rotator looking for hardware error conditions that may arise. The operator may command it to the IDLE state or WAIT_OPR state.

IDLE_FAULT: Some fault has occurred and the state machine is holding with amplifiers disabled and brakes on. The power supply may be on or off. The idea is to hold the hardware in a faulted state so the operator can determine what has happened (as opposed to clearing status and error bits and reverting to WAIT_OPR when a fault occurs).

7.2. Tracker States

These are the main Tracker states. The Tracker object controls the rotator motion.

STOPPED: The state machine is not sending any polynomials to the rotator servo. The Rotator state machine may be in any state.

HOLDING: Rotator is holding position. Tracker is ignoring polynomials from the PCS (there may be none if PCS is not running) and is generating its own polynomials which it sends to the servo. These specify zero velocity and zero acceleration. The Rotator state machine must be READY for the Tracker to be in this state.

SLEW_TO_TRACK: The Tracker is generating its own polynomial stream which is running the rotator to intercept the target that is currently being described by the PCS polynomials. (This is generally a moving target.) When it gets to the target position it will automatically transition to TRACKING state. The Rotator state machine must be READY for the Tracker to be in this state.

SLEW_TO_HOLD: Polynomials being sent by the PCS (if any) are being ignored. The Tracker is generating its own polynomial stream to move the rotator to a new commanded position. When it arrives at that position, it will stop there and transition to HOLDING state. The Rotator state machine must be READY for the Tracker to be in this state.

TRACKING: Tracker is passing polynomials received from PCS on to the Servo which runs on the DSP processor. The rotator is probably tracking an astronomical object. Polynomials which are clearly

invalid (outside the legal bounds for position acceleration or velocity) are ignored. If Tracker goes more than 3 sec. without receiving a polynomial or a valid polynomial, it will transition to HOLDING state. The Rotator state machine must be READY for the Tracker to be in the TRACKING state.

VEL_MODE: The servo is in velocity mode (as opposed to the normal position mode operation). In this mode, the servo strives to maintain a specified rotator velocity, rather than a specified rotator position. Polynomials received from PCS (if any) are ignored. No polynomials are sent to the servo DSP. The operator may command the velocity. This is normally for diagnostic or maintenance purposes. Tracking of astronomical objects should not be attempted with the servo in this mode. Commanding the rotator into velocity mode always sets the commanded velocity to 0, so that is a safe, graceful way to make the rotator decelerate to a stop. The Rotator state machine would normally be READY for the Tracker to be in this state.

7.3. Cable Chain States

The cable chain's purpose is to follow the rotator as best it can and it does this automatically as far as the MCSPU is concerned. The cable chain never knows where the rotator is going or why or when it is going to stop. It just tries to keep up with it. For this reason, the cable chain needs no control attention from the MCSPU once it is up and running, but it does need monitoring (to see if it is operating normally) and needs to be turned on and off. The CableChain object has a simple state machine which performs these functions. Its major states (there will be other, very transient states) will be:

WAIT_OPR: Power supplies and power amplifiers are off. The state machine is waiting for a command from the operator to turn on.

READY: The CableChain is turned on and is servoing off of the difference angle between the rotator and the cable chain.

IDLE_FAULT: Some fault has occurred. It stays in this state until the condition is corrected or it is commanded to turn off.

7.4. MCSPU Rotator State Transitions

The operator may command the rotator to hold, track (if PCS is providing polynomials) or slew at any time as long as it is powered up and not disabled by some hardware condition.

Note that this state behavior allows the rotator software to run if the TCS is not running. Like the main drive software, the rotator control software can be used with or without the TCS. With no polynomials being received from PCS, the rotator cannot do tracking, but it can do everything else. If the TCS is not running, then the MCS GUI cannot be used to control the rotators. In this case operator control of the rotator system is achieved by using the Engineering Interface software. See [RD1].

In the event that the flow of tracking polynomials for a rotator stops, (see TRACKING state above) that rotator will be stopped. This will not have any effect on main axis drives, as long their flow of polynomials remains normal. Error status bits reflecting this will be communicated to the MCS (within a 100 – 250 milliseconds), and via reflective memory to the rest of the system. If the system continues to run otherwise, the rotator position error will eventually get large enough so that the rotator on-source flag will also go false. An instrument that is using a rotator should monitor that rotator's on-source flag.

7.5. Rotator Position Information

The MCS will make the rotator positions be available to the TCS (through reflective memory) in two forms. The first is degrees relative to the telescope structure and the second is degrees relative to gravity. The rotator can also be commanded to a service position which is described by an angle relative to the telescope structure or by an angle relative to gravity. These commands, `rslewToHold` and `rslewToHoldGrav`, are described in section 10 below.

7.6. Software Interaction with Limit Switches and Brakes

Each rotator and cable chain has multiple limit switches which control the maximum allowed clockwise and counterclockwise rotation. Other limit switches control the maximum angular difference between a rotator and its cable chain.

The cable chains and the rotators themselves have a "proximity limit" switch at the normal end of CW and CCW rotation. The proximity limits are monitored by the DSP processor. A couple degrees beyond that is an "emergency limit" which is not dependent on software to function. When any proximity limit is hit, the DSP software will stop the rotator and cable chain. If an emergency limit is hit, relay logic turns off power to the drive

An LVDT is used to measure the angular difference between the rotator and the cable chain. The difference angle also has proximity limits which are set at +-2.5 degrees. A fraction of a degree beyond the proximity limits are emergency limit switches.

When the MCSPU software detects a limit switch being hit, it will command the motors to stop and set the brakes. The system does not depend on the MCSPU software to stop it when it hits any limit. This behavior is redundant, but harmless. In the extremely unlikely event of just the right type of failure, this redundancy might get the rotator stopped when the DSP software or the relay logic alone would not. Otherwise, it serves to keep the MCSPU logic in sync with the rotator state.

