The DEIMOS Rotation Control System Software

William T.S. Deich, Robert I. Kibrick, S. M. Faber, De Clarke, and Vernon Wallace

UCO/Lick Observatory, University of California, Santa Cruz, CA 95064

ABSTRACT

The Deep Imaging Multi-Object Spectrograph (DEIMOS) was delivered to the Keck II telescope during February 2002, and has been commissioned in the several months since then. Most of the instrument is in a barrel that rests on a cradle at the Nasmyth focus, and rotates to track field rotation. This paper describes the architecture of the rotator control software, including the communications protocols, time synchronization with the telescope control software, methods adopted for meeting the real-time control requirements, safety issues for a multi-ton rotating mass, and unusual position encoder challenges.

Keywords: instrument rotator, real-time control, KTL, DEIMOS, Galil, Linux, Renishaw, encoders

1. INTRODUCTION

The DEIMOS instrument is described by Faber et. al. elsewhere in these Proceedings.[1] From the perspective of the rotation software, the key requirement is that DEIMOS' 7-ton rotating mass tracks the field rotation to 17 asec rms. The instrument's optical train is mounted in a barrel that is supported at the front by a 2.4 m drive disk which serves as a front bearing; the disk rests on a friction drive roller and three idler rollers. The rear of the barrel is supported by a spherical bearing. A Renishaw encoder tape is wrapped around a disk at the rear and is read by two Renishaw encoders spaced 131 degrees apart.

This paper describes the rotation computer and software architecture, giving particular attention to major issues affecting the design, and evaluating the strengths and weaknesses of our solutions. Since instrument rotators are hardly a novelty, we focus on distinctive features of the software. We start with a summary of the rotator control hardware, briefly review the software control interface, outline the software architecture, and close with a description of features that may be of interest to software developers.

Unlike most instruments at Keck Observatory, DEIMOS can be rotated independently of the telescope drive and control system (DCS). This provides several major benefits to both developers and observers: instrument tests at different rotation were carried out before shipping; testing at Keck can be done during the daytime when procedures normally prevent rotation, and can be done regardless of whether DEIMOS is the currently-attached instrument; and observers can carry out calibrations at any rotation angles in the afternoon hours.

The software design was driven by the functional requirements, instrument design, and hardware selected for rotation control. The functional specification for rotation were few: 810 deg of wrap, 3 deg/sec slew speed, and $17''$ rms tracking error to within 0.5 degrees of zenith. The achieved range of wrap is 740°, the slew speed is 2.3 deg/s, and outside the 0.5 deg zenith hole, the tracking is within $17''$ except for a small band of hour angle close to the zenith at which the tracking rate excites a 2 Hz mechanical resonance. Figure 1 shows the servo tracking performance at declinations of 16° and 19° (zenith distance 3.8° and 0.8°).

2. THE COMPUTER, MOTOR CONTROLLER, AND ENCODERS

A Galil motor controller was a natural choice because Galil controllers have for many years been a *de facto* standard for Lick instruments. The limitations of the Galil controllers with serial RS-232 interfaces — namely very low bandwidth for both commands and status information — prompted us to select a new version which was packaged on a PCI card.

Further author information: (Send correspondence to W.T.S.D.)
W.T.S.D.: E-mail: will@ucolick.org, Telephone: 1 831 459-3913

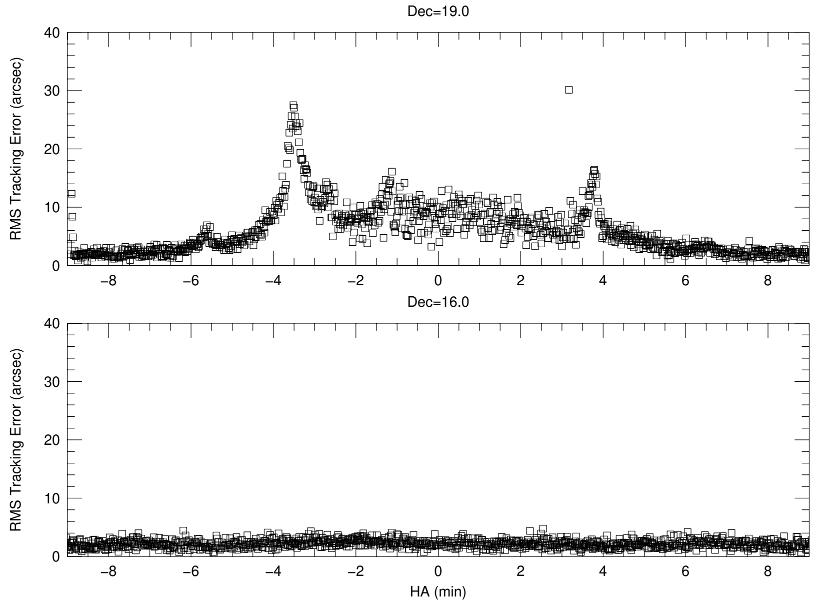

Figure 1. RMS servo tracking error at two declinations. At $\delta = 16^\circ$, the tracking is always very good (lower). At $\delta = 19^\circ$, close to the zenith, the tracking is within the $17''$ threshold except for HA $= \pm 3.5'$, where a mechanical resonance is excited in the instrument barrel (upper).

