The Design and Implementation of an Execution Domain for the Lowell Observatory Instrumentation System

by

Adam J. Gould

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 21, 1999

[June 1999]

Author____

Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by____

Dr. David J. Osip
Thesis Supervisor

Certified by____ [illegible]

Professor James L. Elliot
Thesis Advisor

Accepted by____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The Design and Implementation of an Execution Domain for the
Lowell Observatory Instrumentation System
by
Adam J. Gould

Submitted to the
Department of Electrical Engineering and Computer Science

May 25, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Lowell Observatory Instrumentation System (LOIS) is an astronomical image acquisition system designed to be used with different telescopes, detectors and instruments. The system is modular so that by replacing modules, it can function at many observatories. By creating scripts with the Tcl/Tk graphics language, observers can automate observing runs.

The first version of LOIS had some deficiencies: failure to update the graphical interface, lack of an error checking feature, and totally synchronous execution of commands. A new version of LOIS was proposed to address these shortcomings.

The main feature of the new version of LOIS is the execution domain. It allows command results to be obtained (for error checking) and affords much more control over command execution. Several designs were considered for the execution domain, and the chosen design included threads in each module to provide asynchronous command execution and result cells for error checking.

The new system offers users almost complete control over the timing of command execution. In the future, a remote access facility will be added to LOIS.

Thesis Supervisor: David J. Osip
Title: Research Scientist, Department of Earth, Atmospheric and Planetary Sciences

Thesis Advisor: James L. Elliot
Title: Director, George R. Wallace, Jr., Astrophysical Observatory
Professor, Department of Earth, Atmospheric and Planetary Sciences
Professor, Department of Physics

Table of Contents

Table of Figures

Chapter 1 Introduction

This project concerns itself with the design and construction of a new execution domain for the Lowell Observatory Instrumentation System, or LOIS. The main purpose of LOIS is to be a flexible astronomical data acquisition system that allows for different combinations of hardware and software domains. Examples of different instrument hardware domains are the MIT Auxiliary Nasmyth Instantly Accessible Camera (MANIAC - P.I.: J. Elliot, MIT) to be deployed on the first of the Magellan Telescopes and High-Speed Occultation Photometer and Imager (HOPI – P.I.: E. Dunham, Lowell Observatory) to be deployed on the Stratospheric Observatory for Infrared Astronomy (SOFIA).

The LOIS components described by this document include an execution domain in which commands will be evaluated and a suite of test hardware domains that can be used to simulate actual instrument and telescope communication. The execution domain will also offer more control over command evaluation than the typical paradigm in which only one command is running at a given time.

1.1 General Background

Due to the volume of data inherent in any astronomical system, as well as the need for computationally intensive analysis and simulation, the quality of the information system is of paramount importance. Its main functionality should include facilities to capture data, analyze and display data in real time, write data into the standard FITS (Flexible Image Transport System) format, as well as other routines more specific to the particular equipment at the observatory.

The hardware-specific commands are the most important part of any astronomical instrumentation system, because without these routines, the other routines (for analysis, storage, etc.) would be useless. LOIS must be capable of handling two different scenarios of hardware communication. The first variety requires LOIS to exchange commands with a controlling server. Many of the older telescopes and detectors have a controlling server that accepts strings from external sources and runs the commands that correspond to those strings. Some newer systems exhibit the second type of hardware communication, in which the image acquisition system communicates directly with a telescope or detector. In this case, the computer running LOIS will have a bus card that can send commands directly to the equipment hardware. LOIS must be able to efficiently send commands to servers or hardware and report if any errors occurred.

While not the primary responsibility of an instrumentation system, the data storage and display routines are nonetheless important. The reasons for a functional data storage system are obvious, however with many hardware domains simply reading data to a buffer and writing it to disk (or other medium) does not meet required data rates (i.e. the high-speed occultation modes for HOPI and MANIAC). Thus a useful storage system can be quite complex. Real-time data display is also a desired feature that serves as an effective initial analysis tool for observers. There are a number of image display packages that are freely distributed (e.g. Ximtool, or SAOtng which is being adopted with this version of LOIS), so the instrumentation system needs only to send data to the display application in a transparent fashion.

Adding remote command execution to the capabilities of a system adds yet another level of complexity in that the system must have some type of prioritizing in order to resolve possible conflicts between local and remote observers. Also, some data (such as graphical display data or captured image frames) cannot be echoed to the remote observers during the observing session.

1.2 General Background of LOIS

Additional specifications on top of those described above were considered mandatory for inclusion into LOIS. Such requirements include a graphical interface with a fast learning curve, allowing new and infrequent users to perform basic operations quickly, a minimum sustained data recording rate of 8 megabytes per second, an abort feature to cancel an exposure, telescope move or instrument operation, and a number of different exposure techniques for optimal execution of a variety of astronomical investigations. When capturing data, users may store the data to disk, to a tape, send the data to another LOIS process, or execute any combination of these options.

In addition to the features required of any instrumentation system, LOIS is also equipped with some convenient features that add to its general effectiveness and efficiency. These features are modularity, remote access, and scripting.

Modularity allows each of the telescope, camera, and other peripheral instruments to be viewed as separate modules. Thus, if a new instrument is added at an observatory, or if the system is installed at a new location, all that is required to obtain a functional system is minor modification of individual existing modules. In addition, routines for display, analysis and storage of data are each encoded in their own separate modules. There are several advantages to using modularity in the proposed design. First, addition of new equipment requires minimal change in the existing system. Although each telescope, camera, and instrument communicates in a different way, the modules for each type of hardware are relatively similar. For example, modules for the different cameras differ only in the commands sent from LOIS to the respective camera hardware. Thus, new modules can be created rather easily using previously existing modules as templates. Also, no changes need to be made in other modules, and the main program needs to be updated only slightly to account for the presence of the new module. Second, testing individual components of the system is made easier. Since the new module does not depend on other modules in the system, it can be tested in a stand-alone fashion. This is desirable because it means that the functioning of the new module depends only on the module itself and (to a small degree) on the main program. Thus, tracking down errors in a module is made easier, since one does not have to search through a multitude of components.

Due to the instant-access feature of MANIAC, as well as some other domains, a facility for remote command execution would be included. While sending commands to LOIS is relatively simple (one could use a simple socket connection over a standard network protocol), remote-access means that commands are not necessarily unique in their origin. As mentioned before, this could lead to conflicts if a local user and one or more remote users are entering commands simultaneously. Also, sources for commands must be traced. This is so that if a remote user enters a series of commands, and then enters further instructions intended to alter (or remove) those commands, they only affect that user's commands.

Scripting is a feature that makes observing runs much more convenient and efficient. Instead of having an observer sit at the local terminal (perhaps for many hours) entering one command at a time, scripting allows the observer to write a function (script) that contains some portion of the commands that the user would enter during the night. In this way, the system can automatically perform some of the routine operations that would otherwise require an observer to directly interact with the terminal. In addition, if an observer has some sequence of commands that is commonly executed, putting these commands in a script can save the observer the time of repeatedly entering the same commands. As part of this functionality, there must be some component of the system that allows for commands to be executed at a specified time or interval. Also, some scripts may desire that a command begin execution when another command ends. Thus, to provide a scripting domain, timing issues will be of paramount importance.

LOIS was designed to run on a machine running any (POSIX compliant) version of the UNIX operating system. UNIX provides a true multi-tasking environment, which is important to maintain efficiency while executing some commands. Also, most of the drivers for the hardware communication boards for many of the current instruments were written for Solaris (the Sun version of UNIX). Since some hardware domains must utilize these boards, LOIS must be compatible with Solaris. C was chosen as the language with which to develop the modules. The language provides a good low-level medium through which network communication, file storage and process management can be implemented. Most of the extra features provided by C++ were not directly applicable to the modules. This, in conjunction with the predicted learning curve for C++ made it an undesirable alternative. Tcl was chosen as the scripting language. It is freely distributed and can be easily installed on any system running a version of UNIX. Tcl also has commands to load shared libraries and it can easily be referenced by C code without loss of efficiency. Tcl is also an interpreted language, this means that users can create procedures and run them from the console without having to compile or load them first. Another advantage of Tcl is the graphical package Tk, which is distributed with the Tcl source code. Tk is a library that is easily integrated into a Tcl application and can be used to create graphical interfaces quickly and easily.

1.3 Previous Work – LOIS version 1.0

An initial version of the LOIS system (version 1.0) had been implemented at Lowell Observatory and received feedback from some users. Overall, the system did function well, allowing for the recording and analysis of significant amounts of data. Nevertheless, there were some notable drawbacks. These included 1) an inefficient drain of system memory, 2) a failure to update the graphical interface, 3) unnecessary synchronization of some commands, and the 4) lack of a mechanism through which third-party software could access the LOIS command interface. Figure 1 diagrams the path a command takes through the different parts of the system. The command is first entered through the console or graphical interface to the single interpreter. The interpreter then executes the command, and returns to service more commands.

The first problem arises because the system forks processes that take a long time to execute. Such processes include camera exposures, image display and data storage. Thus, every time one of these processes is executed, the entire LOIS process is copied (including its memory segment) and memory usage can increase very rapidly, which affects the speed at which LOIS runs and places an unnecessary burden on the workstation.

The second of these problems is also limited to instances where the system forks a process. During a camera exposure, the CCD module continuously updates the graphical interface with new information, such as the time remaining until an exposure is complete. Since the exposure process has been forked, commands generating this “countdown” information (and all other information sent from the exposure routine) take place in a child process. Thus, while an exposure is occurring, there are two sources of graphics commands that write to the same display, the parent LOIS process (which can accept commands from the interface or console) and the child LOIS process, which is running the exposure. Unfortunately, the Xlib graphics library cannot handle this type of asynchronous communication. Since Tk is the graphics language of choice, and it is built on top of the Xlib library, any graphics packages written using Tk will inherit this inability to handle asynchronous communication.

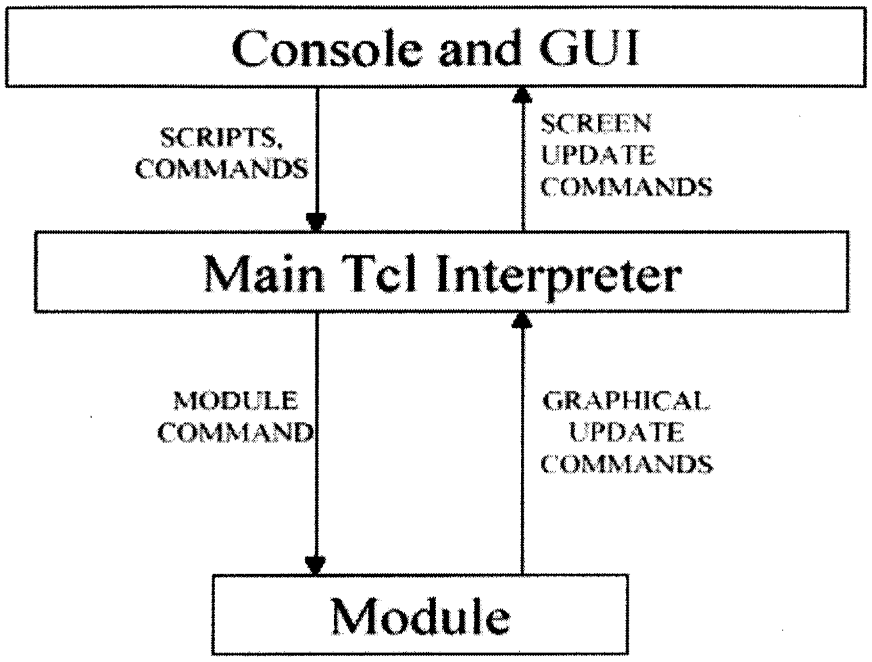


Figure 1 - Command Paths in LOIS version 1.0

The third issue is brought about because every command that is entered into the system (either through the graphical interface or through the console) is executed in the single Tcl/Tk interpreter. Thus, while the interpreter is running one command, it cannot service any other commands. This restricts commands that could be run in parallel to be executed in a serial fashion. While this feature may seem nonfatal, its effects are not limited to compromising the efficiency of the system. The problem is created because *no* commands (including graphical commands) are executed while the interpreter is servicing a command. Thus, if any command has graphics associated with it, these graphics will not be displayed to the screen until after the command finishes.

The fourth drawback is not a problem with LOIS ver. 1.0, but rather a feature that was left out of the implementation. The initial LOIS design calls for a feature to provide compatibility with third-party software, as it would increase the scope and utility of LOIS. However, one must ensure that third-party software only has limited access to the structure of LOIS, as it would be unwise to allow an external package access to module-level implementation of commands.

1.4 Necessary improvements for version 1.1

The first two drawbacks to ver. 1.0 cited above led to the proposal for a design of a new version of LOIS. The main features of this new version were the idea of making LOIS a multithreaded application and the introduction of a message queue to handle graphical commands. The advantage of a multithreaded application is that different routines can run in a single address space. Thus, when an application creates a new thread, the memory block is not copied. This property of threads all but eliminates the first problem. The third problem is also easily overcome by having several threads to run non-graphical commands and a single thread to expressly handle graphical commands. The efficiency of parallelism is realized by executing on different threads those commands that can be run simultaneously. The message queue was the primary instrument used to solve the graphical update problem. Instead of having calling modules run graphics commands internally, modules insert these into a message queue. The main part of the LOIS program (the part that is responsible for initialization) removes commands from the queue and executes these commands on a single interpreter. Thus, the graphics commands are all executed synchronously, and displayed to the screen.

In addition to these improvements, other features not implemented as part of version 1.0, such as delayed command execution and more control over command execution are necessary improvements to version 1.1 of the LOIS software. Delayed command execution is an essential part of the scripting feature, as observers would likely create scripts that should begin execution well into the night. If the observer wishes a script to start at 3:00 am, it should be possible to enter the script when the observing run starts (perhaps at sundown) rather than having to wait until 3:00 in the morning. As part of the delayed command execution feature, users should also have the option of executing scripts multiple times at regular intervals. Giving users more control over command execution refers to allowing users to receive results from commands and to cancel execution of commands. Scripts can perform error checking if they receive results from their constituent commands. Depending on the result returned, a script could either terminate itself, or skip over some of its commands. With a delayed execution feature in place, one must ensure that users can cancel or alter commands that they have entered into the system. Altering commands may be essential to preserve desired timing. Suppose that a sequence of commands is set to begin when an incorrectly entered command finishes. If one removed the incorrect command from the system and entered a correct version of it, it would also be necessary to remove and re-enter the sequence of commands set to follow the first command. These actions could perturb the planned timing, especially if the sequence of commands is to be repeated or executed in the near future.

As with any design, there are disadvantages that accompany the advantages. The primary issue brought about by the use of threads is the management of shared-memory objects. Since multiple threads are executing in the same address space, objects may be altered or accessed by multiple threads. The secondary issues are the standard problems inherent with multithreaded and shared-memory systems. These include the possibility of race conditions and deadlock arising from the need (by the threads) to access resources in shared memory. These potential hazards were averted by introducing locks (on shared memory objects) and wait variables (to prevent race conditions). There is no particular device to prevent deadlock, so care must be taken when utilizing the locks and wait variables. The next version of LOIS must provide new features (such as those outlined above) not present in the version 1.0 system.

The remainder of this document describes, in further depth, the steps taken to address and design facilities for the issues described in this chapter. Chapter 2 proposes a suite of test modules that can be used with the new core system. Motivations for the test modules, as well as problems encountered during their development are described in this chapter. Chapter 3 lists in detail the requirements that the new core system must satisfy, proposes several designs that achieve these requirements, compares and contrasts the designs, and finally concludes why the chosen design is superior. Chapter 4 recapitulates the development track for the new core system, starting with the initial system structure. The implementation questions that relate to the preferred design are discussed, and for each, the chosen path is described after being compared to other possible options. Improvements made on the initial core system are also described in chapter 4. Chapter 5 focuses on a particular problem that was not adequately solved by the new core system, and how this problem was solved. It also describes supplementary issues brought about by the solution. Chapter 6 suggests changes and additions to the current LOIS system that would further improve functionality and/or efficiency.

Chapter 2 Test Modules

Prior to confronting the design and implementation issues associated with creating a new core for the LOIS system, it was proposed that a test module be written for each type of module to simulate the actual communications carried out by that module. These simulations include commands sent from this module as well as time delays associated with particular hardware operations. The test modules were constructed before the new core system because understanding the issues presented by each type of module would be required to construct a functional core system.

2.1 Purpose of Test Modules

There are three main reasons behind construction of a series of test modules. First, the test modules simplify creation, testing and integration of new hardware modules. If a new telescope module needs to be created, it can be derived from the telescope test module since the structure of the test module and the new module will be identical (except for the specific hardware commands). In order to test a new telescope module with the system, one could run LOIS with the test camera and instrument modules with the new module. Under this configuration, the only hardware communication in the system is generated by the new telescope module, thus problems with the communication in this module can be more easily tracked down than if the system had been running with three real modules.

Second, core implementations could be evaluated with these test modules. Such a method is valuable because it abstracts all hardware communication away from an evaluation domain, allowing for more direct analysis of a core system. Also, as with construction of new modules, using the test modules to evaluate a core system simplifies debugging. If a problem occurs while testing a new core system, it could be isolated to the internals of the core or the way in which the core handles communication between the modules.

The third reason behind the test modules involves scripting. With the test modules, an observer could test scripts before actually using them during an observing session. The debugging aspects of this approach provide obvious advantages, as an observer could perfect scripts at noon under normal atmospheric conditions rather than try to correct them at 2:00 am, in a thin atmosphere with many other issues at hand. Since the test modules would simulate the actual equipment at an observatory, if a problem occurred during an observing run, it could be isolated to the hardware at the site.

2.2 Instrument Test Module

The instrument test module was designed to simulate a ten-position filter wheel. Since a filter wheel can only move to a position or reset and go to the original position, it was the simplest of the three test modules to implement. In a traditional instrument module, a string consisting of the command name and any relevant arguments is written to the instrument controller. The controller then parses the string and sends to the instrument hardware commands that move the filter wheel to the desired position. After the filter wheel has moved to the appropriate position, the controller returns status information to the instrument module. The test instrument module, instead of writing a string to the instrument controller, sleeps for an amount of time proportional to the distance between the desired and current position. The calibration routine is implemented as a special case of the move routine, where the desired position is the initial position. For testing purposes, the instrument test module writes to a file the name of any command it runs, with the associated arguments. For example, if a user wishes the filter wheel to move to position 7, the instrument module writes "FILTER 7" to the file.

2.3 Telescope Test Module

The telescope test module had many more functions than the instrument test module, therefore it is more complex than the instrument module and its implementation was confronted after the instrument module had been completed. Like the commands in the instrument module, most of the commands in the telescope module only involve writing data to the telescope controller. These commands were approached in a similar fashion as the commands in the instrument module. Instead of writing to a controller, commands were written to a file along with their arguments, and the function sleeps to simulate the hardware delay.

Some of the problems that arose during construction of the test modules were caused by limitations of version 1.0 described above. The graphical interface for any telescope module includes a field that displays the local sidereal time (LST). This value is continuously changing, so the graphics in the telescope interface must change at least once per second. Since some commands will require more than one second to run, in order to continue updating the LST field, the telescope test module must send graphics commands from a child process (or thread). However, the graphics sent from the child process will not be displayed to the screen as discussed in section 1.3. Therefore, in order for the telescope test module to correctly display all data, the message queue structure described in chapter 1 must be implemented.

2.4 Detector Test Module

The detector module is the most complex of the three modules, consisting of commands that fit into two categories, diagnostic routines and exposure routines. (There is also an abort routine that terminates an exposure, it will be treated separately.) At the time of its construction, only two other camera modules had been created, each of which used a controlling server to send commands to the hardware. Since the test camera module was based primarily on the two previously existing modules, it more closely simulates the communications inherent in a client-server domain than a domain in which LOIS directly controls the detector hardware.

The diagnostic routines include those used to initialize and close communications between the LOIS system and the camera/detector controller, a routine that resets the charge-coupled array in a camera, and a test routine that is particular to each camera. These commands were simulated in a fashion similar to the way in which the telescope and instrument routines were simulated; instead of writing to the camera controller, commands are written to a file. Simulating the abort routine is also rather straightforward. In the client-server paradigm used with older camera systems, camera routines open a second communication socket over which the abort command is sent, and then the abort routine terminates all module-internal commands associated with the exposure. Again, the same approach of replacing socket transmissions with file writes effectively simulates the communication caused when an exposure is aborted.

The exposure routines are those commands that control the different types of exposures. (standard, slow dots, fast dots, strips, focus, etc. – these modes are defined in Appendix A) Currently, three of these routines (single, slow dots and focus) have been simulated. As with the diagnostic camera routines, network transmissions occur during a camera exposure. These transmissions can be simulated as file reads and file writes. While most of the network operations are used to check the status of the exposure, at some point, the actual data contained in the exposed frame must be transferred from the detector to the camera module so that it can be stored and/or displayed. In order to simulate the actual readout of data, the test camera module reads data from a previously generated file and sleeps for a time that appropriately simulates the hardware delay associated with reading out the detector, in this case a charge-coupled array. The data from the file is then used in the storage and display routines.

The file transactions also add a degree of complexity to the camera module that is not present in the instrument or telescope modules. Of the three modules, only the camera module reads data from a network connection. There are many instances where, in the camera module, a routine writes data to a socket and then expects to read that same data from the socket. Since the test module uses files to simulate the network sockets, the test module must reset the position of the file descriptor after writing but before reading data from the file. One method that circumvents this problem replaces the file reads/writes with actual network communication with an echo server. (An echo server, when reading data from a socket, simply writes the data back to the socket. Hence, the node sending the data receives an exact copy of what it sent.) With an echo server, the camera module does not have to adjust any file descriptors.

2.5 Limitations and Lessons of Test Modules

Simulating some parts of the test modules proved difficult. One such instance arises in the telescope module when ephemeris tracking is set. Ephemeris tracking moves the telescope so that a particular object moving at non-sidereal rates remains centered in the field of view. When tracking solar system objects, the right ascension and declination (RA and Dec respectively, the primary celestial coordinate system) change according to the object's rate of motion across the sky. This depends completely on the object, not solely on the Earth's rotation, and hence is particular to each object. In order to simulate effectively a non-sidereal tracking rate, the test telescope module would have to store information about each such object. The camera module is also currently limited to using two-by-two binning, since the images used by the test camera module were originally obtained with a two-by-two binning factor. One possible way to circumvent this limitation is to re-bin the image array after it is read in from the file. This feature, however, was not a priority as it related to the camera test module, and hence has not yet been implemented.

The most instructive aspect of the test modules is quite possibly their structure. During development, it became apparent that commands in each test module are comprised of two parts. The first part parses command line arguments passed in with the function, while the second part takes care of the hardware communications. Since any module can be simulated to some degree by either the detector, telescope or instrument test modules, commands defined in any module would generally conform to the structure of commands in the test modules. This idea became central to the module thread design, one of the schemes for the execution domain. The module thread design and other designs for the execution domain are described in the next chapter.

Chapter 3 Design Possibilities for the Execution Domain

The main instrument of the ver. 1.1 system that addresses the shortcomings of the version 1.0 system is the LOIS execution domain. Its main purpose is to provide a common environment for the execution of module routines and script commands. Due to the introduction of queues and threads, difficulties in timing and data access arise that must be addressed by any execution domain. First these issues are discussed, and then several designs for the execution domain are considered, they are described and then compared to determine which design provides the best environment for execution.

3.1 Functional Requirements

Primary issues addressed within the execution domain include delayed command execution, thread control, tracking of commands and results, full functionality across all module combinations, and the ability to assign a priority to a command so that it executes before or after other commands. Beyond the direct functional requirements, additional improvements in efficiency and convenience that are addressed by the execution domain include differentiation of graphical from non-graphical commands and asynchronous execution of commands, which allows multiple commands to run simultaneously.

3.1.1 Thread Control and Delayed Command Execution

Version 1.1 of LOIS uses multiple threads instead of child processes as adopted under version 1.0. In a multithreaded domain, all threads access the same data and storage structures, whereas each child process is given its own copy of this information when it is created. Thus, some component(s) of the system must control the threads' access to shared memory objects. In addition, many commands (such as camera exposures) require synchronization between threads. Thread synchronization is more difficult than process synchronization because threads do not have a parent-child relationship structure. Another consequence of the lack of hierarchy among threads is that any one thread can terminate any other thread. These difficulties of synchronization and a lack of organized structure mean that an execution domain must have significant control over the threads in the system.

Thread control will also come into play with the scripting feature of version 1.1. Suppose that a user wants each of two scripts to begin at midnight. In order for these scripts to start execution at the same time, each script must be run in a separate thread. The execution domain will be responsible for keeping track of threads that have been created either to run scripts in the future or to run a script periodically. It is also accountable for returning appropriate messages to the user if a thread for script execution cannot be created and for deleting threads for scripts that were to be executed only once.

3.1.2 Compatibility with all Modules

The original design of LOIS called for the system to be fully functional given any combination of detector, telescope and instrument module. Since the execution domain will be running the commands contained in each of these modules, it too should be equally functional regardless of the current set of modules with which it is being run. This means that an observer who uses the system configured at one observatory should be able to achieve the same functionality with any combination of modules, provided that the underlying hardware of each system can support that functionality.

The idea of the test modules means that the interface seen by the execution domain will be relatively uniform across all sets of modules. Therefore, when a new module is added to the LOIS configuration, the execution domain should be able to accommodate this module with little change.

3.1.3 Command Priorities

With the introduction of a message queue to the version 1.1 system came the possibility of having some commands in the queue require long periods to complete execution. While these commands are running, other commands entered to the message queue cannot be serviced by the main interpreter. Hence, the queue grows larger and because commands are spending more time in the queue, calling modules are required to wait for longer periods for results. This situation is unacceptable because at times, some modules require immediate results for commands sent to the command queue. One solution to this problem requires calling modules to assign a priority to a command before passing it to the command queue. In this way, a command with high priority bypasses a large command queue, so the calling module receives the results in less time.

Priorities can also be used to control the order of execution of commands. Since high priority commands will execute sooner than low priority commands, modules can send a series of commands at differing priorities to achieve a particular order of execution. High priority commands can also be used to cancel some portion of a series of commands. If a module or script enters a long sequence of commands to the queue and then a user realizes that there is an error somewhere in the system, the user can send a high priority command to cancel or override the remainder of commands in the sequence.

3.1.4 Tracking commands and results

With the introduction of a scripting facility comes the possibility of many different sequences of commands being present in the system simultaneously. Obviously, a user who enters commands to be executed should have some degree of control over those commands. The user should be able to see if a command has completed or not, should be able to cancel the command from being executed, and should be able to obtain results generated from the command. Also, some modules may need to call external commands (commands defined in other modules). Since each module exists as a stand-alone object, external commands cannot be accessed by simple function calls; the module must pass the command to (and receive an associated result from) the core of the system. Therefore, when a command is sent to the core, the observer or module should receive a handle for that specific command. In that way, the desired command can be referenced without affecting other commands. Also, the user or module can access the command without having to worry about where in the underlying structure of the system the command is located.

Any execution domain must keep two queues in existence: one as storage for commands that have yet to be executed, and another to hold results of commands that have been completed. A user or calling module first passes its routine to the execution domain. The domain then returns a process identification tag (*pidtag*), this is the handle with which the user or module can access the command. After issuing the *pidtag*, the execution controller enqueues the command in the command queue. After some period, the command is executed, and the results are inserted into the result queue. Finally, the user or calling module uses its *pidtag* to acquire the correct results from the results queue.

While this may seem complicated, it applies only to commands called by the modules or by scripts that are executing in the background. A user interacting with the system through the graphical interface or console does not need to store process tags for each command. Commands entered interactively are interpreted immediately and return results to the screen. (Of course, a user interacting with the console can easily obtain command results with the standard Tcl methods used to set variables. (i.e. The command "set s [foo x y]" sets s to hold the value returned by the function foo when foo is called with arguments x and y.)

A requirement of the execution domain is that it returns unique *pidtag* values for each command. There are several implementation possibilities that ensure this condition, the simplest of which is to return a *pidtag* of 1 for the first command sent to the execution domain, 2 for the second command, and so on. The other choices are more specific to the underlying structure of the execution domain, and will be discussed later.

3.1.5 Differentiating graphical vs. non-graphical commands

Resolution of this particular issue is not required for a fully functional system, but failure to achieve this functionality would limit the efficiency of the new version of LOIS. It has been established that all graphics commands must be entered to the main command queue in order that they execute synchronously on a single thread. Naturally, the system could be configured so that *all* commands are executed on this main thread, including commands that do not contain a graphical component. While this configuration provides a functional system, the execution of graphical commands (which usually requires little time) is slowed down by the presence of non-graphical commands that may take a long time to execute. It also ignores the potential benefits of parallelism. If, instead, these non-graphical commands are executed on a different thread, the graphical interface will be updated much more rapidly. One obvious advantage to a rapidly updated graphical interface is that informational messages are displayed much closer to the time they are generated, instead of well after they are generated when their source is more difficult to pinpoint.

The main design question associated with this issue is whether the modules or the execution domain should label the commands. If the modules label the commands, then in every module, every time a command is sent to the execution domain, the module would be required to specify whether the command should be executed on the main thread or on an auxiliary thread. If this decision were left to the execution domain, then it would need to store an internal table that lists the thread on which each particular command should be run. Depending on the number of commands that the system recognizes this table could become quite large.

3.1.6 Synchronous vs. Asynchronous execution of commands

Under the first incarnation of LOIS, if a user were running any command (other than a camera readout), no other commands could be run until the first command returned. This formulation while functional in most cases is neither efficient nor desirable. As far as some observational programs are concerned, this situation bears little attention, since exposure times are far larger than the delay involved with moving the telescope or switching filters. However, for other kinds of studies, the extra time gained by having the filter wheel and telescope move simultaneously is of great importance.

Adding asynchronous execution of commands to a system raises a series of additional questions, most of which relate to exactly which commands can be executed simultaneously. For most systems, the mechanisms within the instrument and those controlling the telescope are unrelated, so they can be allowed to move at the same time. The same flexibility is rarely desired for the detector system, while an exposure is occurring, all other equipment should be locked in position. There are exceptions to these general rules, such as a focus exposure, when the telescope's focus must move while the detector is being exposed. These situations illustrate the need for either the modules or the execution domain to specify which commands can be executed simultaneously. If the specification is left to the modules, each module must have some way of turning off functionality in other modules. On the other hand, if the execution domain stipulates which commands can be executed simultaneously, it must have a listing of which commands can be executed simultaneously among all of the modules.

The previous six sections outline the requirements of any execution domain. The remainder of this chapter will deal with different designs that provide these functionalities. The three-tiered execution kernel was the first design considered for the execution domain.

3.2 Three-tiered execution kernel

The three-tiered execution kernel represents the execution domain with three structures: a scheduler, a notifier, and an execution manager. In general, the scheduler handles the entering and scheduling of commands, the notifier is responsible for storing and returning results from commands, and the execution manager actually executes the commands. The command queue serves as a communication medium between the scheduler and the execution manager, while information is passed between the notifier and execution manager via the results queue. The communication paths present in the execution kernel design are diagrammed in figure 2.

Since the execution manager communicates with both the scheduler and the notifier, and it is in charge of executing the commands, it is the most complex of the three components of the kernel. Of the six issues listed above in section 3.1, the execution manager addresses thread control, command differentiation, module compatibility, and synchronous vs. asynchronous execution. The primary duty of the execution manager is to remove commands from the command queue and execute those commands on the appropriate threads. The execution manager is also accountable for inserting the results of the commands in the results queue and managing the use of shared memory objects. Also, depending on which design is chosen to assign *pidtag* values, the execution manager may also be responsible for assigning process tags to calling modules.

The simplest design for the execution manager calls for it to run on one thread. At startup, the execution manager starts a Tcl/Tk interpreter as a main thread. This

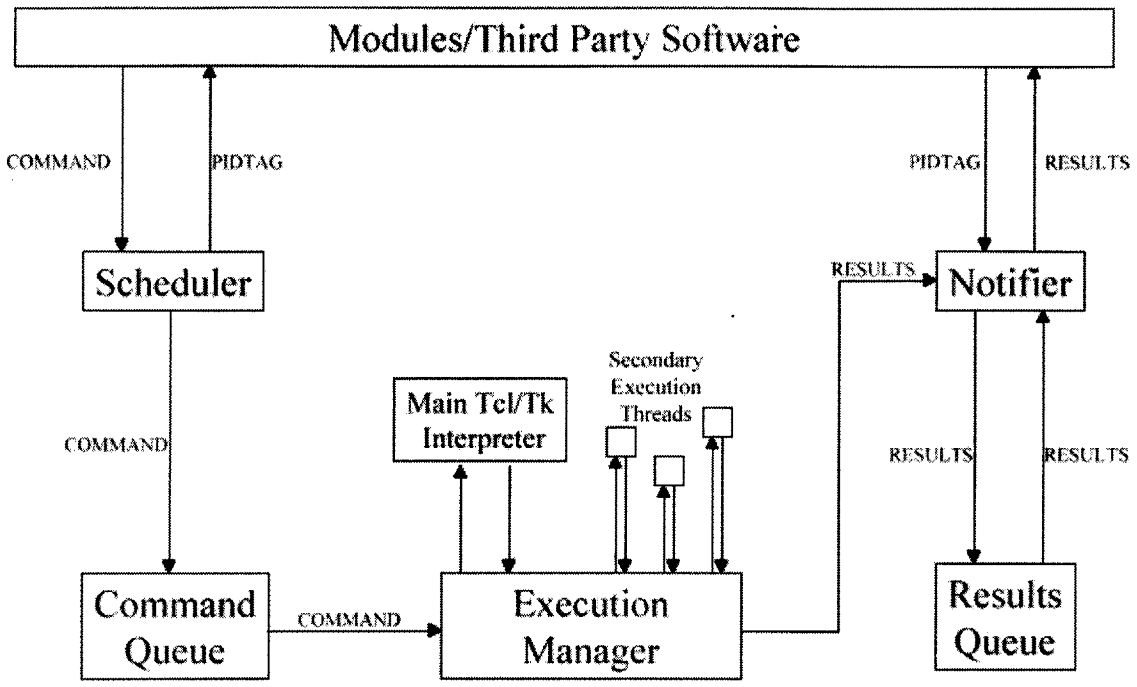


Figure 2 - Command Paths of Three-Tiered Execution Kernel Design

thread remains in existence until LOIS is terminated. In this way, all commands run on this interpreter thread are synchronous in nature. Of course, non-graphical commands should not run on this main thread by the stipulations of command differentiation. In addressing this problem, there are two possible mechanisms for the execution of non-graphical commands. In one design, the execution manager spawns several new threads at startup, when the main Tcl/Tk interpreter thread is spawned. Like the main thread, these non-graphical threads are also in existence until LOIS is terminated. In the second design, the execution manager spawns a unique thread for each non-graphical command. The thread exists while the command is executing, and it is destroyed when the command completes execution. The second design allows for the execution manager to generate process tags for each command. Since each thread has a unique identification number, the execution manager can assign as a *pidtag* value to any non-graphical command the number of the thread on which the command is executing. Of course, if this method is used, some feature that assigns unique *pidtag* values to commands that contain graphical parts must exist. Such a feature would have to be aware of all previously and currently existing thread identification numbers, as well as numbers that might be assigned to threads in the future. The only way to ensure that a *pidtag* does not match a possible future thread identification is to issue to graphical commands *pidtag* values that do not fall in the range of allowed thread identifications. Since this method is rather complicated, and there are simpler ways to return unique *pidtag* values to calling modules, the idea of using thread identification numbers as *pidtag* values was not pursued. In either design, the execution manager must limit parallel computation to those sets of commands that can be executed simultaneously. Since the execution manager is running the commands, it needs to know which sequences of commands must run in a serial fashion, and which sets can be run asynchronously.

When assigning command to threads, the execution manager must be able to differentiate between commands that contain graphical components and commands that require no graphics operations. One way to accomplish this goal is to have the execution manager keep a list of commands in memory, and every time a command is to be executed, the manager can look up the command to determine if it has a graphical component. Another method requires the calling modules to determine if the commands they are sending to the execution kernel contain graphical components, and to add this information (as a flag) to any command sent to the kernel. The execution manager need only look at the value of the flag to assign the command to the proper thread. Under the first design, the execution manager has to look up each command in its table before it is executed. In the second design, a module can be configured so that it sets the flag appropriately before sending the command to the queue. Since setting and referencing the value of a flag is faster than a table lookup, the second approach is more feasible.

The execution manager must also remove commands from the command queue, and insert results into the result queue. Inserting the results should be straightforward, the execution manager can simply pass the results (along with the *pidtag*) to the notifier, and the notifier can insert the results into the results queue. Removing commands from the command queue should also be straightforward, but this feature partly depends on the implementation of the command queues. If there is one command queue, then the execution manager can simply remove the first command on the queue and execute it. However, if there is one queue for each priority level, the execution manager must search through the queues by priority order until it finds a non-empty queue.

Since LOIS will be a multithreaded process, some concurrently executing blocks of code may require access to common blocks of memory. Therefore, modules may need to lock shared memory objects before they can run. Any set of locks must be consistent among the modules for proper shared memory allocation. Since the system can be run with a wide variety of modules, it makes sense to have the execution manager store locks on shared memory objects. In this way, if a command is running that requires access to a shared memory block, the execution manager can determine if the block is free by searching its list of locks.

In the execution kernel design, the scheduler is the only object that can change the command queue, therefore the actual representation of the command queue depends on the requirements of the scheduler. The scheduler needs to enter commands from calling modules and scripts into the queue based on priority and time of execution. It also needs to store commands that are to be executed sometime in the future, as well as commands that are to be executed multiple times. The scheduler also may need to return process identification tags *(pidtags)* to the calling modules and scripts. The scheduler addresses the problem of delayed command execution and provides part of a facility for tracking commands.

In order to maintain the correct order of execution for commands of differing priorities, and to limit access of the command queue by the execution manager, the scheduler must ensure that the command with the highest priority remains at the front of the queue. Thus, the scheduler may have to reorder commands in the queue before they are executed. Another option is to have multiple queues, one for each priority level. The execution manager would then query each queue, beginning with the highest priority queue, until it found a non-empty queue. However, this option would introduce more overhead than would be experienced by having the scheduler reorder the commands in a single queue.

The scheduler also needs to maintain a list of commands that are to be executed in the future or at a regular interval to address the issue of delayed command execution. As previously stated, when the scheduler receives a script that is to be executed in the future, it creates a thread that will sleep until the specified start time. Management of these threads will be a primary responsibility of the scheduler.

Another responsibility that may fall under the jurisdiction of the scheduler is the assignment of unique *pidtag* values to each command that is submitted to the kernel for execution. This process will be part of any feature that tracks commands in the execution domain. A solution is to assign the thread identification number of the calling module as the *pidtag*. Since each thread in execution has a unique identification number, and a single thread can only execute one command at any time, this paradigm would ensure that each command is assigned a unique *pidtag*. (Unless there are remote LOIS processes running. In this case, concatenation of the IP address of LOIS with the thread identification number would yield a unique *pidtag*. See section 6.4 for more about remote access.) This method simplifies the execution kernel because the scheduler does not have to compute *pidtag* values.

The third component of the execution kernel design is the notifier. The notifier is responsible for organizing results obtained from the execution manager as well as returning those results to the proper calling modules. As such, it also plays a part in providing a facility to track commands and results.

Organizing the results in the queue is the simpler of these responsibilities, as the notifier can simply insert any command structure into the queue that it receives from the execution manager. If the queue is full, the notifier can send a signal back to the execution manager, instructing the execution manager to wait until space is freed in the queue.

Signaling the calling module that its results are ready to be returned is a complicated design issue. One solution would have the calling modules periodically query the notifier until their desired results are returned. However, there are trade-offs involved when determining the interval between queries. If the interval is too long (10 milliseconds), then the results queue may be full most of the time. If the querying interval is too short (1-10 microseconds), then the notifier will become overwhelmed by the volume of queries, and will not be able to process the results queue efficiently. Another technique for alerting a calling module that its results are ready is to have the calling module block until it receives a signal from the notifier. Then the notifier would signal the appropriate calling module each time it enqueues a result into the queue. This method would eliminate the overhead of periodic querying, but it would require that the calling module supply its thread identification number with the command structure when the command is sent to the execution kernel. Also, if the notifier is to alert the calling modules, then the modules will block while waiting to be alerted. An upper limit must be set on the length of time for which the calling module will block. If, after this length of time the module has not been alerted, it would query the status of the command to determine if the command required a long time to complete, or if the result was somehow lost in the kernel.

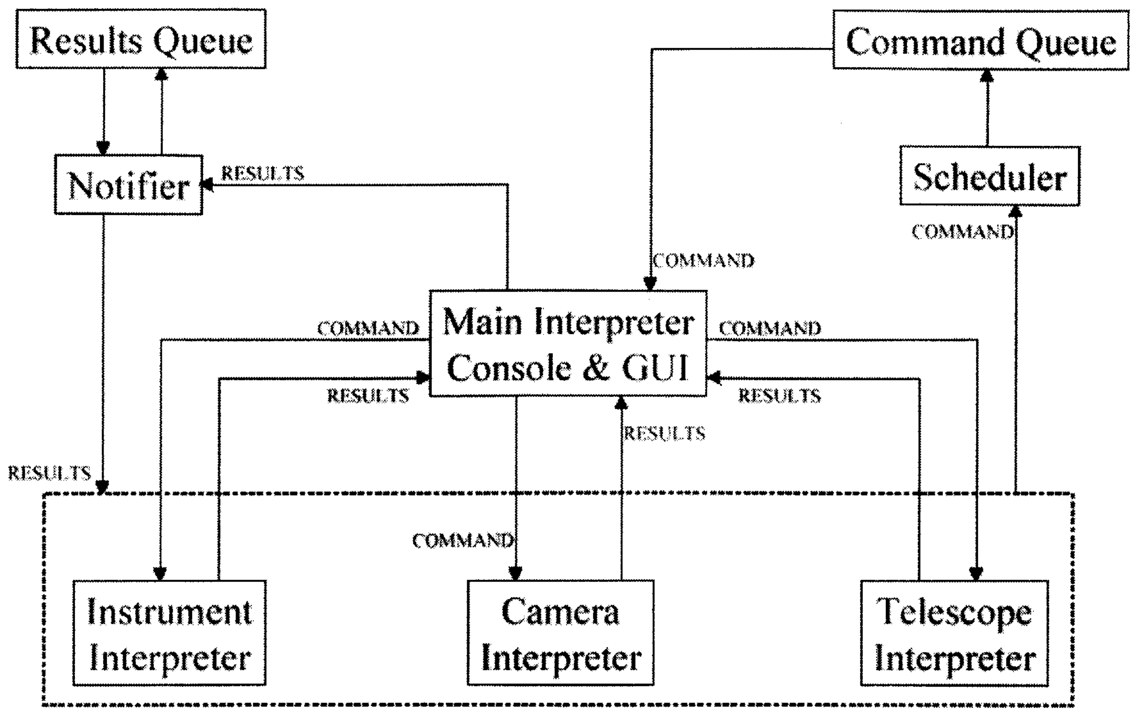
Regardless of which approach is used, the notifier must be certain that the calling module did in fact receive the appropriate data. Only when the data were received may the notifier remove the results from the queue. The best solution to this problem calls for the module to send an "OK" message back to the notifier when the appropriate results have been returned. Otherwise, the module can send an error message, and the notifier can attempt to send the results again. This solution is much like a synchronous network connection, and as such, an appropriate time interval must be determined whereby the notifier will re-send the results if it has not received a signal from the calling module before the interval expires.

While the specification for this execution domain solution seems long and complex, other designs borrow parts of this design, especially for their solutions to command tracking and delayed command execution. The other designs differ mainly in how they execute commands.

3.3 Multiple Interpreter Design

The main premise of the multiple interpreter design is that the main interpreter creates three slave interpreters, one for each of the CCD, telescope and instrument modules. In this design, commands defined in the CCD module are executed on the CCD interpreter (except graphical commands, which are sent to the main interpreter). A command queue still exists which allows the modules (which are running in slave interpreters) to send graphical routines that should be executed on the main interpreter, and the results queue allows the slave interpreters to receive results from any commands that they send. Figure 3 shows the overall structure of the multiple interpreter design. This design basically replaces the concept of using multiple threads with multiple interpreters. Thus, the creation of multiple threads is not necessary in this design, since the commands are being run on different interpreters. Graphical commands can be sent to the main interpreter, so the graphical interface is updated properly. Non-graphical commands can be executed simultaneously so long as they are called from different modules. If two commands reside within the same module (such as a telescope move and a focus move), one command must complete and return a result before the next command is executed. Hence, this design provides a degree of restriction in the form of synchronization of commands within a module. A seemingly obvious solution to this dilemma is to have the telescope interpreter (and the camera and instrument interpreters)

Figure 3 - High Level Command Paths of Multiple Interpreter Design



each create several slave interpreters, and execute some subset of their commands on each interpreter. This idea, however, has severe performance costs, as any application that contains ten simultaneously active Tcl interpreters (which would be required to provide a fully asynchronous domain) will experience significant slowdown.

This design solves the problem of thread control by replacing it with the task of controlling multiple interpreters. Since interpreters do form a hierarchical structure, and one interpreter can obtain a handle on its parent and children, interpreter control is more straightforward than thread control. Command tracking is addressed by having a scheduler and notifier similar to those defined in the execution kernel design, while delayed command execution is provided by the scheduler. This design does not provide a true mechanism for command differentiation, as all commands go through the main interpreter at one point during execution. However, non-graphical module commands, when evaluated in the main interpreter, are passed on to the appropriate slave interpreters for execution. Thus, the only non-graphical commands that are executed in the main interpreter are native Tcl operations (such as setting and querying the value of a variable), which (for the most part) run quickly.

3.4 Module Thread Design

The third design possibility for an execution domain also has a scheduler and notifier similar to the one described for the execution kernel design. The scheduler still creates threads to execute scripts in the future, and returns process tags to calling modules, while the notifier still manages results from commands. Therefore, delayed command execution and command tracking are addressed in this design in much the same way as they are addressed in the execution kernel. The main difference between this design and the execution kernel is that the module thread design places the responsibility for command execution in the hands of the modules. The high level communication paths for this design are illustrated in figure 4. Note how the structure of this system is much less centralized than the execution kernel (figure 2).

At startup, the main program creates a graphics thread. Then, when each (CCD, telescope, instrument) module is initialized, it creates a set of threads on which to execute its commands. All graphical commands are sent to the graphics thread (created by the main program). Also, since each module has its own set of threads (and the telescope and CCD modules will have multiple threads) simultaneous command execution (across and within modules) is possible. A module in this domain consists of two main structures, a Tcl wrapper that parses arguments and writes them to a command structure, and a thread part that reads the arguments from the command structure, and executes the hardware specific part of the command. A pipe exists through which the wrapper functions can send commands to the threads, and each thread has an output block to which results of commands (or error messages) can be written.

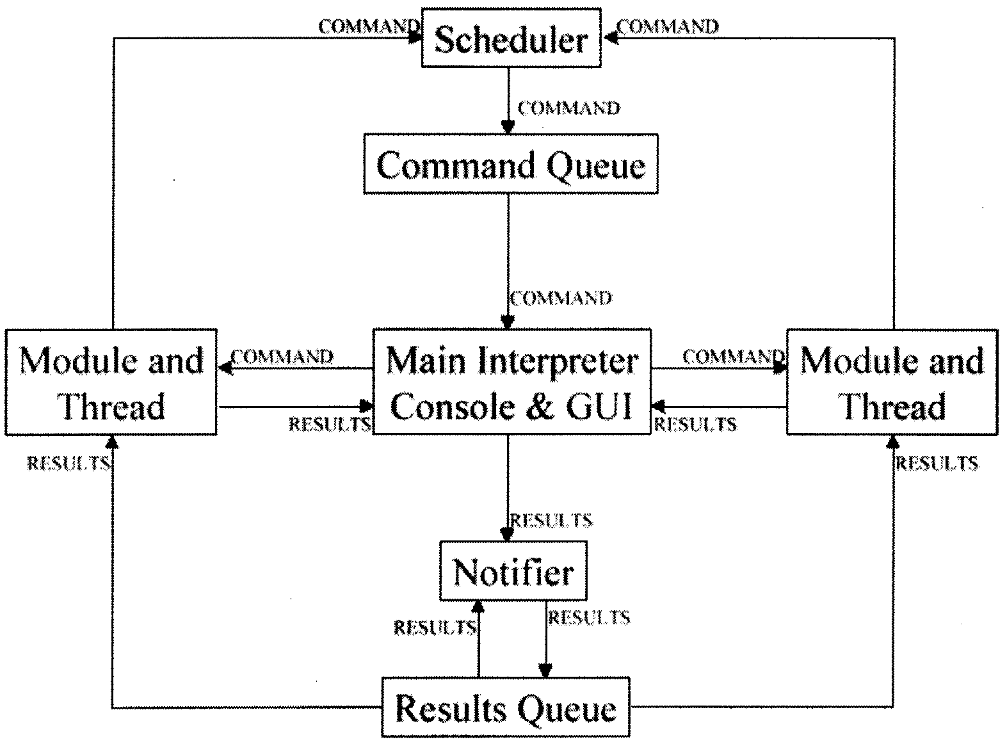


Figure 4 - High Level Command Paths of Module Thread Design

In this design, the scheduler uses the paradigm of assigning *pidtag* values starting from 1 to the calling modules. When a calling module wishes to obtain the results from its command, it sends the associated *pidtag* back to the notifier. If the result is ready, the notifier returns it immediately. If the result is not ready, then the module blocks within the notifier. At any one point, multiple modules and scripts may block within the notifier. When the notifier receives a result, it broadcasts this condition to all modules that are currently blocking within the notifier. One by one, the modules wake up and check if the result from their command is ready. If the results are ready, then the module returns from the notifier. Otherwise, the module puts itself back to sleep and signals another module to check the results queue. In this way, each time a result is inserted into the results queue, the module seeking that result will receive the result. Although modules can potentially wait for a long time before receiving results, each module only tests one condition before deciding whether to signal another module. Furthermore, since most commands sent to the command queue do not request results, the command queue is usually small. Therefore, signaling the correct module occurs rather quickly.

The issue of command differentiation is practically solved by the inner workings of the design. When commands are sent to the main command queue, graphical commands will be executed quickly. Commands that do not contain graphics components (such as those defined inside the modules) also are interpreted in the main thread. However, since the main interpreter only need execute the wrapper part of the function, and the time-consuming parts of non-graphical functions are now executed by the threads, the command completes quickly, freeing the main interpreter to service more graphics commands.

Since each module has its own threads, the modules are responsible for thread control. The modules create threads at initialization and do not destroy them (they are terminated when the user exits LOIS), thus the problem of one thread terminating another will not be encountered. Each module in this domain will also be equipped with routines that, when called, suspend some or all of the threads inherent to that module and reactivate the threads associated with a module. With these routines, an observer can explicitly force commands originating in different modules to be executed in a synchronous fashion. Nevertheless, the design does leave much leeway in the area of synchronous vs. asynchronous execution, especially between commands defined within the same module. While limiting the number of commands that execute synchronously is desirable, some commands must be executed synchronously, such as telescope moves and relative telescope moves. Therefore, the commands of each module must be grouped so that sets of commands like the telescope move and the relative telescope move are executed on the same thread. If the module thread design were used, this process would be required when creating any new module.

3.5 Comparison of Designs

The final part of this chapter will compare and contrast the three execution domain designs: the execution kernel, the multiple interpreter system, and the module thread design. Ultimately, the module thread design was chosen as the one that provided maximum efficiency, functionality, and flexibility. After this section, the reasons for choosing the module thread design should be apparent. The features provided by each design presented in this chapter are summarized in the table at the end of this section.

The functionality provided by the multiple interpreter design does not match the functionality of the other two designs. As explained above, the multiple interpreter design does not allow for multiple commands within the same module to be executed simultaneously. In actuality, Tcl version 8.0 has a flaw that causes all interpreters in a hierarchy to halt when the Tcl procedure "eval" is invoked within one interpreter in the hierarchy. If the "eval" procedure is used to interpret console commands, the system can only service one console command at any time. Seemingly, interpreting the console input without the use of the "eval" command can solve this problem. However, in order to obtain asynchronous command execution with the multiple interpreter system, each interpreter must exist in a separate thread. Under these circumstances, the multiple interpreter design is identical to the module thread design, except that each thread also has the computation overhead associated with a Tcl interpreter. For this reason, the multiple interpreter design will not be considered in further design comparisons.

The two designs left to evaluate are the execution kernel design and the module thread design. In fact, the two designs are equal in their capabilities; that is, any operations that the execution kernel can perform can also be carried out by the module thread design, and vice versa. In terms of speed, the module thread design is superior. This is because each thread is querying its own command pipe, whereas the execution manager is responsible for removing all commands from a single queue and dispensing the commands to the appropriate threads. The module thread design is also superior in the area of determining which commands can be executed synchronously. If the execution kernel design is used, then the execution manager must keep an internal table that contains information about which commands can be executed simultaneously. Whenever a new module is constructed, this table must be updated to account for the new module. On the other hand, if the module thread design is used, this information can be stored within the new module, and preexisting modules will not be affected. Therefore, if the module thread design is used to implement the execution domain, when an outside source builds a new hardware module, no additional knowledge is required regarding the internals of the execution domain. Therefore, integration of new modules is easier under the module thread design than the execution kernel design.

An issue with the module thread design is how a particular module will be able to interact with all types of other modules. For example, how should a new telescope module be constructed so that it can interact with all possible CCD modules? In the execution kernel design, the new telescope module would just send camera commands to the execution manager, which would actually call the camera routines. However, in the module thread design, the new telescope module must handle the calls to any particular camera module. This is where the test modules will be of assistance. Although the underlying hardware communications for different camera modules are different, the interface to any camera module is (for the most part) identical across the various hardware configurations. Thus, a telescope module could interface to all camera modules by interfacing to the test camera module.

An issue that is seemingly a problem with the execution manager is that it must know the appropriate number of threads to spawn on start up. At this point in development, the number of threads is known, given any module configuration. However, future modules may require more threads than currently existing ones, and there may be modules that must handle multiple CCD arrays or instruments that are running at the same time. This problem can be circumvented by having the execution manager start a large number of threads that will cover any combination of modules. The unused threads can be put in a "sleep" state so that they do not waste processor cycles. Nevertheless, it would seem that the number of threads to be spawned is a property inherent in the modules, and not the execution manager.

The main advantage of the module thread design over the execution kernel design is that in the module thread design, modules can be added or removed as separate components from the system, while the execution manager must be updated with the addition or subtraction of any module. Having to update the core system with the construction of any module can be a slow and error-prone process. Since the module thread design doesn't require changes to preexisting components to assimilate a new module, it is a more flexible design and for this reason was the design chosen for the execution domain.

Design Issue	Execution Kernel	Multiple Interpreter	Module Thread
prioritized commands	✓		✓
concurrent command execution within modules	✓		✓
concurrent command execution across modules	✓	✓	✓
modularity / ease of expandability		✓	✓
differentiation of graphical and non-graphical commands	✓	✓	(shown later)

Chapter 4 The Module Thread Design

The main conclusion from chapter 3 is that the module thread design is the best of the three alternatives to provide an execution domain. This chapter describes the structure of the initial implementation of the module thread domain, issues that arose during its construction, and improvements that were made after the initial development had been completed.

4.1 Initial design

The module thread domain was initially constructed as an experiment to explore its potential usefulness. Therefore, little in the way of efficiency was considered when the module thread design was originally built. The original design used UNIX pipes to send commands from the Tcl wrappers to the threads. Each thread within a module had a unique pipe. UNIX pipes function similarly to network sockets, except multiple applications can read or write from either end of the pipe, and all applications using a pipe must be resident on the same host. The pipes were a good tool for a first implementation because they are included in the operating system and any data entered or removed from a pipe is guaranteed to follow a first in-first out paradigm.

Any command entered through the console, the graphical interface, or the main command queue would go through the following steps in the first implementation of the module thread design. (These steps are shown in figure 5) First, the Tcl wrapper part of a function would parse the command arguments and load them into an input data structure.

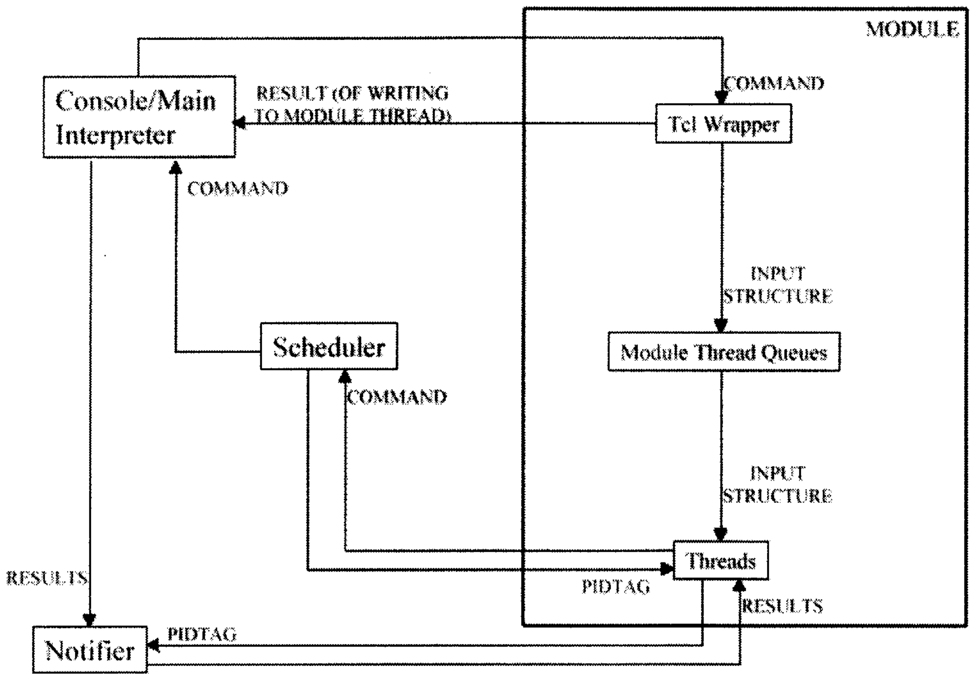


Figure 5 - Initial Internal Command Paths for Module Thread Design with One Module (also shows paths to scheduler, notifier and other modules)

The Tcl wrapper would also set a field in the input structure to be an integer corresponding to the function to be executed. (The functions and their integer representations are defined in a header file.) The Tcl wrapper would then lock a mutual exclusion (mutex), write the input structure to the appropriate pipe, unlock the mutex, and return to the main interpreter. In this design, the pipe to which each module function would send the input structure would be hardwired into that function. The module thread would then lock the pipe, remove the input structure, and unlock the pipe so more commands could be sent to it. The thread would then execute the function coded by the non-argument integer field described above, where the arguments to the function would be stored in the remaining fields of the input structure.

As part of this first implementation, there was no way to obtain results from commands executed on the threads. If an error occurred during command execution, an error message was written to the log. Therefore, modules and scripts could not perform error checking by waiting on the result of a command executed within a module. Also, modules had no way of enforcing synchronous execution of external commands. Under the initial implementation, while a camera exposure was occurring, an observer could continue to enter commands that would move the telescope or instrument. Obviously, some restrictions must be placed on the actions of the telescope and instrument modules while an exposure is occurring.

One difference between the module thread design and the version 1.0 software is how modules call external commands. In the old version, since all commands are executed synchronously as a Tcl procedure, all that was necessary to call an external command was to call the C function that implemented the command. In the module thread design, external commands would be called from the threads. Since the threads have no handle on an interpreter, they cannot simply call an external C function, since the external functions take an interpreter as an argument. The module threads use a different mechanism to call external commands, it is discussed in further detail in section 4.3.

4.2 Blocking and activating threads

Functions for blocking threads and reactivating blocked threads were added to each module to provide a system with which users and other modules could block (and then resume) execution of commands within a module. The block command is structured like the other commands in a module; it contains both a Tcl wrapper and a thread part. The arguments for the block function are treated differently than arguments for standard module functions. Instead of writing the arguments to an input structure, the arguments are used as the threads for which the user or module wishes to suspend execution (if no arguments are provided the function assumes that the user wishes to suspend all threads). For each thread specified, an input structure with the correct command designation is written to the pipe associated with that thread. The thread part of the function is also different from the thread part of most module routines. Since the block command can be used to suspend multiple threads, it does not run on any particular thread. When a thread sees an input structure whose command field is set as *BLOCK*, it immediately waits on a condition variable. This action suspends execution of the thread.

The "activate" routine is unique in that it does not contain a thread part, its only part is executed within a Tcl interpreter. As with the “block” function, the arguments to the “activate” function are interpreted as the threads that should be restarted, and an empty argument list is assumed to mean that all threads should be restarted. For each thread specified, the "activate" function signals the appropriate condition variable, thereby allowing the thread to resume execution.

A serendipitous, yet interesting effect of this implementation is how a block command sent to a blocked thread is interpreted. Since the block only applies to the thread, and not the pipe, any number of block commands can be sent to a pipe, and each will block a thread as it is removed from the pipe. This provides a significant degree of control for the user. Also, since camera exposures use this feature, scripts that contain multiple exposures (as well as other commands) can be written and interpreted quickly (thereby freeing the interpreter to service other commands entered by the user) while the order of execution among the commands is preserved inside the module thread pipes. While this feature is useful, it requires that users issue an activate command for every block command that is sent, otherwise a thread will be blocked and will not execute commands.

Considering that camera exposures use the block and activate routines, an exposure function must have some way of knowing that all telescope and instrument module pipes have been blocked before it can begin the exposure. All exposure functions run on the same thread. So a possible solution would involve making this thread wait on semaphores that are posted when the telescope and instrument threads become blocked.

4.3 Sending commands between modules

In the original version of LOIS, all module commands were implemented as C object command procedures. When a module was loaded into the main interpreter, the module commands became available as Tcl commands. Thus, whenever a module command was executed, the interpreter from which the command was called was passed as an argument. This argument could be used to call other C object command procedures from within a module function. Thus, modules in version 1.0 of LOIS could call external commands simply by calling the C object command procedures that defined these commands.

With the introduction of module threads, however, calling external commands becomes more complicated. Since most of a module command is now executed on the module threads, a module command cannot call an external function simply by referencing its C object command procedure, since the module threads know nothing of interpreters. Thus, a new method of accessing external commands is required.

Two possibilities exist that would provide this functionality. In the first solution, the threads simply send external commands to the main command queue. The commands would then be executed on the main interpreter (which understands all module functions). A thread could also obtain results from the command by accessing the result queue. In the second solution, module threads would write commands directly to the pipes of other threads. This method, however, requires that the thread have access to information about other modules. Such information includes the thread in the external module to which the command should be written and the designation of the desired command in the external module. This information is necessary in order that the thread be able to set the integer command field of the input structure to the appropriate value. Also, implementing a method for returning results to module commands is required for a thread to receive results from a command. Since the thread requires less information to exchange commands via the first method, it was used as the initial path through which threads would call external commands. This path, however, would change as shown in sections 4.8 and 5.4.

4.4 Prioritized Queues

A problem with adopting universal guidelines in all modules for blocking arises with the focus exposure. During a focus exposure, the telescope focus is changed several times and multiple images of the same target are recorded on a single frame in order to aid the comparison of the observed point-spread-function and rapidly determine the best focus position. However, if the telescope threads are blocked, then the focus commands will not take effect until after the exposure is completed, which defeats the purpose of doing a focus exposure. Obviously, extra features must be added to the system to account for the situation where the telescope or instrument should move while an exposure is occurring, as it is not limited to the scenario of doing a focus exposure. One way to solve this problem is to leave unblocked the thread on which telescope focus commands are executed. That way, the telescope focus commands would be executed correctly. This idea, however, would allow any telescope focus commands to be executed during a focus exposure, which may not be a desirable situation. There are other situations (dome flat exposures) where telescope or instrument commands must be executed during an exposure. Again, one could leave activated the telescope move thread while a flat exposure is taking place, but in such a configuration a user could accidentally move the telescope while the exposure was occurring. It seems that a different solution is required to solve these problems in a more secure fashion.

This different solution uses an array of pipes instead of a single pipe to send commands to each module thread. When a thread looks to read a new command, it searches the array of pipes in some order until it finds a non-empty one, at which point it removes a command from that pipe. The order in which the pipes are searched is determined by the priority value assigned to each pipe. Obviously, higher priority pipes are searched before lower priority pipes. The Tcl wrapper for a module command is augmented with code that searches for a priority flag, and if it finds one, writes the input structure to the pipe with the appropriate priority. Thus, appending a priority flag to a standard call for a module command allows the command to be executed at a selected priority; otherwise if no priority flag is present, the command is executed at the lowest priority level.

The block and activate commands have to be reconfigured to account for this new feature. The default action for the block command is to prevent a thread's reading commands from the lowest priority pipe. In this way, a focus exposure routine can issue block commands to prevent users from moving the telescope and instrument, but the camera module can issue focus commands at a higher priority, and they will be executed. An "activate" command with no arguments would still reactivate queues of all priority levels.

4.5 Command Results and Waiting

Despite the extra functionality provided by the prioritized pipes, modules, scripts, and the console still have no way of receiving results from commands sent to the modules. Without a method by which these objects could receive results, error checking would be impossible. Also, there is no way to synchronize execution of commands within a script.

The technique for returning results requires that a command be tagged in order that its results are returned. Default action for a command is that it is sent to the appropriate pipe, its results are ignored, and the interpreter returns immediately. If, however, the priority of the command is flagged with a negative sign, this indicates to the Tcl wrapper that it should wait for the command to finish execution, and return the results of the command to the interpreter. If the priority argument is negative, the module interprets the command to have a priority of the absolute value of the priority argument. Therefore, if a command is entered with a priority argument of -1, the interpreter knows to wait for the results, and the command is entered to the priority 1 queue. (Alternatively, the user can specify a *wait* flag with the command to indicate the desire to receive results from the command. This also delays execution of other commands until the results have been received.)