The rotators have a total normal operating range from -90 degrees to +450 – a total of 540 degrees of rotation. So, for a given starting point and ending point, there may be both a clockwise and a counterclockwise path to the end point. The portion of the range from 0 to -90 deg. is called the negative or CCW wrap and the portion from 360 to 450 deg. is called the positive or CW wrap. The software provides two methods of dealing with the wrap. The normal (default) method is to always take the shortest path

from the current location to the destination. The alternate method is to allow the operator or instrument to "force the wrap", that is, to specify that a particular one of the 2 possible paths to the target position should be used. See the "forceWrap" command below.

The cable chain is slaved to follow the instrument rotator. The software does not command the cable chain to move, although it monitors cable chain signals and turns it on and off.

8. Interfaces

The MCSPU rotator software communicates with and mediates between the high level TCS system "above" it, and the low level real-time DSP processors which run the rotator drives "below" it. Figure 1 above illustrates the system architecture. The operator will normally use the high level MCS GUI to control the rotators. He may also use the Engineering Interface if the TCS is not running or for diagnostic purposes. Both communicate with the MCSPU over the local network.

8.1. TCS Interface

All communication between the TCS (RD2) and the MCSPU rotator control software goes over the local network. The MCS subsystem queries the MCSPU several times/second to obtain rotator status so it can be placed in reflective memory. Once in reflective memory, it can be displayed for the operator by the MCS GUI and is available to the rest of the TCS and any instrument. All error bits will be provided as well as a few "summary" error flags which are the logical OR of the many other error bits. This will simplify the task of checking for errors which the instruments might perform.

Commands to the rotator software from the operator or instrument are passed to it over the local network using the same Remote Procedure Calls (RPC) protocol which is used throughout the TCS. Tracking polynomials generated by the PCS are sent directly from PCS to MCSPU via RPC calls. (See section 6)

The rotator status information contains an "on-source" flag which tells the TCS if the rotator position has been within a specified angle tolerance of the target position for at least a minimum specified period of time. Instruments can use this information to decide when to start collecting data or when to stop if the rotator has moved off the source. This is analogous to the AZ-EL on-source signal which is already being provided by the MCSPU. The instrument should monitor both of these on-source signals.

8.2. Operator's Graphic User Interface

The MCS subsystem GUI program runs on one of the control room computers and provides the operator with status display and command control of all parts of the Mount Control System. More than one instance of the GUI can be run simultaneously. Part of this GUI be a Rotator display page. This will be the operator's main interface to the rotator systems.

The GUI will provide forms so that the user, the TO and Observers, can easily observe the state of all the Rotators. In general all necessary variables and status information will come through the TCS Reflective Memory system. There will also be

engineering forms to access the full functionality of Rotator control and error handling through the MCSPU. Some of the features of this system are:

Rotator Graphic: This will be some simple graphics, indicators that give a visual representation of the positions and wraps, of all the rotators. This will provide a simple display and numerical value of where the rotator is with respect to its cable wrap.

State Indicators: This will indicate the current Tracker state; Idle, Tracking, Slewing or Holding for all the rotators.

Detail for any 2 Rotators: This provides a way to select any 2 different rotators for detailed display and control. This will open up an area where interaction to these 2 specific rotators is available. Any 2 rotators may be easily selected.

Sequence Area: Access to main command startup and shutdown sequences and display of their current state variables (Ready, Idle, Wait_Opr, Idle_Fault)

Command Area: Access to all individual rotator commands as defined in this document. Command argument limits will be noted and arguments will be checked for validity. Through these commands the operator will have full control of the rotators and will be able to override instrument control of the rotators.

Service Positions: The GUI will remember a list of rotator service positions unique for each rotator so that the operator can command a rotator to go to a particular orientation. The GUI will allow the operator to add new service positions to the list.

Status area: All status will be available as “good”, “warning” or “in error”.

Memo Area: This scrolling area will list the interaction comments between the GUI and the control system. This will acknowledge commands and display actions and errors.

Update Rate: The GUI will update all Rotator status bits at some reasonable rate (~5 Hz).

8.3. Engineering Interface

As a development and diagnostic tool the Engineering Interface is also available to display rotator status and accept rotator commands. It will have several display pages that show all known information about the rotators. All RPC-callable commands that are available to the TCS can also be executed manually from the Engineering Interface command line. The engineering interface can be used independent of the TCS, so it allows control of the rotators when TCS is not running.

The MCS GUI should afford the operator all the same information and control capability as the Engineering interface and do it in a much more user-friendly way. So, if

the TCS is running, it is assumed that the MCS GUI would be the preferred way of controlling and monitoring the rotators.

8.4. Servo interface

The rotator servo control is comprised of dedicated DSP processors which read the rotator position angle encoders and, using the polynomial described above, compute the instantaneous commanded rotator position. The difference between these is the position error. The DSP's servo algorithms use the position error and velocity information to compute commands to the power amplifiers which power the drive motors. The Tracker class object creates a Servo class object which it uses to handle writing of all commands and polynomials to the servo DSPs and reading of all status information from them. This status information includes things like position angle, velocity, motor torque, limit switch status, brake status, etc. That information is in turn made available to the Rotator class which is queried periodically by the MCS. . The existing Servo class in the MCSPU is being expanded to support the DSP rotator servo software. See the next section for details on the rotator DSP-resident software and section 10 for the Tracker to Servo software interface

The MCSPU software downloads the DSP run-time binary files into the DSP's memory and starts the DSPs executing the code when the MCSPU is initialized. It can also be commanded by the operator to restart the DSPs, which it does by repeating the initialization sequence. Numerous servo parameters, such as rotator maximum angular speed and acceleration, servo gains, etc. are programmable.

