Magdalena Ridge Observatory Interferometer: The control system of the unit telescopes

Chris Mayer*[a], Marion Fisher[a], Alan Greer[a], James Parkhurst[a], Philip Taylor[a]
[a]Observatory Sciences Ltd, William James House, Cowley Road, Cambridge, CB4 0WX, UK

ABSTRACT

This paper describes the telescope control system for the Magdalena Ridge Observatory Interferometer. To achieve the rapid development time required by the project we made use of two software packages, LabVIEW from National Instruments and TCSpk from Tpoint Software. The telescope control system is built from a set of components that conform to a standard interface and implement a set of component specific commands. Data is distributed throughout the system in a uniform manner by an event system that uses the publish-subscribe paradigm.

Keywords: MROI, Telescope Control System, LabVIEW

1. INTRODUCTION

The Magdalena Ridge Observatory Interferometer (MROI) is a sensitive optical, infra-red instrument designed for model-independent imaging. The array will ultimately consist of 10 unit telescopes arranged in an equilateral "Y" configuration although operations will commence with a complement of 6 telescopes. The telescopes are relocatable to 28 discrete locations to give baselines between 7.5m and 350m. Each unit telescope sends a parallel beam to the Beam Combining Facility (BCF) at the center of the array where the individual telescope beams are combined for fringe tracking or science. A schematic of the layout of the array is shown in Figure 1 which shows the center of the array in its close packed configuration

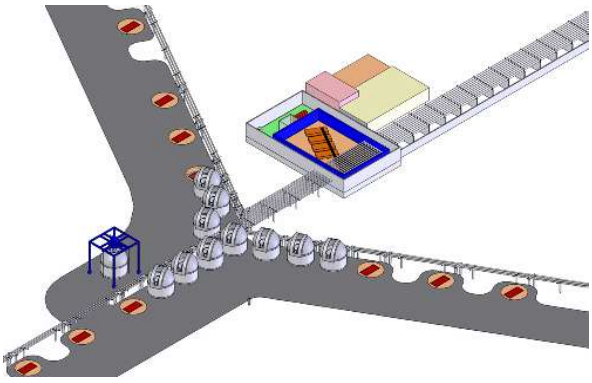

Figure 1 Layout of MROI in its close packed configuration

The beam combining, control and delay line area can be seen in the upper right of the diagram along with the Y arm configuration and some of the unit telescope pads.

Each telescope of the array operates as a Mersenne beam compressor. An input collimated beam of diameter 1400mm is delivered as a collimated exit beam of diameter 95mm. The mount has an elevation-over-elevation geometry together with a flat tertiary to minimize the number of reflections to achieve a horizontal output beam along the fixed "outer" elevation axis. This axis is nominally aligned at 104.47 degrees East of North. An independent Unit Telescope Control System (UTCS) controls each telescope under the co-ordination of the Interferometer Control System (ICS). The design

* cjm@observatorysciences.co.uk; phone 44 (0)1223 508257; fax 44 (0)1223-508258

and construction of the unit telescopes has been contracted to AMOS S.A. (Advanced Mechanical and Optical Systems). Further details about the interferometer, its goals and status and the opto-mechanical design of the unit telescopes are described in other papers at this conference. It is the UTCS that is the subject of this paper.

1.1 UTCS context and deployment

The figure below shows the context within which the UTCS must operate.

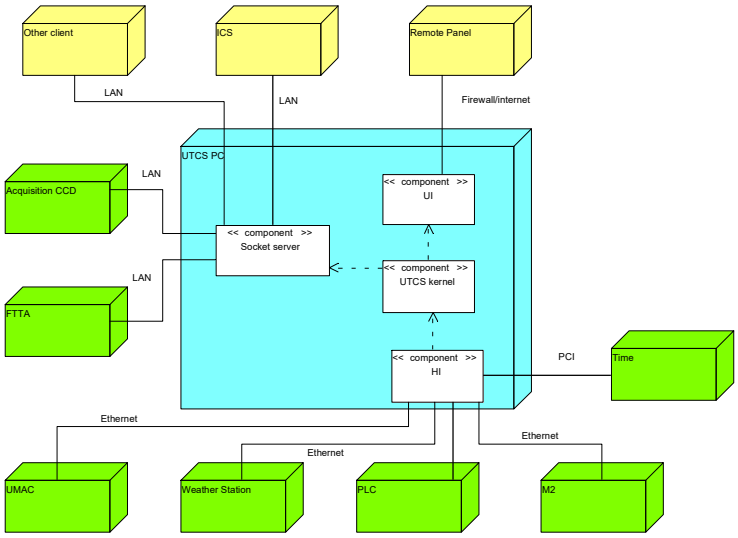

Figure 2 UTCS Deployment

The large central block represents the PC on which the UTCS executes. At the level of decomposition shown here the UTCS software is divided into 4 parts