In order to store the results, each thread within a module is assigned a memory cell to which command results should be written. Each thread has only a cell (which can store one result at any time, as opposed to a queue, which could store multiple results) because the result cells will be used mostly for error checking. Therefore, if a series of commands is to be executed on a thread, any one command should receive results (to ensure that no errors occurred) before subsequent commands are run by the thread.

In order to understand better how results are managed within the modules, the status of the thread and the interpreter will be traced while a typical command is executed. (The communication paths between a module and the interpreter are illustrated in figure 6.) First, the Tcl wrapper parses the arguments and creates the input structure. Then, the interpreter writes the structure to the appropriate pipe by calling the *write* function for the module (the *write* function is explained in section 4.8), and immediately after queries the result cell for that pipe. Upon first querying the cell, the results will (most likely) not have been entered, so the interpreter will wait on a condition variable associated with the result cell. At some point, the thread will execute the command, and write the results to the result cell. After the results have been written, the thread will signal the condition variable, which wakes up the interpreter. Finally, the interpreter obtains the results and returns.

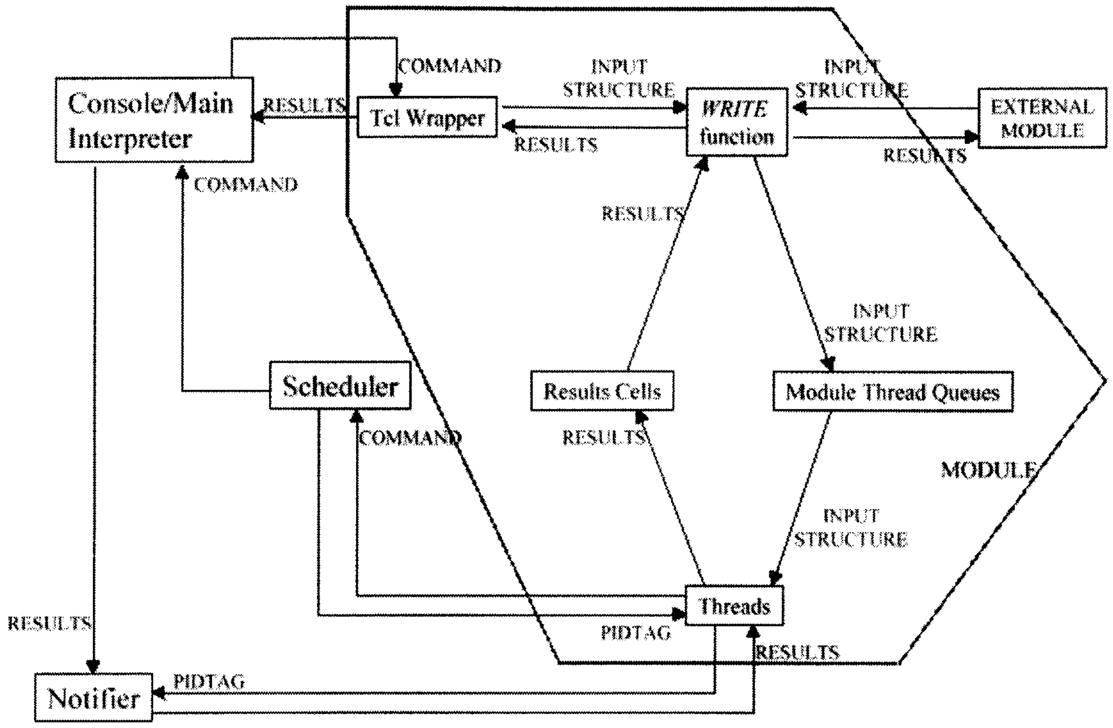


Figure 6 - Internal Command Paths for Module Thread Design with One Module (also shows paths to scheduler, notifier and other modules)

4.6 The Scheduler and Notifier in the Module Thread Domain

Since the module thread pipes offer a significant degree of command synchronization, the scheduler in this domain is simpler than originally designed. The notifier is also somewhat simpler than originally planned. The simplified scheduler and notifier were implemented as part of the execution domain, rather than as separate entities.

The scheduler and notifier will also use the paradigm of a wait flag (the negative sign preceding the assigned priority) to determine when a command would like its results returned. This practice (of forcing modules to explicitly specify when they want results back) keeps the results queue short, as most of the commands sent to the scheduler are graphical in nature, and results are not required for those types of commands. The structure of the scheduler and notifier might best be explained by tracing a command from the time it is sent to the core system by a module until the module receives the results from the command. Figure 6 can again be used to trace the path taken by the command. First, the module sends the command to the system by calling the *lois_send* function. The scheduler generates a *pidtag* and sends this to the calling module as a return value for the *lois_send* function (the scheduler assigns *pidtag* values successively starting from one). The scheduler then parses the command structure to determine if the command should be executed in the future, or if it should be executed multiple times (or both). If either of the previous conditions is true, the scheduler creates a thread for the command, which sleeps and then enters the command to the command queue at the appropriate time. If the script is to be executed multiple times, then the script performs the above two actions as many times as necessary. After this step, the scheduler determines the priority of the command, and sends the command to the main command queue. If the priority is preceded by a negative sign, the scheduler locks a mutex for the results queue, creates a new entry in the results queue that can be referenced by the *pidtag* that was returned to the module, and unlocks the mutex. Before the scheduler unlocks the mutex, it also sets the value of the new entry in the results queue to be empty.

When the command is executed on the main interpreter, the interpreter realizes that the priority of the command is flagged with a negative sign, and waits for the results of the command. When the results are obtained, they are enqueued in the results queue. The main interpreter then goes on to service the next command in the command queue. The module, meanwhile, has been issued a *pidtag* for its command. It then sends this *pidtag* to the notifier by calling the *lois_receive* function. Inside this function, the results queue is first locked. Then, if the results have not yet been enqueued (i.e. the result associated with the *pidtag* argument is empty), the function waits on a condition variable that is ultimately signaled when the results are enqueued by the core system. The action of waiting on the condition variable halts the module and releases the lock on the results queue. When the results are ready, the *lois_receive* function reacquires the lock on the results queue, obtains the results, frees the storage for the results and unlocks the results queue.

The *lois_send* and *lois_receive* functions are equipped with error checking capabilities. For example, the *lois_send* function returns and error if the priority of the command is illegal or if the main command queue is full. (The maximum number of elements in a kernel message queue is a system-dependent parameter, the LINUX version of LOIS supports a message queue of at most 256 elements, while the LOIS on Solaris can hold 1024 messages at a time.) The *lois_receive* function signals an error if there is no entry in the results queue associated with the *pidtag* argument.

4.7 Removing commands from queues

This feature refers to removing commands from main message queue, as well as the module thread queues. A user may accidentally enter a command to the system (some console interaction will be occurring in the wee hours of the morning, at high altitudes, under tight time constraints, etc.). Thus, allowing users to remove commands from queues before they are executed would be a useful characteristic for the system. A second use for command removal would occur if a user wanted to alter a command that has a significant amount of system control (such as a camera exposure). During a camera exposure, the system issues commands to block the telescope and instrument pipes. The user could remove these "block" commands before they are executed by the telescope and instrument threads, respectively. (Although issuing telescope and instrument commands at high priority and reactivating the telescope and instrument queues would achieve the same effect.)

Given a *pidtag*, the *lois_remove* function first blocks the main message queue and all unblocked module thread queues and prevents the issuance of activate commands. Next, it traverses through the main message queue and then through the module thread queues to find the associated command. If the search is successful, *lois_remove* removes the command and repairs the queue; otherwise it returns an error code. Before returning, the *lois_remove* function reactivates all queues that it previously blocked. Note that the presence of the main message queue and the module thread queues means that the system must be able to find a command, no matter its location. Using the *pidtag* to locate a command in a queue solves this problem. Since each command in the system has an associated *pidtag*, the *lois_remove* function can search for commands with a specific *pidtag* and remove only those commands.

The current *lois_remove* function is not particularly efficient. It searches each queue in the system until it finds the associated command. Another method for command removal would limit the number of queues searched by the *lois_remove* function. This design calls for each module (and the main command queue) to compute independent *pidtag* values. Since there will only be a single *lois_remove* function for all of LOIS, care must be taken to insure that no two modules (or a module and the command queue) issue identical *pidtag* values to different commands. Limiting the range of a *pidtag* returned by a module satisfies this requirement. As an example, the telescope module would only issue *pidtag* values in the range 1 to 1000, while the instrument module can use values between 1001 and 2000 and the main command queue utilizes the range 2001 to 3000. When a module reaches the maximum allowed *pidtag*, it can reuse *pidtag* values that were previously issued (so the telescope module would issue a *pidtag* of 1 to the 1001st command it sees). Such a paradigm also simplifies the *lois_remove* function. Since each module and command queue is assigned a specific range, when the *lois_remove* function is called, it only needs to search the queues corresponding to the range that contains the *pidtag* argument.

The main argument against this design is that a command's *pidtag* depends on where it is located in the system. For example, if the user enters a telescope move command, it will initially have a *pidtag* between 2001 and 3000. When the interpreter sends the command to the telescope module, the *pidtag* for the command will change to some value between 1 and 1000. Requiring the user to keep track of these changes is an unfeasible option. Instead, a table would have to be stored in memory that would associate the initial *pidtag* values (those values assigned to commands when they first enter the system) with *pidtag* values assigned by the modules. In effect, this transfers the burden of *pidtag* tracking from the user to the system.

4.8 Improvements

Despite the additions to the initial module thread design, some parts of the system still required refinement because they were inefficient. The most significant area of change was with the pipes through which commands would be sent from the Tcl wrappers to the module threads. The UNIX pipes are implemented at the operating system level, so every time a command is read from or written to the pipe, a system interrupt takes place. If these structures were moved to user space, they would become more efficient. There are two possible designs that would replace the functionality of the pipes. In one design, each pipe is replaced by an array of a fixed size, and in another design, a dynamically allocated array is used to replace each pipe. If a static array is used, a condition must be checked whenever a command is sent to the pipe to ensure that it is not full, whereas a dynamically allocated array never becomes full. Also, a dynamically allocated array is more space-efficient. The potential disadvantage of using a dynamically allocated array is in terms of speed. Whenever a command is written to an array, the *malloc* procedure must be called, and *free* must be used to remove the space associated with a command when it is read from a queue. Therefore, the speed of static vs. dynamic queues was evaluated. The results showed that dynamic and static queues function at about the same speed. Thus, dynamic queues were used since there are fewer restrictions associated with them.

A second design change was to create a *write* command within each of the camera, telescope and instrument modules. The responsibility of assigning commands to threads was moved to this function, so if a camera module wishes to write to a telescope thread queue, it simply calls the *write* function in the telescope module and passes an input structure and the desired priority as arguments, instead of having to know to which thread its command should be sent. The *write* command can also be used to obtain command results, instead of having calling modules explicitly lock and unlock the result cells. One concern with this approach is how a camera module would be able to write commands to the threads of any telescope module. Since all telescope modules will have a common interface, a camera module that is configured to correctly send commands to the test telescope module should be able to interact with any telescope module. Thus, the *write* function provides another method through which module-to-module communication can be achieved.

Another improvement on the original module thread design was to have a thread sleep while its command queue is empty. This improvement greatly reduces the load placed on the processor by LOIS, as originally each thread continuously searched its queue until a command was entered. With the introduction of multiple queues for prioritization of commands came the necessity to create arrays of flags that indicate the state (blocked/non-blocked) and the number of elements (empty/non-empty) of each queue. In the interests of space and time efficiency, these arrays of flags were represented as bits of an integer. The space efficiency of this approach is obvious, as one integer requires less storage than an array of integers. The time efficiency comes into play when determining if a thread should be put in the sleep state. Instead of having to determine, one by one, if each queue were either empty or blocked, the bit-wise operators inherent to C can be used to determine, in one operation, whether a thread has any queues that are both non-empty and non-blocked.

The module thread design solves many of the problems of version 1.0 of LOIS while providing a significant amount of execution control for the user. Nevertheless, the module thread design still left one problem unsolved. This problem occurs when a user or script wishes to obtain results from a camera exposure. While the camera exposure command is being evaluated, the main Tcl interpreter cannot service other commands. Thus, none of the graphics inherent in the camera exposure routine are displayed to the screen until after the exposure completes. The resolution of this problem is the subject of the next chapter.

Chapter 5 The Dual Interpreter System

The design improvements previously discussed solved a number of problems with the first version of LOIS. However, there still existed a specific class of problems that the module thread design could not solve. These problems appeared when the interpreter was required to execute one command (such as a graphical update) while it was occupied servicing another command. As a result, the main command queue would fill up, and eventually no more commands could be written to it. The solution, therefore, was to create a second interpreter that would help the main interpreter. By executing some commands in the second interpreter (most notably those that take long periods to execute) the main interpreter is free to service the main queue, so commands do not get lost.

5.1 Specific Cases Requiring the Use of a Dual Interpreter System

The first case has been visited several times in previous chapters. It occurs when a command containing graphical updates is entered to the system with a wait-flagged priority. Such commands include the *focus_go* command (defined in most telescope modules) as well as all camera exposure routines. If any of these commands are called with a wait flagged priority, the main interpreter blocks until the results from the command have returned. Thus, the graphics with these commands cannot be displayed until execution of the command has been completed. While this is not of primary importance with the *focus_go* routine, failure to update the graphical interface during a telescope exposure is a significant problem. If graphics are disabled during an exposure, no status information is displayed to the screen. Furthermore, exposures with wait flagged priority cannot be aborted, since the interpreter doesn't run the abort command until the exposure is complete.

The second case comes into play when long scripts are executed. Suppose a user constructs a script with many commands. The interpreter must run at least the Tcl wrapper part of the routines before it can service any other commands. Of course, if any of the commands are entered to receive results, then the interpreter must also wait until the thread parts of these commands are completed. While the interpreter is running the script, the user cannot enter any commands to the system, since the interpreter is occupied by servicing the script. Thus, if a user enters a script and realizes that the telescope is not at the correct position, rectification of this condition cannot occur until the script completes (which would most likely be wasted time). A more desirable solution would correct the problem while the script is still in its early stages.

5.2 General Design of the Dual Interpreter System

As the name indicates, the dual interpreter system employs two Tcl/Tk interpreters, each of which is capable of executing commands. Since graphical commands must be executed on a single interpreter, the Tk graphical package is loaded only into the main interpreter, the graphics interpreter. At initialization, the main program creates a separate thread in which the secondary (command) interpreter runs. Commands entered through the console are interpreted in the command interpreter, and if nothing is known about the commands in the secondary interpreter, they are executed in the graphics interpreter. This approach solves the first problem above because the wait-flagged exposure command causes only the command interpreter to wait for a result. The graphics interpreter can continue to service commands, and graphics are displayed to the screen. While a long script is executing, the dual interpreter system provides a path into the system through the graphics interpreter. Thus, if commands to abort or pause scripts are defined in the graphics interpreter, the user can stop (or cancel) the script. After the script has been signaled, the user can enter instructions to the command interpreter to correct the problem.

The presence of the second interpreter mandates a change in each telescope, instrument and camera module. Commands that may take a while to complete must be defined in the secondary interpreter, so that the main interpreter is free to service graphical and other commands. Therefore, some reconfiguration of each of the telescope, instrument and camera modules is necessary. Instead of restricting definition of every routine in each module to the main interpreter, all routines (except for the initialization routine, status routine, block and activate routines and FITS header routine) are part of both interpreters.

The initialization routine is run only a few times (if not once) and contains only graphics and other commands that are executed quickly. The status routine in each module updates the graphics of the corresponding window, so it should be run in the main interpreter. The FITS header routine is part of the main interpreter because the storage module is loaded into the main interpreter. All commands in the storage module execute quickly, so they do not delay the main interpreter and cause the command queue to fill up. If, in the future, the storage routines must be defined inside of the secondary interpreter, then the secondary interpreter would be capable of running the FITS header routines in each of the telescope, instrument and camera modules. The reason for defining the block and activate commands inside of the graphics interpreter will be explained after the inner workings of the dual interpreter system are explained in more detail.

5.3 Structure of the Dual Interpreter System

Before the implementation of the dual interpreter system is described, recall a disadvantage of the multiple interpreter system as a design for the execution domain. (section 3.5, paragraph 2) The problem occurred when an interpreter invoked the "eval" command. While the "eval" command was executed, no commands could be run on any interpreters in the hierarchy. Therefore, if both the command and graphics interpreters are to run commands simultaneously, neither can invoke the "eval" command as part of a Tcl script. In the original design, commands entered to the console were invoked with the "eval" command. In order to circumvent this problem, commands entered to the console are parsed, and then passed to a secondary command queue. This queue, like the main command queue, is also prioritized. The command interpreter (which was initialized in a separate thread) reads commands from the secondary command queue one at a time (based on priority) and executes them.

When a Tcl interpreter encounters an unknown command, it runs a procedure titled *unknown*. The default *unknown* procedure displays an error message to the standard output. Since all graphical commands (and some other commands) are defined only in the graphics interpreter, the secondary interpreter has no knowledge of these commands. If a user is to include graphics inside a script, the graphics interpreter must evaluate these commands. Therefore, the secondary interpreter is configured so that its *unknown* command immediately evaluates commands in the graphics interpreter. If the command is also undefined in the graphics interpreter, an error message is returned. (Obviously, the default action for the graphics interpreter cannot be to send the command to the secondary interpreter. If this were the case, unknown commands would bounce between the interpreters.)

Since scripts and users now have access to two different interpreters, consistency among variables must be maintained for both interpreters. The main choice in providing this paradigm is the scope of variables in the command interpreter. In one design, all script and user variables are part of the graphics interpreter, and no variables are defined as part of the command interpreter. Also, variables are automatically referenced in the scope of the graphics interpreter. Thus, the additional scopes created by procedures are ignored in the first design. The second design allows variables to exist within the command interpreter. When referencing a variable in this design, the variable lookup starts in the global scope of the graphics interpreter. If the variable does not exist in the graphics interpreter, the local scope of the command interpreter is searched. Finally, the global scope of the command interpreter is searched. If the variable does not exist in either interpreter, an error message is returned.

The only difference between the two designs is the range of possible scopes in which variables can be defined. The first design only allows definition of variables at the global scope in the graphics interpreter, which imposes a significant inconvenience on users who wish to create their own scripts. First, the user must ensure that all variables (both global variables and those defined inside procedures) have unique linkage. That is, if one procedure in the script creates a variable *i*, all references to *i* within the script (including references in other procedures) must refer to the same variable. Second, the user must also ensure that variables created within his script have different names from previously existing variables within the graphics interpreter, including variables defined in other users' scripts. Since the second design does not suffer from these deficiencies and it allows a broader range of scopes, it is the one being employed by the dual interpreter system. Figure 7 shows the interface paths between the two interpreters, the console, the graphical interface, and (for simplicity) one module. Note that the module in figure 7 retains the communication paths shown in figure 6.

Note that scripts and console commands are currently parsed and executed by the secondary interpreter. Therefore, if a script is running, the secondary interpreter cannot service console input, so the user cannot enter console commands to cancel the script while it is being parsed. Instead, the user must first abort the currently executing script by calling the "abort command" function, which runs in the graphics interpreter. This function terminates the current script, so the user can make adjustments. In the future,

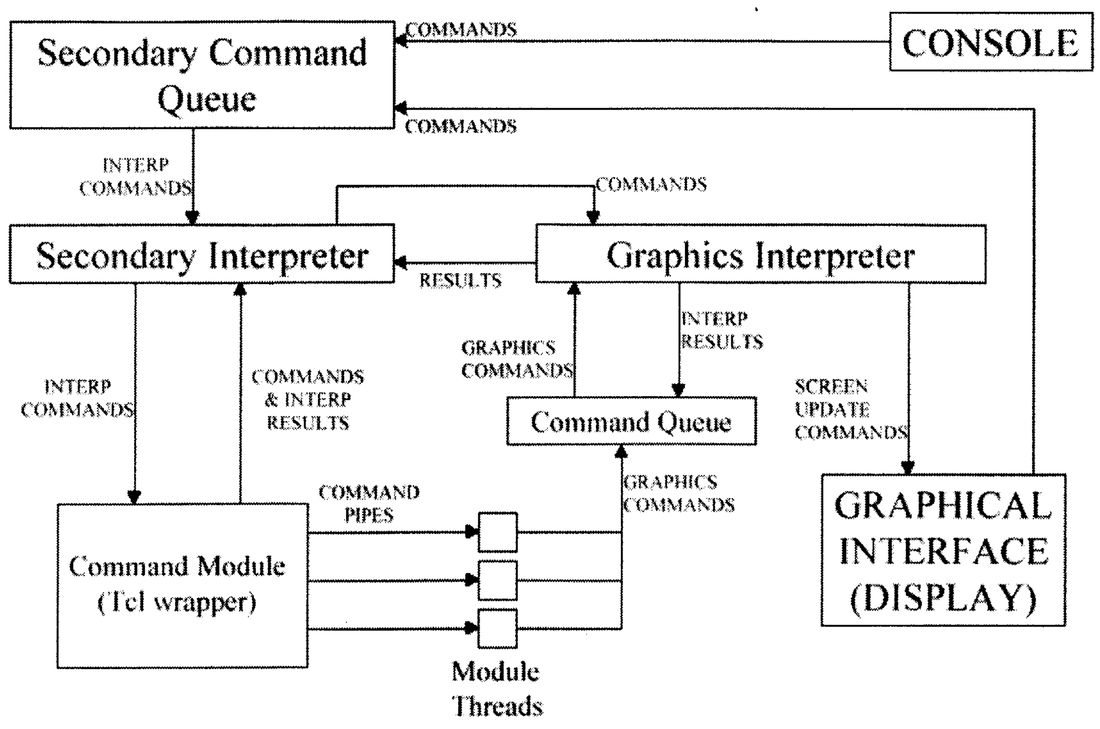


Figure 7 - Dual Interpreter System Internal Command Paths

users will be able to terminate (or pause) scripts and commands by entering console commands. This feature will be provided by creating another console for the graphics interpreter or linking input from a single console to the graphics interpreter when the secondary interpreter is busy.

5.4 Design Choices Implied by the Dual Interpreter System

The use of the dual interpreter system as described above necessitated several design choices in other areas of the core system. The first such area lacking a definite path was the transfer of commands between modules. Since most functions in the modules are defined as part of the secondary interpreter, external commands cannot be called via the main command queue. Two possible solutions for this dilemma arose. One required any module to send external commands directly to the thread pipes of other modules, (via the *write* command as discussed in section 4.8) and the second involved calling external commands via the secondary command queue. As a concrete example, the second design would work better if the *focus_go* commands of the various telescope modules have different prototypes (number and meaning of arguments, such as the command designation). Since most modules will have similar *focus_go* functions, the first solution is appropriate, and was actually implemented. In addition, the secondary command queue was designed so that only the console could write commands to it. So allowing modules to write to the secondary queue would require more change than would implementation of the first design. If, however, another function is created that has widely varying prototypes across a set of modules, then the second solution will be implemented.

Another design choice that was alluded to above was the placement of the "block" and "activate" routines in the main interpreter. The "activate" routine was linked to the main interpreter because the thread that runs the camera exposure routines reactivates the instrument and telescope pipes when the exposure is finished. Since "activate" is called from within the thread part of the camera module, at the time of the call the camera module has no handle on either the graphics or command interpreter. Thus, it must send the "activate" command to a queue. Since only the console has access to the secondary queue, the "activate" command must be sent to the main queue, which requires that the "activate" command be defined within the main interpreter. The "block" routine is defined inside the main interpreter only because the "activate" routine is also defined there. There is no known reason why a "block" command cannot be moved to the secondary interpreter.

The introduction of the secondary interpreter also mandates a design change in the *lois_remove* function described in section 4.7. Since the secondary interpreter is now responsible for executing scripts and running the majority of the commands, it issues *pidtag* values to all commands that are removed from the secondary command queue. In order that the modules use consistent *pidtag* values, the secondary interpreter records the *pidtag* of the command that is currently in execution and updates this value every time it begins execution of a new command. Therefore, if the secondary interpreter passes a command to a module, the module can access the *pidtag* of the command it received, and record this information in the input structure before calling the *write* command. Since every command in the system is labeled with a *pidtag* value, the *lois_remove* function can search for a particular command by traversing the main and secondary command queues, as well as the module thread queues.

5.5 Command Execution (version 1.0 vs. version 1.1)

This section will serve to summarize all design changes between versions 1.0 and 1.1 of LOIS. The changes will be illustrated by following the path of execution for several series of commands in both systems. The first series consists of a focus exposure.

First, the original LOIS will be analyzed. After the command is parsed and the arguments are read in, the focus exposure creates a separate process in which to run the exposure. The array is exposed and the focus is moved in increments, but the new focus values are not displayed to the screen, since they are executed in a child process. The frame is read from the CCD array, at which time it is stored and displayed. Note that the display spawns another child process, further increasing the memory usage of LOIS. Finally, the focus exposure routine terminates both child processes and returns to the interpreter. During the focus exposure, any other commands can be run, particularly a focus exposure. This is because a child process executes the focus exposure, thus allowing the parent LOIS to run other commands.

In the new version of LOIS, the command is parsed and the arguments are loaded into a structure. The focus routine then issues commands to block execution of telescope and instrument routines, writes the structure to a pipe, and returns to the command interpreter (since the priority was positive). Here are two advantages of the new system. First, it can run other commands without having to copy the LOIS process. Second, no telescope or instrument commands will be run until their respective pipes are reactivated. At any rate, the exposure thread waits for the telescope and instrument threads to become inactive, and then reads the exposure routine (and its arguments) out of the pipe. After performing some operations with the arguments, the thread part of the focus exposure routine creates another thread to move the focus, and expose and read out the CCD array. Since the graphics associated with the focus routine are sent to the main command queue, they are displayed to the screen as the focus changes. Furthermore, the use of a thread for the exposure routine greatly increases the memory efficiency of LOIS. After the frame is read out, it is stored and displayed (without the aid of an additional process). Finally, the thread part of the focus exposure routine returns, and the telescope and instrument pipes are reactivated. An additional advantage of the dual interpreter system is realized if the focus command is entered with a negative priority. Instead of blocking the main interpreter, preventing the update of the focus value, the command interpreter is blocked, so the main interpreter can evaluate the commands necessary to change the displayed value of the focus.

Now consider a sequence of commands as follows. The sequence starts with a relative telescope move of 5 degrees RA and 5 degrees declination. The filter wheel is then rotated one position, and a single exposure occurs. These three commands are repeated any number of times.

For further comparisons, issues that affect the performance of the system (such as memory usage) will not be considered. The two versions will be compared only on how the commands are executed. When faced with the above sequence of commands, version 1.0 executes each command serially. Therefore, before the filter wheel can move, the telescope must finish moving. Of course, the camera will not start exposing until the filter wheel has finished moving, so those commands that must be executed synchronously are executed synchronously. However, the filter wheel and telescope could likely move simultaneously, so there is a bit of inefficiency when executing this sequence of commands. Also, the single Tcl interpreter cannot service any other commands while it is running the filter wheel and telescope commands, so these commands cannot be canceled under the version 1.0 scheme.

The functionality of the module thread design with this script will now be evaluated. Suppose that none of the commands in the script are wait-flagged. The command interpreter parses the entire script, entering the commands into the module thread queues. Since this takes very little time, the command interpreter can service other commands almost immediately after the script is entered into the system. Thus, cancellation of any of the commands in the script is possible. The relative move and filter wheel commands execute simultaneously, since they run on different threads. The exposure command (*single*) makes sure to block the telescope and instrument pipes before beginning the exposure. Thus, those commands that must be executed synchronously (the *single* commands) are executed synchronously, but those commands that can run simultaneously are not executed synchronously.

As a final example, consider a script containing commands that alternate between telescope relative moves and single exposures. In other words, the script looks like *rmove, single, rmove, single, etc.* Once again, version 1.0 of LOIS will execute each command in the script one at a time. This paradigm ensures that while an exposure is occurring, none of the *rmove* commands will be running. Since the telescope cannot move while an exposure is occurring, every command in the script must be executed synchronously.

Now consider the case where the module thread execution domain executes this script. Since every command in the script must be executed synchronously, the parallel processing afforded by the threads is of no advantage, in fact, care must be taken to ensure that no command in the script starts until the one preceding it finishes. The command interpreter parses the script, and first sends the relative move to the telescope module. When the exposure is written to the camera module thread, it sends block commands to all threads in the telescope and instrument modules. The exposure routine then waits for all of these blocks to take effect, and when they do, the exposure begins. Thus, the telescope module cannot execute any commands while the exposure is occurring. When the exposure finishes, the telescope and instrument threads are reactivated, and the execution domain runs the next relative move command. Note that because each command must be executed synchronously, the module thread design executes this script no more efficiently than the version 1.0 system does (not considering memory usage). However, the module thread design should not be able to execute this script any faster. Furthermore, the module thread design has the additional benefit of being able to abort commands in the queue.

Chapter 6 Future Work/Improvements

The module thread domain and dual interpreter configuration greatly enhanced the execution environment of LOIS, giving the user more control over those commands that should be run synchronously, while also increasing the efficiency of the system. Some features, however, are currently not implemented due to priorities among the different aspects of the new core system. Other features could be made more efficient by changing their implementation. Both types of features are listed here; in addition, each feature is accompanied by a general description of a design that would provide the feature.

6.1 Multiple LOIS processes

The majority of use cases for LOIS will have only a single process running at any time. However, the case of multiple LOIS processes must be accounted for. Suppose that a team of two users is in the middle of an observing run. One user is recording data and running scripts, while the other user is analyzing previously captured frames. In such a situation, if both users were running sessions on the same machine, two LOIS processes would be running simultaneously on that machine. With the current system, if the machine were running the LINUX operating system, the second LOIS process would die trying to initialize the shared memory. The secondary process would also corrupt some of the data in the main command queue of the first LOIS process. These problems might be avoided if the machine were running a version of Solaris. However, difficulties may be encountered with conflicting *pidtag* values (since both processes would issue *pidtag* values consecutively starting from 1) and with the display software (since both processes would attempt to write data to the same display process).

Under LINUX, when shared memory is assigned, it is associated with a key (an integer between 1 and 128). The keys for the shared memory blocks (CCD, telescope and instrument vectors and structures, the information vector and image buffer) are defined in a header file and are constant for any LOIS process. Therefore, if one LOIS process is running, when a second process tries to initialize the shared memory, it encounters an error because the keys it wants to use have been claimed by the first process. An easy workaround for this problem is to have the secondary process choose keys that have not been previously claimed. This solution allows two processes to run, but it is limited to having a maximum of 21 LOIS processes running simultaneously. (Each LOIS process requires six keys, one each for the CCD, telescope and instrument vectors and structures, one for the information vector, and one each for the image buffer and the main message queue. Thus, 21 processes would use 126 keys. Since there are only 128 keys in total, 21 LOIS processes can run simultaneously.)

Although the *pidtag* values passed to scripts and modules by each process reside in separate address spaces, problems will occur if one process has access to the *pidtag* values of another process. In particular, a module in one process could wait on a particular *pidtag*, while another process could set the results for that *pidtag*, or remove the *pidtag*. In either case, an error would occur because the first process would wait forever (if it tried to receive the results associated with a removed *pidtag*) or it would return with incorrect results, and the true results would remain in the queue.

Displaying images generated by multiple LOIS processes may also introduce problems. If two or more processes simultaneously write data to a buffer controlled by the display software, the displayed image might be a combination of the frames captured by each process. One technique that eliminates this problem calls for display module routines to lock any image data buffers before writing to them. After an image has been displayed, the display module unlocks any locked buffers. This procedure ensures that only one LOIS process can write to the image display buffers at any time.

6.2 Remote Access

Remote command execution remains as an unspecified feature of LOIS. Especially as it relates to MANIAC and some other programs, LOIS should allow a user to interact with the system as if running it locally. This idea has two parts, first, the user should be able to enter commands and have those commands transmitted to the local copy of LOIS, and second, the user should see updates, as would a local user. (This is subject to interpretation, because some parameters may not apply or may not be of interest to a remote observer.) Furthermore, the user should still have the ability to run scripts remotely and to abort scripts that are in the process of execution.

The most likely design for a remote command facility would not have a user connect directly to a locally running LOIS session. Instead, the remote user would run LOIS on a remote machine with the correct set of modules to interact with the LOIS process running on the machine at the observatory (the master LOIS). The remote LOIS, however, would not initialize any of the module threads or associated pipes. Instead, commands would either be sent from the wrapper part of the modules to the pipes of the master LOIS, or command entered to the console of the remote LOIS would be sent to the secondary command queue of the master LOIS. The second approach would likely be the one that is utilized, since the first approach has multiple secondary command queues, and hence multiple structures that issue *pidtag* values. This condition could cause an error when a *lois_remove* function is called. The remote LOIS would also utilize the main command queues and the results queue of the master LOIS process.

While the general idea of this feature could easily be implemented with standard UNIX sockets and a TCP/IP network connection, there are a variety of details that need to be specified. Such details include the list of functions that may be run from a remote session, as well as the relative priorities of local and remote commands. For example, suppose that while a remote user is moving the telescope, weather or wind conditions change rapidly. A local user should be able to override the remote move so that the telescope can be safely stowed. Providing this feature in the local system is rather straightforward (one could use the prioritized queues and enter remote commands at a lower priority than local commands). However, the local core system also must be able to differentiate between commands of remote origin and those that were generated by the local process itself. By requiring calling modules to provide an IP address as an argument to the *lois_send* function, the local core system would be able to determine from where a command was generated, and assign to the command the appropriate priority.

Another situation in which the protocol is unclear occurs when the network connection being used by a remote user malfunctions. In such a situation, it may be advantageous to transfer control of LOIS to a local user so that the observing run can be completed. If maintenance or on-site monitoring is unnecessary, control should be transferred to another remote user. Such decisions will most likely be made by telephone coordination between local and remote personnel, since at such a point any data network transmissions would be disabled.

References

[1] Burnett, Keith, *www.xylem.demon.co.uk/kepler/altaz.html*

[2] Dietz, Hank, *yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto-2.html#ss2.4*

[3] Dunham, Edward W. and Taylor, Brian W., "Lowell Observatory Instrumentation System Functional Requirements."

[4] Dunham, Edward W. and Taylor, Brian W., "LOIS Internal Command Interface Paths."

[5] Dunham, Edward W. and Taylor, Brian W., "LOIS Execution Kernel."

[6] Nichols, B., Buttlar, D., and Proulx-Farrell, J., Pthreads Programming, © 1996 O'Reilly & Associates, Inc., Sabastopol, CA 95472

[7] Welch, Brent B., Practical Programming in Tcl and Tk, © 1997 Prentice-Hall, Inc., Upper Saddle River, NJ 07458

Appendix A – Glossary of Terms

Bias: An exposure frame of zero length. A bias exposure is used to measure the readout gain and other hardware characteristics of a CCD array.

Binning: The binning factor is the squareroot of the ratio of the resolution of the CCD array to the resolution of the displayed (and stored) image. Suppose that an image is captured on an array with 800 rows and 800 columns with a binning factor of 2. Then the displayed image will have 400 rows and 400 columns, and each pixel in the displayed image will be some combination of a corresponding set of four pixels of the CCD array.

Charged-Coupled Device (CCD): An array of light-sensitive elements. An electrical potential difference is maintained across each element of the array. When light strikes an element, the potential difference of the element is changed to reflect the amount of light energy that struck the element. In this way, a CCD array can transform signals of light energy (photons) into signals of electrical energy (electrons) that can be interpreted by a digital signal processor or computer.

Dark: An exposure frame that results from not exposing the CCD array. A dark exposure is used to measure the "dark current" associated with a CCD array, or the level of the background signal generated by sources other than light (heat, electrical, etc.)

Declination (Dec): A coordinate used to express positions of objects on the celestial sphere. If one views all objects in the sky as being located on a sphere surrounding the Earth, this sphere is called the celestial sphere. The declination of an object on the celestial sphere measures the angle between the object and the celestial equator, which is defined as the circle that is concentric with respect to the Earth's equator. Declination is the celestial equivalent of latitude.

Flat: An exposure frame that is used to measure the responsivity of a CCD array. Flat exposures are usually taken at dawn or dusk, when the sky is somewhat light. Flats reflect the illumination of the "background sky", and can be used to normalize data obtained in object frame exposures.

Focus Control: A telescope command that moves the focus at the specified speed for a given time period.

Focus Exposure: An exposure mode used to fine-tune the telescope focus for subsequent exposures. During a focus exposure, the following process is repeated. First, a number of rows are exposed. The number of exposed rows is given by the *shift_width* parameter. After the rows are exposed, they are shifted to another location on the CCD, the focus is changed by a fixed interval (called the *focus_shift*), and the process repeats. The number of iterations performed is given by the *num_of_shifts* parameter.

Focus Go: A telescope command that moves the focus to the given position.

Object: An exposure frame used to capture an object of interest. Only in an object frame can different exposure modes be expressed. The single exposure mode is always used in the bias, dark and flat frames.

Relative Move: A telescope command that changes the right ascension and/or the declination of the telescope's field of view by the specified interval(s), measured in arcseconds.

Right Ascension (RA): A coordinate used to express positions of objects on the celestial sphere. RA is the equivalent of longitude on the celestial sphere. The RA is the angle measured east from the vernal equinox, which is defined as the zero point. RA is usually expressed as time, rather than degrees or radians.

Single: An exposure mode in which the array is exposed for a period of time and then read out. This exposure mode is the only mode available to bias, dark and flat frames.

Slow Dots: An exposure mode similar to the focus exposure. During a slow dots exposure, the following process is repeated. First, a number of rows are exposed. The number of exposed rows is given by the *shift_width* parameter. After the rows are exposed, they are copied to another location on the CCD, and the process repeats. The number of iterations performed is given by the *num_of_shifts* parameter. Note that the same area of the CCD array is being exposed, then shifted, then exposed, etc. In this way, a slow dots exposure produces several copies of an object in one frame, where each copy is of a different brightness.

Strips: An exposure mode in which the CCD array is exposed, and particular rows of the CCD are read out at specified intervals.

Appendix B – User's Guide Addendum

This appendix is meant to serve as an addendum to the LOIS User's Guide currently being written by Brian Taylor. It will explain how some of the new features added by the module thread domain (such as priorities and synchronous vs. asynchronous execution) can be used.

For the following discussion, a "module" refers to any instrument, telescope or detector module. Currently, the storage and display modules do not conform to the module thread design. It is likely, however, that in the near future, the following argument could also be applied to the storage and display modules.

Every command in a module executes asynchronously at priority level 1 by default. By specifying a "-wait" flag, the command will execute synchronously at priority level 1. The "-wait" flag *must* be specified after all other arguments, otherwise the system will raise an error, indicating the correct format for the command. Commands entered through the console can also be assigned higher priority levels. This is accomplished by appending the "-prio" flag to the end of the command, followed by the desired priority level. Again, the "-prio" flag and the new priority level for the command *must* go after all other command arguments. If this format is not met, the system will return, as an error message, the correct format for the command. An error message is also returned for an illegal priority value. Note that these flags can only be appended to commands that are entered through the console. Commands entered through the GUI always run at priority level 1. The following examples show how to enter commands to the console using the "-wait" and "-prio" flag.

Enter a focus_go command with the "-wait" flag.
```
LOIS % focus_go 60 –wait
```
Enter a focus_ctl command with the "-prio" flag (second command has negative priority).
```
LOIS % focus_ctl –spd slow –time 10 –prio 2
LOIS % focus_ctl –spd slow –time 10 –prio -3
```

Without any arguments, the *BLOCK* routines block each pipe in a module from executing lowest priority commands. The user can provide arguments to the *BLOCK* commands to indicate that only a subset of pipes should be blocked. The labels of the threads in the telescope module are "move", "focus", and "other", the threads in the camera module are titled "status", "exposure", "abort", and "other", and the instrument module has a "filter" thread and an "other" thread. Given these designations, some example commands and their results will be presented.

```
LOIS % tel_block
```
(Result: Blocks all telescope threads, "move", "focus" and "other")
```
LOIS % inst_block filter
```
(Result: Blocks only the "filter" thread in the instrument module)
```
LOIS % tel_block move other blahblah
```

(Result: Blocks the "move" and "other" threads in the telescope module, the "blahblah" argument, which corresponds to no thread in the telescope module, is ignored.)

A small testing bed was developed with which users can test the status of the system. The tests are located in the file *tests.tcl* in the root LOIS directory, so to make the commands available to a session of LOIS, enter the following command into the console:

```
LOIS % source $LOISHOME/tests.tcl
```

This command will list the names of each test. The convention is that a test whose name is of the form "gr_test*i*" contains graphical components, while a test named "test*i*" contains no explicit graphical components (it may, however contain exposure routines, which themselves generate graphics). In order to obtain a description of what a particular test does, enter the test name followed by a question mark to the console. For example, to learn about what routine *gr_test2* does, enter:

```
LOIS % gr_test2 ?
```

A general summary of the test, which includes the significance and default values of any arguments, is printed to the screen.