9. The Rotator DSP Environment

The design of the interface to the DSP rotator software will follow the conventions initially established for the LBT mount control system (MCS). The Bittware Reef boards used for the rotators and their associated cable chains pose no change to the current implementation. The shared library interface is extended to operate on the new DSPs in the same thread-safe manner as those DSPs on the MCS.

DSPs on the Bittware boards are addressed by the MCS control program (MCSCP) through an opaque pointer called a DSP descriptor, or *dsp_t*. Each function call (aside from *dsp_init()* and *dsp_halt()* functions) require this descriptor to be passed as an argument.

9.1. Initializing the Rotators

Initializing the rotators is handled during the *dsp_init()* call made by the MCSCP to configure and start all defined DSP applications used by the MCS. Once successfully initialized, the MCSCP obtains DSP descriptors for azimuth and elevation axes by calling *dsp_get_proc()*, passing the DSP cluster and processor as arguments. Since two rotators are combined into one DSP, the MCSCP must use *dsp_get_rotator()*, specifying the cluster, bus, and rotator position, either left or right.

```
dsp_t dsp_get_rotator(int cluster, int proc, int pos)
```

where:

cluster: the hardware DSP cluster

proc: the processor on that cluster

pos: the rotator position – 0=left, 1=right

Returns:

NULL: invalid DSP or rotator specified or undefined

Non-NULL: address of DSP descriptor.

9.1.1. The dsp.conf Configuration File

Prior to initializing the library and DSPs, the entire MCS DSP environment must be described and defined. This is accomplished through a DSP configuration file. The name of this configuration file can be passed to *dsp_init()* as the sole argument, though for ease of code maintenance, the passed argument normally NULL, using the default /etc/dsp.conf.

While Linux is able to identify and communicate with multiple DSPs, it is not able to determine which DSP is used for what purpose. Specifically, those DSPs on Bittware Reef Boards (using the FPGA for I/O) have cabling requirements that Linux cannot determine dynamically.

In order to uniquely identify the DSPs which run a pair of rotators, the DSP configuration file is extended to provide an additional keyword to denote the number and, if single, the location of the rotator being used. The *rotator* keyword, as seen in the sample configuration stanza below, can denote one rotator is being used, either *left* or *right*, or *both.*

```
cluster 8 {
  dsp 1 {
        name “FBG”
        app-prefix /home/dsp/bin
        app rotator.dxe
        telemetry-buffers 20
        telemetry-freq 100
        telemetry-enable
        fpga-freq 4000
        rotator both
  }
}
```

With this configuration, a call to *dsp_get_rotator(8,1,0)* would return the *dsp_t* for the rotator on the left.

9.2. Rotator Library Functions and Operations

Library operations and functions on the Linux host are handled asynchronously in the DSP. When the operations to be performed are sufficiently brief, the task can be handled by the DSP in its interrupt handler. For critical sections in the DSP, such as evaluation a polynomial, the DSP application is expected to globally disable interrupts, masking any attempts by the Linux host system to add a new polynomial. When an operation might require waiting on a PCI device to complete an operation or the DSP to return to the idle task, the operation may require an interrupt to set a latch in the DSP, allowing the DSP to handle the operation during its idle task.

9.2.1. Rotator Operations

The following list describes the functions added to the Linux library to support rotators, drives and the cable chains. Operations that follow here can be performed and then checked for completion by the *dsp_get_status()* call.

`int dsp_rotator_power_reset(dsp_t)`

Performs a reset on the specified rotator's three-phase power monitor.

Returns:

0: command successful

DSPERR_EINVAL: invalid command for this DSP.

`int dsp_rotator_drive_reset(dsp_t)`

Performs a reset on the specified rotator and cable chain drives.

Returns:

0: command successful

DSPERR_EINVAL: invalid command for this DSP

`int dsp_rotator_power_enable(dsp_t)`

Turn on the three-phase power to the specified rotator.

Returns:

0: successful

DSPERR_EINVAL: invalid command for this DSP

DSPERR_PWR: low-level error occurred; check status.

`int dsp_rotator_power_disable(dsp_t)`

Turn off the three-phase power to the specified rotator.

Returns:

0: successful

DSPERR_EINVAL: invalid command for this DSP

DSPERR_PWR: low-level error occurred; check status.

`int dsp_rotator_drive_enable(dsp_t)`

Enables the drives on the specified rotator.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified.

DSPERR_EIO: low-level error occurred; check extended error status.

`int dsp_rotator_drive_disable(dsp_t)`

Disables the drives on the specified rotator.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_EIO: low-level error occurred; check extended error status.

`int dsp_rotator_brakes_off(dsp_t)`

Releases the brakes on the specified rotator.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_EIO: low-level error occurred; check extended error status

`int dsp_rotator_brakes_on(dsp_t)`

Engages the brakes on the specified rotator.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_EIO: low-level error occurred; check extended status

`int dsp_get_rotator_status(dsp_t)`

Returns the operating status of the specified rotator, drive and cable chain as a bitmask value.

Returns:

> 0: bitmask with rotator, drive and cable chain status.

DSPERR_EINVAL: invalid DSP specified

DSPERR_EIO: low-level error occurred; check extended status

9.2.2. Cable Chain Operations

Cable chains are controlled independently of the rotator commands, although they use the same DSP descriptor as the rotator to which they are associated.

`int dsp_cable_chain_power_enable(dsp_t)`

Enables power to the cable chain.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified.