- The socket server – this provides access to the public interface to the UTCS.
- The User Interface (UI) - the screens that the operator or observer would use to interactively monitor and control the UTCS.
- The UTCS kernel – the core of the UTCS. It implements command checking and validation, system monitoring and the pointing kernel that converts astrometric inputs to hardware demands.
- The Hardware Interface (HI) - isolates the kernel from the details of the hardware operation. Ideally if the physical hardware were changed this would be the only software that would need re-writing.

Most of these components will be described in greater detail in later sections.

The other blocks in the figure represent the external systems that the UTCS must control or interface to. Briefly these are

- ICS – The Interferometry Control System will control and sequence the observatory of which the individual UTCS's are just a part.
- Acquisition CCD(s) – These will use the same interface as the ICS to offset and adjust the telescope when acquiring a target.
- Fast Tip/Tilt Actuators (FTTA) – a dedicated channel for offloading the low frequency components of the corrections to the fast tip/tilt mirror to the mount.

- UMAC – the Delta Tau UMAC will be used for motion control. The UMAC is a flexible multi-axis controller that provides servoing, curve fitting and interpolation as well as the implementation of custom motion control programs.
- Weather Station – provides input data for the UTCS refraction calculations.
- PLC – sensor readings, switching and interlocks are provided by a Programmable Logic Controller (PLC).
- M2 Hexapod – the M2 Hexapod for positioning the M2 mirror and maintaining alignment with M1.
- Time – accurate time signals will be provided with a Symmetricom bc637PCI-U GPS synchronized time reference card. Each unit telescope will have its own card.

2. IMPLEMENTATION

The UTCS is being implemented using LabVIEW from National Instruments (NI) running on a standard Red Hat Enterprise Linux (RHEL) installation. Currently development is being done using RHEL 4 and LabVIEW 8.5. The decision to adopt LabVIEW was influenced by the work done by both SOAR [1] and SALT [2] together with the short timescales with which an essentially fully functional UTCS was needed for testing other parts of the overall control system. An alpha release of the system was made in May 2008 following a design start in September 2007. A more comprehensive beta release is scheduled for November this year.

LabVIEW provides an integrated development environment based on a data flow paradigm i.e. code executes when data is available at all of a component's nodes. A LabVIEW application is built from a collection of Virtual Instruments (VIs) each of which implements some specific functionality. A VI can in some ways be thought of as a subroutine call where the subroutine may be made re-entrant if desired. This is an important point as LabVIEW is inherently a multi-threaded language. Separate loops within a LabVIEW VI will run in parallel unless there is a data flow between them. Indeed, unless there is a data flow the programmer has no control over which loop will start first.

Some of the major reasons for adopting LabVIEW were:

- The rapid prototyping and productivity gains that have been reported with LabVIEW on other projects.
- The integrated development environment that provides a GUI as an integral part of the development of a VI.
- Simple integration of application libraries.
- Built in debug features such as probes, custom probes, highlighted data flows, break points etc.
- Ongoing active development and support by a large company together with a significant user community.

LabVIEW's main market is on Windows based machines but support for Linux has increased considerably since pioneering work by SOAR. Our decision to adopt Linux was influenced mainly by the much better remote access provided by that OS without having to purchase additional tools, compilers etc. along with our much better familiarity with it. The core LabVIEW facilities themselves are fully portable between Linux and Windows platforms and so VIs developed on Linux can be checked out onto Windows and run without modification and vice versa. It is in the area of LabVIEW add-ons that NI often only provides support under Windows so by using only using the subset of facilities available in Linux we ensure portability to Windows if required. Any application libraries called by LabVIEW must of course be compiled for the appropriate platform.