2.1. The Rotation Computer

The rotation control computer is mounted on the DEIMOS instrument cradle, which in turn sits on the telescope's Nasmyth platform. In the rare event of a sudden loss of power, the platform – and the computer – could be shaken hard. Such shaking could potentially damage a disk, and therefore the rotation control computer is diskless. Keck Observatory standards call for Sun computers and the Solaris operating system, but Sun no longer supports diskless computers. Instead, we chose to use a Dell PC in a "pizza box" form factor, running RedHat Linux. Using a PC offers the advantages of commodity hardware, in particular low cost and high availability for replacement parts, and Dell was Keck's preferred PC supplier.

We chose to use Linux as the runtime environment because of its excellent development environment, portability to run on a wide variety of hardware, and the convenience and efficiency that result when one uses the same development and runtime environments. We used RedHat's then-current release (RH 6.1) rather than a real-time variant of Linux such as RT-Linux because (a) tests showed that our applications were satisfactorily responsive under ordinary Linux, and (b) we wanted to keep the hardware and software as "vanilla" and mainstream as possible. It was practical to use a non-realtime Linux because the computer has no tasks other than rotation, so there are few other processes to compete for system resources.

The diskless computer requires some deviations from the usual Linux boot sequence, including a modified version 2.2.16 Linux kernel, but all the libraries and applications (/lib, /usr, /bin, /sbin) mounted and used after booting are ordinary files requiring no special modifications. We used the Linux Terminal Server Project[2] (LTSP) because it provides a well-documented setup that was easy to customize for our needs. The diskless computer starts by running special boot code from the EEPROM of its network card. The boot code uses BOOTP requests to learn its own IP address and get the address of a TFTP server. It then downloads its kernel using TFTP and begins executing the kernel, which has been customized to use BOOTP to get its networking information and the NFS address of its root partition. The kernel NFS-mounts its root partition, and then invokes the init process, and thereafter the boot sequence follows the normal Linux methods. The initialization script called for by the */etc/inittab* `sysinit` entry is */etc/rc.local*, which creates a ramdisk for */tmp*, */var*, and */dev*, populates those directories, and then NFS-mounts the other Linux filesystems. A few system daemons are started – *syslogd*, *xntpd*, and *portmapper* – and the last action is to start the rotation control software. The entire boot sequence after power-on takes about 45 seconds.

2.2. The Galil Motor Controller Device Driver

The rotation motor is controlled by a Galil DMC-1840 4-axis motion controller PCI card. The DMC-1840 has 4 motor axes, 8 uncommitted digital inputs, 8 digital outputs, and 8 analog inputs. The rotation controller needs to handle some two dozen I/O bits, and therefore a 64-channel digital I/O daughter board is also used.

The DMC-18x0 cards have two PCI interfaces: a primary channel for sending commands and receiving responses, and a secondary channel from which records of status information can be read. The secondary status record is updated every 2 ms, by default. It contains encoder positions, velocity, torque, a wide variety of status flags, and so on.

The fast I/O provided by the primary and secondary channels represent a very significant enhancement over serial I/O-based Galil controllers that have been used in a number of previous instruments built at Lick. Those controllers were limited to a serial connection running at tty speeds (sometimes at 9600 or 19200 baud), providing only the "primary channel" interface and an auxiliary serial line through which status data could be sent, again at a slow rate. The slow communications and difficulty of getting status values in a timely manner meant that a great deal of the control logic was written in Galil code and run onboard the motor controller. Unfortunately, the motor controllers also have very limited memory and a primitive programmming language. The program memory on the DMC-18x0 series is just 1000 lines of 80 characters each. All variables are global; there is no scoping. Control flow statements are limited to a test-and-branch statement that can also be used for simple goto's; an `IF/ELSE/ENDIF` statement; and "subroutine" calls that are really just a jump to a global label with an implied jump back at the end of the labeled section.