DSPERR_EIO: low-level error occurred; check extended status

`int dsp_cable_chain_power_disable(dsp_t)`

Disables power to the cable chain.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_EIO: low-level error occurred; check extended status

`int dsp_cable_chain_enable(dsp_t)`

Enables the cable chain drive.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_EIO: low-level error occurred; check extended status

9.2.3. Rotator Functions

`int dsp_get_rotator_encoders(dsp_t, rotatorEncoder_t *)`

Returns the encoder values from the specified rotator. The encoder values are returned in the *rotatorEncoder_t* structure, defined as follows:

```
typedef struct {
        int motor[2];
        int main[2];
```

```
        int cable;
    }
```

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve address of encoder data in the DSP application.

```
int dsp_get_drive_status(dsp_t, driveStatus_t *, driveStatus_t *, driveStatus_t *)
```

Returns the status of both rotator drives and the cable drive for the specified rotator in the structures defined as follows:

```
typedef struct {
        unsigned int status;
        int direct_current;
        int quad_current;
        int abs_position;
        int amb_temp;
        int heatsink_temp;
        int motor_temp;
        int bus_voltage;
} driveStatus_t;
```

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve address for drive status in the DSP application.

```
int dsp_get_rotator_power_status(dsp_t, powerStatus_t *)
```

Returns the current power status as seen by the low-level FPGA. The status is returned to the caller defined as follows:

```
typedef struct {
        int p24status;
        int p15status;
        int m15status;
        int p5status;
        int m5status;
} powerStatus_t;
```

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for the power status variables in the DSP application.

`int dsp_get_rotator_brake_pressure(dsp_t, int *)`

Returns the brake air pressure of the rotator as sampled by the FPGA.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for the brake air pressure in the DSP application.

`int dsp_set_rotator_encoders(dsp_t, rotatorEncoder_t *)`

Sets the encoders of the rotator described in the *rotatorEncoder_t* structure. Any non-zero values in the structure will be immediately set in the respective rotator encoder.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for setting the rotator encoders in the DSP application.

```
int dsp_get_rotator_torque(dsp_t, int *torque0, int *torque1, int *chain)
```

Returns the torque commands as written to the rotators and the cable chain. These torque commands are retrieved from the servo structures that hold the torque commands and returned to the caller atomically. The rotator torque values are retrieved from the DSP *rotatorOutput_t* structure and the cable chain torque value is retrieved from the *chainOutput_t* structure.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for rotator torques and cable chain torque in the DSP application.

```
int dsp_get_rotator_atomic_position(dsp_t, float *pos, double *time, float *posError)
```

Returns the current position of the rotator and the position error with the current time at which these values were collected. All values are atomic.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the addresses for the position, position error and/or time in the DSP application.

```
int dsp_get_rotator_position(dsp_t, float *)
```

Returns the position current estimated position of the rotator from the servo structure *rotatorInput_t*.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for the current estimated position in the DSP application.

`int dsp_get_rotator_velocity(dsp_t, float *)`

Returns the current velocity from the estimated rate in the servo structure *rotatorInput_t*. The estimated rate is given in radians/sec.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for the current estimated rate in the DSP application.

`int dsp_set_rotator_velocity(dsp_t, float)`

Sets the current velocity for the rotator in the servo structure *rotatorInput_t*. The velocity is given in radians/sec.

Returns:

0: successful

DSPERR_EINVAL: invalid DSP specified

DSPERR_RESOLVE: unable to resolve the address for the commanded rate in the DSP application.

.

9.3. Rotator Telemetry

Rotator telemetry will operate in a manner identical to that which currently exists for the MCS. Configured and enabled rotators will be capable of generating telemetry at a frequency defined either by the *telemetry-freq* keyword in the configuration file or via the *dsp_telemetry_set_frequency()* function. Telemetry for the rotators and cable chains will be forwarded from the DSP to the Linux host system from the *commonBuffer_t* DSP structure, to be passed on by the telemetry thread in Linux to the TCS telemetry subsystem.

10. The Rotator Command Set

At this early stage of development, it is not possible to outline all necessary commands. Those relating to starting and stopping the system, brakes, and various error conditions and limit conditions are not completely defined at this time. However, the majority of the required commands can be defined. The functionality of a rotator is very similar to that of the AZ and EL axes of the telescope, so many of the commands and procedures for the rotators will be taken from the main axis control logic.

It is assumed that the rotators on each side of the telescope might be used in any combination. In other words because Left Center Bent Gregorian is being used does not necessarily imply that Right Center Bent Gregorian is also being used. Further, while one rotator on each side of the telescope is expected to be used at the same time, that does not necessarily preclude operation of other rotators simultaneously.

All commands are executed via RPC calls and so take a C++ string as their argument and return a C++ string as their return value. If the argument string for a command has more than one value in it, the individual arguments in the string should be separated by one or more blanks.

Since there are 10 different rotators, it is inconvenient duplicate each command 10 times. A "hold position" command, for instance could be implemented as 10 different commands, one for each rotator. `rhold_LDG, rhold_RDG, rhold_LFBG`, etc. It is more convenient to have one rotator-hold command and have it take as its first argument the name of the rotator it is intended to address. So the rotator hold command would be of the form `rhold("LDG") or rhold("RFBG")`.

The commands which pass tracking polynomials to the rotator are an exception to this rule. In the interest of minimizing RPC and XML overhead, the PCS will (probably) use a single RPC command to send the polynomials for all 10 rotators directly to the MCSPU. As is done with the AZ and EL drive polynomials, these will be formatted as a binary structure coerced into a string object. This skips the conversion of binary to ASCII on PCS and conversion from ASCII back to binary form in MCSPU. The PCS will use the "rtrack" command, described below, no more than 20 times/second to send this data to the MCSPU which will then distribute it to its various internal state machines.

10.1. MCS Rotator Command List

Below is an alphabetical list of the MCS RPC-callable commands that can control a rotator. These are available to the whole TCS control system. The majority of them will normally be executed only by the PCS and the operator using the MCS GUI. All are also available through the engineering interface.