Our experience using LabVIEW for this project has so far been positive. Early tests on the timing stability of the pointing kernel which we require to run every 50ms showed variations of only a few milliseconds on a 2 GHz Pentium class machine. However, this stability was lost as soon as significant user activity took place and in particular moving and re-organizing windows on the screen. This behaviour is common to both Windows and Linux platforms and is not confined to LabVIEW applications but is symptomatic of using a non real time OS. As our real time requirements for the UTCS are fairly relaxed (the hard real time requirements for closing servo loops will happen in the UMAC processors) rather than switch to a real time platform we will use a multi core machine for the final implementation. Tests with a dual core machine showed that timing stability within a millisecond or so for the 20 Hz fast pointing loop was restored in spite of running background tasks to write large volumes of data to disk whilst simultaneously rapidly moving windows

on the screen. Although it is possible to assign VIs to run on particular cores on a multi-core machine we have not yet found this necessary and it would not overcome the non real time nature of the OS we are using.

LabVIEW is often portrayed (and marketed) as an easy tool for non programmers to write applications. We would not dispute this for a simple application but as soon as the application becomes more than a few tens of VIs it is essential that proper software engineering principles are applied to the development. See for example [3]. Although LabVIEW itself is not fully object oriented (it does provide classes) we have used UML to describe the design.

One area we had not foreseen when starting this project was the need to design the user interface separately from the front panels of the LabVIEW VIs. All LabVIEW VIs consist of two parts: the GUI (or front panel as it is known in LabVIEW) and the block diagram which is equivalent to the code in a non dataflow language. An example is shown below

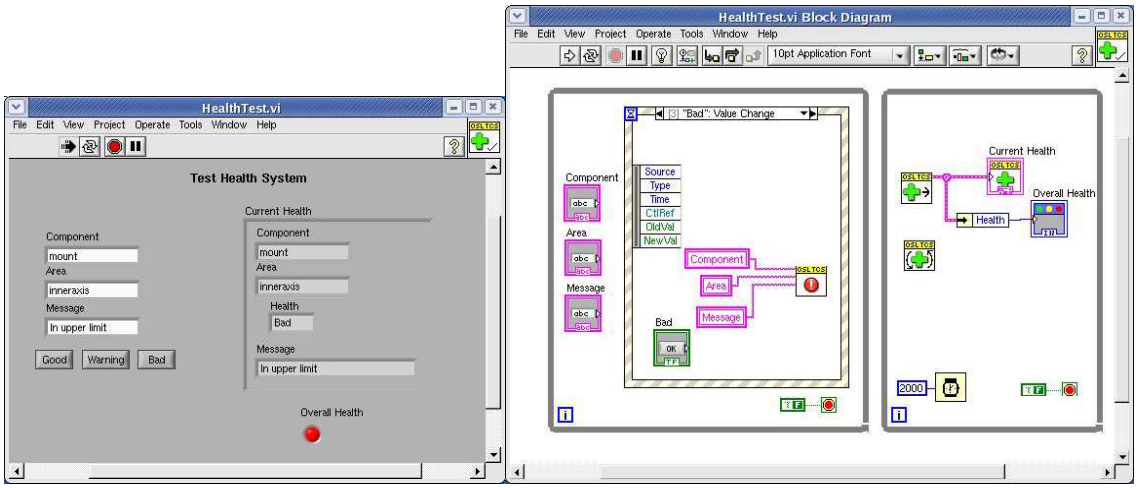

Figure 3 VI Front panel and block diagram

This VI was developed to test the health service used by UTCS components. The front panel on the left can be used to specify component names and health conditions and display the results. The block diagram on the right handles the logic of accepting the health status and then combining it to form an overall health for the whole system. The block diagram and front panel are inseparable. A VI is normally coded by laying out the user interface components such as text boxes, indicators and buttons and then switching to the block diagram to wire these components together with application logic. In general this provides a free test harness for every VI developed. Whilst this is invaluable during development, the hierarchy of VIs that represents the application is not the hierarchy of front panels you would want to present to the user to control that application. Our approach to this problem has been to design specific front panels for the user interface that then runs as a separate application. These front panels have very little application logic but simply send commands and display status. Multiple copies of these applications can then be run remotely from the UTCS PC if required. This decouples the user displays from the UTCS and avoids unnecessarily loading the control PC.

3. COMMON SOFTWARE

The code for the UTCS is split into two parts – common software and application specific software. Both parts consist of VIs and C code. The Common Software, as its name implies, is intended to be re-usable across multiple telescopes. Most of the VIs in the Common Software are infrastructure to support the component model used to construct the UTCS application. Most of the C code is from the SLALIB, TPOINT and TCSpk packages which are used under license from Tpoint software. The following sections describe both the more important VI packages and the code

3.1 Common Software VIs

Attributes and attribute tables – in order to make the transfer of data within the UTCS uniform a package of VIs has been created that implements the concept of attributes and attribute tables. Attributes are name value pairs. The type definition of an attribute stores both the name and value as a string. It is therefore possible to turn any data type into an attribute as long as the data can be stringified and decoded back to the same type. A polymorphic VI is available that will accept any of the standard types (string, integer, double or boolean) and the data can be a scalar or 1 or 2-D array. An attribute table consists of a name, description and array of attributes. This allows arbitrary structures of data to be created.