Appendix C

Brian Taylor
Lowell Observatory
1400 W. Mars Hill Road
Flagstaff, AZ 86001

```
#ifdef __LINUX__
#include <sys/ipc.h>
#include <sys/msg.h>
#elif defined(__SOLARIS_5x__)
#include <mqueue.h>
#endif

#ifdef __LINUX__
#include <sys/shm.h>
#endif
#include <string.h>

#define IMAGE_BUFFER_SIZE       700000 /* 4096000 */
#define IMAGE_BUFFER_NAME       "/image_buffer"

#define NONE     -1
#define IRAF      0
#define FITS      1
#define RAW       2

#define OFF      0
#define ON       1

#define SAVE_STATE    0
#define RESTORE_STATE 1

#define MAX_PRIORITY 30
#define DEF_PRIORITY 15 /* Default priority for commands send to queue */
#define MIN_PRIORITY 0

#define ABSOLUTE(x) ((x) < 0 ? -(x) : (x))



int shm_fd;
#ifdef __LINUX__
/* structures for shared memory information blocks */

#define IMAGE_KEY 10
#define CCD_KEY   11
#define TEL_KEY   12
#define INST_KEY  13
#define INFO_KEY  14

#define QUEUE_KEY 100
#define MSG_Q_SIZE 256  /* size of text messages (in bytes) sent to graphics
                           thread. For some reason, this cannot be 1024 (like
                           it is on the Sun LOIS) because that value hangs on
                           multiple exposures */
#endif

#ifdef __SOLARIS_5x__
mqd_t           lois_queue;
#else
int             lois_qid;
#endif

/* The Display Structure */
typedef struct {
                CDLPtr  cdl;
                int     status;
                int     frame;
                int     fbconfig;
                int     zscale;
                int     format;
                char    fname[80];
```

```
                int     nx;
                int     ny;
                int     depth;
                int     hskip;
                float   z1;
                float   z2;
                int     fb_w;
                int     fb_h;
                int     nf;
                int     lx;
                int     ly;
} display_struct;


typedef struct {
                int  mem_fd;
                char observers[80];
                char obsaffil[80];
                char loisvers[30];
                char ccdmod[30];
                char telmod[30];
                char instmod[30];
                char dispmod[30];
                char *loishome;
                char *cfghome;
                char path[80];
                char filename[80];
                Tcl_Interp *lois_main;
                Tcl_Interp *lois_grx;
                Tcl_Interp *lois_cmd;
                void *memory;
                unsigned int imageno;
                unsigned int store;
                unsigned int test;
                unsigned int focus;
                unsigned int dome_flat;
                unsigned int frameno;
} info_struct;

typedef struct {
                int             priority; /* Priority level of command */
                unsigned int    t_interval; /* delay between repetitions of
                                               command execution. */
                unsigned int    t_times; /* Number of times to repeat command*/
                unsigned int    t_execute; /* When to execute command */
                char            command[256]; /* Command to execute */
} cmd_struct;

/*************************************************************************/
/********************** Command Queue Structures ***********************/

#include <pthread.h>

struct _command {
  cmd_struct value;
  long pidtag;
  struct _command *next;
};

typedef struct {
  int             blocked;
  int             waiting;
  struct _command *head;
  struct _command *tail;
  pthread_mutex_t *mutex;
  pthread_cond_t  *go;
```

```
} command_queue;


/*************************************************************************/
/********************** Results Queue Structures ***********************/
#define RESULT_LEN 256
#define WAIT_STATUS -100

typedef struct {
  int rtn_status;
  char rtn_message[128];
} lois_results;

struct resq_elt {
  int key;
  struct resq_elt *next;
  lois_results value;
};

typedef struct {
  int cur_count;
  pthread_mutex_t *mutex;
  pthread_cond_t *go;
  struct resq_elt *head;
} resq;

/**************** End results queue structures ************************/
/*********************************************************************/

extern display_struct l_display;

Tk_Window       tk_mainwindow, tk_mapmain;

int lois_main(Tcl_Interp *lois_interp);
int lois_send (cmd_struct * command);
#ifdef __LINUX__
typedef unsigned int useconds_t;
#endif
int lois_usleep(useconds_t usec);

/*Logging function*/
int lois_log0(char * message, ...);
int lois_log1(char * message, ...);
int lois_log2(char * message, ...);
int lois_log3(char * message, ...);
int lois_log4(char * message, ...);
int lois_log5(char * message, ...);

/* Shared memory functions */
void shm_read(void *to, void *from, int size);
void shm_write(void *to, void *from, int size);

/*
Inter-Module communication functions that allow communiction between
modules without going into the Tcl Interp.
*/

extern char * Lois_GetVar (char *variable, int flags);
extern Tcl_Obj * Lois_ObjGetVar2(Tcl_Obj *obj1_ptr, Tcl_Obj *obj2_ptr,
                                 int flags);
```

```
Tcl_ObjCmdProc *ccd_slave_func[ccd_slave_NUM_FUNCTIONS] = {
                        ccd_single,
                        ccd_series,
                        ccd_strips,
                        ccd_sdots,
                        ccd_fdots,
                        ccd_focus,
                        ccd_go,
                        ccd_test,
};

Tcl_ObjCmdProc *ccd_main_func[ccd_main_NUM_FUNCTIONS]= {
                        ccd_init,
                        ccd_gui,
                        ccd_delay,
                        ccd_activate,
                        ccd_update,
                        ccd_bit,
                        ccd_status,
                        ccd_open,
                        ccd_close,
                        ccd_abort,
                        ccd_pause,
                        ccd_header,
                        ccd_wait,
                        ccd_readparam,
};

/***************************************************************************/
/*************************  CCD definitions *****************************/
/***************************************************************************/

#define NUM_ELEM 12
char *ccd_array[NUM_ELEM] = {
                "exptime",
                "numexp",
                "row",
                "col",
                "pscan",
                "oscan",
                "binning",
                "exp_unit",
                "title",
                "comment1",
                "comment2",
                "frame",
};
#define CCD_main_NUM_FUNCTIONS  14
#define CCD_slave_NUM_FUNCTIONS  8

char *ccd_main_cmds[CCD_main_NUM_FUNCTIONS]= {
                        "ccd_init",
                        "ccd_gui",
                        "ccd_block",
                        "ccd_activate",
                        "ccd_update",
                        "ccd_bit",
                        "ccd_status",
                        "ccd_open",
                        "ccd_close",
                        "ccd_abort",
                        "pause",
                        "ccd_header",
                        "waitcam",
                        "getparam",
};
```

```
char *ccd_slave_cmds[CCD_slave_NUM_FUNCTIONS]={
                         "single",
                         "series",
                         "strips",
                         "sdots",
                         "fdots",
                         "focrun",
                         "go",
                         "test",
};

/* CCD Mode Array */
#define NUM_MODES       7

char *ccd_mode[NUM_MODES] = {
                "Functions",
                "Single",
                "Series",
                "Strips",
                "SDots",
                "FDots",
                "Focus"
};

/* end CCD definitions */
```

```
#include <pthread.h>
#include <semaphore.h>

#define NUMCCDTHREADS   3
#define NUMINSTTHREADS  2
#define NUMTELTHREADS   4
#define NUMPRIORITIES   4 /* number of total priorities (including lowest priori
ty)
                             priority 4 is a system only priority

/************************************************************************
***/
/************************* miscellaneous structures ************************
***/
/************************************************************************
***/

#define MAX_ARGS 10

/* Thread pipe Errors */

#define EILLPRIO 0
#define EINTRNL  1
#define EBADWAIT 2
#define ENOCMD   3
#define EREMOVED 4

#define WAITOPT "-wait"  /* Option supplied to commands, sets priority to -1 */

static char write_errs[5][80] = {
        "Illegal priority\0",
        "Error in internal command\0",
        "Bad argument in BLOCK\0",
        "Unknown command identifier\0",
        "Command removed from queue\0"
};

int command_removed_flag;

/* End thread pipe Errors */

#define CCD_IND     0
#define INST_IND    1
#define TEL_IND     2
#define STATUS      0

struct _input {
  long pidtag;
  int command;
  int numargs;
  struct _input *next; /* for dynamically allocated module thread queues, if
                          we decide to go with that implementation */
  char args[MAX_ARGS][20];
};

typedef struct _input input;

typedef struct {
  input *head;
  input *tail;
} q_struct;

typedef struct {
  int non_blocked; /* Integer whose bits represent whether a queue is blocked
                      or not.  If the bit is 0, the queue is blocked (that is,
                      the thread doesn't read from the queue).  The LSB
```

```
                      signifies the lowest priority queue */
  int non_empty; /* Integer whose bits represent whether a queue is empty or not
                    If the bit is 0, the queue is empty.  The LSB signifies the
                    lowest priority queue */
  q_struct p_queue[NUMPRIORITIES];  /* dynamically allocated queues */
  pthread_mutex_t *mutex;
  pthread_cond_t *go;
  input curr_run;
} input_queue;

#define RUN_REMOVED -2
#define RUN_ERROR   -1
#define RUN_OK       0
#define RUN_WAIT     1

typedef struct {
  int value;
  pthread_mutex_t *mutex;
  pthread_cond_t *go;
} output_block;

/************************************************************************
***/
/************************* instrument module threads ************************
***/
/************************************************************************
***/

void * INST_main(void * arg);
pthread_t inst_thread[NUMINSTTHREADS];
input_queue inst_cmds[NUMINSTTHREADS];

/* Synchronization objects for thread blocking routines */

/* pthread_mutex_t *inst_block_mutex[NUMINSTTHREADS];
   pthread_cond_t *inst_block_free[NUMINSTTHREADS];
*/
sem_t *inst_block[NUMINSTTHREADS];
output_block inst_results[NUMINSTTHREADS];

/************************************************************************
***/
/************************** telescope module threads ************************
***/
/************************************************************************
***/

void * TEL_main(void * arg);
pthread_t tel_thread[NUMTELTHREADS];
input_queue tel_cmds[NUMTELTHREADS];

/* Synchronization objects for thread blocking routines */

/* pthread_mutex_t *tel_block_mutex[NUMTELTHREADS];
   pthread_cond_t *tel_block_free[NUMTELTHREADS];
*/
sem_t *tel_block[NUMTELTHREADS];
output_block tel_results[NUMTELTHREADS];

/*
int tel_write(input send_cmd, int level);
int tel_receive(int thread_num);
int tel_remove(long rm_pidtag, int flag);
int tel_flush(long pidtag);
int tel_blocked_state(int option);
*/
```

```
/*******************************************************************************
***/
/****************************** CCD module threads ******************************
***/
/*******************************************************************************
***/

/*
 * ccd function defines - use function and exposure defs from above
 * maintenance commands (open, bit, close, etc.) run on thread 0
 * exposure commands (single, sdots, focrun) run on thread 1
 * other commands run on thread 2
 */

/* CCD thread initialization routine and management objects */

void * CCD_main(void * arg);
pthread_t ccd_thread[NUMCCDTHREADS];
input_queue ccd_cmds[NUMCCDTHREADS];

sem_t *ccd_block[NUMCCDTHREADS];
output_block ccd_results[NUMCCDTHREADS];

pthread_mutex_t *shared_mutex;
```

```
/**************************************************************************
                          Lowell Observatory
                   CCD Acquisitions Software Package


Description:
   --------------
Module:          testccd.so
Called From:     loas.e
File Name:       $RCSfile: testccd.c,v $
Started:         07/06/98
Revision:        $Revision: 1.12.4.14 $
Last Revised:    $Date: 1999/05/24 02:31:55 $
By:              $Name:  $

Included in:     LOIS loadable module

Explanation:     This program is the test module for the client side of
                 the CCD acquisitions.

        Copyright 1998
, Lowell Observatory, All Rights Reserved

---------------------------------------------------------------------------
$Id: testccd.c,v 1.12.4.14 1999/05/24 02:31:55 agould Exp $
**************************************************************************/

/* Network Includes */
#include <netinet/in.h>

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#define _REENTRANT
#define _P __P
#endif

#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <syslog.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <strings.h>

/* Tcl/Tk Includes */
#include <tcl.h>
#include <tk.h>
#include <cdl.h>

#ifdef __SOLARIS_5x__
#include <mqueue.h>
#endif

/* CCD Camera Include */
```

```
#include <lois.h>
#include <LowellCCD.h>
#include <ccd_cmd.h>
#include <LowellTel.h>
#include <ModThrd.h>
#include <fitsio.h>


#define NUMKEYS 8

/* Structures for input queue holding ccd_struct information */
/************************************************************/
fitsfile *fits_ptr;

struct sched_param ccd_param;
ccd_struct nccd;
static ccd_struct ccd;

static long old_state[NUMCCDTHREADS]; /* long variables to store and restore blo
cked states
                                         of threads */
static pthread_mutex_t *actv_block[NUMCCDTHREADS]; /* Need these extra blocks to
 prevent
                                                      an activate from occurring
 while you're
                                                      removing commands.  Lockin
g the CCD
                                                      module thread queues is no
t a good solution
                                                      because it prevents comman
ds from being
                                                      written to the queues duri
ng a remove call,
                                                      and we may want to allow c
ommands to be
                                                      written during a remove */

struct  sockaddr_in server;
Tcl_Obj         *ccd_obj[NUM_ELEM], *ccd_obj_array, *ccd_index,
                *ccd_objcmd;
int GUI_state=1, wr_err;
int exp_number, test_exp=0;
ccd_vectors ccdvec;
struct telescope_vectors telvec;

tel_struct telescope; /* don't need this if focusrun is threadized */

info_struct info;

int vecsize, ccdsize, readpid, roi=0;
static char     ccdmsg[80];

#ifdef __SOLARIS_5x__
mqd_t cmd_queue;
#endif

cmd_struct ccdcmd;

static char ccdkeywords[NUMKEYS][9]={
        "DETECTOR\0",
        "CCDMODE\0",
        "GAIN\0",
        "RDNOISE\0",
        "TRIGGER\0",
        "NUMSUB\0",
        "SUBFNUM\0",
        "ORIGSEC\0",
};
static char ccdcomments[NUMKEYS][71]={
        "detector name\0",
        "CCD exposure mode\0",
        "gain, electrons per adu\0",
        "read noise, electrons\0",
        "trigger source(internal,etc..)\0",
        "Number of Subframes\0",
        "Subframe Number\0",
        "original size full frame\0",
};

Tcl_Obj  *ccdkeys[NUMKEYS*3];

/*
*
* Loadable Module Init called from Tcl/Tk load
*
*/
int Testccd_Init(Tcl_Interp *interp)
{

  char          ccd_cmd[80], *user_name=NULL;
  int           count=0, expnum=0, result, shm_fd;
  Tcl_Interp    *cmd_interp;

  /*
   *
   * Tcl/Tk Command definitions
   *
   */

  if ((cmd_interp = Tcl_GetSlave(interp, "command")) == NULL) {
    lois_log0("Error getting command interpreter!\n");
    return(TCL_ERROR);
  }
    /* Createing Generic Commands for all CCDs */
  for (count = 0 ; count < ccd_main_NUM_FUNCTIONS ; count++) {
    Tcl_CreateObjCommand(interp, ccd_main_cmds[count], ccd_main_func[count],
                         (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);

  }

  for (count = 0; count < ccd_slave_NUM_FUNCTIONS ; count++) {
    Tcl_CreateObjCommand(cmd_interp, ccd_slave_cmds[count], ccd_slave_func[count
],
                         (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  }

  /*
   *
   * Tcl/Tk Package Provide and Revision
   *
   */

  Tcl_PkgProvide(interp, "Testccd", "1.0");
  Tcl_PkgProvide(cmd_interp, "Testccd", "1.0");

  /*
   *
   * Start the Initialization of the testCCD system
   *
   */
  for (count =0 ; count < NUM_ELEM ; count++)
    ccd_obj[count]=Tcl_NewStringObj(ccd_cmd, sizeof(ccd_cmd));
```

```
  ccd_obj_array=Tcl_NewStringObj(ccd_cmd, sizeof(ccd_cmd));
  ccd_index=Tcl_NewStringObj(ccd_cmd, sizeof(ccd_cmd));
  ccd_objcmd=Tcl_NewStringObj(ccd_cmd, sizeof(ccd_cmd));
  ccdsize=sizeof(ccd);
  vecsize=sizeof(ccdvec);

  for (count =0 ; count < NUMKEYS*3 ; count++)
    ccdkeys[count]=Tcl_NewStringObj(ccd_cmd, sizeof(ccd_cmd));

#ifdef __SOLARIS_5x__
  cmd_queue = mq_open("/LOIS_queue", O_RDWR);
  if (cmd_queue == (mqd_t) -1) {
    lois_log0("CCD: Command Queue not opened");
    return(TCL_ERROR);
  }
#endif
  ccdcmd.priority=DEF_PRIORITY;
  ccdcmd.t_interval = 0;
  ccdcmd.t_times = 0;
  ccdcmd.t_execute = 0;
  lois_log4("CCD: TestCCD Module\0");

#ifdef __LINUX__

  /* shared memory initialization */

  if ((ccdvec.mem_fd=shmget(CCD_KEY, sizeof(ccdvec)+ccdsize, (SHM_R | SHM_W))) <
 0) {
    lois_log0("CCD: Cannot Open CCD vector shared memory");
    printf("Error no %d\n", errno);
    return(TCL_ERROR);
  }
  ccdvec.memory = shmat(ccdvec.mem_fd, 0, 0);

/*
*
* Initialize the shared memory mapping for the Telescope Vectors and
* Structures.
*
*/

  if ((telvec.mem_fd=shmget(TEL_KEY, sizeof(telescope)+sizeof(telvec),
                            (SHM_R | SHM_W))) < 0) {
    lois_log0("Cannot Open Telescope Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }
  telvec.memory = shmat(telvec.mem_fd, 0, 0);

  if ((info.mem_fd=shmget(INFO_KEY, sizeof(info), (SHM_R | SHM_W))) < 0) {
    lois_log0("Cannot Open Information Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }
  info.memory = shmat(info.mem_fd, 0, 0);

  if (ccdvec.memory == (void *) -1) {
    lois_log0("Memory Map failed for CCD Buffer");
    return(TCL_ERROR);
  }

  if (telvec.memory == (void *) -1) {
    lois_log0("Memory Map failed for Telescope Buffer");
    return(TCL_ERROR);
  }
  if (info.memory == (void *) -1) {
    lois_log0("Memory Map failed for Information Buffer");
    return(TCL_ERROR);
  }
#elif defined(__SOLARIS_5x__)

  if (( ccdvec.mem_fd=shm_open("/ccd", O_RDWR, S_IRWXU)) < 0 )
    {
      lois_log0("CCD: Cannot Open CCD vector shared memory");
      printf("Error no %d\n", errno);
      return(TCL_ERROR);
    }
/*
 *
 * Initialize the shared memory mapping for the Telescope Vectors and
 * Structures.
 *
 */

  if (( telvec.mem_fd=shm_open("/telescope", O_RDWR, S_IRWXU)) < 0 ) {
    lois_log0("Cannot Open Telescope Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }

  if (( info.mem_fd=shm_open("/information", O_RDWR, S_IRWXU)) < 0 ) {
    lois_log0("Cannot Open Information Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }



  ccdvec.memory=mmap(NULL, sizeof(ccdvec)+ccdsize, PROT_READ | PROT_WRITE,
                     MAP_SHARED, ccdvec.mem_fd, 0);

  telvec.memory=mmap(NULL, sizeof(telvec)+sizeof(telescope),
                     PROT_READ | PROT_WRITE,MAP_SHARED,
                     telvec.mem_fd, 0);

  info.memory=mmap(NULL, sizeof(info), PROT_READ | PROT_WRITE,
                   MAP_SHARED, info.mem_fd, 0);

  if (ccdvec.memory == NULL) {
    lois_log0("Memory Map failed for CCD Buffer");
    return(TCL_ERROR);
  }

  if (telvec.memory == NULL) {
    lois_log0("Memory Map failed for Telescope Buffer");
    return(TCL_ERROR);
  }

  if (info.memory == NULL) {
    lois_log0("Memory Map failed for Information Buffer");
    return(TCL_ERROR);
  }
#endif

  /* copy structure containing data for CCD network connection to
     location in shared memory - read existing information structure
     into local copy from shared memory */

  shm_write(ccdvec.memory, (void *)&ccdvec, vecsize);
```

```
  shm_write((char *)ccdvec.memory+vecsize, (void *)&ccd, ccdsize);
  shm_read((void *) &info, info.memory, sizeof(info));
  strcpy(info.ccdmod, "testccd\0");
  shm_write(info.memory, (void *) &info, sizeof(info));

  sprintf(ccd_cmd, "source %s/scripts/testccd.tcl\0", info.loishome);
  Tcl_SetStringObj(ccd_objcmd, ccd_cmd, -1);
  Tcl_EvalObj(interp, ccd_objcmd);
  /*      Tcl_SetStringObj(ccd_objcmd, "ccd_init\0", -1);
          Tcl_EvalObj(interp, ccd_objcmd); */

  /*****************************************/
  /** Following ifdef only in test module **/
  /*****************************************/

#ifdef __SOLARIS_5x__
  user_name = getenv("USER");
  if (user_name == NULL) {
    user_name = Tcl_GetVar2(interp, "env", "USER", TCL_GLOBAL_ONLY);
  }
#elif defined(__LINUX__)
  user_name = Tcl_GetVar2(interp, "env", "USER", TCL_GLOBAL_ONLY);
#endif

  if (Init_CCDThreadInfo(user_name) < 0) {
    lois_log0("Error in CCD thread initialization routine!\n");
    return(TCL_ERROR);
  }

  for (count = 0; count < NUMCCDTHREADS; count++) {
    actv_block[count] = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(actv_block[count], NULL);
  }

  return(TCL_OK);
}

int ccd_init (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int count, shm_fd,sub, imgsize;

  GUI_state=1;

     if (objc > 1) {
       for (count=1; count < objc; count++) {
         if (strcasecmp (objv[count]->bytes, "-nogui") == 0)
           GUI_state=0;
       }
     }

/*
*
* Load the NCCD Tcl script into the interpreter
*
*/

/*
*
* Init the NCCD GUI
*
*/
     if (GUI_state) ccd_gui (clientdata, interp, objc, objv);


     /* Setting the image buffer size to two times the size of a full
```

```
        unbinned frame size.
     */
#ifdef __SOLARIS_5x__
      if (( shm_fd=shm_open(IMAGE_BUFFER_NAME, O_RDWR,S_IRWXU)) < 0 ) {
        lois_log0("CCD: Cannot Open Shared Memory Buffer");
        printf("Error no %d\n", errno);
        return(TCL_ERROR);
      }

      /* Set the image buffer size to twice the default values for row and
         column */
     imgsize=IMAGE_BUFFER_SIZE*2*2;

     if (ftruncate(shm_fd, imgsize) < 0) {
                       perror("Cannot Set Image Shared Memory Size");
                       return(TCL_ERROR);
     }

     ccdvec.buffer=mmap(NULL, imgsize, PROT_READ | PROT_WRITE,
                               MAP_SHARED, shm_fd, 0);
#elif defined(__LINUX__)

     imgsize = IMAGE_BUFFER_SIZE;
     if ((shm_fd=shmget(IMAGE_KEY, imgsize, (IPC_CREAT | 0666))) < 0) {
       lois_log0("Cannot Open Image shared memory buffer");
       lois_log0("Error number: %d\n", errno);
       return(TCL_ERROR);
     }
     ccdvec.buffer = shmat(shm_fd, 0, 0);

     if (ccdvec.buffer == (void *) -1) {
       lois_log0("Memory Map failed for Image Buffer");
       return(TCL_ERROR);
     }

#endif
     shm_write(ccdvec.memory, (void *)&ccdvec, vecsize);
     shm_write((char *)ccdvec.memory+vecsize, (void *)&ccd, ccdsize);


     lois_log4("CCD: CCD Module Initialization Complete\0");
     if (ccd_open(clientdata, interp, objc, objv) == TCL_ERROR)
       return(TCL_ERROR);
     if (ccd_bit(clientdata, interp, objc, objv) == TCL_ERROR)
       return(TCL_ERROR);


     ccd_param.sched_priority=sched_get_priority_max(SCHED_FIFO);

     return(TCL_OK);

}

int ccd_gui (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  GUI_state=1;

  if (GUI_state) {
    Tcl_SetStringObj(ccd_objcmd, "nccd_main\0",-1);
    Tcl_EvalObj(interp, ccd_objcmd);
    GUI_state=1;
    lois_log5("CCD: GUI Opened");
    return(TCL_OK);
```

```
  } else {
    lois_log1("CCD: GUI Already Opened");
    return(TCL_ERROR);

  }

}

int ccd_delay (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
{
  input pipe_send;
  int count, num=0, priority = 1, to_be_blocked[NUMCCDTHREADS];
  char blocked_threads[10]="", tmp_str[5], ret_str[80];

  pipe_send.command = WAIT_CMD;
  pipe_send.numargs = 1;
  pipe_send.pidtag = get_curevalpidtag();

  if (objc > 2) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-2], NULL), "-prio") == 0) {
      if (Tcl_GetIntFromObj(interp, objv[objc-1], &priority) == TCL_ERROR) {
        Tcl_SetResult(interp, "Error getting priority for ccd_block command!", N
ULL);
        return(TCL_ERROR);
      }
      objc -= 2;
    }
  }

  if (objc > 1) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-1], NULL), "-subc") == 0) {
      pipe_send.pidtag *= -1;
      objc--;
    }
  }

  if (objc > 1) {
    for (count = 1; count < objc; count++) {
      if (strcmp(objv[count]->bytes, "status") == 0) {
        to_be_blocked[num] = 0;
        num++;
      } else {
        if (strcmp(objv[count]->bytes, "expose") == 0) {
          to_be_blocked[num] = 1;
          num++;
        } else {
          if (strcmp(objv[count]->bytes, "abort") == 0) {
            to_be_blocked[num] = 2;
            num++;
          }
        }
      }
    }
  } else {
    for (count = 0; count < NUMCCDTHREADS; count++) {
      to_be_blocked[num] = count;
      num++;
    }
  }

  for (count = 0; count < num; count++) {
    sprintf(pipe_send.args[0], "%d\0", to_be_blocked[count]);
    if (ccd_write(pipe_send, priority) < 0) {
      sprintf(interp->result, "Error sending delay command to CCD thread #%d\nEr
ror: %s\n",
            to_be_blocked[count], write_errs[wr_err]);
      return(TCL_ERROR);
    }
    sprintf(tmp_str, "%d ", count);
    strcat(blocked_threads, tmp_str);
  }

  sprintf(ret_str, "log_4 (CCD threads %shave been blocked)", blocked_threads);
  Tcl_Eval(interp, ret_str);
  sprintf(interp->result, "%d", pipe_send.pidtag);
  return(TCL_OK);
}

int ccd_activate (ClientData clientdata, Tcl_Interp *interp,
                      int objc, Tcl_Obj *CONST objv[])
{
  int count;
  char err_str[80];

  for (count = 0; count < NUMCCDTHREADS; count++) {
    if (sem_trywait(ccd_block[count]) != 0) {
      if (errno != EAGAIN) {
        sprintf(err_str, "Error reinitializing semaphore #%d in ccd_activate, er
rno: %d",
                count, errno);
        lois_log0(err_str);
      }
    }
  }

  for (count = 0; count < NUMCCDTHREADS; count++) {
    pthread_mutex_lock(ccd_cmds[count].mutex);
    if (pthread_mutex_trylock(actv_block[count]) < 0) {
      if (errno == EBUSY) {
        Tcl_SetResult(interp, "Cannot activate thread while removing command!",
NULL);
      } else {
        sprintf(err_str, "Error in trylock, ccd_activate. Error no %d\n", errno)
;

        Tcl_SetResult(interp, err_str, TCL_VOLATILE);
        pthread_mutex_unlock(actv_block[count]);
      }
      return(TCL_ERROR);
    }
    ccd_cmds[count].non_blocked |= 1; /* so that lowest priority queue is
                                         now unblocked */
    pthread_cond_signal(ccd_cmds[count].go); /* If thread was sleeping but comma
nds were
                                                in its queue */
    pthread_mutex_unlock(actv_block[count]);
    pthread_mutex_unlock(ccd_cmds[count].mutex);
  }

  sprintf(interp->result, "CCD threads have been reactivated\n");
  return(TCL_OK);
}

int ccd_update (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[])
{

  /* Updates CCD vectors after most commands */

  shm_write((void *) ccdvec.memory, (void *) &ccdvec, sizeof(ccdvec));
  return(TCL_OK);
}
```

```
/*
 *
 * Built In Test
 *
 */
int ccd_bit (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  input pipe_send;
  int priority = 1;
  static char err_str[80];

  switch (objc) {
  case 1:
    break;
  case 2:
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: ccd_bit [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1 * NUMPRIORITIES;
    break;
  case 3:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "Usage: ccd_bit [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority in ccd_bit!", NULL);
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(err_str, "Usage: ccd_bit [%s]", WAITOPT);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = BIT;
  pipe_send.numargs = 0;
  pipe_send.pidtag = get_curevalpidtag();
  sprintf(pipe_send.args[0], "");

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending ccd_bit command\nError: %s\n",
            write_errs[wr_err]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);

}

int ccd_open (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  input pipe_send;
  char *tmp_str;
  static char err_str[80];
  int priority = 1;

  switch (objc) {
  case 1:
    break;
  case 2:
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: ccd_open [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 3:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "Usage: ccd_open [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority in ccd_open!", NULL);
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(err_str, "Usage: ccd_open [%s]", WAITOPT);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = OPEN;
  pipe_send.numargs = 0;
  pipe_send.pidtag = get_curevalpidtag();

  /************************************************************/
  /* Following if statement and sprintf ONLY IN TEST MODULE!!!!! */
  /************************************************************/

  /*
  if ((tmp_str = Tcl_GetVar2(interp, "env", "USER", TCL_GLOBAL_ONLY)) == NULL) {
    lois_log0("Could not get user information in open!\n");
    return(TCL_ERROR);
  }
  sprintf(pipe_send.args[0], "%s\0", tmp_str);
  */

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending ccd_open command\nError: %s\n", write_errs[w
r_err]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int ccd_close (ClientData clientdata, Tcl_Interp *interp,
```

```
                 int objc, Tcl_Obj *CONST objv[])
{
  input pipe_send;
  char *tmp_str;
  static char err_str[80];
  int priority = 1;

  switch (objc) {
  case 1:
    break;
  case 2:
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: ccd_close [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 3:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "Usage: ccd_close [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority in ccd_close!", NULL);
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(err_str, "Usage: ccd_close [%s]", WAITOPT);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = CLOSE;
  pipe_send.numargs = 0;
  pipe_send.pidtag = get_curevalpidtag();

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending ccd_close command\nError: %s\n", write_errs[
wr_err]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);

}

int ccd_single (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  input         pipe_send;
  int           mode=CCD_SINGLE, count, priority = 1;
  char          *cpos;
  static char   err_str[80];
  Tcl_Obj       *single_obj;
  ccd_struct    *share_ccd;
```

```
/*
* Getting and Parsing the Command Line options or from
* the GUI if no keywords. If GUI is not inplace use the
* defaults
*/

  if (GUI_state) CCD_getdata(Tcl_GetMaster(interp), mode);

  /* We know that the -prio flag or the -wait flag must come after
     the frame argument */

  if (objc > 1) {

    if (strcmp(Tcl_GetStringFromObj(objv[objc-2], NULL), "-prio") == 0) {
      if (Tcl_GetIntFromObj(interp, objv[objc-1], &priority) == TCL_ERROR) {
        Tcl_SetResult(interp, "Error getting priority value!", NULL);
        return(TCL_ERROR);
      }
    } else {
      if (strcmp(Tcl_GetStringFromObj(objv[objc-1], NULL), WAITOPT) == 0) {
        priority = -1;
      }
    }

    /* Now get the frame argument, if there is one */

    for (count=1; count < objc; count++) {

      /* Get the Frame type */
      if ((cpos=strstr(Tcl_GetStringFromObj(objv[count],NULL),
                       "frame=")) != NULL) {
        switch(*(cpos+strlen("frame="))) {
        case 'B':
        case 'b':
          ccd.frame=CCD_BIAS;
          break;
        case 'O':
        case 'o':
          ccd.frame=CCD_OBJECT;
          break;
        case 'F':
        case 'f':
          ccd.frame=CCD_FLAT;
          break;
        case 'D':
        case 'd':
          ccd.frame=CCD_DARK;
          break;
        default:
          printf("Unknown frame: %c\n", *(cpos+strlen("frame=")));
          lois_log0("Error: Frame Type Unknown\n");
          return(TCL_ERROR);
        }
      }
    }
  }

  /* make sure that telescope and instrument threads are blocked */

  single_obj=Tcl_NewStringObj(err_str, sizeof(err_str));
  if (strcmp(info.telmod, "none") != 0) {
    Tcl_SetStringObj(single_obj, "tel_block -subc\0", -1);
    if (Tcl_EvalObj(Tcl_GetMaster(interp), single_obj) == TCL_ERROR) {
      sprintf(err_str, "Error blocking telescope threads in SINGLE!\n");
      lois_log0(err_str);
```

```
      return(TCL_ERROR);
    }
  }
  if (strcmp(info.instmod, "none") != 0) {
    Tcl_SetStringObj(single_obj, "inst_block -subc\0", -1);
    if (Tcl_EvalObj(Tcl_GetMaster(interp), single_obj) == TCL_ERROR) {
      sprintf(err_str, "Error blocking instrument threads in SINGLE!\n");
      lois_log0(err_str);
      return(TCL_ERROR);
    }
  }

  pipe_send.command = CCD_SINGLE;
  pipe_send.numargs = 2;
  pipe_send.pidtag = get_curevalpidtag();

  if (test_exp) {
    sprintf(pipe_send.args[0], "TEST\0");
  } else {
    sprintf(pipe_send.args[0], "REAL\0");
  }
  share_ccd = (ccd_struct *) ckalloc(sizeof(ccd_struct));
  bcopy((void *) &ccd, (void *) share_ccd, sizeof(ccd_struct));
  sprintf(pipe_send.args[1], "%p\0", share_ccd);
  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending single command\nError: %s\n", write_errs[wr_
err]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int ccd_series (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  lois_log0("CCD: Series Command not yet implemented");
  return(TCL_ERROR);

}

int ccd_strips (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  lois_log0("CCD: Strip Command not yet implemented");
  return(TCL_ERROR);

}

int ccd_sdots (ClientData clientdata, Tcl_Interp *interp,
               int objc, Tcl_Obj *CONST objv[])
{

  input         pipe_send;
  int           mode=CCD_S_DOTS, priority = 1, dummy;
  static char   err_str[80];
  Tcl_Obj       *sdots_obj;
  ccd_struct    *share_ccd;
  switch(objc) {
  case 2:
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: sdots [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 3:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "Usage: sdots [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority value in sdots!", NULL);
      return(TCL_ERROR);
    }
    break;
  }

  /* make sure instrument and telescope threads are blocked */

  sdots_obj=Tcl_NewStringObj(err_str, sizeof(err_str));
  if (strcmp(info.telmod, "none") != 0) {
    Tcl_SetStringObj(sdots_obj, "tel_block -subc\0", -1);
    if (Tcl_EvalObj(Tcl_GetMaster(interp), sdots_obj) == TCL_ERROR) {
      sprintf(err_str, "Error blocking telescope threads in SINGLE!\n");
      lois_log0(err_str);
      return(TCL_ERROR);
    }
  }
  if (strcmp(info.instmod, "none") != 0) {
    Tcl_SetStringObj(sdots_obj, "inst_block -subc\0", -1);
    if (Tcl_EvalObj(Tcl_GetMaster(interp), sdots_obj) == TCL_ERROR) {
      sprintf(err_str, "Error blocking instrument threads in SINGLE!\n");
      lois_log0(err_str);
      return(TCL_ERROR);
    }
  }

  if (GUI_state) CCD_getdata(Tcl_GetMaster(interp), mode);

  /* Commented out because for now, we're fixing the number of shifts
     and rows per shift in the test module */

  Tcl_SetStringObj(ccd_index, "SDots", -1);
  Tcl_SetStringObj(ccd_obj_array, "row_shift", -1);
  sdots_obj=Tcl_ObjGetVar2(Tcl_GetMaster(interp),ccd_obj_array, ccd_index,0);
  if (Tcl_GetIntFromObj(interp, sdots_obj, &dummy) == TCL_ERROR) {
    return(TCL_ERROR);
  }
  ccd.num_rshifts = (unsigned short) dummy;

  Tcl_SetStringObj(ccd_index, "SDots", -1);
  Tcl_SetStringObj(ccd_obj_array, "shift_width", -1);
  sdots_obj=Tcl_ObjGetVar2(Tcl_GetMaster(interp),ccd_obj_array, ccd_index,0);
  if (Tcl_GetIntFromObj(interp, sdots_obj, &dummy) == TCL_ERROR) {
    return(TCL_ERROR);
  }
  ccd.shift_width = (unsigned short) dummy;
  /* next two lines should only be in the test module */

  ccd.num_rshifts = 10;
  ccd.shift_width = 50;
```

```
  pipe_send.command = CCD_S_DOTS;
  pipe_send.numargs = 2;
  pipe_send.pidtag = get_curevalpidtag();

  if (test_exp) {
    sprintf(pipe_send.args[0], "TEST\0");
  } else {
    sprintf(pipe_send.args[0], "REAL\0");
  }

  share_ccd = (ccd_struct *) ckalloc(sizeof(ccd_struct));
  bcopy((void *) &ccd, (void *) share_ccd, sizeof(ccd_struct));
  sprintf(pipe_send.args[1], "%p\0", share_ccd);

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending sdots command\nError: %s\n", write_errs[wr_e
rr]);
    Tcl_SetResult(interp,err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);

}

int ccd_fdots (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  lois_log0("CCD: Dots Command not yet implemented");
  return(TCL_OK);

}

int ccd_focus (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  input            pipe_send;
  int              mode = CCD_FOCUS, priority = 1, nshifts, rows;
  char             *focstep;
  static char      err_str[80];
  ccd_struct       *share_ccd;
  Tcl_Obj          *focus_obj;

  switch(objc) {
  case 4:
    Tcl_GetIntFromObj(interp,objv[1],&rows);
    Tcl_GetIntFromObj(interp,objv[2],&nshifts);
    focstep=Tcl_GetStringFromObj(objv[3], NULL);
    break;
  case 5:
    Tcl_GetIntFromObj(interp,objv[1],&rows);
    Tcl_GetIntFromObj(interp,objv[2],&nshifts);
    focstep=Tcl_GetStringFromObj(objv[3], NULL);
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[4],  NULL)) != 0) {
      sprintf(err_str, "Usage: focrun rows nshifts focsteps [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority= -1;
    break;
  case 6:
    Tcl_GetIntFromObj(interp,objv[1],&rows);
    Tcl_GetIntFromObj(interp,objv[2],&nshifts);
    focstep=Tcl_GetStringFromObj(objv[3], NULL);
    if (strcmp(Tcl_GetStringFromObj(objv[4], NULL), "-prio") != 0) {
      Tcl_SetResult(interp, "Usage: focrun rows nshifts focsteps [-prio priority
]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[5], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority value in focrun!", NULL);
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(err_str, "Usage: focrun rows nshifts focsteps [%s]", WAITOPT);
    Tcl_SetResult(interp, err_str,  NULL);
    return(TCL_ERROR);
    break;
  }

  /* Next two lines should only be in the test module */

  rows = 50;
  nshifts = 10;

  /* make sure that telescope and instrument threads are blocked */

  focus_obj=Tcl_NewStringObj(err_str, sizeof(err_str));
  if (strcmp(info.telmod, "none") != 0) {
    Tcl_SetStringObj(focus_obj, "tel_block -subc\0", -1);
    if (Tcl_EvalObj(Tcl_GetMaster(interp), focus_obj) == TCL_ERROR) {
      sprintf(err_str, "Error blocking telescope threads in SINGLE!\n");
      lois_log0(err_str);
      return(TCL_ERROR);
    }
  }
  if (strcmp(info.instmod, "none") != 0) {
    Tcl_SetStringObj(focus_obj, "inst_block -subc\0", -1);
    if (Tcl_EvalObj(Tcl_GetMaster(interp), focus_obj) == TCL_ERROR) {
      sprintf(err_str, "Error blocking instrument threads in SINGLE!\n");
      lois_log0(err_str);
      return(TCL_ERROR);
    }
  }

  if (GUI_state) CCD_getdata(Tcl_GetMaster(interp), mode);

  ccd.mode=CCD_FOCUS;

  ccd.num_rshifts = nshifts;
  ccd.shift_width = rows;

  pipe_send.command = CCD_FOCUS;
  pipe_send.numargs = 3;
  pipe_send.pidtag = get_curevalpidtag();

  if (test_exp) {
    sprintf(pipe_send.args[0], "TEST\0");
  } else {
    sprintf(pipe_send.args[0], "REAL\0");
  }

  sprintf(pipe_send.args[1], "%s\0", focstep);
```

```
  share_ccd = (ccd_struct *) ckalloc(sizeof(ccd_struct));
  bcopy((void *) &ccd, (void *) share_ccd, sizeof(ccd_struct));
  sprintf(pipe_send.args[2], "%p\0", share_ccd);

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending focrun command\nError: %s\n", write_errs[wr_
err]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);

}


int ccd_go (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
        char            puts_str[40], ret_str[10];
        static char     *c_mode;

        /* Don't need to worry about parsing priority args here because they
           will be passed on */

        c_mode=Tcl_GetVar(Tcl_GetMaster(interp), "c_mode" , 0);

        shm_read((void *)&info,info.memory, sizeof(info));

        if(strcasecmp(c_mode, "Single") == 0) {
          if (ccd_single (clientdata, interp, objc, objv)
              == TCL_ERROR) return(TCL_ERROR);
        }
        if(strcasecmp(c_mode, "SDots") == 0) {
          if (ccd_sdots (clientdata, interp, objc, objv)
              == TCL_ERROR) return(TCL_ERROR);
        }
        if(strcasecmp(c_mode, "Focus") == 0) {
          if (ccd_focus (clientdata, interp, objc, objv)
              == TCL_ERROR) return(TCL_ERROR);
        }

        sprintf(ret_str, "%s\0", Tcl_GetStringResult(interp));
        sprintf(puts_str,"puts {Current Mode %s}", c_mode);
        Tcl_Eval(interp, puts_str);
        Tcl_SetResult(interp, ret_str, TCL_VOLATILE);
        return(TCL_OK);
}

int ccd_status (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

        sprintf(interp->result,
                        "CCD: Status Command not yet implemented");
        return(TCL_ERROR);


}
```

```
int ccd_test (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

        shm_read((void *)&info,info.memory, sizeof(info));

        if ( ccd_go(clientdata, interp,objc, objv) == TCL_ERROR) {
          test_exp = 0;
          return(TCL_ERROR);
        }
        test_exp = 0;
        return(TCL_OK);

}

int ccd_abort (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  input       pipe_send;
  char        *tmp_str;
  int         priority = NUMPRIORITIES;
  static char err_str[80];

  switch (objc) {
  case 1:
    break;
  case 2:
    if (strcasecmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: ccd_abort [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1 * NUMPRIORITIES;
    break;
  case 3:
    if (strcasecmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "Usage: ccd_abort [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority in abort!", NULL);
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(err_str, "Usage: ccd_abort [%s]", WAITOPT);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = ABORT;
  pipe_send.numargs = 0;
  pipe_send.pidtag = get_curevalpidtag();

  /****************************************************************/
  /* Following if statement and sprintf ONLY IN TEST MODULE!!!!!!!! */
  /****************************************************************/

  /*
  if ((tmp_str = Tcl_GetVar2(interp, "env", "USER", TCL_GLOBAL_ONLY)) == NULL) {
    lois_log0("Could not get user information!\n");
    return(TCL_ERROR);
  }
  sprintf(pipe_send.args[0], "%s\0", tmp_str);
  */
```

```

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending abort command\nError: %s\n", write_errs[wr_e
rr]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int ccd_setdim (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
{
  /* Command:
   *ccddim sub=<1-5> row=<Row Start>:<Row End> col=<Col Start>:<Col End>
   *            [oscan=<row overscan>:col overscan>] prescan=[prescan]
   */


  int          r1=-1, c1=-1, r2=-1, c2=-1, roscan=-1, coscan=-1, prescan=-1;
  char         *tmp_str, *spos_ptr, *epos_ptr, value[6];
  int          count, roi=-1, draw=0;

  /*
   * Parsing the arguments for the command
   */

  if (objc > 1) {
    for (count=1; count < objc; count++) {
        tmp_str=Tcl_GetStringFromObj(objv[count], NULL);
        switch(tmp_str[0]) {
        case 's':
        case 'S':
            if (strncasecmp(tmp_str, "sub", 3) != 0) {
              sprintf(interp->result, "SETDIM: SUB= Key not enter correctly.");
              return(TCL_ERROR);
            }
            spos_ptr=strstr(tmp_str,"=");
            roi=atoi(strcpy(value,spos_ptr+1));
            roi--;
            if(roi < 0 || roi > 4 ) {
              sprintf(interp->result, "SETDIM: SUBFRAME are between 1 and 5");
              return(TCL_ERROR);
            }
            break;
        case 'r':
        case 'R':
            if (strncasecmp(tmp_str, "row", 3) != 0) {
              sprintf(interp->result, "SETDIM: ROW= Key not enter correctly.");
              return(TCL_ERROR);
            }
            epos_ptr=strstr(tmp_str,":");
            r2=atoi(epos_ptr+1);
            spos_ptr=strstr(tmp_str,"=");
            r1=atoi(strncpy(value,spos_ptr+1, epos_ptr-spos_ptr));
            break;
        case 'o':       /* roscan = Number of rows to read past the */
        case 'O':       /* physical dim. of the chip */
          if (strncasecmp(tmp_str, "osc", 3) != 0) {
              sprintf(interp->result,
                      "SETDIM: OSCAN= Key not enter correctly.");
              return(TCL_ERROR);
            }
            epos_ptr=strstr(tmp_str,":");
            coscan=atoi(epos_ptr+1);
            spos_ptr=strstr(tmp_str,"=");
            roscan=atoi(strncpy(value,spos_ptr+1, epos_ptr-spos_ptr));
            break;
        case 'c':
        case 'C':
            if (strncasecmp(tmp_str, "col", 3) != 0) {
              sprintf(interp->result,
                      "SETDIM: COL= Key not enter correctly.");
              return(TCL_ERROR);
            }
            epos_ptr=strstr(tmp_str,":");
            c2=atoi(epos_ptr+1);
            spos_ptr=strstr(tmp_str,"=");
            c1=atoi(strncpy(value,spos_ptr+1, epos_ptr-spos_ptr));
        break;
        case 'p':
          if (strncasecmp(tmp_str, "pre", 3) != 0) {
              sprintf(interp->result,
                      "SETDIM: PRESCAN= Key not enter correctly.");
              return(TCL_ERROR);
            }
          spos_ptr=strstr(tmp_str,"=");
          prescan=atoi(spos_ptr+1);
          break;
        case '-':
          if (strncasecmp(tmp_str, "-draw", 5) != 0) {
            sprintf(interp->result,
                    "SETDIM: -DRAW Key not enter correctly.");
            return(TCL_ERROR);
          }
          draw=1;
          break;
        case '?':
          if (roi > -1) {
            sprintf(interp->result,
                 "Roi=%d Row=[%d:%d] Col=[%d:%d] Overscan=[%d:%d] Prescan=%d",
                 roi+1, ccd.r1[roi], ccd.r2[roi], ccd.c1[roi], ccd.c2[roi],
                 ccd.s_oscan, ccd.p_oscan, ccd.prescan);
          } else {
             sprintf(interp->result,
                 "Row=[1:%d] Col=[1:%d] Overscan=[%d:%d] Prescan=%d",
                 ccd.row, ccd.col, ccd.s_oscan, ccd.p_oscan, ccd.prescan);
          }
          break;
        default:
            sprintf(interp->result,"SETDIM: Unknown Keyword");
            return(TCL_ERROR);
            break;
        }
    }
} else {
       sprintf(interp->result,
             "Usage: setdim ? sub=<1-5> row=<Row Start>:<Row End> col=<Col Start>
:<Col End> oscan=<Row>:<Col> prescan=<Prescan> ");
    return(TCL_ERROR);
    }

  /* End of Command Parser */
```

```
  if (roi > -1 ) {
    if ( r1 > -1 && r2 > -1 && c1 > -1 && c2 > -1 ) {
      ccd.r1[roi]=r1;
      ccd.r2[roi]=r2;
      ccd.c1[roi]=c1;
      ccd.c2[roi]=c2;
      lois_log3("New Dim: Sub=%d r1=%d r2=%d c1=%d c2=%d coscan=%d roscan=%d",
            roi,r1,r2,c1,c2,coscan, roscan);
    } else {
      sprintf(interp->result,
              "Row=[%d:%d] Col=[%d:%d] Overscan=[%d:%d] Prescan=%d",
              ccd.r1[roi], ccd.r2[roi], ccd.c1[roi], ccd.c2[roi],
              ccd.s_oscan, ccd.p_oscan, ccd.prescan);
      return(TCL_OK);
    }
  }


  shm_write((char *)ccdvec.memory+vecsize, (void *)&ccd, ccdsize);

  if (draw) {
     sprintf(ccdcmd.command,"draw sub=%d shape=1 color=%d width=10",
             roi+1, roi+204);
     lois_send(&ccdcmd);
     printf("Drawing ROI\n");
  }
  return(TCL_OK);

}
int ccd_pause (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
        sprintf(interp->result,
                      "CCD: PAUSE Command not yet implemented");
        return(TCL_ERROR);

}

int ccd_header (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
        int     count;
        static char     keyvalue[70];


        for (count = 0 ; count < NUMKEYS ; count++) {
                Tcl_SetStringObj(ccdkeys[count*3], ccdkeywords[count],
                                        -1);
                Tcl_SetStringObj(ccdkeys[count*3+2], ccdcomments[count],
                                        -1);
        }

        /* Set the Dectector Name */
                Tcl_SetStringObj(ccdkeys[1], "test CCD 432x400", -1);
        /* Set the CCD Mode */
                switch(ccd.mode) {
                        case CCD_SINGLE:
                                sprintf(keyvalue,"S Single\0");
                                break;
                        case CCD_S_DOTS:
                                sprintf(keyvalue,"S Slow Dots\0");
                                break;
                        case CCD_FOCUS:
                                sprintf(keyvalue,"S Focus\0");
                                break;
                        default:
                                sprintf(keyvalue,"S Unknown Mode\0");
                                break;
                }

                Tcl_SetStringObj(ccdkeys[4], keyvalue, -1);
        /* Set the Gain Mode */
                sprintf(keyvalue,"F %f", ccd.gain);
                Tcl_SetStringObj(ccdkeys[7], keyvalue, -1);
        /* Set the RDNois Mode */
                Tcl_SetStringObj(ccdkeys[10], "F 0.00", -1);
        /* Set the Trigger Mode */
                Tcl_SetStringObj(ccdkeys[13], "S Internal", -1);
        /* Set the Number of SubFrames */
                Tcl_SetStringObj(ccdkeys[16], "I 0", -1);
        /* Set the SubFrame Number */
                Tcl_SetStringObj(ccdkeys[19], "I 0", -1);
        /* Set the OrigSec Number */
                Tcl_SetStringObj(ccdkeys[22], "S [1:856,1:800]", -1);

                Store_header(clientdata, interp, NUMKEYS*3, ccdkeys);
}

int ccd_readparam (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{


  shm_read((void *) &info, info.memory, sizeof(info));
  ccd_readparameters("testccd", &ccd, &info);
  printf("Returning from readparameters\n");
  shm_write(info.memory, (void *) &info, sizeof(info));

   sprintf(interp->result,"Leaving Function\n");
   return(TCL_OK);
}
/*********************************************************/
/****** Procedure to wait until an exposure is complete *****/
/*********************************************************/

int ccd_wait (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  input           pipe_send;
  static char     err_str[80];
  int             priority = -1 * NUMPRIORITIES;

  switch(objc) {
  case 1:
    break;
  case 2:
    if (strcasecmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: waitcam [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    break;
  case 3:
    if (strcasecmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "Usage: waitcam [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority in waitcam!", NULL);
      return(TCL_ERROR);
```

```
    }
    break;
  default:
    sprintf(err_str, "Usage: waitcam [%s]", WAITOPT);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = WAITCAM;
  pipe_send.numargs = 0;
  pipe_send.pidtag = get_curevalpidtag();

  if (ccd_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending waitcam command\nError: %s\n", write_errs[wr
_err]);
    Tcl_SetResult(interp, err_str, NULL);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

/*****************************************************************
*
*   Private Functions Not Exported As Script Commands
*
*****************************************************************/


int CCD_getdata(Tcl_Interp *interp, int mode)
{
  int count=0;
  char *temp, modestr[10], ccdcmd[80];

  /* fix number of rows, columns, overscan pixels, prescan pixels
     and binning in test module so image files make sense when read in */

  switch (mode) {

  case CCD_SINGLE:
    strcpy(modestr, "Single\0");
    break;
  case CCD_S_DOTS:
    strcpy(modestr, "SDots\0");
    break;
  case CCD_FOCUS:
    strcpy(modestr, "Focus\0");
    break;
  default:
    sprintf(interp->result, "Error: Mode not Implemented");
    return(TCL_ERROR);
  }

  sprintf(ccdcmd, "status_window %s", modestr);
  Tcl_Eval(interp, ccdcmd);

  Tcl_SetStringObj(ccd_index, ccd_mode[mode], -1);

  for (count=0 ; count < NUM_ELEM ; count++) {
    Tcl_SetStringObj(ccd_obj_array, ccd_array[count], -1);
    ccd_obj[count]=Tcl_ObjGetVar2(interp,ccd_obj_array, ccd_index,TCL_NAMESPACE_
ONLY);
  }

  ccd.exp_time=atoi(ccd_obj[0]->bytes);
  ccd.num_exp=atoi(ccd_obj[1]->bytes);
  ccd.def_row=800;
  ccd.def_col=800;
  ccd.prescan=56;
  ccd.overscan=0;
  ccd.row_bin=2;
  ccd.col_bin=2;
  ccd.frame=atoi(ccd_obj[11]->bytes);
  switch(ccd_obj[7]->internalRep.longValue) {
  case 1: /* 1/100 sec Case */
    ccd.exp_multi=1;
    break;
  case 2: /* Seconds Case...Convert to 1/100 sec */
    ccd.exp_multi=100;
    break;
  case 3: /* Mins Case....Convert to 1/100 sec */
    ccd.exp_multi=60*100;
    break;
  default:
    break;
  }
  sprintf(ccdmsg, "CCD: Exp String %s, Exp Time %d\n",
          ccd_obj[0]->bytes, ccd.exp_time);
  lois_log3(ccdmsg);
  strcpy(ccd.title, ccd_obj[8]->bytes);
  strcpy(ccd.comment[0], ccd_obj[9]->bytes);
  strcpy(ccd.comment[1], ccd_obj[10]->bytes);
  return(TCL_OK);

}

/******************************************************************************
******************/
/*************** commands to write to CCD module threads and get results *******
******************/
/******************************************************************************
******************/

int ccd_write(input send_cmd, int level) {

  int thread_num, get_results = 0, count;
  input *tmp_ptr;

  if ((level == 0) || (ABSOLUTE(level) > NUMPRIORITIES)) {
    wr_err = EILLPRIO; /* illegal priority level */
    return (-1);
  }

  switch(send_cmd.command) {
  case BIT:case OPEN:case CLOSE:
    thread_num = 0;
    break;
  case CCD_SINGLE:case CCD_S_DOTS:case CCD_FOCUS:case CCD_SERIES:
  case CCD_STRIPS:case CCD_F_DOTS:
    thread_num = 1;
    break;
  case ABORT:case WAITCAM:
    thread_num=2;
    break;
  case WAIT_CMD:
```

```
    thread_num = strtol(send_cmd.args[0], (char **) NULL, 10);
    if ((thread_num < 0) || (thread_num >= NUMCCDTHREADS)) {
      wr_err = EBADWAIT;
      return(-1);
    }
    send_cmd.numargs = 0;
    break;
  default:
    wr_err = ENOCMD; /* no defined command */
    return(-1);
    break;
  }

  pthread_mutex_lock(ccd_cmds[thread_num].mutex);

  if (level < 0) {
    get_results = 1;
    level = level * -1;
    send_cmd.command = -1 * send_cmd.command;
  }
  printf("Latest CCD process tag: %d, Thread: %d, Command: %d\n", send_cmd.pidta
g, thread_num,
         send_cmd.command);
  /*  get_results = 0; for now in test ccd module */
  level--;

  tmp_ptr = (input *) malloc(sizeof(input));
  send_cmd.next = NULL;
  shm_write((void *) tmp_ptr, (void *) &send_cmd, sizeof(send_cmd));
  if (ccd_cmds[thread_num].p_queue[level].head == NULL) {
    ccd_cmds[thread_num].p_queue[level].head = tmp_ptr;
    ccd_cmds[thread_num].p_queue[level].tail = tmp_ptr;
  } else {
    ccd_cmds[thread_num].p_queue[level].tail->next = tmp_ptr;
    ccd_cmds[thread_num].p_queue[level].tail = tmp_ptr;
  }

  if ((ccd_cmds[thread_num].non_blocked & ccd_cmds[thread_num].non_empty) == 0)
{
    pthread_cond_signal(ccd_cmds[thread_num].go);
  }

  ccd_cmds[thread_num].non_empty |= (1 << level);
  pthread_mutex_unlock(ccd_cmds[thread_num].mutex);

  /******************************************/
  /* Get results if command asked for them */
  /******************************************/

  if (get_results) {
    return(ccd_receive(thread_num));
  } else {
    return(0);
  }
}

int ccd_receive(int thread_num) {

  printf("\nGetting results (CCD %d)\n", thread_num);
  pthread_mutex_lock(ccd_results[thread_num].mutex);
  while (ccd_results[thread_num].value == RUN_WAIT) {
    printf("Waiting...(CCD %d)\n", thread_num);
    pthread_cond_wait(ccd_results[thread_num].go, ccd_results[thread_num].mutex)
;
  }
  if (ccd_results[thread_num].value == RUN_ERROR) {
    wr_err = EINTRNL; /* error executing internal part of command */
    ccd_results[thread_num].value = RUN_WAIT;
    pthread_mutex_unlock(ccd_results[thread_num].mutex);
    printf("Got results (CCD %d)\n", thread_num);
    return(-1);
  }

  if (ccd_results[thread_num].value == RUN_REMOVED) {
    wr_err = EREMOVED; /* command removed from thread queue */
    ccd_results[thread_num].value = RUN_WAIT;
    pthread_mutex_unlock(ccd_results[thread_num].mutex);
    printf("Got results (CCD %d)\n", thread_num);
    return(-1);
  }

  ccd_results[thread_num].value = RUN_WAIT;
  pthread_mutex_unlock(ccd_results[thread_num].mutex);
  printf("Got results (CCD %d)\n", thread_num);

  return(0);
}

/***************************************************************************
*/
/************ Commands for removing and adjusting module thread queues *********
*/
/***************************************************************************
*/

int ccd_remove(long rm_pidtag, int flag)
{

  int p_index, t_index, found = 0, blocks_this_queue, total = 0;
  input *previous, *traverse;

  if (rm_pidtag == 0) {
    printf("Illegal pidtag value!");
    return(-1);
  }
  if (rm_pidtag > 0) {
    for (t_index = 0; t_index < NUMCCDTHREADS; t_index++) {
      pthread_mutex_lock(ccd_cmds[t_index].mutex);
      blocks_this_queue = 0;
      for (p_index = 0; p_index < NUMPRIORITIES; p_index++) {
        traverse = ccd_cmds[t_index].p_queue[p_index].head;
        previous = NULL;
        while (traverse != NULL) {
          if (traverse->pidtag == rm_pidtag) {

            /* We found a command with the desired pidtag.  Remove it,
               but keep the rest of the list structure intact.  We could
               actually break out of the while here, since there should be
               only one command with the desired pidtag, but we keep going
               just in case (there's some case I haven't thought of) */

            /* Special case for block commands, because a block issues multiple
               inputs to the queue */

            if (traverse->command != WAIT_CMD) {
              found++;
            } else {
              blocks_this_queue++;
            }

            if (flag) {
```

```
              if (traverse->command < 0) {
                pthread_mutex_lock(ccd_results[t_index].mutex);
                ccd_results[t_index].value = RUN_REMOVED;
                pthread_cond_signal(ccd_results[t_index].go);
                pthread_mutex_unlock(ccd_results[t_index].mutex);
              }

              if (previous == NULL) {
                if ((ccd_cmds[t_index].p_queue[p_index].head = traverse->next) =
= NULL) {

                  ccd_cmds[t_index].non_empty &= ~(1 << p_index);
                }

                ckfree(traverse);
                traverse = ccd_cmds[t_index].p_queue[p_index].head;

              } else {
                ckfree(previous->next);
                previous->next = traverse->next;
                traverse = traverse->next;
              }
            } else {
              previous = traverse;
              traverse = traverse->next;
            }

          } else {
            previous = traverse;
            traverse = traverse->next;
          }
        }
      }

      /* See if the currently running command is a block.  If not, check to see
that the
         currently running command doesn't have the same process tag as the proc
ess tag
         we're trying to remove (if it does, then you can't remove it, since it'
s being executed!)
         If the currently running command is a block, check its process tag.  If
 the
         process tag of the current block is the one we're trying to remove, it'
s too late
          (we can't remove it) so return 0 */

      if (ccd_cmds[t_index].curr_run.command != WAIT_CMD) {
        if (found && (rm_pidtag == ccd_cmds[t_index].curr_run.pidtag)) {
          found++;
        }
      } else {
        if (rm_pidtag == ccd_cmds[t_index].curr_run.pidtag) {
          if (blocks_this_queue) {
            blocks_this_queue++;
          } else {
            pthread_mutex_unlock(ccd_cmds[t_index].mutex);
            return(0); /* The currently running process tag is the one we're try
ing to
                              remove, so the process tag couldn't be found */
          }
        }
      }
      pthread_mutex_unlock(ccd_cmds[t_index].mutex);
      if (blocks_this_queue > 1) {
        return(blocks_this_queue);
      }
      total += blocks_this_queue;
```

```
    }
    if ((found == 0) && (total != 0)) {
      found = 1; /* If we got here but didn't find any commands, then we are rem
oving blocks,
                    so send to the remove command that we should still look here
 for blocks */
    }
  } else {
    /* pidtag is negative, so it is associated with a block command sent by anot
her routine */

    for (t_index = 0; t_index < NUMCCDTHREADS; t_index++) {
      pthread_mutex_lock(ccd_cmds[t_index].mutex);
      traverse = ccd_cmds[t_index].p_queue[0].head; /* use priority of zero beca
use
                                                       that is where blocks are
going */
      previous = NULL;
      while (traverse != NULL) {
        if (traverse->pidtag == rm_pidtag) {
          if (traverse->command < 0) {
            pthread_mutex_lock(ccd_results[t_index].mutex);
            ccd_results[t_index].value = RUN_REMOVED;
            pthread_cond_signal(ccd_results[t_index].go);