`string` **`getRotatorReport`**`(string x)`

This is called, normally by the MCS, to obtain a status report on one rotator and its cable chain. The argument is the rotator name. A number of parameters, such as position and speed, rotator state machine status (tracking, slewing, holding), error flags, etc. This will probably be called 3 or 4 times per second for each active

rotator. The status of inactive rotators will normally be queried much less often. It returns the data in an ArgumentList which has been serialized into an XML string. This is a standard LBT Common Software format.

Returns: status information in the form of a string object which contains a serialized ArgumentList or "BAD rotator name" if the rotator name was unrecognized.

`string` **`rForceWrap`**`(string x)`

This gives the operator the ability to force the rotator to use the CW wrap or the CCW wrap instead of always going the shortest way around (which is the default behavior). The first argument in the string is the rotator name. The second is -1, 0, or 1 to indicate the negative (CCW) wrap, no preference (go the shortest way), or the positive wrap (CW), respectively. The default is no preference. If the rotator is at 0 deg. and is commanded to go to 350 deg, it will normally go CCW to a position angle of -10 deg. However, if the forcewrap flag has been set to 1, it will go CW around to 350 deg. The rule is: if forcewrap = 1, any angle < ROT_POS_WRAP_LIMIT (nominally 90 deg) will have 360 degrees added to it. If forcewrap = -1, any angle > ROT_NEG_WRAP_LIMIT (nominally 270 deg.) will have 360 degrees subtracted from it. The forcewrap flag retains its value until the next time it is changed by the operator (or instrument).

Returns: "OK" if arg is -1, 0, or 1. "BAD argument" otherwise or "BAD rotator name" if the rotator name was unrecognized.

`string` **`rHold`**`(string x)`

The argument x contains the name of the rotator (from the table above) for which this command is intended. This commands the rotator to hold its present position relative to the telescope structure. The drive motor is still on, but it stops turning. It will ignore any tracking polynomials being sent to it when in the 'holding' state.

Returns: "OK" if the state machine can transition to HOLDING state. "ERROR" if it did not change to HOLDING. "BAD rotator name" if the rotator identifier was unrecognized.

`string` **`rIdle`**`(string x)`

Commands the rotator whose name appears in 'x' to go to the IDLE state.

Returns: "OK" if the command has been accepted. "ERROR" if the command queue is full or drive is in a state which prevents if from complying (such as having it's motors or power amplifiers turned off). "BAD rotator name" if the rotator name was unrecognized.

`string` **`rLoadEnc`**`(string x)`

The argument string has a rotator name followed by a new encoder value in decimal degrees. It loads the argument into the rotator encoders effectively forcing them to read back the specified value. Note that if the rotator is in the position mode, a large,

abrupt change in position like this can cause the servo to rail in an attempt to achieve the desired target position. This should be done when the servo is in velocity mode.

Returns: "OK" if the argument is within legal range (-88 to 448 degrees). "ERROR" if the argument is out of range. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rMaxAcc`**`(string x)`

Called to specify the max allowed acceleration to be used for slewing (not tracking). Arguments in 'x' are a rotator name followed by an acceleration limit in deg/sec/sec. If the new acceleration limit is accepted, it will appear in the Engineering Interface rotator status display for this rotator. If the argument is out of range, the max acceleration will not be changed.

Returns: "OK" if the argument was within range and has been accepted. "ERROR" if the argument is out of range. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rMaxVel`**`(string x)`

Called to specify the max allowed velocity to be used for slewing (not tracking). Arguments in 'x' are a rotator name followed by a velocity limit in deg/sec. If the new velocity limit is accepted, it will appear in the Engineering Interface rotator status display for this rotator. If the argument is out of range, the max velocity will not be changed.

Returns: "OK" if the argument was within range and has been accepted. "ERROR" if the argument is out of range. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rOffset`**`(string x)`

Specifies an offset to the rotator position. The first argument in the string is the rotator name. The second is the number of arcseconds to be used as an offset. This signed number is added to the encoder reading before the program uses it. This would normally used to enter small adjustments to the nominal position, though it can be used for large adjustments too. This command is best used when the rotator is in the velocity mode (change to vel. mode, do the offset, zero the integrator and change back to position mode.) If this were used when the servo was running in the position mode it would put a step function into the servo error.

Returns: "OK" if the absolute value of the argument is within range. "ERROR: Tracker state should be in velocity mode!" if the Tracker is not in velocity mode. "ERROR: arg. out of range" if the absolute value of the offset is > 334800 arcsec (=93 deg.). "BAD rotator name" if the rotator name was unrecognized.

`string` **`rPosMode`**`(string x)`

This command puts the rotator servo into position mode. This makes it servo on the position read from the encoder (as opposed to the velocity). This is the normal mode of operation. The argument is the name of the rotator which is being commanded.

Returns: "OK" unconditionally.

`string` **`rReady`**`(string x)`

This commands the rotator sequencer state machine to attempt to turn on the rotator and bring it up to a ready state. In the ready state it can respond to any motion command. The argument is the name of the rotator which is being commanded.

Returns: "OK" ready command queued" if the command is accepted. "ERROR queue full! Command ignored" if the command queue is full. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rRec`**`(string x)`

This records rotator position data to an ASCII file named rTrack.txt. Each line in the file has a timestamp relative to the start of the recording, the actual position in arcsec and the commanded position in arcsec. The argument x contains the name of the rotator (from the table above) for which this command is intended plus the number of seconds for which position data should be recorded. Data is recorded at 50ms time resolution. Recording starts immediately when the command is received. See rslewrec.

Returns: "OK" if the command was accepted. "ERROR can't open tracker data logging file!" if it can't open a file to write the data to, or "ERROR: tracker must be HOLDING or TRACKING to record data." "BAD rotator name" if the rotator name was unrecognized.