Fast communications and readily available status permit us to put almost all the control logic into an external C program and use the Galil to execute commands that do not require flow control: the Galil's strengths lie in moving a motor at a precise speed (jogging) or to a precise position, and almost all the commands that we send to the Galil are simple jog or position requests, or configuration parameters. One additional useful feature of the Galil is its ability to execute up to eight threads of commands in parallel. For example, the rotation controller is responsible for monitoring a set of analog input values multiplexed onto one analog input. The multiplexing is done by using three output bits as a 3-bit address to select one of the eight inputs from a hardware multiplexer. A Galil thread is used to cycle through the eight possible addresses, pausing briefly after each one is selected. This runs forever, entirely on the Galil, and thus avoids the need to interleave a stream of multiplexer-update commands with the rotation control commands from the control program.

At the time the rotation controller project began, no Linux device driver was available from Galil, and therefore we wrote our own.[3] Galil has since released a Linux interface library, but we continue to use the Lick-written driver because it is more conformant with Unix programming conventions. That is, *read(2)* and *write(2)* work as expected, and the Lick driver provides a *select(2)* interface for both the primary and secondary channels. Although the DMC18x0 can generate an interrupt for various conditions related to the primary channel, it has no such capability for the secondary channel. The device driver has to implement the *select()*'able interface by rapidly polling the secondary channel for updates and marking the file descriptor as readable when the channel has new data. Programs using the device driver will appear to consume all idle CPU time due to the polling, but in fact each poll takes only a few microseconds and other processes are nearly unaffected by the polling.

2.3. Encoders

The rotating barrel is driven by a Galil 500-1000C 1/2 HP motor, acting through a 3160:1 reduction. The motor encoder has 4000 counts/rev, or ≈ 9 counts/arcsec of barrel rotation.

The instrument position is read using two Renishaw RGH22Y encoders, reading from a Renishaw encoder tape wrapped around a disk at the rear of the instrument. The readheads are offset from each other by 131 degrees around the tape, and each has a resolution of ≈ 2.59 counts/arcsec resolution. (Mechanical packaging constraints prevented the readheads from being located the preferred amount of 180° apart.) The tape ends are carefully butted together so that the gap between them is negligible, and each Renishaw readhead counts accurately across any residual gap. They have continued to do so after the instrument was installed in the Keck dome at 0 C. Two readheads were installed due to concerns that the gap would not remain negligible over time, and thus the readhead counts would have to be ignored in the region of the gap. A true rotary encoder would not have been subject to these problems, but was not used because (a) it is more costly than the linear tape, and (b) replacement of a damaged rotatory encoder would be very difficult due to the large instrument cable bundle running through the center of the rotating barrel.

There have been both headaches and advantages associated with the use of two readheads. The biggest problem occurs if we want to disregard a Renishaw's counting when it is close to the tape gap, and then adjust its count afterwards so that it matches the span of counts from the other Renishaw. The readhead mounts have a very slight position jitter with respect to each other, and, during the time of any particular crossing of the tape gap, one of the encoders might be one count different from the other encoder. Over many crossings of the tape gap, these phase differences will come and go but stay close to zero. However, if software forces the span of counts from the two readheads to match precisely during some window while the tape gap is being crossed by a readhead, a random walk will be introduced by occasionally adding or subtracting a count. This random walk can grow to be significant (perhaps $5''$) during the course of a night, depending on the observing program. A great deal of time was wasted trying to cure this problem, but no algorithmic solution was found. Instead, we are relying on the fact that in practice, both readheads work well near the tape gap, and no attempts are made to adjust the readhead counters.

A second disadvantage of using two Renishaw readheads is that the motor controller's dual-loop capability cannot be used. There is substantial backlash and slippage in the rotator drive system, and as a result the motor's position encoder does not give a precise absolute position of the rotating barrel. The Renishaw encoders do give the absolute precision needed; if there were just a single Renishaw encoder in use, its signal would be used with the Galil's dual-loop capability to provide accurate tracking. However, the Galil has no mechanism to handle two auxiliary encoders, and so backlash compensation is handled by a routine that monitors the Renishaw position and sends explicit adjustment commands to the motor.

The important advantage of using two readheads turns out to be a cancellation of calibration errors. As described in Ref. 1, the relationship between each Renishaw encoder and the true rotator position angle has a $\cos\theta$ term with peak-to-peak amplitude $\approx 500''$, as if the encoder tape were mounted on a slightly off-center disk. The average value of the $\cos\theta$ term nearly cancels, however, because the two encoders are mounted 131 degrees apart, and the residual $\pm 15''$ amplitude is easier to calibrate. More importantly, hysteresis in the *separation* of the two readheads was also found when the rotator reversed direction, but fortunately this nearly cancels out in the average of the two encoders.