            pthread_mutex_unlock(ccd_results[t_index].mutex);
          }

          if (previous == NULL) {
            if ((ccd_cmds[t_index].p_queue[0].head = traverse->next) == NULL) {
              ccd_cmds[t_index].non_empty &= ~(1 << 0);
            }

            ckfree(traverse);
            traverse = ccd_cmds[t_index].p_queue[0].head;

          } else {
            ckfree(previous->next);
            previous->next = traverse->next;
            traverse = traverse->next;
          }
          found++;
        } else {
          previous = traverse;
          traverse = traverse->next;
        }
      }

      if (!found && (rm_pidtag == ccd_cmds[t_index].curr_run.pidtag)) {

        /* If the block command wasn't in the queue for some thread, then a coup
le things
           could have happened:

           1. The block command could have gone through already.  The pidtag fie
ld
              of the curr_run structure will be the same as the rm_pidtag argume
nt passed
              to this function.  In this case, we want to reactivate the queue a
nd signal
              if necessary.  We know that the block we're looking for must be th
e one
              blocking the queue because of the pidtag value of the rm_pidtag ar
gument.
              Since the rm_pidtag argument is NEGATIVE, this part of the removal
 routine
              (to remove block subcommands) is only called if the main command (
```

```
the one
              that sent the blocks, i.e. an exposure) was found in the queue.  S
ince the
              main command is in the queue, it means that it MUST have written i
ts blocks
              already.

           2. The command we're trying to remove didn't send any block commands.
  In that
              case, the pidtag field of the curr_run structure would be differen
t than
              the rm_pidtag argument passed to this function.  We don't want to
reactivate
              the queue, nor do we want to signal in this case.

        */

        if (sem_trywait(ccd_block[t_index]) != 0) {
          if (errno != EAGAIN) {

            /* The semaphore doesn't have to be zero, someone could call an expo
sure, and
               before trying to remove it, reactivate a queue, which would wait
the semaphore */

            lois_log0("Error reinitializing semaphore #%d in ccd_remove, errno:
%d",
                      t_index, errno);
          }

        }

        old_state[t_index] |= 1; /* unblock lowest priority queue */
        pthread_cond_signal(ccd_cmds[t_index].go);  /* If thread was sleeping bu
t commands were
                                                       in its queue */
      }
      found = 0; /* for next thread... */
      pthread_mutex_unlock(ccd_cmds[t_index].mutex);
    }
  }
  return(found);
}

int ccd_flush(long pidtag)

{

  int count, p_index;
  input *traverse, *previous;

  for (count = 0; count < NUMCCDTHREADS; count++) {
    pthread_mutex_lock(ccd_cmds[count].mutex);
    for (p_index = 0; p_index < NUMPRIORITIES; p_index++) {
      traverse = ccd_cmds[count].p_queue[p_index].head;
      previous = NULL;
      while (traverse != NULL) {
        if ((pidtag == ABSOLUTE(traverse->pidtag)) || (pidtag == 0)) {
          if (traverse->command < 0) {
            pthread_mutex_lock(ccd_results[count].mutex);
            ccd_results[count].value = RUN_REMOVED;
            pthread_cond_signal(ccd_results[count].go);
            pthread_mutex_unlock(ccd_results[count].mutex);
          }
          if (previous == NULL) {
            if ((ccd_cmds[count].p_queue[p_index].head = traverse->next) == NULL
) {
              ccd_cmds[count].non_empty &= ~(1 << p_index);
            }
            ckfree(traverse);
            traverse = ccd_cmds[count].p_queue[p_index].head;
          } else {
            ckfree(previous->next);
            previous->next = traverse->next;
            traverse = traverse->next;
          }
        } else {
          previous = traverse;
          traverse = traverse->next;
        }
      }
    }
    if (ABSOLUTE(ccd_cmds[count].curr_run.pidtag) == pidtag) {
      /* If this thread is currently blocked... */
      old_state[count] |= 1;
    }
    pthread_mutex_unlock(ccd_cmds[count].mutex);
  }
  return(0);
}

int ccd_blocked_state(int option)

{
  int count;

  for (count = 0; count < NUMCCDTHREADS; count++) {
    pthread_mutex_lock(ccd_cmds[count].mutex);
    switch (option) {
    case SAVE_STATE:
      pthread_mutex_lock(actv_block[count]);
      old_state[count] = ccd_cmds[count].non_blocked;
      ccd_cmds[count].non_blocked = (~0 << (NUMPRIORITIES - 1));

      /* To block all user ccd module thread queues while a command is being
         removed */
      break;
    case RESTORE_STATE:
      ccd_cmds[count].non_blocked = old_state[count];
      pthread_cond_signal(ccd_cmds[count].go); /* If a queue with commands in it
                                                  just became unblocked...*/
      pthread_mutex_unlock(actv_block[count]);
      break;
    default:
      break;
    }
    pthread_mutex_unlock(ccd_cmds[count].mutex);
  }
  return 0;
}



/***************************************************************************
*****************/
/******************************** Stuff from focusrun routine *****************
*****************/
/***************************************************************************
*****************/

/*  ccdcmd.priority = -1*(MIN_PRIORITY+1); */ /* set command priority to be the
lowest possible
```

```
                                        value, so that when the block is execute
d,
                                        there are no other commands in graphics
queue */


/*
  sprintf(ccdcmd.command, "ccd_block\0");
  block_proc_tag = lois_send(&ccdcmd);
  printf("Before lois_receive ccd_block_id = %d\n", block_proc_tag);
  lois_receive(block_proc_tag, &block_res);
  printf("After lois_receive\n");
  if (block_res.rtn_status < 0) {
    sprintf(err_str, "Error blocking ccdescope threads in SINGLE!\n");
    lois_log0(err_str);
    return(TCL_ERROR);
  }
  printf("Before instrument part\n");
  sprintf(ccdcmd.command, "inst_block\0");
  block_proc_tag = lois_send(&ccdcmd);
  printf("Before instrument lois_receive, inst_block_id = %d\n", block_proc_tag)
;
  lois_receive(block_proc_tag, &block_res);
  printf("After instrument lois_receive\n");
  if (block_res.rtn_status != 0) {
    sprintf(err_str, "Error blocking instrument threads in SINGLE!\n");
    lois_log0(err_str);
    return(TCL_ERROR);
    }
    ccdcmd.priority = DEF_PRIORITY; */ /* reset the default command priority */

/* ------------------------------------------------------------------------

Change Log:

$Log: testccd.c,v $
Revision 1.12.4.14  1999/05/24 02:31:55  agould
Updating ccd_remove and ccd_flush to account for new abort_cmd feature

Revision 1.12.4.13  1999/05/19 00:01:47  agould
Added commands to help remove elements from a CCD module thread queue

Revision 1.12.4.12  1999/04/28 20:01:38  taylor
Commiting changes for Solaris compatibility.

Revision 1.12.4.11  1999/04/26 21:27:00  taylor
Merging changes to be compatible with Solaris.

Revision 1.12.4.10  1999/04/23 22:22:30  taylor
Fixed module load function to read Testccd_Init instead of Ccc_Init. trying
to be toooooo generic.

Revision 1.12.4.9  1999/04/23 21:46:50  taylor
Updating changes to make the CCD module a little more generic.

Revision 1.12.4.8  1999/04/23 17:57:26  taylor
Makeing changes to the ccd module to migrate to a more generic template
for each module. Function call are now named ccd_xxxx instead of testccd_xxx
to provide a more generic call. Functions and command definitions are now
defined in the ccd_cmd.h include file in the main include directory.

Revision 1.12.4.7  1999/04/22 20:32:30  agould
Merging changes from 1_1b (dynamic queues and multiple interpreters) to
branch rel_1_1a

Revision 1.12.4.5.2.1  1999/04/08 01:03:56  agould
```

```
Changing initialization for compatibility with dual interpreter system.

Revision 1.12.4.5  1999/03/16 19:23:18  agould
Merging new revisions to hans

Revision 1.12.4.5  1999/03/15 17:23:07  agould
Changed testccd and focusrun exposures to work in module thread
domain.

Revision 1.12.4.4  1999/03/04 22:10:06  agould
Added linux compatibility, threadized readout procedure, added feature
to abort so it asks user what to do with frame

Revision 1.12.2.1  1999/01/14 07:05:56  agould
Fixed CCD module so Single and Focus exposures display appropriate data.
Fixed cols = 800, rows = 800, binning = 2, prescan = 56, and for focrun,
numshifts = 10, rows/shift = 50, focus step = 50

Revision 1.12  1999/01/05 21:18:11  agould
Fixed ccdescope module so a focus exposure can be run.  REmoved printf
statements from focrun routine.

Revision 1.11  1998/12/11 00:40:26  agould
Have functional test_single and test_sdots routines.  test_focus does
not work currently because I haven't figured out a good semantic
for a ccdtest_fg (focus run) routine.  (I need a functional
ccdtest_fg, which I don't currently have).  Also, module ignores
display parameters.  Will ask BWT if I should change "log" calls
to prioritized log calls in newer version.

Revision 1.10  1998/12/02 22:02:45  agould
Trying test read chip (reads data from file).  File might be too short, as
program continues to read data until it has the proper number of rows,
and we might never get to that number.

Revision 1.9  1998/11/24 16:00:50  agould
Trying to get file read to work as CCD read.  I believe I am experiencing
scoping errors, I'm trying to fix them.

Revision 1.8  1998/11/23 17:04:12  agould
*** empty log message ***

Revision 1.7  1998/11/12 02:48:11  agould
Changed test_gui to ccd_gui

Revision 1.6  1998/11/10 02:37:35  agould
Put memory initialization parameters in fits.c (for now).  For some reason,
access to the shared info structure is causing segmentation faults.  This
is not happening with the ccd or telescope structures.

Revision 1.5  1998/11/09 17:36:41  agould
Added lseek command to test_open and test_bit (since we're doing
file transactions in test module).  Initialization routine works.  Need
to get dark, bias, flat frames for testing.

Revision 1.4  1998/10/29 03:03:37  agould
tried commenting out some stuff from other modules (like Store_header from
fits module).  Library loads into wish...need to configure library to deal
with network comm.

 * Revision 1.3  1998/10/28  23:48:53  agould
 * Changed all external CCD routines (didn't change from getdata on, but may
 * have to later.)  Also, assumed that connection to daemon computer is not
 * needed for test CCD module.
 *
 * Revision 1.2  1998/10/24  22:52:14  agould
```

```
 * Edited up to line 388 of testccd.c
 *
Revision 1.1  1998/10/24 22:14:36  agould
Initial revision

Revision 1.1  1998/10/24 18:52:02  agould
Initial revision

Revision 1.3  1998/09/25 18:53:05  taylor
Renamed NCCDacq.c to NCCD.c and set this as the module
for the Navy CCD system.

Revision 1.2  1998/07/14 16:40:08  taylor
Added scripting command and now have CCD_single working using fork
call to CCD_read. The CCDread function should be usable for all
the other modes but changes will need to be made to the nccd camera
module to handle the different modes with different exposure sequences.

Revision 1.1  1998/07/06 16:26:18  taylor
Initial revision


---------------------------------------------------------------------------*/
```

```
/*************************************************************************
                         Lowell Observatory
                  CCD Acquisitions Software Package


Description:
   --------------
Module:         testccd.so
Called From:    lois
File Name:      $RCSfile: ccdthread.c,v $
Started:        03/06/99
Revision:       $Revision: 1.1.4.10 $
Last Revised:   $Date: 1999/05/24 02:27:54 $
By:             $Name:  $

Included in:    LOIS loadable module

Explanation:    This program is the ccd module for the client side of
                the CCD acquisitions with David Koch's CCD camera.

        Copyright 1998
, Lowell Observatory, All Rights Reserved

-----------------------------------------------------------------------------
*/
#include <netinet/in.h>

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#define _REENTRANT
#define _P __P
#endif

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <termios.h>
#include <math.h>
#include <sys/time.h>
#ifdef __SOLARIS_5x__
#include <sys/socket.h>
#endif
/* Tcl/Tk Includes */
#include <tcl.h>
#include <tk.h>

/* CDL Library Include */
#include <cdl.h>

/* LOIS Library Include */
#include <lois.h>

/* Instrument Include */
#include <LowellCCD.h>
```

```
#include <LowellTel.h>
#include <ModThrd.h>
#include <fitsio.h>

int vecsize, ccdsize;
float          fstep;
fitsfile *fits_ptr;
ccd_info info_nccd;

/*pthread_cond_t dummy_cond = PTHREAD_COND_INITIALIZER;
  pthread_mutex_t dummy_mutex = PTHREAD_MUTEX_INITIALIZER; */
int exposure_aborted = 0;
struct sockaddr_in server;
static ccd_struct tccd[NUM_MODES];
ccd_vectors ccdvec;
struct telescope_vectors telvec;
static tel_struct telescope; /* for focus position in focusrun */
info_struct info;
static info_struct tmp_info;

display_struct l_display;
cmd_struct ccdcmd;

typedef struct {
  int rtn_val;
  char rtn_msg[120];
} thr_rtn_struct;

thr_rtn_struct thr_status;
char ccdmsg[40], username[20];

static void reset_curr_run(int number);
void * ccd_read(void * arg);

pthread_t       read_thread;
pthread_mutex_t *exposure_lock; /* lock that waitcam tries to get */
int started_ro; /* flag for abort_exp to determine if readout had started */

int Init_CCDThreadInfo(char *user)
{
  int count, count2;
  char err_str[80], sem_name[12];

  vecsize = sizeof(ccdvec);
  ccdsize = sizeof(tccd[0]);

  for (count = 0; count < NUMCCDTHREADS; count++) {
    ccd_cmds[count].mutex= (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
    ccd_cmds[count].go = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
    pthread_mutex_init(ccd_cmds[count].mutex, NULL);
    pthread_cond_init(ccd_cmds[count].go, NULL);

    ccd_results[count].mutex = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_
t));
    ccd_results[count].go = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
    pthread_mutex_init(ccd_results[count].mutex, NULL);
    pthread_cond_init(ccd_results[count].go, NULL);
    ccd_results[count].value = RUN_WAIT;

    ccd_block[count] = (sem_t *) ckalloc(sizeof(sem_t));
    if (sem_init(ccd_block[count], 0, 0) < 0) {
      sprintf(err_str, "Error creating semaphore #%d, error no: %d\n", count, er
rno);
      lois_log0(err_str);
      return(-1);
    }
    for (count2 = 0; count2 < NUMPRIORITIES; count2++) {
      ccd_cmds[count].p_queue[count2].head = ccd_cmds[count].p_queue[count2].tai
l = NULL;
    }
    reset_curr_run(count); /* Pre-set "currently-running" command in each
                              thread to hold a pidtag of 0 and a command of 0
                              (so we won't select it during any search) */

  }

  for (count = 0; count < NUMCCDTHREADS; count++) {
    pthread_create(&ccd_thread[count], NULL, CCD_main, (void *) count);
  }

  shared_mutex = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
  pthread_mutex_init(shared_mutex, NULL);
  exposure_lock = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
  pthread_mutex_init(exposure_lock, NULL);

  ccdcmd.priority=DEF_PRIORITY;
  ccdcmd.t_interval=0;
  ccdcmd.t_times=0;
  ccdcmd.t_execute=0;

  sprintf(username, "%s\0", user);

  return(0);
}

void * CCD_main(void *arg)

{
  int number, count, level;
  int look_std_prio, result;
  lois_results stat_rslt;
  char stat_str[40];

  number = (int) arg;
  ccd_cmds[number].non_blocked = ~(~0 << NUMPRIORITIES);
  ccd_cmds[number].non_empty = 0;
  for ( ; ; ) {
    look_std_prio = 1;
    pthread_mutex_lock(ccd_cmds[number].mutex);

    while ((ccd_cmds[number].non_blocked & ccd_cmds[number].non_empty) == 0) {
      pthread_cond_wait(ccd_cmds[number].go, ccd_cmds[number].mutex);
    }
    for (level = NUMPRIORITIES-1; level>0; level--) {
      if (ccd_cmds[number].p_queue[level].head != NULL) {
        shm_write((void *) &ccd_cmds[number].curr_run, (void *) ccd_cmds[number]
.p_queue[level].head,
                  sizeof(ccd_cmds[number].curr_run));
        ckfree((void *) ccd_cmds[number].p_queue[level].head);
        if ((ccd_cmds[number].p_queue[level].head = ccd_cmds[number].curr_run.ne
xt) == NULL) {
          ccd_cmds[number].non_empty &= ~(1 << level);
        }
        look_std_prio = 0;
        break;
      }
    }
    if (look_std_prio & ccd_cmds[number].non_blocked) {
      shm_write((void *) &ccd_cmds[number].curr_run, (void *) ccd_cmds[number].p
_queue[0].head,
                sizeof(ccd_cmds[number].curr_run));
```

```
      ckfree((void *) ccd_cmds[number].p_queue[0].head);
      if ((ccd_cmds[number].p_queue[0].head = ccd_cmds[number].curr_run.next) ==
 NULL) {
        ccd_cmds[number].non_empty &= ~(1 << 0);
      }
    }

    pthread_mutex_unlock(ccd_cmds[number].mutex);
    if (ccd_cmds[number].curr_run.command == WAIT_CMD) {
      ccd_cmds[number].non_blocked &= ~(1 << 0);

      printf("CCD thread #%d halted!\n", number);


      number = WAIT_CASE;
    }
    switch(number) {
      /* dispatch correct function - ccd maintenance functions */
    case 0:
      switch(ABSOLUTE(ccd_cmds[number].curr_run.command)) {
      case BIT:
        result = bit_ccd(ccd_cmds[number].curr_run);
        break;
      case OPEN:
        result = open_ccd(ccd_cmds[number].curr_run);
        break;
      case CLOSE:
        result = close_ccd(ccd_cmds[number].curr_run);
        break;
      default:
        lois_log1("Error: Unknown function in thread 0\n");
        break;
      }
      break;
    case 1:
      /* dispatch correct function - exposure functions
         these will be more complex because they will create new
         threads for the readout */
      if (strcmp(info.telmod, "none") != 0) {
        for (count = 0; count < NUMTELTHREADS; count++) {
          sem_wait(tel_block[count]);
        }
      }
      if (strcmp(info.instmod, "none") != 0) {
        for (count = 0; count < NUMINSTTHREADS; count++) {
          sem_wait(inst_block[count]);
        }
      }
      if (!command_removed_flag) { /* flag set if command was removed
                                      while exposure was waiting on semaphores *
/
        exposure_aborted = 0;
        switch(ABSOLUTE(ccd_cmds[number].curr_run.command)) {
        case CCD_SINGLE:
          result = single_exp(ccd_cmds[number].curr_run);
          break;
        case CCD_SERIES:
          result = series_exp(ccd_cmds[number].curr_run);
          break;
        case CCD_STRIPS:
          result = strips_exp(ccd_cmds[number].curr_run);
          break;
        case CCD_S_DOTS:
          result = sdots_exp(ccd_cmds[number].curr_run);
          break;
        case CCD_F_DOTS:
          result = fdots_exp(ccd_cmds[number].curr_run);
          break;
        case CCD_FOCUS:
          result = focusrun(ccd_cmds[number].curr_run);
          break;
        default:
          lois_log1("Error: Unknown function in thread 1\n");
          break;
        }
        ccdcmd.priority = MAX_PRIORITY;
        if (strcmp(info.telmod, "none") != 0) {
          sprintf(ccdcmd.command, "tel_activate\0");
          lois_send(&ccdcmd);
        }
        if (strcmp(info.instmod, "none") != 0) {
          sprintf(ccdcmd.command, "inst_activate\0");
          lois_send(&ccdcmd);
        }
        ccdcmd.priority = DEF_PRIORITY;
        ccdvec.exp_num = 0;
      } else {
        result = -1; /* Return error if command removed...*/
      }
      command_removed_flag = 0;
      break;
    case 2:
      switch(ABSOLUTE(ccd_cmds[number].curr_run.command)) {
      case ABORT:
        result = abort_exp(ccd_cmds[number].curr_run);
        break;
      case WAITCAM:
        result = wait_exp(ccd_cmds[number].curr_run);
        break;
      default:
        lois_log1("Error: Unknown function in thread 2\n");
        break;
      }
      break;
    case WAIT_CASE:
      /* wait condition - do nothing */
      number = (int) arg;
      if (ccd_cmds[number].curr_run.command < 0) {
        ccd_cmds[number].curr_run.command *= -1;
      }
      /* Don't want to reset threads if they're blocked, otherwise we'd
         lose the pidtag of the command that is blocking the thread! */
      result = 0;
      break;
    default:
      lois_log1("Error: Unknown thread identification\n");
      break;
    }
    sprintf(ccdcmd.command, "ccd_update\0");
    lois_send(&ccdcmd);
    if (ccd_cmds[number].curr_run.command < 0) {
      pthread_mutex_lock(ccd_results[number].mutex);
      ccd_results[number].value = result;
      pthread_cond_signal(ccd_results[number].go);
      pthread_mutex_unlock(ccd_results[number].mutex);
    }
    if (ccd_cmds[number].curr_run.command != WAIT_CMD) {
      reset_curr_run(number);
    }
  }
}
```

```
int bit_ccd(input cmd_input) {

  int bytes, nbytes, mode=CCD_FUNCTIONS;
  ccd_struct nccd;

  if (cmd_input.numargs != 0) {
    lois_log1("Error: Incorrect number of arguments in CCD_BIT\n");
    return(-1);
  }

  printf("CCD Command: ccd_bit\n");

  if (ccdvec.ccd_fd == 0 ) {
    lois_log1("CCD: Error No Initialization Done. Please Run CCD_init");
    return(-1);
  }

  tccd[mode].func=CCD_TEST;
  bcopy((void *) &tccd[mode], (void *) &nccd, sizeof(nccd));
  bswap((char *)&nccd, sizeof(nccd));
  write(ccdvec.ccd_fd, (void *) &nccd, sizeof(nccd));
  if (lseek(ccdvec.ccd_fd, 0L, 0) < 0) {
    lois_log0("CCD: Test Module Network Write Error In Ccd_Bit!");
    return(-1);
  }

  bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd) );
  while ( bytes < sizeof(nccd)) {
    nbytes=read(ccdvec.ccd_fd, (char *) &nccd+bytes,sizeof(nccd)-bytes);
    bytes=nbytes+bytes;
  }

  bswap((char *)&nccd, sizeof(nccd));
  if (nccd.error == ECCD_NODEV) {
    lois_log0("CCD: ERROR! CCD Module Not Loaded!");
    return(-1);
  }

  lois_log4("CCD: Built In Testing Passed!");

  return(0);
}

int open_ccd(input cmd_input) {

  int    byte_count=0;
  char     filename[80];
  ccd_struct nccd;

  if (cmd_input.numargs != 0) {
    lois_log1("Error: Incorrect number of args in ccd_open!\n");
    return(-1);
  }
  /*
  if (strchr(cmd_input.args[0], ' ') != NULL) {
    lois_log1("Error: Badly formatted input string in ccd_open: %s\n", cmd_input
.args[0]);
    return(-1);
  }
  */
  printf("CCD Command: ccd_open\n");

/*
* Init the network socket - in the test module, this will just be a file
* descriptor
*/
```

```
  sprintf(filename, "%s/users/%s/ccd_text.txt", info.loishome, username);
  if ((ccdvec.ccd_fd = open(filename, O_RDWR | O_CREAT | O_NONBLOCK, 0666)) < 0)
    {
      switch (errno) {
      case EACCES:
        lois_log0("CCD: ERROR! No Permission To Create Socket");
        break;
      case EDQUOT:
        lois_log0("CCD: ERROR! No Space on Device");
        break;
      case EEXIST:
        lois_log0("CCD: ERROR! File already exists");
        break;
      case EMFILE:
        lois_log0("CCD: ERROR! File Descriptor Table is Full");
        break;
      case ENOMEM:
        lois_log0("CCD: ERROR! Not Enough Memory to Create Socket");
        break;
      case ENOSR:
        lois_log0("CCD: ERROR! Not Enough Streams Resources Available");
        break;
      default:
        lois_log0("CCD: ERROR! Other file creation error, error number %d", errn
o);
        break;

      }
      return(-1);
    }
  /*  printf("Open: Row: %d Col: %d\n", tccd[0].row, tccd[0].col); */
  shm_write((char *)ccdvec.memory+vecsize, (void *)&tccd[0], ccdsize);

  lois_log5("CCD: Connection Controller Is Active\n");
  bcopy((void *) &tccd[0], (void *) &nccd, ccdsize);
  nccd.mode=CCD_FUNCTIONS;
  nccd.func=CCD_OPEN;
  if (ccd_send(nccd) != 0) return(-1);
  lois_log4("CCD: CCD Camera Module Opened");
  return(0);
}

int close_ccd(input cmd_input) {

  ccd_struct nccd;

  if (cmd_input.numargs != 0) {
    lois_log1("Error: Incorrect number of args in ccd_close!\n");
    return(-1);
  }

  printf("CCD Command: ccd_close\n");

  /*
   * Close the CCD Connection
   */

  shm_read((void *) &nccd, (void *) &tccd[CCD_FUNCTIONS], ccdsize);
  nccd.mode=CCD_FUNCTIONS;
  nccd.func=CCD_CLOSE;

  if (ccd_send(nccd) != 0) return(-1);
  if (close(ccdvec.ccd_fd) < 0) {
    lois_log0("Error closing camera connection!");
    return(-1);
```

```
  }
  lois_log4("CCD: CCD Camera Module Closed");
  return(0);
}

int single_exp(input cmd_input) {

  int mode = CCD_SINGLE, result;
  thr_rtn_struct  *read_status;
  char            ccd_cmd[80], err_str[140];
  ccd_struct      *share_ccd;
  pthread_attr_t  read_attribs;

  if (cmd_input.numargs != 2) {
    lois_log1("Error: Incorrect number of args in SINGLE\n");
    return(-1);
  }

  sscanf(cmd_input.args[1], "%p", &share_ccd);
  shm_write((void *) &tccd[mode], (void *) share_ccd, sizeof(ccd_struct));
  shm_read((void *) &tmp_info, info.memory, sizeof(info));
  free(share_ccd);

  if (strcmp(cmd_input.args[0], "TEST") == 0) {
    tmp_info.test = ON;
  } else {
    if (strcmp(cmd_input.args[0], "REAL") == 0) {
      tmp_info.test = OFF;
    } else {
      sprintf(err_str, "Badly formatted input string in SINGLE!\nArg1 = %s", cmd
_input.args[0]);
      lois_log0(err_str);
      return(-1);
    }
  }
  shm_write(info.memory, (void *) &tmp_info, sizeof(info));

  printf("CCD Command: single %s\n", cmd_input.args[0]);

  /* sprintf(read_status->rtn_msg, "\0"); */
  tccd[mode].mode=CCD_SINGLE;

  /*
   * Read the CCD and Info shared memory blocks
   */
  shm_read((void *)&ccdvec, ccdvec.memory, sizeof(ccdvec));
  /*  shm_read((void *)&info,info.memory, sizeof(info)); */
  if (ccdvec.exp_num != 0) {
    lois_log1("CCD: Camera Exposure Already In Progress");
    return(-1);
  } else ccdvec.exp_num=tccd[mode].num_exp;

  sprintf(ccdcmd.command,"exp_proc Single %d\0", tccd[mode].num_exp);
  lois_send(&ccdcmd);

  if (tmp_info.test == ON) tccd[mode].num_exp=1;
  tccd[mode].col=(tccd[mode].prescan+tccd[mode].overscan
                +tccd[mode].def_col)/tccd[mode].col_bin;
  tccd[mode].row=tccd[mode].def_row/tccd[mode].row_bin;

/* Send the Log command */

  sprintf(ccd_cmd, "Number of Exposures %d", ccdvec.exp_num);
  lois_log3(ccd_cmd);
  /*  pthread_attr_init(&read_attribs);
  pthread_attr_setscope(&read_attribs, PTHREAD_SCOPE_SYSTEM); */
  if (pthread_create(&read_thread, NULL, ccd_read, &mode) < 0) {
    lois_log0("Error creating read thread in CCD single!");
    return(-1);
  }
  pthread_mutex_lock(exposure_lock);
  if (pthread_join(read_thread, (void *) &read_status) < 0) {
    lois_log0("Error reading return value from read thread in CCD single!");
    pthread_mutex_unlock(exposure_lock);
    pthread_detach(read_thread);
    sprintf(ccdcmd.command, "clean_exp\0");
    lois_send(&ccdcmd);
    return(-1);
  }

  pthread_mutex_unlock(exposure_lock);

  /* This outer branch is necessary because if the exposure is aborted, the read
     thread will return with an error status of PTHREAD_CANCELED, which is does
not
     fit the structure of the standard return value.  Thus, if we try to access
     members of the structure and the structure's value is PTHREAD_CANCELED, we'
ll
     get errors.  We do nothing because in the case of an abort, the abort routi
ne
     does the dirty work */

  if (read_status != PTHREAD_CANCELED) {
    if (read_status->rtn_val < 0) {
      sprintf(err_str, "Error in ccd_read, SINGLE: %s!\n", read_status->rtn_msg)
;
      lois_log0(err_str);
      result = -1;
    } else {

      if (read_status->rtn_val == 0) {
        lois_log2("CCD: Single Exposure Complete");
      } else {
        switch (read_status->rtn_val) {
        case 1:
          sprintf(err_str, "Error in ccd_read, SINGLE: %s.\n", read_status->rtn_
msg);
          lois_log1(err_str);
          break;
        default:
          sprintf(err_str, "Other nonfatal error in ccd_read, SINGLE: %s.\n",
                  read_status->rtn_msg);
          lois_log1(err_str);
          break;
        } /* end switch on read_status->rtn_val */
      }
      result = read_status->rtn_val;
    }
  } else {
    /* exposure aborted - reinitialize communications */
    lois_log1("CCD: Reinitializing CCD communications");
    sprintf(ccdcmd.command, "ccd_init");
    lois_send(&ccdcmd);
    lois_log0("CCD: Exposure Aborted!!");

    result = -1;
  }

  sprintf(ccdcmd.command, "clean_exp\0");
  lois_send(&ccdcmd);

  return(result);
```

```
}

int series_exp(input cmd_input) {

  lois_log2("Series command not yet implemented\n");
  return(0);
}

int strips_exp(input cmd_input) {

  lois_log2("Strips command not yet implemented\n");
  return(0);
}

int fdots_exp(input cmd_input) {

  lois_log2("Fdots command not yet implemented\n");
  return(0);
}

int sdots_exp(input cmd_input) {

  int mode = CCD_S_DOTS, result;
  thr_rtn_struct  *read_status;
  char             ccd_cmd[80], err_str[140];
  ccd_struct       *share_ccd;

  if (cmd_input.numargs != 2) {
    lois_log1("Error: Incorrect number of args in S_DOTS\n");
    return(-1);
  }

  sscanf(cmd_input.args[1], "%p", &share_ccd);
  shm_write((void *) &tccd[mode], (void *) share_ccd, sizeof(ccd_struct));
  shm_read((void *) &tmp_info, info.memory, sizeof(info));
  free(share_ccd);

  if (strcmp(cmd_input.args[0], "TEST") == 0) {
    tmp_info.test = ON;
  } else {
    if (strcmp(cmd_input.args[0], "REAL") == 0) {
      tmp_info.test = OFF;
    } else {
      sprintf(err_str, "Badly formatted input string in S_DOTS!\nArg1 = %s", cmd
_input.args[0]);
      lois_log0(err_str);
      return(-1);
    }
  }

  shm_write(info.memory, (void *) &tmp_info, sizeof(info));

  printf("CCD Command: sdots %s\n", cmd_input.args[0]);

  tccd[mode].mode=CCD_S_DOTS;

  /*
   * Read the CCD and Info shared memory blocks
   */
  shm_read((void *)&ccdvec, ccdvec.memory, sizeof(ccdvec));
  /*  shm_read((void *)&info,info.memory, sizeof(info)); */
  sprintf(ccdcmd.command,"exp_proc SDots %d\0", tccd[mode].num_exp);
  lois_send(&ccdcmd);
  if (ccdvec.exp_num != 0) {
    lois_log1("CCD: Camera Exposure Already In Progress");
    return(-1);
  } else ccdvec.exp_num=tccd[mode].num_exp;

  sprintf(err_str, "# shifts: %d; rows/shift: %d", tccd[mode].num_rshifts,
          tccd[mode].shift_width);
  sprintf(ccdcmd.command, "puts {\n%s\n}", err_str);
  lois_send(&ccdcmd);
  tccd[mode].func=CCD_OBJECT;
  tccd[mode].row=(tccd[mode].shift_width*tccd[mode].num_rshifts)/
    tccd[mode].row_bin;
  tccd[mode].col=(tccd[mode].prescan+tccd[mode].overscan
                  +tccd[mode].def_col)/tccd[mode].col_bin;

  if (pthread_create(&read_thread, NULL, ccd_read, &mode) < 0) {
    lois_log0("Error creating read thread in CCD sdots!");
    return(-1);
  }
  pthread_mutex_lock(exposure_lock);
  if (pthread_join(read_thread, (void *) &read_status) < 0) {
    lois_log0("Error reading return value from read thread in CCD sdots!");
    pthread_mutex_unlock(exposure_lock);
    pthread_detach(read_thread);
    sprintf(ccdcmd.command, "clean_exp\0");
    lois_send(&ccdcmd);
    return(-1);
  }
  pthread_mutex_unlock(exposure_lock);

  if (read_status != PTHREAD_CANCELED) {
    if (read_status->rtn_val < 0) {
      sprintf(err_str, "Error in ccd_read, SDOTS: %s!\n", read_status->rtn_msg);
      lois_log0(err_str);
      result = -1;
    } else {

      if (read_status->rtn_val == 0) {
        lois_log2("CCD: Slow Dots Exposure Complete");
      } else {
        switch (read_status->rtn_val) {
        case 1:
          sprintf(err_str, "Error in ccd_read, SDOTS: %s.\n", read_status->rtn_m
sg);
          lois_log1(err_str);
          break;
        default:
          sprintf(err_str, "Other nonfatal error in ccd_read, SDOTS: %s.\n",
                  read_status->rtn_msg);
          lois_log1(err_str);
          break;
        } /* end switch on read_status->rtn_val */
      }
      result = read_status->rtn_val;
    }
  } else {
    /* exposure aborted */
    result = -1;
  }

  sprintf(ccdcmd.command, "clean_exp\0");
  lois_send(&ccdcmd);

  return(result);

}

int focusrun(input cmd_input) {
```

```
  static char     foccmd[140];
  int             mode=CCD_FOCUS, result;
  thr_rtn_struct *read_status;
  ccd_struct      *share_ccd;

  if (cmd_input.numargs != 3) {
    lois_log1("Error: Incorrect number of args in FOCUSRUN\n");
    return(-1);
  }

  fstep = (float) strtod(cmd_input.args[1], (char **) NULL);
  if (errno == ERANGE) {
    sprintf(foccmd, "Badly formatted focus argument in FOCUSRUN!\nArg1 = %s; Arg
2 = %s",
            cmd_input.args[0], cmd_input.args[1]);
    lois_log0(foccmd);
    return(-1);
  }

  sscanf(cmd_input.args[2], "%p", &share_ccd);
  shm_write((void *) &tccd[mode], (void *) share_ccd, sizeof(ccd_struct));
  shm_read((void *) &tmp_info, info.memory, sizeof(info));
  free(share_ccd);

  shm_read((void *) &tmp_info, info.memory, sizeof(info));
  if (strcmp(cmd_input.args[0], "TEST") == 0) {
    tmp_info.test = ON;
  } else {
    if (strcmp(cmd_input.args[0], "REAL") == 0) {
      tmp_info.test = OFF;
    } else {
      sprintf(foccmd, "Badly formatted input string in FOCUSRUN!\nArg1 = %s; Arg
2 = %s",
              cmd_input.args[0], cmd_input.args[1]);
      lois_log0(foccmd);
      return(-1);
    }
  }

  shm_write(info.memory, (void *) &tmp_info, sizeof(info));

  printf("CCD Command: focusrun %f %s\n", fstep, cmd_input.args[0]);

  tccd[mode].mode = CCD_FOCUS;

  sprintf(foccmd, "# shifts: %d; rows/shift: %d; focus step: %d",
          tccd[mode].num_rshifts, tccd[mode].shift_width, (int) fstep);
  sprintf(ccdcmd.command, "puts {\n%s\n}", foccmd);
  lois_send(&ccdcmd);
  sprintf(ccdcmd.command, "exp_proc Focus %d\0", tccd[mode].num_exp);
  lois_send(&ccdcmd);

  sprintf(ccdmsg,"CCD: Mode %d Exposure Time %d", mode, tccd[mode].exp_time);
  lois_log3(ccdmsg);

  shm_read((void *)&ccdvec, ccdvec.memory, sizeof(ccdvec));
  shm_read((void *)&telescope, (char *)telvec.memory+sizeof(telvec), sizeof(tele
scope));
  tmp_info.focus=ON;
  shm_write(info.memory, (void *) &tmp_info, sizeof(info));

  if (ccdvec.exp_num != 0) {
    lois_log1("CCD: Camera Exposure Already In Progress");
    return(TCL_ERROR);
  } else ccdvec.exp_num=tccd[mode].num_exp;
  tccd[mode].col=(tccd[mode].prescan+tccd[mode].overscan
                  +tccd[mode].def_col)/tccd[mode].col_bin;
  tccd[mode].row=(tccd[mode].shift_width*tccd[mode].num_rshifts)/
    tccd[mode].row_bin;

/*
*Start the Exposure Sequence
*/

  if (pthread_create(&read_thread, NULL, ccd_read, &mode) < 0) {
    lois_log0("Error creating read thread in CCD focusrun!");
    return(-1);
  }
  pthread_mutex_lock(exposure_lock);
  if (pthread_join(read_thread, (void *) &read_status) < 0) {
    lois_log0("Error reading return value from read thread in CCD focusrun!");
    pthread_mutex_unlock(exposure_lock);
    pthread_detach(read_thread);
    sprintf(ccdcmd.command, "clean_exp\0");
    lois_send(&ccdcmd);
    return(-1);
  }
  pthread_mutex_unlock(exposure_lock);

  if (read_status != PTHREAD_CANCELED) {
    if (read_status->rtn_val < 0) {
      sprintf(foccmd, "Error in ccd_read, FOCUSRUN: %s!\n", read_status->rtn_msg
);
      lois_log0(foccmd);
      result = -1;
    } else {

      if (read_status->rtn_val == 0) {
        lois_log2("CCD: Focus Exposure Complete");
      } else {
        switch (read_status->rtn_val) {
        case 1:
          sprintf(foccmd, "Error in ccd_read, FOCUSRUN: %s.\n", read_status->rtn
_msg);
          lois_log1(foccmd);
          break;
        default:
          sprintf(foccmd, "Other nonfatal error in ccd_read, FOCUSRUN: %s.\n",
                  read_status->rtn_msg);
          lois_log1(foccmd);
          break;
        } /* end switch on read_status->rtn_val */
      }
      result = read_status->rtn_val;
    }
  } else {
    /* exposure aborted */
    result = -1;
  }

  sprintf(ccdcmd.command, "clean_exp\0");
  lois_send(&ccdcmd);

  return(result);
}

int abort_exp (input cmd_input) {

  int rmpid, newfd, mode=CCD_FUNCTIONS;
  lois_results rm_res;
  char  filename[80];
```

```
  ccd_struct nccd;

  if (cmd_input.numargs != 0) {
    lois_log1("Error: Incorrect number of arguments in ABORT!\n");
    return(-1);
  }
  /*
  if (strchr(cmd_input.args[0], ' ') != NULL) {
    lois_log1("Error: Badly formatted input in ABORT: %s\n", cmd_input.args[0]);
    return(-1);
  }
  */
  printf("CCD Command: abort\n");

  sprintf(filename, "%s/users/%s/ccd_text.other", info.loishome, username);
  if ((newfd = creat(filename, 0644)) < 0)
    {
      switch (errno) {
      case EACCES:
        lois_log0("CCD: ERROR in ABORT! No Permission To Create Socket");
        break;
      case EMFILE:
        lois_log0("CCD: ERROR in ABORT! File Descriptor Table is Full");
        break;
      case ENOMEM:
        lois_log0("CCD: ERROR in ABORT! Not Enough Memory to Create Socket");
        break;
      case ENOSR:
        lois_log0("CCD: ERROR in ABORT! Not Enough Streams Resources Available")
;
        break;

      }
      return(-1);
    }

  tccd[mode].func=CCD_ABORT;

/*
*Start the Send the Abort Command
*/

  bcopy((void *) &tccd[mode], (void *) &nccd, sizeof(nccd));
  bswap((char *)&nccd, CCD_DATASZ);
  write(newfd, (void * ) &nccd, sizeof(nccd));

  ccdvec.exp_num=0;
  close(newfd);
  /*
    if (pthread_cancel(read_thread) != 0) {
    lois_log0("ABORT: Error cancelling read thread!");
    return(-1);
    }
    ccdcmd.priority=MAX_PRIORITY;
  sprintf(ccdcmd.command, "clean_exp\0");
  lois_send(&ccdcmd); */
  exposure_aborted = 1;
  /*
  if (started_ro == 1) {
    ccdcmd.priority = -1 * MAX_PRIORITY;
    sprintf(ccdcmd.command, "st_opt");
    rmpid = lois_send(&ccdcmd);
    ccdcmd.priority = DEF_PRIORITY;
    lois_receive(rmpid, &rm_res);
    if (rm_res.rtn_status != 0) {
      lois_log0("ABORT: Error in st_opt!");
      return(-1);
    }
  } else {
    ccdcmd.priority = -1 * MAX_PRIORITY;
    printf("CCD expnum: %d\n", ccdvec.exp_num);
    sprintf(ccdcmd.command, "rmfile");
    rmpid = lois_send(&ccdcmd);
    ccdcmd.priority = DEF_PRIORITY;
    lois_receive(rmpid, &rm_res);
    if (rm_res.rtn_status != 0) {
      lois_log0("ABORT: Error in rmfile!");
      return(-1);
    }
  }
  */
  /*  printf("CCD expnum: %d\n", ccdvec.exp_num);
  lois_log1("CCD: Reinitializing CCD communications");
  sprintf(ccdcmd.command, "ccd_init");
  lois_send(&ccdcmd);
  lois_log0("CCD: Exposure Aborted!!");
  */
  return(0);
}

int wait_exp (input cmd_input) {

  int tmp_int;

  if (cmd_input.numargs != 0) {
    lois_log1("Error: Incorrect number of arguments in WAITCAM!\n");
    return(-1);
  }

  printf("CCD Command: waitcam\n");

  /*******************************************************************/
  /* The logical thing to do here is to call a pthread_join on the read */
  /* thread.  However, the routine that created the read thread also    */
  /* called pthread_join on the thread, so we can't do that.  Instead   */
  /* the exposure routines will lock a mutex when they create a thread  */
  /* and this routine will try to grab that lock.                       */
  /*******************************************************************/

  tmp_int = pthread_mutex_trylock(exposure_lock);

  if (tmp_int == 0) {
    pthread_mutex_unlock(exposure_lock);
    lois_log1("No exposure in progress");
    return(-1);
  } else {
    if (tmp_int == EBUSY) {
      pthread_mutex_lock(exposure_lock);
      pthread_mutex_unlock(exposure_lock);
      return(0);
    } else {
      lois_log0("Error trying to lock mutex in waitcam, error #%d", tmp_int);
      return(-1);
    }
  }

}

int ccd_socket() {

  char err_str[60];
```

```
/*
*
* Init the NCCD network socket
*
*/
  bzero((char *) &server, sizeof(server));

  server.sin_family       = AF_INET;
  server.sin_addr.s_addr  =inet_addr(CCD_HOST);
  server.sin_port         =htons(TCP_PORT);

  if ((ccdvec.ccd_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
      switch (errno) {
      case EACCES:
        lois_log0("CCD: ERROR! No Permission To Create Socket");
        break;
      case EMFILE:
        lois_log0("CCD: ERROR! File Descriptor Table is Full");
        break;
      case ENOMEM:
        lois_log0("CCD: ERROR! Not Enough Memory to Create Socket");
        break;
      case ENOSR:
        lois_log0("CCD: ERROR! Not Enough Streams Resources Available");
        break;
      default:
        sprintf(err_str, "Other error creating socket, error number %d", errno);
        lois_log0(err_str);
        break;
      }
      return(-1);
    }

  return(0);
}

int ccd_connect()
{

  char err_str[60];

/*
*
* Init the NCCD network connection
*
*/

  if (connect(ccdvec.ccd_fd, (struct sockaddr *) &server, sizeof(server)) < 0) {
    switch (errno) {
    case ETIMEDOUT:
      lois_log0("CCD:NCCD Connection Timed out! Check IC System");
      break;
    case ECONNREFUSED:
      lois_log0("CCD:Server Daemon is Not Running, Restart Daemon on IC");
      break;
    case EHOSTDOWN:
      lois_log0("CCD:IC Server is not running. Boot IC system");
      break;
    case EHOSTUNREACH:
      lois_log0("CCD:No Route to NCCD IC Server. Check IC Route Table");
      break;
    default:
      sprintf(err_str, "CCD: Unknown Connect Error No: %d", errno);
      lois_log0(err_str);
      break;
    }
    return(-1);
  }
  return(0);

}

int ccd_send (ccd_struct nccd)
{
  int byte_count;
  char err_str[80];

  bswap((char *) &nccd, CCD_DATASZ );
  byte_count=0;
  byte_count=write(ccdvec.ccd_fd, (void *) &nccd, sizeof(nccd));

  if (byte_count != sizeof(nccd)) {
    lois_log0("CCD: Network Write Error");
    return(-1);
  }
  if (lseek(ccdvec.ccd_fd, 0L, 0) < 0) {
    lois_log0("CCD: Test Module Network Write Error");
    return(-1);
  }

  byte_count=0;
  byte_count=read(ccdvec.ccd_fd, (void *) &nccd, sizeof(nccd));

  if (byte_count != sizeof(nccd)) {
    lois_log0("CCD: Network Write Error");
    return(-1);
  }
  bswap((char *)&nccd, sizeof(nccd));
  if (nccd.error !=0) {
    sprintf(err_str, "CCD: Camera ERROR! Errno no: %d", nccd.error);
    lois_log0(err_str);
    return(-1);
  }
  return(0);
}

/* CCD_Read cleanup routine (closes fits file, returns command priority to norma
l */

void exp_done() {
  /*
  int status, bytes, nbytes, rmpid, choice;
  ccd_struct nccd;
  lois_results rm_res;

  ccdcmd.priority = -1 * MAX_PRIORITY;
  sprintf(ccdcmd.command, "st_opt");
  rmpid = lois_send(&ccdcmd);
  lois_receive(rmpid, &rm_res);
  if (rm_res.rtn_status != 0) {
    lois_log0("ABORT: Error in st_opt!");
  } else {
    choice = atoi(rm_res.rtn_message);
    switch(choice) {
    case 0:
      printf("Result = 0\n");
      break;
    case 1:
      printf("Result = 1\n");
      break;
    case 2:
```

```
      printf("Result = 2\n");
      break;
    case 3:
      printf("Result = 3\n");
      break;
    default:
      printf("Result = other: %d\n", choice);
      break;
    }
  }
    */

  return;
  /*
    bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
    while ( bytes < sizeof(nccd)) {
      nbytes=read(ccdvec.ccd_fd, (char *) &nccd+bytes,
                  sizeof(nccd)-bytes);
      bytes=nbytes+bytes;
    }
    bswap((char *)&nccd, CCD_DATASZ);

    sprintf(ccdmsg,"CCD: Date:%s Time:%s", nccd.ut_date, nccd.start_time);
    lois_log3(ccdmsg);
    printf("%d %d\n", nccd.col, nccd.row);
    nccd.exp_time=nccd.exp_time*10;
    nccd.dark_time=nccd.dark_time*10;
    shm_write((char *)ccdvec.memory+vecsize, (void *)&nccd, ccdsize);
    */
}


void * ccd_read(void * arg)
{

  int    bytes=0, nimage, foc_count, filecount, tmp_time=0, anynull=0, mode;
  long   timestart, timediff, timetot, cur_elem;
  long  count, row_bytes, nbytes=0;
  int   get_bytes, disp_pid, rmpid, status=0, result, num_of_shifts = 1;
  struct timeval cur_timer;
  struct timespec delay_tspec;
  char filename[80], err_msg[80], imtype[20];
  ccd_struct nccd;
  lois_results disp_rslt, rm_result;
  input tel_cmd_send;

  mode = *(int *) arg;

  thr_status.rtn_val = -1;
  sprintf(thr_status.rtn_msg, "OK\0");

  /*  result=sched_setscheduler(getpid(),SCHED_FIFO, &ccd_param); */
  ccdvec.framepix=(tccd[mode].col*tccd[mode].row);
  shm_write(ccdvec.memory, (void *)&ccdvec, vecsize);
  shm_write((char *)ccdvec.memory+vecsize, (void *)&tccd[mode], ccdsize);
  sprintf(ccdmsg,"CCD: Row:%d Col:%d R_Bin:%d C_Bin:%d", tccd[mode].row,
          tccd[mode].col, tccd[mode].row_bin, tccd[mode].col_bin);
  lois_log3(ccdmsg);

  filecount = 0;
  switch (tccd[mode].frame) {
  case CCD_BIAS:
    tccd[mode].func=CCD_BIAS;
    tccd[mode].exp_time=0;
    sprintf(imtype, "bias.0\0");
    break;
  case CCD_OBJECT:
    tccd[mode].func=CCD_OBJECT;
    sprintf(imtype, "object.0\0");
      break;
  case CCD_FLAT:
    tccd[mode].func=CCD_FLAT;
    sprintf(imtype, "flat.0\0");
    break;
  case CCD_DARK:
    tccd[mode].func=CCD_DARK;
    sprintf(imtype, "dark.0\0");
    break;
  default:
    sprintf(thr_status.rtn_msg, "unknown frame type");
    pthread_exit((void *) &thr_status);
    break;
  }

  timetot=tccd[mode].exp_time*tccd[mode].exp_multi;


  /*
   *Start the Exposure Sequence
   */

  started_ro = 0;

  bcopy((void *) &tccd[mode], (void *) &nccd, sizeof(nccd));
  bswap((char *)&nccd, CCD_DATASZ);
  if (mode == CCD_FOCUS) {

    sprintf(ccdmsg,"CCD: Focus:%f focsteps:%f", telescope.focus, fstep);
    lois_log3(ccdmsg);
    sprintf(ccdmsg, "CCD: Starting Focus Run");
    lois_log3(ccdmsg);
    num_of_shifts = tccd[mode].num_rshifts;
  } else {

    /* if not a focusrun exposure, do the initial write to the daemon here */

    write(ccdvec.ccd_fd, (void * ) &nccd, sizeof(nccd));
    bswap((char *)&nccd, CCD_DATASZ);
  }
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL); /* to prevent cancellati
on until exposure is occurring
                                                         it is before the for
loop because after each readout,
                                                         the thread is set to
ignore cancellation requests */

  for (nimage=0 ; (unsigned int) nimage < tccd[mode].num_exp ; nimage++) {

    gettimeofday(&cur_timer, NULL);
    timestart=cur_timer.tv_sec;

    switch (mode) {
    case CCD_SINGLE:
      sprintf(filename, "%s/ccd/testccd/%s%d\0", info.loishome,imtype,filecount)
;
      printf("%s\n", filename);
      if (tccd[mode].frame == CCD_OBJECT) {
        if ((filecount += 1) == 5) filecount = 0;
      }
      break;
    case CCD_S_DOTS:
      sprintf(filename, "%s/ccd/testccd/focus_input.test\0", info.loishome);
```

```
      printf("%s\n", filename);
      break;
    case CCD_FOCUS:
      sprintf(filename, "%s/ccd/testccd/focus_input.test\0", info.loishome);
      printf("%s\n", filename);
      break;
    default:
      sprintf(thr_status.rtn_msg, "mode not implemented");
      pthread_exit((void *) &thr_status);
      break;
    }

    cur_elem = 0;

    if (mode != CCD_FOCUS) {
      sprintf(ccdcmd.command,"$status dchar proctag 0 end");
      lois_send(&ccdcmd);
      sprintf(ccdcmd.command,"$status dchar etag 0 end");
      lois_send(&ccdcmd);
      sprintf(ccdcmd.command,"$status insert etag 0 %d\0", nimage+1);
      lois_send(&ccdcmd);
      sprintf(ccdcmd.command,"$status insert proctag 0 {Exposing}\0");
      lois_send(&ccdcmd);
    }

    /*    pthread_testcancel(); */
    ccdcmd.priority = -1*DEF_PRIORITY;
    sprintf(ccdcmd.command, "create_file\0");
    disp_pid = lois_send(&ccdcmd);
    lois_receive(disp_pid, &disp_rslt);
    if (disp_rslt.rtn_status != 0) {
      sprintf(thr_status.rtn_msg, "Error in create_file");
      ccdcmd.priority = DEF_PRIORITY;
      pthread_exit((void *) &thr_status);
    }
    ccdcmd.priority = DEF_PRIORITY;

    if (mode == CCD_FOCUS) {
      write(ccdvec.ccd_fd, (void *) &nccd, sizeof(nccd));
      timetot = 0; /* because we don't want to time-expose the frame in a focusr
un */
      pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

      for (foc_count = 0; foc_count < num_of_shifts; foc_count++) {
        pthread_testcancel();
        if (exposure_aborted) {
          break;
        }
        sprintf(ccdcmd.command,"$status dchar proctag 0 end");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command,"$status dchar etag 0 end");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command,"$status insert proctag 0 {Exposing}\0");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "$status insert etag 0 %d\0", num_of_shifts-foc_
count);
        lois_send(&ccdcmd);

        /* Only in test module */

        if (lseek(ccdvec.ccd_fd, -1 * sizeof(nccd), SEEK_CUR) < 0) {
          sprintf(thr_status.rtn_msg, "CCD: Test Exposure Network Write Error");
          pthread_exit((void *) &thr_status);
        }

        /* End test module only part */
        bytes = read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
        while (bytes < sizeof(nccd)) {
          nbytes = read(ccdvec.ccd_fd, (char *) &nccd+bytes,
                        sizeof(nccd)-bytes);
          bytes = nbytes+bytes;
        }
        bswap((char *) &nccd, CCD_DATASZ);
        pthread_testcancel();
        sprintf(ccdcmd.command,"$status dchar proctag 0 end");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "$status insert proctag 0 {Moving}\0");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "log {Sending the Telescope Move Commands}");
        lois_send(&ccdcmd);

        /* Updating the telescope structure, this is unnecessary if we make the
           tel_status functions within focus_go wait for results.  If we take
           that approach, however, then we can't have the interpreter waiting
           for focus_go commands (must send them directly to pipe with tel_write
) */

        ccdcmd.priority = -1 * MAX_PRIORITY;
        if (strcmp(info.telmod, "none") != 0) {
          sprintf(ccdcmd.command, "tel_status\0");
          disp_pid = lois_send(&ccdcmd);
          lois_receive(disp_pid, &disp_rslt);
          if (disp_rslt.rtn_status != 0) {
            thr_status.rtn_val = -1;
            sprintf(thr_status.rtn_msg, "Error in tel_status!");
            pthread_exit((void *) &thr_status);
          }
        }
        ccdcmd.priority = DEF_PRIORITY;

        shm_read((void *)&telescope, (char *)telvec.memory+sizeof(telvec),
                 sizeof(telescope));

        /* here, we're sending the focus control commands directly to the telesc
ope
           pipe.  we could set up another routine that works like lois_send but
sends
           commands to the command queue.  This other way would be the desired i
mplementation
           if we expect to have telescope modules where the number of arguments
for focus_go
           will not be one */

        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        tel_cmd_send.command = FOC_GO;
        tel_cmd_send.numargs = 1;
        /*      tel_cmd_send.pidtag = ccd_cmds[1].curr_run.pidtag; */
        sprintf(tel_cmd_send.args[0], "%f\0", telescope.focus+fstep);
        if (strcmp(info.telmod, "none") != 0) {
          if (tel_write(tel_cmd_send, -3) < 0) {
            sprintf(thr_status.rtn_msg, "Error in focus_go\n");
            pthread_exit((void *) &thr_status);
          }
        }
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        /* Old way of sending command through main message queue

        sprintf(ccdcmd.command,"focus_go %f -prio -3\0", telescope.focus+fstep);

        ccdcmd.priority = -1 * DEF_PRIORITY;
```

```
        disp_pid = lois_send(&ccdcmd);
        lois_receive(disp_pid, &disp_rslt);
        if (disp_rslt.rtn_status != 0) {
          thr_status.rtn_val = -1;
          sprintf(thr_status.rtn_msg, "Error in focus_go!");
          pthread_exit((void *) &thr_status);
        }

        ccdcmd.priority = DEF_PRIORITY;
        */
        pthread_testcancel();
        bswap((char *)&nccd, CCD_DATASZ);

        /*  Send back to the Daemon that telescope has finished moving  */

        write(ccdvec.ccd_fd, (void *) &nccd, sizeof(nccd));
        bswap((char *)&nccd, CCD_DATASZ);
      }
      pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    }

    pthread_testcancel();
    /*      ccdvec.exp_num=0;
            shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec));
            sprintf(thr_status.rtn_msg, "Error in store_nfile");
            pthread_exit((void *) &thr_status); */


    shm_read((void *) &tmp_info,info.memory, sizeof(info));
    sprintf(ccdcmd.command,"$status dchar filetag 0 end");
    lois_send(&ccdcmd);


    /*    pthread_testcancel(); */
    if (tmp_info.test == OFF) {
      if (tmp_info.focus == OFF) {
        sprintf(l_display.fname, "%s.%03d",tmp_info.filename, tmp_info.imageno);
      } else strcpy(l_display.fname, "focus.fits");
    } else strcpy(l_display.fname, "test.fits");

    l_display.ny=tccd[mode].row;
    l_display.nx=tccd[mode].col;
    l_display.depth=16;

    sprintf(ccdcmd.command,"$status insert filetag 0 {%s}\0", l_display.fname);
    lois_send(&ccdcmd);
    sprintf(ccdmsg, "CCD: File=%s", l_display.fname);
    lois_log3(ccdmsg);

    if (strstr(tmp_info.telmod, "none") == NULL) {
      sprintf(ccdcmd.command,"tel_status\0");
      lois_send(&ccdcmd);
    }

    ccdvec.pix_read=0;
    sprintf(ccdmsg,"Image Count Number %d", nimage+1);
    lois_log3(ccdmsg);
    row_bytes=tccd[mode].col*2;
    tmp_time=0;
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
    pthread_cleanup_push(exp_done, NULL);

    if (timetot/100 >= 1) {
      do {
        if (exposure_aborted) {
          break;
        }
        if (gettimeofday(&cur_timer, NULL) < 0) {
          sprintf(thr_status.rtn_msg, "getting time of day");
          pthread_exit((void *) &thr_status);
        }
        timediff=cur_timer.tv_sec-timestart;
        if (timediff != tmp_time) {
          sprintf(ccdcmd.command,"$status dchar ttag 0 end\0");
          lois_send(&ccdcmd);
          sprintf(ccdcmd.command,"$status insert ttag 0 (%04d)\0",
                  timetot/100-timediff);
          lois_send(&ccdcmd);
          tmp_time=timediff;
          /* usleep(900000); */
          lois_sleep(900000000);
        }
      } while (timediff < timetot/100);
    }
    pthread_cleanup_pop(0);
    /*    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL); */
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    /* Testccd module only */

    fits_open_file(&fits_ptr, filename, READONLY, &status);
    if (status != 0) {
      sprintf(thr_status.rtn_msg, "fits_open_file, number %d", status);
      status = 0;
      fits_close_file(fits_ptr, &status);
      pthread_exit((void *) &thr_status);
    }

    /* End testccd module only part */

    bzero((void *) ccdvec.buffer, ccdvec.framepix*2);
    for (count = 0 ; (unsigned int) count < tccd[mode].row ; count++) {
      nbytes=0;
      started_ro = 1;
      /*      pthread_testcancel(); */
      fits_read_img_usht(fits_ptr, 0, cur_elem+1, (long) tccd[mode].col, 0,
                         (unsigned short *) ccdvec.buffer+ccdvec.pix_read,
                         &anynull, &status);
      if (status != 0) {
        if (status == END_OF_FILE) { /* reached end of input file */
#ifdef __LINUX__
            nbytes = fits_ptr->Fptr->filehandle - (long) ccdvec.pix_read;
            if (lseek(fits_ptr->Fptr->filehandle, 0L, SEEK_SET) != 0) {
              /*                lois_log0("ccd_read: Error in file seek EOF");
              sprintf(err_msg, "ccd_read: Error number: %d\0",errno);
              lois_log0(err_msg); */
              sprintf(thr_status.rtn_msg, "file seek EOF, error number %d", stat
us);
              status = 0;
              fits_close_file(fits_ptr, &status);
              pthread_exit((void *) &thr_status);
            }
#elif defined(__SOLARIS_5x__)
            nbytes = fits_ptr->Fptr->filehandle - (long) ccdvec.pix_read;
            if (fseek(fits_ptr->Fptr->filehandle, 0L, SEEK_SET) != 0) {
              lois_log0("NCCD_read: Error in file seek EOF");
              sprintf(err_msg, "NCCD_read: Error number: %d\0",errno);
              lois_log0(err_msg);
              status = 0; fits_close_file(fits_ptr, &status);
              pthread_exit((void *) &thr_status);
            }
```

```
#endif
              status = 0;
            } else { /* another error */
              /* lois_log0("ccd_read: Error in fits_read_data");
              sprintf(err_msg, "ccd_read: Error number: %d\nCount: %d\0",
              status, count); */
              sprintf(thr_status.rtn_msg, "fits read error number %d; Count: %d",
status, count);
              status = 0;
              fits_close_file(fits_ptr, &status);
              lois_log0(err_msg);
              pthread_exit((void *) &thr_status);
            }
          } else {
            nbytes = row_bytes;
          }

          if (count == 0 ) {
            sprintf(ccdcmd.command,"$status dchar proctag 0 end");
            lois_send(&ccdcmd);
            sprintf(ccdcmd.command,"$status insert proctag 0 (Reading)\0");
            lois_send(&ccdcmd);
          }
          /*

            if (mode != CCD_SINGLE) {
            if (fseek(fits_ptr->fileptr, 0L, SEEK_SET) != 0) {
              lois_log0("ccd_read: Error in file seek NON_SINGLE");
              sprintf(err_msg, "ccd_read: Error number: %d\0", errno);
              lois_log0(err_msg);
              fits_close_file(fits_ptr, &status);
              return(TCL_ERROR);
            }
          }
          */
          /* usleep(12000); */
          lois_sleep(12000000);
          /* simulate chip readout */
          ccdvec.pix_read += nbytes / 2;
          cur_elem = cur_elem + nbytes / 2;

          shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec));

        }

    /*    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL); */
    fits_close_file(fits_ptr, &status);
    if (status != 0) {
      /*      lois_log0("ccd_read: Error in fits_close_file");
      sprintf(err_msg, "ccd_read: Error number: %d\0", status);
      lois_log0(err_msg); */
      sprintf(thr_status.rtn_msg, "fits close file error number %d", status);
      pthread_exit((void *) &thr_status);
    }

/*
* Finished the Image Send an acknowlegment back to the Server
* Daemon.
*/
    sprintf(ccdmsg,"CCD: Sending Acknowlegment");
    lois_log5(ccdmsg);
    write(ccdvec.ccd_fd, "got", 3);
    sprintf(ccdmsg,"***Finished Image Number: %d***", nimage+1);
    lois_log1(ccdmsg);

    /* CCD test module ONLY! */
    if (lseek(ccdvec.ccd_fd, -1 * (sizeof(nccd)+3), SEEK_CUR) < 0) {
      sprintf(thr_status.rtn_msg, "CCD: Test Exposure Network Write Error");
      pthread_exit((void *) &thr_status);
    }

/*
* Display the Image and decrease the number of exp by one
*/
    /* Swap the bytes around so they make sense */

    /*    bswap((char *) ccdvec.buffer, ccdvec.framepix*2); */
    if (mode != CCD_FOCUS) {
      bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
      while ( bytes < sizeof(nccd)) {
        nbytes=read(ccdvec.ccd_fd, (char *) &nccd+bytes,
                    sizeof(nccd)-bytes);
        bytes=nbytes+bytes;
      }
      bswap((char *)&nccd, CCD_DATASZ);
    }

    sprintf(ccdmsg,"CCD: Date:%s Time:%s", nccd.ut_date, nccd.start_time);
    lois_log3(ccdmsg);

    nccd.exp_time=nccd.exp_time*10;
    nccd.dark_time=nccd.dark_time*10;
    shm_write((char *)ccdvec.memory+vecsize, (void *)&nccd, ccdsize);
    if (exposure_aborted) {
      lois_log1("CCD: Aborting Exposure!!!");
      ccdcmd.priority = -1*MAX_PRIORITY;
      sprintf(ccdcmd.command, "st_opt");
      rmpid = lois_send(&ccdcmd);
      lois_receive(rmpid, &rm_result);
      if (rm_result.rtn_status != 0) {
        lois_log0("Error in st_opt procedure");
      }
      pthread_exit(PTHREAD_CANCELED);
    }
    nccd.exp_time=nccd.exp_time/10; /* To handle the microsec */
    nccd.dark_time=nccd.dark_time/10;

    /*    if ((disp_pid=fork()) < 0) {
      lois_log0("CCD: Error cannot fork process\n");
    } else  if (disp_pid == 0 ) {
      Display_image (clientdata, interp, objc, objv);
      exit(0);
      } */
    ccdcmd.priority = -1*MAX_PRIORITY;
    sprintf(ccdcmd.command, "disp_image\0");
    disp_pid = lois_send(&ccdcmd);
    lois_receive(disp_pid, &disp_rslt);
    if (disp_rslt.rtn_status != 0) {
      thr_status.rtn_val = 1;
      sprintf(thr_status.rtn_msg, "Error in disp_image");
    }
    ccdcmd.priority = DEF_PRIORITY;
    sprintf(ccdcmd.command,"$status dchar proctag 0 end");
    lois_send(&ccdcmd);
    sprintf(ccdcmd.command,"$status insert proctag 0 {Storing}\0");
    lois_send(&ccdcmd);

    ccdcmd.priority = -1*MAX_PRIORITY;
    sprintf(ccdcmd.command, "store\0");
    disp_pid = lois_send(&ccdcmd);
    lois_receive(disp_pid, &disp_rslt);
```

```
    if (disp_rslt.rtn_status != 0) {
      thr_status.rtn_val = 1;
      sprintf(thr_status.rtn_msg, "Error in store");
    }
    ccdcmd.priority = DEF_PRIORITY;
    started_ro = 0;
      /*      printf("Error storing image!!\n");
              return(TCL_ERROR); */

    if (nccd.num_exp < 0 ) {
      sprintf(thr_status.rtn_msg, "Exposure Sequence Aborted!!! Illegal value of
 num_exp\n\a");
      pthread_exit((void *) &thr_status);
    }

    if (tccd[mode].frame == CCD_FLAT) {
      if (tmp_info.dome_flat == OFF ) {
        if (strstr(tmp_info.telmod, "none") == NULL) {

          /* In real modules, there needs to be a way for this procedure (read)
to wait
             until the internals of the rmove command are done, otherwise we mig
ht
             start exposing again before the telescope is done moving */

          /* sending rmove command directly to telescope pipe.  See comment abou
t sending focus_go
             commands during a focusrun exposure (line 1214 of this file) */

          tel_cmd_send.command = RMOVE;
          tel_cmd_send.numargs = 2;
          sprintf(tel_cmd_send.args[0], "120");
          sprintf(tel_cmd_send.args[1], "0");
          if (tel_write(tel_cmd_send, -3) < 0) {
            thr_status.rtn_val = 1;
            sprintf(thr_status.rtn_msg, "Error in rmove!");
          }

          /* Could also make tel_write external, and do a call to tel_send...thi
s raises many
             issues, do we want to have scripts and/or users calling prioritized
 commands, or
             should high priority commands be restricted to the internals of the
 modules?  Do
             we want this command going through the main message queue, where it