`string` **`rRecIPolys`**`(string x)`

This starts recording the incoming polynomial stream which is (normally) coming from the PCS. The 'x' string has 2 arguments. The first is the name of the rotator for which the command is intended. The second is the number of seconds that data should be written to the file. Data recording begins as soon as the command is received. The data is recorded regardless of whether tracker is sending the incoming polynomials on to the DSP or not. If rotator is HOLDING, slewing, or stopped, the incoming polynomials are not being passed on to the servo. The data is written to a file called `rIPoly.txt`.

Returns: "OK" if the command is accepted. "BAD argument to iPolyRec. Too small or too large." if the time argument is < 0.01 or > 1000. "ERROR: can't open file!" if it cannot open a file to record the data. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rRecPolys`**`(string x)`

Starts recording the polynomials that are actually being sent from the rotator program to the rotator servo regardless of the source of the polynomials (whether they are coming from the PCS or being generated by the rotator software). The arguments are the rotator name followed by the number of seconds that data should be recorded. Recording begins immediately. Polynomials are written with a time stamp in ASCII form to a file called `rPoly.txt`. If the rotator has been stopped (with the rstop command) , no polynomials are being sent to the servo, so none will be recorded.

Returns: "OK" if the comman is accepted. "BAD argument to polyRec, Too small or too large." if the seconds argument is < 0.01 or > 1000. "ERROR: can't open file!" if it cannot open a file to record the data. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rSlewRec`**`(string x)`

The argument x contains the name of the rotator (from the table above) for which this command is intended. This arms the recording of the rotator positions for the next slew operation that is performed. Polynomials are written in ASCII form to a file called `rTrack.txt`. Recording starts when the next slew begins and ends 5 seconds after the slew finishes. Each line of text is timestamped relative to the start of the slew an has the commanded position (in arcseconds) and the actual position.

Returns: "ERROR" if Tracker is not in HOLDING or TRACKING state. "OK" if it accepted the command. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rSlewToHold`**`(string x)`

The argument x contains the name of the rotator (from the table above) for which this command is intended and the position angle (in degrees relative to the telescope structure) to which the rotator should move. The rotator state machine will change to slew-to-hold mode, do the slew and then change to hold mode regardless of what state it is in (slew, track, hold) when the command was received.

Returns: "OK" if the position legal and the rotator state machine is in a condition to be able to accept the command. "ERROR" otherwise. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rSlewToHoldGrav`**`(string x)`

The argument x contains the name of the rotator (from the table above) for which this command is intended and the position angle (in degrees relative to the direction of gravity) to which the rotator should move. The rotator state machine will change to slew-to-hold mode, do the slew and then change to hold mode regardless of what state it is in (slew, track, hold) when the command was received.

Returns: "OK" if the position legal and the rotator state machine is in a condition to be able to accept the command. "ERROR" otherwise. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rSlewToTrack`**`(string x)`

The argument x contains the name of the rotator (from the table above) for which this command is intended. If there is presently a stream of tracking polynomials for this rotator and they evaluate to a legal position, it will switch to SLEW_TO_TRACK state and intercept the target, switching to TRACKING mode as the slew terminates. If the current target position is illegal, this command returns "ERROR" and does not start a slew.

Returns: "OK" if the target position is legal. "ERROR" if the target position is bad or the rotator is not in a condition to be able to respond to the command. "BAD rotator name" if the rotator name was unrecognized.

`string` **`rStop`**`(string x)`

The argument x contains the name of the rotator (from the table above) for which this command is intended. This puts the rotator into the "stop" mode. In this mode it sends no polynomials to the servo regardless of whether any tracking polynomials are being received. see rVelMode() and rPosMode().

Returns: "OK" or "BAD rotator name" if the rotator name was unrecognized.

`string` **`rtrack`**`(string x)`

This is called (normally by the PCS) to supply the MCSPU with rotator polynomials for tracking purposes for all 8 rotators. Any of the rotators which are in TRACKING state, will pass their polynomial on to the servo which will do its best to make the rotator follow the polynomial. Polynomials for rotators which are not in a TRACKING state will be ignored. The polynomial is of the same form as those used for tracking on the main axis control as described in section 8 above. The argument 'x' is binary structure containing 8 tracking polynomials coerced to be a C++ string. Polynomials for rotators that have not requested PCS to supply a polynomial stream will be all zeroes.

Section 6 describes this polynomial data format, but also indicates that it may change if implementation problems are found. In that case this command will have to change to accommodate the format used by PCS.

Returns: an 8 character string where each character in the string relates to one of the 8 rotators.

An "O" means OK. The polynomial was used.

An "E" means "ERROR" which indicates that the rotator is stopped or if it failed to transition from HOLDING to SLEW_TO_TRACK or SLEW_TO_HOLD state. In this case the polynomial has been ignored.

An "H" means that rotator is currently HOLDING, so the polynomial was ignored.

A "B" means that the polynomial was ignored because it was bad. It probably contained out of range data.

`string` **`rVelCmd`**`(string x)`

When the rotator servo is in the velocity mode (as opposed to the position mode), this tells commands it to a certain velocity. The first argument in the string is the rotator name. The second is the velocity which is a signed floating point number. The units are in degrees/second. Changing from position mode to velocity mode will overwrite the last commanded velocity with a 0, so executing this command when the system is in position mode has no real effect. See rVelMode()

Returns: “OK” if the commanded velocity is within range. “ERROR: arg. out of range!” if the argument is out of range. “BAD rotator name” if the rotator name was unrecognized.

`string` **`rVelMode`**`(string x)`

Commands the rotator servo to change to velocity mode. The argument is the rotator name. If it is in position mode, it is advisable to command it to hold first and then use this command. When changing from position mode to velocity mode, the commanded velocity is set to 0.

Returns: “OK” if the servo changes to velocity mode within about 500ms. “ERROR” if the servo does not change mode. “BAD rotator name” if the rotator name was unrecognized.