Events – events follow the publish-subscribe paradigm and form the fundamental means by which data is passed around the application. An application writer has only to deal with two VIs, an event poster and an event receiver. The event poster takes the name of the channel to post on and an attribute table. The poster neither knows nor cares if there are any subscribers. An event receiver supplies the name of the channel and if required a timeout period and receives an attribute table each time the poster posts an event. As well as responding to the named channel, an event receiver is always responsive to a special exit event. This provides a convenient method for breaking out of a loop if the event receiver is waiting indefinitely. By default, event channels are local to the application but the developer can configure them to be network events. The names of all network events are then broadcast periodically along with the name, IP address and port on which those events are available. A remote application does not need to know whether an event is generated internally from another part of its application or it is generated externally, all it does is subscribe to the named channel. Internally when it subscribes a direct TCP connection will be made using the port and IP address supplied in the broadcast message. All data on the wire is formatted as XML.

Monitors – monitors are similar to events but are used in conjunction with the external socket interface. Further details are given later.

Health – the health service is intended to make the setting and monitoring of the whole system as simple as possible. The UTCS is decomposed into components (see below) and each component from the point of view of health can have an arbitrary number of named areas. Each area is associated with a health value of Good, Warning or Bad along with an associated message that explains the reason for that health value. The health service then combines the health of all the named areas such that if any area has a bad health the overall health of the component is bad. If any area is set to warning then the overall health is warning provided that none has health set to bad. If no areas are set to warning or bad then the overall health is good. As well as combining the health of the areas of a given component the health service also combines the health of the components in a similar way. If the health service finds that there is more than one reason for the current health state it provides an output that cycles through the health messages at that health level. This is convenient for displaying on the user interface. Any health transition is automatically logged to the log service.

Logging – a number of different logs are provided for diagnosing problems with the UTCS. All log entries are time stamped. The system log is always produced and records any errors in the system. For example as mentioned above any health transitions are logged automatically. This log is intended for the daytime support engineers as a record of any problems that occurred during the night. All logging to this file happens automatically and can't be turned off. For developers a debug facility is provided. Each UTCS component has a debug level and accepts a debug command to set that level. Internally debug information is written whenever the component's debug level is greater than or equal to the level set for a particular message. The higher the debug level is set the more information is logged to the debug file. The default debug level is 0 which means nothing gets written to the file. The third log file is associated with the external socket interface (see below). By default this will record all external connections, disconnections and errors but by increasing the log level all commands and status can all be recorded.

Components – components are at the heart of the UTCS design. A component is characterized by the set of commands it responds to and possesses an overall health and activity. The component VIs implement the common functionality that all components within the UTCS must have. The component model of the UTCS is described in the next section

Configuration – the configuration service centralizes the handling of configuration data for the application. The configuration manager reads configuration file at startup. The file is a standard "ini" file separated into key word value pairs and split into sections.

3.2 Pointing kernel

The pointing kernel of the UTCS is implemented using SLALIB and TCSpk. The principles behind this kernel are described in [4]. For the UTCS all access to the functionality of these libraries is through Call Library Function Nodes (CLFN). A CLFN is the means by which LabVIEW can call custom code from a shareable library and return data to the application. In only a few cases do we need to call TCSpk functions directly as it is not necessary to expose the inner workings of the kernel at the LabVIEW level. Instead a single shareable library of wrapper functions has been written each function of which is encapsulated as a separate VI. Further details of the pointing kernel are contained in the next section.