The encoder tape's "index pulse" is normally used for homing the rotation control system. When combined with the Galil motor controller's ability to use input signals to trigger a latch of Renishaw encoder values, it can also be used as an indicator of pointing quality. Normally, an index pulse is fed to a motor controller's designated index-pulse input for a particular axis. In this case, the index pulse signal is not only fed to the inputs for both Renishaw axes, but also to the auxiliary inputs. The DMC-1840 can latch the Renishaw encoder value within 20 μsec of a downward-going signal on these inputs. The control software uses this to latch the Renishaw value at every crossing of the index pulse, and compares each latched value with the expected value. Although the index pulse occurs at just one rotation angle, it has proven to be a helpful diagnostic of the stability of the overall rotation pointing.

3. THE DEIROT KEYWORD INTERFACE

In ordinary operations, communications with the rotation controller occur through a KTL keyword service.[4] *

KTL keywords provide a uniform method for client programs such as user interfaces to communicate with servers such as the rotation controller or DCS. The power of these keyword interfaces comes from their simplicity, from the uniformity of the interface syntax across instruments, and from the fact that keywords can be easily read or modified via shareable libraries that use a common API which can be accessed from Unix shell commands, TCL, or C.

KTL keywords are grouped into *services*; the DEIMOS rotation controller provides the `deirot` service. For example, the rotator angle can be read from a shell script with the command `show -s deirot ROTATVAL`, where `ROTATVAL` is the KTL keyword name. Similarly, the instrument can be sent to a rotator angle of (say) 225.1 degrees by using the shell command `modify -s deirot ROTATVAL=225.1`. Similar commands can be included in a TCL script or a C program.

KTL services can be implemented using a variety of transport protocols. KTL is primarily a client-side tool: the transport method is hidden from client libraries by the standard KTL keyword API, but there is no similar KTL server-side API. Thus each service must directly handle messages using the underlying transport method: the `deirot` service is implemented using `MUSIC` messages,[5] and the `DCS` service uses EPICS Channel Access.[6]

In addition to providing the `deirot` keyword service, the rotator is also a keyword client: it monitors and tracks the position demands that are computed and broadcast by the DCS system, using `DCS`-service keywords. The rotator uses other `DCS` keywords to send the achieved rotator position and tracking error back to the `DCS` service. The interface between DCS and rotators is described in Refs. 7 and 8.

The `deirot` service contains a set of keywords that control and monitor the rotation. In addition, it contains supplementary keywords that provide environmental information: rotator power-supply voltages, various temperature sensors, and coolant flow levels. Most of these values are not used directly by the rotator, but are simply read by the rotator controller on behalf of other processes; otherwise, an additional general-purpose I/O controller would have been required. The `deirot` keywords are summarized online; see Ref..[9]

4. SOFTWARE ARCHITECTURE

The software design was driven by several considerations:

- safety: the rotation controller must respond promptly to all inputs, such as a STOP button being pressed on a hand paddle;
- tracking performance must meet requirements;
- client program feedback: answers to KTL queries and broadcasts of state changes must be prompt;
- general pointing performance: in order to characterize and calibrate the system, the non-tracking positioning must be highly accurate and repeatable.

A threaded implementation would be a natural fit to the requirements because there are several parallel activities running on different timescales:

- the motor controller's state updates appear every 2 ms on the secondary channel;
- the tracking control loop updates every ± 20 ms;
- most rotator updates are broadcast on a 1-sec clock;
- interpolated DCS demands arrive every ± 10 ms;

*For debugging purposes, there are several non-keyword methods of control: Unix signals, dynamic reconfigurations, and a direct terminal-emulation command interface to the Galil controller.

- user requests arrive asynchronously with respect to any other activities.

One thread for each of the periodic actions would make for a reasonably straightforward implementation. Nonetheless, we did not implement a threaded model, but instead used an event-driven design, because we did not regard threading as mature on Linux at the time project was started. In hindsight this was a mistake, because Linux threading was rock-solid by the time DEIMOS was delivered, whereas the event-driven model has made it tricky to implement long-running commands such as tracking and position moves.

The main event loop begins by waiting for new input in the *gio_select()* routine. Upon return, it handles the ready input events, which are mainly of two kinds:

1. a new DCS position and speed demand has arrived;
2. a new keyword message has arrived.

Note that there is no mention of the secondary channel, which provides data such as the motor encoder position, speed, and torque values; input bits status; and so on. That is because the secondary channel is automatically handled in the background by the *gio_select()* routine. Our purpose was to simplify the high-level design so that it did not need to handle the 500 Hz update rate of the secondary channel.