`string` **`rWaitOpr`**`(string x)`

This commands the rotator sequencer state machine to cycle back to its lowest state, which is basically turned off. The argument is the rotator name. It turns off the power and the power amplifiers. It waits for the operator to command it to turn on.

Returns:”OK” if the command is queued. “ERROR” if the drive is in a disabled state. “BAD rotator name” if the rotator name was unrecognized.

`string` **`rZeroInteg`**`(string x)`

Enables or disables the integrator of the rotator loop. The argument is the rotator name and a 1 or a 0. A 1 enables the integrator. A 0 forces it to be held at 0. The latter makes the servo respond only to the instantaneous servo error because the error integrator term is 0.

Returns: “OK” or “BAD rotator name” if the rotator name was unrecognized.

10.2. Command Groups

The alphabetic list of commands above helps you find a command if you know its name, but gives no indication of what group of commands relate to a common area of control. Here they are listed (name only) in functional groups.

10.2.1. Sequencer group

These commands direct the behavior of the state machine which turns the whole rotator system on and off, enables or disables it, and monitors for various error conditions.

`string rIdle(string x)` Put sequencer in idle state.

`string rReady(string x)` Sequence Rotator into the “ready” state.

`string rWaitOpr(string x)` Sequence rotator to a turned off state.

10.2.2. Status group

The status commands return various status information to the caller, or cause it to be recorded to a disk file.

`string getRotatorReport(string x)` Get status information.

`string rRec(string x)` Record rotator positions to a file.

`string rRecPolys(string x)` Record outgoing (to servo) polynomials to a file.

`string rRecIPolys(string x)` Record incoming polynomials to a file.

`string rSlewRec(string x)` Record slew polynomials to a file.

10.2.3. Operating mode/motion control group

`string rForceWrap(string x)` Controls whether positive or negative wrap is used.

`string rHold(string x)` Hold position.

`string rLoadEnc(string x)` Force encoder to read a specified angle.

`string rMaxAcc(string x)` Specify max acceleration.

`string rMaxVel(string x)` Specify max velocity.

`string rOffset(string x)` Specify an offset to the encoder reading.

`string rPosMode(string x)` Put servo into position mode.

`string rSlewToHold(string x)` Slew to specified angle & stop.

`string rSlewToTrack(string x)` Slew to moving object and start tracking.

`string rStop(string x)` Stop sending polynomials to the servo.

`string rVelCmd(string x)` In velocity mode, command a specified velocity.

`string rZeroInteg(string x)` Enable or disable (and zero) the servo integrator.

`string rtrack(string x)` This command delivers tracking polynomials for all 8 rotators.

11. Normal Start-up, Shut Down Processes. Enabling and Disabling drives

The Rotator class contains the state machine that handles these processes. This will, among other things, execute the sequences described in section 11.3 of CAN 670s007 (Instrument Rotators and Cable Chain Detailed Design Description) (RD3). It will additionally relate these events to other sections of the software such as the Tracker state machine, system reflective memory and do logging and Event generation for all relevant events.

12. Engineering Interface Rotator Display Pages

The Engineering Interface will have several different display pages. All information available to the MCSPU about all the rotators can be displayed in one or more of these pages. Any rotator command (or any other command, for that matter) can be typed on the command line regardless of what display page is being shown.

12.1. Engineering Interface "**rot**" Display Page

The "rot" display is used to show status of one left rotator and one right rotator. The letters F, D, C, and R are used as arguments to "rot" to indicate which rotator's status should be displayed.

Argument letter	Implied Rotator
F	Front bent Gregorian
D	Direct Gregorian
C	Center bent Gregorian.
R	Rear bent Gregorian.

To configure the "rot" display page, one enters a command of the form

"`rot <left arg> <right arg>`"

or

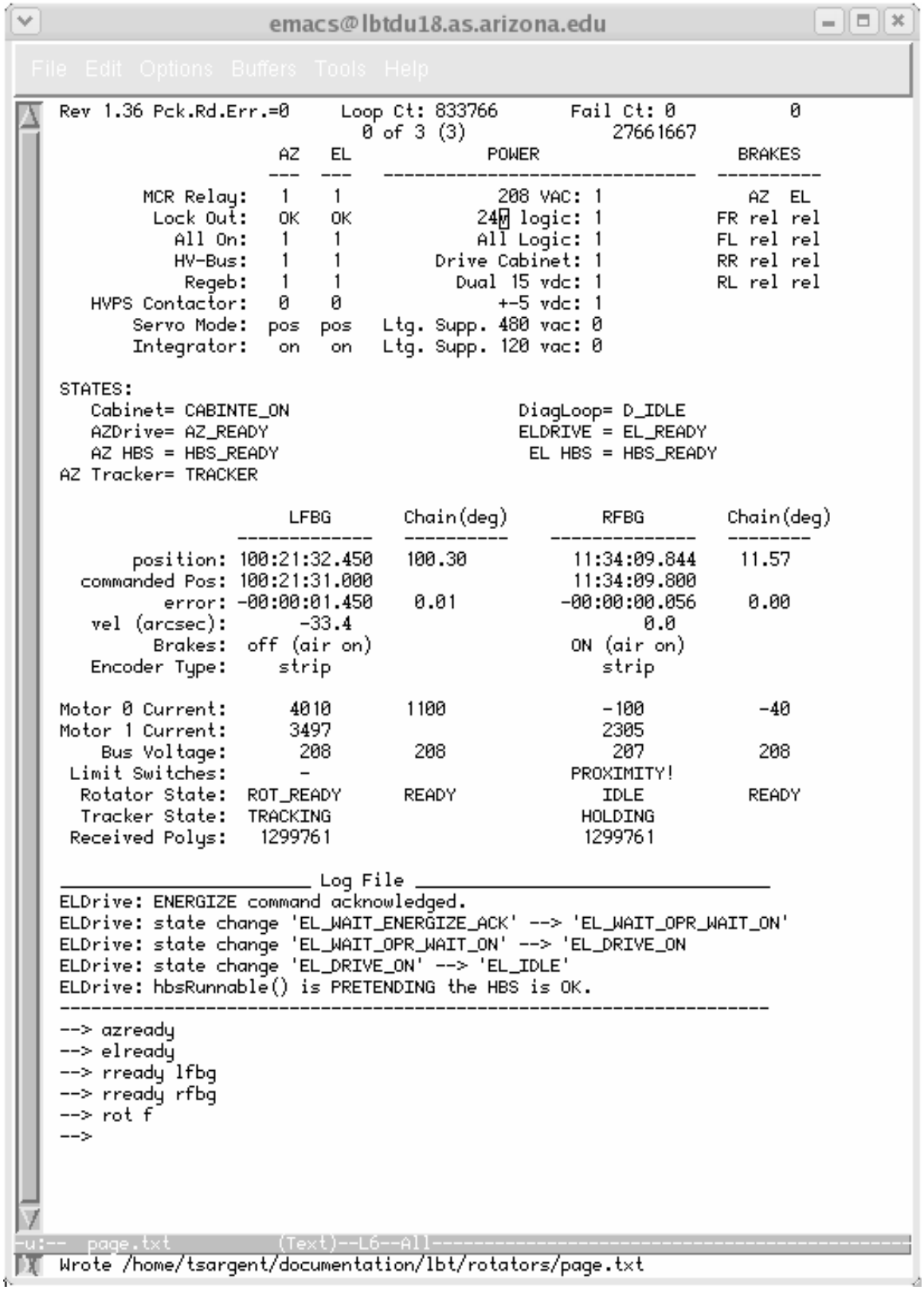
"`rot <arg>`"