4. UTCS DESIGN

The UTCS has been decomposed into a set of components with a fairly flat hierarchy as shown in the figure below

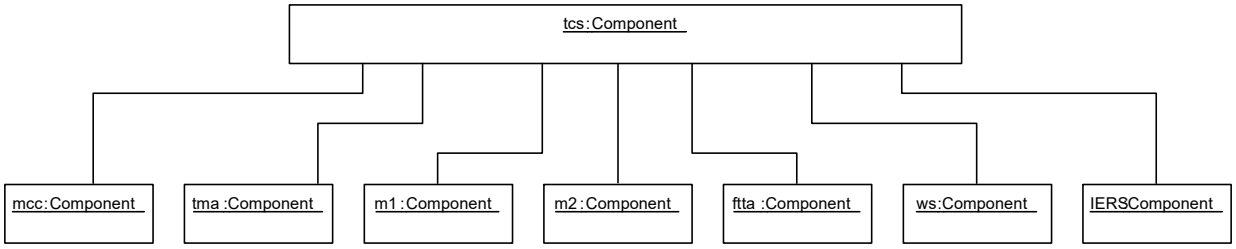

Figure 4 Component diagram for the UTCS

A typical component is designed to control some piece of physical hardware so, in the diagram above we have components to control the mount (mcc), tertiary (tma) and m1 and m2. A component however can be built to encapsulate any desired functionality so we also have components that control the off load from the fast tip/tilt mirror (ftta), read the weather station data (ws) and update the International Earth Rotation Service (IERS) parameters. In our present design we have kept the hierarchy flat. By this we mean that although a component can send commands to another component in practice it is only the tcs component that does this. All components however have access to the event system and can exchange data freely with each other.

Components have a simple state structure illustrated in Figure 5

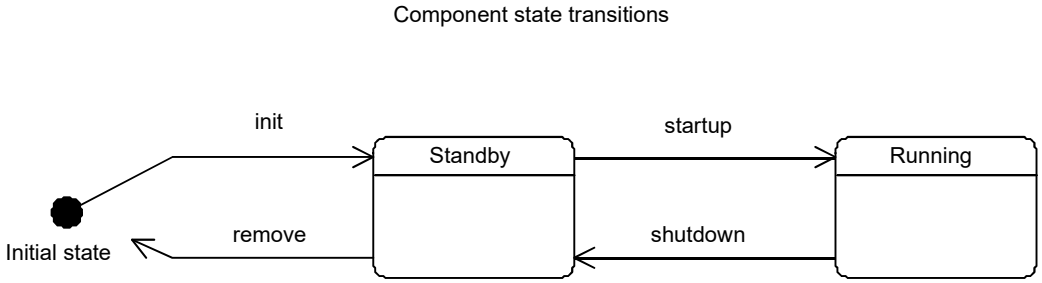

Figure 5 Component state transition diagram

On loading an implicit init takes the component to the Standby state. In standby a component would be monitoring any connected hardware but will not respond to any commands. To make the component responsive to commands it is issued a startup which takes it to the running state. This is the state the component must be in to be used operationally. A shutdown command will take the component back to standby and a remove is executed when the component is removed from memory. Operationally it is expected that the UTCS will run continuously but that during the day it will be left in the standby state.

One distinguishing feature of a component is that it responds to commands. The command set that a component responds to consists of an array of command structures where a command structure is made up of the name of the command, the

parameters it requires and a timeout. The command structure has provision for maximum and minimum values for the parameters plus default values to be used when no value is supplied. The component infrastructure will automatically validate command parameters against these limits if they are available. More complex parameter checking can be handled by providing the command structure with the name of a custom VI to run once the standard checking is done. This custom VI must meet a certain interface in terms of its inputs and outputs but otherwise the developer is free to perform whatever validation is required. A good example where this is used is the target command where it is necessary to check if the target is currently accessible. The final entry in the command structure is the VI that is to be executed to perform the action required by the command.

An important feature of the component infrastructure is that when the component is loaded it automatically registers its command array with the socket server. This immediately makes its commands available externally as well as internally

The UTCS components follow a command action model. Once a command is received, validated and acknowledged the command is considered complete. The component is then free to receive and process another command. Meanwhile the action started by the command is processed in a separate thread of control. If the action completes successfully then a completion signal is sent back to the originator of the command. Alternatively the action may end in error or it may time out. In either case the completion message sent back will now be an error along with a suitable error message. Whilst the action is running it is in the busy state and depending on how it completes it will end up in either the idle or error state. The busy/idle/error states of all the command actions are combined into an overall busy/idle/error activity state for the component. Note that each action has its own activity state. If a command is sent to a component that ends in error and then a different command is sent that completes successfully the command sender of the second command will see a successful completion even though the overall activity state of the whole component is still error. There are two ways to clear the activity state that is in error: firstly send the command again once the cause of the error is fixed or secondly send an explicit clear to the component that will clear all the activity states down to idle.