4.1. Secondary Channel Handler

The secondary channel handler executes at ≈ 500 Hz, the rate at which the Galil motor controller updates the secondary record. In addition to reading the secondary channel, we take advantage of its high rate and use its handler to also deal with a variety of other rapid-update values. The primarly list of tasks executed by this handler includes the following:

1. Update encoder information. Check that the encoder signal quality is adequate, that encoders all agree, that torque level is not too high, etc.
2. Check all status bits, including the indicators for an open hatch, band brake, panic button, etc.
3. Check the hand paddle and execute any commands required by changes in its switches and buttons.
4. Broadcast keyword values that have changed. In general, the broadcast rate of rapidly changing keywords (such as the position angle during a slew) is throttled to 1 Hz, and these are broadcast together on one-second clock ticks. However, if more than one second elapses between the changes of a keyword value, its new value is broadcast immediately, regardless of the exact time. This provides quick responsiveness to keyword clients.
5. Read the analog and digital inputs.
6. If the rotator is in DCS-tracking mode and a 20-ms interval has been reached, schedule the tracking-update routine to be run by the top-level dispatcher loop.

The secondary channel handler may run recursively. For example, if the secondary handler detects a condition that requires tracking to stop — such as switching to manual mode, or the band brake's being applied – the stop command is immediately sent to the Galil from within the secondary handler. The handler waits for a response from the Galil's primary channel, and while it waits it uses one of a family of *waitfor* routines that (a) recursively call the secondary channel handler, (b) check the primary channel, and (c) check and process any new keyword messages and other MUSIC messages, which can again cause a recursive call to the secondary channel handler.

Executing slow commands from the secondary handler is counter-intuitive to the notion of a 500 Hz update rate. The software model is that almost all actions occur instantaneously, except for very long-running commands such as DCS tracking. Those long-running commands are handled from the top-level event loop. The "instantaneous" commands can be handled at any level, including the secondary channel handler. The fact that many commands are not-quite-instantaneous is swept under the rug by recursively calling the secondary channel handler whenever a *waitfor* routine is used, because that *waitfor* routine will in turn respond promptly to events that occur during its wait. However, recursion is never very deep, because the system has been carefully structured so that input sequences which take more than a few seconds and could therefore cause deep recursion are handled by routines that are run from the top-level event loop, do a small amount of work quickly, and then re-schedule themselves to be called again in the top-level event loop.

The *waitfor* routines return a success code when the motor control sends the expected response. Since these routines also monitor new DCS demands, keywords, and secondary channel data that can arrive during the wait time, the *waitfor* routines must monitor the overall state that can be changed by a recursive call to the secondary channel handler. They will return non-success values to indicate conditions such as: a timeout elapsed without response from the motor controller; a new keyword command (such as changing to the `Halt` state or moving to a different position) interrupted the current command; a switch on the hand paddle disabled automatic tracking); and so on. When the return value from a *waitfor* routine is not the success code, the caller must interrupt its activities and make an early return.

4.2. Tracking

Telescope tracking starts with DCS demands. DCS broadcasts timestamped rotator position angle demands. The demands are extrapolated using a second-order polynomial to the current time by *watch_keywords* (see the **Co-processes** section), numerically differentiated to give the current demand velocity, and the position and velocity updates are passed at a 10 Hz rate to the main tracking program.

In the main tracking program, the secondary channel handler re-schedules the tracking routine *run_track* to run at each 20ms clock tick. At the top level, the main event loop receives the 10-Hz extrapolated DCS demands and checks for a pending command to execute — which will be the *run_track* routine as long as DCS tracking is enabled. Each time it is called, *run_track* compares the current extrapolated DCS position demand with the current rotator position. The position error feeds a velocity PID loop. That is, the velocity request sent to the Galil motor controller is the velocity computed from the interpolated DCS position demands plus a correction term computed from the instantaneous position error.

As mentioned in the Introduction, the tracking procedure works well at most points on the sky except for a narrow band of hour angles near the zenith where the tracking speed excites a 2 Hz mechanical resonance. The region of poor tracking only occurs when the declination is within about two degrees of zenith, and typically spans an arcminute of hour angle, a few minutes of arc away from the zenith crossing (see Fig. 1). At that point, the tracking errors approximate a sine wave with up to $40''$ peak-to-peak errors. The 2 Hz frequency is too fast to be handled by the PID loop, because the mechanical response time of the DEIMOS barrel to a change in motor speed is about 100 ms. Thus, the PID loop cannot track changes that occur on a timescale within a few hundred ms, whereas the 2 Hz oscillation rises from 0 to peak amplitude in 125 ms. On the other hand, if we could predict the amplitude and phase of the resonance, we might be able to inject a cancelling sine wave by inserting a speed adjustment 180° out of phase, plus ≈ 100 ms phase lead, to allow for the response time of the barrel. To that end, we have made limited experiments in which we monitored the tracking errors for a sine wave-like pattern and used those measured values to inject a cancelling speed adjustment. So far these experiments have not paid off, but the tests are not yet finished.