In the first form, the first letter argument relates to the left side rotators and the second argument relates to the right side rotators. "`rot f d`" means to display the left front bent Gregorian and the right direct Gregorian rotator status. If only one letter

argument is supplied to “rot” that is taken to mean it should display status for that rotator on both left and right sides. So, “`rot r`” would display status for both left and right rear bent Gregorian rotators.

The example below is for the ‘rot page. It is similar to the existing “pos” (position) page for the AZ and EL drives.

This is just an example. It will change in detail. There will be as many other display pages as necessary to display all known information about the rotator systems: all encoders, all temperatures, currents, all status bits, etc.


emacs@lbtdu18.as.arizona.edu
File Edit Options Buffers Tools Help
Rev 1.36 Pck.Rd.Err.=0 Loop Ct: 833766 Fail Ct: 0 0
0 of 3 (3) 27661667
AZ EL POWER BRAKES
MCR Relay: 1 1 208 VAC: 1 AZ EL
Lock Out: OK OK 24V logic: 1 FR rel rel
All On: 1 1 All Logic: 1 FL rel rel
HV-Bus: 1 1 Drive Cabinet: 1 RR rel rel
Regeb: 1 1 Dual 15 vdc: 1 RL rel rel
HVPS Contactor: 0 0 +-5 vdc: 1
Servo Mode: pos pos Ltg. Supp. 480 vac: 0
Integrator: on on Ltg. Supp. 120 vac: 0
STATES:
Cabinet= CABINTE_ON DiagLoop= D_IDLE
AZDrive= AZ_READY ELDRIVE = EL_READY
AZ HBS = HBS_READY EL HBS = HBS_READY
AZ Tracker= TRACKER
LFBG Chain(deg) RFBG Chain(deg)
position: 100:21:32.450 100.30 11:34:09.844 11.57
commanded Pos: 100:21:31.000 11:34:09.800
error: -00:00:01.450 0.01 -00:00:00.056 0.00
vel (arcsec): -33.4 0.0
Brakes: off (air on) ON (air on)
Encoder Type: strip strip
Motor 0 Current: 4010 1100 -100 -40
Motor 1 Current: 3497 2305
Bus Voltage: 208 208 207 208
Limit Switches: - PROXIMITY!
Rotator State: ROT_READY READY IDLE READY
Tracker State: TRACKING HOLDING
Received Polys: 1299761 1299761
Log File
ELDrive: ENERGIZE command acknowledged.
ELDrive: state change 'EL_WAIT_ENERGIZE_ACK' --> 'EL_WAIT_OPR_WAIT_ON'
ELDrive: state change 'EL_WAIT_OPR_WAIT_ON' --> 'EL_DRIVE_ON
ELDrive: state change 'EL_DRIVE_ON' --> 'EL_IDLE'
ELDrive: hbsRunnable() is PRETENDING the HBS is OK.
--> azready
--> elready
--> rready lfbg
--> rready rfbg
--> rot f
-->
-u:-- page.txt (Text)--L6--All
Wrote /home/tsargent/documentation/lbt/rotators/page.txt

12.2. Other Engineering Interface Commands.

Several other Engineering Interface commands will be required, but it is too early in the system design to be able to specify what those will be.

--oOo--

Doc_info_start
Title: Preliminary *Rotator Control Software Specification*
Document Type: *Specification*
Source: Steward Observatory
Issued by: *Tom Sargent*
Date_of_Issue: *07-aug-06*
Revised by: Tom Sargent
Date_of_Revision: 02-jan-07
Checked by: Norm Cushing
Date_of_Check:
Accepted by:
Date_of_Acceptance:
Released by:
Date_of_Release:
File Type: MS Word
Local Name: MCSPU-rotatorControl-5.doc
Category: *678*
Sub-Category: *001*
Assembly: *Telescope Control Software*
Sub-Assembly:
Part Name:
CAN Designation: 678s001
Revision: *E*
Doc_info_end