4.1 Pointing kernel

The pointing kernel of the UTCS has been incorporated into the tcs component. It runs continuously and is configured by data read from the configuration manager. It is initiated by reading the current axis positions and then using an upstream transformation to derive an RA and Dec. that can then be used in a downstream transformation to generate the demand axis positions. The effect of this is that at startup the initial mount demands are the same as the current position

The kernel consists of three main loops labeled slow, medium and fast that are scheduled once every 60s, 5s and 0.05s respectively. These loops are implemented as three separate VIs and they communicate through a data context

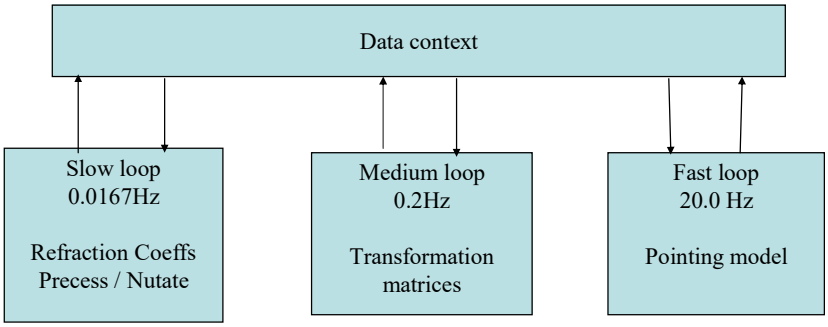

Figure 6 Schematic of kernel implementation

This data context is maintained within the shareable library that holds the pointing code. External access to this data context is provided through a number of VIs that act as getters and setters of the data. The kernel code is configured to control a gimbal mount in which the two axes are nominally perpendicular. The outputs of the kernel are published as events at 20 Hz which the mount and tertiary components subscribe to. Commands to these components control whether they try and follow these position events or not.

An important advantage of using TCSpk to implement the pointing kernel of the UTCS is that it is fully integrated with TPOINT. Any pointing model terms from TPOINT which are used to fit a pointing data set can be incorporated rigorously into the kernel without requiring changes to the kernel C code.

5. UTCS EXTERNAL INTERFACE

LabVIEW is a proprietary infrastructure protected by numerous patents. Although it provides many internal communication methods it does not expose the means to invoke VIs etc. from non LabVIEW applications. To provide a generic interface such that essentially any external application can control the UTCS we have implemented what we call a "socket server". This server implements a concurrent TCP server that accepts connections on a specified port. Multiple clients can therefore connect simultaneously. All communication over this connection is via ASCII strings so that the only requirements we put on an external client is that it can open a TCP socket and write ASCII strings. The OS or language of the client is immaterial. To make full use of the interface a client should be multi-threaded, but this is not required. Another advantage of this interface is that for test purposes you can simply open a telnet connection and type commands from the keyboard.

The number of commands accepted by the server itself is small although there is no limit to the number of commands that can be accepted by the application. The protocol is shown in the table below

Command	Response
open <host> <port>[1]	Connect : <state>
do <command> <attr>=<value> [<attr>=<value>]...	ack <command> <errcode> <message> done <command> <errcode> <message>
get <status-item>	got <timestamp> <status-item> <value>
monitor <status-item> [interval = <ms>]	mon <timestamp> <status-item> <value>
monitorOff <status-item>	ack monitoroff <errcode> <message>
alias <status-item> <item1> <item2>…	ack alias <errcode> <message>
unalias <status-item>	ack unalias <errcode> <message>
loglevel <level>	ack loglevel <errcode> <message>
disable <ack | done>	ack disable <errcode> <message>
enable <ack | done>	ack enable <errcode> <message>

Clients open a connection to the socket by whatever means their socket library provides. The response to this will hopefully be "Connect: Ok" but may be "Connect: Busy" if all available connections are used up. The number of available connections is configurable. An application that has finished with a connection should close the connection explicitly so that the UTCS can free resources at its end. Application commands are then sent to the UTCS by the server's "do" command along with any parameters. For example a new target could minimally be specified with

do target ra=12 0 3.4 dec=30 23 30.4

The response to a "do" command is an immediate acknowledgement to say whether the command is accepted or rejected. In the case above acceptance would return

ack target 0 Ok

whereas a rejection might be

ack target -1 Source does not rise for 2h 15m 17s

In the case of a rejection the error code is non-zero and the remainder of the response is a message indicating the reason for the rejection. If the command is accepted then some time later a completion message will be received. This

[1] Appropriate for a telnet client only

completion message will start “done target” and then like the acknowledgement message be followed by a status code and message.