 will stall
             the interpreter?  Or should it be sent straight to the telescope mo
dule pipe
             without going through the main message queue? */

          /* Old way of sending command through main message queue

          sprintf(ccdcmd.command,"rmove -ra 120 -dec 0 -prio -3 \0");

          ccdcmd.priority = -1 * MAX_PRIORITY;
          disp_pid = lois_send(&ccdcmd);
          lois_receive(disp_pid, &disp_rslt);
          if (disp_rslt.rtn_status != 0) {
            thr_status.rtn_val = 1;
            sprintf(thr_status.rtn_msg, "Error in rmove!");
          }
          ccdcmd.priority = DEF_PRIORITY;
          */
        }
      }
    }
    ccdvec.exp_num--;

    sprintf(ccdmsg, "Number of Exposure Left: %d", ccdvec.exp_num);
    lois_log3(ccdmsg);
    shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec));
  }

  if (lseek(ccdvec.ccd_fd, 0L, 0) < 0) {
    sprintf(thr_status.rtn_msg, "CCD: Test Exposure Network Write Error");
    pthread_exit((void *) &thr_status);
  }

  /*
      for ALL modes (including focus)
      bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
  while ( bytes < sizeof(nccd)) {
    nbytes=read(ccdvec.ccd_fd, (char *) &nccd+bytes,
                sizeof(nccd)-bytes);
    bytes=nbytes+bytes;
  }
  bswap((char *)&nccd, CCD_DATASZ); */
  /* ccdvec.exp_num=0;
     shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec)); */
  /* ccd_param.sched_priority=sched_get_priority_max(SCHED_OTHER);
  result=sched_setscheduler(getpid(),SCHED_OTHER, &ccd_param); */
  if (mode == CCD_FOCUS) bytes = read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd
));
  shm_read((void *) &tmp_info, info.memory, sizeof(info));
  tmp_info.focus = OFF;
  shm_write(info.memory, (void *) &tmp_info, sizeof(info));
  lois_log0("CCD: Exposure Sequence is done!");
  if (thr_status.rtn_val < 0) thr_status.rtn_val = 0;
  pthread_exit((void *) &thr_status);
}

static void reset_curr_run(int number) {

  ccd_cmds[number].curr_run.pidtag = 0;
  ccd_cmds[number].curr_run.command = 0;
}


/*******************************************************************************
*******/
/* Code that used to be part of focusrun routine (now adopted to fit in read rou
tine) */
/*******************************************************************************
*******/

/*      sprintf(ccdmsg,"CCD: Focus:%f focsteps:%f", telescope.focus, fstep);
      lois_log3(ccdmsg);
      sprintf(ccdmsg, "CCD: Starting Focus Run");
      lois_log3(ccdmsg);

      if ( Store_nfile(clientdata, interp, objc, objv) == TCL_ERROR) {
        printf("Error in Store_nfile, ccd_focus\n");
        return(TCL_ERROR);
      }

      test module - don't send data over network connection to ensure tel
         is responding

      bcopy((void *) &tccd[mode], (void *) &nccd, sizeof(nccd));
      bswap((char *)&nccd, CCD_DATASZ);
```

```
      write(ccdvec.ccd_fd, (void * ) &nccd, sizeof(nccd));
      bswap((char *)&nccd, CCD_DATASZ);

      for (count = 0 ; count < nshifts ; count++) {

        sprintf(ccdcmd.command, "$status dchar proctag 0 end");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "$status dchar etag 0 end");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "$status insert etag 0 %d\0", nshifts-count);
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "$status insert proctag 0 {Exposing}\0");
        lois_send(&ccdcmd);

              bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
                while ( bytes < sizeof(nccd)) {
                nbytes=read(ccdvec.ccd_fd, (char *) &nccd+bytes,
                sizeof(nccd)-bytes);
                bytes=nbytes+bytes;
                }
                bswap((char *)&nccd, CCD_DATASZ);

         Send the telescope command to move the focus

        sprintf(ccdcmd.command,"$status dchar proctag 0 end");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "$status insert proctag 0 {Moving}\0");
        lois_send(&ccdcmd);
        sprintf(ccdcmd.command, "log {Sending the Telescope Move Commands}");
        lois_send(&ccdcmd);

        shm_read((void *)&telescope, (char *)telvec.memory+sizeof(telvec),
                 sizeof(telescope));

        sprintf(foccmd,"focus_go %f\0", telescope.focus+fstep);

        sprintf(ccdmsg,"CCD: Focus Command:%s", foccmd);
        lois_log5(ccdmsg);

        Tcl_SetStringObj(nccd_objcmd, foccmd,-1);

        if ( Tcl_EvalObj(interp, nccd_objcmd) == TCL_ERROR) {
          printf("Error in focus_go command!\n");
          exit(-1);
        }

        bswap((char *)&nccd, CCD_DATASZ);

        Send back to the Daemon that telescope has finished moving

        write(ccdvec.ccd_fd, (void *) &nccd, sizeof(nccd));
        bswap((char *)&nccd, CCD_DATASZ);
      }

      sprintf(imfile, "%s/ccd/testccd/focus_input.test\0", info.loishome);
      fits_open_file(&fits_ptr, imfile, READONLY, &status);
      if (status != 0) {
        lois_log0("Focrun: Error in fits_open_file");
        sprintf(errstr, "Focrun: Error number: %d\0", status);
        lois_log0(errstr);
        status = 0;
        fits_close_file(fits_ptr, &status);
        return(TCL_ERROR);
      }

      sprintf(ccdcmd.command, "$status dchar etag 0 end");
      lois_send(&ccdcmd);
      sprintf(ccdcmd.command, "$status dchar proctag 0 end");
      lois_send(&ccdcmd);
      sprintf(ccdcmd.command, "$status insert proctag 0 {Reading}\0");
      lois_send(&ccdcmd);

      ccdvec.pix_read=0;
      row_bytes=tccd[mode].col*2;
      for (count = 0 ; (unsigned int) count < tccd[mode].row ; count++)
        {

          nbytes=0;
          fits_read_img_usht(fits_ptr, 0, ccdvec.pix_read+1,
                             (long) tccd[mode].col, 0,
                             (unsigned short *) ccdvec.buffer+ccdvec.pix_read,
                             &anynull, &status);
          if (status != 0) {
            lois_log0("Focrun: Error in fits_read_data");
            sprintf(errstr, "Focrun: Error number: %d\0", status);
            lois_log0(errstr); status = 0;
            fits_close_file(fits_ptr, &status);
            return(TCL_ERROR);
          } else {
            nbytes = row_bytes;
          }
          usleep(12000);
          ccdvec.pix_read += nbytes / 2;
          shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec));
        }
      sprintf(ccdmsg, "CCD: Total Pixels Read %d", ccdvec.pix_read);
      lois_log5(ccdmsg);

      fits_close_file(fits_ptr, &status);
      if (status != 0) {
        lois_log0("Focrun: Error in fits_close_file");
        sprintf(errstr, "Focrun: Error number: %d\0", status);
        lois_log0(errstr);
        return(TCL_ERROR);
      }


        Finished the Image Send an acknowlegment back to the Server
        Daemon.


      sprintf(ccdmsg, "CCD: Sending Acknowlegment");
      lois_log5(ccdmsg);

      write(ccdvec.ccd_fd, "got", 3);


        Setting the Display Parameters


      strcpy(l_display.fname,tccd[mode].title);
      l_display.ny=tccd[mode].row;
      l_display.nx=tccd[mode].col;
      l_display.depth=16;


       Swaping the bytes around so they make sense


      bswap((char *) ccdvec.buffer, ccdvec.framepix*2);

      if ((disp_pid=fork()) < 0) {
```

```
            lois_log0("CCD: Error cannot fork process\n");
          } else  if (disp_pid == 0 ) {
            Display_image (clientdata, interp, objc, objv);
            exit(0);
          }

          sprintf(ccdcmd.command, "$status dchar proctag 0 end");
          lois_send(&ccdcmd);
          sprintf(ccdcmd.command, "$status insert proctag 0 {Storing}\0");
          lois_send(&ccdcmd);

          if (Store_image (clientdata, interp, objc, objv) == TCL_ERROR) {
            printf("Error in store image routine, ccd_focus.\n");
            return(TCL_ERROR);
          }

          commented out for test module

             bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
             while ( bytes < sizeof(nccd)) {
             nbytes=read(ccdvec.ccd_fd, (char *) &nccd+bytes,
             sizeof(nccd)-bytes);
             bytes=nbytes+bytes;
             }
             bswap((char *)&nccd, CCD_DATASZ);

          ccdvec.exp_num=0;
          shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec));
          ccd_param.sched_priority=sched_get_priority_max(SCHED_OTHER);
          bytes=read(ccdvec.ccd_fd, (char *) &nccd, sizeof(nccd));
          info.focus=OFF;
          shm_write(info.memory, (void *)&info, sizeof(info));
          sprintf(ccdmsg, "CCD:Focus Exposure Complete");
          lois_log1(ccdmsg);
          sprintf(ccdcmd.command, "clean_exp\0");
          lois_send(&ccdcmd);

          exit(0);
}
}

    return(0);
}
*/
```

```
/*************************************************************************
                        Lowell Observatory
                   CCD Acquisitions Software Package


Description:
   --------------
Module:           teltest.so
Called From:      loas.e
File Name:        $RCSfile: teltest.c,v $
Started:          07/15/98
Revision:         $Revision: 1.4.4.11 $
Last Revised:     $Date: 1999/05/24 02:30:57 $
By:               $Name:  $

Included in:      LOIS loadable module

Explanation:      This program is the client side of the 42 inch telescope
                  control. It start the GUI and initiates communications
                  with the move computer.

        Copyright 1998
, Lowell Observatory, All Rights Reserved

---------------------------------------------------------------------------
Change Log:

$Log: teltest.c,v $
Revision 1.4.4.11  1999/05/24 02:30:57  agould
Changed argument assignment to start from 0 (not 1) for consistency across modul
es. Updating tel_remove and tel_flush to account for new abort_cmd feature

Revision 1.4.4.10  1999/05/20 02:55:16  agould
Added functions to abort a relative move and to simulate the delay during a
relative move.

Revision 1.4.4.9  1999/05/18 23:59:20  agould
Added routines to help remove commands from a telescope module thread queue.

Revision 1.4.4.8  1999/05/15 22:12:37  agould
Changed lois_sleep to pthread_cond_timedwait

Revision 1.4.4.6  1999/04/29 16:50:33  agould
Changed decl. of info_struct to be non-static

Revision 1.4.4.5  1999/04/26 21:27:17  taylor
Merging changes to be compatible with Solaris.

Revision 1.4.4.4  1999/04/22 21:21:34  agould
merging changes from branch rel_1_1b for dual interpreter system and dynamic que
ues

Revision 1.4.4.3.2.2  1999/04/14 21:36:49  agould
Changed timer thread to account for tracking.

Revision 1.4.4.3.2.1  1999/04/08 01:22:51  agould
Configuring test telescope module for dual interpreter system.

Revision 1.4.4.3  1999/03/04 22:46:58  agould
Linux compatibility

Revision 1.4.4.1  1999/01/22 18:12:22  taylor
Utilizing the module thread domain.

Revision 1.4  1999/01/05 21:15:04  agould
Got working focus_ctl and focus_go commands, can now do focus exposure with
```

```
CCD module.  Still need to figure out how to simulate RP, SP and ER commands,
as well as putting in a routine to periodically update the local sidereal
time.

Revision 1.3  1998/12/22 15:57:25  agould
Adding new log commands and lois_send commands to module.  Also simulating
focus with a stepper motor that goes between -30000 and 30000.  Finished
updating routines through teltest_fc.

Revision 1.2  1998/11/14 03:34:08  agould
Have mostly working tel_rm procedure.  Only fails when dec goes from + to
-, and has a few calculation errors.

Revision 1.1  1998/11/12 02:31:48  agould
Initial revision

Revision 1.3  1998/11/12 01:45:32  agould
Added two lines to teltest_status so that focus is updated on the GUI.  Also
changed line 277 in hall.tcl so that GUI is updated on a coord move. (It
had not updated the window when tel_co was invoked through the GUI, only
if the command was entered at the command line.)  The code may have been
the way it was so that on a coord move, the position of the telescope could
be periodically updated, so an abort could occur.

Revision 1.2  1998/11/11 23:51:33  agould
Removed all telescope communication commands.  Need to figure out how to
simulate commands like RM, FG, etc that the MOVE system uses to move the
telescope.

Revision 1.1  1998/11/11 20:53:33  agould
Initial revision

Revision 1.1  1998/09/25 18:54:26  taylor
Initial revision

---------------------------------------------------------------------------
$Id: teltest.c,v 1.4.4.11 1999/05/24 02:30:57 agould Exp $
*************************************************************************/

/* Network Includes */
#include <netinet/in.h>

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#define _REENTRANT
#define _P __P
#endif

#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <syslog.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
```

```
#include <termios.h>
#include <math.h>
#include <sys/time.h>

/* Tcl/Tk Includes */
#include <tcl.h>
#include <tk.h>

/* CDL Library Include */
#include <cdl.h>

/* LOIS Include */
#include <lois.h>

/* Telescope Include */
#include <LowellTel.h>
#include <LowellCCD.h>
#include <ModThrd.h>

int GUI_state, foc_fd, wr_err;
int move_delay = ON, focus_delay = ON; /* Switches whether focus or move delays
                                          should be simulated */

static long old_state[NUMTELTHREADS];         /* Longs to store old blocked stat
e of threads */
static pthread_mutex_t *actv_block[NUMTELTHREADS]; /* Need these extra blocks to
 prevent
                                                     an activate from occurring
 while you're
                                                     removing commands.  Lockin
g the telescope
                                                     module thread queues is no
t a good solution
                                                     because it prevents comman
ds from being
                                                     written to the pipes durin
g a remove call,
                                                     and we may want to allow c
ommands to be
                                                     written during a remove */
pthread_t timer_thr;
void * time_routine(void * arg);
ccd_vectors ccdvec;
static ccd_struct ccd;
tel_struct telescope;
struct telescope_vectors telvec;
info_struct info;
time_t cur_time;
struct tm *cur_date;

static char telmsg[80];
cmd_struct telcmd;

#ifdef __SOLARIS_5x__
mqd_t cmd_queue;
#endif

const float test_lat = 42.359001;
const float test_long = -71.094034;

#define NUMKEYS 9

static char telkeywords[NUMKEYS][9]={
        "TELESCOP\0",
        "ST\0",
        "RA\0",
```

```
        "DEC\0",
        "EQUINOX\0",
        "EPOCH\0",
        "ZD\0",
        "AIRMASS\0",
        "FOCUS\0",
};
static char telcomments[NUMKEYS][71]={
        "Telescope name\0",
        "sideral time\0",
        "right ascension(hh:mm:ss)\0",
        "declination (dd:mm:ss)\0",
        "equinox of RA and DEC\0",
        "same as EQUINOX(for back compat.)\0",
        "zenith distance\0",
        "airmass\0",
        "telescope focus position\0",
};

Tcl_Obj         *tel_objcmd, *telkeys[NUMKEYS*3];

/*
*
* Script Command Functions Defined
*
*/
int teltest_init (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_delay (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_activate (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_status (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_track (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_stop (ClientData clientdata, Tcl_Interp *interp,
                  int objc, Tcl_Obj *CONST objv[]);
int teltest_co (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_rm (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_fc (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_fg (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_er (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_cl (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_fw (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_sp (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_rp (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_getfoc (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_locate (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_header (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int teltest_sims (ClientData clientdata, Tcl_Interp *interp,
                       int objc, Tcl_Obj *CONST objv[]);
/*
*
```

```
* Loadable Module Init called from Tcl/Tk load
*
*/
int Teltest_Init(Tcl_Interp *interp)
{
  int count;
  char *user_name=NULL;
  Tcl_Interp *cmd_interp;
/*
*
* Tcl/Tk Command definitions
*
*/
  if ((cmd_interp = Tcl_GetSlave(interp, "command")) == NULL) {
    lois_log0("Error getting command interpreter!\n");
    return(TCL_ERROR);
  }

Tcl_CreateObjCommand(interp, "tel_init", teltest_init,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(interp, "tel_block", teltest_delay,
                   (ClientData) (NULL), (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateObjCommand(interp, "tel_activate", teltest_activate,
                   (ClientData) (NULL), (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateObjCommand(interp, "tel_status", teltest_status,
                   (ClientData) (NULL), (Tcl_CmdDeleteProc *) NULL);
Tcl_CreateObjCommand(interp, "track", teltest_track,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(interp, "tel_stop", teltest_stop,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "move", teltest_co,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "rmove", teltest_rm,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "focus_ctl", teltest_fc,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "focus_go", teltest_fg,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "ephm_rate", teltest_er,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "c_lock", teltest_cl,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "filt", teltest_fw,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "strpos", teltest_sp,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "gopos", teltest_rp,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(interp, "getfoc", teltest_getfoc,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(interp, "tel_header", teltest_header,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(cmd_interp, "locate", teltest_locate,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
Tcl_CreateObjCommand(interp, "tel_simdelays", teltest_sims,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);


/*
*
* Tcl/Tk Package Provide and Revision
*
*/

Tcl_PkgProvide(interp, "Teltest", "1.0");
Tcl_PkgProvide(cmd_interp, "Teltest", "1.0");
/*
```

```
*
* Initialize the shared memory mapping for the Telescope Vectors and
* Structures.
*
*/

#ifdef __LINUX__
 if ((ccdvec.mem_fd=shmget(CCD_KEY, sizeof(ccdvec), (SHM_R|SHM_W))) < 0) {
   lois_log0("Cannot Open CCD Shared Memory Buffer");
   printf("Error no %d\n",errno);
   return(TCL_ERROR);
 }

 ccdvec.memory = shmat(ccdvec.mem_fd, 0, 0);

 if (( telvec.mem_fd=shmget(TEL_KEY, sizeof(telvec) + sizeof(telescope),
                           (SHM_R|SHM_W))) < 0) {
   lois_log0("Cannot Open Telescope Shared Memory Buffer");
   printf("Error no %d\n",errno);
   return(TCL_ERROR);
 }

 telvec.memory = shmat(telvec.mem_fd, 0, 0);

 if ((info.mem_fd = shmget(INFO_KEY, sizeof(info), (SHM_R|SHM_W))) < 0) {
   lois_log0("Cannot Open Information Shared Memory Buffer");
   printf("Error number %d\n", errno);
   return(TCL_ERROR);
 }

 info.memory = shmat(info.mem_fd, 0, 0);

 if (ccdvec.memory == (void *) -1) {
   lois_log0("Memory map failed for CCD buffer");
   return(TCL_ERROR);
 }

 if (telvec.memory == (void *) -1) {
   lois_log0("Memory map failed for telescope buffer");
   return(TCL_ERROR);
 }
 if (info.memory == (void *) -1) {
   lois_log0("Memory map failed for information buffer");
   return(TCL_ERROR);
 }
#elif defined(__SOLARIS_5x__)

  if (( ccdvec.mem_fd=shm_open("/ccd", O_RDWR,
                               S_IRWXU)) < 0 ) {
    lois_log0("Cannot Open CCD Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }

  if (( telvec.mem_fd=shm_open("/telescope", O_RDWR, S_IRWXU)) < 0 ) {
    lois_log0("Cannot Open Telescope Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }

  if ((info.mem_fd = shm_open("/information", O_RDWR, S_IRWXU)) < 0) {
    lois_log0("Cannot Open Information Shared Memory Buffer");
    printf("Error number %d\n", errno);
    return(TCL_ERROR);
  }
  telvec.memory=mmap(NULL, sizeof(telvec)+sizeof(telescope),
                     PROT_READ | PROT_WRITE,MAP_SHARED,
                     telvec.mem_fd, 0);

  ccdvec.memory=mmap(NULL, sizeof(ccdvec), PROT_READ | PROT_WRITE,
                     MAP_SHARED, ccdvec.mem_fd, 0);

  info.memory = mmap(NULL, sizeof(info), PROT_READ | PROT_WRITE,
                     MAP_SHARED, info.mem_fd, 0);

  if (ccdvec.memory == NULL) {
    lois_log0("Memory map failed for CCD buffer");
    return(TCL_ERROR);
  }

  if (telvec.memory == NULL) {
    lois_log0("Memory map failed for telescope buffer");
    return(TCL_ERROR);
  }

  if (info.memory == NULL) {
    lois_log0("Memory map failed for information buffer");
    return(TCL_ERROR);
  }
#endif

  shm_read((void *)&ccdvec, ccdvec.memory, sizeof(ccdvec));
  shm_read((void *)&ccd, (char *)ccdvec.memory+sizeof(ccdvec), sizeof(ccd));
  shm_read((void *)&telvec, telvec.memory, sizeof(telvec));
  shm_read((void *)&info, info.memory, sizeof(info));

  strcpy(info.telmod, "telescope test module\0");
  shm_write(info.memory, (void *)&info, sizeof(info));

  /*
   *
   * Set up the Default telescope Parameters
   *
   */
  sprintf(telescope.dec, "+00:00:00\0");
  sprintf(telescope.ra, "00:00:00.0\0");
  sprintf(telescope.epoch, "      \0");
  /*    sprintf(telescope.lst, "00:00:00\0");
        sprintf(telescope.date, "11/11/98\0");
        sprintf(telescope.ut, " 00:00:00\0");
        sprintf(telescope.airmass, "%.1s.%.2s\0", tel_string+32,tel_string+33);
        sprintf(telescope.dome, "%.3s%.1s\0", tel_string+36, tel_string+35);
        sprintf(telescope.epoch, "%.4s.%.2s\0", tel_string+39, tel_string+43);
  */
  strcpy(telescope.telname, "Test Telescope\0");
  strcpy(telescope.observatory, "Mass. Inst. of Tech.\0");
  strcpy(telescope.altitude,"0m\0");
  strcpy(telescope.longitude,"-71:05:38.5\0");
  strcpy(telescope.latitude,"+42:21:32.4\0");
  telescope.scale=24.3;

  shm_write((char *)telvec.memory+sizeof(telvec), (void *)&telescope,
          sizeof(telescope));
  /*
   * Set up the initial telescope input values
   */

#ifdef __SOLARIS_5x__
  cmd_queue = mq_open("/LOIS_queue", O_RDWR);
  if (cmd_queue == (mqd_t) -1) {
    lois_log0("TELE: Command queue not opened");
```

```
    return(TCL_ERROR);
  }
  user_name = getenv("USER");
  if (user_name == NULL) {
    user_name = Tcl_GetVar2(interp, "env", "USER", TCL_GLOBAL_ONLY);
  }
#elif defined(__LINUX__)
  user_name = Tcl_GetVar2(interp, "env", "USER", TCL_GLOBAL_ONLY);
#endif

  if (Init_TelThreadInfo(user_name) < 0) {
    lois_log0("Error in telescope thread initialization routine!\n");
    return(TCL_ERROR);
  }

  for (count = 0; count < NUMTELTHREADS; count++) {
    actv_block[count] = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(actv_block[count], NULL);
  }

  lois_log4("Telescope: Test module ver 1.0");
  return(TCL_OK);

}

int teltest_init (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int   count, ibaud, obaud, bytes;
  char  foc_cmd[40], on='1';
  static char tel_cmd[80];

  GUI_state=1;
  telvec.tracking=ON;
  if (objc > 1) {
    for (count=1; count < objc; count++) {
      if (strcmp (objv[count]->bytes, "-nogui") == 0)
        GUI_state=0;
    }
  }
  tel_objcmd=Tcl_NewStringObj(tel_cmd, sizeof(tel_cmd));
  sprintf(tel_cmd, "source %s/scripts/teltest.tcl\0", info.loishome);
  Tcl_SetStringObj(tel_objcmd, tel_cmd, -1);
  if (Tcl_EvalObj(interp, tel_objcmd) == TCL_ERROR) {
    lois_log0("Error sourcing teltest script!");
    return(TCL_ERROR);
  }

  for (count =0 ; count < NUMKEYS*3 ; count++)
    telkeys[count]=Tcl_NewStringObj(tel_cmd, sizeof(tel_cmd));

  if (GUI_state) teltest_gui (clientdata, interp, objc, objv);
  pthread_create(&timer_thr, NULL, time_routine, (void *) NULL);
  pthread_detach(timer_thr);

  /* Open file streams for writing commands in test module */

  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);

  return(TCL_OK);

}

int teltest_delay (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
```

```
{
  input pipe_send;
  /*  long pidtag; */
  int count, num = 0, priority = 1, to_be_blocked[NUMTELTHREADS];
  char blocked_threads[10]="", tmp_str[5], ret_str[80];

  pipe_send.command = WAIT_CMD;
  pipe_send.numargs = 1;
  pipe_send.pidtag = get_curevalpidtag();

  if (objc > 2) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-2], NULL), "-prio") == 0) {
      if (Tcl_GetIntFromObj(interp, objv[objc-1], &priority) == TCL_ERROR) {
        Tcl_SetResult(interp, "Error getting priority for tel_block command!", N
ULL);
        return(TCL_ERROR);
      }
      objc -= 2;
    }
  }
  if (objc > 1) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-1], NULL), "-subc") == 0) {
      pipe_send.pidtag *= -1;
      objc--;
    }
  }

  if (objc > 1) {
    for (count = 1; count < objc; count++) {
      if (strcmp(objv[count]->bytes, "move") == 0) {
        to_be_blocked[num] = 0;
        num++;
      } else {
        if (strcmp(objv[count]->bytes, "focus") == 0) {
          to_be_blocked[num] = 1;
          num++;
        } else {
          if (strcmp(objv[count]->bytes, "other") == 0) {
            to_be_blocked[num] = 2;
            num++;
          } else {
            if (strcmp(objv[count]->bytes, "stop") == 0) {
              to_be_blocked[num] = 3;
              num++;
            }
          }
        }
      }
    }
  } else {
    for (count = 0; count < NUMTELTHREADS; count++) {
      to_be_blocked[num] = count;
      num++;
    }
  }

  for (count = 0; count < num; count++) {
    sprintf(pipe_send.args[0], "%d\0", to_be_blocked[count]);
    if (tel_write(pipe_send, priority) < 0) {
      sprintf(interp->result,
              "Error sending delay command to telescope thread #%d\nError: %s\n"
,
              to_be_blocked[count], write_errs[wr_err]);
      return(TCL_ERROR);
    }
    sprintf(tmp_str, "%d ", count);
```

```
    strcat(blocked_threads, tmp_str);
  }

  sprintf(ret_str, "log_4 {Telescope threads %shave been blocked}", blocked_thre
ads);
  Tcl_Eval(interp, ret_str);
  sprintf(interp->result, "%d", pipe_send.pidtag);
  return(TCL_OK);
}

int teltest_activate (ClientData clientdata, Tcl_Interp *interp,
                      int objc, Tcl_Obj *CONST objv[])
{
  int count;
  char err_str[80];

  for (count = 0; count < NUMTELTHREADS; count++) {
    if (sem_trywait(tel_block[count]) != 0) {
      if (errno != EAGAIN) {
        sprintf(err_str,
                "Error reinitializing semaphore #%d in tel_activate, errno: %d",

                count, errno);
        lois_log0(err_str);
      }

    }
  }

  for (count = 0; count < NUMTELTHREADS; count++) {
    pthread_mutex_lock(tel_cmds[count].mutex);
    if (pthread_mutex_trylock(actv_block[count]) < 0) {
      if (errno == EBUSY) {
        Tcl_SetResult(interp, "Cannot activate thread while removing command!",
NULL);
      } else {
        sprintf(err_str, "Error in trylock, tel_activate. Error no %d\n", errno)
;
        Tcl_SetResult(interp, err_str, TCL_VOLATILE);
        pthread_mutex_unlock(actv_block[count]);
      }
      return(TCL_ERROR);
    }
    tel_cmds[count].non_blocked |= 1; /* so that lowest priority queue is
                                         now unblocked */
    pthread_cond_signal(tel_cmds[count].go); /* If thread was sleeping but comma
nds were
                                                 in its queue */
    pthread_mutex_unlock(actv_block[count]);
    pthread_mutex_unlock(tel_cmds[count].mutex);
  }

  sprintf(interp->result, "Telescope threads have been reactivated\n");
  return(TCL_OK);
}

int teltest_gui (ClientData clientdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj *CONST objv[])
{

  if (GUI_state) {
    Tcl_SetStringObj(tel_objcmd, "teltest_init\0", -1);
    if (Tcl_EvalObj(interp, tel_objcmd) == TCL_ERROR) {
      lois_log0("Error starting telescope GUI");
      return(TCL_ERROR);
    }
    GUI_state=1;
    lois_log5("test Telescope GUI Opened");
    return(TCL_OK);
  } else {
    lois_log5("Telescope GUI Already Opened");
    return(TCL_ERROR);

  }

}

int teltest_status (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  static char tel_string[50]="", tel_cmd[80];
  int day_of_year, year_no, ra_hr, ra_min;
  int lst_hr, lst_min, lst_sec, ha_hr, ha_min, ha_sec;
  int bytes, nbytes, tmp_bytes;
  char val_str[8];
  float dec_day, flt_lst, j2000, ra_sec= 0.0, flt_ra, flt_ha;

  /* tcflush(telvec.tel_fd, TCIOFLUSH); */

  if (write(telvec.tel_fd, "TS\r\n", 4) < 0) {
    lois_log0("Error writing to telescope serial port in tel_status!");
    return(TCL_ERROR);
  }

  /* Begin telescope serial port configuration

     bytes = read(telvec.tel_fd, tel_string, 1);
     if (bytes == 0) {
     lois_log1("TELE: Command Did Not Acknowledge!");
     return(TCL_ERROR);
     }
     bytes = read(telvec.tel_fd, tel_string, 46);
     while (bytes < 46) {
     nbytes = read(telvec.tel_fd, (char *) tel_string+bytes, 46-bytes);
     bytes += nbytes;
     }
     tmp_bytes = bytes;

     End telescope serial port reconfiguration */

  cur_time = time(NULL);
  cur_date = gmtime(&cur_time);

  /* compute fraction of day elapsed and day of year*/

  dec_day = (float) ((cur_date->tm_hour + cur_date->tm_min / 60.0 +
                    cur_date->tm_sec/3600.0) / 24.0);

  day_of_year = cur_date->tm_yday+1;

  /* compute number of days since J2000 */

  if ((year_no = cur_date->tm_year-100) < -2) { /* Y2K compliance */
    year_no += 100;
  }

  j2000 = (float) (year_no*365 + (ABSOLUTE(year_no))/4 + day_of_year) - 1.5 +
    dec_day;

  /* compute local sidereal time see www.xylem.demon.co.uk/kepler/altaz.html
     for calculations */
```

```
  flt_lst = (100.46 + 0.985647*j2000 + test_long + 15.0*dec_day*24.0)/15.0;

  while (flt_lst > 24.0) {
    flt_lst -= 24.0;
  }

  while (flt_lst < 0.0) {
    flt_lst += 24.0;
  }

  lst_hr = (int) flt_lst;
  lst_min = (int) (60 * (flt_lst - (float) lst_hr));
  lst_sec = (int) (3600 * (flt_lst - (float) lst_hr - (float) lst_min/60.0));

  sprintf(val_str, "%02d:%02d:%02d\0", lst_hr, lst_min, lst_sec);
  strcpy(telescope.lst, val_str);

  /* compute hour angle (LST - RA) */

  sscanf(telescope.ra, "%2d:%2d:%4f", &ra_hr, &ra_min, &ra_sec);

  flt_ra = (float) (ra_hr+ (float) ra_min/60.0+ (float) ra_sec/3600.0);
  if ((flt_ha = flt_lst - flt_ra) < 0) {
    flt_ha += 24.0;
  }

  ha_hr = (int) flt_ha;
  ha_min = (int) (60 * (flt_ha - (float) ha_hr));
  ha_sec = (int) (3600 * (flt_ha - (float) ha_hr - (float) ha_min/60.0));

  sprintf(val_str, "%02d:%02d:%02d\0", ha_hr, ha_min, ha_sec);
  strcpy(telescope.ha, val_str);

  sprintf(val_str, "%02d:%02d:%02d\0", cur_date->tm_hour, cur_date->tm_min,
          cur_date->tm_sec);
  strcpy(telescope.ut, val_str);
  sprintf(val_str, "%02d/%02d/%02d\0", cur_date->tm_year % 100, cur_date->tm_mon
+1,
          cur_date->tm_mday);
  sprintf(tel_cmd, "$telestat dchars stat_tag 0 end");
  Tcl_Eval(interp, tel_cmd);
  sprintf(tel_cmd, "$telestat insert dec_tag 1 {%s}", telescope.dec);
  if (Tcl_Eval(interp, tel_cmd) == TCL_ERROR) {
    printf("Error dislaying DEC value!:%s:\n", interp->result);
  }
  sprintf(tel_cmd, "$telestat insert ra_tag 1 {%s}", telescope.ra);
  Tcl_Eval(interp, tel_cmd);
  sprintf(tel_cmd, "$telestat insert lst_tag 1 {%s}", telescope.lst);
  Tcl_Eval(interp, tel_cmd);
  strcpy(telescope.date, val_str);
  sprintf(tel_cmd, "$telestat insert date_tag 1 {%s %s}",telescope.date,
          telescope.ut);
  Tcl_Eval(interp, tel_cmd);
  sprintf(tel_cmd, "$telestat insert ha_tag 1 {%s}", telescope.ha);
  Tcl_Eval(interp, tel_cmd);
  sprintf(tel_cmd, "$telestat insert am_tag 1 {%s}", telescope.airmass);
  Tcl_Eval(interp, tel_cmd);

  sprintf(tel_cmd, "$telestat insert dome_tag 1 {%s}", telescope.dome);
  Tcl_Eval(interp, tel_cmd);

  sprintf(tel_cmd, "$telestat insert epoch_tag 1 {%s}", telescope.epoch);
  Tcl_Eval(interp, tel_cmd);

  /*  sprintf(tcl_cmd, "$telestat dchars foc_tag 0 end");
      lois_send(&telcmd);
```

```c
      sprintf(telcmd.command, "$telestat insert foc_tag 1 {%.2f}", telescope.foc
us);
      lois_send(&telcmd); */

  /*  tcflush(telvec.tel_fd, TCIOFLUSH);
      write(telvec.tel_fd, "\r", 1); */

  if (teltest_getfoc (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);
  shm_write((char *)telvec.memory+sizeof(telvec), (void *)&telescope,
            sizeof(telescope));
  return(TCL_OK);

}

int teltest_track (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int   count;  char state[5];
  char tel_cmd[80];

  /* which part to give to main interp, or telescope interp */
  /* tcflush(telvec.tel_fd, TCIOFLUSH); */

  if (objc > 1) {
    for (count=1; count < objc; count++) {
      if (strcmp (objv[count]->bytes, "on") == 0) {
        telvec.tracking = OFF;
      } else {
        if (strcmp (objv[count]->bytes, "off") == 0) {
        telvec.tracking = ON;
        }
      }
    }
  }

  if (telvec.tracking == OFF) {
    if (write(telvec.tel_fd, "TC 1\t", 5) < 0) {
      lois_log0("Error writing to telescope serial port in track!");
      return(TCL_ERROR);
    }
    lois_log3("TELE: Tracking has been turned ON");
    telvec.tracking = ON;
    sprintf(state, "ON\0");
  } else {
    if (telvec.tracking == ON) {
      if (write(telvec.tel_fd, "TC 0\t", 5) < 0) {
        lois_log0("Error writing to telescope serial port in track!");
        return(TCL_ERROR);
      }
      lois_log3("TELE:Tracking has been Turned OFF");
      telvec.tracking = OFF;
      sprintf(state, "OFF\0");
    } else {
      lois_log0("TELE: Tracking neither on nor off!");
      return(TCL_ERROR);
    }
  }

  /* tcflush(telvec.tel_fd, TCIOFLUSH); */

  if (teltest_status(clientdata, interp, objc, objv) != TCL_OK) {
    return(TCL_ERROR);
  }

  return(TCL_OK);
```

```c
}

int teltest_stop (ClientData clientdata, Tcl_Interp *interp,
                  int objc, Tcl_Obj *CONST objv[])
{

  int bytes, priority = 3;
  char err_str[40]; char *tmp_str;
  input pipe_send;


  sprintf(pipe_send.args[0], "both");
  switch (objc) {
  case 1:
    break;
  case 2:
    tmp_str = Tcl_GetStringFromObj(objv[1], NULL);
    if (strcmp(WAITOPT, tmp_str) == 0) {
      priority = -3;
    } else {
      if ((strcmp("move", tmp_str) != 0) && (strcmp("focus", tmp_str) != 0)) {
        Tcl_SetResult(interp, "Usage: tel_stop [move|focus]", NULL);
        return(TCL_ERROR);
      } else {
        sprintf(pipe_send.args[0], "%s\0", tmp_str);
      }
    }
    break;
  case 3:
    tmp_str = Tcl_GetStringFromObj(objv[1], NULL);
    if (strcmp("-prio", tmp_str) == 0) {
      if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
        sprintf(interp->result, "Error getting priority value!");
        return(TCL_ERROR);
      }
    } else {
      if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[2], NULL)) == 0) {
        priority = -3;
        if ((strcmp(tmp_str, "move") != 0) && (strcmp(tmp_str, "focus") != 0)) {
          sprintf(err_str, "Usage: tel_stop [move|focus] [%s]", WAITOPT);
          Tcl_SetResult(interp, err_str, TCL_VOLATILE);
          return(TCL_ERROR);
        } else {
          sprintf(pipe_send.args[0], "%s\0", tmp_str);
        }
      } else {
        sprintf(err_str, "Usage: tel_stop [move|focus] [%s]", WAITOPT);
        Tcl_SetResult(interp, err_str, TCL_VOLATILE);
        return(TCL_ERROR);
      }
    }
    break;
  case 4:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[2], NULL)) != 0) {
      sprintf(interp->result, "Usage: tel_stop [-prio priority]");
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[3], &priority) == TCL_ERROR) {
      sprintf(interp->result, "Error getting priority value!");
      return(TCL_ERROR);
    }
    tmp_str = Tcl_GetStringFromObj(objv[1], NULL);
    if ((strcmp("move", tmp_str) != 0) && (strcmp("focus", tmp_str) != 0)) {
      Tcl_SetResult(interp, "Usage: tel_stop [move|focus]", NULL);
      return(TCL_ERROR);
    } else {
      sprintf(pipe_send.args[0], "%s\0", tmp_str);
    break;
  }
  default:
    sprintf(interp->result, "Usage: tel_stop [move|focus]");
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = STOP;
  pipe_send.numargs = 1;
  pipe_send.pidtag = get_curevalpidtag();
  if (tel_write(pipe_send, priority) < 0) {
    sprintf(interp->result, "Error sending abort command\nError: %s\n", write_er
rs[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int teltest_co (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])

{

  int bytes, priority = 1;
  char *ra,*dec, result[1], err_str[40];
  input pipe_send;

  switch (objc) {
  case 3:
    ra=Tcl_GetStringFromObj(objv[1],NULL);
    dec=Tcl_GetStringFromObj(objv[2],NULL);
    break;
  case 4:
    ra=Tcl_GetStringFromObj(objv[1],NULL);
    dec=Tcl_GetStringFromObj(objv[2],NULL);
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[3], NULL)) != 0) {
      sprintf(err_str, "Usage: move hh:mm:ss.s dd:mm:ss [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 5:
    ra=Tcl_GetStringFromObj(objv[1],NULL);
    dec=Tcl_GetStringFromObj(objv[2],NULL);
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[3], NULL)) != 0) {
      sprintf(interp->result, "Usage: move hh:mm:ss.s dd:mm:ss [-prio priority]"
);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[4], &priority) == TCL_ERROR) {
      sprintf(interp->result, "Error getting priority value!");
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(interp->result, "Usage: move hh:mm:ss.s dd:mm:ss [%s]", WAITOPT);
```

```
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = MOVE;
  pipe_send.numargs = 2;
  sprintf(pipe_send.args[0], "%s\0", ra);
  sprintf(pipe_send.args[1], "%s\0", dec);

  pipe_send.pidtag = get_curevalpidtag();
  if (tel_write(pipe_send, priority) < 0) {
    sprintf(interp->result, "Error sending move command\nError: %s\n", write_err
s[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}


int teltest_rm (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int count, bytes, priority = 1;
  input pipe_send;
  char *ra_input="0", *dec_input="0";

  /* We know the -prio flag or the WAITOPT flag must come after the offset argum
ents */

  if ((objc > 0) && (objc < 8)) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-2], NULL), "-prio") == 0) {
      if (Tcl_GetIntFromObj(interp, objv[objc-1], &priority) == TCL_ERROR) {
        Tcl_SetResult(interp, "Error getting priority value!", NULL);
        return(TCL_ERROR);
      }
    } else {
      if (strcmp(Tcl_GetStringFromObj(objv[objc-1], NULL), WAITOPT) == 0) {
        priority = -1;
      }
    }

    for (count=1; count < objc; count++) {
      if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-ra") == 0) {
        ra_input=Tcl_GetStringFromObj(objv[++count], NULL);
      } else {
        if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-dec") == 0) {
          dec_input=Tcl_GetStringFromObj(objv[++count], NULL);
        }
      }
    }
  } else {
    sprintf(interp->result,
            "Usage: rmove -ra ra_offset -dec dec_offset [%s]", WAITOPT);
    return(TCL_ERROR);
  }

  pipe_send.command = RMOVE;
  pipe_send.numargs = 2;
  sprintf(pipe_send.args[0], "%s\0", ra_input);
  sprintf(pipe_send.args[1], "%s\0", dec_input);
  pipe_send.pidtag = get_curevalpidtag();
  if (tel_write(pipe_send, priority) < 0) {
    sprintf(interp->result, "Error sending rmove command\nError: %s\n", write_er
rs[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);

}

int teltest_fc (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  int count, bytes, priority = 1;
  char *speed, *time, err_str[40];
  input pipe_send;

  switch (objc) {
  case 5:
    for (count=1; count < objc; count++) {
      if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-spd") == 0) {
        speed=Tcl_GetStringFromObj(objv[++count],NULL);
      }
      else {
        if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-time") == 0) {
          time = Tcl_GetStringFromObj(objv[++count],NULL);
        }
      }
    }
    break;
  case 6:
    for (count=1; count < objc; count++) {
      if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-spd") == 0) {
        speed=Tcl_GetStringFromObj(objv[++count],NULL);
      }
      else {
        if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-time") == 0) {
          time = Tcl_GetStringFromObj(objv[++count],NULL);
        }
      }
    }
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[5], NULL)) != 0) {
      sprintf(err_str, "Usage: focus_ctl -spd slow|fast -time tenths_of_sec [%s]
", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 7:
    for (count=1; count < objc; count++) {
      if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-spd") == 0) {
        speed=Tcl_GetStringFromObj(objv[++count],NULL);
      }
      else {
        if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-time") == 0) {
          time = Tcl_GetStringFromObj(objv[++count],NULL);
```

```
        }
      }
    }
    if (strcmp(Tcl_GetStringFromObj(objv[5], NULL), "-prio") != 0) {
      sprintf(interp->result,
              "Usage: focus_ctl -spd slow|fast -time tenths_of_sec [-prio priori
ty]");
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[6], &priority) == TCL_ERROR) {
      sprintf(interp->result, "Error getting priority value!");
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(interp->result, "Usage: focus_ctl -spd slow|fast -time tenths_of_sec
 [%s]", WAITOPT);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = FOC_CTL;
  pipe_send.numargs = 2;
  sprintf(pipe_send.args[0], "%s\0", speed);
  sprintf(pipe_send.args[1], "%s\0", time);
  pipe_send.pidtag = get_curevalpidtag();

  if (tel_write(pipe_send, priority) < 0) {
    sprintf(interp->result, "Error sending focus_ctl command\nError: %s\n", writ
e_errs[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int teltest_fg (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int bytes, priority = 1;
  char *gopos, err_str[40];
  input pipe_send;

  switch (objc) {
  case 2:
    gopos = Tcl_GetStringFromObj(objv[1], NULL);
    break;
  case 3:
    gopos = Tcl_GetStringFromObj(objv[1], NULL);
    if (strcmp (Tcl_GetStringFromObj(objv[2], NULL), WAITOPT) != 0) {
      sprintf(err_str, "Usage: focus_go position [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 4:
    gopos = Tcl_GetStringFromObj(objv[1], NULL);
    if (strcmp (Tcl_GetStringFromObj(objv[2], NULL), "-prio") != 0) {
      sprintf(interp->result,
              "Usage: focus_go position [-prio priority]");
        return(TCL_ERROR);
      }
      if (Tcl_GetIntFromObj(interp, objv[3], &priority) == TCL_ERROR) {
        sprintf(interp->result, "Error getting priority value!");
        return(TCL_ERROR);
      }
      break;
    default:
      sprintf(interp->result, "Usage: focus_go position [%s]", WAITOPT);
      return(TCL_ERROR);
  }

  pipe_send.command = FOC_GO;
  pipe_send.numargs = 1;
  sprintf(pipe_send.args[0], "%s\0", gopos);
  pipe_send.pidtag = get_curevalpidtag();
  if (tel_write(pipe_send, priority) < 0) {
    sprintf(interp->result, "Error sending focus_go command\nError: %s\n", write
_errs[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int teltest_er (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int  count, ra_rate = 0, dec_rate = 0;
  char  tel_cmd[40];


  if (objc > 1) {
    for (count=1; count < objc; count++) {
      if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-ra") == 0) {
        Tcl_GetIntFromObj(interp,objv[++count],&ra_rate);
        ra_rate *= 10;
      }
      else {
        if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "-dec") == 0) {
          Tcl_GetIntFromObj(interp,objv[++count],&dec_rate);
          dec_rate *= 10;
        }
      }
    }
    sprintf(tel_cmd, "ephm_rate %d %d\0", ra_rate, dec_rate);

  } else {
    sprintf(interp->result,"Usage: ephm_rate -ra ra_rate -dec dec_rate");
    return(TCL_ERROR);
  }

  Tcl_GetIntFromObj(interp, objv[1], &ra_rate);
  Tcl_GetIntFromObj(interp, objv[2], &dec_rate);
  /* do stuff */
  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);

  return(TCL_OK);
```

```
}

int teltest_cl (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int  count, rate=0;
  char  tel_cmd[40];

  if (objc > 1) {
    for (count=1; count < objc; count++) {
      if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "siderial") == 0) {
        rate = 1;
      } else {
        if (strcmp (Tcl_GetStringFromObj(objv[count],NULL), "rate") == 0) {
          rate = 2;
        }
      }
    }
    sprintf(tel_cmd, "c_lock %d\0", rate);
  } else {
    sprintf(interp->result,"Usage: c_lock siderial|rate|off(default)");
    return(TCL_ERROR);
  }


  Tcl_GetIntFromObj(interp, objv[1], &rate);
  /* do stuff */

  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);

  return(TCL_OK);
}

int teltest_fw (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  int  count, position=0;
  char tel_cmd[40];

  /* how is this supposed to run if there is no instrument module active? */

  if (objc == 2) {
    Tcl_GetIntFromObj(interp,objv[1],&position);
    sprintf(tel_cmd, "filt %d\0", position);
  } else {
    sprintf(interp->result,"Usage: filt position");
    return(TCL_ERROR);
  }

  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);
  return(TCL_OK);
}

int teltest_sp (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int  count;
  char tel_cmd[40];

  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);
  return(TCL_OK);
}
```

```c
int teltest_rp (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int  count;
  char tel_cmd[40];

  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);
  return(TCL_OK);
}

int teltest_getfoc (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  int bytes;
  char tel_cmd[80]; char focstr[10];

  /* Begin telescope serial port communication

     tcflush(foc_fd, TCIOFLUSH);

     write(foc_fd, ":MEAS:VOLT:DC? 10,0.001\n", 24);
     bytes=read(foc_fd, (void *) focstr, sizeof(focstr));
     telescope.focus=atof(focstr);

     Tcl_Eval(interp, "$telestat dchars foc_tag 0 end");
     sprintf(foccmd, "$telestat insert foc_tag 1 {%s}", focstr);
     Tcl_Eval(interp, foccmd);

     sprintf(interp->result,"Current Focus Position %s", focstr);
     tcflush(foc_fd, TCIOFLUSH);

     End Telescope serial port communication */

  if (write(foc_fd, "GET_FOCUS\n", 10) < 0) {
    lois_log0("Error writing to focus serial port in getfoc!");
    return(TCL_ERROR);
  }

  sprintf(tel_cmd, "$telestat dchars foc_tag 0 end");
  Tcl_Eval(interp, tel_cmd);
  sprintf(tel_cmd, "$telestat insert foc_tag 1 {%.1f steps (-30000 to 30000)}",
telescope.focus);
  Tcl_Eval(interp, tel_cmd);
  return(TCL_OK);

}

int teltest_header (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[])
{
  int   count;
  static char   keyvalue[70];

  if (teltest_status (clientdata, interp, objc, objv) != TCL_OK)
    return(TCL_ERROR);

  for (count = 0 ; count < NUMKEYS ; count++) {
    Tcl_SetStringObj(telkeys[count*3], telkeywords[count], -1);
    Tcl_SetStringObj(telkeys[count*3+2], telcomments[count], -1);
  }

  shm_write((char *)telvec.memory+sizeof(telvec), (void *)&telescope,
            sizeof(telescope));
```

```c
  /* Set the Telescope Name */
  sprintf(keyvalue,"S %s\0", telescope.telname);
  Tcl_SetStringObj(telkeys[1], keyvalue, -1);
  /* Set the Sideral Time*/
  sprintf(keyvalue, "S %s\0", telescope.lst);
  Tcl_SetStringObj(telkeys[4], keyvalue, -1);
  /* Set the RA */
  sprintf(keyvalue, "S %s\0", telescope.ra);
  Tcl_SetStringObj(telkeys[7], keyvalue, -1);
  /* Set the DEC*/
  sprintf(keyvalue, "S %s\0", telescope.dec);
  Tcl_SetStringObj(telkeys[10], keyvalue, -1);
  /* Set the EQUINOX */
  sprintf(keyvalue, "S %s\0", telescope.epoch);
  Tcl_SetStringObj(telkeys[13], keyvalue, -1);
  /* Set the EPOCH*/
  sprintf(keyvalue, "S %s\0", telescope.epoch);
  Tcl_SetStringObj(telkeys[16], keyvalue, -1);
  /* Set the ZD */
  sprintf(keyvalue, "S %s\0", telescope.zd);
  Tcl_SetStringObj(telkeys[19], keyvalue, -1);
  /* Set the AIRMASS*/
  sprintf(keyvalue, "S %s\0", telescope.airmass);
  Tcl_SetStringObj(telkeys[22], keyvalue, -1);
  /* Set the FOCUS*/
  sprintf(keyvalue, "F %f\0", telescope.focus);
  Tcl_SetStringObj(telkeys[25], keyvalue, -1);

  Store_header(clientdata, interp, NUMKEYS*3, telkeys);
}

int teltest_locate (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[])
{

  char *fromxy, *toxy, *pos_ptr;
  char tel_cmd[80], tmp_str[5];
  int x1,x2,y1,y2, count, pos;
  float arcpp, xmove, ymove;

  count=0;
  if (objc == 3) {
    fromxy=Tcl_GetStringFromObj(objv[++count],NULL);
    toxy=Tcl_GetStringFromObj(objv[++count],NULL);
    sprintf(tel_cmd, "locate %s %s\0", fromxy, toxy);
  } else {
    sprintf(interp->result, "Usage: locate from_x:from_y to_x:to_y");
    return(TCL_ERROR);
  }

  fromxy = Tcl_GetStringFromObj(objv[1], NULL);
  toxy = Tcl_GetStringFromObj(objv[2], NULL);

  shm_read((void *)&ccd, (char *)ccdvec.memory+sizeof(ccdvec), sizeof(ccd));
  arcpp=(ccd.pix_size*1e-6)*(telescope.scale/1e-3);

  if ((pos_ptr=strstr(fromxy, ":")) == NULL ) {
    sprintf(interp->result, "Cannot determine From x/y coords");
    return(TCL_ERROR);
  }

  bzero(tmp_str, sizeof(tmp_str));
  strncpy(tmp_str, fromxy, pos_ptr-fromxy);
  x1=atoi(tmp_str);
  strncpy(tmp_str, pos_ptr+1, (fromxy+strlen(fromxy))-pos_ptr);
  y1=atoi(tmp_str);
  if ((pos_ptr=strstr(toxy, ":")) == NULL ) {
    sprintf(interp->result,"Cannot determine To x/y coords");
    return(TCL_ERROR);
  }

  bzero(tmp_str, sizeof(tmp_str));
  strncpy(tmp_str, toxy, pos_ptr-toxy);
  x2=atoi(tmp_str);
  strncpy(tmp_str, pos_ptr+1, (toxy+strlen(toxy))-pos_ptr);
  y2=atoi(tmp_str);
  xmove=(x1-x2)*arcpp*ccd.col_bin;
  ymove=(y1-y2)*arcpp*ccd.row_bin;

  sprintf(tel_cmd, "rmove -ra %-.0f -dec %-.0f", xmove, ymove);
  if (Tcl_Eval(interp, tel_cmd) != TCL_ERROR) {
    return(TCL_OK);
  }

  return(TCL_OK);
}

int teltest_sims(ClientData clientdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj *CONST objv[])
{

  char *which_delay;

  if (objc != 2) {
    Tcl_SetResult(interp, "Usage: tel_simdelays [move|focus]", NULL);
    return(TCL_ERROR);
  }
  if ((which_delay = Tcl_GetStringFromObj(objv[1], NULL)) == NULL) {
    Tcl_SetResult(interp, "Error getting argument to simdelays!", NULL);
    return(TCL_ERROR);
  }

  if (strcmp(which_delay, "move") == 0) {
    if (move_delay == ON) {
      move_delay = OFF;
      sprintf(telcmd.command, "$telestat dchars move_delay_tag 0 end");
      lois_send(&telcmd);
      sprintf(telcmd.command, "$telestat insert move_delay_tag 1 {OFF}");
      lois_send(&telcmd);
    } else {
      move_delay = ON;
      sprintf(telcmd.command, "$telestat dchars move_delay_tag 0 end");
      lois_send(&telcmd);
      sprintf(telcmd.command, "$telestat insert move_delay_tag 1 {ON}");
      lois_send(&telcmd);
    }
  } else {
    if (strcmp(which_delay, "focus") == 0) {
      if (focus_delay == ON) {
        focus_delay = OFF;
        sprintf(telcmd.command, "$telestat dchars foc_delay_tag 0 end");
        lois_send(&telcmd);
        sprintf(telcmd.command, "$telestat insert foc_delay_tag 1 {OFF}");
        lois_send(&telcmd);
      } else {
        focus_delay = ON;
        sprintf(telcmd.command, "$telestat dchars foc_delay_tag 0 end");
        lois_send(&telcmd);
        sprintf(telcmd.command, "$telestat insert foc_delay_tag 1 {ON}");
        lois_send(&telcmd);
      }
```

```
    } else {
      Tcl_SetResult(interp, "Must specify move or focus delay to simulate!", NUL
L);
      return(TCL_ERROR);
    }
  }
  return(TCL_OK);
}



/****************************************************************
*
*   Private Functions Not Exported As Script Commands
*
****************************************************************/

int tel_write(input send_cmd, int level) {

  int thread_num, get_results = 0, count;
  input *tmp_ptr;

  if ((level == 0) || (ABSOLUTE(level) > NUMPRIORITIES)) {
    wr_err = EILLPRIO; /* illegal priority level */
    return (-1);
  }
  switch(send_cmd.command) {
  case MOVE:case RMOVE:case LOCATE:
    thread_num = 0;
    break;
  case FOC_CTL:case FOC_GO:
    thread_num = 1;
    break;
  case ER:case CL:case FW:case SP:case RP:
    thread_num = 2;
    break;
  case STATUS:case STOP:
    thread_num = 3;
    break;
  case WAIT_CMD:
    thread_num = strtol(send_cmd.args[0], (char **) NULL, 10);
    if ((thread_num < 0) || (thread_num >= NUMTELTHREADS)) {
      wr_err = EBADWAIT;
      return(-1);
    }
    send_cmd.numargs = 0;
    break;
  default:
    wr_err = ENOCMD; /* no defined command */
    return(-1);
    break;
  }

  pthread_mutex_lock(tel_cmds[thread_num].mutex);

  /* A negative priority means the calling routine wants the results back */

  if (level < 0) {
    get_results = 1;
    level = level * -1;
    send_cmd.command = -1 * send_cmd.command;
  }

  level--; /* So that priority 1 commands are sent to queue #0, priority 2
              commands are sent to queue #1... */
  tmp_ptr = (input *) ckalloc(sizeof(input));
  send_cmd.next = NULL;
  shm_write((void *) tmp_ptr, (void *) &send_cmd, sizeof(send_cmd));
  if (tel_cmds[thread_num].p_queue[level].head == NULL) {
    tel_cmds[thread_num].p_queue[level].head = tmp_ptr;
    tel_cmds[thread_num].p_queue[level].tail = tmp_ptr;
  } else {
    tel_cmds[thread_num].p_queue[level].tail->next = tmp_ptr;
    tel_cmds[thread_num].p_queue[level].tail = tmp_ptr;
  }
  /*  if (level == 1) {
    tel_cmds[thread_num].num_elts++;
  } else {
    tel_cmds[thread_num].highprio_elts++;
  }
  */

  if ((tel_cmds[thread_num].non_blocked & tel_cmds[thread_num].non_empty) == 0)
{
    pthread_cond_signal(tel_cmds[thread_num].go);
  }

  tel_cmds[thread_num].non_empty |= (1 << level); /* set non_empty value so that
                                                     the bit in position $level$
 is
                                                     set to 1, and the others
                                                     are unaffected */

  /*  if (((1-tel_cmds[thread_num].blocked) * tel_cmds[thread_num].num_elts +
       tel_cmds[thread_num].highprio_elts) == 1) {
    pthread_cond_signal(tel_cmds[thread_num].go);
    }
  */
  pthread_mutex_unlock(tel_cmds[thread_num].mutex);

  /******************************************/
  /* Get results if command asked for them */
  /******************************************/

  if (get_results) {
    return(tel_receive(thread_num));
  } else {
    return(0);
  }

}

int tel_receive(int thread_num) {

  printf("\nTel. Getting results\n");
  pthread_mutex_lock(tel_results[thread_num].mutex);
  while (tel_results[thread_num].value == RUN_WAIT) {
    printf("Tel. Waiting...\n");
    pthread_cond_wait(tel_results[thread_num].go, tel_results[thread_num].mutex)
;
  }

  if (tel_results[thread_num].value == RUN_ERROR) {
    wr_err = EINTRNL; /* error executing internal part of command */
    tel_results[thread_num].value = RUN_WAIT;
    pthread_mutex_unlock(tel_results[thread_num].mutex);
    printf("Tel got results %d\n", thread_num);
    return(-1);
  }

  if (tel_results[thread_num].value == RUN_REMOVED) {
```

```
    wr_err = EREMOVED; /* command removed from thread queue */
    tel_results[thread_num].value = RUN_WAIT;
    pthread_mutex_unlock(tel_results[thread_num].mutex);
    printf("Tel got results\n");
    return(-1);
  }

  tel_results[thread_num].value = RUN_WAIT;
  pthread_mutex_unlock(tel_results[thread_num].mutex);
  printf("Got results\n");
  return(0);
}

/*****************************************************************************
*/
/************ Commands for removing and adjusting module thread queues ********
*/
/*****************************************************************************
*/

int tel_remove(long rm_pidtag, int flag)
{

  int p_index, t_index, found = 0, blocks_this_queue, total = 0;
  input *previous, *traverse;

  if (rm_pidtag == 0) {
    printf("Illegal pidtag value!");
    return(-1);
  }
  if (rm_pidtag > 0) {
    for (t_index = 0; t_index < NUMTELTHREADS; t_index++) {
      pthread_mutex_lock(tel_cmds[t_index].mutex);
      blocks_this_queue = 0;
      for (p_index = 0; p_index < NUMPRIORITIES; p_index++) {
        traverse = tel_cmds[t_index].p_queue[p_index].head;
        previous = NULL;
        while (traverse != NULL) {
          if (traverse->pidtag == rm_pidtag) {

            /* We found a command with the desired pidtag.  Remove it, (if the f
lag is set)
               but keep the rest of the list structure intact.  We could
               actually break out of the while here, since there should be
               only one command with the desired pidtag, but we keep going
               just in case (there's some case I haven't thought of) */

            /* Special case for block commands, because a block issues multiple
               inputs to the queue */

            if (traverse->command != WAIT_CMD) {
              found++;
            } else {
              blocks_this_queue++;
            }

            if (flag) {
              if (traverse->command < 0) {
                pthread_mutex_lock(tel_results[t_index].mutex);
                tel_results[t_index].value = RUN_REMOVED;
                pthread_cond_signal(tel_results[t_index].go);
                pthread_mutex_unlock(tel_results[t_index].mutex);
              }

              if (previous == NULL) {
                if ((tel_cmds[t_index].p_queue[p_index].head = traverse->next) =
= NULL) {
                  tel_cmds[t_index].non_empty &= ~(1 << p_index);
                }

                ckfree(traverse);
                traverse = tel_cmds[t_index].p_queue[p_index].head;

              } else {
                ckfree(previous->next);
                previous->next = traverse->next;
                traverse = traverse->next;
              }
            } else {
              previous = traverse;
              traverse = traverse -> next;
            }

          } else {
            previous = traverse;
            traverse = traverse->next;
          }
        }
      }

      /* See if the currently running command is a block.  If not, check to see
that the
         currently running command doesn't have the same process tag as the proc
ess tag
         we're trying to remove (if it does, then you can't remove it, since it'
s being executed!)
         If the currently running command is a block, check its process tag.  If
 the
         process tag of the current block is the one we're trying to remove, it'
s too late
         (we can't remove it) so return 0 */

      if (tel_cmds[t_index].curr_run.command != WAIT_CMD) {
        if (found && (rm_pidtag == tel_cmds[t_index].curr_run.pidtag)) {
          found++;
        }
      } else {
        if (rm_pidtag == tel_cmds[t_index].curr_run.pidtag) {
          if (blocks_this_queue) {
            blocks_this_queue++;
          } else {
            pthread_mutex_unlock(tel_cmds[t_index].mutex);
            return(0); /* The currently running process tag is the one we're try
ing to
                          remove, so the process tag couldn't be found */
          }
        }
      }
      pthread_mutex_unlock(tel_cmds[t_index].mutex);
      if (blocks_this_queue > 1) {
        return(blocks_this_queue);
      }
      total += blocks_this_queue;
    }
    if ((found == 0) && (total != 0)) {
      found = 1; /* If we got here but didn't find any commands, then we are rem
oving blocks,
                    so send to the remove command that we should still look here
 for blocks */
    }
  } else {
    /* pidtag is negative, so it is associated with a block command sent by anot
```

```
her routine */

    for (t_index = 0; t_index < NUMTELTHREADS; t_index++) {
      pthread_mutex_lock(tel_cmds[t_index].mutex);
      traverse = tel_cmds[t_index].p_queue[0].head; /* use priority of zero beca
use
                                                       that is where blocks are
going */
      previous = NULL;
      while (traverse != NULL) {
        if (traverse->pidtag == rm_pidtag) {
          if (traverse->command < 0) {
            pthread_mutex_lock(tel_results[t_index].mutex);
            tel_results[t_index].value = RUN_REMOVED;
            pthread_cond_signal(tel_results[t_index].go);
            pthread_mutex_unlock(tel_results[t_index].mutex);
          }

          if (previous == NULL) {
            if ((tel_cmds[t_index].p_queue[0].head = traverse->next) == NULL) {
              tel_cmds[t_index].non_empty &= ~(1 << 0);
            }

            ckfree(traverse);
            traverse = tel_cmds[t_index].p_queue[0].head;

          } else {
            ckfree(previous->next);
            previous->next = traverse->next;
            traverse = traverse->next;
          }
          found++;
        } else {
          previous = traverse;
          traverse = traverse->next;
        }
      }

      if (!found && (rm_pidtag == tel_cmds[t_index].curr_run.pidtag)) {

        /* If the block command wasn't in the queue for some thread, then a coup
le things
           could have happened:

           1. The block command could have gone through already.  The pidtag fie
ld
              of the curr_run structure will be the same as the rm_pidtag argume
nt passed
              to this function.  In this case, we want to reactivate the queue a
nd signal
              if necessary.  We know that the block we're looking for must be th
e one
              blocking the queue because of the pidtag value of the rm_pidtag ar
gument.
              Since the rm_pidtag argument is NEGATIVE, this part of the removal
 routine
              (to remove block subcommands) is only called if the main command (
the one
              that sent the blocks, i.e. an exposure) was found in the queue.  S
ince the
              main command is in the queue, it means that it MUST have written i
ts blocks
              already.

           2. The command we're trying to remove didn't send any block commands.
  In that
```