4.3. Co-processes

The rotation controller is implemented using a collection of five processes (see Fig. 2). The key process is *dispatcher*, which implements the event loop described in the preceding section. DCS demands and related DCS keywords are monitored by the *watch_keywords* process,[10] which uses a *watch_rotator* thread to smoothly extrapolate DCS rotator demands to the current instant of time, compute the corresponding demanded velocity, and send the extrapolated demands to the *dispatcher*. The *rotocache* process is a service to provide a few keywords required by the *watch_rotator* thread. (The cache service and *watch_rotator* thread are part of the package referred to in Ref. 10.)

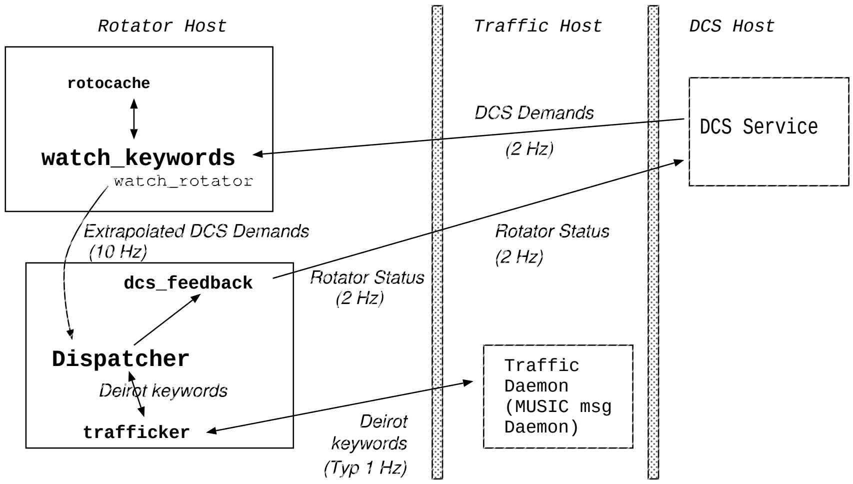

Figure 2. Processes involved in rotation control are shown in boldface. Network connections and the kind of data flowing on them are shown with arrows labelled with slanted text. The processes that run on the DEIMOS rotaion computer are shown in the *Rotator Host* column, and are grouped into the *watch_keywords* and *dispatcher* collection of processes.

Finally, the *dcs_feedback* and *trafficker* processes respectively mediate communications between *dispatcher* and the DCS service, and between *dispatcher* and the MUSIC messsage passing daemon.

The DCS service[8] broadcasts timestamped rotator angle demands at 2Hz. The routines provided by *watch_rotator* extrapolate a smooth higher-rate set of demands for use by the rotator — that is, they provide a DCS servo loop. *Watch_rotator* is implemented as a set of callback routines designed to be run by the *watch_keywords* process. *Watch_keywords* is a general-purpose keyword monitor that starts with a list of keywords to monitor; callback functions that should be invoked when the keywords are received; and shareable libraries that provide the callback routines. Extrapolated, timestamped position demands are passed to *dispatcher* through a Unix-domain datagram socket.

Unix-domain datagram sockets are used for all communications between *dispatcher* and its co-processes *watch_keywords*, *trafficker*, and *dcs_feedback*. Under Linux, sending a non-blocking message on such a socket guarantees that the message will either be immediately sent in its entirety or fail, and using recv() on such a socket will either read an entire message or none at all. This makes it simple to ensure message integrity, because there will be no partially-sent messages, and no need to implement complicated recovery methods. The *trafficker* and *dcs_feedback* processes are children of *dispatcher*, and the datagram sockets are simply created by a *socketpair(2)* call before the child is forked.

Note that *watch_keywords* is completely independent from *dispatcher*. In order for these unrelated peers to share a datagram-oriented socketpair, the process that creates the socketpair must send the file descriptors to its peer. *Dispatcher* listens on a named datagram socket for new connections. *Watch_keywords* creates the socketpair, then connects to the *dispatcher* datagram socket, and uses *sendmsg(2)*'s ability to send file "credentials" or "ancillary data," thereby passing the socketpair descriptors to *dispatcher*. This slightly painful mechanism pays off with the ability to use the simplicity and efficiency of the unnamed socketpair for all future communications between these processes.

Two co-processes are used simply to handle keyword communications: *trafficker* handles the MUSIC messages that implement the KTL protocol, and *dcs_feedback* sends rotator status updates to DCS using the KTL client routines. These operations are segregated into processes separate from the dispatcher because the responsiveness requirement means that *dispatcher* must never block when doing I/O. A problem arises because the actual I/O operations are abstracted by the MUSIC API used for the `deirot` service, and by the KTL API used for the `DCS` service. These API's do not provide a method of guaranteeing that one can write a message without blocking. Segregating the I/O into separate processes ensures that the dispatcher process will never block. It also means that if there are any unexpected problems such as memory leaks within the KTL or MUSIC libraries, they will be somewhat isolated from the dispatcher process and its core responsibilty to maintain safe control of the rotating barrel.