Status can be retrieved in two ways. A get command will return the current value of the status item, whilst a monitor will return a value whenever the data value changes. If data is required at a fixed rate then the monitor can be specified with an interval in millisecs with a minimum interval of 50ms. A monitor request will always result in at least one response which will be the current value of the status item. This is done so that clients that request infrequently changing status items don’t have to wait for the item to change or issue separate get commands in order to find out the current status of the system. Clients can create their own compound status items by issuing an alias command. An alias provides a name by which to refer to many status items. A get or monitor can then be issued on the alias and an array of all the status items is returned in a single message. The only restriction with issuing monitors on status items that are aliases is that an interval must be provided.

Initial tests on throughput by monitoring small numbers of status items at artificially high update rates (the minimum interval was dropped to 1ms) gave some 5000 messages per sec. If two clients monitored the same small numbers of items then the throughput dropped to 4800 messages per second per client. Further tests are needed to establish the throughput in the more realistic situation of, say, a few tens of clients each monitoring a few hundred status items.

6. THE ENGINEERING USER INTERFACE

Control of the MROI as an integrated array is one of the roles of the Interferometry Control System (ICS). Never-the-less for acceptance testing and commissioning we will often need to rapidly switch between controlling one telescope and controlling another. In order to try and avoid hunting for the correct screen for the correct telescope we have adopted a top level display into which separate panes can be loaded in fixed locations. The current prototype can be seen in Figure 7.