138

```
               case, the pidtag field of the curr_run structure would be differen
t than
               the rm_pidtag argument passed to this function.  We don't want to
reactivate
               the queue, nor do we want to signal in this case.

         */

         if (sem_trywait(tel_block[t_index]) != 0) {
           if (errno != EAGAIN) {

             /* The semaphore doesn't have to be zero, someone could call an expo
sure, and
               before trying to remove it, reactivate a queue, which would wait
the semaphore */
             lois_log0("Error reinitializing semaphore #%d in tel_remove, errno:
%d",
                        t_index, errno);
           }
         }

         old_state[t_index] |= 1; /* unblock lowest priority queue */
         pthread_cond_signal(tel_cmds[t_index].go);  /* If thread was sleeping bu
t commands were
                                                          in its queue */
       }
       found = 0; /* for next thread... */
       pthread_mutex_unlock(tel_cmds[t_index].mutex);
     }
   }
   return(found);
}

int tel_flush(long pidtag)

{

  int count, p_index;
  input *traverse, *previous;

  for (count = 0; count < NUMTELTHREADS; count++) {
    pthread_mutex_lock(tel_cmds[count].mutex);
    for (p_index = 0; p_index < NUMPRIORITIES; p_index++) {
      traverse = tel_cmds[count].p_queue[p_index].head;
      previous = NULL;
      while (traverse != NULL) {
        if ((pidtag == ABSOLUTE(traverse->pidtag)) || (pidtag == 0)) {
          if (traverse->command < 0) {
            pthread_mutex_lock(tel_results[count].mutex);
            tel_results[count].value = RUN_REMOVED;
            pthread_cond_signal(tel_results[count].go);
            pthread_mutex_unlock(tel_results[count].mutex);
          }
          if (previous == NULL) {
            if ((tel_cmds[count].p_queue[p_index].head = traverse->next) == NULL
) {

              tel_cmds[count].non_empty &= ~(1 << p_index);
            }
            ckfree(traverse);
            traverse = tel_cmds[count].p_queue[p_index].head;
          } else {
            ckfree(previous->next);
            previous->next = traverse->next;
            traverse = traverse->next;
          }
        } else {
```

```
          previous = traverse;
          traverse = traverse->next;
        }
      }
    }
    if (ABSOLUTE(tel_cmds[count].curr_run.pidtag) == pidtag) {
      /* If this thread is currently blocked... */
      old_state[count] |= 1;
    }
    pthread_mutex_unlock(tel_cmds[count].mutex);
  }
  return(0);
}

int tel_blocked_state(int option)

{
  int count;

  for (count = 0; count < NUMTELTHREADS; count++) {
    pthread_mutex_lock(tel_cmds[count].mutex);
    switch (option) {
    case SAVE_STATE:
      pthread_mutex_lock(actv_block[count]);
      old_state[count] = tel_cmds[count].non_blocked;
      tel_cmds[count].non_blocked = (~0 << (NUMPRIORITIES - 1));

      /* To block all telescope module thread queues while a command is being
         removed */
      break;
    case RESTORE_STATE:
      tel_cmds[count].non_blocked = old_state[count];
      pthread_cond_signal(tel_cmds[count].go); /* If a queue with commands in it
                                                  just became unblocked...*/
      pthread_mutex_unlock(actv_block[count]);
      break;
    default:
      break;
    }
    pthread_mutex_unlock(tel_cmds[count].mutex);
  }
  return 0;
}



/******************************************************************************
*/
/****************************** Counter thread ********************************
*/
/******************************************************************************
*/

void * time_routine (void * arg) {

  struct timeval n_time, l_time;
  cmd_struct stat_cmd;
  int lst_hr, lst_min, lst_sec, ha_hr, ha_min, ha_sec;
  float ra_sec;
  char val_str[8];
  time_t cur_time;
  struct tm *cur_date;
  struct timespec wait_timespec;
  /*  pthread_mutex_t t_mutex = PTHREAD_MUTEX_INITIALIZER;
      pthread_cond_t t_cond = PTHREAD_COND_INITIALIZER; */
  stat_cmd.priority = MIN_PRIORITY;
  stat_cmd.t_times = 0;
  stat_cmd.t_execute = 0;
  stat_cmd.t_interval = 0;

  gettimeofday(&l_time, NULL);
  for (;;) {
    gettimeofday(&n_time, NULL);

    if (n_time.tv_sec - l_time.tv_sec > 0) {
      /* update graphics */
      l_time = n_time;
      gettimeofday(&n_time, NULL);
      cur_time = time(NULL);
      cur_date = gmtime(&cur_time);

      sscanf(telescope.lst, "%02d:%02d:%02d", &lst_hr, &lst_min, &lst_sec);

      if (!(lst_sec = (lst_sec+1) % 60)) {
        if (!(lst_min = (lst_min+1) % 60)) {
          if (!(lst_hr = (lst_hr+1) % 24)) {
            lst_hr=lst_sec=lst_min=0;
          }
        }
      }

      sprintf(val_str, "%02d:%02d:%02d\0", lst_hr, lst_min, lst_sec);
      strcpy(telescope.lst, val_str);

      /* recompute hour angle if tracking is on, recompute ra if tracking is off
 */

      /*      sscanf(telescope.ra, "%2d:%2d:%4f", &ra_hr, &ra_min, &ra_sec);

      flt_ra = (float) (ra_hr+ (float) ra_min/60.0+ (float) ra_sec/3600.0);
      if ((flt_ha = flt_lst - flt_ra) < 0) {
        flt_ha += 24.0;
        } */

      if (telvec.tracking == ON) {
        sscanf(telescope.ha, "%02d:%02d:%02d", &ha_hr, &ha_min, &ha_sec);

        if (!(ha_sec = (ha_sec+1) % 60)) {
          if (!(ha_min = (ha_min+1) % 60)) {
            if (!(ha_hr = (ha_hr+1) % 24)) {
              ha_hr=ha_sec=ha_min=0;
            }
          }
        }
        sprintf(val_str, "%02d:%02d:%02d\0", ha_hr, ha_min, ha_sec);
        strcpy(telescope.ha, val_str);
      } else {
        sscanf(telescope.ra, "%2d:%2d:%4f", &ha_hr, &ha_min, &ra_sec);
        ha_sec = (int) ra_sec;
        ra_sec = ra_sec - (float) ha_sec;

        if (!(ha_sec = (ha_sec+1) % 60)) {
          if (!(ha_min = (ha_min+1) % 60)) {
            if (!(ha_hr = (ha_hr+1) % 24)) {
              ha_hr=ha_sec=ha_min=0;
            }
          }
        }
        sprintf(val_str, "%02d:%02d:%04.1f\0", ha_hr, ha_min, (float) ha_sec + r
a_sec);
        strcpy(telescope.ra, val_str);
```

```
      }
      sprintf(val_str, "%02d:%02d:%02d\0", cur_date->tm_hour, cur_date->tm_min,
              cur_date->tm_sec);
      strcpy(telescope.ut, val_str);
      sprintf(val_str, "%02d/%02d/%02d\0", cur_date->tm_year % 100, cur_date->tm
_mon+1,
              cur_date->tm_mday);
      sprintf(stat_cmd.command, "$telestat dchars lst_tag 0 end");
      lois_send(&stat_cmd);
      sprintf(stat_cmd.command, "$telestat insert lst_tag 1 {%s}", telescope.lst
);
      lois_send(&stat_cmd);
      sprintf(stat_cmd.command, "$telestat dchars date_tag 0 end");
      lois_send(&stat_cmd);
      strcpy(telescope.date, val_str);
      sprintf(stat_cmd.command, "$telestat insert date_tag 1 {%s %s}",telescope.
date,
              telescope.ut);
      lois_send(&stat_cmd);
      if (telvec.tracking == ON) {
        sprintf(stat_cmd.command, "$telestat dchars ha_tag 0 end");
        lois_send(&stat_cmd);
        sprintf(stat_cmd.command, "$telestat insert ha_tag 1 {%s}", telescope.ha
);
        lois_send(&stat_cmd);
      } else {
        sprintf(stat_cmd.command, "$telestat dchars ra_tag 0 end");
        lois_send(&stat_cmd);
        sprintf(stat_cmd.command, "$telestat insert ra_tag 1 {%s}", telescope.ra
);
        lois_send(&stat_cmd);
      }

    } else {
      /*      wait_timespec.tv_sec = time(NULL);
      wait_timespec.tv_nsec = 200000000;
      pthread_cond_timedwait(&t_cond, &t_mutex, &wait_timespec); */
      lois_sleep(200000000);
    }
  }
}
```

```
#include <netinet/in.h>

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#define _REENTRANT
#define _P __P
#endif

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <termios.h>
#include <math.h>
#include <time.h>


/* Tcl/Tk Includes */
#include <tcl.h>
#include <tk.h>

/* CDL Library Include */
#include <cdl.h>

/* LOIS Library Include */
#include <lois.h>

/* Instrument Include */
#include <LowellTel.h>
#include <ModThrd.h>

struct termios srlprm, *serialprm_a, *serialprm_b;
int foc_fd;
int move_delay, focus_delay, move_aborted = 0, foc_aborted = 0;
tel_struct telescope;
struct telescope_vectors telvec;
info_struct info;
static cmd_struct telcmd;
char telmsg[40];

static void reset_curr_run(int number);

const float ra_threshold = 0.1/30.0; /* Thresholds for test modules when simulat
ing moves */
const float dec_threshold = 0.05;     /* in RA and DEC
           */

extern float test_long;

int Init_TelThreadInfo(char *user)
{
  int count, count2, obaud, ibaud;
  char telname[60], focname[60];

  /* Open telescope and focus serial comm ports.  In test
```

```
     module, these are simply file descriptors */
  sprintf(telname, "%s/users/%s/tel_cmds.txt", info.loishome, user);
  sprintf(focname, "%s/users/%s/focus_cmds.txt", info.loishome, user);

  if ((telvec.tel_fd = open(telname, O_RDWR | O_CREAT | O_TRUNC, 0644)) < 0) {
    lois_log0("TELE: Error Opening Telescope Serial Port");
    return(-1);
  }

  if ((foc_fd = open(focname, O_RDWR | O_CREAT | O_TRUNC, 0644)) < 0) {
    lois_log0("TELE: Error Opening Focus Serial Port");
    return(-1);
  }

/* Getting the termios I/O structure */

  /*  Commented out in test module (using a file descriptor)

      if (tcgetattr(telvec.tel_fd, &srlprm) < 0) {
    lois_log0("TELE: Error Getting Serial Attrib");
    return(-1);
  }

  serialprm_a=&srlprm;
  serialprm_b=calloc(1,sizeof(srlprm));

  obaud=cfgetospeed(serialprm_a);
  ibaud=cfgetispeed(serialprm_a);
  if (obaud != B9600 ) obaud=B9600;
  if (ibaud != B9600 ) ibaud=B9600;
  if (cfsetospeed(serialprm_a, obaud) < 0) {
    lois_log0("Error setting output baud rate on telescope serial port!");
    return(-1);
  }
  if (cfsetispeed(serialprm_a, ibaud) < 0) {
    lois_log0("Error setting input baud rate on telescope serial port!");
    return(-1);
  }
  serialprm_a->c_lflag &= ~(ECHO | ICANON | IEXTEN);
  serialprm_a->c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  serialprm_a->c_oflag &= ~OPOST;
  serialprm_a->c_cflag |= CREAD;
  serialprm_a->c_cflag |= CLOCAL;

  serialprm_a->c_cc[VMIN]=0;
  serialprm_a->c_cc[VTIME]=100;

  bcopy(serialprm_a, serialprm_b, sizeof(srlprm));
  if (tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a) < 0) {
    lois_log0("Error setting telescope serial port attributes");
    return(-1);
  }

  if (tcsetattr(foc_fd, TCSANOW, serialprm_b) < 0) {
    lois_log0("Error setting focus serial port attributes");
    return(-1);
    }

  */

  if (write(foc_fd, "SYST:REM\n", 9) < 0) {
    lois_log0("Error writing command to focus serial port");
    return(-1);
  }

  for (count = 0; count < NUMTELTHREADS; count++) {
    tel_cmds[count].mutex= (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
    tel_cmds[count].go = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
    pthread_mutex_init(tel_cmds[count].mutex, NULL);
    pthread_cond_init(tel_cmds[count].go, NULL);

    tel_results[count].mutex = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_
t));
    tel_results[count].go = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
    pthread_mutex_init(tel_results[count].mutex, NULL);
    pthread_cond_init (tel_results[count].go, NULL);
    tel_results[count].value = RUN_WAIT;

    tel_block[count] = (sem_t *) ckalloc(sizeof(sem_t));
    if (sem_init(tel_block[count], 0, 0) < 0) {
      lois_log0("Error creating semaphore #%d, errno: %d\n", count, errno);
      return(-1);
    }
    if (sem_destroy(tel_block[count]) < 0) {
      lois_log0("Error destroying semaphore #%d, errno: %d\n", count, errno);
      return(-1);
    }
    for (count2 = 0; count2 < NUMPRIORITIES; count2++) {
      tel_cmds[count].p_queue[count2].head = tel_cmds[count].p_queue[count2].tai
l = NULL;
    }
    reset_curr_run(count); /* Pre-set "currently-running" command in each
                              thread to hold a pidtag of 0 and a command of 0
                              (so we won't select it during any search) */
  }

  for (count = 0; count < NUMTELTHREADS; count++) {
    pthread_create(&tel_thread[count], NULL, TEL_main, (void *) count);
  }

  /*  tel_wait_mutex= (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
  tel_wait_cond = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
  pthread_mutex_init(tel_wait_mutex, NULL);
  pthread_cond_init(tel_wait_cond, NULL);
  */
  telcmd.priority=DEF_PRIORITY;
  telcmd.t_interval=0;
  telcmd.t_times=0;
  telcmd.t_execute=0;

  return(0);
}

void * TEL_main(void * arg)

{
  int number, level;
  int look_std_prio, result;
  lois_results stat_rslt;
  char stat_str[40];

  number = (int) arg;
  /*  tel_cmds[number].num_elts = 0;
      tel_cmds[number].highprio_elts = 0; */
  tel_cmds[number].non_blocked = ~(~0 << NUMPRIORITIES);
  tel_cmds[number].non_empty = 0;
  for ( ; ; ) {
    look_std_prio = 1;
    pthread_mutex_lock(tel_cmds[number].mutex);
    while ((tel_cmds[number].non_blocked & tel_cmds[number].non_empty) == 0) {
          /*(tel_cmds[number].highprio_elts+(1-tel_cmds[number].blocked) * tel_c
mds[number].num_elts) == 0) {*/
```

```
      pthread_cond_wait(tel_cmds[number].go, tel_cmds[number].mutex);
    }
    for (level = NUMPRIORITIES-1; level > 0; level--) {
      if (tel_cmds[number].p_queue[level].head != NULL) {
        /*      tel_cmds[number].highprio_elts--; */
        shm_write((void *) &tel_cmds[number].curr_run,
                  (void *) tel_cmds[number].p_queue[level].head,
                  sizeof(tel_cmds[number].curr_run));
        ckfree((void *) tel_cmds[number].p_queue[level].head);
        if ((tel_cmds[number].p_queue[level].head = tel_cmds[number].curr_run.ne
xt) == NULL) {
          tel_cmds[number].non_empty &= ~(1 << level);
        }
        look_std_prio = 0;
        break;
      }
    }
    if (look_std_prio & tel_cmds[number].non_blocked) {
      /*      tel_cmds[number].num_elts--; */
      shm_write((void *) &tel_cmds[number].curr_run,
                (void *) tel_cmds[number].p_queue[0].head,
                sizeof(tel_cmds[number].curr_run));
      ckfree((void *) tel_cmds[number].p_queue[0].head);
      if ((tel_cmds[number].p_queue[0].head = tel_cmds[number].curr_run.next) ==
 NULL) {
        tel_cmds[number].non_empty &= ~(1 << 0);
      }
    }
    pthread_mutex_unlock(tel_cmds[number].mutex);
    if (tel_cmds[number].curr_run.command == WAIT_CMD) {

      tel_cmds[number].non_blocked &= ~(1 << 0); /* because for now we can only
                                                    block the low priority queue
 */
      printf("Tel. thread #%d halted!\n", number);
      sem_post(tel_block[number]);

      number = WAIT_CASE;
    }
    switch(number) {
      /* dispatch correct function - telescope move functions */
    case 0:
      move_aborted = 0; /* Set aborted to 0 here so that if an abort was called
while
                           no move was occurring, the flag is reset to allow the
 next
                           move to occur */
      switch(ABSOLUTE(tel_cmds[number].curr_run.command)) {
      case MOVE:
        result = move_tel(tel_cmds[number].curr_run);
        break;
      case RMOVE:
        result = rmove_tel(tel_cmds[number].curr_run);
        break;
      case LOCATE:
        result = locate_tel(tel_cmds[number].curr_run);
        break;
      default:
        lois_log1("Error: Unknown function in thread 0\n");
        result = -1;
        break;
      }
      break;
    case 1:
      /* dispatch correct function - focus move functions */
      foc_aborted = 0;
      switch(ABSOLUTE(tel_cmds[number].curr_run.command)) {
      case FOC_CTL:
        result = foc_ctl(tel_cmds[number].curr_run);
        break;
      case FOC_GO:
        result = foc_go(tel_cmds[number].curr_run);
        sprintf(telcmd.command, "log_3 {TELE: Telescope Focus %.0f}", telescope.
focus);
        lois_send(&telcmd);
        sprintf(telcmd.command, "$telestat dchar teletag 0 end");
        lois_send(&telcmd);
        break;
      default:
        lois_logl("Error: Unknown function in thread 1\n");
        result = -1;
        break;
      }
      break;
    case 2:
      /* dispatch correct function - other functions */
      lois_logl("Error: Unknown function in thread 2\n");
      result = -1;
      break;
    case 3:
      /* dispatch correct function - stop function */
      switch(ABSOLUTE(tel_cmds[number].curr_run.command)) {
      case STOP:
        result = stop_move(tel_cmds[number].curr_run);
        break;
      default:
        lois_logl("Error: Unknown function in thread 3\n");
        result = -1;
        break;
      }
      break;
    case WAIT_CASE:
      /* wait condition - do nothing */
      number = (int) arg;
      if (tel_cmds[number].curr_run.command < 0) {
        tel_cmds[number].curr_run.command *= -1;
      }
      /* Don't want to reset threads if they're blocked, otherwise we'd
         lose the pidtag of the command that is blocking the thread! */
      result = 0;
      /*      printf("Thread #%d reactivated\n", number); */
      break;
    default:
      lois_logl("Error: Unknown thread identification\n");
      break;
    }
    telcmd.priority = MAX_PRIORITY;
    sprintf(telcmd.command, "tel_status\0");
    lois_send(&telcmd);
    telcmd.priority = DEF_PRIORITY;
    /*
    if (update_tel() < 0) {
      result = -1;
    }
    */
    if (tel_cmds[number].curr_run.command < 0) {
      pthread_mutex_lock(tel_results[number].mutex);
      tel_results[number].value = result;
      pthread_cond_signal(tel_results[number].go);
      pthread_mutex_unlock(tel_results[number].mutex);
    }
    if (tel_cmds[number].curr_run.command != WAIT_CMD) {
```

```
      reset_curr_run(number);
    }
  }
}

int move_tel(input cmd_input) {

  int count, new_rahh, new_ramm, new_decdd, new_decmm, new_decss;
  int cur_rahh, cur_ramm, cur_decdd, cur_decmm, cur_decss;
  int ra_stillgoing = 1, dec_stillgoing = 1, new_dec_sign = 1, cur_dec_sign;
  float new_rass, cur_rass, cur_rapos, cur_decpos, new_rapos, new_decpos;
  float ra_slew = 1.0/15.0, dec_slew, ra_diff;
  char result[12], tel_cmd[40]=" ";

  if (cmd_input.numargs != 2) {
    lois_logl("Error: Incorrect number of arguments in MOVE\n");
    return(-1);
  }

  if ((strchr(cmd_input.args[0], ' ') != NULL) || (strchr(cmd_input.args[1], ' '
) != NULL)) {
    lois_logl("Badly formatted input string in MOVE!\n");
    lois_logl("Arg1 = %s, Arg2 = %s\n", cmd_input.args[0], cmd_input.args[1]);
    return(-1);
  }
  printf("Telescope Command: move ra: %s dec: %s\n", cmd_input.args[0], cmd_inpu
t.args[1]);

  sscanf(cmd_input.args[0], "%2d:%2d:%4f", &new_rahh, &new_ramm, &new_rass);
  sscanf(cmd_input.args[1], "%3d:%2d:%2d", &new_decdd, &new_decmm, &new_decss);

  /* Error checking - we don't have the MOVE system in the
     test module */

  if (cmd_input.args[0][0] == '-') {
    lois_logl("Error: RA hour out of range!\n");
    return(-1);
  }

  if ((new_rahh < 0) || (new_rahh > 23)) {
    lois_logl("Error: RA hour out of range!\n");
    return(-1);
  }
  if ((new_ramm < 0) || (new_ramm > 59)) {
    lois_logl("Error: RA minute out of range!\n");
    return(-1);
  }
  if ((new_rass < 0.0) || (new_rass > 59.9)) {
    lois_logl("Error: RA second out of range!\n");
    return(-1);
  }
  if ((new_decdd > 90) || (new_decdd < -90)) {
    lois_logl("Error: DEC degree out of range!\n");
    return(-1);
  }
  if ((new_decmm < 0) || (new_decmm > 59)) {
    lois_logl("Error: DEC minute out of range!\n");
    return(-1);
  }
  if ((new_decss < 0) || (new_decss > 59)) {
    lois_logl("Error: DEC second out of range!\n");
    return(-1);
  }
  if (((new_decdd == 90) || (new_decdd == -90)) &&
      ((new_decmm != 0) || (new_decss != 0))) {
    lois_logl("Error in DEC input!\n");
    return(-1);
  }

  /* Serial port configuration commands (removed in test module)

     serialprm_a->c_cc[VMIN]=1;
     serialprm_a->c_cc[VTIME]=0;

     if (tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a) < 0) {
     lois_log0("Error setting telescope serial port attribs in move_tel!");
     return(-1);
     }

     End serial port configuration commands */

  sscanf(telescope.ra, "%2d:%2d:%4f", &cur_rahh, &cur_ramm, &cur_rass);
  sscanf(telescope.dec, "%3d:%2d:%2d", &cur_decdd, &cur_decmm, &cur_decss);
  if (!(cur_dec_sign = (telescope.dec[0] == '+'))) {
    cur_dec_sign = -1;
  }

  new_rapos = (float) (new_rahh+(float) new_ramm/60.0 + new_rass/3600.0);
  cur_rapos = (float) (cur_rahh+(float) cur_ramm/60.0 + cur_rass/3600.0);
  if (cmd_input.args[1][0] == '-') {
    new_dec_sign = -1;
  }

  new_decpos=(float)(new_decdd +(float) (new_decmm*new_dec_sign)/60.0 +
                 (float)(new_decss*new_dec_sign)/3600.0);
  cur_decpos=(float)(cur_decdd +(float) (cur_decmm*cur_dec_sign)/60.0 +
                 (float)(cur_decss*cur_dec_sign)/3600.0);

  /* Echo command to file tel_fd */

  sprintf(tel_cmd, "CO %06.4f %06.4f\t", new_rapos, new_decpos);
  if (write(telvec.tel_fd, tel_cmd, strlen(tel_cmd)) < 0) {
    lois_log0("Error writing to tel_cmds file in move_tel!");
    return(-1);
  }

  /* The slewing rate is one degree in Dec per second, and 4 minutes
     of RA per second.  The ra_slew and dec_slew values can be negative
     to signal that the telescope is slewing in the negative direction */

  if ((new_decpos - cur_decpos) != 0.0) {
    dec_slew = (new_decpos - cur_decpos) / (ABSOLUTE(new_decpos - cur_decpos));
  }
  ra_diff = new_rapos - cur_rapos;
  if ((ra_diff > 12.0) || ((ra_diff < 0.0) && (ra_diff > -12.0))) {
    ra_slew *= -1.0;
  }

  if (move_delay == OFF) {
    dec_stillgoing = ra_stillgoing = 0;
    sprintf(result,"%02d:%02d:%04.1f\0", new_rahh, new_ramm, new_rass);
    strcpy(telescope.ra, result);
    sprintf(result, "%+03d:%02d:%02d\0", new_decdd, new_decmm, new_decss);
    if (new_dec_sign == -1) {
      result[0] = '-';
    }
    strcpy(telescope.dec, result);
  }
  while (dec_stillgoing || ra_stillgoing) {
    if (move_aborted) {
      /* reset serial port */
      return (-1);
```

```
    }
    if (ra_stillgoing) {
      if ((ABSOLUTE(cur_rapos - new_rapos) > ra_threshold) &&
          (ABSOLUTE(cur_rapos - new_rapos) < (24.0 - ra_threshold))) {
        cur_rapos = cur_rapos + (ra_slew/10.0);
        if (cur_rapos < 0) {
          cur_rapos +=24.0;
        } else {
          if (cur_rapos > 24.0) {
            cur_rapos -= 24.0;
          }
        }
        cur_rahh = (int) cur_rapos;
        cur_ramm = (int) (60 * (cur_rapos - (float) cur_rahh));
        cur_rass = 3600.0 * (cur_rapos - (float) cur_rahh - (float) cur_ramm/60.
0);
        if (cur_rass >= 59.95) {
          cur_rass = 0.0;
          cur_ramm++;
        }
        if (cur_ramm == 60) {
          cur_ramm = 0;
          cur_rahh++;
        }
        if (cur_rahh == 24) {
          cur_rahh = 0;
        }

        sprintf(result,"%02d:%02d:%04.1f\0", cur_rahh, cur_ramm, cur_rass);
      } else {
        ra_stillgoing = 0;
        sprintf(result,"%02d:%02d:%04.1f\0", new_rahh, new_ramm, new_rass);
      }
      strcpy(telescope.ra, result);
      sprintf(telcmd.command, "$telestat dchars ra_tag 0 end");
      lois_send(&telcmd);
      sprintf(telcmd.command, "$telestat insert ra_tag 1 {%s}", result);
      lois_send(&telcmd);
    }
    if (dec_stillgoing) {
      if (ABSOLUTE(cur_decpos - new_decpos) > dec_threshold) {
        cur_decpos = cur_decpos + (dec_slew/10.0);
        if (!(cur_dec_sign = (cur_decpos >= 0.0))) {
          cur_dec_sign = -1;
        }
        cur_decdd = (int) cur_decpos;
        cur_decmm = (int) (60 * cur_dec_sign * (cur_decpos - (float) cur_decdd))
;
        cur_decss = (int) (3600 * cur_dec_sign *
                               (cur_decpos-(float) cur_decdd-(float) (cur_decmm*cur_
dec_sign)/60.0));
        sprintf(result,"%+03d:%02d:%02d\0", cur_decdd, cur_decmm, cur_decss);
        if (cur_dec_sign == -1) {
          result[0] = '-';
        }
      } else {
        dec_stillgoing = 0;
        sprintf(result, "%+03d:%02d:%02d\0", new_decdd, new_decmm, new_decss);
        if (new_dec_sign == -1) {
          result[0] = '-';
        }
      }
      strcpy(telescope.dec, result);
      sprintf(telcmd.command, "$telestat dchars dec_tag 0 end");
      lois_send(&telcmd);
      sprintf(telcmd.command, "$telestat insert dec_tag 1 {%s}", result);
      lois_send(&telcmd);
    }
    lois_sleep(92000000);
  }

  /* Begin serial port reset configuration commands

  if (tcflush(telvec.tel_fd, TCIOFLUSH) < 0) {
    lois_log0("Error flushing telescope serial port in move_tel!");
    lois_log0("Error number %d\n", errno);
    return(-1);
  }

  serialprm_a->c_cc[VMIN] = 0;
  serialprm_a->c_cc[VTIME] = 100;

  if (tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a) < 0) {
    lois_log0("Error resetting telescope serial port in move_tel!");
    return(-1);
  }

  End serial port reconfiguration commands */

  return(0);
}

int rmove_tel(input cmd_input) {

  int decdd, decmm, decss, rahh, ramm, dec_sign, bytes;
  char ra_str[12]; char dec_str[12], move_cmd[40]=" ";
  const float pi=3.14159265358979323846;
  float RApos; /* Current RA position of telescope in degrees */
  float Decpos; /* Current DEC position of telescope in degrees */
  float RAshift; /* RA shift in degrees */
  float Decshift; /* DEC shift in degrees */
  float RAsofar=0.0, Decsofar=0.0;
  float decdep; /* current declination (affects conversion of RA
                   from time to degrees) */
  int raoff; /* RA offset move in arcs */
  int decoff; /* Dec offset move in arcs */
  float rass, ra_slew = 1.0/15.0, dec_slew = 1.0;
  float old_rapos, old_decpos;
  int ra_stillgoing = 1, dec_stillgoing = 1;

  if (cmd_input.numargs != 2) {
    lois_log1("Error: Incorrect number of args in RMOVE\n");
    return(-1);
  }

  if ((strchr(cmd_input.args[0], ' ') != NULL) || (strchr(cmd_input.args[1], ' '
) != NULL)) {
    lois_log1("Error: Badly formatted input string in RMOVE\n");
    lois_log1("Arg1: %s; Arg2: %s\n", cmd_input.args[0], cmd_input.args[1]);
    return(-1);
  }

  /* Begin telescope serial port configuration

     tcflush(telvec.tel_fd, TCIOFLUSH);
     serialprm_a->c_cc[VMIN]=1;
     serialprm_b->c_cc[VTIME]=0;
     tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a);

     End telescope serial port configuration */

  raoff = atoi(cmd_input.args[0]);
```

```
  decoff = atoi(cmd_input.args[1]);
  printf("Telescope Command: rmove ra: %d dec: %d\n", raoff, decoff);

  /* Echo command to tel_cmds file (test module) */

  sprintf(move_cmd, "RM %05d %05d\t", raoff*10, decoff*10);
  if (write(telvec.tel_fd, move_cmd, strlen(move_cmd)) < 0) {
    lois_log0("Error writing to tel_cmds file in rmove_tel!");
    return(-1);
  }

  /* convert RA position to degrees, convert RAshift and Decshift values
     to degrees, compute sign of DEC value */

  sscanf(telescope.dec, "%3d:%2d:%2d", &decdd, &decmm, &decss);

  if (!(dec_sign = (telescope.dec[0] == '+'))) {
    dec_sign = -1;
  }

  Decpos=(float)(decdd +(float) (decmm*dec_sign)/60.0 +
                 (float)(decss*dec_sign)/3600.0);
  decdep = cos((double) (Decpos*(pi/180.0)));
  sscanf(telescope.ra, "%2d:%2d:%4f", &rahh, &ramm, &rass);
  RApos= (float) (rahh+(float)ramm/60.0+ rass/3600.0);
  RAshift = raoff/(15.0*3600.0*decdep);
  Decshift = decoff/3600.0;

  if (((Decpos + Decshift) < -90.0) || ((Decpos + Decshift) > 90.0)) {
    lois_log1("New Dec out of range!\n");
    return(-1);
  }

  if (RAshift < 0) {
    ra_slew *= -1.0;
  }

  if (Decshift < 0) {
    dec_slew = -1.0;
  }

  if (move_delay == OFF) {
    dec_stillgoing = ra_stillgoing = 0;
    /* convert Decpos to DD:MM:SS and RApos to HH:MM:SS */

    Decpos = Decpos + Decshift;

    if (!(dec_sign = (Decpos >= 0.0))) {
      dec_sign = -1;
    }

    decdd = (int) Decpos;
    decmm = (int) (60 * dec_sign * (Decpos-(float) decdd));
    decss = (int) (3600 * dec_sign *
                  (Decpos-(float) decdd- (float) (decmm*dec_sign)/60.0));

    sprintf(dec_str, "%+03d:%02d:%02d\0", decdd, decmm, decss);
    if ((decdd == 0) && (dec_sign == -1)) {
      /* case where you're between -1 and 0 degrees declination */
      dec_str[0] = '-';
    }

    RApos = RApos + RAshift;
    while (RApos >= 24.0) {
      RApos -= 24.0;
    }
    while (RApos < 0.0) {
      RApos += 24.0;
    }

    rahh = (int) RApos;
    ramm = (int) (60 * (RApos - (float) rahh));
    rass = 3600.0 * (RApos-(float) rahh - (float) ramm/60.0);
    if (rass >= 59.95) {
      rass = 0.0;
      ramm++;
    }
    if (ramm == 60) {
      ramm = 0;
      rahh++;
    }
    if (rahh == 24) {
      rahh = 0;
    }
    sprintf(ra_str, "%02d:%02d:%04.1f\0", rahh, ramm, rass);
    strncpy(telescope.dec, dec_str, sizeof(dec_str));
    strncpy(telescope.ra, ra_str, sizeof(ra_str));
  } else {
    old_rapos = RApos;
    old_decpos = Decpos;
  }

  while (dec_stillgoing || ra_stillgoing) {
    if (move_aborted) {
      /* reset serial port */
      return (-1);
    }
    if (dec_stillgoing) {
      if (ABSOLUTE(Decsofar - Decshift) > dec_threshold) {
        Decsofar = Decsofar + (dec_slew/10.0);
        Decpos = Decpos + (dec_slew/10.0);
      } else {
        dec_stillgoing = 0;
        Decpos = old_decpos+Decshift;
      }
      if (!(dec_sign = (Decpos >= 0.0))) {
        dec_sign = -1;
      }

      /* Correction for converting floating value Decpos to integer value dec
         seconds */

      Decpos += (dec_sign * 0.01/3600.0);
      decdd = (int) Decpos;
      decmm = (int) (60 * dec_sign * (Decpos - (float) decdd));
      decss = (int) (3600 * dec_sign *
                     (Decpos -(float) decdd -(float) (decmm*dec_sign)/60.0));

      sprintf(dec_str, "%+03d:%02d:%02d\0", decdd, decmm, decss);
      if ((decdd == 0) && (dec_sign == -1)) {
        dec_str[0] = '-';
      }
      strncpy(telescope.dec, dec_str, sizeof(dec_str));
      sprintf(telcmd.command, "$telestat dchars dec_tag 0 end");
      lois_send(&telcmd);
      sprintf(telcmd.command, "$telestat insert dec_tag 1 (%s)", dec_str);
      lois_send(&telcmd);
    }
    if (ra_stillgoing) {
      if (ABSOLUTE(RAsofar - RAshift) > ra_threshold) {
        RAsofar = RAsofar + (ra_slew/10.0);
        RApos = RApos + (ra_slew/10.0);
```

```
      } else {
        ra_stillgoing = 0;
        RApos = old_rapos+RAshift;
      }

      while (RApos >= 24.0) {
        RApos -= 24.0;
      }
      while (RApos < 0.0) {
        RApos += 24.0;
      }
      rahh = (int) RApos;
      ramm = (int) (60 * (RApos - (float) rahh));
      rass = 3600.0 * (RApos-(float) rahh - (float) ramm/60.0);
      if (rass >= 59.95) {
        rass = 0.0;
        ramm++;
      }
      if (ramm == 60) {
        ramm = 0;
        rahh++;
      }
      if (rahh == 24) {
        rahh = 0;
      }
      sprintf(ra_str, "%02d:%02d:%04.1f\0", rahh, ramm, rass);
      strncpy(telescope.ra, ra_str, sizeof(ra_str));
      sprintf(telcmd.command, "$telestat dchars ra_tag 0 end");
      lois_send(&telcmd);
      sprintf(telcmd.command, "$telestat insert ra_tag 1 (%s)", ra_str);
      lois_send(&telcmd);
    }
    lois_sleep(92000000);
  }
  /* Begin telescope serial port reconfiguration

     tcflush(telvec.tel_fd, TCIOFLUSH);
     serialprm_a->c_cc[VMIN] = 0;
     serialprm_a->c_cc[VTIME] = 100;
     tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a);

     End telescope serial port reconfiguration */

  return(0);
}

int locate_tel(input cmd_input) {
return(0);
}

int foc_ctl(input cmd_input) {

  float oldfocus, newfocus, focus_rate;
  int sec;
  char move_cmd[40];
  cmd_struct foc_cmd;

  foc_cmd.priority=DEF_PRIORITY;
  foc_cmd.t_interval=0;
  foc_cmd.t_times=0;
  foc_cmd.t_execute=0;

  /* going to implement a routine to simulate a stepper motor that
     goes from -30000 to 30000.  Slow speed moves 10 steps per second,
     fast speed moves 1000 steps per second */
  /* Begin telescope serial port configuration

     tcflush(telvec.tel_fd, TCIOFLUSH);
     serialprm_a->c_cc[VMIN]=1;
     serialprm_b->c_cc[VTIME]=0;
     tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a);

     End telescope serial port configuration */

  if (cmd_input.numargs != 2) {
    lois_log1("Error: Incorrect number of args in FOC_CTL\n");
    return(-1);
  }

  if ((strchr(cmd_input.args[0], ' ') != NULL) || (strchr(cmd_input.args[1], ' '
) != NULL)) {
    lois_log1("Error: Badly formatted input string in FOC_CTL\n");
    lois_log1("Arg1: %s; Arg2: %s\n", cmd_input.args[0], cmd_input.args[1]);
    return(-1);
  }

  sec = atoi(cmd_input.args[1]);

  printf("Telescope Command: focus_ctl spd: %s time: %d\n", cmd_input.args[0], s
ec);

  oldfocus=telescope.focus;

  /* Echo command to tel_cmds file and calculate new focus (test module) */

  if (strcmp(cmd_input.args[0], "slow") == 0) {
    newfocus = telescope.focus + (float) sec;
    focus_rate = 1.0;
    sprintf(move_cmd, "FC 0 %03d\t", sec);
  } else {
    newfocus = telescope.focus + 100.0* (float) sec;
    focus_rate = 100.0;
    sprintf(move_cmd, "FC 1 %03d\t", sec);
  }

  if (write(telvec.tel_fd, move_cmd, strlen(move_cmd)) < 0) {
    lois_log0("Error writing to tel_cmds file in foc_ctl");
    return(-1);
  }

  if ((newfocus > 30000.0) || (newfocus < -30000.0)) {
    lois_log1("Error: Resulting focus is out of range\n");
    return(-1);
  }

  if (sec < 0) {
    sec *= -1;
    focus_rate *= -1.0;
  }

  if (focus_delay == OFF) {
    telescope.focus = newfocus;
  }
  while (ABSOLUTE(telescope.focus - newfocus) > 0.5) {
    if (foc_aborted) {
      lois_log1("Focus change operation aborted!");
      return(-1);
    }
    telescope.focus += focus_rate;
    sprintf(telcmd.command, "$telestat dchars foc_tag 0 end");
    lois_send(&telcmd);
```

```
    sprintf(telcmd.command, "$telestat insert foc_tag 1 {%.1f steps (-30000 to 3
0000)}",
            telescope.focus);
    lois_send(&telcmd);
    lois_sleep(96000000);
  }

  /* tcflush(telvec.tel_fd, TCIOFLUSH); */

  telescope.focus = newfocus;

  return(0);
}

int foc_go(input cmd_input) {

  int fasttime=0, slowtime=0, pidtag;
  float newpos, diffvalue;
  input send_to_ctl;
  lois_results stat_rslt;

  send_to_ctl.command = FOC_CTL;
  send_to_ctl.numargs = 2;
  sprintf(send_to_ctl.args[0], "fast\0");

  /* Begin telescope serial port configuration

     tcflush(telvec.tel_fd, TCIOFLUSH);
     serialprm_a->c_cc[VMIN]=1;
     serialprm_b->c_cc[VTIME]=0;
     tcsetattr(telvec.tel_fd, TCSANOW, serialprm_a);

     End telescope serial port configuration */


  if (cmd_input.numargs != 1) {
    lois_log1("Error: Incorrect number of args in FOC_GO\n");
    return(-1);
  }

  if (strchr(cmd_input.args[0], ' ') != NULL) {
    lois_log1("Error: Badly formatted input in FOC_GO\n");
    lois_log1("Arg1: %s\n", cmd_input.args[0]);
    return(-1);
  }

  newpos = atof(cmd_input.args[0]);
  if (newpos > 30000.0 || newpos < -30000.0) {
    lois_log1("Focus Position must be between -30000 and 30000\n");
    return(-1);
  }

  printf("Telescope command: foc_go newpos = %.0f\n", newpos);

  telcmd.priority = -1 * MAX_PRIORITY;
  sprintf(telcmd.command, "tel_status\0");
  pidtag = lois_send(&telcmd);
  lois_receive(pidtag, &stat_rslt);
  if (stat_rslt.rtn_status != 0) {
    lois_log1("Error in tel_status, in focus_go.");
    return(-1);
  }
  telcmd.priority = DEF_PRIORITY;

  /*
  if (update_tel() < 0) {
    lois_log1("Error in update_tel, in focus_go\n");
    return(-1);
  }
  */

  sprintf(telcmd.command, "$telestat dchar teletag 0 end");
  lois_send(&telcmd);
  sprintf(telcmd.command, "$telestat insert teletag 0 {Changing Focus}");
  lois_send(&telcmd);

  if (telescope.focus < newpos ) {
    diffvalue=newpos-telescope.focus;
    fasttime = (int) (diffvalue/100.0);
    sprintf(send_to_ctl.args[1], "%d\0", fasttime);
  } else {
    diffvalue=telescope.focus-newpos;
    fasttime = (int) (diffvalue/100.0);
    sprintf(send_to_ctl.args[1], "-%d\0", fasttime);
  }

  if (foc_ctl(send_to_ctl) < 0) {
    lois_log1("Error in focus_ctl subroutine of focus_go, fast speed\n");
    return(-1);
  }

  telcmd.priority = -1 * MAX_PRIORITY;
  sprintf(telcmd.command, "tel_status\0");
  pidtag = lois_send(&telcmd);
  lois_receive(pidtag, &stat_rslt);
  if (stat_rslt.rtn_status != 0) {
    lois_log1("Error in tel_status, in focus_go.");
    return(-1);
  }
  telcmd.priority = DEF_PRIORITY;

  /*
  if (update_tel() < 0) {
    lois_log1("Error in update_tel, in focus_go\n");
    return(-1);
  }
  */

  /*     shm_read((void *)&telescope, (char *)telvec.memory+sizeof(telvec),
         sizeof(telescope)); */

  sprintf(send_to_ctl.args[0], "slow\0");
  if (telescope.focus < newpos ) {
    diffvalue=newpos-telescope.focus;
    slowtime = (int) diffvalue;
    sprintf(send_to_ctl.args[1], "%d\0", slowtime);
  } else {
    diffvalue=telescope.focus-newpos;
    slowtime = (int) diffvalue;
    sprintf(send_to_ctl.args[1], "-%d\0", slowtime);
    }
  if (foc_ctl(send_to_ctl) < 0) {
    lois_log1("Error in focus_ctl subroutine of focus_go, slow speed\n");
    return(-1);
  }

  /*  shm_read((void *)&telescope, (char *)telvec.memory+sizeof(telvec),
      sizeof(telescope));    */

  sprintf(telmsg,"TELE: Slowtime:%d Fasttime:%d", slowtime, fasttime);
  lois_log3(telmsg);
  return(0);
```

```
}

int stop_move(input cmd_input) {

  if (cmd_input.numargs != 1) {
    lois_log1("Error: Incorrect number of args in STOP_TEL\n");
    return(-1);
  }

  if (strchr(cmd_input.args[0], ' ') != NULL) {
    lois_log1("Error: Badly formatted input string in STOP_TEL\n");
    lois_log1("Arg1: %s\n", cmd_input.args[0]);
    return(-1);
  }

  printf("Telescope Command: tel_stop %s\n", cmd_input.args[0]);
  if (strcmp(cmd_input.args[0], "move") == 0) {
    move_aborted = 1;
  } else {
    if (strcmp(cmd_input.args[0], "focus") == 0) {
      foc_aborted = 1;
    } else {
      move_aborted = 1;
      foc_aborted = 1;
    }
  }

  return(0);
}


int update_tel() {

  /* This routine is here so that when focus_go is run, or when a user
     wants a commands results back, the command doesn't have to wait for
     the interpreter to run the tel_status command.  If we needed command
     results, then the results would have to be returned (to free up the
     interpreter) before the tel_status routine is run.  The code is
     almost identical to the tel_status command.
     This routine was (I think) rendered obsolete with the introduction of
     the dual interpreter system.  Now, if a command needs results, the command
     interpreter is blocked, but the main interpreter can still run the tel_stat
us
     routine.  As far as focus_go is concerned, since only graphics commands are
 on
     the main interpreter, waiting for a tel_status command should not take very
 long
  */

  static char tel_string[50]="";
  int day_of_year, year_no, ra_hr, ra_min;
  int lst_hr, lst_min, lst_sec, ha_hr, ha_min, ha_sec;
  int bytes, nbytes, tmp_bytes;
  char val_str[8];
  float dec_day, flt_lst, j2000, ra_sec= 0.0, flt_ra, flt_ha;
  time_t cur_time;
  struct tm *cur_date;
  cmd_struct stat_cmd;

  stat_cmd.priority = DEF_PRIORITY;
  stat_cmd.t_interval = 0;
  stat_cmd.t_execute = 0;
  stat_cmd.t_times = 0;

  /* tcflush(telvec.tel_fd, TCIOFLUSH); */
  if (write(telvec.tel_fd, "TS\r\n", 4) < 0) {
    lois_log0("Error writing to telescope serial port in update_tel!");
    return(TCL_ERROR);
  }

  /* Begin telescope serial port configuration

     bytes = read(telvec.tel_fd, tel_string, 1);
     if (bytes == 0) {
     lois_log1("TELE: Command Did Not Acknowledge!");
     return(TCL_ERROR);
     }
     bytes = read(telvec.tel_fd, tel_string, 46);
     while (bytes < 46) {
     nbytes = read(telvec.tel_fd, (char *) tel_string+bytes, 46-bytes);
     bytes += nbytes;
     }
     tmp_bytes = bytes;

     End telescope serial port reconfiguration */

  cur_time = time(NULL);
  cur_date = gmtime(&cur_time);

  /* compute fraction of day elapsed and day of year*/

  dec_day = (float) ((cur_date->tm_hour + cur_date->tm_min / 60.0 +
                      cur_date->tm_sec/3600.0) / 24.0);

  day_of_year = cur_date->tm_yday+1;

  /* compute number of days since J2000 */

  if ((year_no = cur_date->tm_year-100) < -2) { /* Y2K compliance */
    year_no += 100;
  }

  j2000 = (float) (year_no*365 + (ABSOLUTE(year_no))/4 + day_of_year) - 1.5 +
    dec_day;

  /* compute local sidereal time see www.xylem.demon.co.uk/kepler/altaz.html
     for calculations */

  flt_lst = (100.46 + 0.985647*j2000 + test_long + 15.0*dec_day*24.0)/15.0;

  while (flt_lst > 24.0) {
    flt_lst -= 24.0;
  }

  while (flt_lst < 0.0) {
    flt_lst += 24.0;
  }

  lst_hr = (int) flt_lst;
  lst_min = (int) (60 * (flt_lst - (float) lst_hr));
  lst_sec = (int) (3600 * (flt_lst - (float) lst_hr - (float) lst_min/60.0));

  sprintf(val_str, "%02d:%02d:%02d\0", lst_hr, lst_min, lst_sec);
  strcpy(telescope.lst, val_str);

  /* compute hour angle (LST - RA) */

  sscanf(telescope.ra, "%2d:%2d:%4f", &ra_hr, &ra_min, &ra_sec);

  flt_ra = (float) (ra_hr+ (float) ra_min/60.0+ (float) ra_sec/3600.0);
  if ((flt_ha = flt_lst - flt_ra) < 0) {
```

```
    flt_ha += 24.0;
  }

  ha_hr = (int) flt_ha;
  ha_min = (int) (60 * (flt_ha - (float) ha_hr));
  ha_sec = (int) (3600 * (flt_ha - (float) ha_hr - (float) ha_min/60.0));

  sprintf(val_str, "%02d:%02d:%02d\0", ha_hr, ha_min, ha_sec);
  strcpy(telescope.ha, val_str);

  sprintf(val_str, "%02d:%02d:%02d\0", cur_date->tm_hour, cur_date->tm_min,
          cur_date->tm_sec);
  strcpy(telescope.ut, val_str);
  sprintf(val_str, "%02d/%02d/%02d\0", cur_date->tm_year % 100, cur_date->tm_mon
+1,
          cur_date->tm_mday);
  sprintf(stat_cmd.command, "$telestat dchars stat_tag 0 end");
  lois_send(&stat_cmd);
  sprintf(stat_cmd.command, "$telestat insert dec_tag 1 {%s}", telescope.dec);
  lois_send(&stat_cmd);
  sprintf(stat_cmd.command, "$telestat insert ra_tag 1 {%s}", telescope.ra);
  lois_send(&stat_cmd);
  sprintf(stat_cmd.command, "$telestat insert lst_tag 1 {%s}", telescope.lst);
  lois_send(&stat_cmd);
  strcpy(telescope.date, val_str);
  sprintf(stat_cmd.command, "$telestat insert date_tag 1 {%s %s}",telescope.date
,
          telescope.ut);
  lois_send(&stat_cmd);
  sprintf(stat_cmd.command, "$telestat insert ha_tag 1 {%s}", telescope.ha);
  lois_send(&stat_cmd);
  sprintf(stat_cmd.command, "$telestat insert am_tag 1 {%s}", telescope.airmass)
;
  lois_send(&stat_cmd);

  sprintf(stat_cmd.command, "$telestat insert dome_tag 1 {%s}", telescope.dome);
  lois_send(&stat_cmd);

  sprintf(stat_cmd.command, "$telestat insert epoch_tag 1 {%s}", telescope.epoch
);
  lois_send(&stat_cmd);

  sprintf(stat_cmd.command, "$telestat dchars foc_tag 0 end");
  lois_send(&stat_cmd);
  /*  sprintf(stat_cmd.command, "$telestat insert foc_tag 1 {%.2f}", telescope.f
ocus); */
  sprintf(stat_cmd.command, "$telestat insert foc_tag 1 {%.0f steps (-30000 to 3
0000)}", telescope.focus);
  lois_send(&stat_cmd);

  if (update_foc() < 0) {
    /*  tcflush(telvec.tel_fd, TCIOFLUSH);
        write(telvec.tel_fd, "\r", 1); */
    return(-1);
  }

  /*  tcflush(telvec.tel_fd, TCIOFLUSH);
      write(telvec.tel_fd, "\r", 1); */

  shm_write((char *)telvec.memory+sizeof(telvec), (void *)&telescope,
            sizeof(telescope));
  return(TCL_OK);

}

int update_foc() {
  int bytes;
  char focstr[10];

  /* Begin telescope serial port communication

     tcflush(foc_fd, TCIOFLUSH);

     write(foc_fd, ":MEAS:VOLT:DC? 10,0.001\n", 24);
     bytes=read(foc_fd, (void *) focstr, sizeof(focstr));
     telescope.focus=atof(focstr);

     tcflush(foc_fd, TCIOFLUSH);

     End Telescope serial port communication */

  if (write(foc_fd, "GET_FOCUS\n", 10) < 0) {
    lois_log0("Error writing to focus serial port in getfoc!");
    return(-1);
  }

  return(0);

}

static void reset_curr_run(int number) {

  tel_cmds[number].curr_run.pidtag = 0;
  tel_cmds[number].curr_run.command = 0;
}
```

```
/***************************************************************************
                        Lowell Observatory
                  CCD Acquisitions Software Package


Description:
   --------------
Module:          testinst.so
Called From:     loas.e
File Name:       $RCSfile: testinst.c,v $
Started:         08/04/98
Revision:        $Revision: 1.9.4.9 $
Last Revised:    $Date: 1999/05/24 02:31:32 $
By:              $Name:  $

Included in:     LOIS loadable module

Explanation:     This program is a test module
                 for the instrument part of the
                 LOIS system.

        Copyright 1998
, Lowell Observatory, All Rights Reserved

---------------------------------------------------------------------------
Change Log:

$Log: testinst.c,v $
Revision 1.9.4.9  1999/05/24 02:31:32  agould
Updating inst_remove and inst_flush to account for new abort_cmd feature

Revision 1.9.4.7  1999/05/20 02:44:12  agould
Adding functions to abort instrument move

Revision 1.9.4.6  1999/05/19 00:04:15  agould
Added commands to help remove elements from an instrument module thread queue

Revision 1.9.4.5  1999/04/22 21:11:57  agould
Merging changes from branch rel_1_1b for dual interpreter system and dynamic que
ues

Revision 1.9.4.4.2.2  1999/04/15 18:16:05  agould
Moved inst_block routines to main interpreter.

Revision 1.9.4.4.2.1  1999/04/08 01:16:06  agould
Configured instrument module for use with two interpreter system

Revision 1.9.4.4  1999/03/16 19:33:46  agould
Static allocation for local structures

Revision 1.9.4.3  1999/03/04 22:26:49  agould
Linux compatibility

Revision 1.9.4.1  1999/01/22 18:18:31  taylor
Adding thread module domain.

Revision 1.9  1998/10/06 18:10:07  taylor
Replaced tel_cmd with inst_cmd in the Obj String init.

Revision 1.8  1998/10/06 17:39:05  taylor
Moved the Tcl String Object initialization into the library load
routine.

Revision 1.1  1998/10/06 16:23:55  taylor
Initial revision
```

```
Revision 1.6  2018/10/02 22:57:49  agould
Avoid segmentation faults by starting GUI before calling routines in test
module.

Revision 1.5  2018/10/02 22:36:47  agould
Running instrument test module with lowell10_init script.  Seems to be
working, if module is loaded from command line in wish.  Causes segmentation
fault if loaded from initializing script.  Also, when changing filters,
if command is entered at command line, button stays green until clicked on.

Revision 1.4  2018/10/02 20:52:01  agould
Trying to test module.

Revision 1.3  2018/09/26 01:34:46  agould
Replaced all calls to tel_fw with sleep calls.  Sleep
parameters may need to be changed.  Total of two calls
to tel_fw.

Revision 1.2  2018/09/26 01:22:32  agould
Removed filter wheel commands, still unsure if I need to remove
instrument GUI commands.

Revision 1.1  2018/09/26 01:06:37  agould
Initial revision

Revision 1.1  2018/09/25 22:22:01  agould
Initial revision

Revision 1.1  1998/09/25 19:02:32  taylor
Initial revision

---------------------------------------------------------------------------
$Id: testinst.c,v 1.9.4.9 1999/05/24 02:31:32 agould Exp $
***************************************************************************/

/* Network Includes */
#include <netinet/in.h>

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#define _REENTRANT
#define _P __P
#endif

#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <syslog.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <termios.h>

/* Tcl/Tk Includes */
#include <tcl.h>
```

```
#include <tk.h>

/* CDL Library Include */
#include <cdl.h>

/* LOIS Include */
#include <lois.h>

/* Instrument Include */
#include <LowellInst.h>
#include <ModThrd.h>

int GUI_state, wr_err;
int filt_delay = ON;
static long old_state[NUMINSTTHREADS]; /* long variables to store and restore bl
ocked states
                                          of threads */
static pthread_mutex_t *actv_block[NUMINSTTHREADS]; /* Need these extra blocks t
o prevent
                                                     an activate from occurring
 while you're
                                                     removing commands.  Lockin
g the instrument
                                                     module thread queues is no
t a good solution
                                                     because it prevents comman
ds from being
                                                     written to the queues duri
ng a remove call,
                                                     and we may want to allow c
ommands to be
                                                     written during a remove */

inst_struct instrument;
struct instrument_vectors instvec;
static info_struct info;

#define NUMKEYS 3

static char instkeywords[NUMKEYS][9]={
        "INSTRUM\0",
        "FILTERS\0",
        "FILTNAME\0",
};
static char instcomments[NUMKEYS][71]={
        "instrument name\0",
        "Filter Poisiton\0",
        "Filter Name\0",
};

static char filt_name[30];

Tcl_Obj  *instkeys[NUMKEYS*3];
Tcl_Obj         *inst_objcmd;
/*
*
* Script Command Functions Defined
*
*/
int testinst_init (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int testinst_delay (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[]);
int testinst_activate (ClientData clientdata, Tcl_Interp *interp,
                       int objc, Tcl_Obj *CONST objv[]);
int testinst_status (ClientData clientdata, Tcl_Interp *interp,
```

```
                     int objc, Tcl_Obj *CONST objv[]);
int testinst_stop (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[]);
int testinst_filter (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int testinst_calibrate (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int testinst_header (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int testinst_sims (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[]);
/*
 *
 * Loadable Module Init called from Tcl/Tk load
 *
 */
int Testinst_Init(Tcl_Interp *interp)
{
  int count;
  static char inst_cmd[80];
  Tcl_Interp *cmd_interp;

  /*
   *
   * Tcl/Tk Command definitions
   *
   */

  if ((cmd_interp = Tcl_GetSlave(interp, "command")) == NULL) {
    lois_log0("Error getting command interpreter!\n");
    return(TCL_ERROR);
  }

  Tcl_CreateObjCommand(interp, "inst_init", testinst_init,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "inst_block", testinst_delay,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "inst_activate", testinst_activate,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "inst_status", testinst_status,
                       (ClientData) (NULL), (Tcl_CmdDeleteProc *) NULL);
  Tcl_CreateObjCommand(interp, "inst_stop", testinst_stop,
                       (ClientData) (NULL), (Tcl_CmdDeleteProc *) NULL);
  Tcl_CreateObjCommand(cmd_interp, "filter", testinst_filter,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(cmd_interp, "inst_calib", testinst_calibrate,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "inst_header", testinst_header,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "inst_simdelays", testinst_sims,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  /*
   *
   * Tcl/Tk Package Provide and Revision
   *
   */

  Tcl_PkgProvide(interp, "Testinst", "1.0");
  Tcl_PkgProvide(cmd_interp, "Testinst", "1.0");
  /* Initialize the shared memory mapping for the instrument vectors and
   * structures
   */

#ifdef __LINUX__

  if ((instvec.mem_fd=shmget(INST_KEY, sizeof(instvec) + sizeof(instrument),
                                (SHM_R|SHM_W))) < 0) {
    lois_log0("Cannot open instrument shared memory buffer");
    printf("Error number %d\n", errno);
    return(TCL_ERROR);
  }
  instvec.memory=shmat(instvec.mem_fd, 0, 0);

  if ((info.mem_fd = shmget(INFO_KEY, sizeof(info), (SHM_R|SHM_W))) < 0) {
    lois_log0("Cannot open information shared memory buffer");
    printf("Error number %d\n", errno);
    return(TCL_ERROR);
  }
  info.memory=shmat(info.mem_fd, 0, 0);

  if (instvec.memory == (void *) -1) {
    lois_log0("Memory Map failed for instrument buffer");
    return(TCL_ERROR);
  }

  if (info.memory == (void *) -1) {
    lois_log0("Memory map failed for information buffer");
    return(TCL_ERROR);
  }

#elif defined(__SOLARIS_5x__)

  if ((instvec.mem_fd=shm_open("/instrument", O_RDWR, S_IRWXU)) < 0) {
    lois_log0("Cannot open instrument shared memory buffer");
    printf("Error number %d\n", errno);
    return(TCL_ERROR);
  }

  if ((info.mem_fd = shm_open("/information", O_RDWR, S_IRWXU)) < 0) {
    lois_log0("Cannot open information shared memory buffer");
    printf("Error number %d\n", errno);
    return(TCL_ERROR);
  }

  instvec.memory = mmap(NULL, sizeof(instvec) + sizeof(instrument),
                        PROT_READ | PROT_WRITE, MAP_SHARED, instvec.mem_fd, 0);

  info.memory = mmap(NULL, sizeof(info), PROT_READ | PROT_WRITE,
                     MAP_SHARED, info.mem_fd, 0);

  if (instvec.memory == NULL) {
    lois_log0("Memory Map failed for instrument buffer");
    return(TCL_ERROR);
  }

  if (info.memory == NULL) {
    lois_log0("Memory map failed for information buffer");
    return(TCL_ERROR);
  }
#endif
  shm_write(instvec.memory, (void *) &instvec, sizeof(instvec));
  shm_write((char *) instvec.memory+sizeof(instvec), (void *) &instrument,
            sizeof(instrument));
  shm_read((void *) &info, info.memory, sizeof(info));
  strcpy(info.instmod, "testinst\0");
  shm_write(info.memory, (void *) &info, sizeof(info));

  for (count =0 ; count < NUMKEYS*3 ; count++)
    instkeys[count]=Tcl_NewStringObj(inst_cmd, sizeof(inst_cmd));

  inst_objcmd=Tcl_NewStringObj(inst_cmd, sizeof(inst_cmd));
```

```
  if (Init_InstThreadInfo() < 0) {
    printf("Error in thread initialization routine!\n");
    return(TCL_ERROR);
  }

  for (count = 0; count < NUMINSTTHREADS; count++) {
    actv_block[count] = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(actv_block[count], NULL);
  }

  lois_log4("Instrument: Test module ver 1.0");
  return(TCL_OK);

}

int testinst_init (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
{
  int   count, ibaud, obaud, bytes;
  char  inst_cmd[80],on='1';

  GUI_state=1;

  if (objc > 1) {
    for (count=1; count < objc; count++) {
      if (strcmp (objv[count]->bytes, "-nogui") == 0)
        GUI_state=0;
    }
  }

  sprintf(inst_cmd, "source %s/scripts/testinst.tcl\0", info.loishome);
  Tcl_SetStringObj(inst_objcmd, inst_cmd,-1);
  Tcl_EvalObj(interp, inst_objcmd);

  if (GUI_state) test_gui (clientdata, interp, objc, objv);

  /* reset filter wheel to position 1 */

  instrument.filt_pos = 1;
  shm_write((char *) instvec.memory+sizeof(instvec), (void *) &instrument,
            sizeof(instrument));
  if (testinst_status(clientdata, interp, objc, objv) != TCL_OK) {
    return(TCL_ERROR);
  }
  return(TCL_OK);

}

int testinst_delay (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[])
{
  input pipe_send;
  int count, num=0, priority = 1, to_be_blocked[NUMINSTTHREADS];
  char blocked_threads[10]="", tmp_str[5], ret_str[80];

  pipe_send.command = WAIT_CMD;
  pipe_send.numargs = 1;
  pipe_send.pidtag = get_curevalpidtag();

  if (objc > 2) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-2], NULL), "-prio") == 0) {
      if (Tcl_GetIntFromObj(interp, objv[objc-1], &priority) == TCL_ERROR) {
        Tcl_SetResult(interp, "Error getting priority for inst_block command!",
NULL);
        return(TCL_ERROR);
      }
      objc -= 2;
    }
  }

  if (objc > 1) {
    if (strcmp(Tcl_GetStringFromObj(objv[objc-1], NULL), "-subc") == 0) {
      pipe_send.pidtag *= -1;
      objc--;
    }
  }

  if (objc > 1) {
    for (count = 1; count < objc; count++) {
      if (strcmp(objv[count]->bytes, "filter") == 0) {
        to_be_blocked[num] = 0;
        num++;
      }
    }
  } else {
    for (count = 0; count < NUMINSTTHREADS; count++) {
      to_be_blocked[num] = count;
      num++;
    }
  }

  for (count = 0; count < num; count++) {
    sprintf(pipe_send.args[0], "%d\0", to_be_blocked[count]);
    if (inst_write(pipe_send, priority) < 0) {
      sprintf(interp->result, "Error sending delay command to instrument thread
#%d\nError: %s\n",
              to_be_blocked[count], write_errs[wr_err]);
      return(TCL_ERROR);
    }
    sprintf(tmp_str, "%d ", count);
    strcat(blocked_threads, tmp_str);
  }

  sprintf(ret_str, "log_4 {Instrument threads %shave been blocked}", blocked_thr
eads);
  Tcl_Eval(interp, ret_str);
  sprintf(interp->result, "%d", pipe_send.pidtag);
  return(TCL_OK);
}

int testinst_activate (ClientData clientdata, Tcl_Interp *interp,
                       int objc, Tcl_Obj *CONST objv[])
{
  int count;
  char err_str[80];

  for (count = 0; count < NUMINSTTHREADS; count++) {
    if (sem_trywait(inst_block[count]) != 0) {
      if (errno != EAGAIN) {
        sprintf(err_str, "Error reinitializing semaphore #%d in inst_activate, e
rrno: %d",
                count, errno);
        lois_log0(err_str);
      }
    }
  }

  for (count = 0; count < NUMINSTTHREADS; count++) {
    pthread_mutex_lock(inst_cmds[count].mutex);
    if (pthread_mutex_trylock(actv_block[count]) < 0) {
      if (errno == EBUSY) {
        Tcl_SetResult(interp, "Cannot activate thread while removing command!",
```

```
NULL);
      } else {
        sprintf(err_str, "Error in trylock, inst_activate. Error no %d\n", errno
);
        Tcl_SetResult(interp, err_str, TCL_VOLATILE);
        pthread_mutex_unlock(actv_block[count]);
      }
      return(TCL_ERROR);
    }
    inst_cmds[count].non_blocked |= 1; /* so that lowest priority queue is
                                          now unblocked */
    pthread_cond_signal(inst_cmds[count].go); /* If thread was sleeping but comm
ands were
                                                  in its queue */
    pthread_mutex_unlock(actv_block[count]);
    pthread_mutex_unlock(inst_cmds[count].mutex);
  }

  sprintf(interp->result, "Instrument threads have been reactivated\n");
  return(TCL_OK);
}

int test_gui (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  if (GUI_state) {
    Tcl_SetStringObj(inst_objcmd, "testinst_init\0",-1);
    Tcl_EvalObj(interp, inst_objcmd);
    GUI_state=1;
    sprintf(interp->result, "Test Instrument GUI Opened");
    return(TCL_OK);
  } else {
    sprintf(interp->result, "Test Instrument GUI Already Opened");
    return(TCL_ERROR);

  }

}

int testinst_status (ClientData clientdata, Tcl_Interp *interp,
                     int objc, Tcl_Obj *CONST objv[])
{

  char str_pos[5];

  /*  shm_read((void *) &instrument, (char *) instvec.memory+sizeof(instvec),
           sizeof(instrument));
  sprintf(str_pos, "%d\0", instrument.filt_pos);
  Tcl_SetVar(interp, "c_filter", str_pos, TCL_GLOBAL_ONLY);  */
  return(TCL_OK);

}

int testinst_stop (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
{

  int bytes, priority = 3;
  char err_str[40];
  input pipe_send;

  switch (objc) {
  case 1:
    break;
  case 2:
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: inst_stop [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, TCL_VOLATILE);
      return(TCL_ERROR);
    }
    priority = -3;
    break;
  case 3:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      Tcl_SetResult(interp, "inst_stop [-prio priority]", NULL);
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      Tcl_SetResult(interp, "Error getting priority value!", NULL);
      return(TCL_ERROR);
    }
    break;
  default:
    Tcl_SetResult(interp, "Usage: inst_stop", NULL);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = INST_STOP;
  pipe_send.numargs = 0;
  pipe_send.pidtag = get_curevalpidtag();
  if (inst_write(pipe_send, priority) < 0) {
    sprintf(err_str, "Error sending abort command\nError: %s\n", write_errs[wr_e
rr]);
    Tcl_SetResult(interp, err_str, TCL_VOLATILE);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(err_str, "%d", pipe_send.pidtag);
      Tcl_SetResult(interp, err_str, TCL_VOLATILE);
    }
  }

  return(TCL_OK);
}

int testinst_filter (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[])
{
  int filt_pos, bytes, priority = 1;
  input pipe_send;
  char err_str[40];

  switch (objc) {
  case 1:
    sprintf(interp->result, "Filter Pos:%d Filter Name:%s\n",
            instrument.filt_pos, instrument.filt_name);
    return(TCL_OK);
    break;
  case 2:
    Tcl_GetIntFromObj(interp,objv[1],&filt_pos);
    if ( filt_pos < 1 || filt_pos > 10) {
      sprintf(interp->result,
              "Error: Filter position must between 1 and 10\n");
      return(TCL_ERROR);
    }
    break;
  case 3:
    Tcl_GetIntFromObj(interp, objv[1], &filt_pos);
```

```
    if (filt_pos < 1 || filt_pos > 10) {
      Tcl_SetResult(interp, "Error: Filter position must be between 1 and 10", N
ULL);
      return(TCL_ERROR);
    }
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[2], NULL)) != 0) {
      sprintf(err_str, "Usage: filter position [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 4:
    Tcl_GetIntFromObj(interp,objv[1],&filt_pos);
    if ( filt_pos < 1 || filt_pos > 10) {
      sprintf(interp->result,
              "Error: Filter position must between 1 and 10\n");
      return(TCL_ERROR);
    }
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[2], NULL)) != 0) {
      sprintf(interp->result, "Usage: filter position [-prio priority]");
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[3], &priority) == TCL_ERROR) {
      sprintf(interp->result, "Error getting priority value!");
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(interp->result,"Usage: filter position [%s]", WAITOPT);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = FILTER;
  pipe_send.numargs = 1;
  pipe_send.pidtag = get_curevalpidtag();
  sprintf(pipe_send.args[0], "%d\0", filt_pos);

  if (inst_write(pipe_send, priority) < 0) {
    sprintf(interp->result, "Error sending filter command to instrument thread 0
!\nError: %s\n",
            write_errs[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);
}

int testinst_calibrate (ClientData clientdata, Tcl_Interp *interp,
                       int objc, Tcl_Obj *CONST objv[])
{
  int   bytes, priority = 1;
  input pipe_send;
  char  err_str[40];

  switch (objc) {
  case 1:
    break;
  case 2:
    if (strcmp(WAITOPT, Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(err_str, "Usage: inst_calib [%s]", WAITOPT);
      Tcl_SetResult(interp, err_str, NULL);
      return(TCL_ERROR);
    }
    priority = -1;
    break;
  case 3:
    if (strcmp("-prio", Tcl_GetStringFromObj(objv[1], NULL)) != 0) {
      sprintf(interp->result, "Usage: inst_calib -prio priority");
      return(TCL_ERROR);
    }
    if (Tcl_GetIntFromObj(interp, objv[2], &priority) == TCL_ERROR) {
      sprintf(interp->result, "Error getting priority value!");
      return(TCL_ERROR);
    }
    break;
  default:
    sprintf(interp->result, "Usage: inst_calib [%s]", WAITOPT);
    return(TCL_ERROR);
    break;
  }

  pipe_send.command = FILTER;
  pipe_send.numargs = 1;
  sprintf(pipe_send.args[0], "1\0");
  pipe_send.pidtag = get_curevalpidtag();
  if (inst_write(pipe_send, priority) < 0) {
    printf("Error sending calib command to instrument thread 0!\nError: %s\n", w
rite_errs[wr_err]);
    return(TCL_ERROR);
  } else {
    if (priority < 0) {
      Tcl_SetResult(interp, "Command successful", NULL);
    } else {
      sprintf(interp->result, "%d", pipe_send.pidtag);
    }
  }

  return(TCL_OK);

}

int testinst_header (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
        int     count, filt_pos;
        static char *filter,posname[6], ftshead[70];


        for (count = 0 ; count < NUMKEYS ; count++) {
                Tcl_SetStringObj(instkeys[count*3], instkeywords[count],
                                        -1);
                Tcl_SetStringObj(instkeys[count*3+2], instcomments[count],
                                        -1);
        }
        filter=Tcl_GetVar(interp,"c_filter",TCL_GLOBAL_ONLY);
        filt_pos=atoi(filter);
        if (filt_pos == 0) filt_pos=1;
        sprintf(posname,"pos%d", filt_pos);
        filter=Tcl_GetVar(interp, posname, TCL_GLOBAL_ONLY);
        /* Set the Dectector Name */
                Tcl_SetStringObj(instkeys[1], "S Testing Instrument Module", -1)
;
        /* Set the Filter Position */
        sprintf(ftshead, "I %d\0", filt_pos);
```

```
                Tcl_SetStringObj(instkeys[4],ftshead, -1);
        /* Set the Filter Type */
        sprintf(ftshead,"S %s\0", filter);
        Tcl_SetStringObj(instkeys[7], ftshead, -1);

        Store_header(clientdata, interp, NUMKEYS*3, instkeys);
}


int testinst_sims (ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
{

  char sim_str[80];
  Tcl_Obj *sim_obj;

  if (objc != 1) {
    Tcl_SetResult(interp, "Usage: inst_simdelays", NULL);
    return(TCL_ERROR);
  }
  sim_obj = Tcl_NewStringObj(sim_str, sizeof(sim_str));
  if (filt_delay == ON) {
    filt_delay = OFF;
    Tcl_SetStringObj(sim_obj, ".lowell10.msg#1 configure -text {Delay=OFF}", -1)
;
    Tcl_EvalObj(interp, sim_obj);
  } else {
    filt_delay = ON;
    Tcl_SetStringObj(sim_obj, ".lowell10.msg#1 configure -text {Delay=ON}", -1);
    Tcl_EvalObj(interp, sim_obj);
  }

  return(TCL_OK);
}



int inst_write(input send_cmd, int level) {

  int thread_num, count, get_results = 0;
  input *tmp_ptr;

  if ((level == 0) || (ABSOLUTE(level) > NUMPRIORITIES)) {
    wr_err = EILLPRIO; /* illegal priority level */
    return (-1);
  }

  switch(send_cmd.command) {
  case FILTER:
    thread_num = 0;
    break;
  case INST_STOP:
    thread_num = 1;
    break;
  case WAIT_CMD:
    thread_num = strtol(send_cmd.args[0], (char **) NULL, 10);
    if ((thread_num < 0) || (thread_num >= NUMINSTTHREADS)) {
      wr_err = EBADWAIT;
      return(-1);
    }
    send_cmd.numargs = 0;
    break;
  default:
    wr_err = ENOCMD; /* no defined command */
    return(-1);
    break;
  }

  pthread_mutex_lock(inst_cmds[thread_num].mutex);

  /* A negative priority means the calling routine wants the results back */

  if (level < 0) {
    get_results = 1;
    level = level * -1;
    send_cmd.command = -1 * send_cmd.command;
  }

  level--; /* So that priority 1 commands are sent to queue #0, priority 2
              commands are sent to queue #1... */

  tmp_ptr = (input *) ckalloc(sizeof(input));
  send_cmd.next = NULL;
  shm_write((void *) tmp_ptr, (void *) &send_cmd, sizeof(send_cmd));
  if (inst_cmds[thread_num].p_queue[level].head == NULL) {
    inst_cmds[thread_num].p_queue[level].head = tmp_ptr;
    inst_cmds[thread_num].p_queue[level].tail = tmp_ptr;
  } else {
    inst_cmds[thread_num].p_queue[level].tail->next = tmp_ptr;
    inst_cmds[thread_num].p_queue[level].tail = tmp_ptr;
  }

  if ((inst_cmds[thread_num].non_blocked & inst_cmds[thread_num].non_empty) == 0
) {
    pthread_cond_signal(inst_cmds[thread_num].go);
  }

  inst_cmds[thread_num].non_empty |= (1 << level); /* set non_empty value so tha
t
                                                      the bit in position $level$
 is
                                                      set to 1, and the others
                                                      are unaffected */
  pthread_mutex_unlock(inst_cmds[thread_num].mutex);

  /*****************************************/
  /* Get results if command asked for them */

  if (get_results) {
    return(inst_receive(thread_num));
  } else {
    return(0);
  }

}


int inst_receive(int thread_num) {

  printf("\nInst. Getting results\n");
  pthread_mutex_lock(inst_results[thread_num].mutex);
  while (inst_results[thread_num].value == RUN_WAIT) {
    printf("Inst. Waiting...\n");
    pthread_cond_wait(inst_results[thread_num].go, inst_results[thread_num].mute
x);
  }

  if (inst_results[thread_num].value == RUN_ERROR) {
    wr_err = EINTRNL; /* error executing internal part of command */
    inst_results[thread_num].value = RUN_WAIT;
    pthread_mutex_unlock(inst_results[thread_num].mutex);
```

```
    printf("Got results\n");
    return(-1);
  }

  if (inst_results[thread_num].value == RUN_REMOVED) {
    wr_err = EREMOVED; /* command removed from thread queue */
    inst_results[thread_num].value = RUN_WAIT;
    pthread_mutex_unlock(inst_results[thread_num].mutex);
    printf("Got results\n");
    return(-1);
  }

  inst_results[thread_num].value = RUN_WAIT;
  pthread_mutex_unlock(inst_results[thread_num].mutex);
  printf("Got results\n");
  return(0);
}


/***************************************************************************
*/
/************ Commands for removing and adjusting module thread queues ********
*/
/***************************************************************************
*/

int inst_remove(long rm_pidtag, int flag)
{

  int p_index, t_index, found = 0, blocks_this_queue, total = 0;
  input *previous, *traverse;

  if (rm_pidtag == 0) {
    printf("Illegal pidtag value!");
    return(-1);
  }
  if (rm_pidtag > 0) {
    for (t_index = 0; t_index < NUMINSTTHREADS; t_index++) {
      pthread_mutex_lock(inst_cmds[t_index].mutex);
      blocks_this_queue = 0;
      for (p_index = 0; p_index < NUMPRIORITIES; p_index++) {
        traverse = inst_cmds[t_index].p_queue[p_index].head;
        previous = NULL;
        while (traverse != NULL) {
          if (traverse->pidtag == rm_pidtag) {

            /* We found a command with the desired pidtag.  Remove it,
               but keep the rest of the list structure intact.  We could
               actually break out of the while here, since there should be
               only one command with the desired pidtag, but we keep going
               just in case (there's some case I haven't thought of) */

            /* Special case for block commands, because a block issues multiple
               inputs to the queue */

            if (traverse -> command != WAIT_CMD) {
              found++;
            } else {
              blocks_this_queue++;
            }

            if (flag) {
              if (traverse->command < 0) {
                pthread_mutex_lock(inst_results[t_index].mutex);
                inst_results[t_index].value = RUN_REMOVED;
                pthread_cond_signal(inst_results[t_index].go);
                pthread_mutex_unlock(inst_results[t_index].mutex);
              }
              if (previous == NULL) {
                if ((inst_cmds[t_index].p_queue[p_index].head = traverse->next)
== NULL) {
                  inst_cmds[t_index].non_empty &= ~(1 << p_index);
                }

                ckfree(traverse);
                traverse = inst_cmds[t_index].p_queue[p_index].head;

              } else {
                ckfree(previous->next);
                previous->next = traverse->next;
                traverse = traverse->next;
              }
            } else {
              previous = traverse;
              traverse = traverse->next;
            }
          } else {
            previous = traverse;
            traverse = traverse->next;
          }
        }
      }

      /* See if the currently running command is a block.  If not, check to see
that the
         currently running command doesn't have the same process tag as the proc
ess tag
         we're trying to remove (if it does, then you can't remove it, since it'
s being executed!)
         If the currently running command is a block, check its process tag.  If
 the
         process tag of the current block is the one we're trying to remove, it'
s too late
         (we can't remove it) so return 0 */

      if (inst_cmds[t_index].curr_run.command != WAIT_CMD) {
        if (found && (rm_pidtag == inst_cmds[t_index].curr_run.pidtag)) {
          found++;
        }
      } else {
        if (rm_pidtag == inst_cmds[t_index].curr_run.pidtag) {
          if (blocks_this_queue) {
            blocks_this_queue++;
          } else {
            pthread_mutex_unlock(inst_cmds[t_index].mutex);
            return(0); /* The currently running process tag is the one we're try
ing to
                          remove, so the process tag couldn't be found */
          }
        }