The *trafficker* process makes a normal tcp connection to the *traffic* daemon used to handle MUSIC messages. Each MUSIC message it receives from *traffic* is promptly written to the datagram socket; each message from the dispatcher received on the datagram socket is re-written to *traffic*. If *trafficker* is blocked for a long time while writing a message to *traffic*, its datagram socket can become full; as long as it continues to be full, the dispatcher's attempts to send messages will fail, and the messages will be discarded. If *trafficker*'s connection to the *traffic* daemon should fail, *trafficker* will try to reconnect to *traffic* every 30 seconds.

The *dcs_feedback* process is only used to send information to the `DCS` service; it does not receive any DCS demands or other keywords. It is used to send

- timestamped rotation position and tracking error messages;
- rotator status (e.g. "tracking", "slewing", "initializing");
- command acknowledgement for state-change demands (e.g. changing from a tracking state to the halted state).

We measured the time required to send messages back and forth between processes using the Unix datagram method. Our tests on a 400 Mhz Pentium PII computer required an average of 11 μsec to send a message, perform a context switch to the receiving process, and collect the message. Even if the dispatcher were to send or receive 200 msgs/sec (far more than actually occurs in practice), it would incur an overhead of 2 ms per second, as compared to the time required by a single-threaded dispatcher without a co-process for I/O. The overhead is even less on the actual computer used (a Pentium PIII running at 750 MHz or 900 MHz).

Had we used a threaded model for the `deirot` service, the I/O would have been handled by separate threads instead of fully independent processes, but the penalty for not using threads is small. The preferred Linux method for thread implementation is to use a "one-to-one" model, in which a separate process is used for each thread; the main difference between thread creation and normal child process forking is that a thread shares its parent memory. Using separate processes for each thread simplifies implementations, especially in handling blocking I/O properly. The cost of this model is that a context switch is needed to change threads, so it is important that context switches be efficient on operating systems implementing the one-to-one model. Measurements[11] show that approximately half the time is spent in context switches (which would be the same for either the co-process or threaded models), and the remainder in socket I/O, which represents the additional overhead for the co-process model. Thus the co-process model costs less than 1 ms/sec overhead relative to the threaded alternative.

5. INSTALLATION AND DEBUGGING

5.1. Installation Issues

Installing the rotation system at Keck Observatory went smoothly. The key difference between operations at Keck and and development and testing at the Lick Observatory shops is that the real DCS system was used instead of a simulator.

Several minor problems occurred. In some cases the actual DCS interface acted slightly differently than the DCS simulator, and the rotation software had to be changed slightly to accomodate the real system. Under some circumstances used during the first-light tests, the DCS system didn't generate correct rotation demands for DEIMOS. A workaround has been developed by Keck staff. Finally, the rotation software would occasionally lose its EPICS connection to DCS. This was unexpected because EPICS is expected to be robust and automatically reconnect as needed. The rotation software has been modified so that it will explicitly reconnect to DCS if the connection fails, and investigations are continuing to identify the cause of the problem.

Finally, new "handoff" procedures had to be developed for the beginning and end of night. The usual DCS rotator initialization sequence doesn't have provisions for an instrument that can be in a state that ignores DCS. However, DEIMOS rotation will typically be under observer control at the beginning of the night, and the observer must push a button on the DEIMOS user interface to hand control over to DCS and allow the DCS rotator initialization to proceed.

5.2. A Terminal Emulator for a Galil

Communications with some previous generations of Galil controllers used a serial line. At UCO/Lick Observatory, we would typically connect these to a network terminal server, and then either our control software or a direct *telnet* connection could be used to communicate with the Galil. This made it simple to use Galil's command language to debug our code and tune the motor controller, but that method is not available with a PCI card. Galil's own Windows-based software provides a terminal emulator among a large variety of other software, but that is not useful for our card in a Linux host.

The *polyterm*[12] terminal emulator has proven quite helpful in developing the rotator software. This emulator is useful for communicating with a variety of hardware over serial lines, sockets, or a special MUSIC message interface. The last interface was developed specifically for DEIMOS' motor controller and the polyterm. It is quite simple so that it is easy to implement for any controller. A specific MUSIC message number is designated as a "motor controller message," and the contents of the message are literal text. Any text typed at a *polyterm* is wrapped in one of these MUSIC messages and sent to *dispatcher*. *Dispatcher* receives the messages, sends the text to the motor controller, and rebroadcasts it for any other *polyterm*'s to receive. Any data from the motor controller's primary channel is also broadcast by *dispatcher*. Finally, *dispatcher* also broadcasts any commands it sends to the motor controller. As a result, any *polyterm* has a complete record of the primary-channel communications among *dispatcher*, motor controller, and other *polyterm*'s. Commands from *polyterm* are interleaved with commands from *dispatcher*. This is both a powerful debugging tool and a dangerously simple access method, so *dispatcher* does not enable the MUSIC message interface unless it is enabled from the command line.