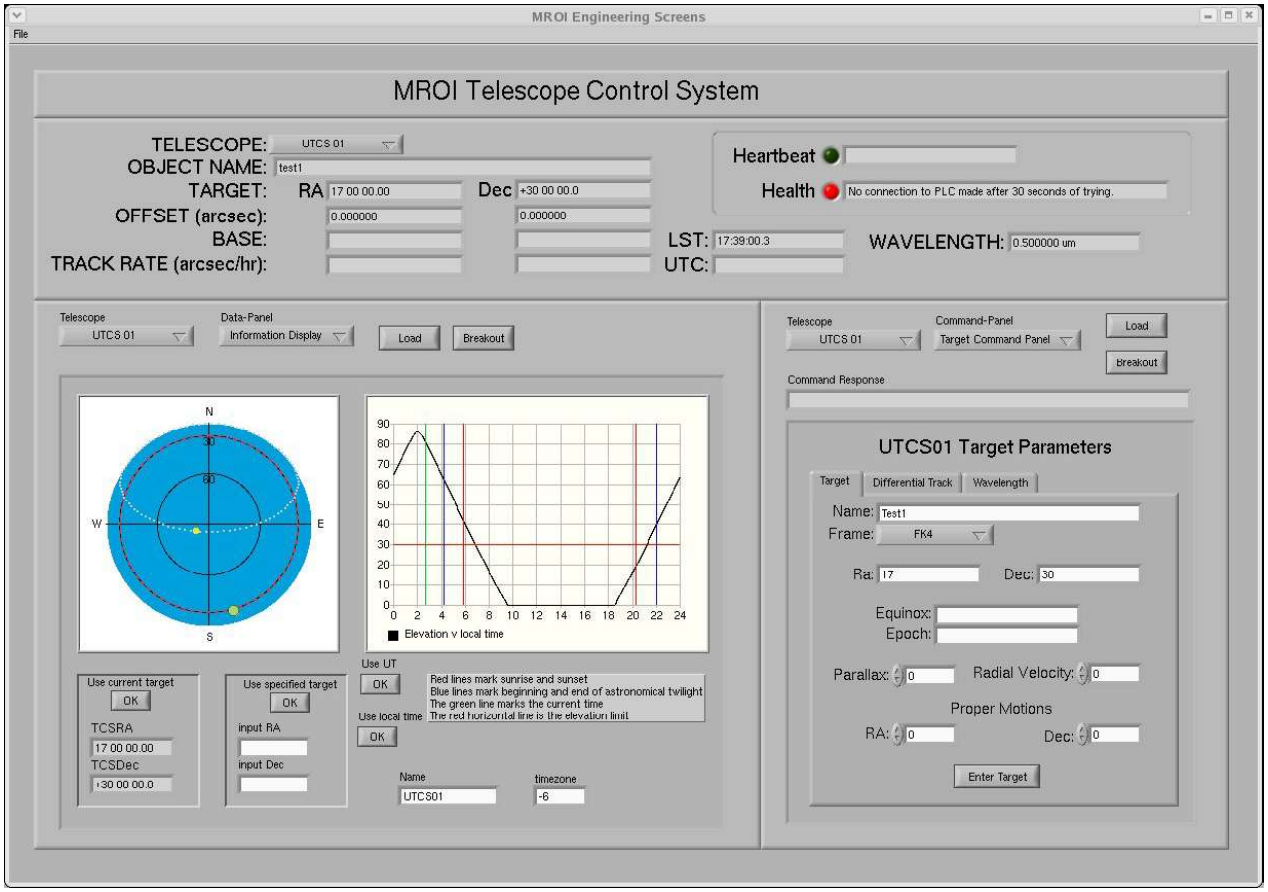

Figure 7 UTCS Engineering display

The top level display consists of three main areas. Across the top is a static area that is not configurable. This displays some key information that the operator will always want to see. The one configuration control that is available is the drop down menu next to the telescope label. This can be used to quickly change the display to pick up the data from any one of the 10 unit telescopes. Currently the screens are attached to UTCS01 and the overall health is bad due to a failure to connect to the PLC hardware.

The lower part of the display is divided into two main regions the larger of which is for “status” panels and the smaller for “command” panels. Although these are called the status and command areas there is nothing to stop status being displayed in a command panel and commands being available in the status panel. In general however we have kept the bulk of commands to the right hand side and the bulk of the status to the left hand side.

In the current screen shot we have loaded the target command and information display panels into the command and status areas respectively. The target command has been used to set an RA and Dec and this is reflected in the fixed part of the display. The target panel has tabbed panes to allow the input of other parameters such as differential track rates or effective wavelength. The information display panel shows graphically the current location of the target both as a track projected onto the plane and as elevation as a function of time. Important limits such as the lower elevation observing limit, sunrise and sunset and the location of the Moon are also marked. The drop-down menus provide a list of alternative panels that can be loaded. Notice also that there is a telescope button for each area. It is therefore possible to load these panels with the appropriate displays for any of the unit telescopes. Once a panel is loaded the breakout button or right mouse click can split the panel from the display and run it as a separate stand alone window. A new panel can then be loaded into the fixed display area. Using this technique we can quickly switch the display to any of the unit telescopes whilst keeping a check on the status of another.

The displays make extensive use of the networked events described earlier to keep updated. They can be started on any machine that has access to the LabVIEW run time and will automatically discover and then connect to the appropriate event streams.

7. TESTING

We are currently using two methods of testing the UTCS. Our first approach was to use LabVIEW itself to auto generate basic tests. Due to the way commands are automatically registered by a component it is possible for another LabVIEW application to open up the appropriate VIs and discover the names of all the commands, parameters and limits that that component responds to. Tests can then be auto generated to check that commands sent with those parameters in and out of range are appropriately accepted or rejected. A tool has been written to do this along with producing a report on the results.

Although the above provides basic tests it is not adequate to automate the test cases that need to be passed for final acceptance of the system. To meet this requirement we have created some Python classes that understand the socket server protocol along with a harness to run them and log the results. Within the run method of these classes it is possible to use any of the scripting facilities of python to produce arbitrarily complex tests.

8. SUMMARY

The MROI project has set tight deadlines for the production of a telescope control system for the unit telescopes of the interferometer. In order to meet those dead lines we have adopted LabVIEW from NI to give us the infrastructure to build our application along with Tpoint Software's TCSpk to implement the pointing kernel. The combination of these two systems has fulfilled their promise of enabling the rapid development of a robust control system that can be integrated into the overall ICS.

ACKNOWLEDGEMENTS

The ideas incorporated into this design have been developed over a number of years from work on a variety of telescopes but in particular we would like to acknowledge the work of Bret Goodrich and Steve Wampler on the ATST and Pat Wallace for the virtual telescope concept and pointing kernel.

REFERENCES

[1] Ashe, M., C. and Schumacher, G., "SOAR Telescope Control System: A Rapid Prototype and Development in LabVIEW", Proc. SPIE 4009, 48-60 (2000).

[2] Swart, G., Bester, D., Brink, J., Gumede C., and Schalekamp, H. , "Facing software complexity on large telescopes" Proc. SPIE 5496, 260-270 (2004).

[3] Conway, J., and Watts, S., "A Software Engineering Approach to LabVIEW", Prentice Hall, (2003).

[4] Wallace, P. ,T., "A rigorous algorithm for telescope pointing", Proc. SPIE 4848, 125-136 (2002).