      }
      pthread_mutex_unlock(inst_cmds[t_index].mutex);
      if (blocks_this_queue > 1) {
        return(blocks_this_queue);
      }
      total += blocks_this_queue;
    }
    if ((found == 0) && (total != 0)) {
      found = 1; /* If we got here but didn't find any commands, then we are rem
oving blocks,
                    so send to the remove command that we should still look here
 for blocks */
```

```
    }
  } else {
    /* pidtag is negative, so it is associated with a block command sent by anot
her routine */

    for (t_index = 0; t_index < NUMINSTTHREADS; t_index++) {
      pthread_mutex_lock(inst_cmds[t_index].mutex);
      traverse = inst_cmds[t_index].p_queue[0].head; /* use priority of zero bec
ause
                                                         that is where blocks are
 going */
      previous = NULL;
      while (traverse != NULL) {
        if (traverse->pidtag == rm_pidtag) {
          if (traverse->command < 0) {
            pthread_mutex_lock(inst_results[t_index].mutex);
            inst_results[t_index].value = RUN_REMOVED;
            pthread_cond_signal(inst_results[t_index].go);
            pthread_mutex_unlock(inst_results[t_index].mutex);
          }

          if (previous == NULL) {
            if ((inst_cmds[t_index].p_queue[0].head = traverse->next) == NULL) {
              inst_cmds[t_index].non_empty &= ~(1 << 0);
            }

            ckfree(traverse);
            traverse = inst_cmds[t_index].p_queue[0].head;

          } else {
            ckfree(previous->next);
            previous->next = traverse->next;
            traverse = traverse->next;
          }
          found++;
        } else {
          previous = traverse;
          traverse = traverse->next;
        }
      }

      if (!found && (rm_pidtag == inst_cmds[t_index].curr_run.pidtag)) {

        /* If the block command wasn't in the queue for some thread, then a coup
le things
           could have happened:

           1. The block command could have gone through already.  The pidtag fie
ld
              of the curr_run structure will be the same as the rm_pidtag argume
nt passed
              to this function.  In this case, we want to reactivate the queue a
nd signal
              if necessary.  We know that the block we're looking for must be th
e one
              blocking the queue because of the pidtag value of the rm_pidtag ar
gument.
              Since the rm_pidtag argument is NEGATIVE, this part of the removal
 routine
              (to remove block subcommands) is only called if the main command (
the one
              that sent the blocks, i.e. an exposure) was found in the queue.  S
ince the
              main command is in the queue, it means that it MUST have written i
ts blocks
              already.
            2. The command we're trying to remove didn't send any block commands.
  In that
               case, the pidtag field of the curr_run structure would be differen
t than
               the rm_pidtag argument passed to this function.  We don't want to
reactivate
               the queue, nor do we want to signal in this case.

        */

        if (sem_trywait(inst_block[t_index]) != 0) {
          if (errno != EAGAIN) {

            /* The semaphore doesn't have to be zero, someone could call an expo
sure, and
               before trying to remove it, reactivate a queue, which would wait
the semaphore */

            lois_log0("Error reinitializing semaphore #%d in inst_remove, errno:
 %d",
                    t_index, errno);
          }
        }

        old_state[t_index] |= 1; /* unblock lowest priority queue */
        pthread_cond_signal(inst_cmds[t_index].go); /* If thread was sleeping bu
t commands were
                                                   in its queue */
      }
      found = 0; /* for next thread... */
      pthread_mutex_unlock(inst_cmds[t_index].mutex);
    }
  }

  return(found);
}
int inst_flush(long pidtag)

{

  int count, p_index;
  input *traverse, *previous;

  for (count = 0; count < NUMINSTTHREADS; count++) {
    pthread_mutex_lock(inst_cmds[count].mutex);
    for (p_index = 0; p_index < NUMPRIORITIES; p_index++) {
      traverse = inst_cmds[count].p_queue[p_index].head;
      previous = NULL;
      while (traverse != NULL) {
        if ((pidtag == ABSOLUTE(traverse->pidtag)) || (pidtag == 0)) {
          if (traverse->command < 0) {
            pthread_mutex_lock(inst_results[count].mutex);
            inst_results[count].value = RUN_REMOVED;
            pthread_cond_signal(inst_results[count].go);
            pthread_mutex_unlock(inst_results[count].mutex);
          }
          if (previous == NULL) {
            if ((inst_cmds[count].p_queue[p_index].head = traverse->next) == NUL
L) {

              inst_cmds[count].non_empty &= ~(1 << p_index);
            }
            ckfree(traverse);
            traverse = inst_cmds[count].p_queue[p_index].head;
          } else {
            ckfree(previous->next);
```

```
            previous->next = traverse->next;
            traverse = traverse->next;
          }
        } else {
          previous = traverse;
          traverse = traverse->next;
        }
      }
    }

    if (ABSOLUTE(inst_cmds[count].curr_run.pidtag) == pidtag) {
      /* If this thread is currently blocked then unblock it. */
      old_state[count] |= 1;
    }
    pthread_mutex_unlock(inst_cmds[count].mutex);
  }
  return(0);
}

int inst_blocked_state(int option)

{

  int count;

  for (count = 0; count < NUMINSTTHREADS; count++) {
    pthread_mutex_lock(inst_cmds[count].mutex);
    switch (option) {
    case SAVE_STATE:
      pthread_mutex_lock(actv_block[count]);
      old_state[count] = inst_cmds[count].non_blocked;
      inst_cmds[count].non_blocked = (~0 << (NUMPRIORITIES - 1));

      /* To block all instrument module thread queues while a command is being
         removed */
      break;
    case RESTORE_STATE:
      inst_cmds[count].non_blocked = old_state[count];
      pthread_cond_signal(inst_cmds[count].go); /* If a queue with commands in i
t
                                                just became unblocked...*/
      pthread_mutex_unlock(actv_block[count]);
      break;
    default:
      break;
    }
    pthread_mutex_unlock(inst_cmds[count].mutex);
  }
  return 0;
}
```

```
#include <netinet/in.h>

#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#define _REENTRANT
#define _P __P
#endif

#include <unistd.h>

#include <sys/types.h>

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <termios.h>

/* Tcl/Tk Includes */
#include <tcl.h>
#include <tk.h>

/* CDL Library Include */
#include <cdl.h>

/* LOIS Library Include */
#include <lois.h>

/* Instrument Include */
#include <LowellInst.h>
#include <ModThrd.h>

inst_struct instrument;
struct instrument_vectors instvec;
cmd_struct instcmd;
char instmsg[40];

static void reset_curr_run(int number);

int aborted = 0; /* flag set when a filter move is aborted */
int filt_delay;  /* flag set when module simulates delay involved in moving filt
ers */

int Init_InstThreadInfo()
{
  int count, count2;
  char err_str[80];

  for (count = 0; count < NUMINSTTHREADS; count++) {
    inst_cmds[count].mutex= (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t))
;
    inst_cmds[count].go = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
    pthread_mutex_init(inst_cmds[count].mutex, NULL);
    pthread_cond_init(inst_cmds[count].go, NULL);

    inst_results[count].mutex = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex
_t));
    inst_results[count].go = (pthread_cond_t *) ckalloc(sizeof(pthread_cond_t));
    pthread_mutex_init(inst_results[count].mutex, NULL);
```

```
    pthread_cond_init(inst_results[count].go, NULL);
    inst_results[count].value = RUN_WAIT;

    inst_block[count] = (sem_t *) ckalloc(sizeof(sem_t));
    if (sem_init(inst_block[count], 0, 0) < 0) {
      sprintf(err_str, "Error creating instrument semaphore #%d, error no: %d\n"
, count, errno);
      lois_log0(err_str);
      return(-1);
    }
    for (count2 = 0; count2 < NUMPRIORITIES; count2++) {
      inst_cmds[count].p_queue[count2].head = inst_cmds[count].p_queue[count2].t
ail = NULL;
    }
    reset_curr_run(count); /* Pre-set "currently-running" command in each thread
 to
                              hold a pidtag of 0 and a command of 0 (so we won't
                              select it during any search) */
  }

  for (count = 0; count < NUMINSTTHREADS; count++) {
    pthread_create(&inst_thread[count], NULL, INST_main, (void *) count);
  }

  instcmd.priority=DEF_PRIORITY;
  instcmd.t_interval=0;
  instcmd.t_times=0;
  instcmd.t_execute=0;

  return(0);
}

void * INST_main(void *arg)

{
  int bytes, number, level;
  int look_std_prio, result;
  char stat_str[40];
  lois_results stat_rslt;

  number = (int) arg;
  inst_cmds[number].non_blocked = ~(~0 << NUMPRIORITIES);
  inst_cmds[number].non_empty = 0;
  for ( ; ; ) {
    look_std_prio = 1;
    pthread_mutex_lock(inst_cmds[number].mutex);
    while((inst_cmds[number].non_blocked & inst_cmds[number].non_empty) == 0) {
      pthread_cond_wait(inst_cmds[number].go, inst_cmds[number].mutex);
    }
    for (level = NUMPRIORITIES-1; level > 0; level--) {
      if (inst_cmds[number].p_queue[level].head != NULL) {
        shm_write((void *) &inst_cmds[number].curr_run, (void *) inst_cmds[numbe
r].p_queue[level].head,
                  sizeof(inst_cmds[number].curr_run));
        ckfree((void *) inst_cmds[number].p_queue[level].head);
        if ((inst_cmds[number].p_queue[level].head = inst_cmds[number].curr_run.
next) == NULL) {
          inst_cmds[number].non_empty &= ~(1 << level);
        }
        look_std_prio = 0;
        break;
      }
    }
    if (look_std_prio & inst_cmds[number].non_blocked) {
      shm_write((void *) &inst_cmds[number].curr_run, (void *) inst_cmds[number]
.p_queue[0].head,
                    sizeof(inst_cmds[number].curr_run));
        ckfree((void *) inst_cmds[number].p_queue[0].head);
        if ((inst_cmds[number].p_queue[0].head = inst_cmds[number].curr_run.next)
== NULL) {
          inst_cmds[number].non_empty &= ~(1 << 0);
        }
      }
      pthread_mutex_unlock(inst_cmds[number].mutex);
      if (inst_cmds[number].curr_run.command == WAIT_CMD) {

        inst_cmds[number].non_blocked &= ~(1 << 0); /* because for now we can only

                                                       block the low priority queu
e */
        printf("Inst. thread #%d halted!\n", number);
        sem_post(inst_block[number]);

        number = WAIT_CASE;
      }

      switch(number) {
        /* dispatch correct function */
      case 0:
        aborted = 0; /* If the last command was aborted, this resets the flag */
        switch(ABSOLUTE(inst_cmds[number].curr_run.command)) {
        case FILTER:
          result = move_filter(inst_cmds[number].curr_run);
          break;
        default:
          lois_logl("Error: Unknown function in thread 0\n");
          result = -1;
          break;
        }
        break;
      case 1:
        switch(ABSOLUTE(inst_cmds[number].curr_run.command)) {
        case INST_STOP:
          result = stop_filter(inst_cmds[number].curr_run);
          break;
        default:
          lois_logl("Error: Unknown function on thread 1\n");
          result = -1;
          break;
        }
        break;
      case WAIT_CASE:
        /* wait condition - do nothing */
        number = (int) arg;
        if (inst_cmds[number].curr_run.command < 0) {
          inst_cmds[number].curr_run.command *= -1;
        }
        result = 0;
        break;
      default:
        lois_logl("Error: Unknown thread identification\n");
        break;
      }
      sprintf(instcmd.command, "inst_status\0");
      lois_send(&instcmd);
      if (inst_cmds[number].curr_run.command < 0) {
        pthread_mutex_lock(inst_results[number].mutex);
        inst_results[number].value = result;
        pthread_cond_signal(inst_results[number].go);
        pthread_mutex_unlock(inst_results[number].mutex);
      }
      if (inst_cmds[number].curr_run.command != WAIT_CMD) {
```

```
          reset_curr_run(number);
        }
    }
}


int move_filter (input cmd_input)
{

  int new_pos, direction, pos_diff;
  char inst_cmd[80];

  if (cmd_input.numargs != 1) {
    lois_logl("Error: Incorrect number of arguments in FILTER\n");
    return(-1);
  }

  if (strchr(cmd_input.args[0], ' ') != NULL) {
    lois_logl("Badly formatted input string in FILTER!\n");
    lois_logl("Arg1 = %s\n", cmd_input.args[0]);
    return(-1);
  }
  new_pos = atoi(cmd_input.args[0]);
  printf("Instrument Command: Filter, pos = %d\n", new_pos);

  shm_read((void *) &instrument, (char *) instvec.memory+sizeof(instvec),
           sizeof(instrument));

  /*  Should send a command to the telescope module here in MOVE module, maybe
      should simulate that here.

      sprintf(inst_cmd,"filt %d\0", filt_pos);

      Tcl_SetStringObj(inst_objcmd, inst_cmd,-1); */

  /* Next lines only in test instrument module */

  pos_diff = new_pos - instrument.filt_pos;

  if ((pos_diff > 5) || ((pos_diff > -5) && (pos_diff < 0))) {
    direction = -1; /* backward */
  } else {
    direction = 1; /* forward */
  }

  if (filt_delay == ON) {
    while (instrument.filt_pos != new_pos) {
      if (aborted) {
        sprintf(instcmd.command, ".lowell10.button#%d configure -bg grey", new_p
os);
        lois_send(&instcmd);
        shm_write((char *) instvec.memory+sizeof(instvec), (void *) &instrument,
                  sizeof(instrument));
        return(-1);
      }
      sprintf(instcmd.command, ".lowell10.button#%d configure -bg grey", instrum
ent.filt_pos);
      lois_send(&instcmd);
      instrument.filt_pos = (instrument.filt_pos+direction) % 10;
      if (!instrument.filt_pos) {
        instrument.filt_pos = 10;
      }

      sprintf(instcmd.command, ".lowell10.button#%d configure -bg green", instru
ment.filt_pos);
      lois_send(&instcmd);
      sprintf(instcmd.command, "set c_filter %d", instrument.filt_pos);
      lois_send(&instcmd);
      lois_sleep(980000000); /* simulate delay involved in moving wheel */
    }
  } else {
    sprintf(instcmd.command, ".lowell10.button#%d configure -bg grey", instrumen
t.filt_pos);
    lois_send(&instcmd);
    instrument.filt_pos = new_pos;
    sprintf(instcmd.command, "set c_filter %d", instrument.filt_pos);
    lois_send(&instcmd);
    sprintf(instcmd.command, ".lowell10.button#%d configure -bg green", instrume
nt.filt_pos);
    lois_send(&instcmd);
  }

  shm_write((char *) instvec.memory+sizeof(instvec), (void *) &instrument,
            sizeof(instrument));

  return(0);
}

int stop_filter (input cmd_input) {

  if (cmd_input.numargs != 0) {
    lois_log1("Error: Incorrect number of args in STOP_FILTER\n");
    return(-1);
  }

  printf("Instrument Command: INST_STOP\n");
  aborted = 1;

  return(0);
}

static void reset_curr_run(int number) {

  inst_cmds[number].curr_run.pidtag = 0;
  inst_cmds[number].curr_run.command = 0;
}
```

```
global lois_win eval LOIS_HOME tmp_lst

set dsk_space " "
set eval(slave) ""
set version 1.0.1
set tmp_lst ""

proc lois_main {root args} {

    global lois_win eval sysstat auto_store atag dsk_space \
            version LOIS_HOME

    # this treats "." as a special case

    if {$root == "."} {
        set base ""
    } else {
        set base $root
    }

    set lois_win $root
    wm title $root "LOIS System Console"
    wm geometry $root +0-40

    frame $base.frame#2

    frame $base.frame#1

    label $base.label#2 \
            -justify left \
            -text {Lowell Observatory
Instrumentation System}

    catch {
        $base.label#2 configure \
                -font -*-6x13-Bold-R-Normal-*-*-140-*-*-*-*-*-*
    }

    button $base.button#1 \
            -height 2 \
            -text Configure \
            -command lois_config \
            -width 10

    button $base.button#2 \
            -height 2 \
            -text Storage \
            -width 10 \
            -command store_cmd

    button $base.button#3 \
            -height 2 \
            -text Analysis \
            -state disabled \
            -width 10

    button $base.button#5 \
            -height 2 \
            -text Display \
            -width 10 \
            -command disp_cmd

    button $base.button#4 \
            -height 2 \
            -text {External
Control} \
```

```
            -state disabled \
            -width 10

    button $base.button#6 \
            -height 2 \
            -text Logging \
            -state disabled \
            -width 10

    frame $base.frame#3

    set t [Scrolled_Text $base.text#1 \
            -height 10 \
            -width 55
          ]

    set log_window [Scrolled_Text $base.text#2 \
            -height 10 \
            -width 60 \
            -bg black
            ]

    label $base.label#1 \
            -text {System Status:}

    set sysstat [       canvas $base.canvas#1 \
                        -borderwidth 1 \
                        -height 100 \
                        -relief ridge \
                        -width 150 \
                        -bg navy
                ]
    set auto_store {ON}

    $sysstat create text 70 10 -text "Auto-Store:" -fill white -anchor e
    $sysstat create text 71 10 -text $auto_store -fill green -anchor w \
            -tag atag
    $sysstat create text 70 25 -text "Avail. Disk:" -fill white -anchor e
    $sysstat create text 128 25 -text $dsk_space -fill green -anchor e \
            -tag dsktag
    $sysstat create text 130 25 -text "KB" -fill white -anchor w

    $sysstat create text 50 90 -text "Version:" -fill white -anchor e
    $sysstat create text 51 90 -text $version -fill green -anchor w
    button $base.button#7 \
            -background red \
            -text Exit \
            -command lexit
    catch {
        $base.button#7 configure \
                -font -*-6x13-Bold-R-Normal-*-*-160-*-*-*-*-*-*
    }

    # Geometry management

    grid $base.frame#2 -in $root        -row 2 -column 1
    grid $base.frame#1 -in $root        -row 1 -column 2 \
            -sticky nw
    grid $base.label#2 -in $root        -row 1 -column 1
    grid $base.button#1 -in $base.frame#1       -row 1 -column 1
    grid $base.button#2 -in $base.frame#1       -row 1 -column 2
    grid $base.button#3 -in $base.frame#1       -row 1 -column 3
    grid $base.button#5 -in $base.frame#1       -row 1 -column 4
    grid $base.button#4 -in $base.frame#1       -row 1 -column 5
    grid $base.button#6 -in $base.frame#1       -row 1 -column 6
    grid $base.frame#3 -in $root        -row 2  -column 2 \
            -sticky nesw

    grid $base.text#1 -in $base.frame#3 -row 2 -column 2  \
            -sticky nesw
    grid $base.text#2 -in $base.frame#3 -row 2 -column 3  \
            -sticky nesw

    grid $base.label#1 -in $base.frame#2        -row 1 -column 1  \
            -sticky w
    grid $base.canvas#1 -in $base.frame#2       -row 2 -column 1  \
            -sticky nesw
    grid $base.button#7 -in $base.frame#2       -row 3 -column 1  \
            -sticky w

    # Resize behavior management

    grid rowconfigure $base.frame#2 1 -weight 0 -minsize 30
    grid rowconfigure $base.frame#2 2 -weight 0 -minsize 30
    grid rowconfigure $base.frame#2 3 -weight 0 -minsize 30
    grid columnconfigure $base.frame#2 1 -weight 0 -minsize 30

    grid rowconfigure $root 1 -weight 0 -minsize 30
    grid rowconfigure $root 2 -weight 0 -minsize 30
    grid columnconfigure $root 1 -weight 0 -minsize 30
    grid columnconfigure $root 2 -weight 0 -minsize 30

    grid rowconfigure $base.frame#1 1 -weight 0 -minsize 30
    grid columnconfigure $base.frame#1 1 -weight 0 -minsize 30
    grid columnconfigure $base.frame#1 2 -weight 0 -minsize 30
    grid columnconfigure $base.frame#1 3 -weight 0 -minsize 30
    grid columnconfigure $base.frame#1 4 -weight 0 -minsize 30
    grid columnconfigure $base.frame#1 5 -weight 0 -minsize 30
    grid columnconfigure $base.frame#1 6 -weight 0 -minsize 30
    # additional interface code

    # Text tags give script output, command errors, command
    # results, and the prompt a different appearance

    $t tag configure prompt -underline false
    $t tag configure result -foreground purple
    $t tag configure error -foreground red
    $t tag configure output -foreground blue
    $t tag configure pidtag -foreground "forest green"
    $log_window tag configure redlog -foreground red
    $log_window tag configure greenlog -foreground green
    $log_window tag configure bluelog -foreground blue
    $log_window tag configure yellowlog -foreground yellow
    $log_window tag configure orangelog -foreground orange
    $log_window tag configure output -foreground cyan

    # Insert the prompt and initialize the limit mark

    set eval(prompt) "LOIS % "
    $t insert insert $eval(prompt) prompt
    $t mark set limit insert
    $t mark gravity limit left
    $log_window mark set limit insert
    $log_window mark gravity limit left

    focus $t
    set eval(text) $t
    set eval(slave) [SlaveInit shell]
    set eval(logging) $log_window

    # Key bindings that limit input and eval things. The break in
    # the bindings skips the default Text binding for the event.
```

```
    bind $t <Return> {EvalTypein ; break}
    bind $t <BackSpace> {
        if {[%W tag nextrange sel 1.0 end] != ""} {
            %W delete sel.first sel.last
        } elseif {[%W compare insert > limit]} {
            %W delete insert-1c
            %W see insert
        }
        break
    }
    bind $t <Key> {
        if [%W compare insert < limit] {
            %W mark set insert end
        }
    }

    set eval(cmd) [interp create [list shell command]]
    interp alias $eval(cmd) pt_write {} PidtagAlias $eval(cmd)

    interp eval $eval(slave) "set LOISHOME $LOIS_HOME"
    interp eval $eval(slave) "olog {LOIS Version 1.0.1}"
    interp eval $eval(slave) {source $LOISHOME/scripts/start.tcl}
    interp eval $eval(slave) {start_lois}

}

# Procedure to turn $ references to 'set' commands.  I use mostly string command
s
# here, especially to find matching parentheses. (By the way, parentheses are ()
,
# brackets are [] and curly (squiggly) braces are {}.) There might be a faster
# (and easier) way to do this with the regular expression matcher in Tcl, but
# I don't know how.

proc Remove_Dollars {input} {

# First get rid of all nested parentheses.  If a dollar sign is followed by
# a variable name, and then a parenthesis comes, it refers to an array, so
# substitution must occur within the parentheses first.

    while {[regexp {[$]([[A-Za-z0-9_]|::)+)\(} $input match_str]} {

        set start_ind [string first $match_str $input]; # index of first paren.
        set pcount 1;  # count to determine when matching right paren is found

        # Start of text inside parentheses
        set inner_text [expr $start_ind + [string length $match_str]]

        for {set i $inner_text} {$i < [string length $input]} {incr i} {
            if { [string compare [string index $input $i] "\\"] == 0 } {
                incr i;       #ignore backslashes and the characters after them
            } elseif { [string compare [string index $input $i] ")"] == 0 } {
                if { [incr pcount -1] == 0 } {
                    break
                }
            } elseif { [string compare [string index $input $i] "("] == 0 } {
                incr pcount
            }
        }
        if {$i == [string length $input]} {

            # If we're here, then the matching right paren wasn't found
            error "Unbalanced parentheses!"

        } else {
                set end_ind $i;                     # index of matching right paren
            }

# First part: part of input before the dollar-variable-paren sequence (we haven'
t
# touched it)
# Bracket part: part of input after dollarsign, up to and including open paren
# Last part: part of input after the matching close paren (we haven't touched it
)
# Middle part: Part between the parentheses.  This part needs to be run through
to
# execute any appropriate substitutions (this is done recursively)

            set first_part [string range $input 0 [expr $start_ind-1]]
            set bracket_part [string range $input [expr $start_ind+1] [expr $inner_t
ext-1]]
            set last_part [string range $input [expr $end_ind+1] end]

# Recursive call to Remove_Dollars for middle part
            set middle_part [Remove_Dollars [string range $input $inner_text \
                    [expr $end_ind-1]]]

# Replace dollar sign with set call, use curly braces to include the inner porti
on containing
# the array index information
            set input [append first_part "\[set " $bracket_part $middle_part "\}\]"
$last_part]
        }


# Replace $ qualifiers (used to get variable values) with set commands to take
# advantage of the SetAlias command. (Also need to worry about namespace qualifi
ers)

    regsub -all {[$](([A-Za-z0-9_]|::)+)} $input {[set \1]} second_stage

# Now change all bracketed references ${varname} to [set {varname}].  By Tcl rul
es,
# the brackets CANNOT be nested. Something like puts ${a b {c d}} will cause an
error

    regsub -all {[$]\{(([^\}]*)\}} $second_stage {[set {\1}]} third_stage

    return $third_stage

}

# Evaluate everything between limit and end as a Tcl command

proc EvalTypein {} {

    global eval tmp_lst

    set input_string [$eval(text) get limit end]

# Strip newline off end of this line of text

    regexp "(.*)\n$" $input_string match command_string newline

# If line ends with \ followed by whitespace, move the cursor down one line and
# add this line to the part of the command that has already been parsed.  Also,
# wait for the rest of the command.  The pattern contains four \ characters beca
use
# that is how \ is expressed in a pattern without curly braces.

    if {[regsub -all "(\\\\\[ \t\]*)$" $command_string { } tmp_cmd] != 0} {
```

```
        $eval(text) insert insert \n
        $eval(text) mark set limit insert
        set tmp_lst [concat $tmp_lst $tmp_cmd]

    } else {

# Otherwise, if the command we have is done being parsed, reset the list in whic
h
# we accumulated the parts of the command, and evaluate the command (if it is co
mplete)

        set dollar_cmd [concat $tmp_lst $tmp_cmd]

# Remove all dollar sign references in the command

        set command [Remove_Dollars $dollar_cmd]

#        puts "cmd:$command"
        $eval(text) insert insert \n
        if [info complete $command] {
            $eval(text) mark set limit insert
            set tmp_lst ""
            Eval $command
        }
    }
}

# Echo the command and evaluate it

proc EvalEcho { command } {

        global eval

        $eval(text) mark set insert end
        $eval(text) insert insert $command\n
        Eval $command eval

}

# Evaluate a command and display its result

proc Eval { command } {
    global eval

    $eval(text) mark set insert end

    # See if there is a priority attached to the command

    set sp_cmd [split $command]
    set pos [lsearch $sp_cmd rio]

    if {$pos == -1} {
        lois_send $command
    } else {
        set sp_cmd [lreplace $sp_cmd $pos $pos]
        set command [join [lrange $sp_cmd 0 [expr $pos-1]]]
        set priority [lindex $sp_cmd $pos]
        lois_send $command $priority
    }

    return
}

# Insert a prompt back into the console window

proc PutPrompt {} {
    global eval

    if {[$eval(text) compare insert != "insert linestart"]} {
        $eval(text) insert insert \n
    }
    $eval(text) insert insert $eval(prompt) prompt
    $eval(text) see insert
    $eval(text) mark set limit insert
    return
}


# Create and initialize the slave interpreter

proc SlaveInit { slave } {
        global lois_win LOIS_HOME

    interp create $slave
    interp eval $slave [list set argv [list -use [winfo id $lois_win]]]
    interp eval $slave [list set argc 2]
    load /usr/local/lib/libtk8.0.so Tk $slave
    interp eval $slave {wm withdraw .}
    interp eval $slave {tk appname slave}
    interp alias $slave reset {} ResetAlias $slave
    interp alias $slave log {} LogAlias $slave
    interp alias $slave rlog {} RLogAlias $slave
    interp alias $slave glog {} GLogAlias $slave
    interp alias $slave ylog {} YLogAlias $slave
    interp alias $slave blog {} BLogAlias $slave
    interp alias $slave olog {} OLogAlias $slave
    interp alias $slave diskfree {} DskAlias $slave
    interp alias $slave stat {} StatusLine $slave
    interp alias $slave puts {} PutsAlias $slave
    interp alias $slave write {} WriteAlias $slave
    interp alias $slave cclose {} CloseAlias $slave
    interp alias $slave astore {} AsaveAlias $slave
    interp alias $slave csend {} ccdsend $slave
    interp alias $slave guisend {} GUIsendAlias $slave
    interp alias $slave sload {} load_script
    interp alias {} single $slave single
    interp alias {} waitcam $slave waitcam
    interp alias $slave lois_send {} lois_send
    interp alias $slave exit {} lexit $slave
#    interp alias $slave load {} load $slave
    return $slave
}

# The reset alias deletes the slave and starts a new one

proc ResetAlias { slave } {
        interp delete $slave
        SlaveInit $slave
}

proc PutsAlias  { slave args } {

    global eval

        if {[llength $args] > 3} {
                error "invalid arguments"
        }
        set newline "\n"
        if {[string match "-nonewline" [lindex $args 0]]} {
                set newline ""
                set args [lreplace $args 0 0]
```

```
        }
        if {[llength $args] == 1} {
                set chan stdout
                set string [lindex $args 0]$newline
        } else {
                set chan [lindex $args 0]
                set string [lindex $args 1]$newline
        }
        if [regexp (stdout|stderr) $chan] {
                global eval
                $eval(text) mark gravity limit right
                $eval(text) insert limit $string output
                $eval(text) see limit
                $eval(text) mark gravity limit left
        } else {
                interp share $slave $chan {}
                puts -nonewline $chan $string
        }
}
proc PidtagAlias  { slave args } {

    global eval

        if {[llength $args] > 3} {
                error "invalid arguments"
        }
        set newline "\n"
        if {[string match "-nonewline" [lindex $args 0]]} {
                set newline ""
                set args [lreplace $args 0 0]
        }
        if {[llength $args] == 1} {
                set chan stdout
                set string [lindex $args 0]$newline
        } else {
                set chan [lindex $args 0]
                set string [lindex $args 1]$newline
        }
        if [regexp (stdout|stderr) $chan] {
                global eval
                $eval(text) mark gravity limit right
                $eval(text) insert limit $string pidtag
                $eval(text) see limit
                $eval(text) mark gravity limit left
        } else {
                interp share $slave $chan {}
                puts -nonewline $chan $string
        }
}

proc LogAlias  { slave args } {

        if {[llength $args] > 3} {
                error "invalid arguments"
        }
        set newline "\n"
        if {[string match "-nonewline" [lindex $args 0]]} {
                set newline ""
                set args [lreplace $args 0 0]
        }
        if {[llength $args] == 1} {
                set chan stdout
                set string [lindex $args 0]$newline
        } else {
                set chan [lindex $args 0]
                set string [lindex $args 1]$newline
```

```
	}
	if [regexp (stdout|stderr) $chan] {
		global eval
		$eval(logging) mark gravity limit right
		$eval(logging) insert limit $string output
		$eval(logging) see limit
		$eval(logging) mark gravity limit left
	} else {
		interp share $slave $chan {}
		puts -nonewline $chan $string
	}
}
proc GLogAlias  { slave args } {

	if {[llength $args] > 3} {
		error "invalid arguments"
	}
	set newline "\n"
	if {[string match "-nonewline" [lindex $args 0]]} {
		set newline ""
		set args [lreplace $args 0 0]
	}
	if {[llength $args] == 1} {
		set chan stdout
		set string [lindex $args 0]$newline
	} else {
		set chan [lindex $args 0]
		set string [lindex $args 1]$newline
	}
	if [regexp (stdout|stderr) $chan] {
		global eval
		$eval(logging) mark gravity limit right
		$eval(logging) insert limit $string greenlog
		$eval(logging) see limit
		$eval(logging) mark gravity limit left
	} else {
		interp share $slave $chan {}
		puts -nonewline $chan $string
	}
}
proc RLogAlias  { slave args } {

	if {[llength $args] > 3} {
		error "invalid arguments"
	}
	set newline "\n"
	if {[string match "-nonewline" [lindex $args 0]]} {
		set newline ""
		set args [lreplace $args 0 0]
	}
	if {[llength $args] == 1} {
		set chan stdout
		set string [lindex $args 0]$newline
	} else {
		set chan [lindex $args 0]
		set string [lindex $args 1]$newline
	}
	if [regexp (stdout|stderr) $chan] {
		global eval
		$eval(logging) mark gravity limit right
		$eval(logging) insert limit $string redlog
		$eval(logging) see limit
		$eval(logging) mark gravity limit left
	} else {
		interp share $slave $chan {}
		puts -nonewline $chan $string
```

```
	}
}
proc OLogAlias  { slave args } {

	if {[llength $args] > 3} {
		error "invalid arguments"
	}
	set newline "\n"
	if {[string match "-nonewline" [lindex $args 0]]} {
		set newline ""
		set args [lreplace $args 0 0]
	}
	if {[llength $args] == 1} {
		set chan stdout
		set string [lindex $args 0]$newline
	} else {
		set chan [lindex $args 0]
		set string [lindex $args 1]$newline
	}
	if [regexp (stdout|stderr) $chan] {
		global eval
		$eval(logging) mark gravity limit right
		$eval(logging) insert limit $string orangelog
		$eval(logging) see limit
		$eval(logging) mark gravity limit left
	} else {
		interp share $slave $chan {}
		puts -nonewline $chan $string
	}
}
proc YLogAlias  { slave args } {

	if {[llength $args] > 3} {
		error "invalid arguments"
	}
	set newline "\n"
	if {[string match "-nonewline" [lindex $args 0]]} {
		set newline ""
		set args [lreplace $args 0 0]
	}
	if {[llength $args] == 1} {
		set chan stdout
		set string [lindex $args 0]$newline
	} else {
		set chan [lindex $args 0]
		set string [lindex $args 1]$newline
	}
	if [regexp (stdout|stderr) $chan] {
		global eval
		$eval(logging) mark gravity limit right
		$eval(logging) insert limit $string yellowlog
		$eval(logging) see limit
		$eval(logging) mark gravity limit left
	} else {
		interp share $slave $chan {}
		puts -nonewline $chan $string
	}
}
proc BLogAlias  { slave args } {

	if {[llength $args] > 3} {
		error "invalid arguments"
	}
	set newline "\n"
	if {[string match "-nonewline" [lindex $args 0]]} {
		set newline ""
```

```
                set args [lreplace $args 0 0]
        }
        if {[llength $args] == 1} {
                set chan stdout
                set string [lindex $args 0]$newline
        } else {
                set chan [lindex $args 0]
                set string [lindex $args 1]$newline
        }
        if [regexp (stdout|stderr) $chan] {
                global eval
                $eval(logging) mark gravity limit right
                $eval(logging) insert limit $string bluelog
                $eval(logging) see limit
                $eval(logging) mark gravity limit left
        } else {
                interp share $slave $chan {}
                puts -nonewline $chan $string
        }
}
proc WriteAlias {slave args} {

        set chan [lindex $args 0]
        set string [lrange $args 1 [llength $args]]

        interp share $slave $chan {}
        puts $chan $string

}
proc CloseAlias {slave args} {

        set chan [lindex $args 0]

        interp share $slave $chan {}
        close $chan
}

proc Scrolled_Text { f args } {

    frame $f
    eval {text $f.text \
            -xscrollcommand [list $f.xscroll set] \
            -yscrollcommand [list $f.yscroll set]} $args
    scrollbar $f.xscroll -orient horizontal \
            -command [list $f.text xview]
    scrollbar $f.yscroll -orient vertical \
            -command [list $f.text yview]
    grid $f.text $f.yscroll -sticky news
    grid $f.xscroll -sticky news
    grid rowconfigure $f 0 -weight 1
    grid columnconfigure $f 0 -weight 1
    return $f.text
}


proc disp_cmd {} {
        global eval

        $eval(slave) eval {display_config}

}
proc store_cmd {} {
        global eval

        $eval(slave) eval {store_cfg}
```

```
}


proc lois_config {} {
        global eval LOIS_HOME
        $eval(slave) eval {source $LOISHOME/scripts/config.tcl}
        $eval(slave) eval {config_lois}

}

proc AsaveAlias {args} {

        global auto_store sysstat atag

        $sysstat dchars atag 0 end

        if { $auto_store == "OFF" } {
           set auto_store "ON"
          $sysstat create text 71 10 -text $auto_store -fill green -anchor w \
                        -tag atag
        } else {
           set auto_store "OFF"
          $sysstat create text 71 10 -text $auto_store -fill red -anchor w \
                        -tag atag
        }

}

proc GUIsendAlias {slave args} {

        global eval

        gsend $args

}
proc DskAlias {slave args} {

        global sysstat

        set dsk [exec df -k $args]
        set ldsk [split $dsk " "]
        set lcnt [llength $ldsk]
        for {set x 0} {$x < $lcnt} {incr x} {
            set lpos [lsearch -exact $ldsk {}]
            set ldsk [lreplace $ldsk $lpos $lpos]
        }
        set df [lindex $ldsk 9]
        $sysstat dchars dsktag 0 end
        $sysstat insert dsktag 0 $df
}

proc lexit {args} {

set base [toplevel .exitdiag]
wm title $base "Exit Lois?"
wm geometry $base +400+400
$base configure -bg blue


        frame $base.frame#2

        frame $base.frame#1

        message $base.message#1 \
                -background blue \
                -foreground white \
                -justify center \
                -text {Do You Really Want
To Exit?} \
                -font -*-6x13-Bold-R-Normal-*-*-140-*-*-*-*-*-* \
                -width 400

#       Causing seg. faults because a font size of larger than 140 causes
#       a crash if it's drawn a second time
#       catch {
#       $base.message#1 configure \
#                       -font -*-6x13-Bold-R-Normal-*-*-240-*-*-*-*-*-*
#       }

        button $base.button#1 \
                -background red \
                -text Cancel \
                -command "destroy .exitdiag"

        catch {
                $base.button#1 configure \
                        -font -*-6x13-Bold-R-Normal-*-*-140-*-*-*-*-*-*
        }

        button $base.button#2 \
                -text Yes \
                -background green \
                -command "exit"

        catch {
                $base.button#2 configure \
                        -font -*-6x13-Bold-R-Normal-*-*-140-*-*-*-*-*-*
        }

        # Geometry management

        grid $base.frame#2 -in $base    -row 1 -column 1  \
                -sticky w
        grid $base.frame#1 -in $base    -row 2 -column 1  \
                -sticky e
        grid $base.message#1 -in $base.frame#2  -row 1 -column 1  \
                -sticky nsw
        grid $base.button#1 -in $base.frame#1   -row 1 -column 1
        grid $base.button#2 -in $base.frame#1   -row 1 -column 2

        # Resize behavior management

        grid rowconfigure $base.frame#2 1 -weight 0 -minsize 30
        grid columnconfigure $base.frame#2 1 -weight 0 -minsize 30

        grid rowconfigure $base 1 -weight 0 -minsize 30
        grid rowconfigure $base 2 -weight 0 -minsize 30
        grid columnconfigure $base 1 -weight 0 -minsize 238

        grid rowconfigure $base.frame#1 1 -weight 0 -minsize 30
        grid columnconfigure $base.frame#1 1 -weight 0 -minsize 30
        grid columnconfigure $base.frame#1 2 -weight 0 -minsize 30
# additional interface code
# end additional interface code

}
proc StatusLine {slave args} {

                global eval

        $eval(slave) eval .nccd.status $args
```

```
}
```

```
# Series of tests for LOIS version 1.1

# Non-graphical recursive function test
proc test1 {{i 0}} {

    if {$i == "?"} {
        set s "Calculates factorial of its argument (default value=0).\nIt is us
ed\
as a procedure to test recursion."
        return $s
    }
    if {$i == 0} {
        return 1
    } else {
        return [expr $i * [test1 [expr $i-1]]]
    }
}

# Graphical recursive function test
proc gr_test1 {{i 0} {destroy 1}} {

    if {$i == "?"} {
        set s "Calculates factorial of its argument (default value=0).\nIt also\
displays a number of buttons given by the\nfirst argument."
        return $s
    }

    if {$destroy} {
        catch {destroy .test1}
    }
    catch {toplevel .test1}
    if {$i == 0} {
        grid [button .test1.#$i -text "number $i" -bg red] -in .test1 \
                -row $i -column 0
        return 1
    } else {
        grid [button .test1.#$i -text "number $i" -bg red] -in .test1 \
                -row $i -column 0
        return [expr $i * [gr_test1 [expr $i-1] 0]]
    }
}

# Procedure to test sequences of filter move, telescope move, frame exposure
proc test2 {{i 1}} {

    if {$i == "?"} {
        set s "Runs a sequence of telescope, instrument and camera\nmoves. The\
argument (default value=1) indicates how\nmany times the sequence should repeat.
"
        return $s
    }
    inst_calib
    move 0:0:0 0:0:0
    for {set j 0} {$j < $i} {incr j} {
        filter [expr [expr [expr 3*$j] % 10] + 1]
        rmove -ra [expr 27000*$j] -dec [expr 3600*$j]
        single frame=o
    }
}

# Procedure to test graphics with a series of telescope and instrument moves
proc gr_test2 {{destroy 1}} {

    if {$destroy == "?"} {
        set s "Changes the focus to 40, then displays a message.\nChanges the fo
cus\
```

```
to 100, displays another message.\nChanges focus to 190, then 150, and displays\
messages\nwhen each of those operations finish."
        return $s
    }
    if {$destroy} {
        catch {destroy .test2}
    }
    catch {toplevel .test2}
    tel_activate
    focus_go 0 -prio -1
    focus_go 40
    grid [message .test2.msg1 -text "Focus should be 0" -bg blue -fg black] -in
.test2 \
            -row 0 -column 0
    focus_go 100 -wait
    grid [message .test2.msg2 -text "Focus should be 100" -bg blue -fg black] -i
n \
            .test2 -row 1 -column 0
    tel_block
    focus_go 150
    focus_go 190 -prio -2
    grid [message .test2.msg3 -text "Focus should be 190" -bg blue -fg black] \
            -in .test2 -row 2 -column 0

# Need the sleep for 5 seconds before tel_activate otherwise activate on "focus"
# thread is called before block on "focus" thread is executed.

    after 5000
    tel_activate
    after 6000
    grid [message .test2.msg4 -text "Focus should be 150" -bg blue -fg black] -i
n \
            .test2 -row 3 -column 0
}

# Procedure to run a focus_exposure, then a series of flats, then some image
# exposures
proc test3 {{j 1}} {

    if {$j == "?"} {
        set s "Runs a focus exposure, then a series of flats. After\nthe\
flats, a message is displayed. Then the procedure\nruns test2. The argument\
indicates how many images\nto take (default value=1)"
        return $s
    }
    move 0:0:0 0:0:0
    focrun 50 10 50
    for {set i 0} {$i < 5} {incr i} {
        puts "Before, step $i"
        single frame=f -wait
        puts "After, step $i"
    }
    catch {destroy .test3}
    catch {toplevel .test3}
    grid [message .test3.flatmsg -text "Flats done" -bg green -fg black] -in .te
st3 \
            -row 0 -column 0
    after 10000
    test2 $j
}

# Procedure to take a series of object exposures, each exposure is one second lo
nger than
# the previous exposure (first exposure lasts for 1 second)
proc gr_test3 {{i 1} {destroy 0}} {
    if {$i == "?"} {
        set s "Runs a sequence of object exposures, where each\nexposure lasts\
one second more than the previous\nexposure.  The first exposure is one second,\
the\nargument indicates how many images to take.\n(default value=1)"
        return $s
    }
    store_cfg
    if {$destroy} {
        catch {destroy .test3}
    }
    catch {toplevel .test3}
    pack [text .test3.txt -width 20 -height 1]
    .test3.txt tag add txtag 1.0 end
    .test3.txt tag configure txtag -font 9x15
    for {set j 1} {$j <= $i} {incr j} {
        .test3.txt delete 1.0 1.end
        .test3.txt insert 1.0 "Exposure of $j seconds"
        single frame=o -wait
    }
}


set pr_names [list test1 gr_test1 test2 gr_test2 test3 gr_test3]
puts "Test cases loaded successfully!"
puts "Procs: $pr_names"
puts "For info about a procedure enter: '<proc_name> ?'"
```

```
global pidtag
set pidtag 1

proc control_init {args} {

    global pidtag
    set root [toplevel .control]
    wm title .control "Execution Control"
    wm geometry .control 500x100+0+574

    frame $root.frame#1
    frame $root.frame#2

    button $root.button#1 \
        -text {Clear
Queues} \
        -height 2 \
        -cursor pirate \
        -command {clear_queues graphics}

    button $root.button#2 \
        -text {Abort
Command} \
        -height 2 \
        -bg red \
        -command {abort_cmd graphics $pidtag}

     button $root.button#3 \
             -text {Remove
Command} \
             -height 2 \
             -bg yellow \
             -command {lois_remove $pidtag}

     label $root.label#1 \
             -text {Pidtag:}

     entry $root.entry#1 \
             -textvariable pidtag \
             -width 7

     label $root.label#2 \
             -text {Result Message:}

     message $root.message#1 \
             -text "" \
             -width 400

     grid $root.frame#1 -in $root -row 1 -column 1
     grid $root.frame#2 -in $root -row 2 -column 1

     grid $root.button#1 -in $root.frame#1 -row 1 -column 1 -sticky ew
     grid $root.button#2 -in $root.frame#1 -row 1 -column 2 -sticky ew
     grid $root.button#3 -in $root.frame#1 -row 1 -column 3 -sticky ew
     grid $root.label#1 -in $root.frame#2 -row 0 -column 0 -sticky e
     grid $root.entry#1 -in $root.frame#2 -row 0 -column 1 -sticky w
     grid $root.label#2 -in $root.frame#2 -row 1 -column 0 -sticky e
     grid $root.message#1 -in $root.frame#2 -row 1 -column 1 -sticky w

     grid rowconfigure $root 1 -weight 0 -minsize 30
     grid rowconfigure $root 2 -weight 0 -minsize 30
     grid columnconfigure $root 1 -weight 0 -minsize 30

     grid rowconfigure $root.frame#1 1 -weight 0 -minsize 30
     grid columnconfigure $root.frame#1 1 -weight 0 -minsize 30
     grid columnconfigure $root.frame#1 2 -weight 0 -minsize 30
     grid columnconfigure $root.frame#1 3 -weight 0 -minsize 30

     grid rowconfigure $root.frame#2 0 -weight 0 -minsize 30
     grid rowconfigure $root.frame#2 1 -weight 0 -minsize 30
     grid columnconfigure $root.frame#2 0 -weight 0 -minsize 30
     grid columnconfigure $root.frame#2 1 -weight 0 -minsize 30
}
```

```
/**************************************************************************
                        Lowell Observatory
                 CCD Acquisitions Software Package


Program:         control.so
Called From:     module
Name:            $RCSfile: control.c,v $
Started:         06/05/98
Revision:        $Revision: 1.1.2.3 $
Last Revised:    $Date: 1999/05/24 02:46:22 $
By:              $Name:  $

Included in:     module


Explanation:     This program is the initializer for the LOAS. It inits the
                 shared memory, starts the configuration and calls the
                 functions for the CCD Camera, Telescope, and Intsturment.


        Copyright 1998
, Lowell Observatory, All Rights Reserved

---------------------------------------------------------------------------


#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#ifndef _XOPEN_SOURCE
#define _XOPEN_SOURCE
#endif
#endif

#include <unistd.h>

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>

#ifdef __LINUX__
#include <sys/ipc.h>
#include <sys/msg.h>
#elif defined(__SOLARIS_5x__)
#include <mqueue.h>
#endif


/* Tcl/Tk Includes */
#include <tk.h>
#include <tcl.h>

/* CDL include */
#include <cdl.h>
/* Lowell Acquisition Include */
#include <lois.h>
```

```
/* LOAS Includes */
#include <LowellCCD.h>
#include <LowellTel.h>
#include <LowellInst.h>
#include <ModThrd.h>

#define SURVEY 0
#define REMOVE 1

static info_struct info;
const char control_msg[] = ".control.message#1 configure -text";
static pthread_mutex_t *control_lock; /* Mutex to ensure that only one
                                         abort_cmd or clear_queues is
                                         going on at any time between
                                         the two interpreters */

int Remove_Command (ClientData clientdata, Tcl_Interp *interp,
                    int objc, Tcl_Obj *CONST objv[]);
int Kill_Script (ClientData clientdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj *CONST objv[]);
int Reset_Queues (ClientData clientdata, Tcl_Interp *interp,
                  int objc, Tcl_Obj *CONST objv[]);
void flush_pipes(long p_tag);

int Control_Init(Tcl_Interp *interp)
{

  char ctl_string[80];
  Tcl_Obj *ctl_obj, *clear_obj, *abort_obj;
  Tcl_Interp *slv_interp;

  Tcl_CreateObjCommand(interp, "lois_remove", Remove_Command,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "abort_cmd", Kill_Script,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_CreateObjCommand(interp, "clear_queues", Reset_Queues,
                       (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
  Tcl_PkgProvide(interp, "control", "1.0");
  ctl_obj = Tcl_NewStringObj(ctl_string, sizeof(ctl_string));

  if ((slv_interp = Tcl_GetSlave(interp, "command")) == NULL) {
    lois_log0("Error getting slave interpreter in control library!");
    return(TCL_ERROR);
  }

  abort_obj = Tcl_NewStringObj("command", -1);
  clear_obj = Tcl_NewStringObj("command", -1);

  if (Tcl_CreateAliasObj(slv_interp, "abort_cmd", interp, "abort_cmd",
                         1, &abort_obj) == TCL_ERROR) {
    lois_log0("Error aliasing lois_remove command in slave interpreter!");
    lois_log0("Error: %s", Tcl_GetStringResult(slv_interp));
  }

  if (Tcl_CreateAliasObj(slv_interp, "clear_queues", interp, "clear_queues",
                         1, &clear_obj) == TCL_ERROR) {
    lois_log0("Error aliasing clear_queues command in slave interpreter!");
    lois_log0("Error: %s", Tcl_GetStringResult(slv_interp));
  }

  control_lock = (pthread_mutex_t *) ckalloc(sizeof(pthread_mutex_t));
  pthread_mutex_init(control_lock, NULL);

/*
*
* Initialize the shared memory mapping for the Information
* Structures.
*
*/

#ifdef __LINUX__

  if (( info.mem_fd=shmget(INFO_KEY,sizeof(info), SHM_R)) < 0) {
    perror("Cannot Open the Information Shared Memory Buffer");
    printf("Error no %d\n",errno);
    return(TCL_ERROR);
  }


  info.memory = shmat(info.mem_fd, 0, 0);

  if (info.memory == (void *) -1) {
    perror("Memory Map failed for Instrument Buffer");
    printf("Error no %d\n", errno);
    return(TCL_ERROR);
  }

#elif defined(__SOLARIS_5x__)

  if (( info.mem_fd=shm_open("/information", O_RDWR, S_IRWXU)) < 0 ) {
    perror("Cannot Open the information Shared Memory Buffer");
    printf("Error no %d\n",errno);
    exit(-1);
  }


  info.memory=mmap(NULL, sizeof(info), PROT_READ | PROT_WRITE,MAP_SHARED,
                   info.mem_fd, 0);

  if (info.memory == NULL) {
    perror("Memory Map failed for Instrument Buffer");
    return(TCL_ERROR);
  }

#endif

  shm_read((void *)&info,info.memory, sizeof(info));

  sprintf(ctl_string, "source %s/scripts/control.tcl", info.loishome);
  Tcl_SetStringObj(ctl_obj, ctl_string, -1);
  if (Tcl_EvalObj(interp, ctl_obj) == TCL_ERROR) {
    lois_log0("Error sourcing control script!");
    return(TCL_ERROR);
  }

  Tcl_SetStringObj(ctl_obj, "control_init", -1);
  if (Tcl_EvalObj(interp, ctl_obj) == TCL_ERROR) {
    lois_log0("Error initializing control interface!");
    return(TCL_ERROR);
  }

  command_removed_flag = 0;

  return(TCL_OK);

}

int Remove_Command(ClientData clientdata, Tcl_Interp *interp,
                   int objc, Tcl_Obj *CONST objv[])
{

  int c, t, i; /* number of commands found in the ccd, telescope and instrument
```

```
                  module threads, respectively during the survey run */
  long proc_tag; /* process tag of command to remove */
  int done_disp = 0, count;
  char rm_msg[200];
  Tcl_Obj *rm_obj;

  rm_obj = Tcl_NewStringObj(rm_msg, sizeof(rm_msg));
  if (Tcl_GetLongFromObj(interp, objv[1], &proc_tag) == TCL_ERROR) {
    sprintf(rm_msg, "%s {Error getting process tag in lois_remove!}", control_ms
g);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    return(TCL_OK);
  }

  switch (cmd_remove(proc_tag)) {
  case -1:
    sprintf(rm_msg, "%s {Error checking secondary command queue}", control_msg);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    return(TCL_OK);
    break;
  case 0:
    break;
  case 1:
    sprintf(rm_msg, "%s {Removed process tag #%d from secondary command queue}",

            control_msg, proc_tag);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    return(TCL_OK);
    break;
  case 2:
    sprintf(rm_msg, "%s {Process tag #%d has not yet been issued}",
            control_msg, proc_tag);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    return(TCL_OK);
    break;
  default:
    sprintf(rm_msg, "%s {Unknown return value from cmd_remove!}", control_msg);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    return(TCL_OK);
    break;
  }

  ccd_blocked_state(SAVE_STATE);
  tel_blocked_state(SAVE_STATE);
  inst_blocked_state(SAVE_STATE);
  c = ccd_remove(proc_tag, SURVEY);
  t = tel_remove(proc_tag, SURVEY);
  i = inst_remove(proc_tag, SURVEY);

  switch (c+t+i) {
  case 0:
    for (count = 0; count < NUMTELTHREADS; count++) {
      if (proc_tag == tel_cmds[count].curr_run.pidtag) {
        sprintf(rm_msg, "%s {Command for pidtag #%d is in execution. Use Command
Abort}",
                control_msg, proc_tag);
        Tcl_SetStringObj(rm_obj, rm_msg, -1);
        Tcl_EvalObj(interp, rm_obj);
        done_disp = 1;
        break;
      }
    }
    if (!done_disp) {
      for (count = 0; count < NUMINSTTHREADS; count++) {
        if (proc_tag == inst_cmds[count].curr_run.pidtag) {
          sprintf(rm_msg, "%s {Command for pidtag #%d is in execution. Use Comma
nd Abort}",
                  control_msg, proc_tag);
          Tcl_SetStringObj(rm_obj, rm_msg, -1);
          Tcl_EvalObj(interp, rm_obj);
          done_disp = 1;
          break;
        }
      }
    }
    if (!done_disp) {
      for (count = 0; count < NUMCCDTHREADS; count++) {
        if (proc_tag == ccd_cmds[count].curr_run.pidtag) {
          sprintf(rm_msg, "%s {Command for pidtag #%d is in execution. Use Comma
nd Abort}",
                  control_msg, proc_tag);
          Tcl_SetStringObj(rm_obj, rm_msg, -1);
          Tcl_EvalObj(interp, rm_obj);
          done_disp = 1;
          break;
        }
      }
    }
    if (!done_disp) {
      sprintf(rm_msg, "%s {No command found for associated process tag}", contro
l_msg);
      Tcl_SetStringObj(rm_obj, rm_msg, -1);
      Tcl_EvalObj(interp, rm_obj);
    }
    break;
  case 1:
    if (c == 1) {
      ccd_remove(proc_tag, REMOVE);
      tel_remove(-1*proc_tag, REMOVE);
      inst_remove(-1*proc_tag, REMOVE);
    } else {
      if (t == 1) {
        tel_remove(proc_tag, REMOVE);
        ccd_remove(-1*proc_tag, REMOVE);
        inst_remove(-1*proc_tag, REMOVE);
      } else {
        /* command is in instrument pipes */
        inst_remove(proc_tag, REMOVE);
        tel_remove(-1*proc_tag, REMOVE);
        ccd_remove(-1*proc_tag, REMOVE);
      }
    }
    sprintf(rm_msg, ".control.message#1 configure -text {Command for pidtag #%d
removed}", proc_tag);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    break;
  default:
    sprintf(rm_msg, ".control.message#1 configure -text {Multiple commands have
pidtag #%d.  Cannot
use lois_remove}", proc_tag);
    Tcl_SetStringObj(rm_obj, rm_msg, -1);
    Tcl_EvalObj(interp, rm_obj);
    break;
  }
  printf("----------------------------------------\n");
  inst_blocked_state(RESTORE_STATE);
```

```
  tel_blocked_state(RESTORE_STATE);
  ccd_blocked_state(RESTORE_STATE);
  return(TCL_OK);

}

int Kill_Script(ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{

  long proc_tag;
  int ccd_waiting=0, tel_waiting=0, inst_waiting=0;
  int graphics_flag; /* flag indicating if we were called from graphics interpre
ter */
  int wait_flag; /* flag indicating if secondary interpreter is running a wait-f
lagged
                        command */
  int count, count2;
  Tcl_Obj *kill_obj;
  char    kill_str[120], *interp_name;

  pthread_mutex_lock(control_lock);

  if (objc != 3) {
    lois_log0("Error: Wrong number of arguments to abort_command!");
    pthread_mutex_lock(control_lock);
    return(TCL_ERROR);
  }

  /* First get the argument to see which interpreter we were called from */

  if ((interp_name = Tcl_GetStringFromObj(objv[1], NULL)) == NULL) {
    lois_log0("Error getting interp argument in abort_cmd!");
    pthread_mutex_unlock(control_lock);
    return(TCL_ERROR);
  }

  if (strcmp(interp_name, "command") == 0) {
    graphics_flag = 0;
  } else {
    if (strcmp(interp_name, "graphics") == 0) {
      graphics_flag = 1;
    } else {
      lois_log0("Error: Unknown interp argument in reset_queues!");
      pthread_mutex_unlock(control_lock);
      return(TCL_ERROR);
    }
  }

  /* Block secondary command queue so that no more commands can be executed */

  if (graphics_flag) {
    if (cmd_suspend() < 0) {
      lois_log0("Error suspending secondary command queue!");
      pthread_mutex_unlock(control_lock);
      return(TCL_ERROR);
    }
  }

  /* Get proc_tag argument */

  if (Tcl_GetLongFromObj(interp, objv[2], &proc_tag) == TCL_ERROR) {
    lois_log0("Error getting argument in kill script!");
    pthread_mutex_unlock(control_lock);
    return(TCL_ERROR);
  }
  if (!(proc_tag > 0)) {
    lois_log0("Error: Invalid process tag argument to abort_command!");
    pthread_mutex_unlock(control_lock);
    return(TCL_ERROR);
  }

  /* Check if command with desired pidtag is still in the secondary command
     queue, and remove it if it is */

  kill_obj = Tcl_NewStringObj(kill_str, sizeof(kill_str));

  switch (cmd_remove(proc_tag)) {
  case -1:
    sprintf(kill_str, "%s (Error checking secondary command queue)", control_msg
);
    Tcl_SetStringObj(kill_obj, kill_str, -1);
    Tcl_EvalObj(interp, kill_obj);
    pthread_mutex_unlock(control_lock);
    return(TCL_OK);
    break;
  case 0:
    break;
  case 1:
    sprintf(kill_str, "%s (Removed process tag #%d from secondary command queue)
",
            control_msg, proc_tag);
    Tcl_SetStringObj(kill_obj, kill_str, -1);
    Tcl_EvalObj(interp, kill_obj);
    pthread_mutex_unlock(control_lock);
    return(TCL_OK);
    break;
  case 2:
    sprintf(kill_str, "%s (Process tag #%d has not yet been issued)",
            control_msg, proc_tag);
    Tcl_SetStringObj(kill_obj, kill_str, -1);
    Tcl_EvalObj(interp, kill_obj);
    pthread_mutex_unlock(control_lock);
    return(TCL_OK);
    break;
  default:
    sprintf(kill_str, "%s (Unknown return value from cmd_remove!)", control_msg)
;
    Tcl_SetStringObj(kill_obj, kill_str, -1);
    Tcl_EvalObj(interp, kill_obj);
    pthread_mutex_unlock(control_lock);
    return(TCL_OK);
    break;
  }

  /* Block the CCD, telescope and instrument module thread queues so that if the
     script enters any other commands, they too are blocked */

  ccd_blocked_state(SAVE_STATE);
  tel_blocked_state(SAVE_STATE);
  inst_blocked_state(SAVE_STATE);

  /* Cancel a module routine if it was waiting for blocks on other modules to
     go through by posting semaphores */

  command_removed_flag = 1;

  if (ccd_cmds[1].curr_run.pidtag == proc_tag) {

    /* Only want to post these semaphores if the pidtag of the command we're
       removing is in the EXPOSURE thread.  If it's in another thread, and
```

```
       we post the semaphores, then an exposure command with a different pidtag
       could start execution, which might screw up the timing. */

    tel_waiting++;
    for (count = 0; count < NUMTELTHREADS; count++) {
      sem_post(tel_block[count]);
    }
    inst_waiting++;
    for (count = 0; count < NUMINSTTHREADS; count++) {
      sem_post(inst_block[count]);
    }
  }

  /* Don't want to post these semaphores since telescope and instrument commands
     don't wait on semaphores.  If, in the future, a telescope command comes alo
ng
     that needs to block the ccd and instrument threads (so it would wait on the
     ccd and instrument semaphores), then the thread running that command would
     have to be checked (like above for the exposure thread in the CCD module).
     For example, suppose that a move command requires the ccd and instrument
     threads blocked.  It would issue blocks and wait on the ccd and inst semaph
ores.
     Therefore, it will only start when the CCD and instrument threads are pause
d.
     Thus, if the CCD exposure thread has a command waiting on semaphores, that
means
     blocks associated with the expose command were entered before blocks for th
e
     telescope command (since the blocks are sent by the command interpreter whi
le it
     runs the TCL wrapper part of the expose command), and thus the telescope mo
ve
     command that needs to wait for the CCD exposure cannot be running (since a
block
     on its thread would have preceded it).

  for (count = 0; count < NUMTELTHREADS; count++) {
    if (tel_cmds[count].curr_run.pidtag == proc_tag) {
      ccd_waiting++;
      for (count = 0; count < NUMCCDTHREADS; count++) {
        sem_post(ccd_block[count]);
      }
      inst_waiting++;
      for (count = 0; count < NUMINSTTHREADS; count++) {
        sem_post(inst_block[count]);
      }
    }
  }
  for (count = 0; count < NUMINSTTHREADS; count++) {
    if (inst_cmds[count].curr_run.pidtag == proc_tag) {
      tel_waiting++;
      for (count = 0; count < NUMTELTHREADS; count++) {
        sem_post(tel_block[count]);
      }
      ccd_waiting++;
      for (count = 0; count < NUMCCDTHREADS; count++) {
        sem_post(inst_block[count]);
      }
    }
  }

  */

  /* Abort the currently executing routines if their process tags match the
     process tag given as the argument */
  if (strcmp(info.telmod, "none") != 0) {
    if (proc_tag == tel_cmds[0].curr_run.pidtag) {
      if (proc_tag == tel_cmds[1].curr_run.pidtag) {

        /* MOVE & FOCUS command module thread in telescope */

        Tcl_SetStringObj(kill_obj, "tel_stop -prio -4", -1);
        if (Tcl_EvalObj(interp, kill_obj) == TCL_ERROR) {
          lois_log0("Error aborting telescope move and focus");
        }
      } else {

        /* MOVE command module thread in telescope */

        Tcl_SetStringObj(kill_obj, "tel_stop move -prio -4", -1);
        if (Tcl_EvalObj(interp, kill_obj) == TCL_ERROR) {
          lois_log0("Error aborting telescope move");
        }
      }
    } else {
      if (proc_tag == tel_cmds[1].curr_run.pidtag) {

        /* FOCUS command module thread in telescope */

        Tcl_SetStringObj(kill_obj, "tel_stop focus -prio -4", -1);
        if (Tcl_EvalObj(interp, kill_obj) == TCL_ERROR) {
          lois_log0("Error aborting telescope focus");
        }
      }
    }
  }
  if (strcmp(info.instmod, "none") != 0) {
    if (proc_tag == inst_cmds[0].curr_run.pidtag) {

      /* FILTER command module thread in instrument */

      Tcl_SetStringObj(kill_obj, "inst_stop -prio -4", -1);
      if (Tcl_EvalObj(interp, kill_obj) == TCL_ERROR) {
        lois_log0("Error aborting instrument command");
      }
    }
  }
  if (strcmp(info.ccdmod, "none") != 0) {
    if (proc_tag == ccd_cmds[1].curr_run.pidtag) {

      /* EXPOSURE command module thread in CCD */

      Tcl_SetStringObj(kill_obj, "ccd_abort -prio -4", -1);
      if (Tcl_EvalObj(interp, kill_obj) == TCL_ERROR) {
        lois_log0("Error aborting camera exposure");
      }
    }
  }

  /* These next three for statements check to see if a wait-flagged command is
     running in one of the modules.  If a wait-flagged command were running and
     had a pidtag equal to the pidtag we want to remove, then it would have been
     aborted (before).  If the wait-flagged command has a pidtag not equal to th
e
     one we want to remove, then it will continue to run.  Since the command was
     wait-flagged, it is in effect tying up the secondary interpreter, so we don
't
     need to wait for it below */

  for (count = 0; count < NUMCCDTHREADS; count++) {
    if (ccd_cmds[count].curr_run.command < 0) {
```

```
      wait_flag = 1;
    }
  }
  for (count = 0; count < NUMTELTHREADS; count++) {
    if (tel_cmds[count].curr_run.command < 0) {
      wait_flag = 1;
    }
  }
  for (count = 0; count < NUMINSTTHREADS; count++) {
    if (inst_cmds[count].curr_run.command < 0) {
      wait_flag = 1;
    }
  }

  /* Remove all associated commands with the proper pidtag - this will return an
     error to the interpreter if it was waiting on a command */

  sprintf(kill_str, "%s {Removing command for pidtag #%d...}", control_msg, proc
_tag);
  Tcl_SetStringObj(kill_obj, kill_str, -1);
  Tcl_EvalObj(interp, kill_obj);
  if (graphics_flag && !wait_flag) {

    /* If we're called from the secondary interpreter, we know that the secondar
y
       interpreter is already waiting, so we don't need to do this while loop */

    while (!cmd_iswaiting()) {

      flush_pipes(proc_tag);
      usleep(50000); /* This sleep command is here to make the kill_script comma
nd
                        wait for the interpreter to finish entering commands.  I
f
                        the script has a lot of async. commands, then the interp
reter
                        will enter all of them before halting.  It may happen th
at
                        the interpreter will enter more commands after the queue
s
                        have been adjusted.  Since the queues might have to be
                        adjusted after
                        the interpreter enters the commands, continually adjusti
ng the
                        queues wastes processor cycles.  This sleep frees up cyc
les
                        for the interpreter to enter commands, then the kill rou
tine
                        wakes up and flushes those commands.  The time is quite
                        arbitrary, 1/20 of a second was chosen so that the user
                        doesn't notice it too much, but the system can get a lot
                        done in that time.
                     */
    }
  }
  flush_pipes(proc_tag); /* Flush the pipes one last time, in case the interpret
er entered
                            commands and then came to a halt while the kill rout
ine was
                            sleeping */

  /* Reset semaphore values that were changed earlier */

  for (count = 0; count < tel_waiting; count++) {
    for (count2 = 0; count < NUMTELTHREADS; count++) {
      if (sem_trywait(tel_block[count2]) < 0) {
        if (errno != EAGAIN) {
          lois_log1("Error resetting telescope semaphore #%d", count2);
        }
      }
    }
  }
  for (count = 0; count < ccd_waiting; count++) {

    /* ccd_waiting should be zero unless a command is changed (see the comment a
bout
       a move command that issues blocks) */

    for (count2 = 0; count < NUMCCDTHREADS; count++) {
      if (sem_trywait(ccd_block[count2]) < 0) {
        if (errno != EAGAIN) {
          lois_log1("Error resetting ccd semaphore #%d", count2);
        }
      }
    }
  }
  for (count = 0; count < inst_waiting; count++) {
    for (count2 = 0; count < NUMINSTTHREADS; count++) {
      if (sem_trywait(inst_block[count2]) < 0) {
        if (errno != EAGAIN) {
          lois_log1("Error resetting instrument semaphore #%d", count2);
        }
      }
    }
  }

  command_removed_flag = 0;

  /* Restore pipes and allow activate commands */

  inst_blocked_state(RESTORE_STATE);
  tel_blocked_state(RESTORE_STATE);
  ccd_blocked_state(RESTORE_STATE);

  /* Restart the secondary command queue */

  if (graphics_flag) {
    if (cmd_resume() < 0) {
      lois_log0("Error resuming secondary command queue!");
      pthread_mutex_unlock(control_lock);
      return(TCL_ERROR);
    }
  }

  pthread_mutex_unlock(control_lock);
  return(TCL_OK);

}

int Reset_Queues(ClientData clientdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj *CONST objv[])
{

  Tcl_Obj *reset_obj;
  char     reset_str[120], *interp_name;
  int      count, wait_flag =0 ;
  int      graphics_flag; /* flag saying if we should try to suspend the
                             secondary command queue */

  pthread_mutex_lock(control_lock);

  /* First get the argument to see which interpreter we were called from */
```

```
  if ((interp_name = Tcl_GetStringFromObj(objv[1], NULL)) == NULL) {
    lois_log0("Error getting interp argument in clear_queues!");
    pthread_mutex_unlock(control_lock);
    return(TCL_ERROR);
  }

  if (strcmp(interp_name, "command") == 0) {
    graphics_flag = 0;
  } else {
    if (strcmp(interp_name, "graphics") == 0) {
      graphics_flag = 1;
    } else {
      lois_log0("Error: Unknown interp argument in reset_queues!");
      pthread_mutex_unlock(control_lock);
      return(TCL_ERROR);
    }
  }

  /* Block secondary command queue so that no more commands can be executed */

  if (graphics_flag) {
    if (cmd_suspend() < 0) {
      lois_log0("Error suspending secondary command queue!");
      pthread_mutex_unlock(control_lock);
      return(TCL_ERROR);
    }
  }

  /* Block the CCD, telescope and instrument module thread queues so that if the
     script enters any other commands, they too are blocked */

  ccd_blocked_state(SAVE_STATE);
  tel_blocked_state(SAVE_STATE);
  inst_blocked_state(SAVE_STATE);

  /* Cancel a command (like a camera exposure) if it was waiting on blocks...
     here we just post everything since every command in the system is going
     to be removed anyway.  We only have to post each semaphore once because
     only one command can be waiting on the semaphores at any given time
     (because the blocks sent by the commands go into the same queue as the
     commands themselves - this assumes that there are multiple commands that
     issue blocks and they are all entered at normal priority.)
  */

  command_removed_flag = 1;
  for (count = 0; count < NUMTELTHREADS; count++) {
    sem_post(tel_block[count]);
  }
  for (count = 0; count < NUMINSTTHREADS; count++) {
    sem_post(inst_block[count]);
  }
  for (count = 0; count < NUMCCDTHREADS; count++) {
    sem_post(ccd_block[count]);
  }

  /* Terminate the currently executing function - send in at system only priorit
y */

  reset_obj = Tcl_NewStringObj(reset_str, sizeof(reset_str));
  if (strcmp(info.telmod, "none") != 0) {
    Tcl_SetStringObj(reset_obj, "tel_stop -prio -4", -1);
    if (Tcl_EvalObj(interp, reset_obj) == TCL_ERROR) {
      lois_log0("Error aborting telescope move");
    }
  }
  if (strcmp(info.instmod, "none") != 0) {
    Tcl_SetStringObj(reset_obj, "inst_stop -prio -4", -1);
    if (Tcl_EvalObj(interp, reset_obj) == TCL_ERROR) {
      lois_log0("Error aborting instrument command");
    }
  }
  if (strcmp(info.ccdmod, "none") != 0) {
    Tcl_SetStringObj(reset_obj, "ccd_abort -prio -4", -1);
    if (Tcl_EvalObj(interp, reset_obj) == TCL_ERROR) {
      lois_log0("Error aborting camera exposure");
    }
  }

  /* If a command is running with negative priority, then we don't have to
     wait for the graphics interpreter to stop */

  for (count = 0; count < NUMCCDTHREADS; count++) {
    if (ccd_cmds[count].curr_run.command < 0) {