5.3. Galil Tuning Over a 4000 km Serial Line

Galil supplies a useful package of software for tuning the control loops in their motor controllers. However, it is designed to tune a motor controller that is either attached to the local network, to a serial port, or a card connected to the local host, and the software only runs under Windows. This is a great inconvenience when the Galil card is installed in a PC in Hawaii running Linux, and the engineer is in Santa Cruz! Fortunately, one can construct a virtual serial line that works successfully across the 4000 km gap.

The method is as follows. On the Galil controller host (in Hawaii), the *pathrelay*[13] TCL program is used to to relay the Galil's primary channel to a local network port. From a Santa Cruz Unix-based PC, *ssh* is used to set up secure port forwarding, listening for connections on a local port, and forwarding them to the *pathrelay*'d port on the Galil host. Another instance of *pathrelay* (this time on the local Unix PC) is used to connect the *ssh* local port to the local serial port, and a Windows laptop with the Galil tuning software has its serial port connected to the Unix PC's serial port. Finally, the Galil software can be run, configured to look for a Galil controller connected to its serial port.

Compared to tuning a locally-installed PCI card, there are visible delays due to the network propagation delays across the ocean, but the overall performance is similar to what would actually occur with a Galil controller with a serial interface connected at 19200 baud.

6. CONCLUSIONS

The DEIMOS rotator was implemented using motor controller packaging that was new to Lick Observatory, which has led to some unusual implementation software design. The event-loop implementation has led to more complicated code than would be required by a threaded model, but the rotator tracks reliably and the software has been robust.

7. ACKNOWLEDGMENTS

DEIMOS was started in 1992 with a Facilities and Instrumentation grant from the National Science Foundation (ARI92-14621) in the amount of $1.79 million. The California Association for Research in Astronomy has so far contributed approximately $7.2 million directly to the project, not including the additional costs of CARA liason and alterations that were needed for the Keck 2 telescope. UCO/Lick Observatory has contributed over $1 million so far in labor.

The authors wish to recognize and acknowledge the very significant cultural role and reverence that the summit of Mauna Kea has always had within the indigenous Hawaiian community. We are most fortunate to have the opportunity to conduct observations from this mountain.

REFERENCES

1. S. M. Faber et.al., *DEIMOS spectrograph for the Keck 2 telescope: Integration and testing*, in these proceedings.
2. J. McQuillan, *LTSP – Linux Terminal Server Project – v3.0*, `http://www.ltsp.org/documentation/ltsp-3.0.0/ltsp-3.0.html`, 2002.
3. W. Deich, *DMC1800 Device Driver*, `http://deimos.ucolick.org/spie2002/rotator/dmc1800.tar.gz`, 2002.
4. A. Conrad and W. Lupton, *KTL: the Keck Task Library*, `http://www.keck.hawaii.edu:8080/kroot/doc/ksd/08/08.ps`, 1991.
5. R. J. Stover, *MUSIC – a Multi-User System for Instrument Control*, UCO/Lick Technical Report 54, 1989.
6. S. A. Lewis, *Overview of the Experimental Physics and Industrial Control System: EPICS*, `http://csg.lbl.gov/EPICS/OverView.pdf`, 2000.
7. W. Lupton, *DCS Keyword Reference Manual*, `http://www.keck.hawaii.edu:8080/kroot/doc/ksd/46/html/46.html`, 1997.
8. W. Lupton, *Updated DCS Keyword Reference Manual (partial)*, `http://www.keck.hawaii.edu:8080/kroot/doc/ksd/46/html/46a.html`, 1997.
9. W. Deich, *DEIMOS rotator keywords*, `http://deimos.ucolick.org/spie2002/rotator/deirotkeys.ps`, 2002.
10. A. Honey, *Watch_keywords*, `http://alamoana.keck.hawaii.edu/realpublic/inst/nirc2/remsoft/preship/soft_pres/source_doc/watch_keywords.intro.html`.
11. J. Regehr, "Inferring Scheduling Behavior with Hourglass," in *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pp. 143–156, jun 2002.
12. W. Deich. The *polyterm* pseudoterminal source code is available at: `http://deimos.ucolick.org/spie2002/rotator/polyterm.tar.gz`, 2002.
13. W. Deich. The *pathrelay* source code is available at: `http://deimos.ucolick.org/spie2002/rotator/pathrelay`, 2002.