      wait_flag = 1;
    }
  }
  for (count = 0; count < NUMTELTHREADS; count++) {
    if (tel_cmds[count].curr_run.command < 0) {
      wait_flag = 1;
    }
  }
  for (count = 0; count < NUMINSTTHREADS; count++) {
    if (inst_cmds[count].curr_run.command < 0) {
      wait_flag = 1;
    }
  }

  /* Remove all associated commands (or flush pipes) - this will return an
     error to the interpreter if it was waiting on a command */

  if (graphics_flag && !wait_flag) {

    /* If we're called from the secondary interpreter, we know that the secondar
y
       interpreter is already waiting, so we don't need to do this while loop */

    while (!cmd_iswaiting()) {

      flush_pipes(0);
      usleep(50000); /* This sleep command is here to make the kill_script comma
nd
                        wait for the interpreter to finish entering commands.  I
f
                        the script has a lot of async. commands, then the interp
reter
                        will enter all of them before halting.  It may happen th
at
                        the interpreter will enter more commands after the queue
s
                        have been cleared.  Since the queues must be cleared aft
er
                        the interpreter enters the commands, continually flushin
g the
                        queues wastes processor cycles.  This sleep frees up cyc
les
                        for the interpreter to enter commands, then the kill rou
tine
                        wakes up and flushes those commands.  The time is quite
                        arbitrary, 1/20 of a second was chosen so that the user
                        doesn't notice it too much, but the system can get a lot
                        done in that time.
```

```
                       */
    }
  }
  flush_pipes(0); /* Flush the pipes one last time, in case the interpreter ente
red
                     commands and then came to a halt while the kill routine was
                     sleeping */

  /* Reset semaphores to what they were before if there was no camera exposure
     occurring */

  for (count = 0; count < NUMTELTHREADS; count++) {
    if (sem_trywait(tel_block[count]) < 0) {
      if (errno != EAGAIN) {
        lois_log1("Error resetting telescope block #%d", count);
      }
    }
  }
  for (count = 0; count < NUMINSTTHREADS; count++) {
    if (sem_trywait(inst_block[count]) < 0) {
      if (errno != EAGAIN) {
        lois_log1("Error resetting instrument block #%d", count);
      }
    }
  }
  for (count = 0; count < NUMCCDTHREADS; count++) {
    if (sem_trywait(ccd_block[count]) < 0) {
      if (errno != EAGAIN) {
        lois_log1("Error resetting ccd block #%d", count);
      }
    }
  }

  /* Restore pipes and allow activate commands */

  inst_blocked_state(RESTORE_STATE);
  tel_blocked_state(RESTORE_STATE);
  ccd_blocked_state(RESTORE_STATE);

  /* Reactivate any queues that were blocked when the command was aborted. We
     don't care if any queues were blocked before, since there are no commands
     in them now. */

  if (strcmp(info.telmod, "none") != 0) {
    Tcl_SetStringObj(reset_obj, "tel_activate", -1);
    if (Tcl_EvalObj(interp, reset_obj) == TCL_ERROR) {
      lois_log0("Error restarting telescope queues");
    }
  }
  if (strcmp(info.instmod, "none") != 0) {
    Tcl_SetStringObj(reset_obj, "inst_activate", -1);
    if (Tcl_EvalObj(interp, reset_obj) == TCL_ERROR) {
      lois_log0("Error restarting instrument queues");
    }
  }
  if (strcmp(info.ccdmod, "none") != 0) {
    Tcl_SetStringObj(reset_obj, "ccd_activate", -1);
    if (Tcl_EvalObj(interp, reset_obj) == TCL_ERROR) {
      lois_log0("Error restarting ccd queues");
    }
  }

  command_removed_flag = 0;

  /* Restart the secondary command queue */
  if (graphics_flag) {
    if (cmd_resume() < 0) {
      lois_log0("Error resuming secondary command queue!");
      pthread_mutex_unlock(control_lock);
      return(TCL_ERROR);
    }
  }
  Tcl_SetResult(interp, "Queues flushed", NULL);
  pthread_mutex_unlock(control_lock);
  return(TCL_OK);
}

void flush_pipes(long p_tag) {

  /* This routine removes all occurrences of commands with process tag
     p_tag from the queues.  If p_tag is 0, then it removes all commands
     from the queues */

  /* It might have to adjust for blocks (in queues and in effect as curr_run
     commands) in the case where the argument is non-zero */

  tel_flush(p_tag);
  inst_flush(p_tag);
  ccd_flush(p_tag);

}
```

```
/*****************************************************************************
                           Lowell Observatory
                    CCD Acquisitions Software Package


Program:        Miscellaneous Function Library for LOIS
Name:           misc
By:             Adam Gould
Started:        02/12/99
Library:        libmisc.so

Explanation:    This library contains the lois_send and prioritized
                logging functions.

        Copyright 1998, Lowell Observatory, All Rights Reserved

-------------------------------------------------------------------------------

Change Log:

05/20/98: Started the program


*****************************************************************************/

#include <tcl.h>
#include <tk.h>
#include <cdl.h>

#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <lois.h>
#include <LowellCCD.h>
#include <LowellTel.h>

#ifdef __SOLARIS_5x__
#define MSG_Q_SIZE 1024
#include <sys/varargs.h>
#elif defined(__LINUX__)
#include <stdarg.h>
#endif

struct lois_msgbuf {
  long mtype;
  char command[MSG_Q_SIZE];
};

extern resq results_queue;
extern long cur_eval_pidtag;
extern long cur_issue_pidtag;
extern command_queue other_commands;

extern pthread_mutex_t *dummy_mutex;
extern pthread_cond_t *dummy_cond;

void * interval_cmd(void * cmdptr);

long get_curevalpidtag() {
  return(cur_eval_pidtag);
}
/*
 * Lois_send to send commands to the Queue based on
 * Priority, time of execution, number of times to execute
 * and time interval between command execution.
 */

#ifdef __LINUX__

/* LINUX version */

int lois_send(cmd_struct * lois_cmd)

{

  /* 2/19/99 - Added mutexes and condition variables to graphics queue, no error
     checking for a full queue, though.  Also trying to add capability of return
     values so calling functions can get results back from queue */

  int result, ret_val=0;
  pthread_t interval_thread;
  static struct lois_msgbuf send_cmd;
  struct resq_elt *new_ptr, *rq_ptr;

  rq_ptr=results_queue.head;

  if (lois_cmd->t_interval != 0 ) {

      result=pthread_create(&interval_thread, NULL,
                            interval_cmd, (void *) lois_cmd);
      if (result != 0) {
        printf("Error: Problem create interval thread\n");
        return(-1);
      }
      return(interval_thread);
  } else {

    if (lois_cmd->priority > MAX_PRIORITY) {
        printf("Error: Priority for command set to high\n");
        return(-1);
    } else {

      pthread_mutex_lock(results_queue.mutex);

      if (lois_cmd->priority < 0) {
        send_cmd.mtype = 31+lois_cmd->priority;
        sprintf(send_cmd.command, "%d+=+%s\0", results_queue.cur_count, lois_cmd
->command);
        new_ptr = (struct resq_elt *) ckalloc(sizeof(struct resq_elt));
        new_ptr->next = NULL;
        new_ptr->key = results_queue.cur_count;
        (new_ptr->value).rtn_status = WAIT_STATUS;
        strncpy((new_ptr->value).rtn_message, "", 128);
      } else {
        send_cmd.mtype = 31-lois_cmd->priority;
        sprintf(send_cmd.command, "%s\0", lois_cmd->command);
      }

      result=msgsnd( lois_qid, &send_cmd, MSG_Q_SIZE, IPC_NOWAIT);

      if (result !=0) {
        pthread_mutex_unlock(results_queue.mutex);
        if (errno == EAGAIN) {
          printf("Queue full: Command '%s' could not be enqueued\n", send_cmd.co
mmand);
          return(-1);
        } else {
          perror("Error sending to Message Queue");
          printf("Error number %d\n", errno);
          return(-1);
        }
```

```
      } else {
        if (lois_cmd->priority < 0) {
          while ((rq_ptr->next) != NULL) {
            rq_ptr = rq_ptr->next;
          }
          rq_ptr->next = new_ptr;
          ret_val = results_queue.cur_count;
          results_queue.cur_count++;
        }
        pthread_mutex_unlock(results_queue.mutex);
        return(ret_val);
      }

    }
  }
}
#elif defined (__SOLARIS_5x__)

/* SOLARIS version of lois_send */

int lois_send(cmd_struct * lois_cmd)

{

  int       result, ret_val=0;
  pthread_t interval_thread;
  static cmd_struct send_cmd;
  struct resq_elt *new_ptr, *rq_ptr;

  rq_ptr = results_queue.head;

  if (lois_cmd->t_interval != 0 ) {

    result=pthread_create(&interval_thread, NULL,
                          interval_cmd, (void *) lois_cmd);
    if (result != 0) {
      printf("Error: Problem create interval thread\n");
      return(-1);
    }
    return(interval_thread);
  } else {

    if (lois_cmd->priority > MAX_PRIORITY) {
      printf("Error: Priority for command set to high\n");
      return(-1);
    } else {

      pthread_mutex_lock(results_queue.mutex);

      if (lois_cmd->priority < 0) {
        send_cmd.priority = -1*lois_cmd->priority;
        sprintf(send_cmd.command, "%d+=+%s\0", results_queue.cur_count, lois_cmd
->command);
        new_ptr = (struct resq_elt *) ckalloc(sizeof(struct resq_elt));
        new_ptr->next = NULL;
        new_ptr->key = results_queue.cur_count;
        (new_ptr->value).rtn_status = WAIT_STATUS;
        strncpy((new_ptr->value).rtn_message, "", 128);
      } else {
        send_cmd.priority = lois_cmd->priority;
        sprintf(send_cmd.command, "%s\0", lois_cmd->command);
      }

      result=mq_send( lois_queue, send_cmd.command,
                      sizeof(send_cmd.command), send_cmd.priority);
      if (result !=0) {
        pthread_mutex_unlock(results_queue.mutex);
        if (errno == EAGAIN) {
          printf("Queue full: Command '%s' could not be enqueued\n", send_cmd.co
mmand);
          return(-1);
        } else {
          perror("Error sending to Message Queue");
          printf("Error number %d\n", errno);
          return(-1);
        }
      } else {
        if (lois_cmd->priority < 0) {
          while ((rq_ptr->next) != NULL) {
            rq_ptr = rq_ptr->next;
          }
          rq_ptr->next = new_ptr;
          ret_val = results_queue.cur_count;
          results_queue.cur_count++;
        }
        pthread_mutex_unlock(results_queue.mutex);
        return(ret_val);
      }
    }
  }
}

#endif

int lois_enqueue(int key, lois_results in_value) {

  struct resq_elt *traverse;

  traverse = results_queue.head;
  pthread_mutex_lock(results_queue.mutex);
  while (traverse->key != key) {
    if ((traverse = traverse->next) == NULL) {
      pthread_mutex_unlock(results_queue.mutex);
      return(-1);
    }
  }
  traverse->value.rtn_status = in_value.rtn_status;
  sprintf(traverse->value.rtn_message, in_value.rtn_message);
  pthread_cond_signal(results_queue.go);
  pthread_mutex_unlock(results_queue.mutex);
  return(0);
}

void lois_receive(int key, lois_results *answer) {

  struct resq_elt *previous, *traverse = results_queue.head;

  pthread_mutex_lock(results_queue.mutex);
  while (traverse->key != key) {
    if (traverse->next == NULL) {
      answer->rtn_status = -1;
      sprintf(answer->rtn_message, "Error: Key not defined");
      pthread_mutex_unlock(results_queue.mutex);
      return;
    }
    previous = traverse;
    traverse = traverse->next;
  }

  while (traverse->value.rtn_status == WAIT_STATUS) {
    pthread_cond_wait(results_queue.go, results_queue.mutex);
```

```
  }

  if (previous == NULL) {
    printf("uh oh\n");
  }
  previous->next = traverse->next;
  bcopy((void *) &(traverse->value), (void *) answer, sizeof(lois_results));

  pthread_cond_signal(results_queue.go);
  pthread_mutex_unlock(results_queue.mutex);
  ckfree(traverse);
  return;
}

void * interval_cmd(void * cmdptr)
{

  int             count=1;
  cmd_struct      lois_qcmd;

  /* First copy the command into the execution structure
   * so it will not get overwritten
   */
  bcopy((void *) cmdptr, (void *) &lois_qcmd, sizeof(cmdptr));

}


int lois_log0(char * message, ...) {

  int             result;
  static struct lois_msgbuf   send_cmd;
  static char     msgcmd[160];
  va_list ap_ptrs;

#ifdef __SOLARIS_5x__
  sprintf(msgcmd, "log_0 %s\0", message);
  result=mq_send(lois_queue, msgcmd, sizeof(msgcmd), MAX_PRIORITY);
#elif defined(__LINUX__)
  sprintf(send_cmd.command, "log_0  \0");
  va_start(ap_ptrs, message);
  vsprintf(send_cmd.command+7, message, ap_ptrs);
  va_end(ap_ptrs);
  send_cmd.mtype = 1;
  result=msgsnd(lois_qid, &send_cmd, MSG_Q_SIZE, 0);
#endif

  if (result != 0) {
    fprintf(stderr, "\aERROR: Unable to send log_0 message\n");
    fprintf(stderr, "\aERROR: Message=%s\n", message);
    return(-1);
  }
  return(0);
}
int lois_log1(char * message, ...) {

  int             result;
  static struct lois_msgbuf   send_cmd;
  static char     msgcmd[160]="log_1  \0";
  va_list         ap_ptrs;

#ifdef __SOLARIS_5x__
  va_start(ap_ptrs, message);
  vsprintf(msgcmd+7, message, ap_ptrs);
  va_end(ap_ptrs);
  result=mq_send(lois_queue, msgcmd, sizeof(msgcmd), MAX_PRIORITY);
```

```
#elif defined(__LINUX__)

  sprintf(send_cmd.command, "log_1  \0");
  va_start(ap_ptrs, message);
  vsprintf(send_cmd.command+7, message, ap_ptrs);
  va_end(ap_ptrs);
  send_cmd.mtype = 1;
  result=msgsnd(lois_qid, &send_cmd, MSG_Q_SIZE, 0);
#endif

  if (result != 0) {
    fprintf(stderr, "\aERROR: Unable to send log_1 message\n");
    fprintf(stderr, "\aERROR: Message=%s\n", message);
    return(-1);
  }
  return(0);
}

int lois_log2(char * message, ...) {

  int            result;
  static struct lois_msgbuf   send_cmd;
  static char    msgcmd[160]="log_2  \0";
  va_list        ap_ptrs;


#ifdef __SOLARIS_5x__
  va_start(ap_ptrs, message);
  vsprintf(msgcmd+7, message, ap_ptrs);
  va_end(ap_ptrs);
  result=mq_send(lois_queue, msgcmd, sizeof(msgcmd), DEF_PRIORITY);

#elif defined(__LINUX__)
  sprintf(send_cmd.command, "log_2  \0");
  va_start(ap_ptrs, message);
  vsprintf(send_cmd.command+7, message, ap_ptrs);
  va_end(ap_ptrs);
  send_cmd.mtype = 16;
  result=msgsnd(lois_qid, &send_cmd, MSG_Q_SIZE, 0);
#endif

  if (result != 0) {
    fprintf(stderr, "\aERROR: Unable to send log_2 message\n");
    fprintf(stderr, "\aERROR: Message=%s\n", message);
    return(-1);
  }
  return(0);
}
int lois_log3(char * message, ...) {


  int            result;
  static struct lois_msgbuf   send_cmd;
    static char    msgcmd[160]="log_3  \0";
  va_list        ap_ptrs;

#ifdef __SOLARIS_5x__
  va_start(ap_ptrs, message);
  vsprintf(msgcmd+7, message, ap_ptrs);
  va_end(ap_ptrs);
  result=mq_send(lois_queue, msgcmd, sizeof(msgcmd), DEF_PRIORITY);
#elif defined(__LINUX__)
  sprintf(send_cmd.command, "log_3  \0");
  va_start(ap_ptrs, message);
  vsprintf(send_cmd.command+7, message, ap_ptrs);
  va_end(ap_ptrs);
```

```
  send_cmd.mtype = 16;
  result=msgsnd(lois_qid, &send_cmd, MSG_Q_SIZE, 0);
#endif

  if (result != 0) {
    fprintf(stderr, "\aERROR: Unable to send log_3 message\n");
    fprintf(stderr, "\aERROR: Message=%s\n", message);
    return(-1);
  }
  return(0);
}
int lois_log4(char * message, ...) {

  int            result;
  static struct lois_msgbuf   send_cmd;
  static char    msgcmd[160]="log_4  \0";
  va_list        ap_ptrs;

#ifdef __SOLARIS_5x__
  va_start(ap_ptrs, message);
  vsprintf(msgcmd+7, message, ap_ptrs);
  va_end(ap_ptrs);
  result=mq_send(lois_queue, msgcmd, sizeof(msgcmd), MIN_PRIORITY);
#elif defined(__LINUX__)
  sprintf(send_cmd.command, "log_4  \0");
  va_start(ap_ptrs, message);
  vsprintf(send_cmd.command+7, message, ap_ptrs);
  va_end(ap_ptrs);
  send_cmd.mtype = 31;
  result=msgsnd(lois_qid, &send_cmd, MSG_Q_SIZE, 0);
#endif

  if (result != 0) {
    fprintf(stderr, "\aERROR: Unable to send log_4 message\n");
    fprintf(stderr, "\aERROR: Message=%s\n", message);
    return(-1);
  }
  return(0);
}
int lois_log5(char * message, ...) {

  int            result;
  static struct lois_msgbuf   send_cmd;
  static char    msgcmd[160]="log_5  \0";
  va_list        ap_ptrs;

#ifdef __SOLARIS_5x__
  va_start(ap_ptrs, message);
  vsprintf(msgcmd+7, message, ap_ptrs);
  va_end(ap_ptrs);
  result=mq_send(lois_queue, msgcmd, sizeof(msgcmd), MIN_PRIORITY);
#elif defined(__LINUX__)

  sprintf(send_cmd.command, "log_5  \0");
  va_start(ap_ptrs, message);
  vsprintf(send_cmd.command+7, message, ap_ptrs);
  va_end(ap_ptrs);
  send_cmd.mtype = 31;
  result=msgsnd(lois_qid, &send_cmd, MSG_Q_SIZE, 0);
#endif

  if (result != 0) {
    fprintf(stderr, "\aERROR: Unable to send log_5 message\n");
    fprintf(stderr, "\aERROR: Message=%s\n", message);
    return(-1);
  }
  return(0);
}

void shm_read(void *to, void *from, int size) {

  bcopy(from, to, size);
}

void shm_write(void *to, void *from, int size) {

  bcopy(from, to, size);
}

/*
 * Read the parameter file for the Camera Module
 *
 */

int ccd_readparameters (char * param,
                   ccd_struct * ccd_ptr, info_struct * info_ptr )

{

  FILE  *cfg_ptr;
  int   flag=0, count;
  short tmp_val;
  static char *file_ptr, line[120], keyword[20], int_value[10], comment[80];
  char *pos_ptr1, *pos_ptr2;


  if (strlen(info_ptr->cfghome) > strlen(info_ptr->loishome))
    file_ptr=(char *)malloc(strlen(info_ptr->cfghome)+sizeof(param));
  else file_ptr=(char *)malloc(strlen(info_ptr->loishome)+sizeof(param));

  bzero(file_ptr, sizeof(strlen(info_ptr->cfghome)+sizeof(param)));

  if (file_ptr == NULL) {
    lois_log0("ERROR: Unable to malloc path name memory for param path for readi
ng");
    return(-1);
  }
 bzero(file_ptr, sizeof(strlen(info_ptr->cfghome)+sizeof(param)));
  file_ptr=strcat(file_ptr, info_ptr->cfghome);
  file_ptr=strcat(file_ptr, "/");
  file_ptr=strcat(file_ptr, param);
  file_ptr=strcat(file_ptr, ".param");
  cfg_ptr=fopen(file_ptr , "r+");

  if (cfg_ptr == NULL) {
    bzero(file_ptr, sizeof(strlen(info_ptr->cfghome)+sizeof(param)));
    file_ptr=strcat(file_ptr, info_ptr->loishome);
    file_ptr=strcat(file_ptr, "/etc/");
    file_ptr=strcat(file_ptr, param);
    file_ptr=strcat(file_ptr, ".param");
    cfg_ptr=fopen(file_ptr, "r+");
    if (cfg_ptr == NULL) {
      lois_log0("ERROR: Unable to get %s Parameter File", file_ptr);
      return(-1);
    }
  }
  fgets(line, sizeof(line), cfg_ptr); /* Get the first Line in the File */
  /* First Read the Interger Definitions from the Parameter File */
 for (count = 0 ; count < CCD_INT_COUNT ; count++)
   {
     bzero(keyword, sizeof(keyword));
     bzero(comment, sizeof(comment));
```

```
     fgets(line, sizeof(line), cfg_ptr);
     printf("Count:%d Line:%s\n", count, line);
     if ((pos_ptr1=strstr(line, "=")) == NULL) {
          lois_log0("Error reading parameter file");
          return(-1);
     }
     strncpy(keyword, line,abs(pos_ptr1-line));
     if ((pos_ptr2=strstr(pos_ptr1,"#")) == NULL ) {
       lois_log0("Error reading parameter file");
       return(-1);
     }
     strncpy(int_value, pos_ptr1+1, (pos_ptr2-pos_ptr1+1));
     tmp_val=atoi(int_value);
     *((short *)ccd_ptr+count)=tmp_val;
     strcpy(comment, pos_ptr2+1);
     /*
     lois_log4("%s = %d\t#%s\0",keyword, *((short *)ccd_ptr+count), comment);
     */
   }
close(cfg_ptr);
free(file_ptr);
lois_log4("%s File Loaded", param);
return(0);
}

/*
 * Write the parameter file for the Camera Module
 *
 */
int ccd_writeparam (char * param, ccd_vectors * ccdvec_ptr,
                   ccd_struct * ccd_ptr, info_struct * info_ptr )

{
  FILE  *cfg_ptr;
  int   flag=0, count;
  char *file_ptr;


    file_ptr=(char *)malloc(strlen(info_ptr->cfghome)+sizeof(param));

  if (file_ptr == NULL) {
    lois_log0("ERROR: Unable to malloc path name memory for param path for writi
ng");
    return(-1);
  }

  file_ptr=strncpy(file_ptr, info_ptr->cfghome, strlen(info_ptr->cfghome));
  file_ptr=strcat(file_ptr, "/");
  file_ptr=strcat(file_ptr, param);

  cfg_ptr=fopen(file_ptr , "r+");

  if (cfg_ptr == NULL) {
    file_ptr=strncpy(file_ptr, info_ptr->loishome, strlen(info_ptr->loishome));
    file_ptr=strcat(file_ptr, "/etc/");
    file_ptr=strcat(file_ptr, param);
    cfg_ptr=fopen(file_ptr, "r+");
    if (cfg_ptr == NULL) {
      lois_log0("ERROR: Unable to get %s Parameter File", param);
      return(-1);
    }
  }
}

void lois_sleep(long nanosecs) {
  struct timespec dt_spec;
  struct timeval  now;
  int retval;

  retval = 0;
  gettimeofday(&now, NULL);
  dt_spec.tv_sec = now.tv_sec;
  dt_spec.tv_nsec = nanosecs + now.tv_usec*1000;
  while (retval != ETIMEDOUT) {
    retval = pthread_cond_timedwait(dummy_cond, dummy_mutex, &dt_spec);
  }
  pthread_mutex_unlock(dummy_mutex);
  return;
}


int cmd_iswaiting() {

  return(other_commands.waiting);
}

int cmd_suspend() {

  int result;

  if (pthread_mutex_lock(other_commands.mutex) < 0) return (-1);
  other_commands.blocked = 1;
  if (pthread_mutex_unlock(other_commands.mutex) < 0) return (-1);
  return(0);
}

int cmd_resume() {

  int result;

  if (pthread_mutex_lock(other_commands.mutex) < 0) return (-1);
  other_commands.blocked = 0;
  pthread_cond_signal(other_commands.go);
  if (pthread_mutex_unlock(other_commands.mutex) < 0) return (-1);
  return(0);
}

int cmd_remove(long rm_pidtag) {

  struct _command *traverse, *previous;

  if (pthread_mutex_lock(other_commands.mutex) < 0) return -1;

  /* This routine (for now) doesn't take into account the fact that the
     pidtag "rolls over" at 1000000.  Therefore, if a session runs more than
     one million commands, there will be some commands that cannot be removed/
     aborted, since their pidtags (around 1000000) will be greater than the
     currently evaluating pidtag (which would be around 0) */

  if (rm_pidtag <= get_curevalpidtag()) {
    pthread_mutex_unlock(other_commands.mutex);
    return(0);
  }
  if (rm_pidtag > cur_issue_pidtag) {
    pthread_mutex_unlock(other_commands.mutex);
    return(2);
  }
  traverse = other_commands.head;
  previous = NULL;
  while (traverse != NULL) {
    if (traverse->pidtag == rm_pidtag) {
```

```
      if (previous == NULL) {
        other_commands.head = traverse->next;
        ckfree(traverse);
        traverse = other_commands.head;
      } else {
        ckfree(previous->next);
        previous->next = traverse->next;
        traverse = traverse->next;
      }
      if (pthread_mutex_unlock(other_commands.mutex) < 0) return -1;
      return (1);
    } else {
      previous = traverse;
      traverse = traverse->next;
    }
  }
  if (pthread_mutex_unlock(other_commands.mutex) < 0) return -1;
  return (0);
}
```

```
/*$Id: lois.c,v 1.9.4.20 1999/05/24 02:43:32 agould Exp $*/
/***************************************************************************
                            Lowell Observatory
                     CCD Acquisitions Software Package


Program:         loas.e
Name:            $RCSfile: lois.c,v $
By:              Brian W. Taylor
Started:         06/05/98
Revision:        $Revision: 1.9.4.20 $
Last Revised:    $Date: 1999/05/24 02:43:32 $

Included in:     Stand-alone



Explanation:     This program is the initializer for the LOIS. It inits the
                 shared memory, starts the configuration and calls the
                 functions for the CCD Camera, Telescope, and Intsturment.



        Copyright 1998
, Lowell Observatory, All Rights Reserved

----------------------------------------------------------------------------
*/


#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199805
#endif

#ifdef __LINUX__
#ifndef _XOPEN_SOURCE
#define _XOPEN_SOURCE
#endif
#endif

#include <unistd.h>

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sched.h>
#include <errno.h>
#include <pthread.h>
#include <X11/Xlib.h>
#include <signal.h>



#ifdef __LINUX__
#include <sys/ipc.h>
#include <sys/msg.h>
#elif defined(__SOLARIS_5x__)
#include <mqueue.h>
#endif
#include <dirent.h>


#define TRUE 1
```

```
#define FALSE 0

/* CDL Library Include */
#include <cdl.h>
/* Tcl/Tk Includes */
#include <tk.h>
#include <tcl.h>

/* Lowell Acquisition Include */
#include <lois.h>

/* CCD Camera Include */
#include <LowellCCD.h>
#include <LowellTel.h>
#include <LowellInst.h>

ccd_vectors ccdvec;
ccd_struct ccd;

struct telescope_vectors telvec;
tel_struct telescope;

struct instrument_vectors instvec;
inst_struct instrument;

info_struct info;

display_struct l_display;
resq           results_queue;

Tcl_Interp      *lois_interp;

int dbgflag = 0;
long cur_issue_pidtag = 1;
long cur_eval_pidtag = 1;

pthread_t       cmd_servicer;
void * no_grx_cmds(void * dummy);
command_queue other_commands;

Tcl_Obj         *lois_obj;

void * interval_cmd(void * cmdptr);

int gui_send (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int LOIS_exit (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int LOIS_script (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[]);
int Command_Send(ClientData clientdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj *CONST objv[]);
int Slave_UnknownProc(ClientData clientdata, Tcl_Interp *interp,
                      int objc, Tcl_Obj *CONST objv[]);
int lois_enqueue (int key, lois_results in_value);

static void Lois_SetQueue(ClientData clientdata, int flags);
static void Lois_CheckQueue(ClientData clientdata, int flags);
static void Lois_Exit(ClientData clientdata);

pthread_mutex_t *dummy_mutex;
pthread_cond_t *dummy_cond;

sigset_t        block_set;

#ifdef __SOLARIS_5x__
```

```
struct mq_attr cmd_attr;
#endif
int main(int argc, char *argv[])

{

  int           ccd_fd, msgsize, toqueue, retval;
  void          *tcl_results;
  char          lois_cmd[1024], text_cmd[256], *pos_ptr;
  lois_results  enqueue_str;
  char          *puts_arg="command"; /* Needed as extra argument to puts */
  DIR           *cfghome;
  Display       *display;
  int           count;
  char          *lois_arg;  /* Linux - this variable MUST be last (or
                               some other variable that is a pointer
                               must be last) */

  /* Parsing the arguments, if any */

  if (argc > 1) {
     for (count=1; count < argc; count++) {
               if (strcmp (argv[count], "-echo") == 0)
                   dbgflag=1;
     }
  }

  results_queue.mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
  results_queue.go = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
  results_queue.head = (struct resq_elt *) malloc(sizeof(struct resq_elt));
  pthread_mutex_init(results_queue.mutex, NULL);
  pthread_cond_init(results_queue.go, NULL);
  results_queue.cur_count = 1;
  results_queue.head->key = 0;
  results_queue.head->next = NULL;
  results_queue.head->value.rtn_status = 0;
  strcpy(results_queue.head->value.rtn_message, "\0");

  dummy_mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
  dummy_cond = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
  pthread_mutex_init(dummy_mutex, NULL);
  pthread_cond_init(dummy_cond, NULL);


  /*  if (sigemptyset(&block_set) < 0) {
    printf("Error setting empty signal set! ");
    printf("Error number %d\n", errno);
    exit(-1);
  }
  if (sigaddset(&block_set, SIGSEGV) < 0) {
    printf("Error blocking SIGALRM signal! ");
    printf("Error number %d\n", errno);
    exit(-1);
  }
  pthread_sigmask(SIG_BLOCK, &block_set, NULL);
  if ((display=XOpenDisplay(NULL)) == NULL) {
    exit(-1);
  }
  */



/*
* Create a message queue to send Tcl/Tk commands back
* to the main thread/interp
*/
```

```
#ifdef __LINUX__
  if ((lois_qid=msgget(QUEUE_KEY, (IPC_CREAT | 0666))) < 0) {
    perror("Cannot open message queue");
    printf("Error number: %d\n", errno);
    exit(-1);
  }

#elif defined(__SOLARIS_5x__)

  cmd_attr.mq_maxmsg=128;
  cmd_attr.mq_msgsize = 1024;

  lois_queue = mq_open("/LOIS_queue", O_CREAT|O_RDWR|O_NONBLOCK, S_IRWXU, cmd_at
tr);
#endif

/*
* Get the LOISHOME and Configuration Dir Enviorment variables
*/

        info.loishome=getenv("LOISHOME");
        if (info.loishome == NULL) {
          printf("\nLOISHOME environment variable not set!! ");
          printf("Please set it with\n\nsetenv LOISHOME path\n\n");
          exit(-1);
        }

        info.cfghome=getenv("LOISDIR");
        if (info.cfghome == NULL) {
          printf("\nLOISDIR environment variable not set ..... ");
          printf("using HOME environment variable.\n\n ");
          info.cfghome=getenv("HOME");
          if (info.cfghome == NULL) {
            printf("\nLOISDIR or HOME environment variable not set!! ");
            printf("Please set it with\n\nsetenv LOISDIR path\n\n");
            exit(-1);
          }
        }

        retval=mkdir(strcat(info.cfghome,"/.lois"),
                     S_IRWXU|S_IRWXG|S_IROTH|S_IXOTH);
        if (retval != 0 ) {
          switch (errno) {
          case EEXIST:
            cfghome=opendir(info.cfghome);
            if (cfghome == NULL ) {
                if (errno == EACCES) {
                  printf("\n ERROR! Permission denied for access");
                  printf(".lois directory \n\n");
                  exit(-1);
                }
              }
            closedir(cfghome);
            break;
          case EACCES:

            printf("\nERROR! You do not have permission to ");
            printf("create the configuration directory\n");
            printf("in the %s directory. \n\n Please change the environment ");
            printf("variable to a permitted directory\n", info.cfghome);
            exit(-1);
            break;
          default:
            perror("Problem with Configuration Dir. Creation:");
            printf("Errno %d\n", errno);
            exit(-1);
          }
        }
/* Shared memory stuff... */

#ifdef __LINUX__
/*
 *
 * Initialize the shared memory mapping for the Image Buffer
 * This is now eliminated because the size of the image buffer is determined in
the CCD module
 */
        /*
          if (( shm_fd=shmget(40, IMAGE_BUFFER_SIZE, (IPC_CREAT | 0666))) < 0) {
          perror("Cannot Open CCD Shared Memory Buffer");
          printf("Error number %d\n", errno);
          exit(-1);
          }
        */

/*
*
* Initialize the shared memory mapping for the CCD Vectors and Structures.
*
*/
        if (( ccdvec.mem_fd=shmget(CCD_KEY, sizeof(ccdvec)+sizeof(ccd),
                                   (IPC_CREAT | 0666))) < 0) {
          perror("Cannot Open CCD Shared Memory Buffer");
          printf("Error no %d\n",errno);
          exit(-1);
        }
/*
*
* Initialize the shared memory mapping for the Telescope Vectors and
* Structures.
*
*/
        if ((telvec.mem_fd=shmget(TEL_KEY, sizeof(telescope)+sizeof(telvec),
                                  (IPC_CREAT | 0666))) < 0) {
          perror("Cannot Open Telescope Shared Memory Buffer");
          printf("Error no %d\n",errno);
          exit(-1);
        }

/*
*
* Initialize the shared memory mapping for the Instrument Vectors and
* Structures.
*
*/
        if ((instvec.mem_fd=shmget(INST_KEY,sizeof(instrument)+
                                   sizeof(instvec), (IPC_CREAT | 0666))) < 0) {
          perror("Cannot Open Instrument Shared Memory Buffer");
          printf("Error no %d\n",errno);
          exit(-1);
        }
/*
*
* Initialize the shared memory mapping for the Information
* Structures.
*
*/
        if (( info.mem_fd=shmget(INFO_KEY, sizeof(info), (IPC_CREAT | 0666))) <
0) {
          perror("Cannot Open the information Shared Memory Buffer");
          printf("Error no %d\n",errno);
          exit(-1);
```

```
        }

/* map shared memory segments */

        /*      ccdvec.buffer=shmat(shm_fd, 0, 0); */
        ccdvec.memory=shmat(ccdvec.mem_fd, 0, 0);
        telvec.memory=shmat(telvec.mem_fd, 0, 0);
        instvec.memory=shmat(instvec.mem_fd, 0, 0);
        info.memory=shmat(info.mem_fd, 0, 0);

        /*      if (ccdvec.buffer == (void *) -1) {
          perror("Memory Map failed for Image Buffer");
          exit(-1);
          }*/
        if (ccdvec.memory == (void *) -1) {
          perror("Memory Map failed for CCD BUffer");
          exit(-1);
        }
        if (telvec.memory == (void *) -1  ) {
          perror("Memory Map failed for Telescope Buffer");
          exit(-1);
        }

        if (instvec.memory == (void *) -1) {
          perror("Memory Map failed for Instrument Buffer");
          exit(-1);
        }
        if (info.memory == (void *) -1) {
          perror("Memory Map failed for Information Buffer");
          exit(-1);
        }
#elif defined(__SOLARIS_5x__)
/*
*
* Initialize the shared memory mapping for the Image Buffer.
*
*/
        if (( shm_fd=shm_open(IMAGE_BUFFER_NAME, O_CREAT | O_TRUNC | O_RDWR,
                              S_IRWXU)) < 0 ) {
                        perror("Cannot Open Shared Memory Buffer");
                        printf("Error no %d\n",errno);
                        exit(-1);
        }
/*
*
* Initialize the shared memory mapping for the CCD Vectors and Structures.
*
*/
        if (( ccdvec.mem_fd=shm_open("/ccd", O_CREAT | O_TRUNC | O_RDWR,
                              S_IRWXU)) < 0 ) {
                        perror("Cannot Open CCD Shared Memory Buffer");
                        printf("Error no %d\n",errno);
                        exit(-1);
        }

/*
*
* Initialize the shared memory mapping for the Telescope Vectors and
* Structures.
*
*/
        if (( telvec.mem_fd=shm_open("/telescope", O_CREAT |
                                O_TRUNC | O_RDWR, S_IRWXU)) < 0 ) {
                    perror("Cannot Open Telescope Shared Memory Buffer");
                    printf("Error no %d\n",errno);
                    exit(-1);
        }

/*
*
* Initialize the shared memory mapping for the Instrument Vectors and
* Structures.
*
*/
        if (( instvec.mem_fd=shm_open("/instrument", O_CREAT |
                                O_TRUNC | O_RDWR, S_IRWXU)) < 0 ) {
                        perror("Cannot Open Instrument Shared Memory Buffer");
                        printf("Error no %d\n",errno);
                        exit(-1);
        }

/*
*
* Initialize the shared memory mapping for the Information
* Structures.
*
*/
        if (( info.mem_fd=shm_open("/information", O_CREAT |
                                O_TRUNC | O_RDWR, S_IRWXU)) < 0 ) {
                perror("Cannot Open the information Shared Memory Buffer");
                printf("Error no %d\n",errno);
                exit(-1);
        }


/*
*
* Shared memory is opened now lets set the size
*
*/


        if (ftruncate(ccdvec.mem_fd, sizeof(ccdvec)+sizeof(ccd)) < 0) {
                        perror("Cannot Set CCD Shared Memory Size");
                        exit(-1);
        }

        if (ftruncate(telvec.mem_fd, sizeof(telvec)+sizeof(telescope)) < 0)
        {
                        perror("Cannot Set Telescope Shared Memory Size");
                        exit(-1);
        }
        if (ftruncate(instvec.mem_fd, sizeof(instvec)+sizeof(instrument))
                        < 0)
        {
                        perror("Cannot Set Instrument Shared Memory Size");
                        exit(-1);
        }
        if (ftruncate(info.mem_fd, sizeof(info))
                        < 0)
        {
                        perror("Cannot Set Information Shared Memory Size");
                        exit(-1);
        }


/*
*
* Map the share memory buffer into the buffer vector
*
*/
```

```
        ccdvec.buffer=mmap(NULL, IMAGE_BUFFER_SIZE, PROT_READ | PROT_WRITE,
                                MAP_SHARED, shm_fd, 0);
        ccdvec.memory=mmap(NULL, sizeof(ccdvec)+sizeof(ccd),
                                PROT_READ | PROT_WRITE,MAP_SHARED,
                                ccdvec.mem_fd, 0);
        telvec.memory=mmap(NULL, sizeof(telvec)+sizeof(telescope),
                                PROT_READ | PROT_WRITE,MAP_SHARED,
                                telvec.mem_fd, 0);
        instvec.memory=mmap(NULL, sizeof(instvec)+sizeof(instrument),
                                PROT_READ | PROT_WRITE,MAP_SHARED,
                                instvec.mem_fd, 0);
        info.memory=mmap(NULL, sizeof(info),
                                PROT_READ | PROT_WRITE,MAP_SHARED,
                                info.mem_fd, 0);
        if (ccdvec.buffer == NULL) {
                        perror("Memory Map failed for CCD Buffer");
                        exit(-1);
        }
        if (telvec.memory == NULL) {
                        perror("Memory Map failed for Telescope Buffer");
                        exit(-1);
        }

        if (instvec.memory == NULL) {
                        perror("Memory Map failed for Instrument Buffer");
                        exit(-1);
        }
        if (info.memory == NULL) {
                        perror("Memory Map failed for Information Buffer");
                        exit(-1);
        }
#endif
        /* Setting the default module names as none to be changed
           by each module when loaded
        */

        strcpy(info.loisvers, "LOIS V1.1\0");
        strcpy(info.ccdmod, "none\0");
        strcpy(info.telmod, "none\0");
        strcpy(info.instmod, "none\0");
        shm_write(ccdvec.memory, (void *)&ccdvec, sizeof(ccdvec));
        shm_write(telvec.memory, (void *)&telvec, sizeof(telvec));
        shm_write(instvec.memory, (void *)&instvec, sizeof(instvec));
        shm_write(info.memory, (void *)&info, sizeof(info));

/*
*
* Now Lock the buffer into memory to prevent the memory being swap to disk
*
*/
        if ( mlock(ccdvec.buffer, IMAGE_BUFFER_SIZE) < 0 ) {
                printf("\nCould Not Lock Image Buffer Into Memory");
                printf(" Continuing anyway....\n");
        }

        lois_interp = Tcl_CreateInterp();
        info.lois_main=lois_interp;
        shm_write(info.memory, (void *)&info, sizeof(info));
        lois_obj=Tcl_NewStringObj(lois_cmd, sizeof(lois_cmd));
        if (Tcl_Init(info.lois_main) == TCL_ERROR) {
                /* return((void *)TCL_ERROR); */
        }
        if (Tk_Init(info.lois_main) == TCL_ERROR) {
          printf("Error initializing Tk in main interp!\n");
          printf("Error: %s\n", Tcl_GetStringResult(info.lois_main));
        }
        printf("Adding LOIS Exit Event Handler....\n");
        Tcl_CreateExitHandler(Lois_Exit, NULL);

        Tcl_StaticPackage(info.lois_main, "Tk", Tk_Init, Tk_SafeInit);
        sprintf(lois_cmd,"global LOIS_HOME");
        Tcl_SetStringObj(lois_obj, lois_cmd, -1);
        if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
               printf("Error Setting LOIS_HOME in Master Interp\n");
        }

        sprintf(lois_cmd,"set LOIS_HOME {%s}", info.loishome);
        Tcl_SetStringObj(lois_obj, lois_cmd, -1);
        if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
               printf("Error Setting LOIS_HOME in Master Interp\n");
        }
        printf("Done.\n");
        printf("Loading the LOIS Console Module......");


        sprintf(lois_cmd, "load %s/lib/console.so", info.loishome);
        Tcl_SetStringObj(lois_obj, lois_cmd, -1);
        if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
#ifdef __LINUX__
          shmctl(shm_fd, IPC_RMID, NULL);
          shmctl(ccdvec.mem_fd, IPC_RMID, NULL);
          shmctl(telvec.mem_fd, IPC_RMID, NULL);
          shmctl(instvec.mem_fd, IPC_RMID, NULL);
          shmctl(info.mem_fd, IPC_RMID, NULL);
#endif
          printf("LOIS Console Module Loading Error!!");
          exit(-1);
        }
        printf("Done!\n");
        printf("Starting the LOIS Console.......");
        Tcl_SetStringObj(lois_obj, "lois_console", -1);
        if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
                printf("Error Starting LOIS Console!!");
                exit(-1);
        }

        Tcl_CreateObjCommand(info.lois_main, "gsend", gui_send,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
        Tcl_CreateObjCommand(info.lois_main, "load_script", LOIS_script,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);
        Tcl_CreateObjCommand(info.lois_main, "lois_send", Command_Send,
                   (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL);


        /*
#ifdef __LINUX__
lois_log4("LOIS: LINUX version");
#elif defined(__SOLARIS_5x__)
lois_log4("LOIS: Sun SOLARIS 5.X version");
#endif
*/
        lois_arg=Tcl_GetVar(info.lois_main, "eval(slave)", 0);

        if ((info.lois_grx=Tcl_GetSlave(info.lois_main, lois_arg)) == NULL) {
          printf("Error getting slave graphics interpreter!\n");
          exit(-1);
        }

        printf("Adding LOIS Queue Event Handler....\n");
        Tcl_CreateEventSource(Lois_SetQueue, Lois_CheckQueue, NULL);
        printf("Done!\n");
```

```
        info.lois_cmd=Tcl_GetSlave(info.lois_main, "shell command");

        if (info.lois_cmd == NULL) {
          printf("Error getting slave interpreter in secondary thread!\n");
          exit(-1);
        }

        shm_write(info.memory, (void *)&info, sizeof(info));


  /* Load libraries into second slave so it can do logging and analysis */

  if (Tcl_CreateAlias(info.lois_cmd, "puts", info.lois_main,
                      "PutsAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing puts command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_grx, "PutPrompt", info.lois_main,
                      "PutPrompt", 0, NULL) == TCL_ERROR) {
    printf("Error aliasing putprompt command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "log", info.lois_main,
                      "LogAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing log command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "rlog", info.lois_main,
                      "RLogAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing rlog command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "glog", info.lois_main,
                      "GLogAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing glog command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "ylog", info.lois_main,
                      "YLogAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing ylog command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "blog", info.lois_main,
                      "BLogAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing blog command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "olog", info.lois_main,
                      "OLogAlias", 1, &puts_arg) == TCL_ERROR) {
    printf("Error aliasing olog command in command interpreter!\n");
  }

  if (Tcl_CreateAlias(info.lois_cmd, "exp_proc", info.lois_grx,
                      "exp_proc", 0, NULL) == TCL_ERROR) {
    printf("Error aliasing exp_proc command in command interpreter!\n");
  }
  if (Tcl_CreateAlias(info.lois_cmd, "clean_exp", info.lois_grx,
                      "clean_exp", 0, NULL) == TCL_ERROR) {
    printf("Error aliasing clean_exp command in command interpreter!\n");
  }

  /* Want to alias set command in command interpreter so it defines commands in
the
     graphics interpreter.  To do that, need to get rid of the "normal" set comm
and
     by changing its definition */

  Tcl_SetStringObj(lois_obj, "rename set setold", -1);
  if (Tcl_EvalObj(info.lois_cmd, lois_obj) == TCL_ERROR) {
    printf("Error changing set command!\n");
    exit(-1);
  }
  if (Tcl_CreateAlias(info.lois_cmd, "set", info.lois_grx, "SetAlias",
                      0, NULL) == TCL_ERROR) {
    printf("Error aliasing set command in command interpreter!\n");
    exit(-1);
  }

  if (Tcl_CreateAlias(info.lois_cmd, "global", info.lois_grx, "global",
                      0, NULL) == TCL_ERROR) {
    printf("Error aliasing global command in command interpreter!\n");
    exit(-1);
  }

  pthread_create(&cmd_servicer, NULL, no_grx_cmds, (void *) info.loishome);

  Tk_MainLoop();


}

/****************************************************************************
*********************/
/***** This procedure is run in a separate thread.  It initializes the secondary
 (command)    *******/
/***** interpreter, and then goes into a loop to pull commands from a prioritize
d queue.      *******/
/***** When a command is removed from the queue, it is executed first on the sec
ondary        *******/
/***** interpreter, and if that fails, it is executed on the graphics interprete
r.  Results   *******/
/***** (and errors) are echoed to the screen.
              *******/
/****************************************************************************
*********************/

void * no_grx_cmds(void * dummy) {

  Tcl_Obj     *interp_rcv;
  Tcl_Interp *cmd_interp;
  char        interp_cmd[1024];
  cmd_struct sec_cmd;
  struct _command *traverse, *previous, *highest, *high_prev;
  int         cmd_index, pidtag, result;
  lois_results main_results;

  interp_rcv=Tcl_NewStringObj(interp_cmd, sizeof(interp_cmd));

  /* Initialize thread variables for secondary message queue */

  other_commands.mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
  other_commands.go = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
  other_commands.blocked = 0;
  other_commands.waiting = 0;
  pthread_mutex_init(other_commands.mutex, NULL);
  pthread_cond_init(other_commands.go, NULL);
  other_commands.head = other_commands.tail = NULL;

  /* Load libraries into second slave so it can do logging and analysis */

  /*
    if (Tcl_CreateAlias(info.lois_cmd, "puts", info.lois_main,
    "puts", 0, NULL) == TCL_ERROR) {
    printf("Error aliasing puts command in command interpreter!\n");
    }
    if (Tcl_CreateAlias(info.lois_grx, "PutPrompt", info.lois_main,
    "PutPrompt", 0, NULL) == TCL_ERROR) {
```

```
    printf("Error aliasing puts command in command interpreter!\n");
    }
  */

  if (Tcl_CreateAlias(info.lois_cmd, "exit", info.lois_main,
                      "exit", 0, NULL) == TCL_ERROR) {
    printf("Error aliasing exit command in command interpreter!\n");
  }

  /* Create a handler in the command interpreter that is called when the command
 interpreter
     tries to interpret a command that it does not know */

  if (Tcl_CreateObjCommand(info.lois_cmd, "unknown_handler", Slave_UnknownProc,
                           (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL) == NUL
L) {
    printf("Error creating unknown handler in command interpreter!\n");
  }

  /*  if (Tcl_CreateObjCommand(info.lois_cmd, "q_block", Delay_CommandQueue,
                           (ClientData)(NULL), (Tcl_CmdDeleteProc *)NULL) == NUL
L) {
    printf("Error creating block routine in command interpreter!\n");
  }
  */
  sprintf(interp_cmd, "load %s/lib/fits.so", (char *) dummy);
  Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
  Tcl_EvalObj(info.lois_cmd, interp_rcv);
  sprintf(interp_cmd, "load %s/lib/logger.so", (char *) dummy);
  Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
  if (Tcl_EvalObj(info.lois_cmd, interp_rcv) == TCL_ERROR) {
    printf("Cannot load logging library into secondary interpreter!\n");
    printf("Error: %s\n", Tcl_GetStringResult(info.lois_cmd));
  }
  /*
  sprintf(interp_cmd, "load %s/lib/analysis.so", (char *) dummy);
  Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
  Tcl_EvalObj(info.lois_cmd, interp_rcv); */

  /* The unknown procedure is called when an interpreter tries to interpret an u
ndefined command
     For the command interpreter, we want this procedure to call the unknown_han
dler, declared above */

  Tcl_SetStringObj(interp_rcv, "proc unknown {args} {\nreturn [unknown_handler $
args]\n}", -1);
  if (Tcl_EvalObj(info.lois_cmd, interp_rcv) == TCL_ERROR) {
    printf("Error setting unknown routine for commands interpreter!\n");
    printf("Error: %s\n", Tcl_GetStringResult(info.lois_cmd));
  }
  printf("Command interpreter initialized!\n");

  for ( ; ; ) {
    pthread_testcancel();
    pthread_mutex_lock(other_commands.mutex);
    while ((other_commands.head == NULL) || (other_commands.blocked)) {
      other_commands.waiting = 1;
      pthread_cond_wait(other_commands.go, other_commands.mutex);
      other_commands.waiting = 0;
    }

    /* Find command with highest priority */

    for (traverse = other_commands.head, highest = other_commands.head, previous
 = NULL,
           high_prev = NULL; traverse != NULL; traverse = traverse->next) {
```

```
      if (traverse->value.priority > highest->value.priority) {
        highest = traverse;
        high_prev = previous;
      }
      previous = traverse;
    }

    if (high_prev != NULL) {
      high_prev->next = highest->next;
    } else {
      other_commands.head = highest->next;
    }

    sec_cmd = highest->value;
    cur_eval_pidtag = highest->pidtag;
    ckfree((void *) highest);
    pthread_mutex_unlock(other_commands.mutex);

    /* Now we have the command, figure out if we should execute it on the second
 "no-graphics"
       interpreter.  If an error occurs determining the interpreter, just contin
ue on to
       the next command, destroying the error-causing command (without executing
 it...is this
       the best thing to do?) */
    if (dbgflag) {
      printf("Sec. Command: %s, pidtag %d\n", sec_cmd.command, cur_eval_pidtag);
    }

    /* Display the value of the current process tag */

    sprintf(interp_cmd, "pt_write {%s: Pidtag=%d}", sec_cmd.command, cur_eval_pi
dtag);
    Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
    Tcl_EvalObj(info.lois_cmd, interp_rcv);


    /*    don't need this code with error handler addition

          sprintf(interp_cmd, "lsearch [info commands] [lindex [split {%s}] 0]",
 sec_cmd.command);

          Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
          if (Tcl_EvalObj(cmd_interp, interp_rcv) == TCL_ERROR) {
          printf("Error determining which interpreter for command '%s'!\n", sec_
cmd.command);
          printf("Error: %s\n", Tcl_GetStringResult(cmd_interp));
          continue;
          }

          If the the command we're running is in the list of registered commands
 for the
          no-graphics interpreter, run the command there, otherwise run it in th
e main
          interpreter

          if (Tcl_GetIntFromObj(cmd_interp, Tcl_GetObjResult(cmd_interp), &cmd_i
ndex) < 0) {
          printf("Error converting result to integer in secondary interpreter!\n
");
          printf("Error: %s\n", Tcl_GetStringResult(cmd_interp));
          continue;
          }

          if (cmd_index < 0) {
```

```
          The command wasn't registered in the secondary interpreter, so send it
 to the
          main interpreter

          sec_cmd.priority = -1 * ABSOLUTE(sec_cmd.priority);
          pidtag = lois_send(&sec_cmd);
          lois_receive(pidtag, &main_results);
          if (main_results.rtn_status != 0) {
          sprintf(interp_cmd,"$eval(text) insert insert {%s} error", main_result
s.rtn_message);
          Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
          Tcl_EvalObj(lois_interp, interp_rcv);
          } else {
          sprintf(interp_cmd,"$eval(text) insert insert {%s} result", main_resul
ts.rtn_message);
          Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
          Tcl_EvalObj(lois_interp, interp_rcv);
          }

          } else {
    */

    sprintf(interp_cmd, "eval {%s}", sec_cmd.command);
    Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
    result = Tcl_EvalObj(info.lois_cmd, interp_rcv);
    if (result == TCL_ERROR) {

      /* See if there was a dangling variable reference.  If so, pass the comman
d on to the
         graphics interpreter */

      if (strstr(Tcl_GetStringResult(info.lois_cmd), "no such variable") != NULL
) {

        if (Tcl_EvalObj(info.lois_grx, interp_rcv) == TCL_ERROR) {
          if (strstr(Tcl_GetStringResult(info.lois_grx), "no such variable") !=
NULL) {

            /* If the variable was undefined in the graphics interpreter, pass t
he command to the main
               interpreter, print the appropriate message according to what happ
ened in the main
               interpreter */

            if (Tcl_EvalObj(info.lois_main, interp_rcv) == TCL_ERROR) {

              /* results of evaluating in main interpreter */

              printf("Error executing command '%s' in all interpreters!\n", sec_
cmd.command);
              sprintf(interp_cmd, "$eval(text) insert insert {Main: %s} error",
                      Tcl_GetStringResult(info.lois_main));
              Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
              Tcl_EvalObj(info.lois_main, interp_rcv);
            } else {
              sprintf(interp_cmd,"$eval(text) insert insert {Pidtag %d;Result: %
s} result",
                      cur_eval_pidtag, Tcl_GetStringResult(info.lois_main));
              Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
              Tcl_EvalObj(info.lois_main, interp_rcv);
            }
            /* end if evaluating in main interpreter */

          } else {

            /* There is some other error (besides an unknown variable) in the gr
aphics interpreter,
               output the error to the console */

            printf("Error executing command '%s' in graphics interpreter!\n", se
c_cmd.command);
            sprintf(interp_cmd,"$eval(text) insert insert {Grx: %s} error",
                    Tcl_GetStringResult(info.lois_grx));
            Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
            Tcl_EvalObj(info.lois_main, interp_rcv);
          }
        } else {
          /* else: the command executed successfully in the graphics interpreter
 */

          sprintf(interp_cmd,"$eval(text) insert insert {Pidtag %d;Result: %s} r
esult",
                  cur_eval_pidtag, Tcl_GetStringResult(info.lois_grx));
          Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
          Tcl_EvalObj(info.lois_main, interp_rcv);
        }
        /* end evalution in graphics interpreter */

      } else {

        /* There is some other error (besides an unknown variable) in the comman
d interpreter, so just
           output the error to the console */

        printf("Error executing command '%s' in secondary interpreter!\n", sec_c
md.command);
        sprintf(interp_cmd,"$eval(text) insert insert {Cmd: %s} error", Tcl_GetS
tringResult(info.lois_cmd));
        Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
        Tcl_EvalObj(info.lois_main, interp_rcv);
      }
    } else {

      /* Everything is OK. Command executed successfully in command interpreter.
  Print
         command and command result to console window */

      sprintf(interp_cmd,"$eval(text) insert insert {Pidtag %d;Result: %s} resul
t",
              cur_eval_pidtag, Tcl_GetStringResult(info.lois_cmd));
      if (strstr(interp_cmd, "{}") == NULL ) {
        Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
        Tcl_EvalObj(info.lois_main, interp_rcv);
      } else {
        if (dbgflag) printf("Caught Empty Results string\n");
      }
    }
    /* end evaluation in command interpreter */

    if (Tcl_Eval(info.lois_main, "PutPrompt") == TCL_ERROR) {
      printf("Error writing prompt to console window!\n");
      printf("Error: %s\n", Tcl_GetStringResult(info.lois_main));
    }
  }
}

/*******************************************************************************
********************/
/***************** Tcl object command procedure to send functions to secondary q
ueue ***************/
/*******************************************************************************
********************/
```

```
int Command_Send(ClientData clientdata, Tcl_Interp *interp,
                 int objc, Tcl_Obj *CONST objv[])
{

  Tcl_Obj       *send_obj;
  char          err_str[80], *tmp_cmd;
  cmd_struct    sched_cmd;
  int           retval;

  sched_cmd.priority = 1;
  sched_cmd.t_interval = 0;
  sched_cmd.t_times = 0;
  sched_cmd.t_execute = 0;

  send_obj = Tcl_NewStringObj(err_str, sizeof(err_str));

  if (objc > 3) {
    Tcl_SetStringObj(send_obj, "Too many arguments in call to lois_send", -1);
    Tcl_SetObjResult(interp, send_obj);
    return(TCL_ERROR);
  }

  if (objc == 3) {
    if (Tcl_GetIntFromObj(interp, objv[2], &sched_cmd.priority) == TCL_ERROR) {
      Tcl_SetStringObj(send_obj, "Unable to get priority in lois_send", -1);
      Tcl_SetObjResult(interp, send_obj);
      return(TCL_ERROR);
    }
  }
  sprintf(sched_cmd.command, "%s\0", Tcl_GetStringFromObj(objv[1], NULL));
  if (strcmp(sched_cmd.command, "") == 0) {
    Tcl_SetStringObj(send_obj, "PutPrompt", -1);
    Tcl_EvalObj(interp, send_obj);
    return(TCL_OK);
  }

  if (cmd_send(sched_cmd) < 0) {
    Tcl_SetStringObj(send_obj, "Error sending command to secondary command queue
!", -1);
    Tcl_SetObjResult(interp, send_obj);
    return(TCL_ERROR);
  }

  return(TCL_OK);
}

/*******************************************************************************
********************/
/********************* Helper procedure for secondary command queue **********
********************/
/*******************************************************************************
********************/

int cmd_send(cmd_struct send_cmd) {

  struct _command *tmp_ptr;

  if (pthread_mutex_lock(other_commands.mutex) < 0) return -1;

  tmp_ptr = (struct _command *) ckalloc(sizeof(struct _command));
  tmp_ptr->value = send_cmd;
  tmp_ptr->pidtag = cur_issue_pidtag;
  cur_issue_pidtag++;
  if (cur_issue_pidtag == 1000000) {
    cur_issue_pidtag = 1;
  }
  tmp_ptr->next = NULL;
  if (other_commands.head == NULL) {
    other_commands.head = other_commands.tail = tmp_ptr;
    if (pthread_cond_signal(other_commands.go) != 0) return -1;
  } else {
    other_commands.tail->next = tmp_ptr;
    other_commands.tail = tmp_ptr;
  }

  if (pthread_mutex_unlock(other_commands.mutex) < 0) return -1;
  return (0);
}

/*******************************************************************************
********************/
/*************************** Error handling procedure for command interpreter **
********************/
/*******************************************************************************
********************/

int Slave_UnknownProc(ClientData clientdata, Tcl_Interp *interp,
                      int objc, Tcl_Obj *CONST objv[])
{

  /*  lois_results main_results;
  int count;
  char cmd_str[256];
  Tcl_Obj *cmd_list; */

  int result;
  Tcl_Interp *grx_interp;

  /*  This next block sends the command to the main message queue to see if the
command can be
      interpreted by the main interpreter

      sec_cmd.priority = -1 * ABSOLUTE(sec_cmd.priority);
      pidtag = lois_send(&sec_cmd);
      lois_receive(pidtag, &main_results);
      if (main_results.rtn_status != 0) {
      sprintf(interp_cmd,"$eval(text) insert insert {%s} error", main_results.rt
n_message);
      Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
      Tcl_EvalObj(lois_interp, interp_rcv);
      } else {
      sprintf(interp_cmd,"$eval(text) insert insert {%s} result", main_results.r
tn_message);
      Tcl_SetStringObj(interp_rcv, interp_cmd, -1);
      Tcl_EvalObj(lois_interp, interp_rcv);
      }

  */

  /*  This block simply executes the unknown command in the primary interpreter
(it bypasses the
      main message queue).  The result returned by the primary interpreter is co
pied to the
      result field of the secondary interpreter */

  /*  cmd_list = Tcl_NewListObj(objc-1, objv+sizeof(Tcl_Obj)); */

  grx_interp = Tcl_GetMaster(interp);

  result = Tcl_EvalObj(grx_interp, objv[1]);
  Tcl_SetResult(interp, Tcl_GetStringResult(grx_interp), NULL);
```

```
  /*  printf("Bytes: %s\n",Tcl_GetStringFromObj(objv[1], NULL));
      printf("Result: %s\n", interp->result); */
  return(result);

}

/*******************************************************************************
*/
/***************** Command to block the secondary command queue ****************
*/
/*******************************************************************************
*/

int Delay_CommandQueue(ClientData clientdata, Tcl_Interp *interp,
                       int objc, Tcl_Obj *CONST objv[])
{

  lois_log1("Main command queue blocked!");
  pthread_cond_wait(other_commands.go, other_commands.mutex);
  pthread_mutex_unlock(other_commands.mutex);
  Tcl_SetResult(interp, "Command queue unblocked", NULL);
  return(TCL_OK);
}

int gui_send (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
        Tcl_Obj         *cmd_ptr;
        char            lois_cmd[240], *lcmd_ptr;

        cmd_ptr=Tcl_NewStringObj(lois_cmd, sizeof(lois_cmd));
        cmd_ptr=Tcl_ConcatObj(objc, objv);
        lcmd_ptr=Tcl_GetStringFromObj(cmd_ptr, NULL);
        sprintf(lois_cmd, "interp eval $eval(slave) %s", lcmd_ptr+6);
        Tcl_SetStringObj(cmd_ptr, lois_cmd, -1);;
        Tcl_EvalObj(info.lois_main, cmd_ptr);



}

static void Lois_Exit (ClientData clientdata)
{

        static char     exit_cmd[10];
        int             retval=0;

        munlock(ccdvec.buffer, IMAGE_BUFFER_SIZE); /* Unlock Image Buffer */
        pthread_cancel(cmd_servicer);
#ifdef __SOLARIS_5x__
        munmap(ccdvec.buffer, IMAGE_BUFFER_SIZE);
        munmap(ccdvec.memory, sizeof(ccdvec)+sizeof(ccd));
        munmap(telvec.memory, sizeof(telvec)+sizeof(telescope));
        munmap(instvec.memory, sizeof(instvec)+sizeof(instrument));
        munmap(info.memory, sizeof(info));
        mq_close(lois_queue);
        retval=shm_unlink(IMAGE_BUFFER_NAME);
        if (retval < 0) {
          perror("Error: Problem unlinking Image Buffer");
        }
        shm_unlink("/ccd");
        shm_unlink("/telescope");
        shm_unlink("/instrument");
        shm_unlink("/information");
        mq_unlink("/LOIS_queue");
```

```
#elif defined(__LINUX__)
        shmdt(ccdvec.buffer);
        shmdt(ccdvec.memory);
        shmdt(telvec.memory);
        shmdt(instvec.memory);
        shmdt(info.memory);
        if ((msgctl(lois_qid, IPC_RMID, NULL)) < 0) {
          perror("Error closing message queue");
          printf("Error number: %d\n", errno);
        }
        shmctl(shm_fd, IPC_RMID, NULL);
        shmctl(ccdvec.mem_fd, IPC_RMID, NULL);
        shmctl(telvec.mem_fd, IPC_RMID, NULL);
        shmctl(instvec.mem_fd, IPC_RMID, NULL);
        shmctl(info.mem_fd, IPC_RMID, NULL);
#endif
        sprintf(exit_cmd,"exit");
        Tcl_SetStringObj(lois_obj, exit_cmd, -1);
        if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
              printf("Error Executing Command from Queue\n");
              printf("Command: %s\n",exit_cmd);
        }

        exit(0);

}

int LOIS_script (ClientData clientdata, Tcl_Interp *interp,
                int objc, Tcl_Obj *CONST objv[])
{
  static char *cmd, terpcmd[1024];
  pid_t         cmdpid;

  if (objc > 1) {
    cmd=Tcl_GetStringFromObj(objv[1],NULL);
  } else {
    sprintf(interp->result,"Usage: load_script cmd");
    return(TCL_OK);
  }
  if ((cmdpid=fork()) < 0) {
    perror("LOIS: Error cannot fork process\n");
  } else {
    if (cmdpid == 0 ) {
      sprintf(terpcmd,"set nslave [interp create $eval(slave) {%s}]", cmd);
      Tcl_SetStringObj(lois_obj, terpcmd, -1);
      if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
        printf("Error Executing Scripting Command\n");
        printf("Command: %s\n",cmd);
      }
      sprintf(terpcmd,"interp alias $nslave single {} single", cmd);
      Tcl_SetStringObj(lois_obj, terpcmd, -1);
      if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
        printf("Error Executing Scripting Command\n");
        printf("Command: %s\n",cmd);
      }
      sprintf(terpcmd,"interp alias $nslave waitcam {} waitcam", cmd);
      Tcl_SetStringObj(lois_obj, terpcmd, -1);
      if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
        printf("Error Executing Scripting Command\n");
        printf("Command: %s\n",cmd);
      }

      sprintf(terpcmd,"interp eval $eval(slave) {%s}", cmd);
      Tcl_SetStringObj(lois_obj, terpcmd, -1);
      if (Tcl_EvalObj(info.lois_main, lois_obj) == TCL_ERROR) {
        printf("Error Executing Scripting Command\n");
```

```
        printf("Command: %s\n",cmd);
      }

      exit(0);
    }
  }

  return(TCL_OK);
}

static void Lois_SetQueue(ClientData clientdata, int flags)
{

  Tcl_Time      blk_time;

/* This function sets the maximun wait time to wait for an event
   to occur. So we set this to the min. time resolution that
   we are prediction to be working on. Note this time that is
   set here is not absolute and can not be use as a RT clock
*/

  blk_time.sec=0;
  blk_time.usec=1000;


  Tcl_SetMaxBlockTime(&blk_time);


}
          /* message queue is below.  Note that we can only receive an array
             of characters in the System V message queue.  So the flag denoting
             whether or not to queue the results is encoded in the string.  It
             might be better (and part of a future implementation) to use a pipe
             instead of a message queue, since only the libmisc.so and this file
             use the message queue.  It would also allow for the definition of
             an arbitrary structure, so the string parsing below would be unnece
ssary.
             If we used pipes, we'd have to integrate a feature to take priority
 of
             messages into account, this is given in the message queue */

static void Lois_CheckQueue(ClientData clientdata, int flags)
{

  int           ccd_fd, msgsize, toqueue, retval, serviceval;
  char          lois_cmd[1024], text_cmd[256], *pos_ptr;
  lois_results  enqueue_str;
#ifdef __LINUX__
  static struct msgbuf cmd_qmsg;
#elif defined(__SOLARIS_5x__)
  char cmd_qmsg[1024];
#endif

  serviceval=Tcl_GetServiceMode();

#ifdef __LINUX__

  msgsize=msgrcv(lois_qid, &cmd_qmsg, MSG_Q_SIZE, -35, IPC_NOWAIT);
  if (msgsize > 0) {
    if (dbgflag) printf("Queue CMD: %s\n", cmd_qmsg.mtext);
  }
  if ( msgsize != -1 ) {
    if ((pos_ptr=strstr(cmd_qmsg.mtext, "+=+")) == NULL) {
      sprintf(text_cmd, "%s", cmd_qmsg.mtext);
      toqueue = -1;
    } else {
        bzero(text_cmd, sizeof(text_cmd));
        strncpy(text_cmd, cmd_qmsg.mtext, pos_ptr-cmd_qmsg.mtext);
        toqueue = atoi(text_cmd);
        bzero(text_cmd, sizeof(text_cmd));
        strncpy(text_cmd, pos_ptr+3, cmd_qmsg.mtext+strlen(cmd_qmsg.mtext)-pos_ptr
-2);
      }

#elif defined(__SOLARIS_5x__)
    msgsize=mq_receive(lois_queue, cmd_qmsg, 1024, NULL);
    if (msgsize > 0) {
      if (dbgflag) printf("Queue CMD:%s\n",cmd_qmsg);
    }
    if ( msgsize != -1 ) {
      if ((pos_ptr=strstr(cmd_qmsg, "+=+")) == NULL) {
        sprintf(text_cmd, "%s", cmd_qmsg);
        toqueue = -1;
      } else {
        bzero(text_cmd, sizeof(text_cmd));
        strncpy(text_cmd, cmd_qmsg, pos_ptr-cmd_qmsg);
        toqueue = atoi(text_cmd);
        bzero(text_cmd, sizeof(text_cmd));
        strncpy(text_cmd, pos_ptr+3, cmd_qmsg+strlen(cmd_qmsg)-pos_ptr-2);
      }
#endif
      sprintf(lois_cmd,"interp eval $eval(slave) {%s}", text_cmd);
      Tcl_SetStringObj(lois_obj, text_cmd, -1);
      if (Tcl_EvalObj(info.lois_grx, lois_obj) == TCL_ERROR) {
        printf("Error Executing Command from Queue\n");
        printf("Error Command: %s\n",lois_cmd);
        if (toqueue >= 0) {
          enqueue_str.rtn_status = -1;
          sprintf(enqueue_str.rtn_message, "%s\0", Tcl_GetStringResult(info.lois_m
ain));
          if (lois_enqueue(toqueue, enqueue_str) != 0) {
            printf("Error enqueueing results %s", enqueue_str.rtn_message);
          }
        }
      } else {
        if (toqueue >= 0) {
          enqueue_str.rtn_status = 0;
          sprintf(enqueue_str.rtn_message, "%s\0", Tcl_GetStringResult(info.lois_m
ain));
          if (lois_enqueue(toqueue, enqueue_str) != 0) {
            printf("Error enqueueing results %s", enqueue_str.rtn_message);
          }
        }
      }
    } else {
#ifdef __LINUX__
      if ( errno != ENOMSG) {
        perror("Error Reading Command from Queue");
      }
#elif defined(__SOLARIS_5x__)
      if ( errno != EAGAIN) {
        perror("Error Reading Command from Queue");
      }
#endif

    }
}

/*******************************************************************
Change Log:

$Log: lois.c,v $
```

```
Revision 1.9.4.20  1999/05/24 02:43:32  agould
Made secondary interpreter print pidtag of current command to console
in green.  Also associated console results with pidtags instead of
command strings.

Revision 1.9.4.19  1999/05/21 18:25:10  agould
added commands for command removal and abort

Revision 1.9.4.18  1999/05/07 22:01:39  agould
Aliased set command so that conflicts between variables in dual interpreter
system is eliminated.

Revision 1.9.4.17  1999/04/29 22:26:48  taylor
Fixing Solaris bug with "\n" and also added aliases from cmd and grx
interps to cmd and grx interps.

Revision 1.9.4.16  1999/04/29 16:56:15  agould
Removed shm image buffer allocation for LINUX (now in CCD module)

Revision 1.9.4.15  1999/04/28 20:01:50  taylor
Commiting changes for Solaris compatibility.

Revision 1.9.4.14  1999/04/27 17:20:14  taylor
Commiting changes for storage and review.

Revision 1.9.4.13  1999/04/26 21:27:13  taylor
Merging changes to be compatible with Solaris.

Revision 1.9.4.12  1999/04/23 21:46:57  taylor
Updating changes to make the CCD module a little more generic.

Revision 1.9.4.11  1999/04/22 20:54:36  agould
Merging changes from branch rel_1_1b for dual interpreter system

Revision 1.9.4.8.2.3  1999/04/22 16:50:05  agould
Added ccd_vectors declaration to lois.c on branch rel_1_1b.

Revision 1.9.4.8.2.2  1999/04/14 21:29:48  agould
Added error handling function to command interpreter so that commands within
procedures are passed to graphics interpreter when necessary.

Revision 1.9.4.8.2.1  1999/04/08 01:18:59  agould
Dual interpreter system

Revision 1.9.4.7  1999/04/06 17:45:17  taylor
Removed hardwire dbgflag...

Revision 1.9.4.6  1999/04/06 17:41:52  taylor
Added max block time while waiting on an event from the LOIS Queue. This
has resolved the polling load that has been a problem from the beginning.

Revision 1.9.4.5  1999/04/06 03:47:26  taylor
Added an event source handler to read the commands off the queue. This
solves the problem of the polling loop to get the commands off the queue.

Revision 1.9.4.4  1999/03/24 21:55:45  taylor
Changing configuration storage from a single file to a directory in either
the user HOME directory or in the LOISDIR directory. Also moved the change
log to the end of the file and removed the image buffer creation size to
be install and configured by each CCD module.

Revision 1.9.4.3  1999/03/16 19:39:38  agould
merging changes to hans

Revision 1.9.4.2  1999/03/04 22:41:51  agould
Linux compatibility for message queue.
```

```
Revision 1.9  1998/11/12 23:29:42  taylor
Refixed the default module values.

Revision 1.8  1998/11/12 22:32:16  taylor
Set the values of the modules to "none" for the default.

Revision 1.7  1998/11/12 21:40:39  taylor
Added LOIS version label into info structure for fits
header version tracking.

Revision 1.6  1998/11/10 21:56:26  taylor
Added getting the LOISHOME environment variable. To allow execution
from any directory.

Revision 1.5  1998/10/27 00:28:56  taylor
Added the Log functions for writting to logs and to
the console windows. Also added the lois_send function
to start setting up and using execution intervals and
times.

Revision 1.4  1998/08/26 21:36:29  taylor
First version to be used at the telescope with both local
and guest observers. This testing resulted in a freeze of
the current version and a new one started hopfully to
correct some of the design problems that we are facing.

Revision 1.3  1998/07/03 23:36:24  taylor
LOIS console load added. Some POSIX thread in place

Revision 1.2  1998/06/09 23:17:04  taylor
Updated the Comment Header to maintain standard RCS comment


*****************************************************************************/
```